

VRC Team 9228A Code Documentation

1.6.2

Thu Jan 18 2018 23:00:45



Contents

1	InTheZoneA	1
2	Todo List	1
3	Namespace Index	1
3.1	Namespace List	1
4	Data Structure Index	1
4.1	Data Structures	1
5	File Index	2
5.1	File List	2
6	Namespace Documentation	4
6.1	testMath Namespace Reference	4
6.1.1	Function Documentation	4
7	Data Structure Documentation	5
7.1	_matrix Struct Reference	5
7.1.1	Detailed Description	5
7.1.2	Field Documentation	5
7.2	accelerometer_odometry Struct Reference	6
7.2.1	Detailed Description	7
7.2.2	Field Documentation	7
7.3	cord Struct Reference	7
7.3.1	Detailed Description	8
7.3.2	Field Documentation	8
7.4	encoder_odometry Struct Reference	8
7.4.1	Detailed Description	9
7.4.2	Field Documentation	9

7.5	lcd_buttons Struct Reference	10
7.5.1	Detailed Description	10
7.5.2	Field Documentation	10
7.6	list_iterator_t Struct Reference	11
7.6.1	Detailed Description	11
7.6.2	Field Documentation	11
7.7	list_node Struct Reference	12
7.7.1	Detailed Description	12
7.7.2	Field Documentation	13
7.8	list_t Struct Reference	13
7.8.1	Detailed Description	14
7.8.2	Field Documentation	14
7.9	location Struct Reference	15
7.9.1	Detailed Description	15
7.9.2	Field Documentation	16
7.10	menu_t Struct Reference	16
7.10.1	Detailed Description	17
7.10.2	Field Documentation	18
7.11	polar_cord Struct Reference	23
7.11.1	Detailed Description	23
7.11.2	Field Documentation	24
7.12	routine_t Struct Reference	24
7.12.1	Detailed Description	25
7.12.2	Field Documentation	25

8 File Documentation	26
8.1 include/auto.h File Reference	26
8.1.1 Detailed Description	26
8.2 auto.h	26
8.3 include/battery.h File Reference	27
8.3.1 Detailed Description	27
8.3.2 Function Documentation	27
8.4 battery.h	29
8.5 include/claw.h File Reference	29
8.5.1 Detailed Description	30
8.5.2 Enumeration Type Documentation	30
8.5.3 Function Documentation	31
8.6 claw.h	35
8.7 include/controller.h File Reference	35
8.7.1 Detailed Description	36
8.7.2 Enumeration Type Documentation	36
8.7.3 Function Documentation	38
8.8 controller.h	39
8.9 include/drive.h File Reference	40
8.9.1 Detailed Description	41
8.9.2 Typedef Documentation	41
8.9.3 Enumeration Type Documentation	41
8.9.4 Function Documentation	42
8.10 drive.h	45
8.11 include/encoders.h File Reference	45
8.11.1 Detailed Description	45
8.11.2 Function Documentation	45
8.12 encoders.h	48

8.13 include/gyro.h File Reference	48
8.13.1 Function Documentation	48
8.14 gyro.h	50
8.15 include/lcd.h File Reference	50
8.15.1 Detailed Description	51
8.15.2 Enumeration Type Documentation	51
8.15.3 Function Documentation	52
8.16 lcd.h	59
8.17 include/lifter.h File Reference	60
8.17.1 Detailed Description	60
8.17.2 Function Documentation	61
8.18 lifter.h	73
8.19 include/list.h File Reference	74
8.19.1 Detailed Description	75
8.19.2 Typedef Documentation	75
8.19.3 Enumeration Type Documentation	76
8.19.4 Function Documentation	77
8.20 list.h	91
8.21 include/localization.h File Reference	92
8.21.1 Detailed Description	92
8.21.2 Function Documentation	93
8.22 localization.h	97
8.23 include/log.h File Reference	97
8.23.1 Detailed Description	98
8.23.2 Function Documentation	98
8.24 log.h	105
8.25 include/main.h File Reference	105
8.25.1 Detailed Description	106

8.25.2 Function Documentation	107
8.26 main.h	112
8.27 include/matrix.h File Reference	113
8.27.1 Detailed Description	114
8.27.2 Typedef Documentation	114
8.27.3 Function Documentation	114
8.28 matrix.h	130
8.29 include/menu.h File Reference	130
8.29.1 Detailed Description	131
8.29.2 Typedef Documentation	131
8.29.3 Enumeration Type Documentation	132
8.29.4 Function Documentation	133
8.30 menu.h	139
8.31 include/mobile_goal_intake.h File Reference	140
8.31.1 Function Documentation	140
8.32 mobile_goal_intake.h	144
8.33 include/motor_ports.h File Reference	144
8.33.1 Detailed Description	144
8.34 motor_ports.h	145
8.35 include/potentiometer.h File Reference	145
8.36 potentiometer.h	145
8.37 include/routines.h File Reference	145
8.37.1 Typedef Documentation	146
8.37.2 Function Documentation	146
8.38 routines.h	151
8.39 include/sensors.h File Reference	151
8.39.1 Variable Documentation	151
8.40 sensors.h	152

8.41 include/slew.h File Reference	152
8.41.1 Detailed Description	152
8.41.2 Function Documentation	152
8.42 slew.h	159
8.43 include/toggle.h File Reference	159
8.43.1 Function Documentation	160
8.44 toggle.h	164
8.45 include/vlib.h File Reference	164
8.45.1 Detailed Description	164
8.45.2 Function Documentation	164
8.46 vlib.h	168
8.47 include/vmath.h File Reference	168
8.47.1 Detailed Description	169
8.47.2 Function Documentation	169
8.48 vmath.h	175
8.49 README.md File Reference	175
8.50 README.md	175
8.51 src/auto.c File Reference	175
8.51.1 Detailed Description	176
8.51.2 Function Documentation	176
8.52 auto.c	185
8.53 src/battery.c File Reference	187
8.53.1 Function Documentation	187
8.54 battery.c	190
8.55 src/claw.c File Reference	190
8.55.1 Function Documentation	190
8.55.2 Variable Documentation	194
8.56 claw.c	195

8.57 src/controller.c File Reference	195
8.57.1 Function Documentation	195
8.58 controller.c	196
8.59 src/drive.c File Reference	197
8.59.1 Function Documentation	197
8.59.2 Variable Documentation	201
8.60 drive.c	201
8.61 src/encoders.c File Reference	202
8.61.1 Function Documentation	202
8.62 encoders.c	205
8.63 src/gyro.c File Reference	205
8.63.1 Function Documentation	205
8.63.2 Variable Documentation	207
8.64 gyro.c	207
8.65 src/init.c File Reference	207
8.65.1 Detailed Description	207
8.65.2 Function Documentation	208
8.65.3 Variable Documentation	209
8.66 init.c	209
8.67 src/lcd.c File Reference	210
8.67.1 Function Documentation	210
8.67.2 Variable Documentation	219
8.68 lcd.c	220
8.69 src/lifter.c File Reference	220
8.69.1 Function Documentation	221
8.69.2 Variable Documentation	237
8.70 lifter.c	238
8.71 src/list.c File Reference	241

8.71.1 Function Documentation	241
8.72 list.c	252
8.73 src/list_iterator.c File Reference	254
8.73.1 Function Documentation	254
8.74 list_iterator.c	259
8.75 src/list_node.c File Reference	259
8.75.1 Function Documentation	259
8.76 list_node.c	260
8.77 src/localization.c File Reference	260
8.77.1 Function Documentation	261
8.77.2 Variable Documentation	268
8.78 localization.c	269
8.79 src/log.c File Reference	270
8.79.1 Function Documentation	271
8.79.2 Variable Documentation	279
8.80 log.c	280
8.81 src/matrix.c File Reference	280
8.81.1 Function Documentation	281
8.82 matrix.c	297
8.83 src/menu.c File Reference	301
8.83.1 Function Documentation	301
8.84 menu.c	310
8.85 src/mobile_goal_intake.c File Reference	312
8.85.1 Function Documentation	312
8.86 mobile_goal_intake.c	316
8.87 src/opcontrol.c File Reference	316
8.87.1 Detailed Description	317
8.87.2 Function Documentation	317

8.88 opcontrol.c	318
8.89 src/routines.c File Reference	319
8.89.1 Function Documentation	319
8.89.2 Variable Documentation	324
8.90 routines.c	324
8.91 src/slew.c File Reference	325
8.91.1 Function Documentation	325
8.91.2 Variable Documentation	332
8.92 slew.c	333
8.93 src/toggle.c File Reference	334
8.93.1 Function Documentation	334
8.93.2 Variable Documentation	338
8.94 toggle.c	338
8.95 src/vlib.c File Reference	340
8.95.1 Function Documentation	340
8.96 vlib.c	344
8.97 src/vmath.c File Reference	345
8.97.1 Function Documentation	345
8.98 vmath.c	350
8.99 test_code/testMath.py File Reference	350
8.100testMath.py	351
8.101testMath.py File Reference	351
8.102testMath.py	351

1 InTheZoneA

Team A code for In The Zone

2 Todo List

Global get_main_gyro_angluar_velocity (p. 205) ()

Global register_routine (p. 148) (void(*routine)(void *), button_t on_buttons, button_t *prohibited_buttons)

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

_matrix A struct representing a matrix	5
accelerometer_odometry Structure for holding an xy position from the accelerometer	6
cord A struct that contains cartesian coordinates	7
encoder_odemtry Structure for holding an xy position and an angle theta from the IMEs	8
lcd_buttons State of the lcd buttons	10
list_iterator_t A iterator representation Allows automatic iteration through linked list	11
list_node A node in a list	12
list_t A struct representing a linked list	13
location Vector storing the cartesian cords and an angle	15
menu_t Represents a specific instance of a menu	16
polar_cord A struct that contains polar coordinates	23
routine_t Routine system that allows mapping buttons to actions	24

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

include/ API.h	Provides the high-level user functionality intended for use by typical VEX Cortex programmers	??
include/ auto.h	Autonomous declarations and macros	26
include/ battery.h	Battery management related functions	27
include/ claw.h	Code for controlling the claw that grabs the cones	29
include/ controller.h	Controller definitions, macros and functions to assist with usig the vex controllers	35
include/ drive.h	Drive base definitions and enumerations	40
include/ encoders.h	Wrapper around encoder functions	45
include/ gyro.h		48
include/ lcd.h	LCD wrapper functions and macros	50
include/ lifter.h	Declarations and macros for controlling and manipulating the lifter	60
include/ list.h	A doubly linked list implementation	74
include/ localization.h	Declarations and macros for determining the location of the robot. [WIP]	92
include/ log.h	Contains logging functions	97
include/ main.h	Header file for global functions	105
include/ matrix.h	Various Matrix operations	113
include/ menu.h	Contains menu functionality and abstraction	130
include/ mobile_goal_intake.h		140
	Macros for the different motors ports	144

include/ potentiometer.h	145
include/ routines.h	145
include/ sensors.h	151
include/ slew.h	
Contains the slew rate controller wrapper for the motors	152
include/ toggle.h	159
include/ vlib.h	
Contains misc helpful functions	164
include/ vmath.h	
Vex Specific Math Functions, includes: Cartesian to polar coordinates	168
src/ auto.c	
File for autonomous code	175
src/ battery.c	187
src/ claw.c	190
src/ controller.c	195
src/ drive.c	197
src/ encoders.c	202
src/ gyro.c	205
src/ init.c	
File for initialization code	207
src/ lcd.c	210
src/ lifter.c	220
src/ list.c	241
src/ list_iterator.c	254
src/ list_node.c	259
src/ localization.c	260
src/ log.c	270
src/ matrix.c	280
src/ menu.c	301
src/ mobile_goal_intake.c	312
src/ opcontrol.c	
File for operator control code	316

src/ routines.c	319
src/ slew.c	325
src/ toggle.c	334
src/ vlib.c	340
src/ vmath.c	345

5 Data Structure Documentation

5.1 `_matrix` Struct Reference

A struct representing a matrix.

```
#include <matrix.h>
```

Data Fields

- double * **data**
- int **height**
- int **width**

5.1.1 Detailed Description

A struct representing a matrix.

Definition at line **16** of file **matrix.h**.

5.1.2 Field Documentation

5.1.2.1 `data`

```
double* _matrix::data
```

Definition at line **19** of file **matrix.h**.

Referenced by **covarianceMatrix()**, **dotDiagonalMatrix()**, **dotProductMatrix()**, **freeMatrix()**, **identityMatrix()**, **makeMatrix()**, **meanMatrix()**, **multiplyMatrix()**, **printMatrix()**, **rowSwap()**, **scaleMatrix()**, **traceMatrix()**, and **transposeMatrix()**.

5.1.2.2 height

```
int _matrix::height
```

Definition at line **17** of file **matrix.h**.

Referenced by **covarianceMatrix()**, **dotDiagonalMatrix()**, **dotProductMatrix()**, **makeMatrix()**, **meanMatrix()**, **multiplyMatrix()**, **printMatrix()**, **rowSwap()**, **scaleMatrix()**, **traceMatrix()**, and **transposeMatrix()**.

5.1.2.3 width

```
int _matrix::width
```

Definition at line **18** of file **matrix.h**.

Referenced by **covarianceMatrix()**, **dotDiagonalMatrix()**, **dotProductMatrix()**, **makeMatrix()**, **meanMatrix()**, **multiplyMatrix()**, **printMatrix()**, **rowSwap()**, **scaleMatrix()**, **traceMatrix()**, and **transposeMatrix()**.

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

- include/**matrix.h**

5.2 accelerometer_odometry Struct Reference

Structure for holding an xy position from the accelerometer.

Data Fields

- double **x**
- double **y**

5.2.1 Detailed Description

Structure for holding an xy position from the accelerometer.

Definition at line **24** of file **localization.c**.

5.2.2 Field Documentation

5.2.2.1 x

```
double accelerometer_odometry::x
```

Definition at line **25** of file **localization.c**.

5.2.2.2 y

```
double accelerometer_odometry::y
```

Definition at line **26** of file **localization.c**.

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

- src/ **localization.c**

5.3 cord Struct Reference

A struct that contains cartesian coordinates.

```
#include <vmath.h>
```

Data Fields

- float **x**
the x coordinate
- float **y**
the y coordinate

5.3.1 Detailed Description

A struct that contains cartesian coordinates.

Date

9/9/2017

Author

Chris Jerrett

Definition at line **36** of file **vmath.h**.

5.3.2 Field Documentation

5.3.2.1 x

float cord::x

the x coordinate

Definition at line **38** of file **vmath.h**.

Referenced by **get_joystick_cord()**, and **update_drive_motors()**.

5.3.2.2 y

float cord::y

the y coordinate

Definition at line **40** of file **vmath.h**.

Referenced by **get_joystick_cord()**, and **update_drive_motors()**.

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

- include/ **vmath.h**

5.4 encoder_odemtry Struct Reference

Structure for holding an xy position and an angle theta from the IMEs.

Data Fields

- double **theta**
- double **x**
- double **y**

5.4.1 Detailed Description

Structure for holding an xy position and an angle theta from the IMEs.

Definition at line **15** of file **localization.c**.

5.4.2 Field Documentation

5.4.2.1 theta

```
double encoder_odemtry::theta
```

Definition at line **18** of file **localization.c**.

Referenced by **calculate_encoder_odometry()**, and **integrate_gyro_w()**.

5.4.2.2 x

```
double encoder_odemtry::x
```

Definition at line **16** of file **localization.c**.

5.4.2.3 y

```
double encoder_odemtry::y
```

Definition at line **17** of file **localization.c**.

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

- src/ **localization.c**

5.5 **lcd_buttons** Struct Reference

represents the state of the lcd buttons

```
#include <lcd.h>
```

Data Fields

- **button_state left**
- **button_state middle**
- **button_state right**

5.5.1 Detailed Description

represents the state of the lcd buttons

Author

Chris Jerrett

Date

9/9/2017

Definition at line **48** of file **Lcd.h**.

5.5.2 Field Documentation

5.5.2.1 left

```
button_state lcd_buttons::left
```

Definition at line **49** of file **Lcd.h**.

Referenced by **Lcd_get_pressed_buttons()**.

5.5.2.2 middle

```
button_state lcd_buttons::middle
```

Definition at line **50** of file **Lcd.h**.

Referenced by **Lcd_get_pressed_buttons()**.

5.5.2.3 right

```
button_state lcd_buttons::right
```

Definition at line **51** of file **Lcd.h**.

Referenced by **Lcd_get_pressed_buttons()**.

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

- include/**Lcd.h**

5.6 list_iterator_t Struct Reference

A iterator representation Allows automatic iteration through linked list.

```
#include <list.h>
```

Data Fields

- `list_direction_t direction`
- `list_node_t * next`

5.6.1 Detailed Description

A iterator representation Allows automatic iteration through linked list.

Author

Chris Jerrett

Date

1/3/18

Definition at line **86** of file **list.h**.

5.6.2 Field Documentation

5.6.2.1 direction

```
list_direction_t list_iterator_t::direction
```

Definition at line **88** of file **list.h**.

5.6.2.2 next

```
list_node_t* list_iterator_t::next
```

Definition at line **87** of file **list.h**.

Referenced by `list_iterator_new_from_node()`.

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

- include/ `list.h`

5.7 list_node Struct Reference

A node in a list.

```
#include <list.h>
```

Data Fields

- struct **list_node** * **next**
- struct **list_node** * **prev**
- void * **val**

5.7.1 Detailed Description

A node in a list.

Author

Chris Jerrett

Date

1/3/18

Definition at line **53** of file **list.h**.

5.7.2 Field Documentation

5.7.2.1 next

```
struct list_node* list_node::next
```

Definition at line **55** of file **list.h**.

Referenced by **list_destroy()**, **list_iterator_next()**, **list_lpop()**, **list_lpush()**, **list_node_new()**, **list_remove()**, **list_rpop()**, and **list_rpush()**.

5.7.2.2 prev

```
struct list_node* list_node::prev
```

Definition at line **54** of file **list.h**.

Referenced by **list_iterator_next()**, **list_lpop()**, **list_lpush()**, **list_node_new()**, **list_remove()**, **list_rpop()**, and **list_rpush()**.

5.7.2.3 val

```
void* list_node::val
```

Definition at line **56** of file **list.h**.

Referenced by **list_destroy()**, **list_find()**, **list_node_new()**, **list_remove()**, **register_routine()**, and **routine_task()**.

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

- include/**list.h**

5.8 **list_t** Struct Reference

A struct representing a linked list.

```
#include <list.h>
```

Data Fields

- void(* **free**)(void *val)
- **list_node_t** * **head**
- unsigned int **len**
- int(* **match**)(void *a, void *b)
- **list_node_t** * **tail**

5.8.1 Detailed Description

A struct representing a linked list.

Author

Chris Jerrett

Date

1/3/18

Definition at line **64** of file **list.h**.

5.8.2 Field Documentation

5.8.2.1 free

```
void(* list_t::free) (void *val)
```

Definition at line **73** of file **list.h**.

5.8.2.2 head

```
list_node_t* list_t::head
```

Definition at line **66** of file **list.h**.

Referenced by **list_iterator_new()**, and **list_new()**.

5.8.2.3 len

```
unsigned int list_t::len
```

Definition at line **70** of file **list.h**.

5.8.2.4 match

```
int(* list_t::match) (void *a, void *b)
```

Definition at line **76** of file **list.h**.

5.8.2.5 tail

```
list_node_t* list_t::tail
```

Definition at line **68** of file **list.h**.

Referenced by **list_iterator_new()**.

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

- include/ **list.h**

5.9 location Struct Reference

Vector storing the cartesian cords and an angle.

```
#include <localization.h>
```

Data Fields

- int **theta**
- int **x**
- int **y**

5.9.1 Detailed Description

Vector storing the cartesian cords and an angle.

Definition at line **24** of file **localization.h**.

5.9.2 Field Documentation

5.9.2.1 theta

```
int location::theta
```

Definition at line **27** of file **localization.h**.

5.9.2.2 x

```
int location::x
```

Definition at line **25** of file **localization.h**.

5.9.2.3 y

```
int location::y
```

Definition at line **26** of file **localization.h**.

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

- include/ **localization.h**

5.10 menu_t Struct Reference

Represents a specific instance of a menu.

```
#include <menu.h>
```

Data Fields

- int **current**
contains the current index of menu.
- unsigned int **length**
*contains the length of options char**.*
- int **max**
contains the maximum int value of menu.
- float **max_f**
contains the maximum float value of menu.
- int **min**
contains the minimum int value of menu.
- float **min_f**
contains the minimum float value of menu.
- char ** **options**
contains the array of string options.
- char * **prompt**
contains the prompt to display on the first line.
- int **step**
contains the step int value of menu.
- float **step_f**
contains the step float value of menu.
- enum **menu_type type**
contains the type of menu.

5.10.1 Detailed Description

Represents a specific instance of a menu.

Will cause a memory leak if not deinitialized via `denint_menu`.

Author

Chris Jerrett

Date

9/8/17

See also

- [menu.h \(p. 130\)](#)
- [menu_t \(p. 16\)](#)
- [create_menu \(p. 303\)](#)
- [init_menu](#)
- [display_menu \(p. 305\)](#)
- [menu_type \(p. 132\)](#)
- [denint_menu \(p. 304\)](#)

Definition at line **66** of file `menu.h`.

5.10.2 Field Documentation

5.10.2.1 current

```
int menu_t::current
```

contains the current index of menu.

Author

Chris Jerrett

Date

9/8/17

Definition at line **142** of file **menu.h**.

Referenced by **calculate_current_display()**, **create_menu()**, **display_menu()**, and **init_menu_int()**.

5.10.2.2 length

```
unsigned int menu_t::length
```

contains the length of options char**.

Author

Chris Jerrett

Date

9/8/17

Definition at line **86** of file **menu.h**.

Referenced by **calculate_current_display()**, and **init_menu_var()**.

5.10.2.3 max

int menu_t::max

contains the maximum int value of menu.

Defaults to minimum int value

Author

Chris Jerrett

Date

9/8/17

Definition at line **102** of file **menu.h**.

Referenced by **calculate_current_display()**, **create_menu()**, and **init_menu_int()**.

5.10.2.4 max_f

float menu_t::max_f

contains the maximum float value of menu.

Defaults to minimum int value

Author

Chris Jerrett

Date

9/8/17

Definition at line **127** of file **menu.h**.

Referenced by **calculate_current_display()**, **create_menu()**, and **init_menu_float()**.

5.10.2.5 min

```
int menu_t::min
```

contains the minimum int value of menu.

Defaults to minimum int value

Author

Chris Jerrett

Date

9/8/17

Definition at line **94** of file **menu.h**.

Referenced by **calculate_current_display()**, **create_menu()**, and **init_menu_int()**.

5.10.2.6 min_f

```
float menu_t::min_f
```

contains the minimum float value of menu.

Defaults to minimum int value

Author

Chris Jerrett

Date

9/8/17

Definition at line **119** of file **menu.h**.

Referenced by **calculate_current_display()**, **create_menu()**, and **init_menu_float()**.

5.10.2.7 options

```
char** menu_t::options
```

contains the array of string options.

Author

Chris Jerrett

Date

9/8/17

Definition at line **79** of file **menu.h**.

Referenced by **calculate_current_display()**, **denint_menu()**, and **init_menu_var()**.

5.10.2.8 prompt

```
char* menu_t::prompt
```

contains the prompt to display on the first line.

Step is how much the int menu will increase or decrease with each press. Defaults to one

Author

Chris Jerrett

Date

9/8/17

Definition at line **150** of file **menu.h**.

Referenced by **create_menu()**, **denint_menu()**, and **display_menu()**.

5.10.2.9 step

```
int menu_t::step
```

contains the step int value of menu.

Step is how much the int menu will increase or decrease with each press. Defaults to one

Author

Chris Jerrett

Date

9/8/17

Definition at line **111** of file **menu.h**.

Referenced by **calculate_current_display()**, **create_menu()**, and **init_menu_int()**.

5.10.2.10 step_f

```
float menu_t::step_f
```

contains the step float value of menu.

Step is how much the int menu will increase or decrease with each press. Defaults to 1.0f

Author

Chris Jerrett

Date

9/8/17

Definition at line **136** of file **menu.h**.

Referenced by **calculate_current_display()**, **create_menu()**, and **init_menu_float()**.

5.10.2.11 type

```
enum menu_type menu_t::type
```

contains the type of menu.

Author

Chris Jerrett

Date

9/8/17

Definition at line **72** of file **menu.h**.

Referenced by **calculate_current_display()**, and **create_menu()**.

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

- include/ **menu.h**

5.11 polar_cord Struct Reference

A struct that contains polar coordinates.

```
#include <vmath.h>
```

Data Fields

- float **angle**
the angle of the vector
- float **magnitue**
the magnitude of the vector

5.11.1 Detailed Description

A struct that contains polar coordinates.

Date

9/9/2017

Author

Chris Jerrett

Definition at line **24** of file **vmath.h**.

5.11.2 Field Documentation

5.11.2.1 angle

```
float polar_cord::angle
```

the angle of the vector

Definition at line **26** of file **vmath.h**.

Referenced by **cartesian_to_polar()**.

5.11.2.2 magnitue

```
float polar_cord::magnitue
```

the magnitude of the vector

Definition at line **28** of file **vmath.h**.

Referenced by **cartesian_to_polar()**.

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

- include/ **vmath.h**

5.12 routine_t Struct Reference

Routine system that allows mapping buttons to actions.

```
#include <routines.h>
```

Data Fields

- **button_t * blocked_buttons**
- **button_t on_button**
- **void(* routine)(void *)**

5.12.1 Detailed Description

Routine system that allows mapping buttons to actions.

Author

Chris Jerrett

Date

1/8/17 Struct representing a routine

Definition at line **11** of file **routines.h**.

5.12.2 Field Documentation

5.12.2.1 blocked_buttons

```
button_t* routine_t::blocked_buttons
```

Definition at line **15** of file **routines.h**.

Referenced by **register_routine()**.

5.12.2.2 on_button

```
button_t routine_t::on_button
```

Definition at line **13** of file **routines.h**.

Referenced by **register_routine()**, and **routine_task()**.

5.12.2.3 routine

```
void(* routine_t::routine) (void *)
```

Definition at line **18** of file **routines.h**.

Referenced by **register_routine()**, and **routine_task()**.

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

- include/**routines.h**

6 File Documentation

6.1 include/API.h File Reference

Provides the high-level user functionality intended for use by typical VEX Cortex programmers.

TypeDefs

- **typedef void * Encoder**
Reference type for an initialized encoder.
- **typedef void * Gyro**
Reference type for an initialized gyro.
- **typedef void(* InterruptHandler) (unsigned char pin)**
Type definition for interrupt handlers.
- **typedef void * Mutex**
Type by which mutexes are referenced.
- **typedef int PROS_FILE**
PROS_FILE is an integer referring to a stream for the standard I/O functions.
- **typedef void * Semaphore**
Type by which semaphores are referenced.
- **typedef void(* TaskCode) (void *)**
Type for defining task functions.
- **typedef void * TaskHandle**
Type by which tasks are referenced.
- **typedef void * Ultrasonic**
Reference type for an initialized ultrasonic sensor.

Functions

- **void __attribute__ ((format(printf, 3, 4))) lcdPrint(PROS_FILE *lcdPort**
- **int analogCalibrate (unsigned char channel)**
Calibrates the analog sensor on the specified channel.
- **int analogRead (unsigned char channel)**
Reads an analog input channel and returns the 12-bit value.
- **int analogReadCalibrated (unsigned char channel)**
Reads the calibrated value of an analog input channel.
- **int analogReadCalibratedHR (unsigned char channel)**
Reads the calibrated value of an analog input channel 1-8 with enhanced precision.
- **void delay (const unsigned long time)**
*Wiring-compatible alias of **taskDelay()** (p. ??).*
- **void delayMicroseconds (const unsigned long us)**
Wait for approximately the given number of microseconds.
- **bool digitalRead (unsigned char pin)**
Gets the digital value (1 or 0) of a pin configured as a digital input.
- **void digitalWrite (unsigned char pin, bool value)**

- **int encoderGet (Encoder enc)**
Sets the digital value (1 or 0) of a pin configured as a digital output.
- **Encoder encoderInit (unsigned char portTop, unsigned char portBottom, bool reverse)**
Initializes and enables a quadrature encoder on two digital ports.
- **void encoderReset (Encoder enc)**
Resets the encoder to zero.
- **void encoderShutdown (Encoder enc)**
Stops and disables the encoder.
- **void fclose (PROS_FILE *stream)**
Closes the specified file descriptor.
- **int fcount (PROS_FILE *stream)**
Returns the number of characters that can be read without blocking (the number of characters available) from the specified stream.
- **int fdelete (const char *file)**
Delete the specified file if it exists and is not currently open.
- **int feof (PROS_FILE *stream)**
Checks to see if the specified stream is at its end.
- **int fflush (PROS_FILE *stream)**
Flushes the data on the specified file channel open in Write mode.
- **int fgetc (PROS_FILE *stream)**
Reads and returns one character from the specified stream, blocking until complete.
- **char * fgets (char *str, int num, PROS_FILE *stream)**
Reads a string from the specified stream, storing the characters into the memory at str.
- **PROS_FILE * fopen (const char *file, const char *mode)**
Opens the given file in the specified mode.
- **void fprintf (const char *string, PROS_FILE *stream)**
Prints the simple string to the specified stream.
- **int fprintf (PROS_FILE *stream, const char *formatString,...)**
Prints the formatted string to the specified output stream.
- **int fputc (int value, PROS_FILE *stream)**
Writes one character to the specified stream.
- **int fputs (const char *string, PROS_FILE *stream)**
Behaves the same as the "fprintf" function, and appends a trailing newline ("\\n").
- **size_t fread (void *ptr, size_t size, size_t count, PROS_FILE *stream)**
Reads data from a stream into memory.
- **int fseek (PROS_FILE *stream, long int offset, int origin)**
Seeks within a file open in Read mode.
- **long int ftell (PROS_FILE *stream)**
Returns the current position of the stream.
- **size_t fwrite (const void *ptr, size_t size, size_t count, PROS_FILE *stream)**
Writes data from memory to a stream.
- **int getchar ()**
Reads and returns one character from "stdin", which is the PC debug terminal.
- **int gyroGet (Gyro gyro)**
Gets the current gyro angle in degrees, rounded to the nearest degree.
- **Gyro gyrolInit (unsigned char port, unsigned short multiplier)**

- Initializes and enables a gyro on an analog port.*
- void **gyroReset** (Gyro gyro)
Resets the gyro to zero.
 - void **gyroShutdown** (Gyro gyro)
Stops and disables the gyro.
 - bool **i2cRead** (uint8_t addr, uint8_t *data, uint16_t count)
i2cRead - Reads the specified number of data bytes from the specified 7-bit I2C address.
 - bool **i2cReadRegister** (uint8_t addr, uint8_t reg, uint8_t *value, uint16_t count)
i2cReadRegister - Reads the specified amount of data from the given register address on the specified 7-bit I2C address.
 - bool **i2cWrite** (uint8_t addr, uint8_t *data, uint16_t count)
i2cWrite - Writes the specified number of data bytes to the specified 7-bit I2C address.
 - bool **i2cWriteRegister** (uint8_t addr, uint8_t reg, uint16_t value)
i2cWriteRegister - Writes the specified data byte to a register address on the specified 7-bit I2C address.
 - bool **imeGet** (unsigned char address, int *value)
Gets the current 32-bit count of the specified IME.
 - bool **imeGetVelocity** (unsigned char address, int *value)
Gets the current rotational velocity of the specified IME.
 - unsigned int **imeInitializeAll** ()
Initializes all IMEs.
 - bool **imeReset** (unsigned char address)
Resets the specified IME's counters to zero.
 - void **imeShutdown** ()
Shuts down all IMEs on the chain; their addresses return to the default and the stored counts and velocities are lost.
 - void **ioClearInterrupt** (unsigned char pin)
Disables interrupts on the specified pin.
 - void **ioSetInterrupt** (unsigned char pin, unsigned char edges, **InterruptHandler** handler)
Sets up an interrupt to occur on the specified pin, and resets any counters or timers associated with the pin.
 - bool **isAutonomous** ()
Returns true if the robot is in autonomous mode, or false otherwise.
 - bool **isEnabled** ()
Returns true if the robot is enabled, or false otherwise.
 - bool **isJoystickConnected** (unsigned char joystick)
Returns true if a joystick is connected to the specified slot number (1 or 2), or false otherwise.
 - bool **isOnline** ()
Returns true if a VEX field controller or competition switch is connected, or false otherwise.
 - int **joystickGetAnalog** (unsigned char joystick, unsigned char axis)
Gets the value of a control axis on the VEX joystick.
 - bool **joystickGetDigital** (unsigned char joystick, unsigned char buttonGroup, unsigned char button)
Gets the value of a button on the VEX joystick.
 - void **LcdClear** (**PROS_FILE** *lcdPort)
Clears the LCD screen on the specified port.
 - void **LcdInit** (**PROS_FILE** *lcdPort)
Initializes the LCD port, but does not change the text or settings.
 - void **LcdPrint** (**PROS_FILE** *lcdPort, unsigned char line, const char *formatString,...)
Prints the formatted string to the attached LCD.
 - void unsigned char const char unsigned int **LcdReadButtons** (**PROS_FILE** *lcdPort)
Reads the user button status from the LCD display.

- void **LcdSetBacklight** (**PROS_FILE** *lcdPort, bool backlight)

Sets the specified LCD backlight to be on or off.
- void **LcdSetText** (**PROS_FILE** *lcdPort, unsigned char **line**, const char *buffer)

Prints the string buffer to the attached LCD.
- void **LcdShutdown** (**PROS_FILE** *lcdPort)

Shut down the specified LCD port.
- unsigned long **micros** ()

Returns the number of microseconds since Cortex power-up.
- unsigned long **millis** ()

Returns the number of milliseconds since Cortex power-up.
- int **motorGet** (unsigned char channel)

Gets the last set speed of the specified motor channel.
- void **motorSet** (unsigned char channel, int speed)

Sets the speed of the specified motor channel.
- void **motorStop** (unsigned char channel)

*Stops the motor on the specified channel, equivalent to calling **motorSet()** (p. ??) with an argument of zero.*
- void **motorStopAll** ()

*Stops all motors; significantly faster than looping through all motor ports and calling **motorSet(channel, 0)** on each one.*
- **Mutex mutexCreate** ()

Creates a mutex intended to allow only one task to use a resource at a time.
- void **mutexDelete** (**Mutex** mutex)

Deletes the specified mutex.
- bool **mutexGive** (**Mutex** mutex)

Relinquishes a mutex so that other tasks can use the resource it guards.
- bool **mutexTake** (**Mutex** mutex, const unsigned long blockTime)

Requests a mutex so that other tasks cannot simultaneously use the resource it guards.
- void **pinMode** (unsigned char pin, unsigned char mode)

Configures the pin as an input or output with a variety of settings.
- unsigned int **powerLevelBackup** ()

Returns the backup battery voltage in millivolts.
- unsigned int **powerLevelMain** ()

Returns the main battery voltage in millivolts.
- void **print** (const char *string)

Prints the simple string to the debug terminal without formatting.
- int **printf** (const char * **formatString**,...)

Prints the formatted string to the debug stream (the PC terminal).
- int **putchar** (int value)

Writes one character to "stdout", which is the PC debug terminal, and returns the input value.
- int **puts** (const char *string)

Behaves the same as the "print" function, and appends a trailing newline ("\n").
- **Semaphore semaphoreCreate** ()

Creates a semaphore intended for synchronizing tasks.
- void **semaphoreDelete** (**Semaphore** semaphore)

Deletes the specified semaphore.
- bool **semaphoreGive** (**Semaphore** semaphore)

Signals a semaphore.
- bool **semaphoreTake** (**Semaphore** semaphore, const unsigned long blockTime)

Waits for the semaphore to be signaled, up to the specified time.

- **void setTeamName (const char *name)**
Sets the team name displayed to the VEX field control and VEX Firmware Upgrade.
- **int sprintf (char *buffer, size_t limit, const char *formatString,...)**
Prints the formatted string to the string buffer with the specified length limit.
- **void speakerInit ()**
Initializes VEX speaker support.
- **void speakerPlayArray (const char **songs)**
Plays up to three RTTTL (Ring Tone Text Transfer Language) songs simultaneously over the VEX speaker.
- **void speakerPlayRtttl (const char *song)**
Plays an RTTTL (Ring Tone Text Transfer Language) song over the VEX speaker.
- **void speakerShutdown ()**
Powers down and disables the VEX speaker.
- **int sprintf (char *buffer, const char *formatString,...)**
Prints the formatted string to the string buffer.
- **void standaloneModeEnable ()**
Enables the Cortex to run the op control task in a standalone mode- no VEXnet connection required.
- **TaskHandle taskCreate (TaskCode taskCode, const unsigned int stackDepth, void *parameters, const unsigned int priority)**
Creates a new task and add it to the list of tasks that are ready to run.
- **void taskDelay (const unsigned long msToDelay)**
Delays the current task for a given number of milliseconds.
- **void taskDelayUntil (unsigned long *previousWakeTime, const unsigned long cycleTime)**
Delays the current task until a specified time.
- **void taskDelete (TaskHandle taskToDelete)**
Kills and removes the specified task from the kernel task list.
- **unsigned int taskGetCount ()**
Determines the number of tasks that are currently being managed.
- **unsigned int taskGetState (TaskHandle task)**
Retrieves the state of the specified task.
- **unsigned int taskPriorityGet (const TaskHandle task)**
Obtains the priority of the specified task.
- **void taskPrioritySet (TaskHandle task, const unsigned int newPriority)**
Sets the priority of the specified task.
- **void taskResume (TaskHandle taskToResume)**
Resumes the specified task.
- **TaskHandle taskRunLoop (void(*fn)(void), const unsigned long increment)**
Starts a task which will periodically call the specified function.
- **void taskSuspend (TaskHandle taskToSuspend)**
Suspends the specified task.
- **int ultrasonicGet (Ultrasonic ult)**
Gets the current ultrasonic sensor value in centimeters.
- **Ultrasonic ultrasonicInit (unsigned char portEcho, unsigned char portPing)**
Initializes an ultrasonic sensor on the specified digital ports.
- **void ultrasonicShutdown (Ultrasonic ult)**
Stops and disables the ultrasonic sensor.
- **void usartInit (PROS_FILE *usart, unsigned int baud, unsigned int flags)**

- **void usartShutdown (PROS_FILE *usart)**
Initializes the specified serial interface with the given connection parameters.
- **void wait (const unsigned long time)**
Disables the specified USART interface.
- **void waitUntil (unsigned long *previousWakeTime, const unsigned long time)**
Alias of taskDelay() (p. ??) intended to help EasyC users.
- **void watchdogInit ()**
Alias of taskDelayUntil() (p. ??) intended to help EasyC users.
- **void watchdogInit ()**
Enables IWDG watchdog timer which will reset the cortex if it locks up due to static shock or a misbehaving task preventing the timer to be reset.

Variables

- **void unsigned char const char * formatString**
- **void unsigned char line**

6.1.1 Detailed Description

Provides the high-level user functionality intended for use by typical VEX Cortex programmers.

This file should be included for you in the predefined stubs in each new VEX Cortex PROS project through the inclusion of "main.h". In any new C source file, it is advisable to include **main.h** (p. 105) instead of referencing **API.h** (p. ??) by name, to better handle any nomenclature changes to this file or its contents.

Copyright (c) 2011-2016, Purdue University ACM SIGBots. All rights reserved.

This Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL was not distributed with this file, You can obtain one at <http://mozilla.org/MPL/2.0/>.

PROS contains FreeRTOS (<http://www.freertos.org>) whose source code may be obtained from <http://sourceforge.net/projects/freertos/files/> or on request.

Definition in file **API.h**.

6.1.2 Typedef Documentation

6.1.2.1 Encoder

```
typedef void* Encoder
```

Reference type for an initialized encoder.

Encoder information is stored as an opaque pointer to a structure in memory; as this is a pointer type, it can be safely passed or stored by value.

Definition at line **605** of file **API.h**.

6.1.2.2 Gyro

```
typedef void* Gyro
```

Reference type for an initialized gyro.

Gyro information is stored as an opaque pointer to a structure in memory; as this is a pointer type, it can be safely passed or stored by value.

Definition at line **548** of file **API.h**.

6.1.2.3 InterruptHandler

```
typedef void(* InterruptHandler) (unsigned char pin)
```

Type definition for interrupt handlers.

Such functions must accept one argument indicating the pin which changed.

Definition at line **332** of file **API.h**.

6.1.2.4 Mutex

```
typedef void* Mutex
```

Type by which mutexes are referenced.

As this is a pointer type, it can be safely passed or stored by value.

Definition at line **1306** of file **API.h**.

6.1.2.5 PROS_FILE

```
typedef int PROS_FILE
```

PROS_FILE is an integer referring to a stream for the standard I/O functions.

PROS_FILE * is the standard library method of referring to a file pointer, even though there is actually nothing there.

Definition at line **750** of file **API.h**.

6.1.2.6 Semaphore

```
typedef void* Semaphore
```

Type by which semaphores are referenced.

As this is a pointer type, it can be safely passed or stored by value.

Definition at line **1312** of file **API.h**.

6.1.2.7 TaskCode

```
typedef void(* TaskCode) (void *)
```

Type for defining task functions.

Task functions must accept one parameter of type "void *"; they need not use it.

For example:

```
void MyTask(void *ignore) { while (1); }
```

Definition at line **1323** of file **API.h**.

6.1.2.8 TaskHandle

```
typedef void* TaskHandle
```

Type by which tasks are referenced.

As this is a pointer type, it can be safely passed or stored by value.

Definition at line **1300** of file **API.h**.

6.1.2.9 Ultrasonic

```
typedef void* Ultrasonic
```

Reference type for an initialized ultrasonic sensor.

Ultrasonic information is stored as an opaque pointer to a structure in memory; as this is a pointer type, it can be safely passed or stored by value.

Definition at line **658** of file **API.h**.

6.1.3 Function Documentation

6.1.3.1 __attribute__()

```
void __attribute__ (
    format( printf, 3, 4))
```

6.1.3.2 analogCalibrate()

```
int analogCalibrate (
    unsigned char channel )
```

Calibrates the analog sensor on the specified channel.

This method assumes that the true sensor value is not actively changing at this time and computes an average from approximately 500 samples, 1 ms apart, for a 0.5 s period of calibration. The average value thus calculated is returned and stored for later calls to the **analogReadCalibrated()** (p. ??) and **analogReadCalibratedHR()** (p. ??) functions. These functions will return the difference between this value and the current sensor value when called.

Do not use this function in **initializeIO()** (p. 110), or when the sensor value might be unstable (gyro rotation, accelerometer movement).

This function may not work properly if the VEX Cortex is tethered to a PC using the orange USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery provides power to sensors.

Parameters

<i>channel</i>	the channel to calibrate from 1-8
----------------	-----------------------------------

Returns

the average sensor value computed by this function

6.1.3.3 analogRead()

```
int analogRead (
    unsigned char channel )
```

Reads an analog input channel and returns the 12-bit value.

The value returned is undefined if the analog pin has been switched to a different mode. This function is Wiring-compatible with the exception of the larger output range. The meaning of the returned value varies depending on the sensor attached.

This function may not work properly if the VEX Cortex is tethered to a PC using the orange USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery provides power to sensors.

Parameters

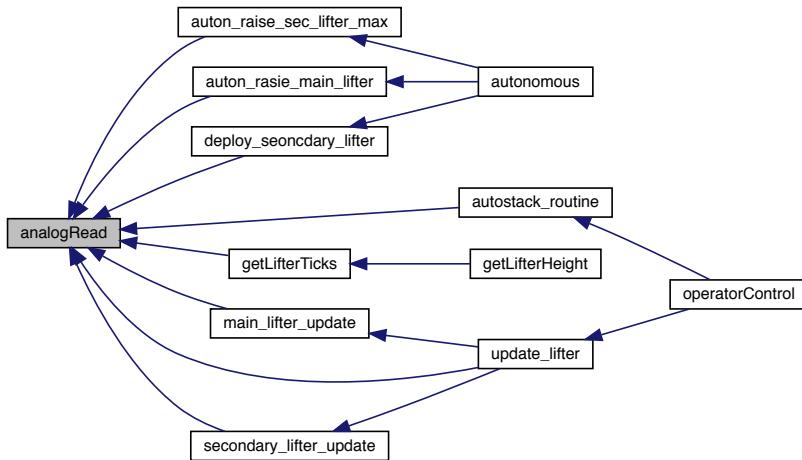
<code>channel</code>	the channel to read from 1-8
----------------------	------------------------------

Returns

the analog sensor value, where a value of 0 reflects an input voltage of nearly 0 V and a value of 4095 reflects an input voltage of nearly 5 V

Referenced by `auton_raise_sec_lifter_max()`, `auton_rasie_main_lifter()`, `autostack_routine()`, `deploy_secondary_lifter()`, `getLifterTicks()`, `main_lifter_update()`, `secondary_lifter_update()`, and `update_lifter()`.

Here is the caller graph for this function:



6.1.3.4 `analogReadCalibrated()`

```
int analogReadCalibrated (
    unsigned char channel )
```

Reads the calibrated value of an analog input channel.

The `analogCalibrate()` (p. ??) function must be run first on that channel. This function is inappropriate for sensor values intended for integration, as round-off error can accumulate causing drift over time. Use `analogReadCalibratedHR()` (p. ??) instead.

This function may not work properly if the VEX Cortex is tethered to a PC using the orange USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery provides power to sensors.

Parameters

<i>channel</i>	the channel to read from 1-8
----------------	------------------------------

Returns

the difference of the sensor value from its calibrated default from -4095 to 4095

6.1.3.5 analogReadCalibratedHR()

```
int analogReadCalibratedHR (
    unsigned char channel )
```

Reads the calibrated value of an analog input channel 1-8 with enhanced precision.

The **analogCalibrate()** (p. ??) function must be run first. This is intended for integrated sensor values such as gyros and accelerometers to reduce drift due to round-off, and should not be used on a sensor such as a line tracker or potentiometer.

The value returned actually has 16 bits of "precision", even though the ADC only reads 12 bits, so that errors induced by the average value being between two values come out in the wash when integrated over time. Think of the value as the true value times 16.

This function may not work properly if the VEX Cortex is tethered to a PC using the orange USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery provides power to sensors.

Parameters

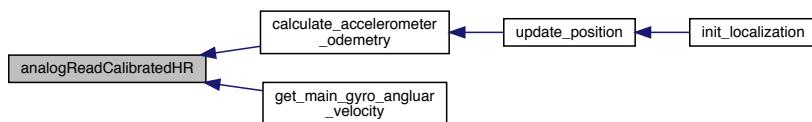
<i>channel</i>	the channel to read from 1-8
----------------	------------------------------

Returns

the difference of the sensor value from its calibrated default from -16384 to 16384

Referenced by **calculate_accelerometer_odometry()**, and **get_main_gyro_angluar_velocity()**.

Here is the caller graph for this function:



6.1.3.6 delay()

```
void delay (
    const unsigned long time )
```

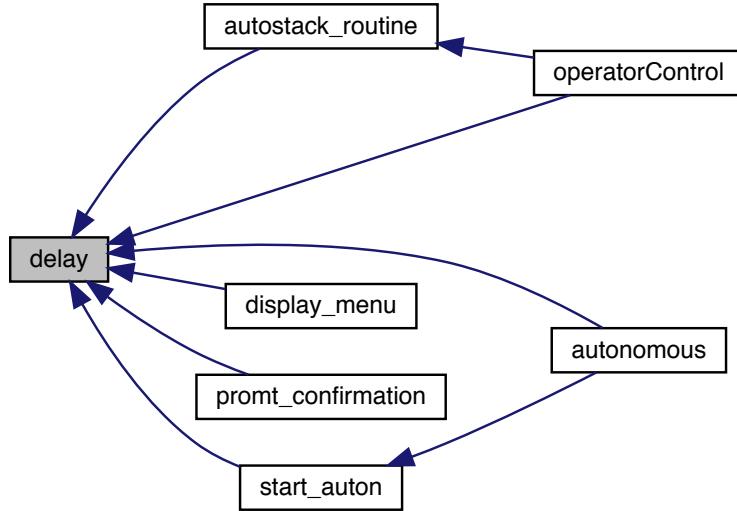
Wiring-compatible alias of **taskDelay()** (p. ??).

Parameters

<i>time</i>	the duration of the delay in milliseconds (1 000 milliseconds per second)
-------------	---

Referenced by **autonomous()**, **autostack_routine()**, **display_menu()**, **operatorControl()**, **prompt_confirmation()**, and **start_auton()**.

Here is the caller graph for this function:



6.1.3.7 delayMicroseconds()

```
void delayMicroseconds (
    const unsigned long us )
```

Wait for approximately the given number of microseconds.

The method used for delaying this length of time may vary depending on the argument. The current task will always be delayed by at least the specified period, but possibly much more depending on CPU load. In general, this function is less reliable than **delay()** (p. ??). Using this function in a loop may hog processing time from other tasks.

Parameters

<i>us</i>	the duration of the delay in microseconds (1 000 000 microseconds per second)
-----------	---

6.1.3.8 digitalRead()

```
bool digitalRead (
    unsigned char pin )
```

Gets the digital value (1 or 0) of a pin configured as a digital input.

If the pin is configured as some other mode, the digital value which reflects the current state of the pin is returned, which may or may not differ from the currently set value. The return value is undefined for pins configured as Analog inputs, or for ports in use by a Communications interface. This function is Wiring-compatible.

This function may not work properly if the VEX Cortex is tethered to a PC using the orange USB A to A cable and has no VEX 7.2V Battery connected and powered on, as the VEX Battery provides power to sensors.

Parameters

<i>pin</i>	the pin to read from 1-26
------------	---------------------------

Returns

true if the pin is HIGH, or false if it is LOW

6.1.3.9 digitalWrite()

```
void digitalWrite (
    unsigned char pin,
    bool value )
```

Sets the digital value (1 or 0) of a pin configured as a digital output.

If the pin is configured as some other mode, behavior is undefined. This function is Wiring-compatible.

Parameters

<i>pin</i>	the pin to write from 1-26
<i>value</i>	an expression evaluating to "true" or "false" to set the output to HIGH or LOW respectively, or the constants HIGH or LOW themselves

6.1.3.10 encoderGet()

```
int encoderGet (
    Encoder enc )
```

Gets the number of ticks recorded by the encoder.

There are 360 ticks in one revolution.

Parameters

<i>enc</i>	the Encoder object from encoderInit() (p. ??) to read
------------	--

Returns

the signed and cumulative number of counts since the last start or reset

6.1.3.11 encoderInit()

```
Encoder encoderInit (
    unsigned char portTop,
    unsigned char portBottom,
    bool reverse )
```

Initializes and enables a quadrature encoder on two digital ports.

Neither the top port nor the bottom port can be digital port 10. NULL will be returned if either port is invalid or the encoder is already in use. Initializing an encoder implicitly resets its count.

Parameters

<i>portTop</i>	the "top" wire from the encoder sensor with the removable cover side UP
<i>portBottom</i>	the "bottom" wire from the encoder sensor
<i>reverse</i>	if "true", the sensor will count in the opposite direction

Returns

an Encoder object to be stored and used for later calls to encoder functions

6.1.3.12 encoderReset()

```
void encoderReset (
    Encoder enc )
```

Resets the encoder to zero.

It is safe to use this method while an encoder is enabled. It is not necessary to call this method before stopping or starting an encoder.

Parameters

<code>enc</code>	the Encoder object from encoderInit() (p. ??) to reset
------------------	---

6.1.3.13 encoderShutdown()

```
void encoderShutdown (
    Encoder enc )
```

Stops and disables the encoder.

Encoders use processing power, so disabling unused encoders increases code performance. The encoder's count will be retained.

Parameters

<code>enc</code>	the Encoder object from encoderInit() (p. ??) to stop
------------------	--

6.1.3.14 fclose()

```
void fclose (
    PROS_FILE * stream )
```

Closes the specified file descriptor.

This function does not work on communication ports; use **uartShutdown()** (p. ??) instead.

Parameters

<code>stream</code>	the file descriptor to close from fopen() (p. ??)
---------------------	--

6.1.3.15 fcount()

```
int fcount (
    PROS_FILE * stream )
```

Returns the number of characters that can be read without blocking (the number of characters available) from the specified stream.

This only works for communication ports and files in Read mode; for files in Write mode, 0 is always returned.

This function may underestimate, but will not overestimate, the number of characters which meet this criterion.

Parameters

<i>stream</i>	the stream to read (stdin, uart1, uart2, or an open file in Read mode)
---------------	--

Returns

the number of characters which meet this criterion; if this number cannot be determined, returns 0

6.1.3.16 fdelete()

```
int fdelete (
    const char * file )
```

Delete the specified file if it exists and is not currently open.

The file will actually be erased from memory on the next re-boot. A physical power cycle is required to purge deleted files and free their allocated space for new files to be written. Deleted files are still considered inaccessible to **fopen()** (p. ??) in Read mode.

Parameters

<i>file</i>	the file name to erase
-------------	------------------------

Returns

0 if the file was deleted, or 1 if the file could not be found

6.1.3.17 feof()

```
int feof (
    PROS_FILE * stream )
```

Checks to see if the specified stream is at its end.

This only works for communication ports and files in Read mode; for files in Write mode, 1 is always returned.

Parameters

<i>stream</i>	the channel to check (stdin, uart1, uart2, or an open file in Read mode)
---------------	--

Returns

0 if the stream is not at EOF, or 1 otherwise.

6.1.3.18 fflush()

```
int fflush (
    PROS_FILE * stream )
```

Flushes the data on the specified file channel open in Write mode.

This function has no effect on a communication port or a file in Read mode, as these streams are always flushed as quickly as possible by the kernel.

Successful completion of an fflush function on a file in Write mode cannot guarantee that the file is valid until **fclose()** (p. ??) is used on that file descriptor.

Parameters

<i>stream</i>	the channel to flush (an open file in Write mode)
---------------	---

Returns

0 if the data was successfully flushed, EOF otherwise

6.1.3.19 fgetc()

```
int fgetc (
    PROS_FILE * stream )
```

Reads and returns one character from the specified stream, blocking until complete.

Do not use **fgetc()** (p. ??) on a VEX LCD port; deadlock may occur.

Parameters

<i>stream</i>	the stream to read (stdin, uart1, uart2, or an open file in Read mode)
---------------	--

Returns

the next character from 0 to 255, or -1 if no character can be read

6.1.3.20 fgets()

```
char* fgets (
    char * str,
    int num,
    PROS_FILE * stream )
```

Reads a string from the specified stream, storing the characters into the memory at str.

Characters will be read until the specified limit is reached, a new line is found, or the end of file is reached.

If the stream is already at end of file (for files in Read mode), NULL will be returned; otherwise, at least one character will be read and stored into str.

Parameters

<i>str</i>	the location where the characters read will be stored
<i>num</i>	the maximum number of characters to store; at most (num - 1) characters will be read, with a null terminator ('\0') automatically appended
<i>stream</i>	the channel to read (stdin, uart1, uart2, or an open file in Read mode)

Returns

str, or NULL if zero characters could be read

6.1.3.21 fopen()

```
PROS_FILE* fopen (
    const char * file,
    const char * mode )
```

Opens the given file in the specified mode.

The file name is truncated to eight characters. Only four files can be in use simultaneously in any given time, with at most one of those files in Write mode. This function does not work on communication ports; use **uartInit()** (p. ??) instead.

mode can be "r" or "w". Due to the nature of the VEX Cortex memory, the "r+", "w+", and "a" modes are not supported by the file system.

Opening a file that does not exist in Read mode will fail and return NULL, but opening a new file in Write mode will create it if there is space. Opening a file that already exists in Write mode will destroy the contents and create a new blank file if space is available.

There are important considerations when using of the file system on the VEX Cortex. Reading from files is safe, but writing to files should only be performed when robot actuators have been stopped. PROS will attempt to continue to handle events during file writes, but most user tasks cannot execute during file writing. Powering down the VEX Cortex mid-write may cause file system corruption.

Parameters

<i>file</i>	the file name
<i>mode</i>	the file mode

Returns

a file descriptor pointing to the new file, or NULL if the file could not be opened

6.1.3.22 fprintf()

```
void fprintf (
    const char * string,
    PROS_FILE * stream )
```

Prints the simple string to the specified stream.

This method is much, much faster than **fprintf()** (p. ??) and does not add a new line like **fputs()** (p. ??). Do not use **fprintf()** (p. ??) on a VEX LCD port. Use **LcdSetText()** (p. ??) instead.

Parameters

<i>string</i>	the string to write
<i>stream</i>	the stream to write (stdout, uart1, uart2, or an open file in Write mode)

6.1.3.23 fprintf()

```
int fprintf (
    PROS_FILE * stream,
    const char * formatString,
    ... )
```

Prints the formatted string to the specified output stream.

The specifiers supported by this minimalistic **printf()** (p. ??) function are:

- %d: Signed integer in base 10 (int)
- %u: Unsigned integer in base 10 (unsigned int)
- %x, %X: Integer in base 16 (unsigned int, int)
- %p: Pointer (void *, int *, ...)
- %c: Character (char)

- %s : Null-terminated string (char *)
- %%: Single literal percent sign
- %f : Floating-point number

Specifiers can be modified with:

- 0: Zero-pad, instead of space-pad
- a.b: Make the field at least "a" characters wide. If "b" is specified for "%f", changes the number of digits after the decimal point
- -: Left-align, instead of right-align
- +: Always display the sign character (displays a leading "+" for positive numbers)
- l: Ignored for compatibility

Invalid format specifiers, or mismatched parameters to specifiers, cause undefined behavior. Other characters are written out verbatim. Do not use **fprintf()** (p. ??) on a VEX LCD port. Use **LcdPrint()** (p. ??) instead.

Parameters

<i>stream</i>	the stream to write (stdout, uart1, or uart2)
<i>formatString</i>	the format string as specified above

Returns

the number of characters written

6.1.3.24 fputc()

```
int fputc (
    int value,
    PROS_FILE * stream )
```

Writes one character to the specified stream.

Do not use **fputc()** (p. ??) on a VEX LCD port. Use **LcdSetText()** (p. ??) instead.

Parameters

<i>value</i>	the character to write (a value of type "char" can be used)
<i>stream</i>	the stream to write (stdout, uart1, uart2, or an open file in Write mode)

Returns

the character written

6.1.3.25 fputs()

```
int fputs (
    const char * string,
    PROS_FILE * stream )
```

Behaves the same as the "fprint" function, and appends a trailing newline ("\n").

Do not use **fputs()** (p. ??) on a VEX LCD port. Use **LcdSetText()** (p. ??) instead.

Parameters

<i>string</i>	the string to write
<i>stream</i>	the stream to write (stdout, uart1, uart2, or an open file in Write mode)

Returns

the number of characters written, excluding the new line

6.1.3.26 fread()

```
size_t fread (
    void * ptr,
    size_t size,
    size_t count,
    PROS_FILE * stream )
```

Reads data from a stream into memory.

Returns the number of bytes thus read.

If the memory at *ptr* cannot store (*size* * *count*) bytes, undefined behavior occurs.

Parameters

<i>ptr</i>	a pointer to where the data will be stored
<i>size</i>	the size of each data element to read in bytes
<i>count</i>	the number of data elements to read
<i>stream</i>	the stream to read (stdout, uart1, uart2, or an open file in Read mode)

Returns

the number of bytes successfully read

6.1.3.27 fseek()

```
int fseek (
    PROS_FILE * stream,
    long int offset,
    int origin )
```

Seeks within a file open in Read mode.

This function will fail when used on a file in Write mode or on any communications port.

Parameters

<i>stream</i>	the stream to seek within
<i>offset</i>	the location within the stream to seek
<i>origin</i>	the reference location for offset: SEEK_CUR, SEEK_SET, or SEEK_END

Returns

0 if the seek was successful, or 1 otherwise

6.1.3.28 ftell()

```
long int ftell (
    PROS_FILE * stream )
```

Returns the current position of the stream.

This function works on files in either Read or Write mode, but will fail on communications ports.

Parameters

<i>stream</i>	the stream to check
---------------	---------------------

Returns

the offset of the stream, or -1 if the offset could not be determined

6.1.3.29 fwrite()

```
size_t fwrite (
    const void * ptr,
    size_t size,
    size_t count,
    PROS_FILE * stream )
```

Writes data from memory to a stream.

Returns the number of bytes thus written.

If the memory at *ptr* is not as long as (*size* * *count*) bytes, undefined behavior occurs.

Parameters

<i>ptr</i>	a pointer to the data to write
<i>size</i>	the size of each data element to write in bytes
<i>count</i>	the number of data elements to write
<i>stream</i>	the stream to write (stdout, uart1, uart2, or an open file in Write mode)

Returns

the number of bytes successfully written

6.1.3.30 getchar()

```
int getchar ( )
```

Reads and returns one character from "stdin", which is the PC debug terminal.

Returns

the next character from 0 to 255, or -1 if no character can be read

6.1.3.31 gyroGet()

```
int gyroGet (
    Gyro gyro )
```

Gets the current gyro angle in degrees, rounded to the nearest degree.

There are 360 degrees in a circle.

Parameters

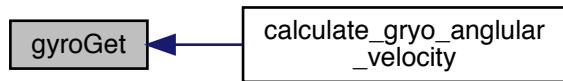
<code>gyro</code>	the Gyro object from <code>gyroInit()</code> (p. ??) to read
-------------------	--

Returns

the signed and cumulative number of degrees rotated around the gyro's vertical axis since the last start or reset

Referenced by `calculate_gryo_angular_velocity()`.

Here is the caller graph for this function:

**6.1.3.32 gyroInit()**

```
Gyro gyroInit (
    unsigned char port,
    unsigned short multiplier )
```

Initializes and enables a gyro on an analog port.

NULL will be returned if the port is invalid or the gyro is already in use. Initializing a gyro implicitly calibrates it and resets its count. Do not move the robot while the gyro is being calibrated. It is suggested to call this function in `initialize()` (p. 109) and to place the robot in its final position before powering it on.

The multiplier parameter can tune the gyro to adapt to specific sensors. The default value at this time is 196; higher values will increase the number of degrees reported for a fixed actual rotation, while lower values will decrease the number of degrees reported. If your robot is consistently turning too far, increase the multiplier, and if it is not turning far enough, decrease the multiplier.

Parameters

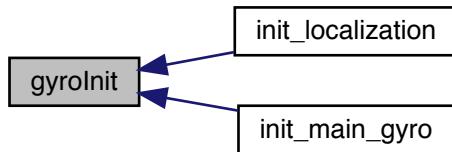
<code>port</code>	the analog port to use from 1-8
<code>multiplier</code>	an optional constant to tune the gyro readings; use 0 for the default value

Returns

a Gyro object to be stored and used for later calls to gyro functions

Referenced by `init_localization()`, and `init_main_gyro()`.

Here is the caller graph for this function:

**6.1.3.33 gyroReset()**

```
void gyroReset (
    Gyro gyro )
```

Resets the gyro to zero.

It is safe to use this method while a gyro is enabled. It is not necessary to call this method before stopping or starting a gyro.

Parameters

<code>gyro</code>	the Gyro object from <code>gyroInit()</code> (p. ??) to reset
-------------------	---

6.1.3.34 gyroShutdown()

```
void gyroShutdown (
    Gyro gyro )
```

Stops and disables the gyro.

Gyros use processing power, so disabling unused gyros increases code performance. The gyro's position will be retained.

Parameters

<i>gyro</i>	the Gyro object from gyroInit() (p. ??) to stop
-------------	--

6.1.3.35 i2cRead()

```
bool i2cRead (
    uint8_t addr,
    uint8_t * data,
    uint16_t count )
```

i2cRead - Reads the specified number of data bytes from the specified 7-bit I2C address.

The bytes will be stored at the specified location. Returns true if successful or false if failed. If only some bytes could be read, false is still returned.

The I2C address should be right-aligned; the R/W bit is automatically supplied.

Since most I2C devices use an 8-bit register architecture, this method has limited usefulness. Consider i2cReadRegister instead for the vast majority of applications.

6.1.3.36 i2cReadRegister()

```
bool i2cReadRegister (
    uint8_t addr,
    uint8_t reg,
    uint8_t * value,
    uint16_t count )
```

i2cReadRegister - Reads the specified amount of data from the given register address on the specified 7-bit I2C address.

Returns true if successful or false if failed. If only some bytes could be read, false is still returned.

The I2C address should be right-aligned; the R/W bit is automatically supplied.

Most I2C devices support an auto-increment address feature, so using this method to read more than one byte will usually read a block of sequential registers. Try to merge reads to separate registers into a larger read using this function whenever possible to improve code reliability, even if a few intermediate values need to be thrown away.

6.1.3.37 i2cWrite()

```
bool i2cWrite (
    uint8_t addr,
    uint8_t * data,
    uint16_t count )
```

i2cWrite - Writes the specified number of data bytes to the specified 7-bit I2C address.

Returns true if successful or false if failed. If only smoe bytes could be written, false is still returned.

The I2C address should be right-aligned; the R/W bit is automatically supplied.

Since most I2C devices use an 8-bit register architecture, this method is mostly useful for setting the register position (most devices remember the last-used address) or writing a sequence of bytes to one register address using an auto-increment feature. In these cases, the first byte written from the data buffer should have the register address to use.

6.1.3.38 i2cWriteRegister()

```
bool i2cWriteRegister (
    uint8_t addr,
    uint8_t reg,
    uint16_t value )
```

i2cWriteRegister - Writes the specified data byte to a register address on the specified 7-bit I2C address.

Returns true if successful or false if failed.

The I2C address should be right-aligned; the R/W bit is automatically supplied.

Only one byte can be written to each register address using this method. While useful for the vast majority of I2C operations, writing multiple bytes requires the **i2cWrite** method.

6.1.3.39 imeGet()

```
bool imeGet (
    unsigned char address,
    int * value )
```

Gets the current 32-bit count of the specified IME.

Much like the count for a quadrature encoder, the tick count is signed and cumulative. The value reflects total counts since the last reset. Different VEX Motor Encoders have a different number of counts per revolution:

- 240.448 for the 269 IME
- 627.2 for the 393 IME in high torque mode (factory default)
- 392 for the 393 IME in high speed mode

If the IME address is invalid, or the IME has not been reset or initialized, the value stored in **value* is undefined.

Parameters

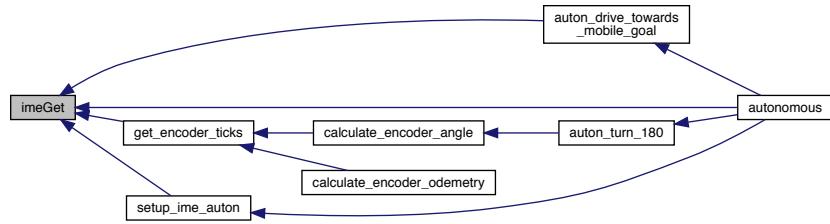
<i>address</i>	the IME address to fetch from 0 to IME_ADDR_MAX
<i>value</i>	a pointer to the location where the value will be stored (obtained using the "&" operator on the target variable name e.g. <code>imeGet(2, &counts)</code>)

Returns

true if the count was successfully read and the value stored in **value* is valid; false otherwise

Referenced by **auton_drive_towards_mobile_goal()**, **autonomous()**, **get_encoder_ticks()**, and **setup_ime_← auton()**.

Here is the caller graph for this function:



6.1.3.40 imeGetVelocity()

```
bool imeGetVelocity (
    unsigned char address,
    int * value )
```

Gets the current rotational velocity of the specified IME.

In this version of PROS, the velocity is positive if the IME count is increasing and negative if the IME count is decreasing. The velocity is in RPM of the internal encoder wheel. Since checking the IME for its type cannot reveal whether the motor gearing is high speed or high torque (in the 2-Wire Motor 393 case), the user must divide the return value by the number of output revolutions per encoder revolution:

- 30.056 for the 269 IME
- 39.2 for the 393 IME in high torque mode (factory default)
- 24.5 for the 393 IME in high speed mode

If the IME address is invalid, or the IME has not been reset or initialized, the value stored in *value is undefined.

Parameters

<i>address</i>	the IME address to fetch from 0 to IME_ADDR_MAX
<i>value</i>	a pointer to the location where the value will be stored (obtained using the "&" operator on the target variable name e.g. <code>imeGetVelocity(2, &counts)</code>)

Returns

true if the velocity was successfully read and the value stored in *value is valid; false otherwise

Referenced by `get_encoder_velocity()`.

Here is the caller graph for this function:



6.1.3.41 imelInitializeAll()

```
unsigned int imelInitializeAll ( )
```

Initializes all IMEs.

IMEs are assigned sequential incrementing addresses, beginning with the first IME on the chain (closest to the VEX Cortex I2C port). Therefore, a given configuration of IMEs will always have the same ID assigned to each encoder. The addresses range from 0 to IME_ADDR_MAX, so the first encoder gets 0, the second gets 1, ...

This function should most likely be used in **initialize()** (p. 109). Do not use it in **initializeO()** (p. 110) or at any other time when the scheduler is paused (like an interrupt). Checking the return value of this function is important to ensure that all IMEs are plugged in and responding as expected.

This function, unlike the other IME functions, is not thread safe. If using imelInitializeAll to re-initialize encoders, calls to other IME functions might behave unpredictably during this function's execution.

Returns

the number of IMEs successfully initialized.

Referenced by **init_encoders()**.

Here is the caller graph for this function:



6.1.3.42 imeReset()

```
bool imeReset (
    unsigned char address )
```

Resets the specified IME's counters to zero.

This method can be used while the IME is rotating.

Parameters

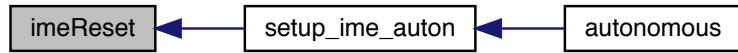
<i>address</i>	the IME address to reset from 0 to IME_ADDR_MAX
----------------	---

Returns

true if the reset succeeded; false otherwise

Referenced by `setup_ime_auton()`.

Here is the caller graph for this function:



6.1.3.43 `imeShutdown()`

```
void imeShutdown ( )
```

Shuts down all IMEs on the chain; their addresses return to the default and the stored counts and velocities are lost.

This function, unlike the other IME functions, is not thread safe.

To use the IME chain again, wait at least 0.25 seconds before using `imelInitializeAll` again.

6.1.3.44 `ioClearInterrupt()`

```
void ioClearInterrupt (
    unsigned char pin )
```

Disables interrupts on the specified pin.

Disabling interrupts on interrupt pins which are not in use conserves processing time.

Parameters

<i>pin</i>	the pin on which to reset interrupts from 1-9,11-12
------------	---

6.1.3.45 ioSetInterrupt()

```
void ioSetInterrupt (
    unsigned char pin,
    unsigned char edges,
    InterruptHandler handler )
```

Sets up an interrupt to occur on the specified pin, and resets any counters or timers associated with the pin.

Each time the specified change occurs, the function pointer passed in will be called with the pin that changed as an argument. Enabling pin-change interrupts consumes processing time, so it is best to only enable necessary interrupts and to keep the InterruptHandler function short. Pin change interrupts can only be enabled on pins 1-9 and 11-12.

Do not use API functions such as **delay()** (p. ??) inside the handler function, as the function will run in an ISR where the scheduler is paused and no other interrupts can execute. It is best to quickly update some state and allow a task to perform the work.

Do not use this function on pins that are also being used by the built-in ultrasonic or shaft encoder drivers, or on pins which have been switched to output mode.

Parameters

<i>pin</i>	the pin on which to enable interrupts from 1-9,11-12
<i>edges</i>	one of INTERRUPT_EDGE_RISING, INTERRUPT_EDGE_FALLING, or INTERRUPT_EDGE_BOTH
<i>handler</i>	the function to call when the condition is satisfied

6.1.3.46 isAutonomous()

```
bool isAutonomous ( )
```

Returns true if the robot is in autonomous mode, or false otherwise.

While in autonomous mode, joystick inputs will return a neutral value, but serial port communications (even over VexNET) will still work properly.

6.1.3.47 isEnabled()

```
bool isEnabled ( )
```

Returns true if the robot is enabled, or false otherwise.

While disabled via the VEX Competition Switch or VEX Field Controller, motors will not function. However, the digital I/O ports can still be changed, which may indirectly affect the robot state (e.g. solenoids). Avoid performing externally visible actions while disabled (the kernel should take care of this most of the time).

6.1.3.48 isJoystickConnected()

```
bool isJoystickConnected (
    unsigned char joystick )
```

Returns true if a joystick is connected to the specified slot number (1 or 2), or false otherwise.

Useful for automatically merging joysticks for one operator, or splitting for two. This function does not work properly during **initialize()** (p. 109) or **initializeIO()** (p. 110) and can return false positives. It should be checked once and stored at the beginning of **operatorControl()** (p. 110).

Parameters

<i>joystick</i>	the joystick slot to check
-----------------	----------------------------

6.1.3.49 isOnline()

```
bool isOnline ( )
```

Returns true if a VEX field controller or competition switch is connected, or false otherwise.

When in online mode, the switching between **autonomous()** (p. 107) and **operatorControl()** (p. 110) tasks is managed by the PROS kernel.

6.1.3.50 joystickGetAnalog()

```
int joystickGetAnalog (
    unsigned char joystick,
    unsigned char axis )
```

Gets the value of a control axis on the VEX joystick.

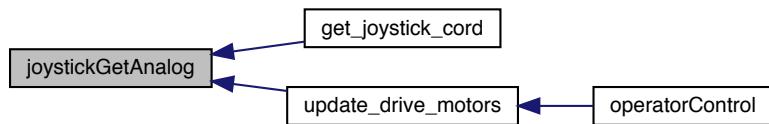
Returns the value from -127 to 127, or 0 if no joystick is connected to the requested slot.

Parameters

<i>joystick</i>	the joystick slot to check
<i>axis</i>	one of 1, 2, 3, 4, ACCEL_X, or ACCEL_Y

Referenced by **get_joystick_cord()**, and **update_drive_motors()**.

Here is the caller graph for this function:



6.1.3.51 joystickGetDigital()

```
bool joystickGetDigital (
    unsigned char joystick,
    unsigned char buttonGroup,
    unsigned char button )
```

Gets the value of a button on the VEX joystick.

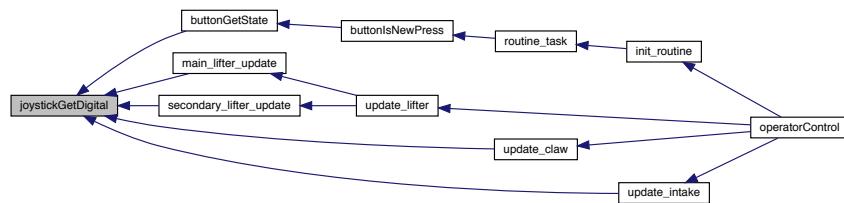
Returns true if that button is pressed, or false otherwise. If no joystick is connected to the requested slot, returns false.

Parameters

<i>joystick</i>	the joystick slot to check
<i>buttonGroup</i>	one of 5, 6, 7, or 8 to request that button as labelled on the joystick
<i>button</i>	one of JOY_UP, JOY_DOWN, JOY_LEFT, or JOY_RIGHT; requesting JOY_LEFT or JOY_RIGHT for groups 5 or 6 will cause an undefined value to be returned

Referenced by **buttonGetState()**, **main_lifter_update()**, **secondary_lifter_update()**, **update_claw()**, and **update_intake()**.

Here is the caller graph for this function:



6.1.3.52 lcdClear()

```
void lcdClear (
    PROS_FILE * lcdPort )
```

Clears the LCD screen on the specified port.

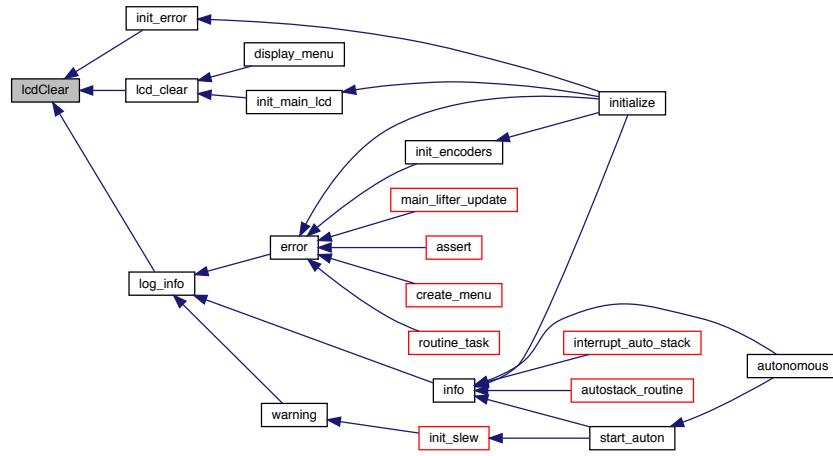
Printing to a line implicitly overwrites the contents, so clearing should only be required at startup.

Parameters

<i>lcdPort</i>	the LCD to clear, either uart1 or uart2
----------------	---

Referenced by `init_error()`, `lcd_clear()`, and `log_info()`.

Here is the caller graph for this function:



6.1.3.53 lcdInit()

```
void lcdInit (
    PROS_FILE * lcdPort )
```

Initializes the LCD port, but does not change the text or settings.

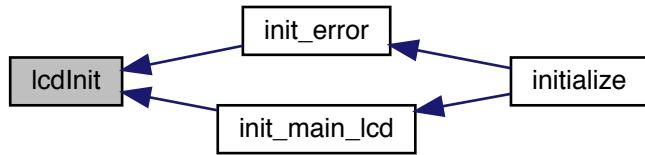
If the LCD was not initialized before, the text currently on the screen will be undefined. The port will not be usable with standard serial port functions until the LCD is stopped.

Parameters

<code>lcdPort</code>	the LCD to initialize, either uart1 or uart2
----------------------	--

Referenced by `init_error()`, and `init_main_lcd()`.

Here is the caller graph for this function:



6.1.3.54 `LcdPrint()`

```
void lcdPrint (
    PROS_FILE * lcdPort,
    unsigned char line,
    const char * formatString,
    ... )
```

Prints the formatted string to the attached LCD.

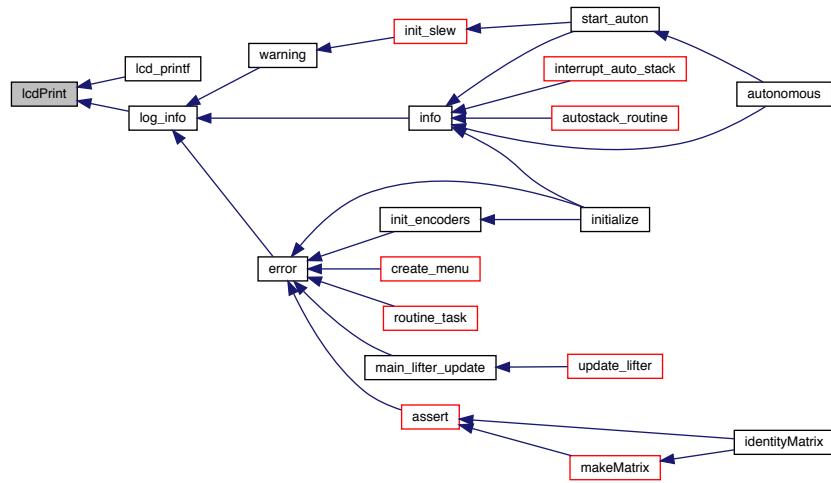
The output string will be truncated as necessary to fit on the LCD screen, 16 characters wide. It is probably better to generate the string in a local buffer and use `LcdSetText()` (p. ??) but this method is provided for convenience.

Parameters

<code>lcdPort</code>	the LCD to write, either uart1 or uart2
<code>line</code>	the LCD line to write, either 1 or 2
<code>formatString</code>	the format string as specified in <code>fprintf()</code> (p. ??)

Referenced by `lcd_printf()`, and `log_info()`.

Here is the caller graph for this function:



6.1.3.55 lcdReadButtons()

```
void unsigned char const char unsigned int lcdReadButtons (
    PROS_FILE * lcdPort )
```

Reads the user button status from the LCD display.

For example, if the left and right buttons are pushed, $(1 | 4) = 5$ will be returned. 0 is returned if no buttons are pushed.

Parameters

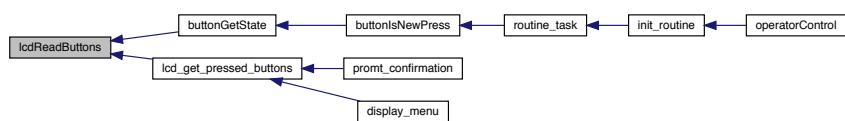
<i>lcdPort</i>	the LCD to poll, either uart1 or uart2
----------------	--

Returns

the buttons pressed as a bit mask

Referenced by `buttonGetState()`, and `lcd_get_pressed_buttons()`.

Here is the caller graph for this function:



6.1.3.56 lcdSetBacklight()

```
void lcdSetBacklight (
    PROS_FILE * lcdPort,
    bool backlight )
```

Sets the specified LCD backlight to be on or off.

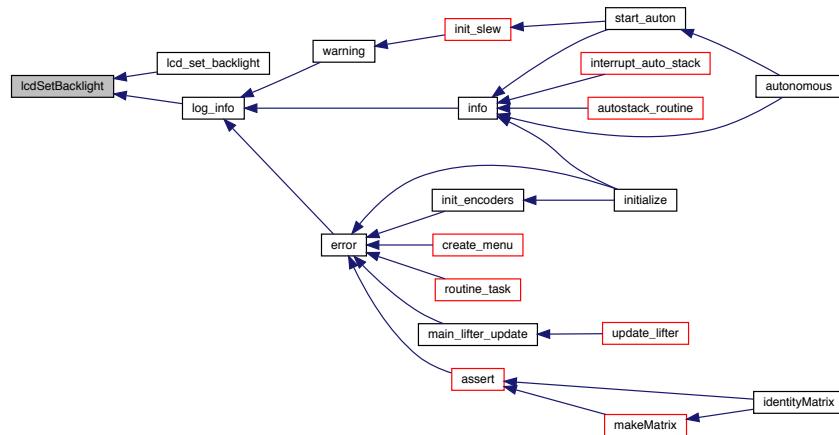
Turning it off will save power but may make it more difficult to read in dim conditions.

Parameters

<i>lcdPort</i>	the LCD to adjust, either uart1 or uart2
<i>backlight</i>	true to turn the backlight on, or false to turn it off

Referenced by `lcd_set_backlight()`, and `log_info()`.

Here is the caller graph for this function:



6.1.3.57 lcdSetText()

```
void lcdSetText (
    PROS_FILE * lcdPort,
    unsigned char line,
    const char * buffer )
```

Prints the string buffer to the attached LCD.

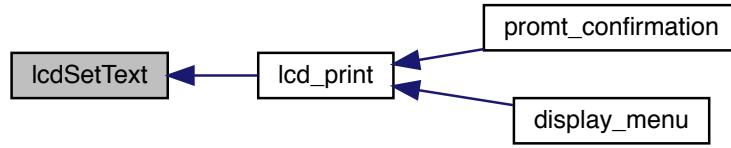
The output string will be truncated as necessary to fit on the LCD screen, 16 characters wide. This function, like `fprint()` (p. ??), is much, much faster than a formatted routine such as `lcdPrint()` (p. ??) and consumes less memory.

Parameters

<i>lcdPort</i>	the LCD to write, either uart1 or uart2
<i>line</i>	the LCD line to write, either 1 or 2
<i>buffer</i>	the string to write

Referenced by **lcd_print()**.

Here is the caller graph for this function:

**6.1.3.58 lcdShutdown()**

```
void lcdShutdown (
    PROS_FILE * lcdPort )
```

Shut down the specified LCD port.

Parameters

<i>lcdPort</i>	the LCD to stop, either uart1 or uart2
----------------	--

6.1.3.59 micros()

```
unsigned long micros ( )
```

Returns the number of microseconds since Cortex power-up.

There are 10^6 microseconds in a second, so as a 32-bit integer, this will overflow and wrap back to zero every two hours or so.

This function is Wiring-compatible.

Returns

the number of microseconds since the Cortex was turned on or the last overflow

6.1.3.60 millis()

```
unsigned long millis ( )
```

Returns the number of milliseconds since Cortex power-up.

There are 1000 milliseconds in a second, so as a 32-bit integer, this will not overflow for 50 days.

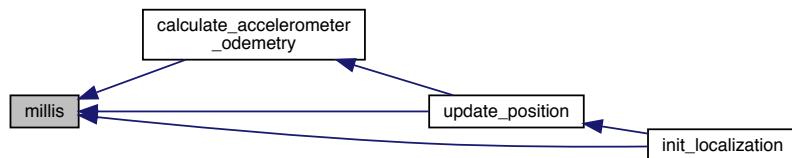
This function is Wiring-compatible.

Returns

the number of milliseconds since the Cortex was turned on

Referenced by **calculate_accelerometer_odometry()**, **init_localization()**, and **update_position()**.

Here is the caller graph for this function:

**6.1.3.61 motorGet()**

```
int motorGet (
    unsigned char channel )
```

Gets the last set speed of the specified motor channel.

This speed may have been set by any task or the PROS kernel itself. This is not guaranteed to be the speed that the motor is actually running at, or even the speed currently being sent to the motor, due to latency in the Motor Controller 29 protocol and physical loading. To measure actual motor shaft revolution speed, attach a VEX Integrated Motor Encoder or VEX Quadrature Encoder and use the velocity functions associated with each.

Parameters

<i>channel</i>	the motor channel to fetch from 1-10
----------------	--------------------------------------

Returns

the speed last sent to this channel; -127 is full reverse and 127 is full forward, with 0 being off

6.1.3.62 motorSet()

```
void motorSet (
    unsigned char channel,
    int speed )
```

Sets the speed of the specified motor channel.

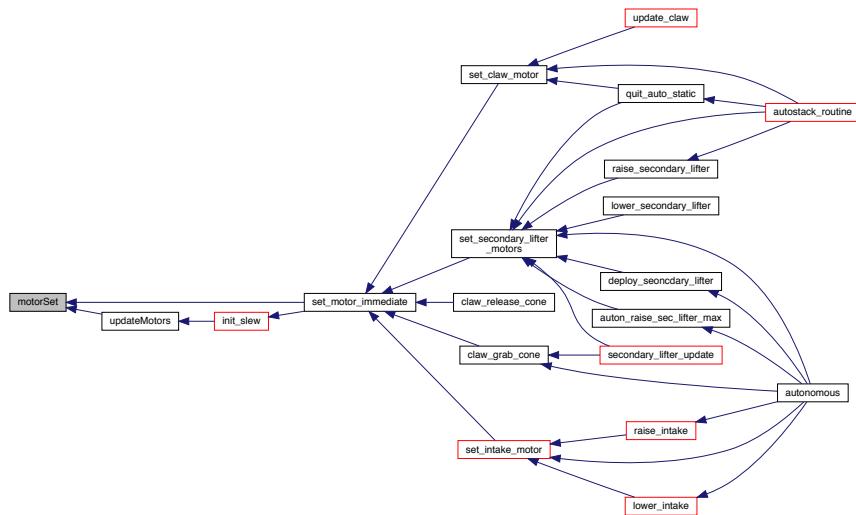
Do not use **motorSet()** (p. ??) with the same channel argument from two different tasks. It is safe to use **motorSet()** (p. ??) with different channel arguments from different tasks.

Parameters

<i>channel</i>	the motor channel to modify from 1-10
<i>speed</i>	the new signed speed; -127 is full reverse and 127 is full forward, with 0 being off

Referenced by **set_motor_immediate()**, and **updateMotors()**.

Here is the caller graph for this function:



6.1.3.63 motorStop()

```
void motorStop (
    unsigned char channel )
```

Stops the motor on the specified channel, equivalent to calling **motorSet()** (p. ??) with an argument of zero.

This performs a coasting stop, not an active brake. Since motorStop is similar to motorSet(0), see the note for **motorSet()** (p. ??) about use from multiple tasks.

Parameters

<i>channel</i>	the motor channel to stop from 1-10
----------------	-------------------------------------

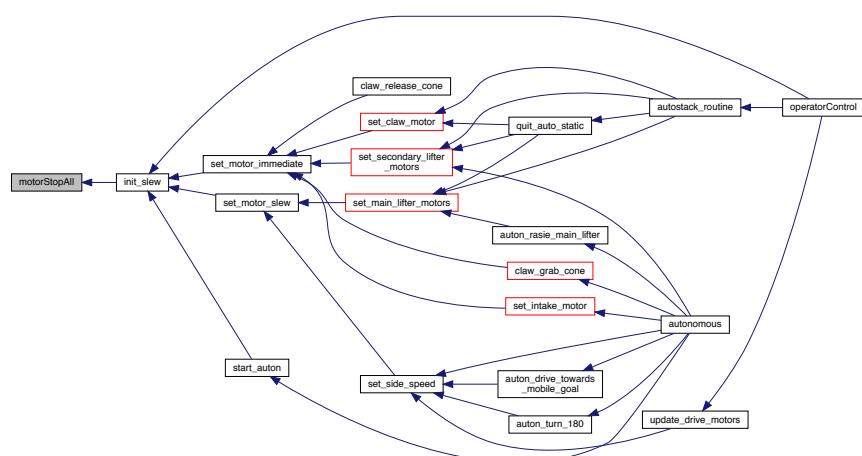
6.1.3.64 motorStopAll()

```
void motorStopAll ( )
```

Stops all motors; significantly faster than looping through all motor ports and calling motorSet(channel, 0) on each one.

Referenced by **init_slew()**.

Here is the caller graph for this function:



6.1.3.65 mutexCreate()

```
Mutex mutexCreate ( )
```

Creates a mutex intended to allow only one task to use a resource at a time.

For signalling and synchronization, try using semaphores.

Mutexes created using this function can be accessed using the **mutexTake()** (p. ??) and **mutexGive()** (p. ??) functions. The semaphore functions must not be used on objects of this type.

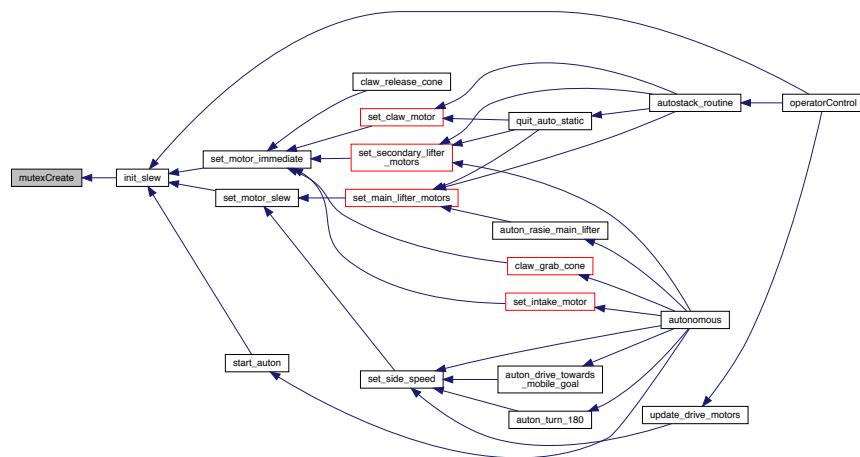
This type of object uses a priority inheritance mechanism so a task 'taking' a mutex MUST ALWAYS 'give' the mutex back once the mutex is no longer required.

Returns

a handle to the created mutex

Referenced by **init_slew()**.

Here is the caller graph for this function:



6.1.3.66 mutexDelete()

```
void mutexDelete (
    Mutex mutex )
```

Deletes the specified mutex.

This function can be dangerous; deleting semaphores being waited on by a task may cause deadlock or a crash.

Parameters

<code>mutex</code>	the mutex to destroy
--------------------	----------------------

6.1.3.67 mutexGive()

```
bool mutexGive (
    Mutex mutex )
```

Relinquishes a mutex so that other tasks can use the resource it guards.

The mutex must be held by the current task using a corresponding call to mutexTake.

Parameters

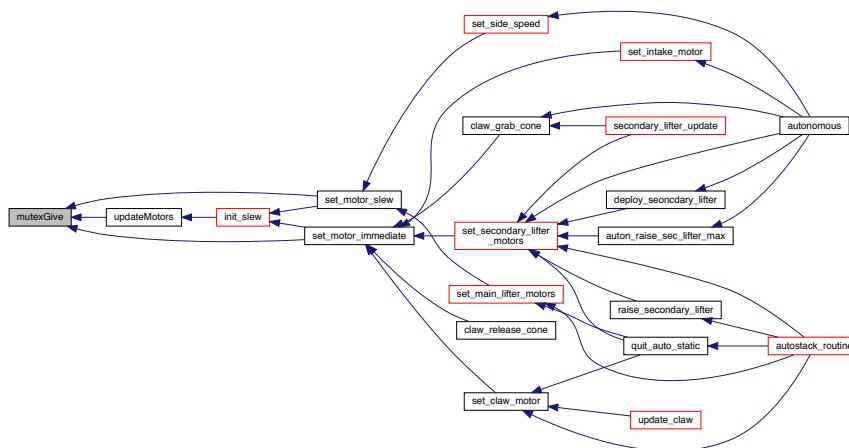
<code>mutex</code>	the mutex to release
--------------------	----------------------

Returns

true if the mutex was released, or false if the mutex was not already held

Referenced by `set_motor_immediate()`, `set_motor_slew()`, and `updateMotors()`.

Here is the caller graph for this function:



6.1.3.68 mutexTake()

```
bool mutexTake (
    Mutex mutex,
    const unsigned long blockTime )
```

Requests a mutex so that other tasks cannot simultaneously use the resource it guards.

The mutex must not already be held by the current task. If another task already holds the mutex, the function will wait for the mutex to be released. Other tasks can run during this time.

Parameters

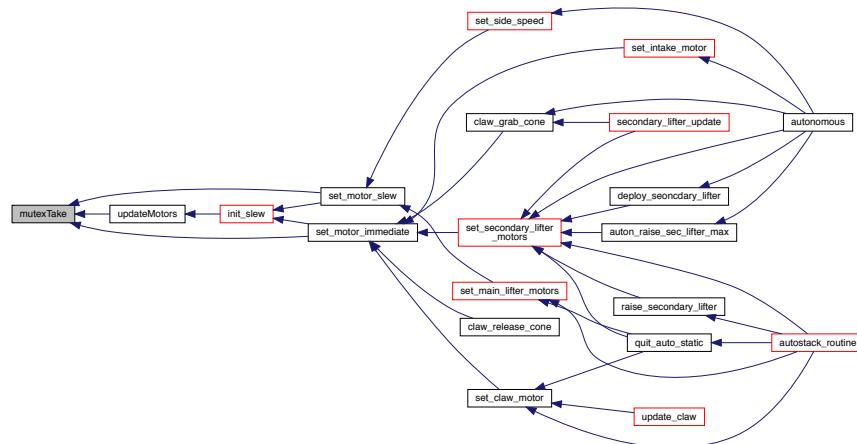
<i>mutex</i>	the mutex to request
<i>blockTime</i>	the maximum time to wait for the mutex to be available, where -1 specifies an infinite timeout

Returns

true if the mutex was successfully taken, or false if the timeout expired

Referenced by `set_motor_immediate()`, `set_motor_slew()`, and `updateMotors()`.

Here is the caller graph for this function:



6.1.3.69 pinMode()

```
void pinMode (
    unsigned char pin,
    unsigned char mode )
```

Configures the pin as an input or output with a variety of settings.

Do note that INPUT by default turns on the pull-up resistor, as most VEX sensors are open-drain active low. It should not be a big deal for most push-pull sources. This function is Wiring-compatible.

Parameters

<i>pin</i>	the pin to modify from 1-26
<i>mode</i>	one of INPUT, INPUT_ANALOG, INPUT_FLOATING, OUTPUT, or OUTPUT_OD

6.1.3.70 powerLevelBackup()

```
unsigned int powerLevelBackup ( )
```

Returns the backup battery voltage in millivolts.

If no backup battery is connected, returns 0.

Referenced by **backup_battery_voltage()**.

Here is the caller graph for this function:



6.1.3.71 powerLevelMain()

```
unsigned int powerLevelMain ( )
```

Returns the main battery voltage in millivolts.

In rare circumstances, this method might return 0. Check the output value for reasonability before blindly blasting the user.

Referenced by **main_battery_voltage()**.

Here is the caller graph for this function:



6.1.3.72 print()

```
void print (
    const char * string )
```

Prints the simple string to the debug terminal without formatting.

This method is much, much faster than **printf()** (p. ??).

Parameters

<i>string</i>	the string to write
---------------	---------------------

6.1.3.73 printf()

```
int printf (
    const char * formatString,
    ... )
```

Prints the formatted string to the debug stream (the PC terminal).

Parameters

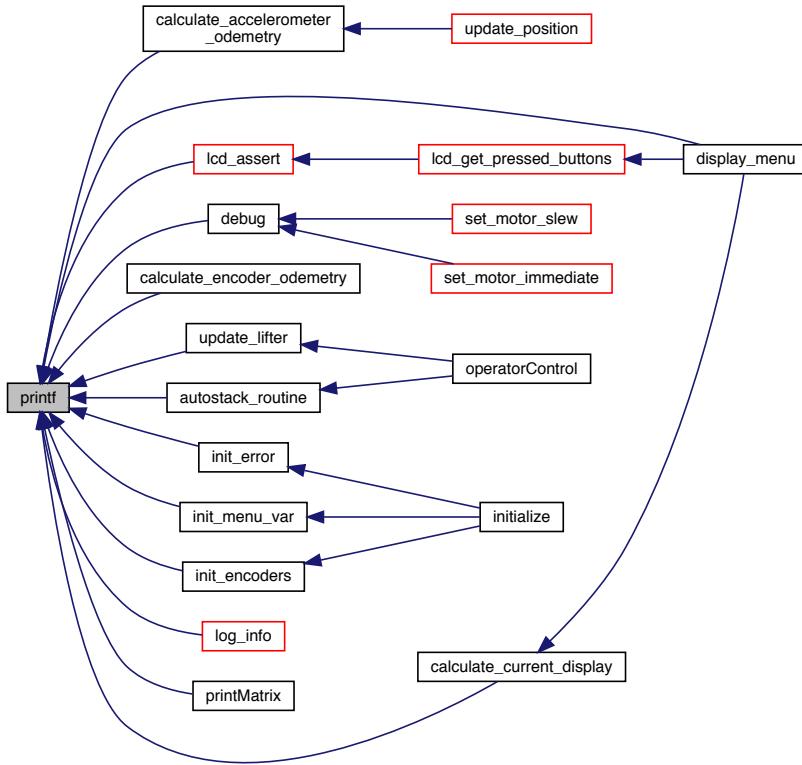
<i>formatString</i>	the format string as specified in fprintf() (p. ??)
---------------------	--

Returns

the number of characters written

Referenced by **autostack_routine()**, **calculate_accelerometer_odometry()**, **calculate_current_display()**, **calculate_encoder_odometry()**, **debug()**, **display_menu()**, **init_encoders()**, **init_error()**, **init_menu_var()**, **lcd_assert()**, **log_info()**, **printMatrix()**, and **update_lifter()**.

Here is the caller graph for this function:



6.1.3.74 putchar()

```
int putchar (
    int value )
```

Writes one character to "stdout", which is the PC debug terminal, and returns the input value.

When using a wireless connection, one may need to press the spacebar before the input is visible on the terminal.

Parameters

<code>value</code>	the character to write (a value of type "char" can be used)
--------------------	---

Returns

the character written

6.1.3.75 puts()

```
int puts (
    const char * string )
```

Behaves the same as the "print" function, and appends a trailing newline ("\n").

Parameters

<code>string</code>	the string to write
---------------------	---------------------

Returns

the number of characters written, excluding the new line

6.1.3.76 semaphoreCreate()

```
Semaphore semaphoreCreate ( )
```

Creates a semaphore intended for synchronizing tasks.

To prevent some critical code from simultaneously modifying a shared resource, use mutexes instead.

Semaphores created using this function can be accessed using the **semaphoreTake()** (p. ??) and **semaphoreGive()** (p. ??) functions. The mutex functions must not be used on objects of this type.

This type of object does not need to have balanced take and give calls, so priority inheritance is not used. Semaphores can be signalled by an interrupt routine.

Returns

a handle to the created semaphore

6.1.3.77 semaphoreDelete()

```
void semaphoreDelete (
    Semaphore semaphore )
```

Deletes the specified semaphore.

This function can be dangerous; deleting semaphores being waited on by a task may cause deadlock or a crash.

Parameters

<code>semaphore</code>	the semaphore to destroy
------------------------	--------------------------

6.1.3.78 semaphoreGive()

```
bool semaphoreGive (
    Semaphore semaphore )
```

Signals a semaphore.

Tasks waiting for a signal using **semaphoreTake()** (p. ??) will be unblocked by this call and can continue execution.

Slow processes can give semaphores when ready, and fast processes waiting to take the semaphore will continue at that point.

Parameters

<i>semaphore</i>	the semaphore to signal
------------------	-------------------------

Returns

true if the semaphore was successfully given, or false if the semaphore was not taken since the last give

6.1.3.79 semaphoreTake()

```
bool semaphoreTake (
    Semaphore semaphore,
    const unsigned long blockTime )
```

Waits on a semaphore.

If the semaphore is already in the "taken" state, the current task will wait for the semaphore to be signaled. Other tasks can run during this time.

Parameters

<i>semaphore</i>	the semaphore to wait
<i>blockTime</i>	the maximum time to wait for the semaphore to be given, where -1 specifies an infinite timeout

Returns

true if the semaphore was successfully taken, or false if the timeout expired

6.1.3.80 setTeamName()

```
void setTeamName (
    const char * name )
```

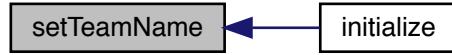
Sets the team name displayed to the VEX field control and VEX Firmware Upgrade.

Parameters

<i>name</i>	a string containing the team name; only the first eight characters will be shown
-------------	--

Referenced by **initialize()**.

Here is the caller graph for this function:



6.1.3.81 snprintf()

```
int snprintf (
    char * buffer,
    size_t limit,
    const char * formatString,
    ... )
```

Prints the formatted string to the string buffer with the specified length limit.

The length limit, as per the C standard, includes the trailing null character, so an argument of 256 will cause a maximum of 255 non-null characters to be printed, and one null terminator in all cases.

Parameters

<i>buffer</i>	the string buffer where characters can be placed
<i>limit</i>	the maximum number of characters to write
<i>formatString</i>	the format string as specified in fprintf() (p. ??)

Returns

the number of characters stored

6.1.3.82 speakerInit()

```
void speakerInit ( )
```

Initializes VEX speaker support.

The VEX speaker is not thread safe; it can only be used from one task at a time. Using the VEX speaker may impact robot performance. Teams may benefit from an if statement that only enables sound if **isOnline()** (p. ??) returns false.

6.1.3.83 speakerPlayArray()

```
void speakerPlayArray (
    const char ** songs )
```

Plays up to three RTTTL (Ring Tone Text Transfer Language) songs simultaneously over the VEX speaker.

The audio is mixed to allow polyphonic sound to be played. Many simple songs are available in RTTTL format online, or compose your own.

The song must not be NULL, but unused tracks within the song can be set to NULL. If any of the three song tracks is invalid, the result of this function is undefined.

The VEX speaker is not thread safe; it can only be used from one task at a time. Using the VEX speaker may impact robot performance. Teams may benefit from an if statement that only enables sound if **isOnline()** (p. ??) returns false.

Parameters

<i>songs</i>	an array of up to three (3) RTTTL songs as string values to play
--------------	--

6.1.3.84 speakerPlayRtttl()

```
void speakerPlayRtttl (
    const char * song )
```

Plays an RTTTL (Ring Tone Text Transfer Language) song over the VEX speaker.

Many simple songs are available in RTTTL format online, or compose your own.

The song must not be NULL. If an invalid song is specified, the result of this function is undefined.

The VEX speaker is not thread safe; it can only be used from one task at a time. Using the VEX speaker may impact robot performance. Teams may benefit from an if statement that only enables sound if **isOnline()** (p. ??) returns false.

Parameters

<i>song</i>	the RTTTL song as a string value to play
-------------	--

6.1.3.85 speakerShutdown()

```
void speakerShutdown ( )
```

Powers down and disables the VEX speaker.

If a song is currently being played in another task, the behavior of this function is undefined, since the VEX speaker is not thread safe.

6.1.3.86 sprintf()

```
int sprintf (
    char * buffer,
    const char * formatString,
    ... )
```

Prints the formatted string to the string buffer.

If the buffer is not big enough to contain the complete formatted output, undefined behavior occurs. See **sprintf()** (p. ??) for a safer version of this function.

Parameters

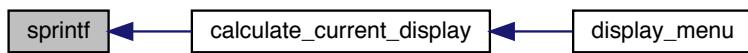
<i>buffer</i>	the string buffer where characters can be placed
<i>formatString</i>	the format string as specified in fprintf() (p. ??)

Returns

the number of characters stored

Referenced by **calculate_current_display()**.

Here is the caller graph for this function:



6.1.3.87 standaloneModeEnable()

```
void standaloneModeEnable ( )
```

Enables the Cortex to run the op control task in a standalone mode- no VEXnet connection required.

This function should only be called once in **initializeIO()** (p. 110)

6.1.3.88 taskCreate()

```
TaskHandle taskCreate (
    TaskCode taskCode,
    const unsigned int stackDepth,
    void * parameters,
    const unsigned int priority )
```

Creates a new task and add it to the list of tasks that are ready to run.

Parameters

<i>taskCode</i>	the function to execute in its own task
<i>stackDepth</i>	the number of variables available on the stack (4 * stackDepth bytes will be allocated on the Cortex)
<i>parameters</i>	an argument passed to the taskCode function
<i>priority</i>	a value from TASK_PRIORITY_LOWEST to TASK_PRIORITY_HIGHEST determining the initial priority of the task

Returns

a handle to the created task, or NULL if an error occurred

Referenced by **routine_task()**.

Here is the caller graph for this function:



6.1.3.89 taskDelay()

```
void taskDelay (
    const unsigned long msToDelay )
```

Delays the current task for a given number of milliseconds.

Delaying for a period of zero will force a reschedule, where tasks of equal priority may be scheduled if available. The calling task will still be available for immediate rescheduling once the other tasks have had their turn or if nothing of equal or higher priority is available to be scheduled.

This is not the best method to have a task execute code at predefined intervals, as the delay time is measured from when the delay is requested. To delay cyclically, use **taskDelayUntil()** (p. ??).

Parameters

<i>msToDelay</i>	the number of milliseconds to wait, with 1000 milliseconds per second
------------------	---

6.1.3.90 taskDelayUntil()

```
void taskDelayUntil (
    unsigned long * previousWakeTime,
    const unsigned long cycleTime )
```

Delays the current task until a specified time.

The task will be unblocked at the time *previousWakeTime + cycleTime, and *previousWakeTime will be changed to reflect the time at which the task will unblock.

If the target time is in the past, no delay occurs, but a reschedule is forced, as if **taskDelay()** (p. ??) was called with an argument of zero. If the sum of cycleTime and *previousWakeTime overflows or underflows, undefined behavior occurs.

This function should be used by cyclical tasks to ensure a constant execution frequency. While **taskDelay()** (p. ??) specifies a wake time relative to the time at which the function is called, **taskDelayUntil()** (p. ??) specifies the absolute future time at which it wishes to unblock. Calling taskDelayUntil with the same cycleTime parameter value in a loop, with previousWakeTime referring to a local variable initialized to **millis()** (p. ??), will cause the loop to execute with a fixed period.

Parameters

<i>previousWakeTime</i>	a pointer to the location storing the last unblock time, obtained by using the "&" operator on a variable (e.g. "taskDelayUntil(&now, 50);")
<i>cycleTime</i>	the number of milliseconds to wait, with 1000 milliseconds per second

6.1.3.91 taskDelete()

```
void taskDelete (
```

```
TaskHandle taskToDelete )
```

Kills and removes the specified task from the kernel task list.

Deleting the last task will end the program, possibly leading to undesirable states as some outputs may remain in their last set configuration.

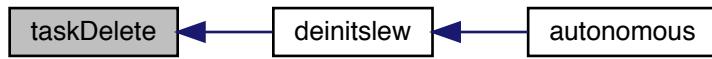
NOTE: The idle task is responsible for freeing the kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of processing time. Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

Parameters

<i>taskToDelete</i>	the task to kill; passing NULL kills the current task
---------------------	---

Referenced by **deinitSlew()**.

Here is the caller graph for this function:



6.1.3.92 taskGetCount()

```
unsigned int taskGetCount ( )
```

Determines the number of tasks that are currently being managed.

This includes all ready, blocked and suspended tasks. A task that has been deleted but not yet freed by the idle task will also be included in the count. Tasks recently created may take one context switch to be counted.

Returns

the number of tasks that are currently running, waiting, or suspended

6.1.3.93 taskGetState()

```
unsigned int taskGetState (
    TaskHandle task )
```

Retrieves the state of the specified task.

Note that the state of tasks which have died may be re-used for future tasks, causing the value returned by this function to reflect a different task than possibly intended in this case.

Parameters

<i>task</i>	Handle to the task to query. Passing NULL will query the current task status (which will, by definition, be TASK_RUNNING if this call returns)
-------------	--

Returns

A value reflecting the task's status, one of the constants TASK_DEAD, TASK_RUNNING, TASK_RUNNABLE, TASK_SLEEPING, or TASK_SUSPENDED

6.1.3.94 taskPriorityGet()

```
unsigned int taskPriorityGet (
    const TaskHandle task )
```

Obtains the priority of the specified task.

Parameters

<i>task</i>	the task to check; passing NULL checks the current task
-------------	---

Returns

the priority of that task from 0 to TASK_MAX_PRIORITIES

6.1.3.95 taskPrioritySet()

```
void taskPrioritySet (
    TaskHandle task,
    const unsigned int newPriority )
```

Sets the priority of the specified task.

A context switch may occur before the function returns if the priority being set is higher than the currently executing task and the task being mutated is available to be scheduled.

Parameters

<i>task</i>	the task to change; passing NULL changes the current task
<i>newPriority</i>	a value between TASK_PRIORITY_LOWEST and TASK_PRIORITY_HIGHEST inclusive indicating the new task priority

6.1.3.96 taskResume()

```
void taskResume (
    TaskHandle taskToResume )
```

Resumes the specified task.

A task that has been suspended by one or more calls to **taskSuspend()** (p. ??) will be made available for scheduling again by a call to **taskResume()** (p. ??). If the task was not suspended at the time of the call to **taskResume()** (p. ??), undefined behavior occurs.

Parameters

taskToResume	the task to change; passing NULL is not allowed as the current task cannot be suspended (it is obviously running if this function is called)
---------------------	--

6.1.3.97 taskRunLoop()

```
TaskHandle taskRunLoop (
    void(*)(void) fn,
    const unsigned long increment )
```

Starts a task which will periodically call the specified function.

Intended for use as a quick-start skeleton for cyclic tasks with higher priority than the "main" tasks. The created task will have priority **TASK_PRIORITY_DEFAULT + 1** with the default stack size. To customize behavior, create a task manually with the specified function.

This task will automatically terminate after one further function invocation when the robot is disabled or when the robot mode is switched.

Parameters

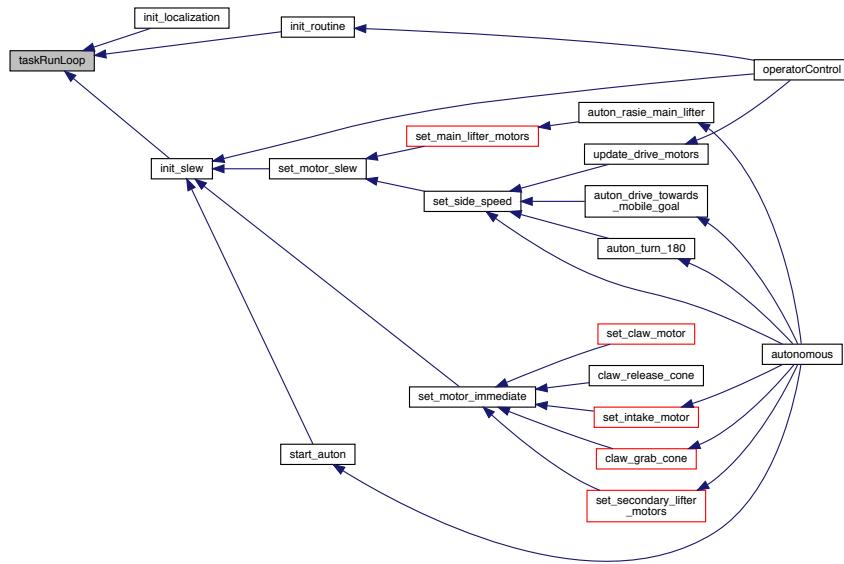
<i>fn</i>	the function to call in this loop
<i>increment</i>	the delay between successive calls in milliseconds; the taskDelayUntil() (p. ??) function is used for accurate cycle timing

Returns

a handle to the task, or NULL if an error occurred

Referenced by **init_localization()**, **init_routine()**, and **init_slew()**.

Here is the caller graph for this function:



6.1.3.98 taskSuspend()

```
void taskSuspend (
    TaskHandle taskToSuspend )
```

Suspends the specified task.

When suspended a task will not be scheduled, regardless of whether it might be otherwise available to run.

Parameters

<code>taskToSuspend</code>	the task to suspend; passing NULL suspends the current task
----------------------------	---

6.1.3.99 ultrasonicGet()

```
int ultrasonicGet (
    Ultrasonic ult )
```

Gets the current ultrasonic sensor value in centimeters.

If no object was found or if the ultrasonic sensor is polled while it is pinging and waiting for a response, -1 (ULTRA_B←AD_RESPONSE) is returned. If the ultrasonic sensor was never started, the return value is undefined. Round and fluffy objects can cause inaccurate values to be returned.

Parameters

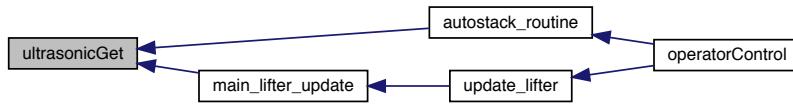
<i>ult</i>	the Ultrasonic object from ultrasonicInit() (p. ??) to read
------------	--

Returns

the distance to the nearest object in centimeters

Referenced by **autostack_routine()**, and **main_lifter_update()**.

Here is the caller graph for this function:

**6.1.3.100 ultrasonicInit()**

```
Ultrasonic ultrasonicInit (
    unsigned char portEcho,
    unsigned char portPing )
```

Initializes an ultrasonic sensor on the specified digital ports.

The ultrasonic sensor will be polled in the background in concert with the other sensors registered using this method. NULL will be returned if either port is invalid or the ultrasonic sensor port is already in use.

Parameters

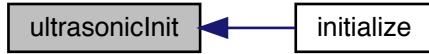
<i>portEcho</i>	the port connected to the orange cable from 1-9,11-12
<i>portPing</i>	the port connected to the yellow cable from 1-12

Returns

an Ultrasonic object to be stored and used for later calls to ultrasonic functions

Referenced by **initialize()**.

Here is the caller graph for this function:



6.1.3.101 ultrasonicShutdown()

```
void ultrasonicShutdown (
    Ultrasonic ult )
```

Stops and disables the ultrasonic sensor.

The last distance it had before stopping will be retained. One more ping operation may occur before the sensor is fully disabled.

Parameters

<i>ult</i>	the Ultrasonic object from ultrasonicInit() (p. ??) to stop
------------	--

6.1.3.102 usartInit()

```
void usartInit (
    PROS_FILE * usart,
    unsigned int baud,
    unsigned int flags )
```

Initialize the specified serial interface with the given connection parameters.

I/O to the port is accomplished using the "standard" I/O functions such as **fputs()** (p. ??), **fprintf()** (p. ??), and **fputc()** (p. ??).

Re-initializing an open port may cause loss of data in the buffers. This routine may be safely called from **initializeIO()** (p. 110) or when the scheduler is paused. If I/O is attempted on a serial port which has never been opened, the behavior will be the same as if the port had been disabled.

Parameters

<i>usart</i>	the port to open, either "uart1" or "uart2"
<i>baud</i>	the baud rate to use from 2400 to 1000000 baud
<i>flags</i>	a bit mask combination of the SERIAL_* flags specifying parity, stop, and data bits

6.1.3.103 usartShutdown()

```
void usartShutdown (
    PROS_FILE * usart )
```

Disables the specified USART interface.

Any data in the transmit and receive buffers will be lost. Attempts to read from the port when it is disabled will deadlock, and attempts to write to it may deadlock depending on the state of the buffer.

Parameters

<i>usart</i>	the port to close, either "uart1" or "uart2"
--------------	--

6.1.3.104 wait()

```
void wait (
    const unsigned long time )
```

Alias of **taskDelay()** (p. ??) intended to help EasyC users.

Parameters

<i>time</i>	the duration of the delay in milliseconds (1 000 milliseconds per second)
-------------	---

6.1.3.105 waitUntil()

```
void waitUntil (
    unsigned long * previousWakeTime,
    const unsigned long time )
```

Alias of **taskDelayUntil()** (p. ??) intended to help EasyC users.

Parameters

<i>previousWakeTime</i>	a pointer to the last wakeup time
<i>time</i>	the duration of the delay in milliseconds (1 000 milliseconds per second)

6.1.3.106 watchdogInit()

```
void watchdogInit( )
```

Enables IWDG watchdog timer which will reset the cortex if it locks up due to static shock or a misbehaving task preventing the timer to be reset.

Not recovering from static shock will cause the robot to continue moving its motors indefinitely until turned off manually.

This function should only be called once in **initializeIO()** (p. 110)

Referenced by **initializeIO()**.

Here is the caller graph for this function:



6.1.4 Variable Documentation

6.1.4.1 formatString

```
void unsigned char const char* formatString
```

Definition at line **1182** of file **API.h**.

6.1.4.2 line

```
void unsigned char line
```

Definition at line **1182** of file **API.h**.

6.2 API.h

```
00001
00021 #ifndef API_H_
00022 #define API_H_
00023
00024 // System includes
00025 #include <stdlib.h>
00026 #include <stdbool.h>
00027 #include <stdarg.h>
00028 #include <stdint.h>
00029
00030 // Begin C++ extern to C
00031 #ifdef __cplusplus
00032 extern "C" {
00033 #endif
00034
00035 // ----- VEX competition functions -----
00036
00040 #define JOY_DOWN 1
00041
00044 #define JOY_LEFT 2
00045
00048 #define JOY_UP 4
00049
00052 #define JOY_RIGHT 8
00053
00056 #define ACCEL_X 5
00057
00060 #define ACCEL_Y 6
00061
00068 bool isAutonomous();
00077 bool isEnabled();
00088 bool isJoystickConnected(unsigned char joystick);
00096 bool isOnline();
00104 int joystickGetAnalog(unsigned char joystick, unsigned char axis);
00114 bool joystickGetDigital(unsigned char joystick, unsigned char buttonGroup,
00115     unsigned char button);
00121 unsigned int powerLevelBackup();
00128 unsigned int powerLevelMain();
00134 void setTeamName(const char *name);
00135
00136 // ----- Pin control functions -----
00137
00141 #define BOARD_NR_ADC_PINS 8
00142
00151 #define BOARD_NR_GPIO_PINS 27
00152
00157 #define HIGH 1
00158
00163 #define LOW 0
00164
00172 #define INPUT 0x0A
00173
00179 #define INPUT_ANALOG 0x00
00180
00186 #define INPUT_FLOATING 0x04
00187
00193 #define OUTPUT 0x01
00194
00200 #define OUTPUT_OD 0x05
00201
00221 int analogCalibrate(unsigned char channel);
00237 int analogRead(unsigned char channel);
00252 int analogReadCalibrated(unsigned char channel);
00271 int analogReadCalibratedHR(unsigned char channel);
00287 bool digitalRead(unsigned char pin);
00298 void digitalWrite(unsigned char pin, bool value);
00309 void pinMode(unsigned char pin, unsigned char mode);
00310
00311 /*
00312 * Digital port 10 cannot be used as an interrupt port, or for an encoder. Plan accordingly.
00313 */
00314
00318 #define INTERRUPT_EDGE_RISING 1
00319
00322 #define INTERRUPT_EDGE_FALLING 2
00323
00327 #define INTERRUPT_EDGE_BOTH 3
00328
```

```
00332 typedef void (*InterruptHandler)(unsigned char pin);
00333
00341 void ioClearInterrupt(unsigned char pin);
00362 void ioSetInterrupt(unsigned char pin, unsigned char edges, InterruptHandler handler);
00363
00364 // ----- Physical output control functions -----
00365
00379 int motorGet(unsigned char channel);
00390 void motorSet(unsigned char channel, int speed);
00400 void motorStop(unsigned char channel);
00405 void motorStopAll();
00406
00414 void speakerInit();
00429 void speakerPlayArray(const char * * songs);
00443 void speakerPlayRttl(const char *song);
00450 void speakerShutdown();
00451
00452 // ----- VEX sensor control functions -----
00453
00458 #define IME_ADDR_MAX 0x1F
00459
00479 unsigned int imeInitializeAll();
00500 bool imeGet(unsigned char address, int *value);
00523 bool imeGetVelocity(unsigned char address, int *value);
00532 bool imeReset(unsigned char address);
00540 void imeShutdown();
00541
00548 typedef void * Gyro;
00549
00559 int gyroGet(Gyro gyro);
00579 Gyro gyroInit(unsigned char port, unsigned short multiplier);
00588 void gyroReset(Gyro gyro);
00597 void gyroShutdown(Gyro gyro);
00598
00605 typedef void * Encoder;
00614 int encoderGet(Encoder enc);
00627 Encoder encoderInit(unsigned char portTop, unsigned char portBottom, bool
    reverse);
00636 void encoderReset(Encoder enc);
00645 void encoderShutdown(Encoder enc);
00646
00650 #define ULTRA_BAD_RESPONSE -1
00651
00658 typedef void * Ultrasonic;
00670 int ultrasonicGet(Ultrasonic ult);
00682 Ultrasonic ultrasonicInit(unsigned char portEcho, unsigned char portPing);
00691 void ultrasonicShutdown(Ultrasonic ult);
00692
00693 // ----- Custom sensor control functions -----
00694
00695 // ---- I2C port control ----
00706 bool i2cRead(uint8_t addr, uint8_t *data, uint16_t count);
00719 bool i2cReadRegister(uint8_t addr, uint8_t reg, uint8_t *value, uint16_t count);
00732 bool i2cWrite(uint8_t addr, uint8_t *data, uint16_t count);
00742 bool i2cWriteRegister(uint8_t addr, uint8_t reg, uint16_t value);
00743
00750 typedef int PROS_FILE;
00751
00752
00753 #ifndef FILE
00754
00759 #define FILE PROS_FILE
00760 #endif
00761
00765 #define SERIAL_DATABITS_8 0x0000
00766
00769 #define SERIAL_DATABITS_9 0x1000
00770
00773 #define SERIAL_STOPBITS_1 0x0000
00774
00777 #define SERIAL_STOPBITS_2 0x2000
00778
00781 #define SERIAL_PARITY_NONE 0x0000
00782
00785 #define SERIAL_PARITY_EVEN 0x0400
00786
00789 #define SERIAL_PARITY_ODD 0x0600
00790
00793 #define SERIAL_8N1 0x0000
00794
00811 void usartInit(PROS_FILE *usart, unsigned int baud, unsigned int flags);
```

```
00821 void usartShutdown(PROS_FILE *usart);
00822
00823 // ----- Character input and output -----
00824
00825 #define stdout ((PROS_FILE *)3)
00826
00827 #define stdin ((PROS_FILE *)3)
00828
00829 #define uart1 ((PROS_FILE *)1)
00830
00831 #define uart2 ((PROS_FILE *)2)
00832
00833 #ifndef EOF
00834
00835 #define EOF ((int)-1)
00836 #endif
00837
00838 #ifndef SEEK_SET
00839
00840 #define SEEK_SET 0
00841 #endif
00842
00843 #ifndef SEEK_CUR
00844
00845 #define SEEK_CUR 1
00846 #endif
00847
00848 #ifndef SEEK_END
00849
00850 #define SEEK_END 2
00851 #endif
00852
00853 void fclose(PROS_FILE *stream);
00854 int fcount(PROS_FILE *stream);
00855 int fdelete(const char *file);
00856 int feof(PROS_FILE *stream);
00857 int fflush(PROS_FILE *stream);
00858 int fgetc(PROS_FILE *stream);
00859 char* fgets(char *str, int num, PROS_FILE *stream);
00860 PROS_FILE * fopen(const char *file, const char *mode);
00861 void fprintf(const char *string, PROS_FILE *stream);
00862 int fputc(int value, PROS_FILE *stream);
00863 int fputs(const char *string, PROS_FILE *stream);
00864 size_t fread(void *ptr, size_t size, size_t count, PROS_FILE *stream);
00865 int fseek(PROS_FILE *stream, long int offset, int origin);
00866 long int ftell(PROS_FILE *stream);
00867 size_t fwrite(const void *ptr, size_t size, size_t count, PROS_FILE *stream);
00868 int getchar();
00869 void print(const char *string);
00870 int putchar(int value);
00871 int puts(const char *string);
00872
00873 int fprintf(PROS_FILE *stream, const char *formatString, ...);
00874 int printf(const char *formatString, ...);
00875 int sprintf(char *buffer, size_t limit, const char *formatString, ...);
00876 int snprintf(char *buffer, size_t limit, const char *formatString, ...);
00877
00878 #define LCD_BTN_LEFT 1
00879
00880 #define LCD_BTN_CENTER 2
00881
00882 #define LCD_BTN_RIGHT 4
00883
00884 void lcdClear(PROS_FILE *lcdPort);
00885 void lcdInit(PROS_FILE *lcdPort);
00886 #ifdef DOXYGEN
00887 void lcdPrint(PROS_FILE *lcdPort, unsigned char line, const char *formatString, ...);
00888 #else
00889 void __attribute__((format (printf, 3, 4))) lcdPrint(PROS_FILE *lcdPort, unsigned char
00890 line,
00891 const char *formatString, ...);
00892 #endif
00893
00894 unsigned int lcdReadButtons(PROS_FILE *lcdPort);
00895 void lcdSetBacklight(PROS_FILE *lcdPort, bool backlight);
00896 void lcdSetText(PROS_FILE *lcdPort, unsigned char line, const char *buffer);
00897 void lcdShutdown(PROS_FILE *lcdPort);
00898
00899 // ----- Real-time scheduler functions -----
00900 #define TASK_MAX 16
00901
00902 #define TASK_MAX_PRIORITIES 6
00903
```

```
01243 #define TASK_PRIORITY_LOWEST 0
01244
01249 #define TASK_PRIORITY_DEFAULT 2
01250
01254 #define TASK_PRIORITY_HIGHEST (TASK_MAX_PRIORITIES - 1)
01255
01262 #define TASK_DEFAULT_STACK_SIZE 512
01263
01270 #define TASK_MINIMAL_STACK_SIZE 64
01271
01275 #define TASK_DEAD 0
01276
01279 #define TASK_RUNNING 1
01280
01284 #define TASK_RUNNABLE 2
01285
01289 #define TASK_SLEEPING 3
01290
01293 #define TASK_SUSPENDED 4
01294
01300 typedef void * TaskHandle;
01306 typedef void * Mutex;
01312 typedef void * Semaphore;
01323 typedef void (*TaskCode)(void *);
01324
01336 TaskHandle taskCreate(TaskCode taskCode, const unsigned int stackDepth, void *parameters,
01337     const unsigned int priority);
01352 void taskDelay(const unsigned long msToDelay);
01373 void taskDelayUntil(unsigned long *previousWakeTime, const unsigned long cycleTime);
01387 void taskDelete(TaskHandle taskToDelete);
01397 unsigned int taskGetCount();
01409 unsigned int taskGetState(TaskHandle task);
01416 unsigned int taskPriorityGet(const TaskHandle task);
01427 void taskPrioritySet(TaskHandle task, const unsigned int newPriority);
01438 void taskResume(TaskHandle taskToResume);
01454 TaskHandle taskRunLoop(void (*fn)(void), const unsigned long increment);
01463 void taskSuspend(TaskHandle taskToSuspend);
01464
01477 Semaphore semaphoreCreate();
01489 bool semaphoreGive(Semaphore semaphore);
01499 bool semaphoreTake(Semaphore semaphore, const unsigned long blockTime);
01506 void semaphoreDelete(Semaphore semaphore);
01507
01520 Mutex mutexCreate();
01528 bool mutexGive(Mutex mutex);
01540 bool mutexTake(Mutex mutex, const unsigned long blockTime);
01547 void mutexDelete(Mutex mutex);
01548
01554 void delay(const unsigned long time);
01565 void delayMicroseconds(const unsigned long us);
01575 unsigned long micros();
01584 unsigned long millis();
01590 void wait(const unsigned long time);
01597 void waitUntil(unsigned long *previousWakeTime, const unsigned long time);
01605 void watchdogInit();
01611 void standaloneModeEnable();
01612
01613 // End C++ extern to C
01614 #ifdef __cplusplus
01615 }
01616 #endif
01617
01618 #endif
```

6.3 include/auto.h File Reference

Autonomous declarations and macros.

6.3.1 Detailed Description

Autonomous declarations and macros.

Authors

Chris Jerrett, Christian Desimone

Date

9/18/2017

Definition in file **auto.h**.

6.4 auto.h

```

00001
00007 #ifndef _AUTO_H_
00008 #define _AUTO_H_
00009
00010 #include "claw.h"
00011 #include "drive.h"
00012 #include "lifter.h"
00013 #include "localization.h"
00014 #include "mobile_goal_intake.h"
00015 #include "sensors.h"
00016
00020 #define FRONT_LEFTIME 0
00021
00025 #define MID_LEFT_DRIVE 1
00026
00030 #define MID_RIGHT_DRIVE 4
00031
00035 #define STOP_ONE 500
00036
00040 #define GOAL_HEIGHT 1325
00041
00045 #define DEPLOY_HEIGHT 2000
00046
00050 #define LOWEST_HEIGHT 0
00051
00055 #define MOBILE_GOAL_HEIGHT 3570
00056
00060 #define MOBILE_GOAL_DISTANCE 4000
00061
00065 #define MAX_HEIGHT 3570
00066
00069 #define ZONE_DISTANCE 1000
00070
00073 #define HALF_ROTATE M_PI
00074
00075#endif

```

6.5 include/battery.h File Reference

Battery management related functions.

Functions

- double **backup_battery_voltage ()**
gets the backup battery voltage
- bool **battery_level_acceptable ()**
returns if the batteries are acceptable
- double **main_battery_voltage ()**
gets the main battery voltage

6.5.1 Detailed Description

Battery management related functions.

Author

Chris Jerrett

Date

9/18/2017

Definition in file **battery.h**.

6.5.2 Function Documentation

6.5.2.1 backup_battery_voltage()

double backup_battery_voltage ()

gets the backup battery voltage

Author

Chris Jerrett

Definition at line 14 of file **battery.c**.

References **powerLevelBackup()**.

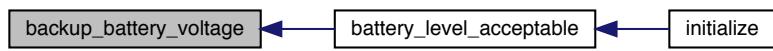
Referenced by **battery_level_acceptable()**.

00014 { **return** powerLevelBackup() / 1000.0; }

Here is the call graph for this function:



Here is the caller graph for this function:



6.5.2.2 battery_level_acceptable()

`bool battery_level_acceptable ()`

returns if the batteries are acceptable

See also

`MIN_MAIN_VOLTAGE`
`MIN_BACKUP_VOLTAGE`

Author

Chris Jerrett

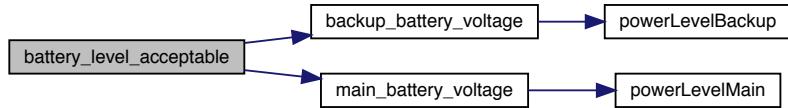
Definition at line **23** of file **battery.c**.

References `backup_battery_voltage()`, and `main_battery_voltage()`.

Referenced by `initialize()`.

```
00023     {
00024     if (main_battery_voltage() < MIN_MAIN_VOLTAGE)
00025         return false;
00026     if (backup_battery_voltage() < MIN_BACKUP_VOLTAGE)
00027         return false;
00028     return true;
00029 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.5.2.3 main_battery_voltage()

```
double main_battery_voltage ( )
```

gets the main battery voltage

Author

Chris Jerrett

Definition at line 8 of file **battery.c**.

References **powerLevelMain()**.

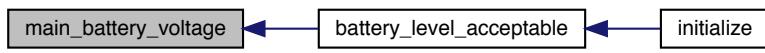
Referenced by **battery_level_acceptable()**.

```
00008 { return powerLevelMain() / 1000.0; }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.6 battery.h

```
00001
00007 #ifndef _BATTERY_H_
00008 #define _BATTERY_H_
00009
00010 #include <API.h>
00011
00015 #define MIN_MAIN_VOLTAGE 7.8
00016
00020 #define MIN_BACKUP_VOLTAGE 7.8
00021
00026 double main_battery_voltage();
00027
00032 double backup_battery_voltage();
00033
00041 bool battery_level_acceptable();
00042
00043 #endif
```

6.7 include/claw.h File Reference

Code for controlling the claw that grabs the cones.

Enumerations

- enum **claw_state** { **CLAW_OPEN_STATE**, **CLAW_CLOSE_STATE**, **CLAW_NEUTRAL_STATE** }
The different states of the claw.

Functions

- void **claw_grab_cone** ()
Drives the motors to open the claw.
- void **claw_release_cone** ()
Drives the motors to close the claw.
- unsigned int **getClawTicks** ()
Gets the claw position in potentiometer ticks.
- void **set_claw_motor** (const int v)
sets the claw motor speed
- void **update_claw** ()
Updates the claw motor values.

6.7.1 Detailed Description

Code for controlling the claw that grabs the cones.

Author

Chris Jerrett, Christian Desimone

Date

8/30/2017

Definition in file **claw.h**.

6.7.2 Enumeration Type Documentation

6.7.2.1 claw_state

```
enum claw_state
```

The different states of the claw.

Author

Chris Jerrett

Enumerator

CLAW_OPEN_STATE	
CLAW_CLOSE_STATE	
CLAW_NEUTRAL_STATE	

Definition at line **85** of file **claw.h**.

```
00085 { CLAW_OPEN_STATE, CLAW_CLOSE_STATE, CLAW_NEUTRAL_STATE };
```

6.7.3 Function Documentation

6.7.3.1 claw_grab_cone()

```
void claw_grab_cone ( )
```

Drives the motors to open the claw.

Author

Chris Jerrett

Drives the motors to open the claw.

Author

Chris Jerrett

See also

CLAW_MOTOR
MAX_CLAW_SPEED

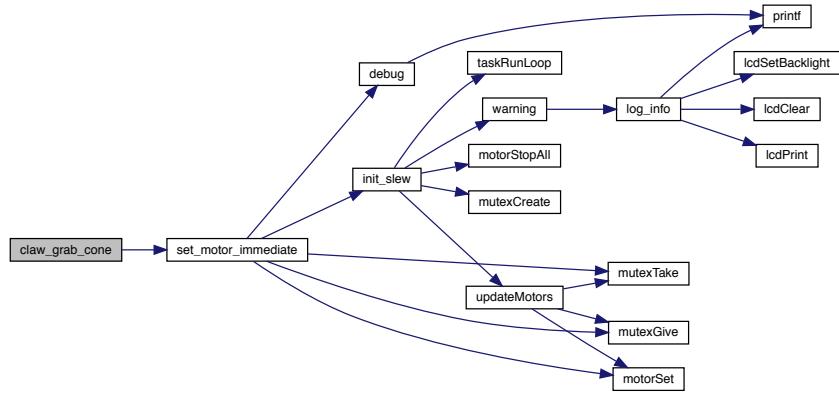
Definition at line **50** of file **claw.c**.

References **set_motor_immediate()**.

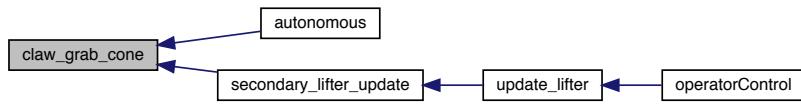
Referenced by **autonomous()**, and **secondary_lifter_update()**.

```
00050 { set_motor_immediate(CLAW_MOTOR, MAX_CLAW_SPEED); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.7.3.2 claw_release_cone()

```
void claw_release_cone( )
```

Drives the motors to close the claw.

Author

Chris Jerrett

Drives the motors to close the claw.

Author

Chris Jerrett

See also

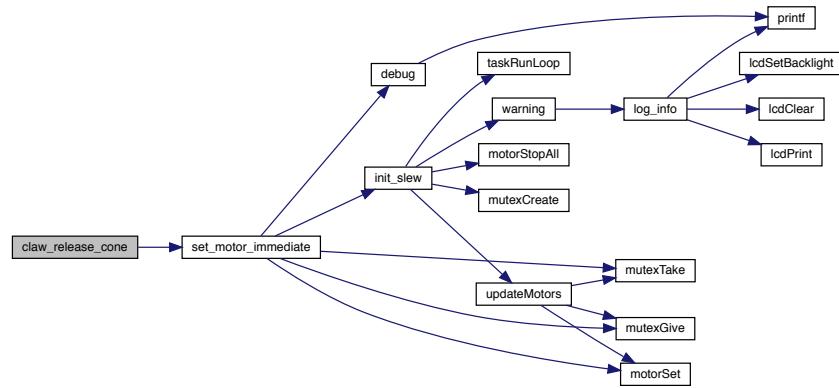
CLAW_MOTOR
MIN_CLAW_SPEED

Definition at line **58** of file **claw.c**.

References **set_motor_immediate()**.

```
00058 { set_motor_immediate(CLAW_MOTOR, MIN_CLAW_SPEED); }
```

Here is the call graph for this function:



6.7.3.3 getClawTicks()

```
unsigned int getClawTicks ( )
```

Gets the claw position in potentiometer ticks.

Author

Chris Jerrett

6.7.3.4 set_claw_motor()

```
void set_claw_motor (
    const int v )
```

sets the claw motor speed

Author

Chris Jerrett
Chris Jerrett

See also

CLAW_MOTOR

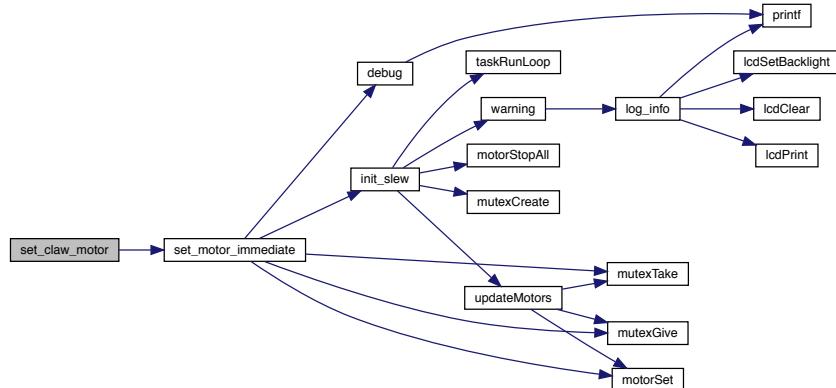
Definition at line 42 of file **claw.c**.

References **set_motor_immediate()**.

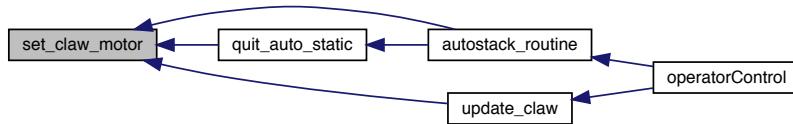
Referenced by **autostack_routine()**, **quit_auto_static()**, and **update_claw()**.

```
00042 { set_motor_immediate(CLAW_MOTOR, v); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.7.3.5 update_claw()

```
void update_claw ( )
```

Updates the claw motor values.

Author

Chris Jerrett
Chris Jerrett

See also

CLAW_CLOSE
CLAW_CLOSE_STATE (p. 30)
CLAW_OPEN
CLAW_OPEN_STATE (p. 30)
CLAW_NEUTRAL_STATE (p. 30)
MAX_CLAW_SPEED
MIN_CLAW_SPEED

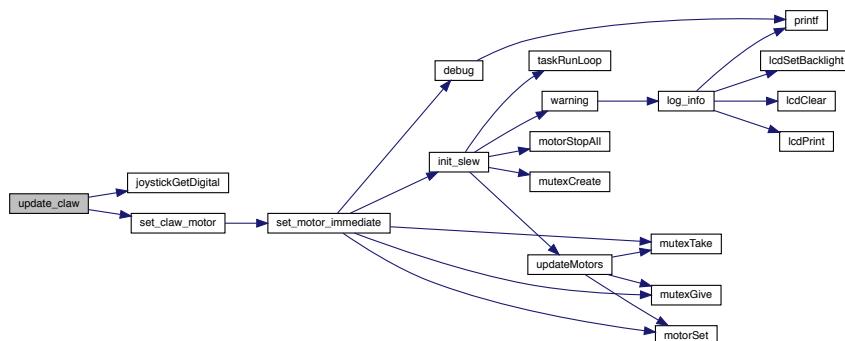
Definition at line 17 of file **claw.c**.

References **CLAW_CLOSE_STATE**, **CLAW_NEUTRAL_STATE**, **CLAW_OPEN_STATE**, **joystickGetDigital()**, **lifter_autostack_running**, **set_claw_motor()**, and **state**.

Referenced by **operatorControl()**.

```
00017     {
00018     if (lifter_autostack_running)
00019         return;
00020     if (joystickGetDigital(CLAW_CLOSE)) {
00021         state = CLAW_CLOSE_STATE;
00022     } else if (joystickGetDigital(CLAW_OPEN)) {
00023         state = CLAW_OPEN_STATE;
00024     } else {
00025         state = CLAW_NEUTRAL_STATE;
00026     }
00027
00028     if (state == CLAW_CLOSE_STATE) {
00029         set_claw_motor(MAX_CLAW_SPEED);
00030     } else if (state == CLAW_OPEN_STATE) {
00031         set_claw_motor(MIN_CLAW_SPEED);
00032     } else {
00033         set_claw_motor(0);
00034     }
00035 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.8 claw.h

```

00001
00007 #ifndef _CLAW_H_
00008 #define _CLAW_H_
00009
00010 #include "controller.h"
00011 #include "motor_ports.h"
00012 #include "sensors.h"
00013 #include "slew.h"
00014 #include <API.h>
00015
00020 #define MAX_CLAW_SPEED 127
00021
00025 #define MIN_CLAW_SPEED -127
00026
00031 #define CLAW_CLOSE PARTNER, 8, JOY_DOWN
00032
00037 #define CLAW_OPEN PARTNER, 8, JOY_RIGHT
00038
00043 #define CLAW_CLOSE_VAL 3000
00044
00049 #define CLAW_OPEN_VAL 1500
00050
00055 void update_claw();
00056
00061 void set_claw_motor(const int v);
00062
00067 unsigned int getClawTicks();
00068
00073 void claw_grab_cone();
00074
00079 void claw_release_cone();
00080
00085 enum claw_state { CLAW_OPEN_STATE, CLAW_CLOSE_STATE, CLAW_NEUTRAL_STATE };
00086
00087 #endif
  
```

6.9 include/controller.h File Reference

controller definitions, macros and functions to assist with usig the vex controllers.

Enumerations

- enum **button_t** {

JOY1_5D = 0, JOY1_5U = 1, JOY1_6D = 2, JOY1_6U = 3,

JOY1_7U = 4, JOY1_7L = 5, JOY1_7R = 6, JOY1_7D = 7,

JOY1_8U = 8, JOY1_8L = 9, JOY1_8R = 10, JOY1_8D = 11,
 }

```
JOY2_5D = 12, JOY2_5U = 13, JOY2_6D = 14, JOY2_6U = 15,  
JOY2_7U = 16, JOY2_7L = 17, JOY2_7R = 18, JOY2_7D = 19,  
JOY2_8U = 20, JOY2_8L = 21, JOY2_8R = 22, JOY2_8D = 23,  
LCD_LEFT = 24, LCD_CENT = 25, LCD_RIGHT = 26 }
```

renames the controller inputs.

- enum **joystick** { **RIGHT_JOY**, **LEFT_JOY** }

Represents a joystick on the controller.

Functions

- struct **cord get_joystick_cord** (enum **joystick side**, int controller)

Gets the location of a joystick on the controller.

6.9.1 Detailed Description

controller definitions, macros and functions to assist with usig the vex controllers.

Author

Chris Jerrett, Christian Desimone

Date

9/9/2017

Definition in file **controller.h**.

6.9.2 Enumeration Type Documentation

6.9.2.1 button_t

```
enum button_t
```

renames the controller inputs.

Allows more readable controls.

Author

Chris Jerrett

Date

12/18/17

Enumerator

JOY1_5D	
JOY1_5U	
JOY1_6D	
JOY1_6U	
JOY1_7U	
JOY1_7L	
JOY1_7R	
JOY1_7D	
JOY1_8U	
JOY1_8L	
JOY1_8R	
JOY1_8D	
JOY2_5D	
JOY2_5U	
JOY2_6D	
JOY2_6U	
JOY2_7U	
JOY2_7L	
JOY2_7R	
JOY2_7D	
JOY2_8U	
JOY2_8L	
JOY2_8R	
JOY2_8D	
LCD_LEFT	
LCD_CENT	
LCD_RIGHT	

Definition at line **25** of file **controller.h**.

```

00025          {
00026     JOY1_5D = 0,
00027     JOY1_5U = 1,
00028     JOY1_6D = 2,
00029     JOY1_6U = 3,
00030     JOY1_7U = 4,
00031     JOY1_7L = 5,
00032     JOY1_7R = 6,
00033     JOY1_7D = 7,
00034     JOY1_8U = 8,
00035     JOY1_8L = 9,
00036     JOY1_8R = 10,
00037     JOY1_8D = 11,
00038
00039     JOY2_5D = 12,
00040     JOY2_5U = 13,
00041     JOY2_6D = 14,
00042     JOY2_6U = 15,
00043     JOY2_7U = 16,
00044     JOY2_7L = 17,
00045     JOY2_7R = 18,
00046     JOY2_7D = 19,
00047     JOY2_8U = 20,
00048     JOY2_8L = 21,
00049     JOY2_8R = 22,
```

```
00050     JOY2_8D = 23,  
00051  
00052     LCD_LEFT = 24,  
00053     LCD_CENT = 25,  
00054     LCD_RIGHT = 26  
00055 } button_t;
```

6.9.2.2 joystick

```
enum joystick
```

Represents a joystick on the controller.

Date

9/10/2017

Author

Chris Jerrett

Enumerator

RIGHT_JOY	The right joystick.
LEFT_JOY	The left joystick.

Definition at line **104** of file **controller.h**.

```
00104     {  
00106     RIGHT_JOY,  
00108     LEFT_JOY,  
00109 };
```

6.9.3 Function Documentation

6.9.3.1 get_joystick_cord()

```
struct cord get_joystick_cord (  
    enum joystick side,  
    int controller )
```

Gets the location of a joystick on the controller.

Author

Chris Jerrett
Chris Jerrett

See also

RIGHT_JOY (p. 38)
RIGHT_JOY_X
RIGHT_JOY_Y
LEFT_JOY_X
LEFT_JOY_Y

Definition at line **12** of file **controller.c**.

References **joystickGetAnalog()**, **RIGHT_JOY**, **cord::x**, and **cord::y**.

```

00012
00013     int x;
00014     int y;
00015     // Get the joystick value for either the right or left,
00016     // depending on the mode
00017     if (side == RIGHT_JOY) {
00018         y = joystickGetAnalog(controller, RIGHT_JOY_X);
00019         x = joystickGetAnalog(controller, RIGHT_JOY_Y);
00020     } else {
00021         y = joystickGetAnalog(controller, LEFT_JOY_X);
00022         x = joystickGetAnalog(controller, LEFT_JOY_Y);
00023     }
00024     // Define a coordinate for the joystick value
00025     struct cord c;
00026     c.x = x;
00027     c.y = y;
00028     return c;
00029 }
```

Here is the call graph for this function:



6.10 controller.h

```

00001
00009 #ifndef _CONTROLLER_H_
00010 #define _CONTROLLER_H_
00011
00012 #include "vmath.h"
00013 #include <API.h>
00014
00015 #define RIGHT_BUTTONS 8
00016 #define LEFT_BUTTONS 7
```

```

00017 #define RIGHT_BUMPERS 5
00018 #define LEFT_BUMPERS 6
00019
00025 typedef enum {
00026     JOY1_5D = 0,
00027     JOY1_5U = 1,
00028     JOY1_6D = 2,
00029     JOY1_6U = 3,
00030     JOY1_7U = 4,
00031     JOY1_7L = 5,
00032     JOY1_7R = 6,
00033     JOY1_7D = 7,
00034     JOY1_8U = 8,
00035     JOY1_8L = 9,
00036     JOY1_8R = 10,
00037     JOY1_8D = 11,
00038
00039     JOY2_5D = 12,
00040     JOY2_5U = 13,
00041     JOY2_6D = 14,
00042     JOY2_6U = 15,
00043     JOY2_7U = 16,
00044     JOY2_7L = 17,
00045     JOY2_7R = 18,
00046     JOY2_7D = 19,
00047     JOY2_8U = 20,
00048     JOY2_8L = 21,
00049     JOY2_8R = 22,
00050     JOY2_8D = 23,
00051
00052     LCD_LEFT = 24,
00053     LCD_CENT = 25,
00054     LCD_RIGHT = 26
00055 } button_t;
00056
00062 #define MASTER 1
00063
00069 #define PARTNER 2
00070
00076 #define RIGHT_JOY_X 1
00077
00083 #define RIGHT_JOY_Y 2
00084
00090 #define LEFT_JOY_X 4
00091
00097 #define LEFT_JOY_Y 3
00098
00104 enum joystick {
00106     RIGHT_JOY,
00108     LEFT_JOY,
00109 };
00110
00115 struct cord get_joystick_cord(enum joystick side, int controller);
00116
00117 #endif

```

6.11 include/drive.h File Reference

Drive base definitions and enumerations.

Typedefs

- **typedef enum side side_t**
enumeration indication side of the robot.

Enumerations

- **enum side { LEFT, BOTH, RIGHT }**
enumeration indication side of the robot.

Functions

- void **set_side_speed** (**side_t side**, int speed)
sets the speed of one side of the robot.
- void **setThresh** (int t)
Sets the deadzone threshold on the drive.
- void **update_drive_motors** ()
Updates the drive motors during teleop.

6.11.1 Detailed Description

Drive base definitions and enumerations.

Author

Chris Jerrett

Date

9/9/2017

Definition in file **drive.h**.

6.11.2 Typedef Documentation

6.11.2.1 **side_t**

```
typedef enum side side_t  
enumeration indication side of the robot.
```

Author

Christian Desimone

Date

9/7/2017 Side can be right, both or left. Contained in side typedef, so enum is unnecessary.

6.11.3 Enumeration Type Documentation

6.11.3.1 **side**

```
enum side  
enumeration indication side of the robot.
```

Author

Christian Desimone

Date

9/7/2017 Side can be right, both or left. Contained in side typedef, so enum is unnecessary.

Enumerator

LEFT	
BOTH	
RIGHT	

Definition at line **26** of file **drive.h**.

```
00026 { LEFT, BOTH, RIGHT } side_t;
```

6.11.4 Function Documentation**6.11.4.1 set_side_speed()**

```
void set_side_speed (
    side_t side,
    int speed )
```

sets the speed of one side of the robot.

Author

Christian Desimone

Parameters

<i>side</i>	a side enum which indicates the size.
<i>speed</i>	the speed of the side. Can range from -127 - 127 negative being back and positive forwards

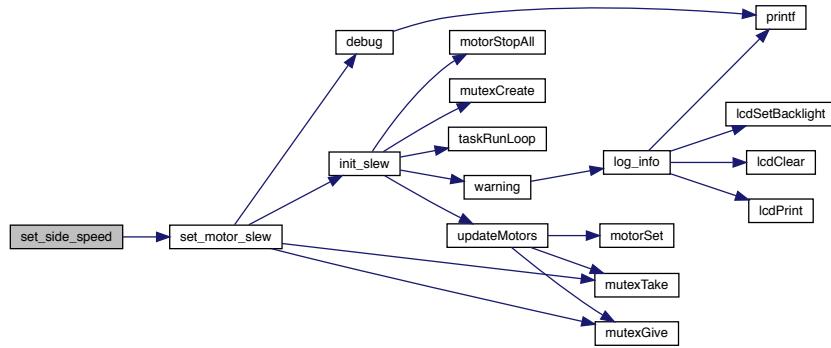
Definition at line **54** of file **drive.c**.

References **BOTH**, **LEFT**, **RIGHT**, and **set_motor_slew()**.

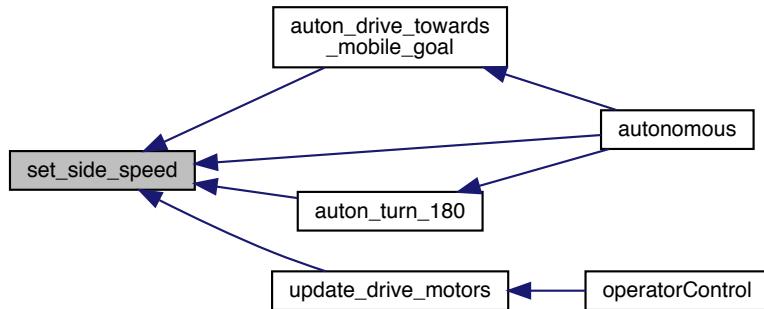
Referenced by **auton_drive_towards_mobile_goal()**, **auton_turn_180()**, **autonomous()**, and **update_drive_motors()**.

```
00054 {
00055     if (side == RIGHT || side == BOTH) {
00056         set_motor_slew(MOTOR_BACK_RIGHT, -speed);
00057         set_motor_slew(MOTOR_FRONT_RIGHT, -speed);
00058         set_motor_slew(MOTOR_MIDDLE_RIGHT, -speed);
00059     }
00060     if (side == LEFT || side == BOTH) {
00061         set_motor_slew(MOTOR_BACK_LEFT, speed);
00062         set_motor_slew(MOTOR_MIDDLE_LEFT, speed);
00063         set_motor_slew(MOTOR_FRONT_LEFT, speed);
00064     }
00065 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.11.4.2 setThresh()

```
void setThresh (
    int t )
```

Sets the deadzone threshhold on the drive.

Author

Chris Jerrett

Sets the deadzone threshhold on the drive.

Author

Christian Desimone

Definition at line **18** of file **drive.c**.

References **thresh**.

```
00018 { thresh = t; }
```

6.11.4.3 update_drive_motors()

```
void update_drive_motors ( )
```

Updates the drive motors during teleop.

Author

Christian Desimone

Date

9/5/17

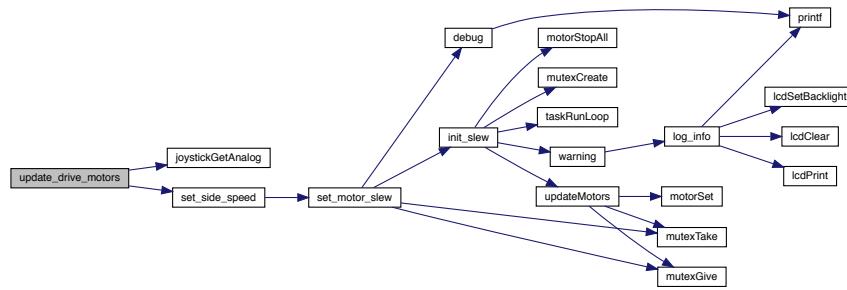
Definition at line **25** of file **drive.c**.

References **joystickGetAnalog()**, **LEFT**, **RIGHT**, **set_side_speed()**, **thresh**, **cord::x**, and **cord::y**.

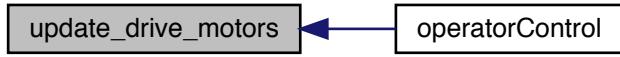
Referenced by **operatorControl()**.

```
00025 {
00026 // Get the joystick values from the controller
00027 int x = 0;
00028 int y = 0;
00029 x = -(joystickGetAnalog(MASTER, 3));
00030 y = (joystickGetAnalog(MASTER, 1));
00031 // Make sure the joystick values are significant enough to change the motors
00032 if (x < thresh && x > -thresh) {
00033     x = 0;
00034 }
00035 if (y < thresh && y > -thresh) {
00036     y = 0;
00037 }
00038 // Create motor values for the left and right from the x and y of the joystick
00039 int r = (x + y);
00040 int l = -(x - y);
00041
00042 // Set the drive motors
00043 set_side_speed(LEFT, l);
00044 set_side_speed(RIGHT, -r);
00045 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.12 drive.h

```

00001
00008 #ifndef _DRIVE_H_
00009 #define _DRIVE_H_
0010
0011 #include <API.h>
0012
0017 #define THRESHOLD 10
0018
0026 typedef enum side { LEFT, BOTH, RIGHT } side_t;
0027
0035 void set_side_speed(side_t side, int speed);
0036
0041 void setThresh(int t);
0042
0048 void update_drive_motors();
0049
0050 #endif
  
```

6.13 include/encoders.h File Reference

wrapper around encoder functions

Functions

- int **get_encoder_ticks** (unsigned char address)
Gets the encoder ticks since last reset.
- int **get_encoder_velocity** (unsigned char address)
Gets the encoder reads.
- bool **init_encoders** ()
Initializes all motor encoders.

6.13.1 Detailed Description

wrapper around encoder functions

Author

Chris Jerrett, Christian Desimone

Date

9/9/2017

Definition in file **encoders.h**.

6.13.2 Function Documentation

6.13.2.1 get_encoder_ticks()

```
int get_encoder_ticks (
    unsigned char address )
```

Gets the encoder ticks since last reset.

Author

Chris Jerrett

Date

9/15/2017

Definition at line **30** of file **encoders.c**.

References **imeGet()**.

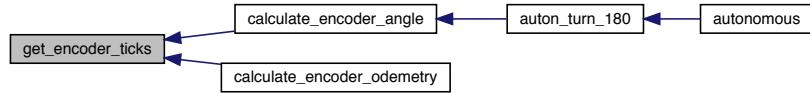
Referenced by **calculate_encoder_angle()**, and **calculate_encoder_odometry()**.

```
00030
00031     int i = 0;
00032     imeGet(address, &i);
00033     return i;
00034 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.13.2.2 **get_encoder_velocity()**

```
int get_encoder_velocity (
    unsigned char address )
```

Gets the encoder reads.

Author

Chris Jerrett

Date

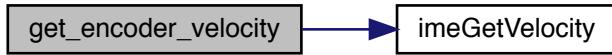
9/15/2017

Definition at line 41 of file **encoders.c**.

References **imeGetVelocity()**.

```
00041
00042     int i = 0;
00043     imeGetVelocity(address, &i);
00044     return i;
00045 }
```

Here is the call graph for this function:



6.13.2.3 init_encoders()

```
bool init_encoders( )
```

Initializes all motor encoders.

Author

Chris Jerrett

Date

9/9/2017

See also

[IME_NUMBER](#)

Definition at line 11 of file [encoders.c](#).

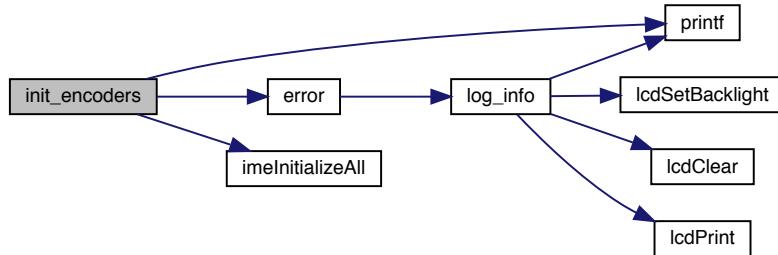
References [error\(\)](#), [imeInitializeAll\(\)](#), and [printf\(\)](#).

Referenced by [initialize\(\)](#).

```

00011
00012 #ifdef IME_NUMBER
00013     int count = imeInitializeAll();
00014     if (count != IME_NUMBER) {
00015         printf("detected only %d\n", count);
00016         error("Wrong Number of IMEs Connected");
00017         return false;
00018     }
00019     return true;
00020 #else
00021     return imeInitializeAll();
00022 #endif
00023 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.14 encoders.h

```
00001
00007 #ifndef _ENCODERS_H_
00008 #define _ENCODERS_H_
00009
00010 #include <API.h>
00011
00020 #define IME_NUMBER 2
00021
00028 bool init_encoders();
00029
00035 int get_encoder_ticks(unsigned char address);
00036
00042 int get_encoder_velocity(unsigned char address);
00043
00044 #endif
```

6.15 include/gyro.h File Reference

Functions

- float **get_main_gyro_angluar_velocity ()**
Gets the Gyro angular velocity.
- bool **init_main_gyro ()**
Initializes the main robot gyroscope/ Only call function when robot still and ready to start autonomous.

6.15.1 Function Documentation

6.15.1.1 get_main_gyro_angluar_velocity()

```
float get_main_gyro_angluar_velocity ( )
```

Gets the Gyro angular velocity.

Todo

Returns

the angular velocity

Author

Chris Jerrett

Date

11/30/17

Gets the Gyro angular velocity.

Author

Chris Jerrett

See also

GYRO_PORT

Definition at line **20** of file **gyro.c**.

References **analogReadCalibratedHR()**.

```
00020
00021     uint32_t port = GYRO_PORT;
00022     int32_t reading = (int32_t)analogReadCalibratedHR(port + 1);
00023     return 0;
00024 }
```

Here is the call graph for this function:



6.15.1.2 init_main_gyro()

```
bool init_main_gyro( )
```

Initializes the main robot gyroscope/ Only call function when robot still and ready to start autonomous.

Robot should not move for five seconds while Gyro calibrates

Returns

if the Gyro was sucessfully calibrated

Date

11/30/17

Author

Chris Jerrett

Initializes the main robot gyroscope/ Only call function when robot still and ready to start autonomous.

Author

Chris Jerrett

See also

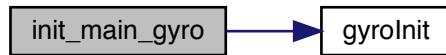
GYRO_PORT
GYRO_MULTIPLIER

Definition at line **10** of file **gyro.c**.

References **gyroInit()**, and **main_gyro**.

```
00010
00011     main_gyro = gyroInit(GYRO_PORT, GYRO_MULTIPLIER);
00012     return main_gyro != NULL;
00013 }
```

Here is the call graph for this function:

**6.16 gyro.h**

```
00001
00006 #ifndef _GYRO_H_
00007 #define _GYRO_H_
00008
00009 #include "API.h"
00010
00014 #define GYRO_PORT 1
00015
00020 #define GYRO_MULTIPLIER 0
00021
00030 bool init_main_gyro();
00031
00039 float get_main_gyro_angluar_velocity();
00040
00041 #endif
```

6.17 include/lcd.h File Reference

LCD wrapper functions and macros.

Data Structures

- struct **lcd_buttons**
represents the state of the lcd buttons

Enumerations

- enum **button_state** { **RELEASED** = false, **PRESSED** = true }
Represents the state of a button.

Functions

- void **init_main_lcd** (FILE *lcd)
Initializes the lcd screen.
- void **lcd_clear** ()
Clears the lcd.
- **lcd_buttons lcd_get_pressed_buttons** ()
Returns the pressed buttons.
- void **lcd_print** (unsigned int **line**, const char *str)
prints a string to a line on the lcd
- void **lcd_printf** (unsigned int **line**, const char *format_str,...)
prints a formated string to a line on the lcd.
- void **lcd_set_backlight** (bool **state**)
sets the backlight of the lcd
- void **prompt_confirmation** (const char *confirm_text)
Prompts the user to confirm a string.

6.17.1 Detailed Description

LCD wrapper functions and macros.

Author

Chris Jerrett

Date

9/9/2017

Definition in file **lcd.h**.

6.17.2 Enumeration Type Documentation

6.17.2.1 button_state

```
enum button_state
```

Represents the state of a button.

A button can be pressed or RELEASED. Release is false which is also 0. PRESSED is true or 1.

Author

Chris Jerrett

Date

9/9/2017

Enumerator

RELEASED	A released button.
PRESSED	A pressed button.

Definition at line **36** of file **lcd.h**.

```
00036      {
00038     RELEASED = false,
00040     PRESSED = true,
00041 } button_state;
```

6.17.3 Function Documentation

6.17.3.1 init_main_lcd()

```
void init_main_lcd (
    FILE * lcd )
```

Initializes the lcd screen.

Also will initialize the lcd_port var. Must be called before any lcd function can be called.

Parameters

<i>lcd</i>	the urart port of the lcd screen
------------	----------------------------------

See also

uart1
uart2

Author

Chris Jerrett

Date

9/9/2017

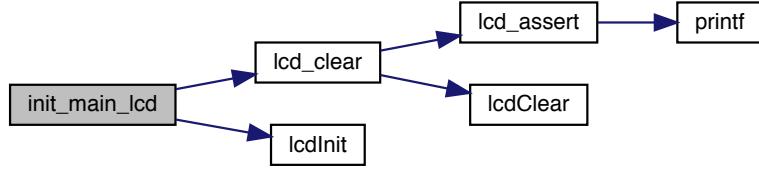
Definition at line **62** of file **Lcd.c**.

References **lcd_clear()**, **lcd_port**, and **lcdInit()**.

Referenced by **initialize()**.

```
00062
00063     lcd_port = lcd;
00064     lcdInit(lcd);
00065     lcd_clear();
00066 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.17.3.2 lcd_clear()

```
void lcd_clear ( )
```

Clears the lcd.

Author

Chris Jerrett

Date

9/9/2017

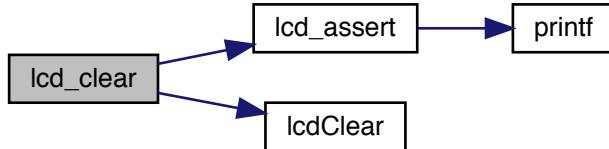
Definition at line 47 of file **lcd.c**.

References **lcd_assert()**, **lcd_port**, and **LcdClear()**.

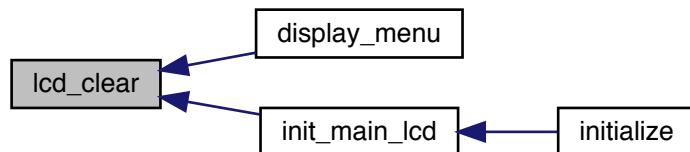
Referenced by **display_menu()**, and **init_main_lcd()**.

```
00047     {  
00048     lcd_assert();  
00049     lcdClear(lcd_port);  
00050 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.17.3.3 lcd_get_pressed_buttons()

```
lcd_buttons lcd_get_pressed_buttons ( )
```

Returns the pressed buttons.

Returns

a struct containing the states of all three buttons.

Author

Chris Jerrett

Date

9/9/2017

See also

[lcd_buttons](#) (p. 10)

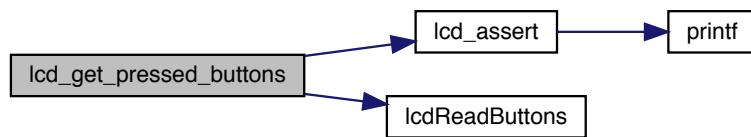
Definition at line **28** of file [lcd.c](#).

References [lcd_assert\(\)](#), [lcd_port](#), [lcdReadButtons\(\)](#), [lcd_buttons::left](#), [lcd_buttons::middle](#), [PRESSED](#), [RELEASED](#), and [lcd_buttons::right](#).

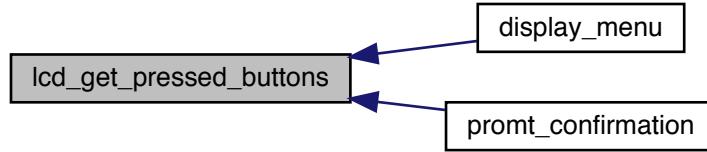
Referenced by [display_menu\(\)](#), and [prompt_confirmation\(\)](#).

```
00028                                     {
00029     lcd_assert();
00030     unsigned int btn_binary = lcdReadButtons(lcd_port);
00031     bool left = btn_binary & 0x1; // 0001
00032     bool middle = btn_binary & 0x2; // 0010
00033     bool right = btn_binary & 0x4; // 0100
00034     lcd_buttons btns;
00035     btns.left = left ? PRESSED : RELEASED;
00036     btns.middle = middle ? PRESSED : RELEASED;
00037     btns.right = right ? PRESSED : RELEASED;
00038
00039     return btns;
00040 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.17.3.4 `lcd_print()`

```
void lcd_print (
    unsigned int line,
    const char * str )
```

prints a string to a line on the lcd

Parameters

<i>line</i>	the line to print on
<i>str</i>	string to print

Author

Chris Jerrett

Date

9/9/2017

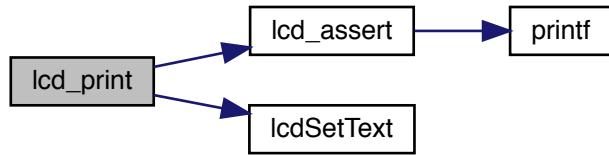
Definition at line **75** of file **lcd.c**.

References `lcd_assert()`, `lcd_port`, and `lcdSetText()`.

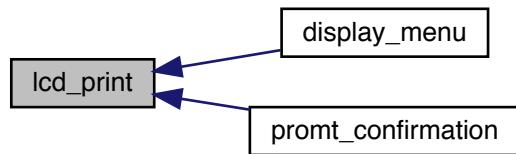
Referenced by `display_menu()`, and `prompt_confirmation()`.

```
00075
00076     lcd_assert();
00077     lcdSetText(lcd_port, line, str);
00078 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.17.3.5 `lcd_printf()`

```
void lcd_printf (
    unsigned int line,
    const char * format_str,
    ... )
```

prints a formated string to a line on the lcd.

Similar to `printf`

Parameters

<code>line</code>	the line to print on
<code>format_str</code>	format string string to print

Author

Chris Jerrett

Date

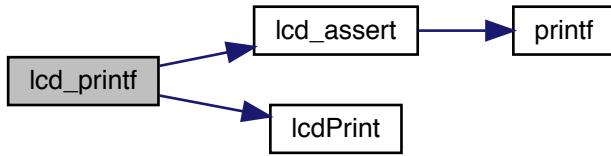
9/9/2017

Definition at line **87** of file **lcd.c**.

References **lcd_assert()**, **lcd_port**, and **lcdPrint()**.

```
00087 {  
00088     lcd_assert();  
00089     lcdPrint(lcd_port, line, format_str);  
00090 }
```

Here is the call graph for this function:



6.17.3.6 lcd_set_backlight()

```
void lcd_set_backlight (  
    bool state )
```

sets the backlight of the lcd

Parameters

<code>state</code>	a boolean representing the state of the backlight. true = on, false = off.
--------------------	--

Author

Chris Jerrett

Date

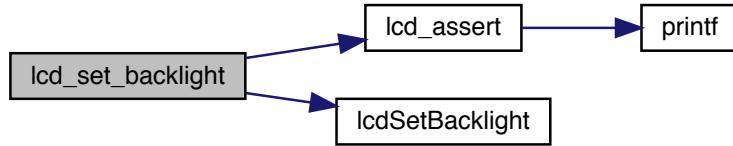
9/9/2017

Definition at line **99** of file **lcd.c**.References **lcd_assert()**, **lcd_port**, and **lcdSetBacklight()**.

```

00099
00100     lcd_assert();
00101     lcdSetBacklight(lcd_port, state);
00102 }
```

Here is the call graph for this function:

**6.17.3.7 prompt_confirmation()**

```

void prompt_confirmation (
    const char * confirm_text )
```

Prompts the user to confirm a string.

User must press middle button to confirm. Function is not thread safe and will stall a thread.

Parameters

<i>confirm_text</i>	the text for the user to confirm.
---------------------	-----------------------------------

Author

Chris Jerrett

Date

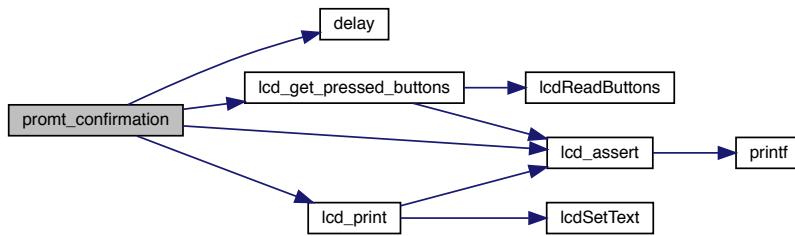
9/9/2017

Definition at line 113 of file **lcd.c**.References **delay()**, **lcd_assert()**, **lcd_get_pressed_buttons()**, **lcd_print()**, and **PRESSED**.

```

00113     {
00114     lcd_assert();
00115     lcd_print(1, confirm_text);
00116     while (lcd_get_pressed_buttons().middle != PRESSED) {
00117         delay(200);
00118     }
00119 }
```

Here is the call graph for this function:



6.18 lcd.h

```

00001
00008 #ifndef _LCD_H_
00009 #define _LCD_H_
00010
00011 #include <API.h>
00012
00018 #define TOP_ROW 1
00019
00025 #define BOTTOM_ROW 2
00026
00036 typedef enum {
00038     RELEASED = false,
00040     PRESSED = true,
00041 } button_state;
00042
00048 typedef struct {
00049     button_state left;
00050     button_state middle;
00051     button_state right;
00052 } lcd_buttons;
00053
00061 lcd_buttons lcd_get_pressed_buttons();
00062
00068 void lcd_clear();
00069
00080 void init_main_lcd(FILE *lcd);
00081
00089 void lcd_print(unsigned int line, const char *str);
00090
00098 void lcd_printf(unsigned int line, const char *format_str, ...);
00099
00107 void lcd_set_backlight(bool state);
00108
00118 void prompt_confirmation(const char *confirm_text);
00119
00120 #endif
```

6.19 include/lifter.h File Reference

Declarations and macros for controlling and manipulating the lifter.

Functions

- void **autostack_routine** (void *param)
Autostacks a cone once picked up.
- double **getLifterHeight** ()
Gets the height of the lifter in inches.
- int **getLifterTicks** ()
Gets the value of the lifter pot.
- void **interrupt_auto_stack** (void *param)
Stops an autostack in case of an error.
- float **lifterPotentiometerToDegree** (int x)
height of the lifter in degrees from 0 height
- void **lower_main_lifter** ()
Lowers the main lifter.
- void **lower_secondary_lifter** ()
Lowers the secondary lifter.
- void **raise_main_lifter** ()
Raises the main lifter.
- void **raise_secondary_lifter** ()
Raises the main lifter.
- void **set_lifter_pos** (int pos)
Sets the lifter positions to the given value.
- void **set_main_lifter_motors** (const int v)
Sets the main lifter motors to the given value.
- void **set_secondary_lifter_motors** (const int v)
Sets the secondary lifter motors to the given value.
- void **update_lifter** ()
Updates the lifter in teleop.

6.19.1 Detailed Description

Declarations and macros for controlling and manipulating the lifter.

Author

Chris Jerrett, Christian Desimone

Date

8/27/2017

Definition in file **lifter.h**.

6.19.2 Function Documentation

6.19.2.1 autostack_routine()

```
void autostack_routine (
    void * param )
```

Autostacks a cone once picked up.

Parameters

<i>param</i>	ignored parameter
--------------	-------------------

Definition at line 20 of file **lifter.c**.

References `analogRead()`, `delay()`, `info()`, `lifter_autostack_routine_interrupt`, `lifter_autostack_running`, `lifter

- _ultrasonic`, `printf()`, `quit_auto_static()`, `raise_secondary_lifter()`, `set_claw_motor()`, `set_main_lifter_motors()`, `set_secondary_lifter_motors()`, and `ultrasonicGet()`.

Referenced by `operatorControl()`.

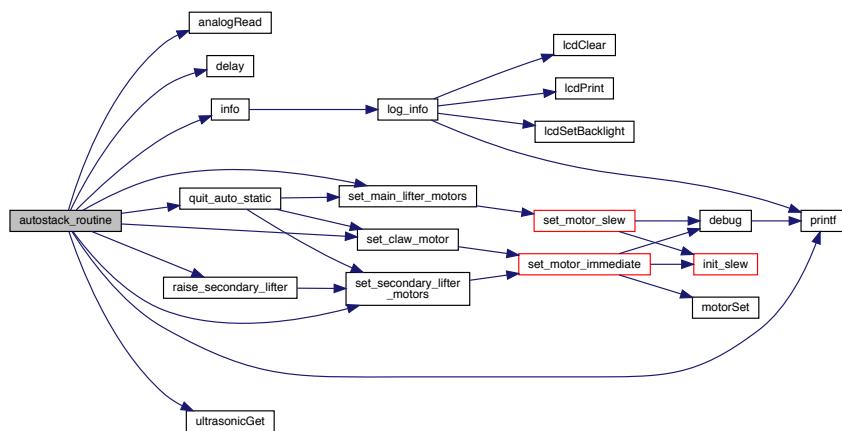
```
00020
00021     lifter_autostack_routine_interrupt = false;
00022     lifter_autostack_running = true;
00023     raise_secondary_lifter();
00024     while (analogRead(SECONDARY_LIFTER_POT_PORT) < 1600) {
00025         set_secondary_lifter_motors(MIN_SPEED);
00026         if (lifter_autostack_routine_interrupt) {
00027             quit_auto_static();
00028             return;
00029         }
00030         delay(50);
00031         info("1");
00032     }
00033     set_secondary_lifter_motors(0);
00034     bool lifted = false;
00035     int val = ultrasonicGet(lifter_ultrasonic);
00036     printf("%d\n", val);
00037     while (val < 10 && val != ULTRA_BAD_RESPONSE) {
00038         if (lifter_autostack_routine_interrupt) {
00039             quit_auto_static();
00040             return;
00041         }
00042         set_main_lifter_motors(MAX_SPEED);
00043         info("2");
00044         lifted = true;
00045         delay(50);
00046         val = ultrasonicGet(lifter_ultrasonic);
00047         printf("%d\n", val);
00048     }
00049     if (lifter_autostack_routine_interrupt) {
00050         quit_auto_static();
00051         return;
00052     }
00053     delay(200);
00054     if (lifted)
00055         delay(50);
00056     if (lifter_autostack_routine_interrupt) {
00057         quit_auto_static();
00058         return;
00059     }
00060     set_main_lifter_motors(0);
00061     set_secondary_lifter_motors(0);
```

```

00062
00063     while (analogRead(SECONDARY_LIFTER_POT_PORT) < 3000) {
00064         if (lifter_autostack_routine_interrupt) {
00065             quit_auto_static();
00066             return;
00067         }
00068         set_secondary_lifter_motors(MIN_SPEED);
00069         delay(50);
00070         info("3");
00071     }
00072
00073     set_main_lifter_motors(MIN_SPEED / 1.333);
00074
00075     while (val > 10) {
00076         if (lifter_autostack_routine_interrupt) {
00077             quit_auto_static();
00078             return;
00079         }
00080         info("2");
00081         lifted = true;
00082         delay(30);
00083         val = ultrasonicGet(lifter_ultrasonic);
00084         printf("%d\n", val);
00085     }
00086
00087     set_main_lifter_motors(0);
00088
00089     set_claw_motor(MIN_CLAW_SPEED);
00090     if (lifter_autostack_routine_interrupt) {
00091         quit_auto_static();
00092         return;
00093     }
00094     delay(500);
00095     if (lifter_autostack_routine_interrupt) {
00096         quit_auto_static();
00097         return;
00098     }
00099     set_main_lifter_motors(MAX_SPEED);
00100    if (lifter_autostack_routine_interrupt) {
00101        quit_auto_static();
00102        return;
00103    }
00104    delay(300);
00105
00106    set_main_lifter_motors(MIN_SPEED);
00107    set_claw_motor(0);
00108    set_secondary_lifter_motors(0);
00109
00110    lifter_autostack_running = false;
00111 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.19.2.2 getLifterHeight()

```
double getLifterHeight ( )
```

Gets the height of the lifter in inches.

Returns

the height of the lifter.

Author

Chris Jerrett

Date

9/17/2017

Definition at line **306** of file **lifter.c**.

References **getLifterTicks()**.

```
00306     {
00307     unsigned int ticks = getLifterTicks();
00308     return (-2 * pow(10, (-9 * ticks)) + 6 * (pow(10, (-6 * ticks * ticks))) +
00309             0.0198 * ticks + 2.3033);
00310 }
```

Here is the call graph for this function:



6.19.2.3 getLifterTicks()

```
int getLifterTicks ( )
```

Gets the value of the lifter pot.

Returns

the value of the pot.

Author

Chris Jerrett

Date

9/9/2017

Definition at line **297** of file **lifter.c**.

References **analogRead()**.

Referenced by **getLifterHeight()**.

```
00297 { return analogRead(LIFTER); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.19.2.4 interrupt_auto_stack()

```
void interrupt_auto_stack (
    void * param )
```

Stops an autostack in case of an error.

Parameters

<i>param</i>	ignore parameter
--------------	------------------

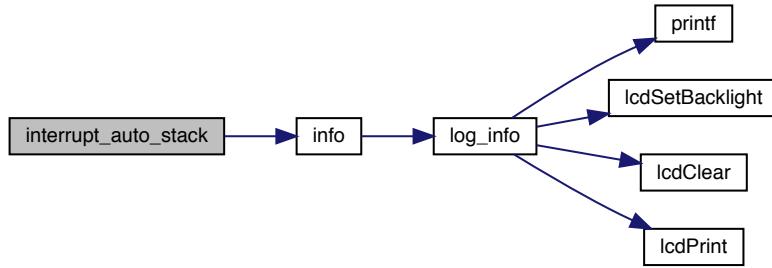
Definition at line **8** of file **lifter.c**.

References **info()**, and **lifter_autostack_routine_interrupt**.

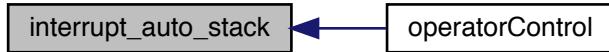
Referenced by **operatorControl()**.

```
00008
00009     info("int");
00010     lifter_autostack_routine_interrupt = true;
00011 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.19.2.5 lifterPotentiometerToDegree()

```
float lifterPotentiometerToDegree (
    int x )
```

height of the lifter in degrees from 0 height

Parameters

x	the pot value
---	---------------

Returns

the positions in degrees

Author

Chris Jerrett

Date

10/13/2017

Definition at line **286** of file **lifter.c**.

```
00286     {
00287     return (x - INIT_ROTATION) / TICK_MAX * DEG_MAX;
00288 }
```

6.19.2.6 lower_main_lifter()

```
void lower_main_lifter ( )
```

Lowers the main lifter.

Author

Christian DeSimone

Date

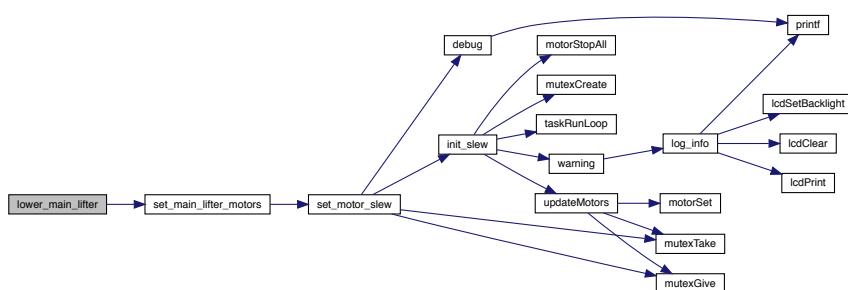
9/12/2017

Definition at line **160** of file **lifter.c**.

References **set_main_lifter_motors()**.

```
00160 { set_main_lifter_motors(MAX_SPEED); }
```

Here is the call graph for this function:



6.19.2.7 lower_secondary_lifter()

```
void lower_secondary_lifter ( )
```

Lowers the secondary lifter.

Author

Christian DeSimone

Date

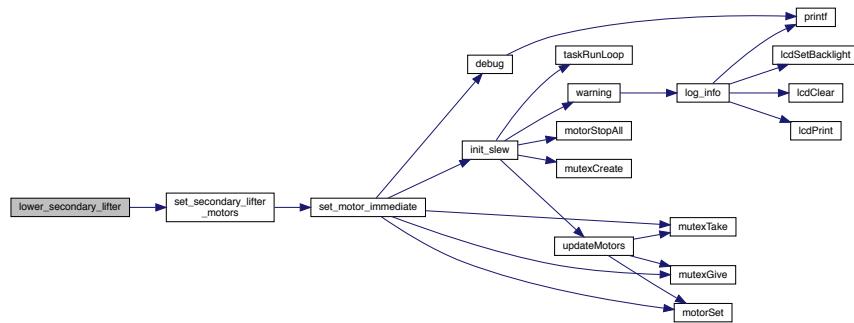
9/12/2017

Definition at line **176** of file **lifter.c**.

References **set_secondary_lifter_motors()**.

```
00176 { set_secondary_lifter_motors(MAX_SPEED); }
```

Here is the call graph for this function:



6.19.2.8 raise_main_lifter()

```
void raise_main_lifter ( )
```

Raises the main lifter.

Author

Christian DeSimone

Date

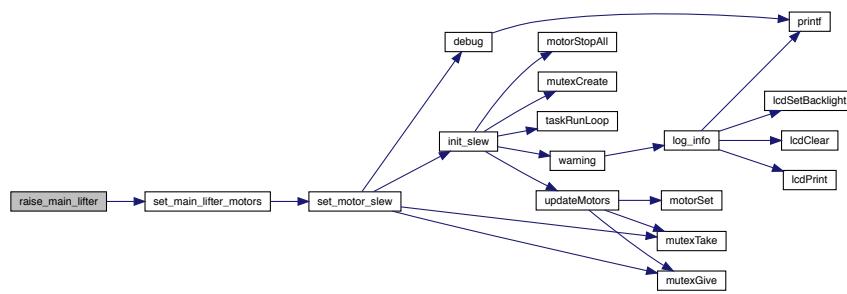
9/12/2017

Definition at line **152** of file **lifter.c**.

References **set_main_lifter_motors()**.

```
00152 { set_main_lifter_motors(MAX_SPEED); }
```

Here is the call graph for this function:

**6.19.2.9 raise_secondary_lifter()**

```
void raise_secondary_lifter( )
```

Raises the main lifter.

Author

Christian DeSimone

Date

9/12/2017

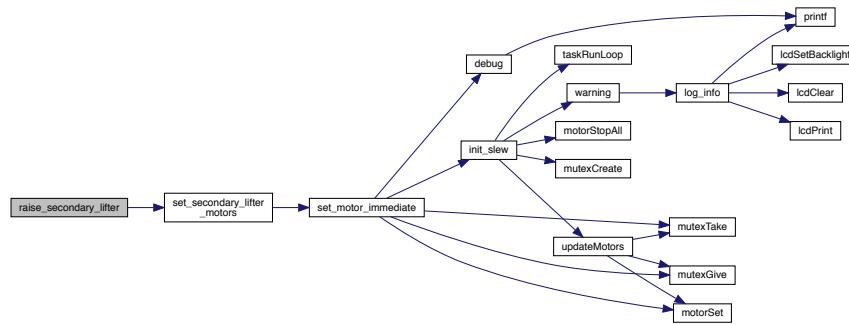
Definition at line **168** of file **lifter.c**.

References **set_secondary_lifter_motors()**.

Referenced by **autostack_routine()**.

```
00168 { set_secondary_lifter_motors(MIN_SPEED / 1.5); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.19.2.10 set_lifter_pos()

```
void set_lifter_pos (
    int pos )
```

Sets the lifter positions to the given value.

Parameters

<i>pos</i>	The height in inches
------------	----------------------

Author

Chris Jerrett

Date

9/12/2017

Definition at line 144 of file **lifter.c**.

```
00144 { }
```

6.19.2.11 set_main_lifter_motors()

```
void set_main_lifter_motors (
    const int v )
```

Sets the main lifter motors to the given value.

Parameters

v	value for the lifter motor. Between -128 - 127, any values outside are clamped.
---	---

Author

Chris Jerrett

Date

9/9/2017

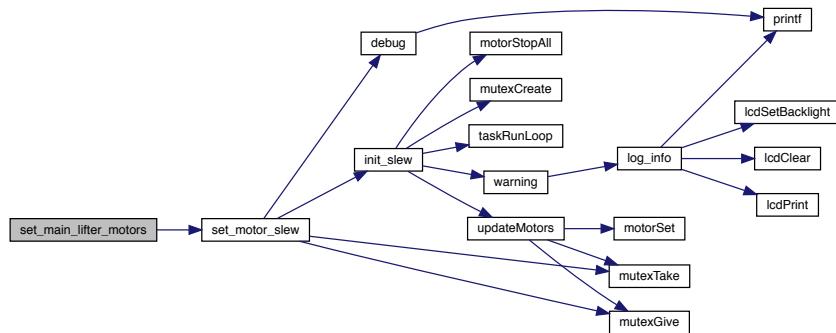
Definition at line **133** of file **lifter.c**.

References **set_motor_slew()**.

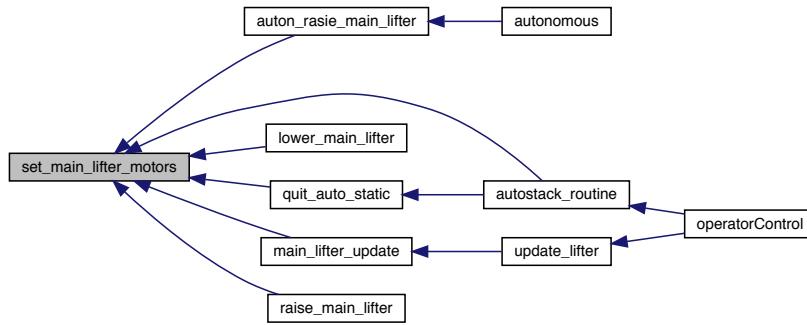
Referenced by **auton_rasie_main_lifter()**, **autostack_routine()**, **lower_main_lifter()**, **main_lifter_update()**, **quit_auto_static()**, and **raise_main_lifter()**.

```
00133     {
00134     set_motor_slew(MOTOR_MAIN_LIFTER, v);
00135 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.19.2.12 set_secondary_lifter_motors()

```
void set_secondary_lifter_motors (
    const int v )
```

Sets the secondary lifter motors to the given value.

Parameters

<code>v</code>	value for the lifter motor. Between -128 - 127, any values outside are clamped.
----------------	---

Author

Chris Jerrett

Date

1/6/2018

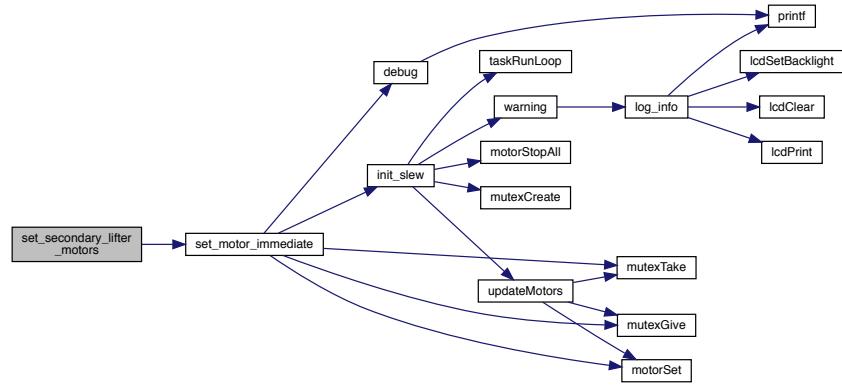
Definition at line 121 of file **lifter.c**.

References `set_motor_immediate()`.

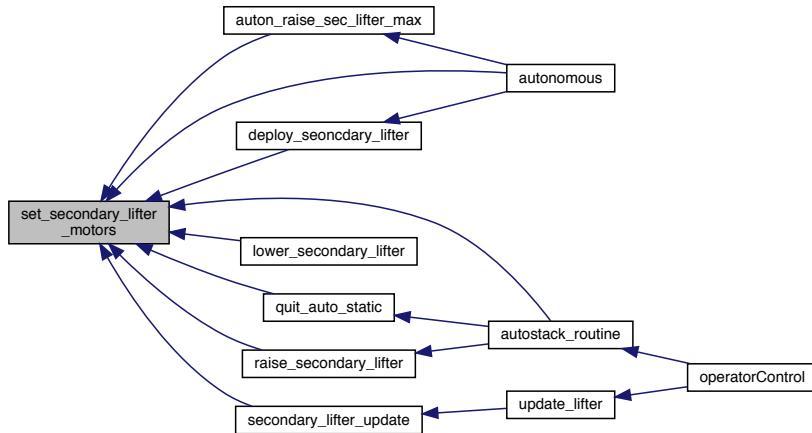
Referenced by `auton_raise_sec_lifter_max()`, `autonomous()`, `autostack_routine()`, `deploy_seoncdary_lifter()`, `lower_secondary_lifter()`, `quit_auto_static()`, `raise_secondary_lifter()`, and `secondary_lifter_update()`.

```
00121
00122     set_motor_immediate(MOTOR_SECONDARY_LIFTER, v);
00123 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.19.2.13 update_lifter()

```
void update_lifter( )
```

Updates the lifter in teleop.

Author

Chris Jerrett

Date

9/9/2017

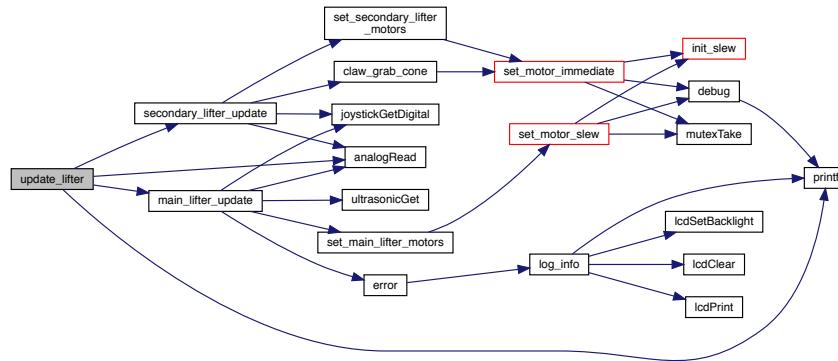
Definition at line 272 of file **lifter.c**.

References **analogRead()**, **main_lifter_update()**, **printf()**, **secondary_lifter_update()**, and **secondary_override**.

Referenced by **operatorControl()**.

```
00272     {
00273     printf("%d \n", analogRead(SECONDARY_LIFTER_POT_PORT));
00274     main_lifter_update();
00275     if (!secondary_override)
00276         secondary_lifter_update();
00277 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.20 lifter.h

```

00001
00007 #ifndef _LIFTER_H_
00008 #define _LIFTER_H_
00009
00010 #include "controller.h"
00011 #include "drive.h"
00012 #include "motor_ports.h"
00013 #include "potentiometer.h"
00014 #include "sensors.h"
00015 #include "slew.h"
00016 #include <API.h>
00017
00021 #define INIT_ROTATION 680
00022
00026 #define SECONDARY_LIFTER_P .05
00027
00031 #define SECONDARY_LIFTER_D 0
00032
00036 #define SECONDARY_LIFTER_I 0.000
00037
00041 #define MAIN_LIFTER_P 0
00042
00046 #define MAIN_LIFTER_D 0
00047
00051 #define MAIN_LIFTER_I 0.0000001
00052
00056 #define THRESHOLD 10
00057
00061 #define HEIGHT 19.1 - 3.8
00062
00066 #define LIFTER_UP PARTNER, 6, JOY_UP
00067
00071 #define LIFTER_DOWN PARTNER, 6, JOY_DOWN
00072
00076 #define SECONDARY_LIFTER_UP PARTNER, 5, JOY_UP
00077
00081 #define SECONDARY_LIFTER_DOWN PARTNER, 5, JOY_DOWN
00082
00086 #define LIFTER_DRIVER_LOAD MASTER, RIGHT_BUTTONS, JOY_RIGHT
00087
00091 #define LIFTER_UP_PARTNER PARTNER, 5, JOY_UP
00092
00096 #define LIFTER_DOWN_PARTNER PARTNER, 5, JOY_DOWN
00097
00101 #define SECONDARY_LIFTER_POT_PORT 2
00102
00106 #define SECONDARY_LIFTER_MAX_HEIGHT 3120
00107
00111 #define SECONDARY_LIFTER_MIN_HEIGHT 2000
00112
00116 #define MAIN_LIFTER_POT 1
00117
00121 #define MAIN_LIFTER_MIN_HEIGHT 1700
00122
00131 void set_secondary_lifter_motors(const int v);
00132
00141 void set_main_lifter_motors(const int v);
00142
00150 void set_lifter_pos(int pos);
00151
00158 void raise_main_lifter();
00159
00166 void lower_main_lifter();
00167
00174 void raise_secondary_lifter();
00175
00182 void lower_secondary_lifter();
00183
00190 void update_lifter();
00191
00200 float lifterPotentiometerToDegree(int x);
00201
00209 int getLifterTicks();
00210
00218 double getLifterHeight();
00219
00224 void autostack_routine(void *param);
00225

```

```
00230 void interrupt_auto_stack(void *param);  
00231  
00232 #endif
```

6.21 include/list.h File Reference

A doubly linked list implementation.

Data Structures

- struct **list_iterator_t**
A iterator representation Allows automatic iteration through linked list.
- struct **list_node**
A node in a list.
- struct **list_t**
A struct representing a linked list.

Typedefs

- typedef struct **list_node** **list_node_t**
A node in a list.

Enumerations

- enum **list_direction_t** { **LIST_HEAD**, **LIST_TAIL** }
list_t (p. 13) iterator direction.

Functions

- **list_node_t * list_at (list_t *self, int index)**
Finds a node a given index.
- **void list_destroy (list_t *self)**
Deallocates a list.
- **list_node_t * list_find (list_t *self, void *val)**
Finds a node in a list with a given value.
- **void list_iterator_destroy (list_iterator_t *self)**
Destroys the iterator.
- **list_iterator_t * list_iterator_new (list_t *list, list_direction_t direction)**
Creates a new iterator.
- **list_iterator_t * list_iterator_new_from_node (list_node_t *node, list_direction_t direction)**
Creates a new iterator by using the node to start at.
- **list_node_t * list_iterator_next (list_iterator_t *self)**
The next node in the iterator and advances the iterator.
- **list_node_t * list_lpop (list_t *self)**

- **list_node_t * list_lpush (list_t *self, list_node_t *node)**
Pushed a node to the start of a list.
- **list_t * list_new ()**
Allocated a new list.
- **list_node_t * list_node_new (void *val)**
Allocates a new node.
- **void list_remove (list_t *self, list_node_t *node)**
removes and returns the a given node from the list
- **list_node_t * list_rpop (list_t *self)**
removes and returns the end node
- **list_node_t * list_rpush (list_t *self, list_node_t *node)**
Pushed a node to the end of a list.

6.21.1 Detailed Description

A doubly linked list implementation.

A linked list is a linear data structure where each element is a separate object.

Each element (we will call it a node) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference. A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

A Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.

Author

Chris Jerrett

Date

1/3/18

Definition in file **list.h**.

6.21.2 Typedef Documentation

6.21.2.1 list_node_t

```
typedef struct list_node list_node_t
```

A node in a list.

Author

Chris Jerrett

Date

1/3/18

6.21.3 Enumeration Type Documentation

6.21.3.1 list_direction_t

```
enum list_direction_t
```

list_t (p. 13) iterator direction.

Tells a iterator what direction to traverse the linked list in.

Author

Chris Jerrett

Date

1/3/18

Enumerator

LIST_HEAD	start at head	start at tail
LIST_TAIL		

Definition at line 36 of file **list.h**.

```
00036          {
00040      LIST_HEAD
00044      ,
00045      LIST_TAIL
00046 } list_direction_t;
```

6.21.4 Function Documentation

6.21.4.1 list_at()

```
list_node_t* list_at (
    list_t * self,
    int index )
```

Finds a node a given index.

Parameters

<i>self</i>	the list
<i>index</i>	the index

Returns

the node

Author

Chris Jerrett

Date

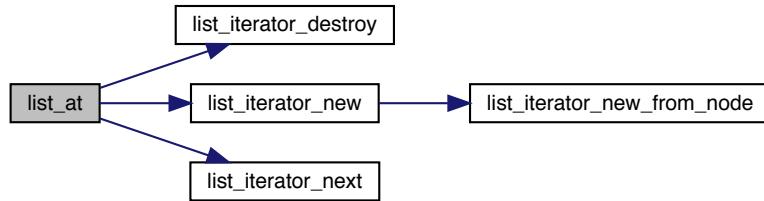
1/3/18

Definition at line **202** of file **list.c**.

References **LIST_HEAD**, **list_iterator_destroy()**, **list_iterator_new()**, **list_iterator_next()**, and **LIST_TAIL**.

```
00202
00203     list_direction_t direction = LIST_HEAD;
00204
00205     if (index < 0) {
00206         direction = LIST_TAIL;
00207         index = ~index;
00208     }
00209
00210     if ((unsigned)index < self->len) {
00211         list_iterator_t *it = list_iterator_new(self, direction);
00212         list_node_t *node = list_iterator_next(it);
00213         while (index--)
00214             node = list_iterator_next(it);
00215         list_iterator_destroy(it);
00216         return node;
00217     }
00218
00219     return NULL;
00220 }
```

Here is the call graph for this function:



6.21.4.2 list_destroy()

```
void list_destroy (
    list_t * self )
```

Deallocates a list.

Parameters

self	the list
------	----------

Author

Chris Jerrett

Date

1/3/18

Definition at line **50** of file **list.c**.

References **list_node::next**, and **list_node::val**.

Referenced by **deinit_routines()**.

```
00050
00051     unsigned int len = self->len;
00052     list_node_t *next;
00053     list_node_t *curr = self->head;
00054
00055     while (len--) {
00056         next = curr->next;
00057         if (self->free)
00058             self->free(curr->val);
00059         free(curr);
00060         curr = next;
00061     }
00062
00063     free(self);
00064 }
```

Here is the caller graph for this function:



6.21.4.3 list_find()

```
list_node_t* list_find (
    list_t * self,
    void * val )
```

Finds a node in a list with a given value.

Parameters

<i>self</i>	the list
<i>val</i>	the value

Returns

the node

Author

Chris Jerrett

Date

1/3/18

Definition at line 172 of file **list.c**.

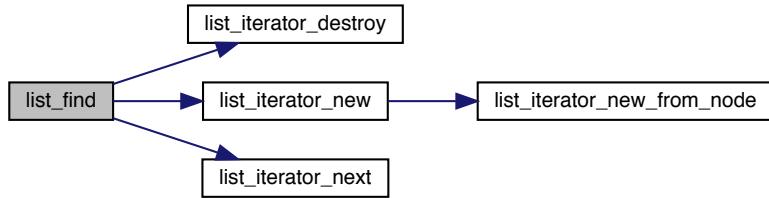
References **LIST_HEAD**, **list_iterator_destroy()**, **list_iterator_new()**, **list_iterator_next()**, and **list_node::val**.

```

00172     {
00173         list_iterator_t *it = list_iterator_new(self, LIST_HEAD);
00174         list_node_t *node;
00175
00176         while ((node = list_iterator_next(it))) {
00177             if (self->match) {
00178                 if (self->match(val, node->val)) {
00179                     list_iterator_destroy(it);
  
```

```
00180         return node;
00181     }
00182 } else {
00183     if (val == node->val) {
00184         list_iterator_destroy(it);
00185         return node;
00186     }
00187 }
00188 }
00189 list_iterator_destroy(it);
00190 return NULL;
00191 }
```

Here is the call graph for this function:



6.21.4.4 list_iterator_destroy()

```
void list_iterator_destroy (
    list_iterator_t * self )
```

Destroys the iterator.

Parameters

<code>self</code>	the iterator
-------------------	--------------

Author

Chris Jerrett

Date

1/3/17

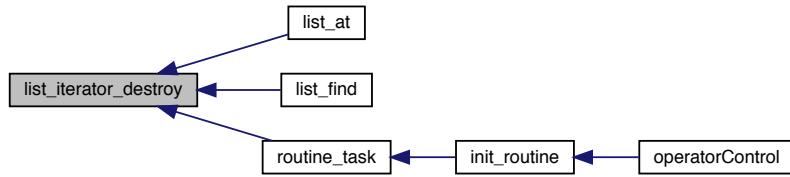
Definition at line **56** of file **list_iterator.c**.

Referenced by **list_at()**, **list_find()**, and **routine_task()**.

```

00056
00057     free(self);
00058     self = NULL;
00059 }
```

Here is the caller graph for this function:



6.21.4.5 list_iterator_new()

```
list_iterator_t* list_iterator_new (
    list_t * list,
    list_direction_t direction )
```

Creates a new iterator.

Parameters

<i>list</i>	the list
<i>direction</i>	direction the iterator should progress in

Returns

the iterator created

Author

Chris Jerrett

Date

1/3/18

Definition at line 11 of file **list_iterator.c**.

References **list_t::head**, **LIST_HEAD**, **list_iterator_new_from_node()**, and **list_t::tail**.

Referenced by **list_at()**, **list_find()**, and **routine_task()**.

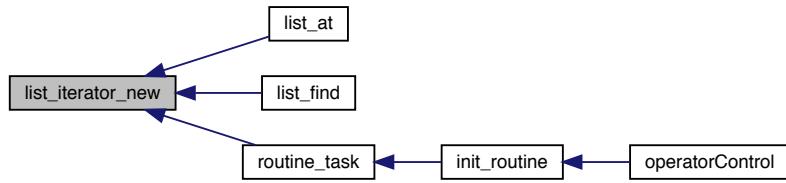
```

00011
00012     list_node_t *node = direction == LIST_HEAD ? list->head : list->tail;
00013     return list_iterator_new_from_node(node, direction);
00014 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.21.4.6 list_iterator_new_from_node()

```
list_iterator_t* list_iterator_new_from_node (
    list_node_t * node,
    list_direction_t direction )
```

Creates a new iterator by using the node to start at.

Parameters

<i>node</i>	the start node
<i>direction</i>	direction the iterator should progress in

Returns

the iterator created

Author

Chris Jerrett

Date

1/3/18

Definition at line **24** of file **list_iterator.c**.

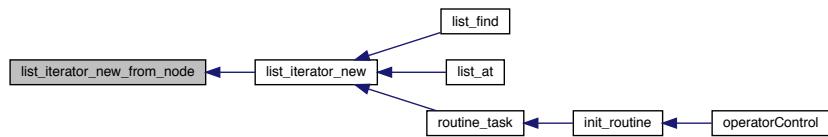
References **list_iterator_t::next**.

Referenced by **list_iterator_new()**.

```

00025
00026     list_iterator_t *self;
00027     if (!(self = (list_iterator_t *)malloc(sizeof(list_iterator_t))))
00028         return NULL;
00029     self->next = node;
00030     self->direction = direction;
00031     return self;
00032 }
```

Here is the caller graph for this function:



6.21.4.7 `list_iterator_next()`

```

list_node_t* list_iterator_next (
    list_iterator_t * self )
  
```

The next node in the iterator and advances the iterator.

Returns NULL when done.

Parameters

<code>self</code>	the iterator
-------------------	--------------

Returns

the next node.

Author

Chris Jerrett

Date

1/3/17

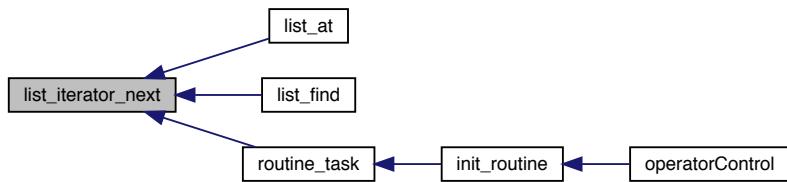
Definition at line **42** of file **list_iterator.c**.

References **LIST_HEAD**, **list_node::next**, and **list_node::prev**.

Referenced by **list_at()**, **list_find()**, and **routine_task()**.

```
00042     {  
00043     list_node_t *curr = self->next;  
00044     if (curr) {  
00045         self->next = self->direction == LIST_HEAD ? curr->next : curr->prev;  
00046     }  
00047     return curr;  
00048 }
```

Here is the caller graph for this function:



6.21.4.8 list_lpop()

```
list_node_t* list_lpop (  
    list_t * self )
```

removes and returns the start node

Parameters

<i>self</i>	the list
-------------	----------

Returns

the node removed

Author

Chris Jerrett

Date

1/3/18

Definition at line **122** of file **list.c**.

References **list_node::next**, and **list_node::prev**.

```

00122                                     {
00123     if (!self->len)
00124         return NULL;
00125
00126     list_node_t *node = self->head;
00127
00128     if (--self->len) {
00129         (self->head = node->next)->prev = NULL;
00130     } else {
00131         self->head = self->tail = NULL;
00132     }
00133
00134     node->next = node->prev = NULL;
00135     return node;
00136 }
```

6.21.4.9 list_lpush()

```
list_node_t* list_lpush (
    list_t * self,
    list_node_t * node )
```

Pushed a node to the start of a list.

Parameters

<i>self</i>	the list
<i>node</i>	the node

Returns

the node added

Author

Chris Jerrett

Date

1/3/18

Definition at line 146 of file **list.c**.

References **list_node::next**, and **list_node::prev**.

```
00146      {  
00147      if (!node)  
00148          return NULL;  
00149  
00150      if (self->len) {  
00151          node->next = self->head;  
00152          node->prev = NULL;  
00153          self->head->prev = node;  
00154          self->head = node;  
00155      } else {  
00156          self->head = self->tail = node;  
00157          node->prev = node->next = NULL;  
00158      }  
00159  
00160      ++self->len;  
00161      return node;  
00162 }
```

6.21.4.10 list_new()

```
list_t* list_new ( )
```

Allocated a new list.

Returns

the new list

Author

Chris Jerrett

Date

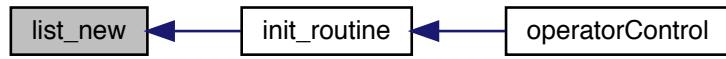
1/3/18

Definition at line 34 of file **list.c**.References **list_t::head**.Referenced by **init_routine()**.

```

00034     {
00035     list_t *self;
00036     if (!(self = (list_t *)malloc(sizeof(list_t))))
00037         return NULL;
00038     self->head = NULL;
00039     self->tail = NULL;
00040     self->free = NULL;
00041     self->match = NULL;
00042     self->len = 0;
00043     return self;
00044 }
```

Here is the caller graph for this function:

**6.21.4.11 list_node_new()**

```
list_node_t* list_node_new (
    void * val )
```

Allocates a new node.

Parameters

<i>val</i>	The value the node contains.
------------	------------------------------

Returns

The newly allocated node.

Author

Chris Jerrett

Date

1/3/18 Node must be freed

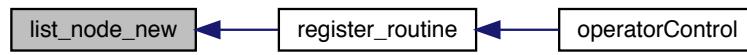
Definition at line **12** of file **list_node.c**.

References **list_node::next**, **list_node::prev**, and **list_node::val**.

Referenced by **register_routine()**.

```
00012     {
00013     list_node_t *self;
00014     if (! (self = (list_node_t *)malloc(sizeof(list_node_t))))
00015         return NULL;
00016     self->prev = NULL;
00017     self->next = NULL;
00018     self->val = val;
00019     return self;
00020 }
```

Here is the caller graph for this function:



6.21.4.12 list_remove()

```
void list_remove (
    list_t * self,
    list_node_t * node )
```

removes and returns the a given node from the list

Parameters

<i>self</i>	the list
-------------	----------

Returns

the node removed

Author

Chris Jerrett

Date

1/3/18

Definition at line **229** of file **list.c**.

References **list_node::next**, **list_node::prev**, and **list_node::val**.

```
00229     {  
00230     node->prev ? (node->prev->next = node->next) : (self->head = node->next);  
00231     node->next ? (node->next->prev = node->prev) : (self->tail = node->prev);  
00232     if (self->free)  
00233         self->free(node->val);  
00234     free(node);  
00235     --self->len;  
00236 }  
00237  
00238 }
```

6.21.4.13 list_rpop()

```
list_node_t* list_rpop (  
    list_t * self )
```

removes and returns the end node

Parameters

<i>self</i>	the list
-------------	----------

Returns

the node removed

Author

Chris Jerrett

Date

1/3/18

Definition at line 99 of file **list.c**.References **list_node::next**, and **list_node::prev**.

```
00099      {
00100  if (!self->len)
00101      return NULL;
00102
00103  list_node_t *node = self->tail;
00104
00105  if (--self->len) {
00106      (self->tail = node->prev)->next = NULL;
00107  } else {
00108      self->tail = self->head = NULL;
00109  }
00110
00111  node->next = node->prev = NULL;
00112  return node;
00113 }
```

6.21.4.14 list_rpush()

```
list_node_t* list_rpush (
    list_t * self,
    list_node_t * node )
```

Pushed a node to the end of a list.

Parameters

<i>self</i>	the list
<i>node</i>	the node

Returns

the node added

Author

Chris Jerrett

Date

1/3/18

Definition at line 74 of file **list.c**.References **list_node::next**, and **list_node::prev**.Referenced by **register_routine()**.

```

00074     if (!node)
00075         return NULL;
00077
00078     if (self->len) {
00079         node->prev = self->tail;
00080         node->next = NULL;
00081         self->tail->next = node;
00082         self->tail = node;
00083     } else {
00084         self->head = self->tail = node;
00085         node->prev = node->next = NULL;
00086     }
00087
00088     ++self->len;
00089     return node;
00090 }
```

Here is the caller graph for this function:



6.22 list.h

```

00001
00025 #ifndef LIST_H
00026 #define LIST_H
00027
00028 #include <API.h>
00029
00036 typedef enum {
00040     LIST_HEAD
00044     ,
00045     LIST_TAIL
00046 } list_direction_t;
00047
00053 typedef struct list_node {
00054     struct list_node *prev;
00055     struct list_node *next;
00056     void *val;
00057 } list_node_t;
00058
00064 typedef struct {
00065     // start of list
00066     list_node_t *head;
00067     // end of list
00068     list_node_t *tail;
00069     // length of list
00070     unsigned int len;
00071
00072     // the value
00073     void (*free)(void *val);
00074
00075     // a compare function
00076     int (*match)(void *a, void *b);
00077 } list_t;
00078
00086 typedef struct {
00087     list_node_t *next;
00088     list_direction_t direction;
00089 } list_iterator_t;
00090
```

```
00099 list_node_t *list_node_new(void *val);
00100
00107 list_t *list_new();
00108
00117 list_node_t *list_rpush(list_t *self, list_node_t *node);
00118
00127 list_node_t *list_lpush(list_t *self, list_node_t *node);
00128
00137 list_node_t *list_find(list_t *self, void *val);
00138
00147 list_node_t *list_at(list_t *self, int index);
00148
00156 list_node_t *list_rpop(list_t *self);
00157
00165 list_node_t *list_lpop(list_t *self);
00166
00174 void list_remove(list_t *self, list_node_t *node);
00175
00182 void list_destroy(list_t *self);
00183
00192 list_iterator_t *list_iterator_new(list_t *list, list_direction_t direction);
00193
00202 list_iterator_t *list_iterator_new_from_node(list_node_t *node,
00203                                     list_direction_t direction);
00204
00213 list_node_t *list_iterator_next(list_iterator_t *self);
00214
00221 void list_iterator_destroy(list_iterator_t *self);
00222
00223 #endif
```

6.23 include/localization.h File Reference

Declarations and macros for determining the location of the robot. [WIP].

Data Structures

- struct **location**

Vector storing the cartesian cords and an angle.

Functions

- int **calculate_encoder_angle ()**
Calculates the angle using the encoders.
- struct **location get_position ()**
Gets the current position of the robot.
- bool **init_localization** (const unsigned char gyro1, unsigned short multiplier, int start_x, int start_y, int start_theta)
Starts the localization process.
- void **update_position ()**
Updates the position from the localization.

6.23.1 Detailed Description

Declarations and macros for determining the location of the robot. [WIP].

Author

Chris Jerrett, Christian Desimone

Date

9/27/2017

Definition in file **localization.h**.

6.23.2 Function Documentation

6.23.2.1 calculate_encoder_angle()

`int calculate_encoder_angle ()`

Calculates the angle using the encoders.

Returns

the angle

Calculates the angle using the encoders.

Returns

the angle of rotation

Author

Chris Jerrett, Christian DeSimone

Definition at line 129 of file **localization.c**.

References **get_encoder_ticks()**.

Referenced by **auton_turn_180()**.

```
00129
00130 #define WIDTH 13.5
00131 #define CPR 392.0
00132 #define WHEEL_RADIUS 2
00133     int dist_r = get_encoder_ticks(0) / CPR;
00134     int dist_l = get_encoder_ticks(1) / CPR;
00135     return ((dist_r - dist_l) / WIDTH);
00136 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.23.2.2 get_position()

```
struct location get_position ( )
```

Gets the current position of the robot.

Parameters

<code>gyro1</code>	The first gyro
--------------------	----------------

Returns

the location of the robot as a struct.

Author

Chris Jerrett

Parameters

<code>gyro1</code>	The first gyro
--------------------	----------------

Returns

the location of the robot as a struct.

Definition at line **38** of file **localization.c**.

```
00038 { }
```

6.23.2.3 init_localization()

```
bool init_localization (
    const unsigned char gyrol,
    unsigned short multiplier,
    int start_x,
    int start_y,
    int start_theta )
```

Starts the localization process.

Author

Chris Jerrett

Parameters

gyro1	The first gyro multiplier parameter can tune the gyro to adapt to specific sensors. The default value at this time is 196; higher values will increase the number of degrees reported for a fixed actual rotation, while lower values will decrease the number of degrees reported. If your robot is consistently turning too far, increase the multiplier, and if it is not turning far enough, decrease the multiplier.
--------------	---

Starts the localization process.

Author

Chris Jerrett

Returns

returns true when initialization is complete.

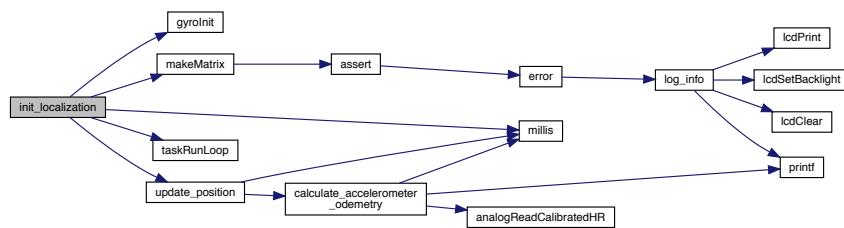
Definition at line **160** of file **localization.c**.

References **g1**, **gyroInit()**, **last_call**, **localization_task**, **makeMatrix()**, **millis()**, **taskRunLoop()**, and **update_position()**.

```

00161     g1 = gyroInit(gyro1, multiplier);
00162     // init state matrix
00163
00164     // one dimensional vector with x, y, theta, acceleration in x and y
00165     state_matrix = makeMatrix(1, 5);
00166     localization_task =
00167         taskRunLoop(update_position, LOCALIZATION_UPDATE_FREQUENCY * 1000);
00168     last_call = millis();
00169     return true;
00170 }
```

Here is the call graph for this function:



6.23.2.4 update_position()

```
void update_position ( )
```

Updates the position from the localization.

Author

Chris Jerrett

Chris Jerrett, Christian DeSimone

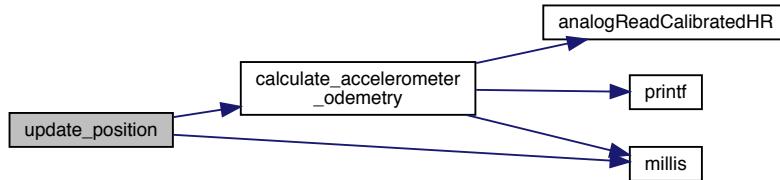
Definition at line 45 of file **localization.c**.

References **calculate_accelerometer_odometry()**, **last_call**, and **millis()**.

Referenced by **init_localization()**.

```
00045     {
00046     // int curr_theta = calculate_angle();
00047
00048     struct accelerometer_odometry oddem = calculate_accelerometer_odometry();
00049     // printf("x: %d y: %d T: %d\n", a.x, a.y, 0);
00050
00051     /*int l = 1;
00052     int vr = get_encoder_velocity(1);
00053     int vl = get_encoder_velocity(2);
00054     int theta_dot = (vr - vl) / l;
00055     int curr_theta = theta + theta_dot;
00056     double dt = LOCALIZATION_UPDATE_FREQUENCY;
00057     double v_tot = (vr+vl)/2.0;
00058     int x_curr = x - v_tot*dt*sin(curr_theta);
00059     int y_curr = y + v_tot*dt*cos(curr_theta);
00060     x = x_curr;
00061     y = y_curr; */
00062     last_call = millis();
00063 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.24 localization.h

```

00001
00008 #ifndef _LOCALIZATION_H_
00009 #define _LOCALIZATION_H_
00010
00011 #include "encoders.h"
00012 #include "matrix.h"
00013 #include <API.h>
00014 #include <math.h>
00015
00019 #define LOCALIZATION_UPDATE_FREQUENCY 5
00020
00024 struct location {
00025     int x;
00026     int y;
00027     int theta;
00028 };
00029
00042 bool init_localization(const unsigned char gyro1, unsigned short multiplier,
00043                           int start_x, int start_y, int start_theta);
00044
00051 struct location get_position();
00052
00058 void update_position();
00059
00064 int calculate_encoder_angle();
00065
00066 #endif

```

6.25 include/log.h File Reference

Contains logging functions.

Functions

- void **debug** (const char *debug_message)
prints a info message
- void **error** (const char *error_message)
prints a error message and displays on lcd.
- void **info** (const char *info_message)
prints a info message
- void **init_error** (bool use_lcd, FILE *lcd)
Initializes the error lcd system Only required if using lcd.
- void **warning** (const char *warning_message)
prints a warning message and displays on lcd.

6.25.1 Detailed Description

Contains logging functions.

Author

Chris Jerrett

Date

9/16/2017

Definition in file **log.h**.

6.25.2 Function Documentation

6.25.2.1 debug()

```
void debug (
    const char * debug_message )
```

prints a info message

Only will print and display if log_level is greater than info

See also

log_level (p. 279)

Parameters

<i>debug_message</i>	the message
----------------------	-------------

Definition at line **83** of file **log.c**.

References **log_level**, and **printf()**.

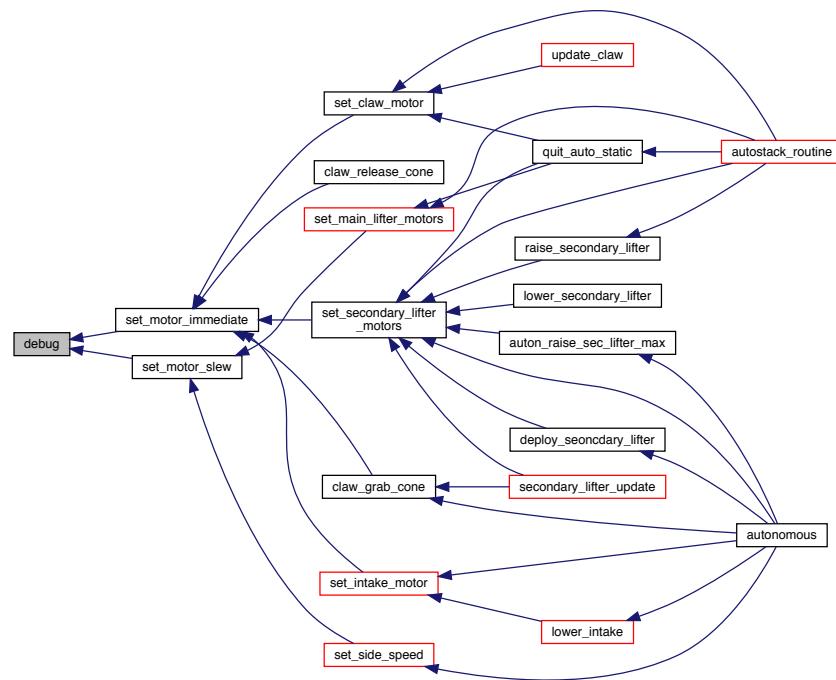
Referenced by **set_motor_immediate()**, and **set_motor_slew()**.

```
00083
00084     if (log_level > ERROR) {
00085         printf("[INFO]: %s\n", debug_message);
00086     }
00087 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.25.2.2 error()

```
void error ( const char * error_message )
```

prints a error message and displays on lcd.

Only will print and display if log_level is greater than NONE

See also

log_level (p. 279)

Author

Chris Jerrett

Date

9/10/17

Date _____

9/10/17

9/10/17

9/10/17

9/10/17

9/10/17

Parameters

<code>error_message</code>	the message
----------------------------	-------------

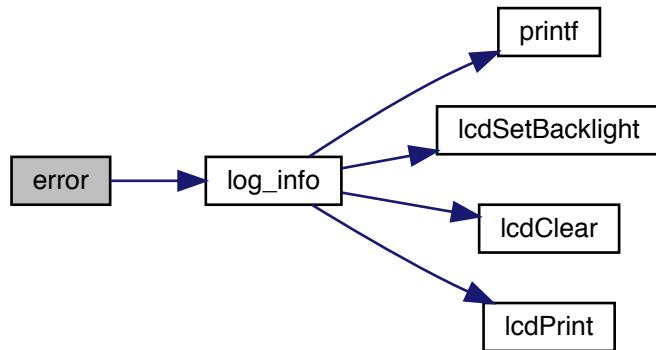
Definition at line **45** of file **log.c**.

References **log_info()**, and **log_level**.

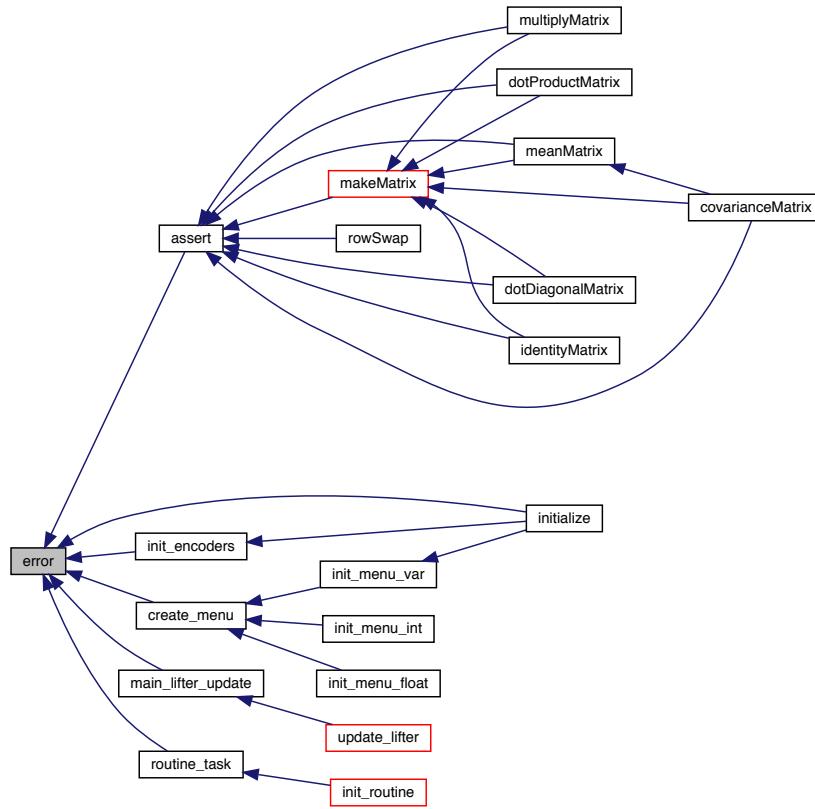
Referenced by **assert()**, **create_menu()**, **init_encoders()**, **initialize()**, **main_lifter_update()**, and **routine_task()**.

```
00045     {
00046     if (log_level > NONE)
00047         log_info("ERROR", error_message);
00048 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.25.2.3 info()

```
void info (
    const char * info_message )
```

prints a info message

Only will print and display if log_level is greater than ERROR

See also

[log_level](#) (p. 279)

Parameters

<i>info_message</i>	the message
---------------------	-------------

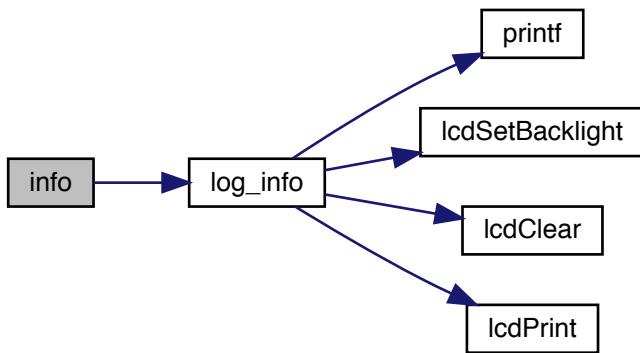
Definition at line 70 of file **log.c**.

References **log_info()**, and **log_level**.

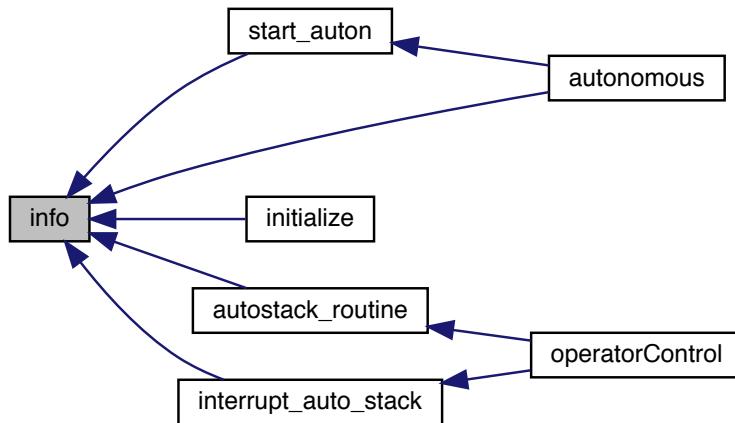
Referenced by **autonomous()**, **autostack_routine()**, **initialize()**, **interrupt_auto_stack()**, and **start_auton()**.

```
00070          {
00071      if (log_level > ERROR) {
00072          log_info("INFO", info_message);
00073      }
00074 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.25.2.4 init_error()

```
void init_error (
    bool use_lcd,
    FILE * lcd )
```

Initializes the error lcd system Only required if using lcd.

Author

Chris Jerrett

Date

9/10/17

Parameters

<i>use_lcd</i>	whether to use the lcd
<i>lcd</i>	the lcd

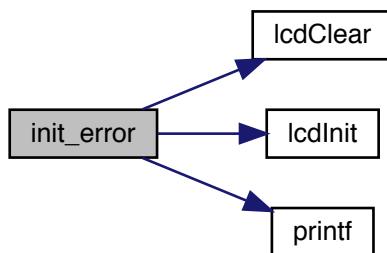
Definition at line 14 of file **log.c**.

References **LcdClear()**, **LcdInit()**, **log_lcd**, and **printf()**.

Referenced by **initialize()**.

```
00014
00015     if (use_lcd) {
00016         lcdInit(lcd);
00017         log_lcd = lcd;
00018         lcdClear(log_lcd);
00019         printf("LCD Init\n");
00020     }
00021 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.25.2.5 warning()

```
void warning (
    const char * warning_message )
```

prints a warning message and displays on lcd.

Only will print and display if log_level is greater than NONE

See also

[log_level](#) (p. 279)

Author

Chris Jerrett

Date

9/10/17

Parameters

<code>warning_message</code>	the message
------------------------------	-------------

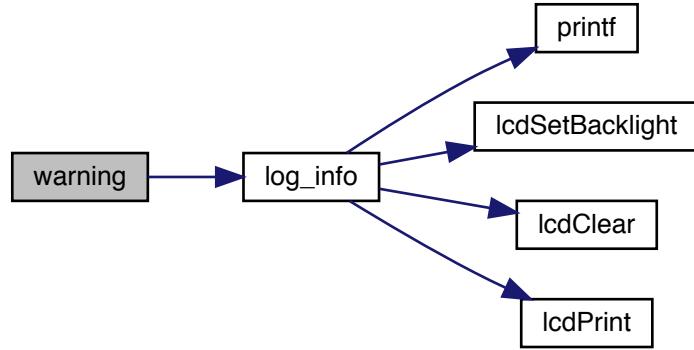
Definition at line **58** of file **log.c**.

References [log_info\(\)](#), and [log_level](#).

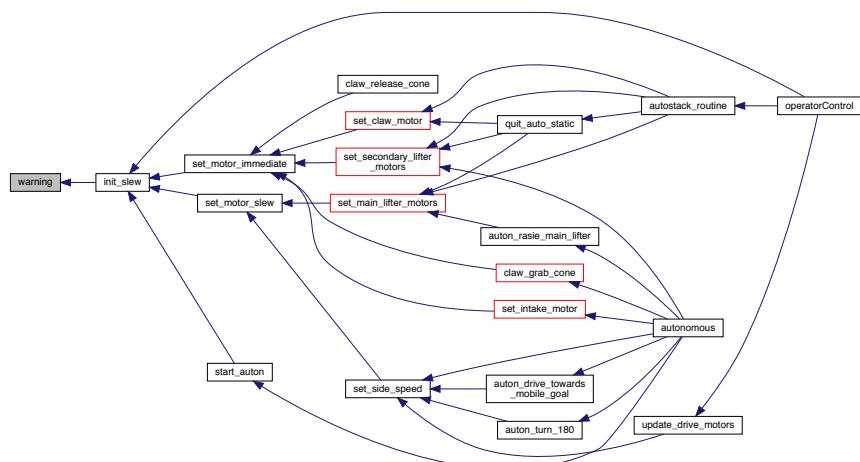
Referenced by [init_slew\(\)](#).

```
00058
00059     if (log_level > WARNING)
00060         log_info("WARNING", warning_message);
00061 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.26 log.h

```

00001
00007 #ifndef _LOG_H_
00008 #define _LOG_H_
00009
00010 #include "lcd.h"
00011 #include <API.h>
00012
00019 #define NONE 0
00020
00027 #define ERROR 1
  
```

```
00028
00035 #define WARNING 2
00036
00042 #define INFO 3
00043
00050 #define DEBUG 4
00051
00060 void init_error(bool use_lcd, FILE *lcd);
00061
00070 void error(const char *error_message);
00071
00080 void warning(const char *warning_message);
00081
00089 void info(const char *info_message);
00090
00098 void debug(const char *debug_message);
00099
00100 #endif
```

6.27 include/main.h File Reference

Header file for global functions.

Functions

- **void autonomous ()**
Runs the user autonomous code.
- **void initialize ()**
Runs user initialization code.
- **void initializeIO ()**
Runs pre-initialization code.
- **void operatorControl ()**
Runs the user operator control code.

6.27.1 Detailed Description

Header file for global functions.

Any experienced C or C++ programmer knows the importance of header files. For those who do not, a header file allows multiple files to reference functions in other files without necessarily having to see the code (and therefore causing a multiple definition). To make a function in "opcontrol.c", "auto.c", "main.c", or any other C file visible to the core implementation files, prototype it here.

This file is included by default in the predefined stubs in each VEX Cortex PROS Project.

Copyright (c) 2011-2014, Purdue University ACM SIG BOTS. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Purdue University ACM SIG BOTS nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL PURDUE UNIVERSITY ACM SIG BOTS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Purdue Robotics OS contains FreeRTOS (<http://www.freertos.org>) whose source code may be obtained from <http://sourceforge.net/projects/freertos/files/> or on request.

Definition in file **main.h**.

6.27.2 Function Documentation

6.27.2.1 autonomous()

```
void autonomous ( )
```

Runs the user autonomous code.

This function will be started in its own task with the default priority and stack size whenever the robot is enabled via the Field Management System or the VEX Competition Switch in the autonomous mode. If the robot is disabled or communications is lost, the autonomous task will be stopped by the kernel. Re-enabling the robot will restart the task, not re-start it from where it left off.

Code running in the autonomous task cannot access information from the VEX Joystick. However, the autonomous function can be invoked from another task if a VEX Competition Switch is not available, and it can access joystick information if called in this way.

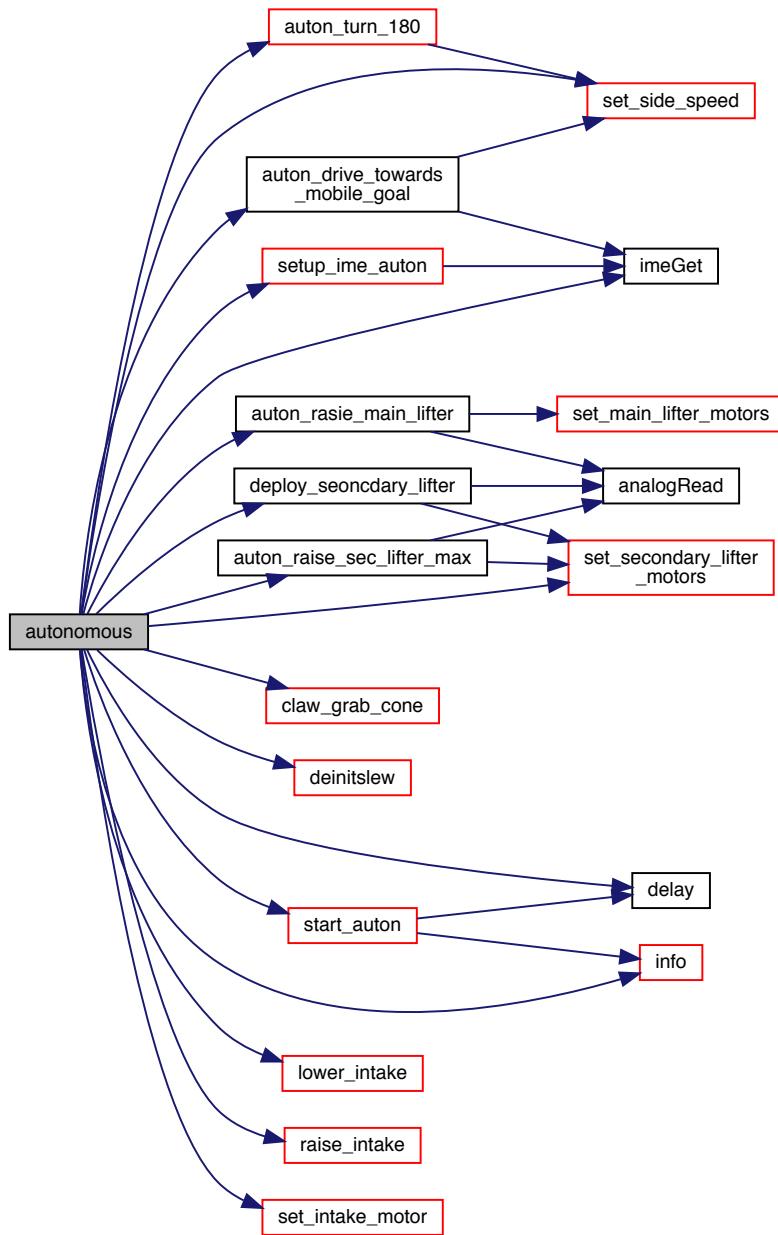
The autonomous task may exit, unlike **operatorControl()** (p. 110) which should never exit. If it does so, the robot will await a switch to another mode or disable/enable cycle.

Definition at line 125 of file **auto.c**.

References **auton_drive_towards_mobile_goal()**, **auton_raise_sec_lifter_max()**, **auton_rasie_main_lifter()**, **auton_turn_180()**, **BOTH**, **claw_grab_cone()**, **deinitslew()**, **delay()**, **deploy_seoncrary_lifter()**, **imeGet()**, **info()**, **lower_intake()**, **raise_intake()**, **set_intake_motor()**, **set_secondary_lifter_motors()**, **set_side_speed()**, **setup_imé_auton()**, and **start_auton()**.

```
00125         {
00126     start_autom();
00127
00128     // How far the left wheels have gone
00129     int counts_drive_left;
00130     // How far the right wheels have gone
00131     int counts_drive_right;
00132     // The average distance traveled forward
00133     int counts_drive;
00134
00135     // Reset the integrated motor controllers
00136     setup_ime_autom(&counts_drive_left, &counts_drive_right, &counts_drive);
00137     info("break 0");
00138     // Deploy claw
00139     deploy_secondary_lifter();
00140     info("break 1");
00141
00142     info("break 2");
00143     set_secondary_lifter_motors(0);
00144
00145     // Grab pre-load cone
00146     delay(300);
00147
00148     auton_raise_sec_lifter_max();
00149     // Raise the lifter
00150     auton_rasie_main_lifter();
00151     // Drive towards the goal
00152
00153     lower_intake();
00154     delay(300);
00155     set_intake_motor(0);
00156
00157     auton_drive_towards_mobile_goal(counts_drive, counts_drive_left,
00158                                     counts_drive_right);
00159     // Stop moving
00160     set_side_speed(BOTH, 0);
00161     delay(1000);
00162
00163     raise_intake();
00164     delay(300);
00165     set_intake_motor(0);
00166
00167     // Drop the cone on the goal
00168     claw_grab_cone();
00169     delay(1000);
00170
00171     auton_turn_180();
00172
00173     counts_drive = 0;
00174
00175     while (counts_drive < MOBILE_GOAL_DISTANCE + ZONE_DISTANCE) {
00176         set_side_speed(BOTH, 127);
00177         // Restablish the distance traveled
00178         imeGet(MID_LEFT_DRIVE, &counts_drive_left);
00179         imeGet(MID_RIGHT_DRIVE, &counts_drive_right);
00180         counts_drive = counts_drive_left + counts_drive_right;
00181         counts_drive /= 2;
00182     }
00183
00184     lower_intake();
00185     delay(300);
00186     set_intake_motor(0);
00187
00188     set_side_speed(BOTH, MIN_SPEED);
00189     delay(1000);
00190     set_side_speed(BOTH, 0);
00191
00192     deinitslaw();
00193 }
```

Here is the call graph for this function:



6.27.2.2 initialize()

```
void initialize ()
```

Runs user initialization code.

This function will be started in its own task with the default priority and stack size once when the robot is starting up. It is possible that the VEXnet communication link may not be fully established at this time, so reading from the VEX Joystick may fail.

This function should initialize most sensors (gyro, encoders, ultrasonics), LCDs, global variables, and IMEs.

This function must exit relatively promptly, or the **operatorControl()** (p. 110) and **autonomous()** (p. 107) tasks will not start. An autonomous mode selection menu like the pre_auton() in other environments can be implemented in this task if desired.

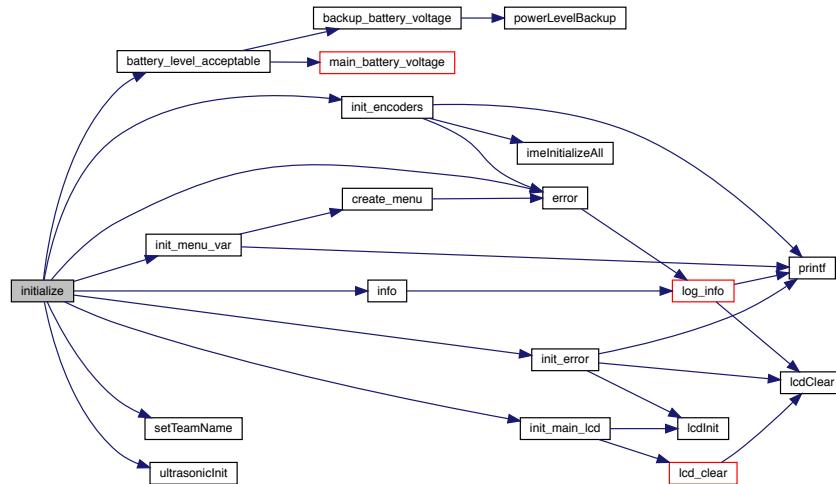
Definition at line 52 of file **init.c**.

References **battery_level_acceptable()**, **error()**, **info()**, **init_encoders()**, **init_error()**, **init_main_lcd()**, **init_menu_var()**, **lifter_ultrasonic**, **setTeamName()**, **STRING_TYPE**, and **ultrasonicInit()**.

```

00052     {
00053     init_main_lcd(uart1);
00054     info("LCD Init");
00055     if (!battery_level_acceptable())
00056         error("Bad main/backup bat");
00057     menu_t *t =
00058         init_menu_var(STRING_TYPE, "TEST Menu", 5, "1", "2", "3", "4", "5");
00059     init_error(true, uart2);
00060     setTeamName("9228A");
00061     init_encoders();
00062     lifter_ultrasonic = ultrasonicInit(4, 5);
00063 }
```

Here is the call graph for this function:



6.27.2.3 initializeIO()

```
void initializeIO ( )
```

Runs pre-initialization code.

This function will be started in kernel mode one time while the VEX Cortex is starting up. As the scheduler is still paused, most API functions will fail.

The purpose of this function is solely to set the default pin modes (**pinMode()** (p. ??)) and port states (**digitalWrite()** (p. ??)) of limit switches, push buttons, and solenoids. It can also safely configure a UART port (**uartOpen()**) but cannot set up an LCD (**lcdInit()** (p. ??)).

Definition at line **36** of file **init.c**.

References **watchdogInit()**.

```
00036 { watchdogInit(); }
```

Here is the call graph for this function:



6.27.2.4 operatorControl()

```
void operatorControl ( )
```

Runs the user operator control code.

This function will be started in its own task with the default priority and stack size whenever the robot is enabled via the Field Management System or the VEX Competition Switch in the operator control mode. If the robot is disabled or communications is lost, the operator control task will be stopped by the kernel. Re-enabling the robot will restart the task, not resume it from where it left off.

If no VEX Competition Switch or Field Management system is plugged in, the VEX Cortex will run the operator control task. Be warned that this will also occur if the VEX Cortex is tethered directly to a computer via the USB A to A cable without any VEX Joystick attached.

Code running in this task can take almost any action, as the VEX Joystick is available and the scheduler is operational. However, proper use of **delay()** (p. ??) or **taskDelayUntil()** (p. ??) is highly recommended to give other tasks (including system tasks such as updating LCDs) time to run.

This task should never exit; it should end with some kind of infinite loop, even if empty.

This function will be started in its own task with the default priority and stack size whenever the robot is enabled via the Field Management System or the VEX Competition Switch in the operator control mode. If the robot is disabled or communications is lost, the operator control task will be stopped by the kernel. Re-enabling the robot will restart the task, not resume it from where it left off.

If no VEX Competition Switch or Field Management system is plugged in, the VEX Cortex will run the operator control task. Be warned that this will also occur if the VEX Cortex is tethered directly to a computer via the USB A to A cable without any VEX Joystick attached.

Code running in this task can take almost any action, as the VEX Joystick is available and the scheduler is operational. However, proper use of **delay()** (p. ??) or **taskDelayUntil()** (p. ??) is highly recommended to give other tasks (including system tasks such as updating LCDs) time to run.

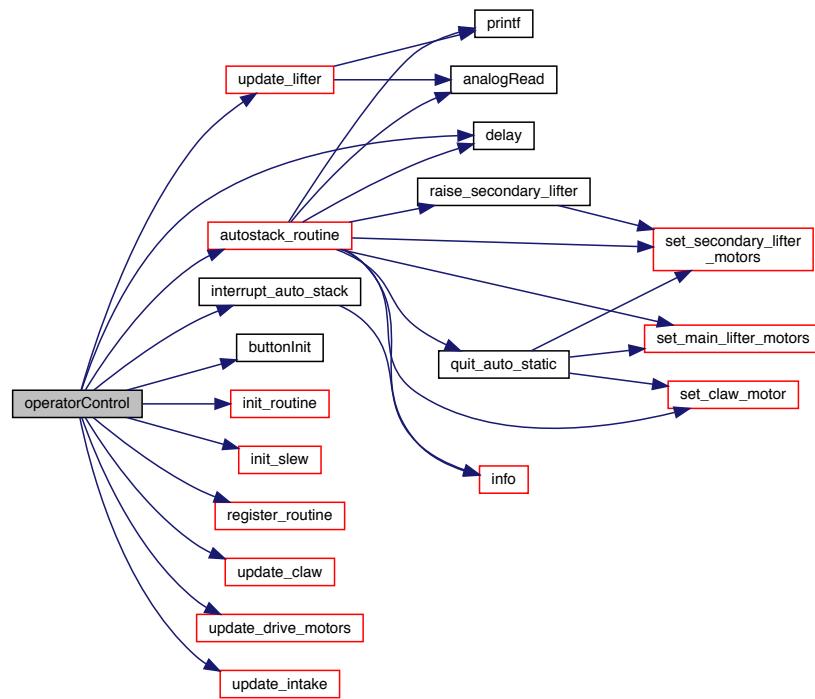
This task should never exit; its should end with some kind of infinite loop, even if empty.

Definition at line 48 of file **opcontrol.c**.

References **autostack_routine()**, **buttonInit()**, **delay()**, **init_routine()**, **init_slew()**, **interrupt_auto_stack()**, **JOY2_7D**, **JOY2_7R**, **register_routine()**, **update_claw()**, **update_drive_motors()**, **update_intake()**, and **update_lifter()**.

```
00048     {
00049         buttonInit();
00050         init_routine();
00051         init_slew();
00052         register_routine(&autostack_routine, JOY2_7D, NULL);
00053         register_routine(&interrupt_auto_stack, JOY2_7R, NULL);
00054         while (1) {
00055             update_claw();
00056             update_intake();
00057             update_lifter();
00058             update_drive_motors();
00059             delay(20);
00060         }
00061     }
```

Here is the call graph for this function:



6.28 main.h

```

00001
00044 #ifndef MAIN_H_
00045
00046 // This prevents multiple inclusion, which isn't bad for this file but is good
00047 // practice
00048 #define MAIN_H_
00049
00050 #include <API.h>
00051
00052 // Allow usage of this file in C++ programs
00053 #ifdef __cplusplus
00054 extern "C" {
00055 #endif
00056
00057 // Prototypes for initialization, operator control and autonomous
00058
00076 void autonomous();
00087 void initializeIO();
00101 void initialize();
00123 void operatorControl();
00124
00125 // End C++ export structure
00126 #ifdef __cplusplus
00127 }
00128 #endif
00129
00130 #endif
  
```

6.29 include/matrix.h File Reference

Various Matrix operations.

Data Structures

- struct **_matrix**
A struct representing a matrix.

TypeDefs

- typedef struct **_matrix** **matrix**
A struct representing a matrix.

Functions

- void **assert** (int assertion, const char *message)
Asserts a condition is true.
- **matrix** * **copyMatrix** (**matrix** *m)
Copies a matrix.
- **matrix** * **covarianceMatrix** (**matrix** *m)
returns the covariance of the matrix
- **matrix** * **dotDiagonalMatrix** (**matrix** *a, **matrix** *b)
performs a diagonal matrix dot product.
- **matrix** * **dotProductMatrix** (**matrix** *a, **matrix** *b)
returns the matrix dot product.
- void **freeMatrix** (**matrix** *m)
Frees the resources of a matrix.
- **matrix** * **identityMatrix** (int n)
Returns an identity matrix of size n by n.
- **matrix** * **makeMatrix** (int width, int height)
Makes a matrix with a width and height parameters.
- **matrix** * **meanMatrix** (**matrix** *m)
Given an "m rows by n columns" matrix, return a matrix where each element represents the mean of that full column.
- **matrix** * **multiplyMatrix** (**matrix** *a, **matrix** *b)
Given a two matrices, returns the multiplication of the two.
- void **printMatrix** (**matrix** *m)
Prints a matrix.
- void **rowSwap** (**matrix** *a, int p, int q)
swaps the rows of a matrix.
- **matrix** * **scaleMatrix** (**matrix** *m, double value)
scales a matrix.
- double **traceMatrix** (**matrix** *m)
Given an "m rows by n columns" matrix.
- **matrix** * **transposeMatrix** (**matrix** *m)
returns the transpose matrix.

6.29.1 Detailed Description

Various Matrix operations.

None of the matrix operations below change the input matrices if an input is required. They all return a new matrix with the new changes. Because memory issues are so prevalent, be sure to use the function to reclaim some of that memory.

Definition in file **matrix.h**.

6.29.2 Typedef Documentation

6.29.2.1 matrix

```
typedef struct __matrix matrix
```

A struct representing a matrix.

6.29.3 Function Documentation

6.29.3.1 assert()

```
void assert (
    int assertion,
    const char * message )
```

Asserts a condition is true.

If the assertion is non-zero (i.e. true), then it returns. If the assertion is zero (i.e. false), then it displays the string and aborts the program. This is meant to act like Python's assert keyword.

Parameters

<i>assertion</i>	the condition, acts like a boolean 0 = false else true
<i>message</i>	the message to print if it fails

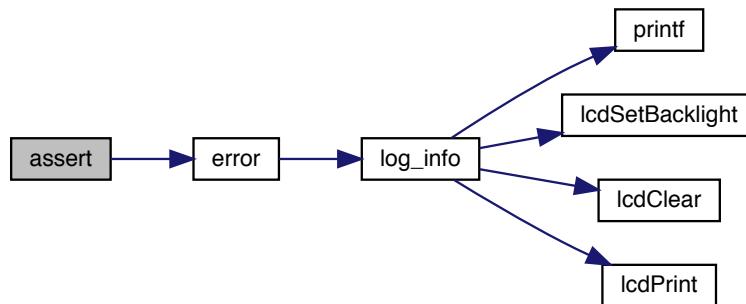
Definition at line 17 of file **matrix.c**.

References **error()**.

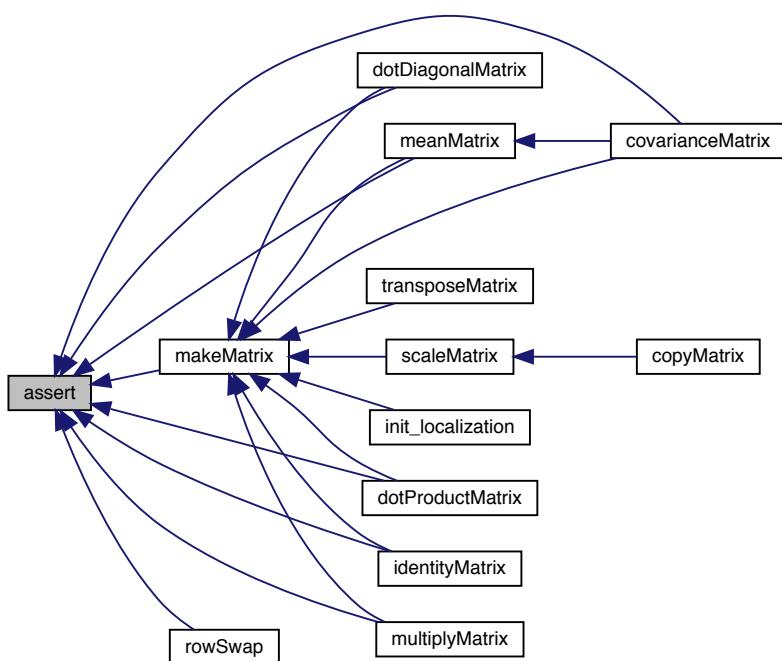
Referenced by **covarianceMatrix()**, **dotDiagonalMatrix()**, **dotProductMatrix()**, **identityMatrix()**, **makeMatrix()**, **meanMatrix()**, **multiplyMatrix()**, and **rowSwap()**.

```
00017
00018     if (assertion == 0) {
00019         error(message);
00020         exit(1);
00021     }
00022 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.29.3.2 copyMatrix()

```
matrix* copyMatrix (
    matrix * m )
```

Copies a matrix.

This function uses scaleMatrix, because scaling matrix by 1 is the same as a copy.

Parameters

m	a pointer to the matrix
----------	-------------------------

Returns

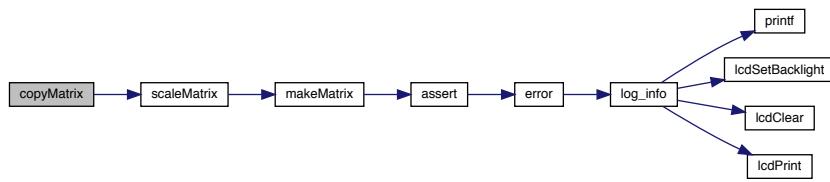
a copied matrix

Definition at line **55** of file **matrix.c**.

References **scaleMatrix()**.

```
00055 { return scaleMatrix(m, 1); }
```

Here is the call graph for this function:



6.29.3.3 covarianceMatrix()

```
matrix* covarianceMatrix (
    matrix * m )
```

returns the covariance of the matrix

Parameters

<i>the</i>	<i>matrix</i>
------------	---------------

Returns

a matrix with n row and n columns, where each element represents covariance of 2 columns.

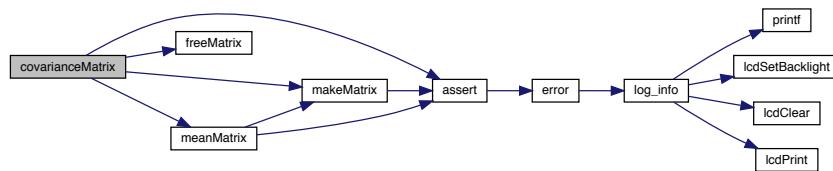
Definition at line 171 of file **matrix.c**.

References **assert()**, **_matrix::data**, **freeMatrix()**, **_matrix::height**, **makeMatrix()**, **meanMatrix()**, and **_matrix::width**.

```

00171
00172     int i, j, k = 0;
00173     matrix *out;
00174     matrix *mean;
00175     double *ptrA;
00176     double *ptrB;
00177     double *ptrOut;
00178
00179     assert(m->height > 1, "Height of matrix cannot be zero or one.");
00180
00181     mean = meanMatrix(m);
00182     out = makeMatrix(m->width, m->width);
00183     ptrOut = out->data;
00184
00185     for (i = 0; i < m->width; i++) {
00186         for (j = 0; j < m->width; j++) {
00187             ptrA = &m->data[i];
00188             ptrB = &m->data[j];
00189             *ptrOut = 0.0;
00190             for (k = 0; k < m->height; k++) {
00191                 *ptrOut += (*ptrA - mean->data[i]) * (*ptrB - mean->data[j]);
00192                 ptrA += m->width;
00193                 ptrB += m->width;
00194             }
00195             *ptrOut /= m->height - 1;
00196             ptrOut++;
00197         }
00198     }
00199
00200     freeMatrix(mean);
00201     return out;
00202 }
```

Here is the call graph for this function:



6.29.3.4 dotDiagonalMatrix()

```
matrix* dotDiagonalMatrix (
    matrix * a,
    matrix * b )
```

performs a diagonal matrix dot product.

Given a two matrices (or the same matrix twice) with identical widths and heights, this method returns a 1 by a->height matrix of the cross product of each matrix along the diagonal.

Dot product is essentially the sum-of-squares of two vectors.

If the second parameter is NULL, it is assumed that we are performing a cross product with itself.

Parameters

<i>a</i>	the first matrix
<i>b</i>	the second matrix

Returns

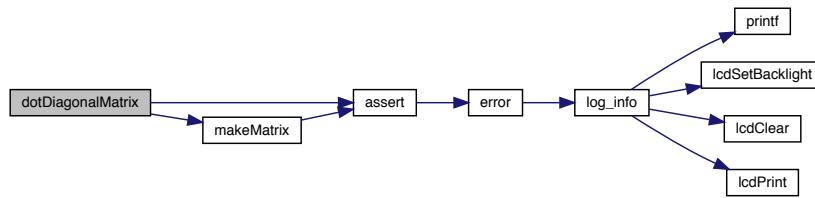
the matrix result

Definition at line **391** of file **matrix.c**.

References **assert()**, **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

```
00391
00392     matrix *out;
00393     double *ptrOut;
00394     double *ptrA;
00395     double *ptrB;
00396     int i, j;
00397
00398     if (b != NULL) {
00399         assert(a->width == b->width && a->height == b->height,
00400                 "Matrices must be of the same dimensionality.");
00401     }
00402
00403     // Are we computing the sum of squares of the same matrix?
00404     if (a == b || b == NULL) {
00405         b = a; // May not appear safe, but we can do this without risk of losing b.
00406     }
00407
00408     out = makeMatrix(1, a->height);
00409     ptrOut = out->data;
00410     ptrA = a->data;
00411     ptrB = b->data;
00412
00413     for (i = 0; i < a->height; i++) {
00414         *ptrOut = 0;
00415         for (j = 0; j < a->width; j++) {
00416             *ptrOut += *ptrA * *ptrB;
00417             ptrA++;
00418             ptrB++;
00419         }
00420         ptrOut++;
00421     }
00422
00423     return out;
00424 }
```

Here is the call graph for this function:



6.29.3.5 dotProductMatrix()

```
matrix* dotProductMatrix (
    matrix * a,
    matrix * b )
```

returns the matrix dot product.

Given a two matrices (or the same matrix twice) with identical widths and different heights, this method returns a $a->\text{height} \times b->\text{height}$ matrix of the cross product of each matrix.

Dot product is essentially the sum-of-squares of two vectors.

Also, if the second parameter is NULL, it is assumed that we are performing a cross product with itself.

Parameters

<i>a</i>	the first matrix
<i>the</i>	second matrix

Returns

the result of the dot product

Definition at line **338** of file **matrix.c**.

References **assert()**, **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

```

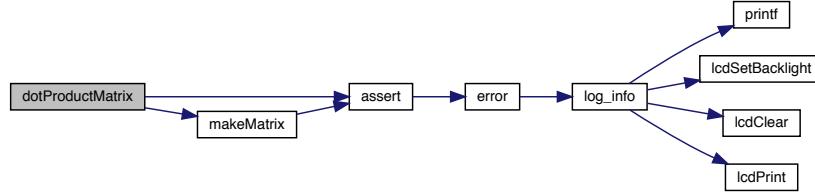
00338
00339     matrix *out;
00340     double *ptrOut;
00341     double *ptrA;
00342     double *ptrB;
00343     int i, j, k;
00344
00345     if (b != NULL) {
00346         assert(a->width == b->width,
  
```

```

00347         "Matrices must be of the same dimensionality.");
00348     }
00349
00350 // Are we computing the sum of squares of the same matrix?
00351 if (a == b || b == NULL) {
00352     b = a; // May not appear safe, but we can do this without risk of losing b.
00353 }
00354
00355 out = makeMatrix(b->height, a->height);
00356 ptrOut = out->data;
00357
00358 for (i = 0; i < a->height; i++) {
00359     ptrB = b->data;
00360
00361     for (j = 0; j < b->height; j++) {
00362         ptrA = &a->data[i * a->width];
00363
00364         *ptrOut = 0;
00365         for (k = 0; k < a->width; k++) {
00366             *ptrOut += *ptrA * *ptrB;
00367             ptrA++;
00368             ptrB++;
00369         }
00370         ptrOut++;
00371     }
00372 }
00373
00374 return out;
00375 }

```

Here is the call graph for this function:



6.29.3.6 freeMatrix()

```

void freeMatrix (
    matrix * m )

```

Frees the resources of a matrix.

Parameters

<i>m</i>	the matrix to free
<i>the</i>	matrix to free

Definition at line **61** of file **matrix.c**.

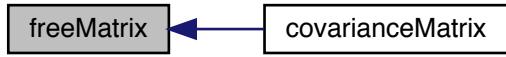
References **_matrix::data**.

Referenced by **covarianceMatrix()**.

```

00061
00062     if (m != NULL) {
00063         if (m->data != NULL) {
00064             free(m->data);
00065             m->data = NULL;
00066         }
00067         free(m);
00068     }
00069     return;
00070 }
```

Here is the caller graph for this function:



6.29.3.7 identityMatrix()

```
matrix* identityMatrix (
    int n )
```

Returns an identity matrix of size n by n.

Parameters

<i>n</i>	the input matrix. parameter.
<i>n</i>	the input matrix.

Returns

the identity matrix parameter.

Definition at line 95 of file **matrix.c**.

References **assert()**, **_matrix::data**, and **makeMatrix()**.

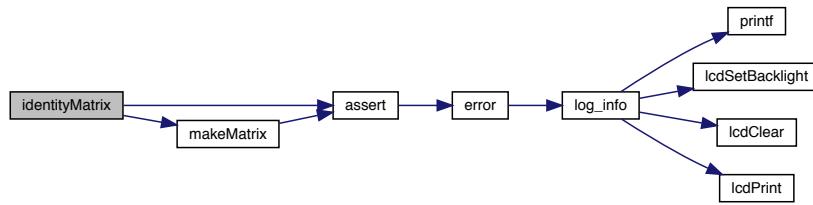
```

00095
00096     int i;
00097     matrix *out;
00098     double *ptr;
00099 }
```

```

00100     assert(n > 0, "Identity matrix must have value greater than zero.");
00101
00102     out = makeMatrix(n, n);
00103     ptr = out->data;
00104     for (i = 0; i < n; i++) {
00105         *ptr = 1.0;
00106         ptr += n + 1;
00107     }
00108
00109     return out;
00110 }
```

Here is the call graph for this function:



6.29.3.8 makeMatrix()

```

matrix* makeMatrix (
    int width,
    int height )
```

Makes a matrix with a width and height parameters.

Parameters

<i>width</i>	The width of the matrix
<i>height</i>	the height of the matrix

Returns

the new matrix

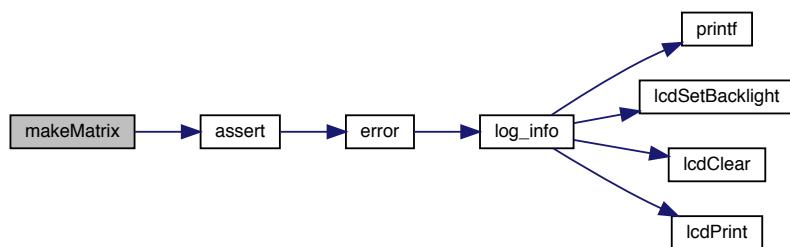
Definition at line **30** of file **matrix.c**.

References **assert()**, **_matrix::data**, **_matrix::height**, and **_matrix::width**.

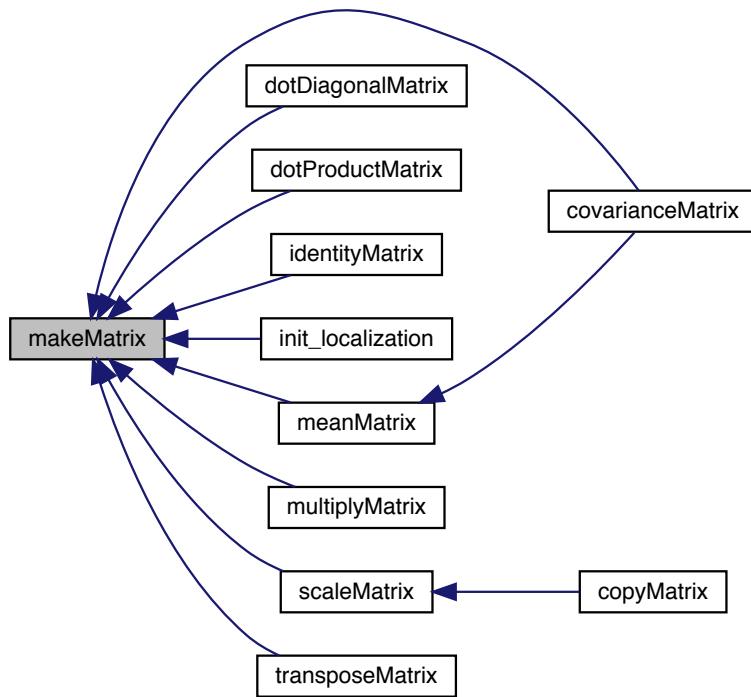
Referenced by **covarianceMatrix()**, **dotDiagonalMatrix()**, **dotProductMatrix()**, **identityMatrix()**, **init_localization()**, **meanMatrix()**, **multiplyMatrix()**, **scaleMatrix()**, and **transposeMatrix()**.

```
00030     matrix *out;
00031     assert(width > 0 && height > 0, "New matrix must be at least a 1 by 1");
00032     out = (matrix *)malloc(sizeof(matrix));
00033     assert(out != NULL, "Out of memory.");
00034
00035     out->width = width;
00036     out->height = height;
00037     out->data = (double *)malloc(sizeof(double) * width * height);
00038     assert(out->data != NULL, "Out of memory.");
00039
00040     memset(out->data, 0.0, width * height * sizeof(double));
00041
00042     return out;
00043 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.29.3.9 meanMatrix()

```
matrix* meanMatrix (
    matrix * m )
```

Given an "m rows by n columns" matrix, return a matrix where each element represents the mean of that full column.

the matrix

Returns

matrix with 1 row and n columns each element represents the mean of that full column.

Definition at line 144 of file **matrix.c**.

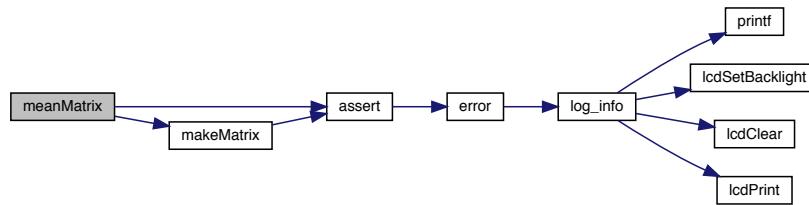
References **assert()**, **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

Referenced by **covarianceMatrix()**.

```

00144
00145     int i, j;
00146     matrix *out;
00147
00148     assert(m->height > 0, "Height of matrix cannot be zero.");
00149
00150     out = makeMatrix(m->width, 1);
00151
00152     for (i = 0; i < m->width; i++) {
00153         double *ptr;
00154         out->data[i] = 0.0;
00155         ptr = &m->data[i];
00156         for (j = 0; j < m->height; j++) {
00157             out->data[i] += *ptr;
00158             ptr += out->width;
00159         }
00160         out->data[i] /= (double)m->height;
00161     }
00162     return out;
00163 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.29.3.10 multiplyMatrix()

```

matrix* multiplyMatrix (
    matrix * a,
    matrix * b )
```

Given a two matrices, returns the multiplication of the two.

Parameters

<i>a</i>	the first matrix
<i>b</i>	the seconf matrix return the result of the multiplication

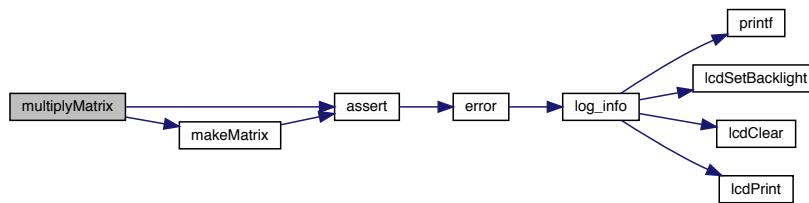
Definition at line **233** of file **matrix.c**.

References **assert()**, **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

```

00233     int i, j, k;
00234     int i, j, k;
00235     matrix *out;
00236     double *ptrOut;
00237     double *ptrA;
00238     double *ptrB;
00239
00240     assert(a->width == b->height,
00241             "Matrices have incorrect dimensions. a->width != b->height");
00242
00243     out = makeMatrix(b->width, a->height);
00244     ptrOut = out->data;
00245
00246     for (i = 0; i < a->height; i++) {
00247
00248         for (j = 0; j < b->width; j++) {
00249             ptrA = &a->data[i * a->width];
00250             ptrB = &b->data[j];
00251
00252             *ptrOut = 0;
00253             for (k = 0; k < a->width; k++) {
00254                 *ptrOut += *ptrA * *ptrB;
00255                 ptrA++;
00256                 ptrB += b->width;
00257             }
00258             ptrOut++;
00259         }
00260     }
00261
00262     return out;
00263 }
```

Here is the call graph for this function:



6.29.3.11 printMatrix()

```
void printMatrix (
    matrix * m )
```

Prints a matrix.

Parameters

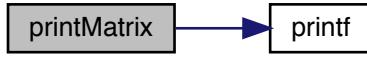
<i>m</i>	the matrix
<i>the</i>	matrix

Definition at line 76 of file **matrix.c**.

References **_matrix::data**, **_matrix::height**, **printf()**, and **_matrix::width**.

```
00076                               {
00077     int i, j;
00078     double *ptr = m->data;
00079     printf("%d %d\n", m->width, m->height);
00080     for (i = 0; i < m->height; i++) {
00081         for (j = 0; j < m->width; j++) {
00082             printf(" %.9.6f", *(ptr++));
00083         }
00084         printf("\n");
00085     }
00086     return;
00087 }
```

Here is the call graph for this function:

**6.29.3.12 rowSwap()**

```
void rowSwap (
    matrix * a,
    int p,
    int q )
```

swaps the rows of a matrix.

This method changes the input matrix. Given a matrix, this algorithm will swap rows p and q, provided that p and q are less than or equal to the height of matrix A and p and q are different values.

Parameters

<i>the</i>	matrix to swap. This method changes the input matrix.
<i>the</i>	first row
<i>the</i>	second row

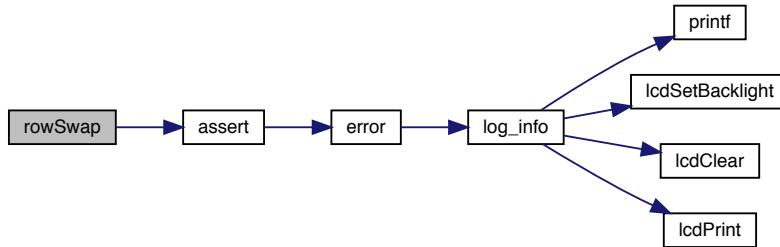
Definition at line 294 of file **matrix.c**.

References **assert()**, **_matrix::data**, **_matrix::height**, and **_matrix::width**.

```

00294     {
00295     int i;
00296     double temp;
00297     double *pRow;
00298     double *qRow;
00299
00300     assert(a->height > 2, "Matrix must have at least two rows to swap.");
00301     assert(p < a->height && q < a->height,
00302             "Values p and q must be less than the height of the matrix.");
00303
00304     // If p and q are equal, do nothing.
00305     if (p == q) {
00306         return;
00307     }
00308
00309     pRow = a->data + (p * a->width);
00310     qRow = a->data + (q * a->width);
00311
00312     // Swap!
00313     for (i = 0; i < a->width; i++) {
00314         temp = *pRow;
00315         *pRow = *qRow;
00316         *qRow = temp;
00317         pRow++;
00318         qRow++;
00319     }
00320
00321     return;
00322 }
```

Here is the call graph for this function:



6.29.3.13 scaleMatrix()

```

matrix* scaleMatrix (
    matrix * m,
    double value )
```

scales a matrix.

Parameters

<i>m</i>	the matrix to scale
<i>the</i>	value to scale by

Returns

a new matrix where each element in the input matrix is multiplied by the scalar value

Definition at line **272** of file **matrix.c**.

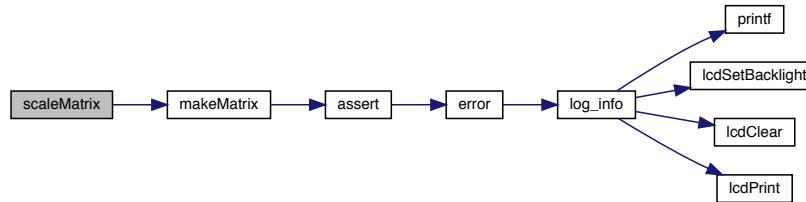
References **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

Referenced by **copyMatrix()**.

```

00272
00273     int i, elements = m->width * m->height;
00274     matrix *out = makeMatrix(m->width, m->height);
00275     double *ptrM = m->data;
00276     double *ptrOut = out->data;
00277
00278     for (i = 0; i < elements; i++) {
00279         *(ptrOut++) = *(ptrM++) * value;
00280     }
00281
00282     return out;
00283 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.29.3.14 traceMatrix()

```
double traceMatrix (
    matrix * m )
```

Given an "m rows by n columns" matrix.

Returns

the sum of the elements along the diagonal. param m the matrix to trace

Given an "m rows by n columns" matrix.

Returns

the sum of the elements along the diagonal.

Definition at line **117** of file **matrix.c**.

References **_matrix::data**, **_matrix::height**, and **_matrix::width**.

```
00117     {
00118     int i;
00119     int size;
00120     double *ptr = m->data;
00121     double sum = 0.0;
00122
00123     if (m->height < m->width) {
00124         size = m->height;
00125     } else {
00126         size = m->width;
00127     }
00128
00129     for (i = 0; i < size; i++) {
00130         sum += *ptr;
00131         ptr += m->width + 1;
00132     }
00133
00134     return sum;
00135 }
```

6.29.3.15 transposeMatrix()

```
matrix* transposeMatrix (
    matrix * m )
```

returns the transpose matrix.

Parameters

<i>the</i>	matrix to transpose.
------------	----------------------

Returns

the transposed matrix.

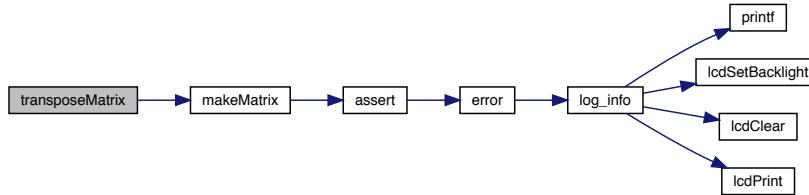
Definition at line 209 of file **matrix.c**.

References **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

```

00209         {
00210     matrix *out = makeMatrix(m->height, m->width);
00211     double *ptrM = m->data;
00212     int i, j;
00213
00214     for (i = 0; i < m->height; height; i++) {
00215         double *ptrOut;
00216         ptrOut = &out->data[i];
00217         for (j = 0; j < m->width; j++) {
00218             *ptrOut = *ptrM;
00219             ptrM++;
00220             ptrOut += out->width;
00221         }
00222     }
00223
00224     return out;
00225 }
```

Here is the call graph for this function:

**6.30 matrix.h**

```

00001
00010 #ifndef _MATRIX_H_
00011 #define _MATRIX_H_
00012
00016 typedef struct _matrix {
00017     int height;
00018     int width;
00019     double *data;
00020 } matrix;
00021
00032 void assert(int assertion, const char *message);
00033
00040 matrix *makeMatrix(int width, int height);
00041
00049 matrix *copyMatrix(matrix *m);
00050
00055 void freeMatrix(matrix *m);
00056
00061 void printMatrix(matrix *m);
00062
00068 matrix *identityMatrix(int n);
00069
```

```

00076 double traceMatrix(matrix *m);
00077
00083 matrix *transposeMatrix(matrix *m);
00084
00092 matrix *meanMatrix(matrix *m);
00093
00100 matrix *multiplyMatrix(matrix *a, matrix *b);
00101
00109 matrix *scaleMatrix(matrix *m, double value);
00110
00117 matrix *covarianceMatrix(matrix *m);
00118
00128 void rowSwap(matrix *a, int p, int q);
00143 matrix *dotProductMatrix(matrix *a, matrix *b);
00144
00159 matrix *dotDiagonalMatrix(matrix *a, matrix *b);
00160
00161 #endif

```

6.31 include/menu.h File Reference

Contains menu functionality and abstraction.

Data Structures

- struct **menu_t**

Represents a specific instance of a menu.

Typedefs

- typedef struct **menu_t** **menu_t**

Represents a specific instance of a menu.

Enumerations

- enum **menu_type** { **INT_TYPE**, **FLOAT_TYPE**, **STRING_TYPE** }

Represents the different types of menus.

Functions

- void **deinit_menu** (**menu_t** *menu)

Destroys a menu. Menu must be freed or will cause memory leak

- int **display_menu** (**menu_t** *menu)

Displays a menu context.

- **menu_t** * **init_menu_float** (enum **menu_type** type, float **min**, float **max**, float step, const char *prompt)

Creates a menu context, but does not display.

- **menu_t** * **init_menu_int** (enum **menu_type** type, int **min**, int **max**, int step, const char *prompt)

Creates a menu context, but does not display.

- **menu_t** * **init_menu_var** (enum **menu_type** type, const char *prompt, int nums,...)

Creates a menu context, but does not display.

6.31.1 Detailed Description

Contains menu functionality and abstraction.

Author

Chris Jerrett

Date

9/9/2017

Definition in file **menu.h**.

6.31.2 Typedef Documentation

6.31.2.1 menu_t

```
typedef struct menu_t menu_t
```

Represents a specific instance of a menu.

Will cause a memory leak if not deinitialized via denint_menu.

Author

Chris Jerrett

Date

9/8/17

See also

[menu.h](#) (p. 130)
[menu_t](#) (p. 16)
[create_menu](#) (p. 303)
[init_menu](#)
[display_menu](#) (p. 305)
[menu_type](#) (p. 132)
[denint_menu](#) (p. 304)

6.31.3 Enumeration Type Documentation

6.31.3.1 menu_type

enum **menu_type**

Represents the different types of menus.

Author

Chris Jerrett

Date

9/8/17

See also

menu.h (p. 130)
menu_t (p. 16)
create_menu (p. 303)
init_menu
display_menu (p. 134)
menu_type (p. 132)

Enumerator

INT_TYPE	Menu type allowing user to select a integer. The integer type menu has a max, min and a step value. Each step is calculated. Will return the index of the selected value. Example: User goes forwards twice then it will return 2.
FLOAT_TYPE	Menu type allowing user to select a float. The float type menu has a max, min and a step value. Each step is calculated. Will return the index of the selected value. Example: User goes forwards twice then it will return 2.
STRING_TYPE	Menu type allowing user to select a string from a array of strings. Will return the index of the selected value. Example: User goes forwards twice then it will return 2.

Definition at line **30** of file **menu.h**.

```
00030
00037     INT_TYPE,
00044     FLOAT_TYPE,
00050     STRING_TYPE
00051 };
```

6.31.4 Function Documentation

6.31.4.1 denint_menu()

```
void denint_menu (
    menu_t * menu )
```

Destroys a menu. *Menu must be freed or will cause memory leak*

Parameters

<i>menu</i>	the menu to free
-------------	------------------

See also

menu

Author

Chris Jerrett

Date

9/8/17

Definition at line **203** of file **menu.c**.

References **menu_t::options**, and **menu_t::prompt**.

```
00203
00204     free(menu->prompt);
00205     if (menu->options != NULL)
00206         free(menu->options);
00207     free(menu);
00208 }
```

6.31.4.2 display_menu()

```
int display_menu (
    menu_t * menu )
```

Displays a menu context.

Menu must be freed or will cause memory leak! Will exit if robot is enabled. This prevents menu from locking up system in even of a reset.

Parameters

<i>menu</i>	the menu to display
-------------	---------------------

See also

menu_type (p. 132)

Author*Chris Jerrett***Date**

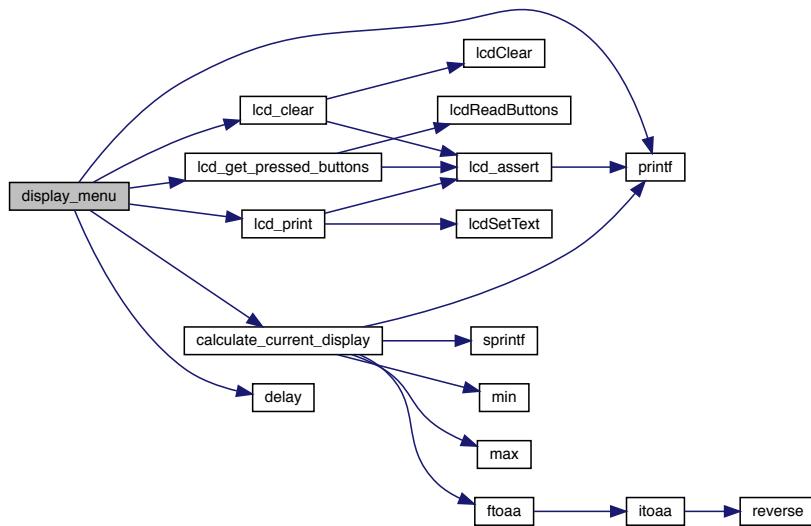
9/8/17

Definition at line **164** of file **menu.c**.References **calculate_current_display()**, **menu_t::current**, **delay()**, **Lcd_clear()**, **Lcd_get_pressed_buttons()**, **Lcd_print()**, **PRESSED**, **printf()**, **menu_t::prompt**, and **RELEASED**.

```

00164         {
00165     lcd_print(TOP_ROW, menu->prompt);
00166     printf("printed prompt\n");
00167     // Will exit if teleop or autonomous begin. This is extremely important if
00168     // robot disconnects or resets.
00169     char val[16];
00170     while (lcd_get_pressed_buttons().middle == RELEASED) {
00171         calculate_current_display(val, menu);
00172
00173         if (lcd_get_pressed_buttons().right == PRESSED) {
00174             menu->current += 1;
00175         }
00176         if (lcd_get_pressed_buttons().left == PRESSED) {
00177             menu->current -= 1;
00178         }
00179         printf("%s\n", val);
00180         printf("%d\n", menu->current);
00181         lcd_print(2, val);
00182         delay(300);
00183     }
00184     printf("%d\n", menu->current);
00185     printf("return\n");
00186     lcd_clear();
00187     lcd_print(1, "Thk Cm Agn");
00188     lcd_print(2, val);
00189     delay(800);
00190     lcd_clear();
00191     return menu->current;
00192 }
```

Here is the call graph for this function:



6.31.4.3 init_menu_float()

```
menu_t* init_menu_float (
    enum menu_type type,
    float min,
    float max,
    float step,
    const char * prompt )
```

Creates a menu context, but does not display.

Menu must be freed or will cause memory leak!

Parameters

type	<i>the type of menu</i>
-------------	-------------------------

See also

[menu_type](#) (p. 132)

Parameters

min	<i>the minimum value</i>
max	<i>the maximum value</i>
step	<i>the step value</i>
prompt	<i>the prompt to display to user</i>

Author

Chris Jerrett

Date

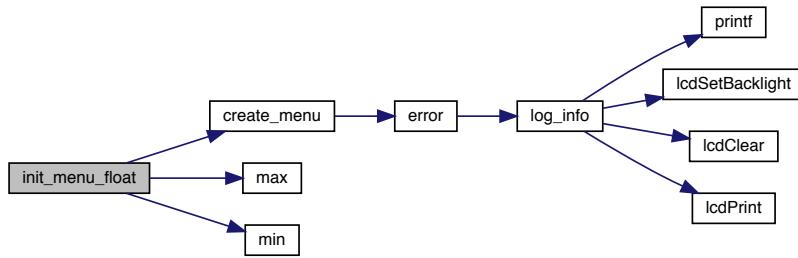
9/8/17

Definition at line 111 of file **menu.c**.

References **create_menu()**, **max()**, **menu_t::max_f**, **min()**, **menu_t::min_f**, and **menu_t::step_f**.

```
00112                                     {
00113     menu_t *menu = create_menu(type, prompt);
00114     menu->min_f = min;
00115     menu->max_f = max;
00116     menu->step_f = step;
00117     return menu;
00118 }
```

Here is the call graph for this function:



6.31.4.4 init_menu_int()

```
menu_t* init_menu_int (
    enum menu_type type,
    int min,
    int max,
    int step,
    const char * prompt )
```

Creates a menu context, but does not display.

Menu must be freed or will cause memory leak

Parameters

type	<i>the type of menu</i>
------	-------------------------

See also

[menu_type](#) (p. 132)

Parameters

min	<i>the minimum value</i>
max	<i>the maximum value</i>
step	<i>the step value</i>
prompt	<i>the prompt to display to user</i>

Author

Chris Jerrett

Date

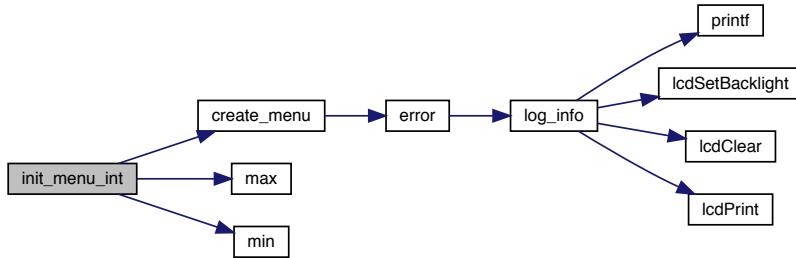
9/8/17

Definition at line **88** of file **menu.c**.

References **create_menu()**, **menu_t::current**, **max()**, **menu_t::max**, **min()**, **menu_t::min**, and **menu_t::step**.

```
00089     {
00090     menu_t *menu = create_menu(type, prompt);
00091     menu->min = min;
00092     menu->max = max;
00093     menu->step = step;
00094     menu->current = 0;
00095     return menu;
00096 }
```

Here is the call graph for this function:



6.31.4.5 init_menu_var()

```
menu_t* init_menu_var (
    enum menu_type type,
    const char * prompt,
    int nums,
    ...
)
```

Creates a menu context, but does not display.

Menu must be freed or will cause memory leak

Parameters

<code>type</code>	<i>the type of menu</i>
-------------------	-------------------------

See also

[menu_type](#) (p. 132)

Parameters

<code>nums</code>	<i>the number of elements passed to function</i>
<code>prompt</code>	<i>the prompt to display to user</i>
<code>options</code>	<i>the options to display for user</i>

Author

Chris Jerrett

Date

9/8/17

Definition at line **60** of file **menu.c**.

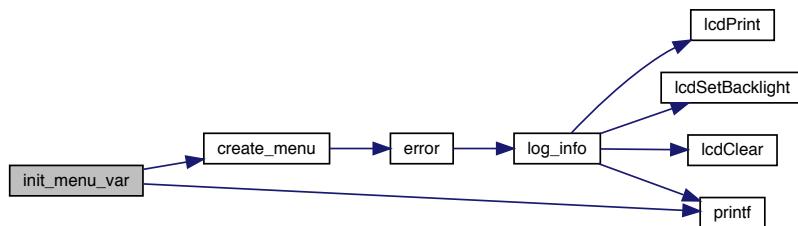
References `create_menu()`, `menu_t::length`, `menu_t::options`, and `printf()`.

Referenced by `initialize()`.

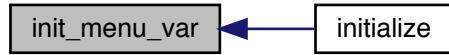
```

00060
00061     menu_t *menu = create_menu(type, prompt);
00062     va_list ap;
00063     char **options_array = (char **)calloc(sizeof(char *), nums);
00064     va_start(ap, nums);
00065     for (int i = 0; i < nums; i++) {
00066         options_array[i] = (char *)va_arg(ap, char *);
00067         printf("%s\n", options_array[i]);
00068     }
00069     va_end(ap);
00070     menu->options = options_array;
00071     menu->length = nums;
00072     return menu;
00073 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.32 menu.h

```
00001
00008 #ifndef _MENU_H_
00009 #define _MENU_H_
00010
00011 #include "API.h"
00012 #include "lcd.h"
00013 #include "log.h"
00014 #include <float.h>
00015 #include <limits.h>
00016 #include <string.h>
00017 #include <vlib.h>
00018
00030 enum menu_type {
00037     INT_TYPE,
00044     FLOAT_TYPE,
00050     STRING_TYPE
00051 };
00052
00066 typedef struct menu_t {
00072     enum menu_type type;
00073     char **options;
00080
00086     unsigned int length;
00087
00094     int min;
00095
00102     int max;
00103
00111     int step;
00112
00119     float min_f;
00120
00127     float max_f;
00128
00136     float step_f;
00142     int current;
00150     char *prompt;
00151 } menu_t;
00152
00165 menu_t *init_menu_var(enum menu_type type, const char *prompt, int nums, ...);
00166
00180 menu_t *init_menu_int(enum menu_type type, int min, int max, int step,
00181                         const char *prompt);
00182
00196 menu_t *init_menu_float(enum menu_type type, float min, float max, float step,
00197                           const char *prompt);
00198
00209 int display_menu(menu_t *menu);
00210
00220 void denint_menu(menu_t *menu);
00221
00222 #endif
```

6.33 include/mobile_goal_intake.h File Reference

Functions

- **void lower_intake ()**
lowers the intake
- **void raise_intake ()**
raises the intake
- **void set_intake_motor (int n)**
sets the intake motor
- **void update_intake ()**
updates the mobile goal intake in teleop.

6.33.1 Function Documentation

6.33.1.1 lower_intake()

```
void lower_intake ( )
```

lowers the intake

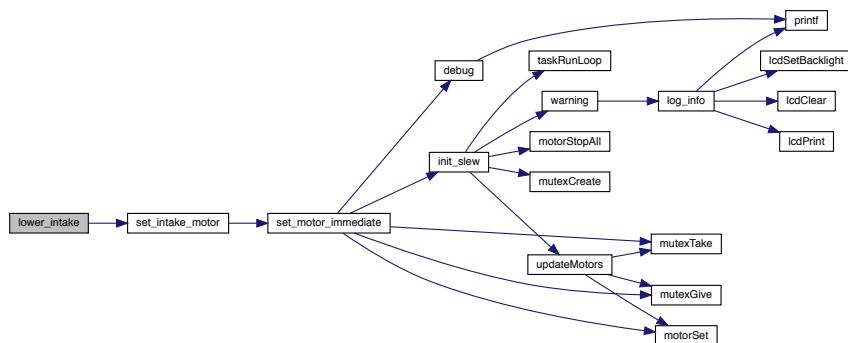
Definition at line 5 of file **mobile_goal_intake.c**.

References **set_intake_motor()**.

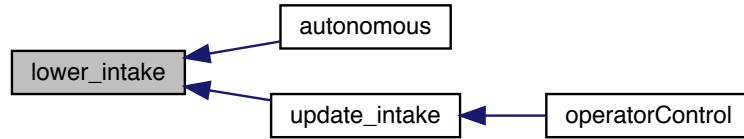
Referenced by **autonomous()**, and **update_intake()**.

```
00005 { set_intake_motor(-100); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.33.1.2 raise_intake()

```
void raise_intake( )
```

raises the intake

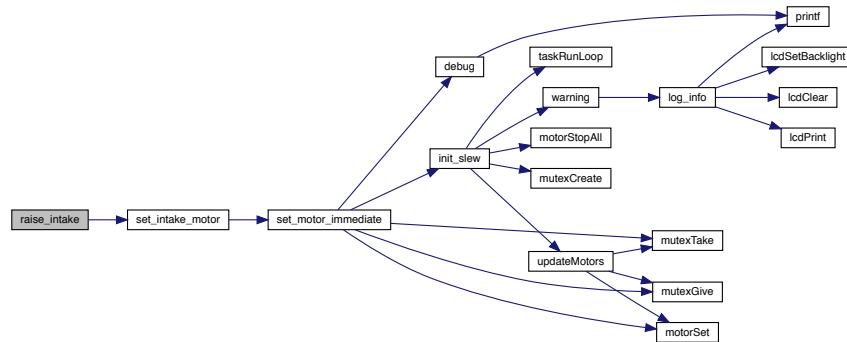
Definition at line 7 of file **mobile_goal_intake.c**.

References **set_intake_motor()**.

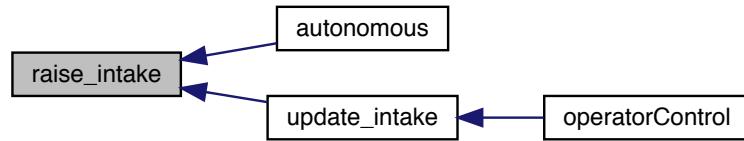
Referenced by **autonomous()**, and **update_intake()**.

```
00007 { set_intake_motor(100); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.33.1.3 set_intake_motor()

```
void set_intake_motor (
    int n )
```

sets the intake motor

Author

Chris Jerrett

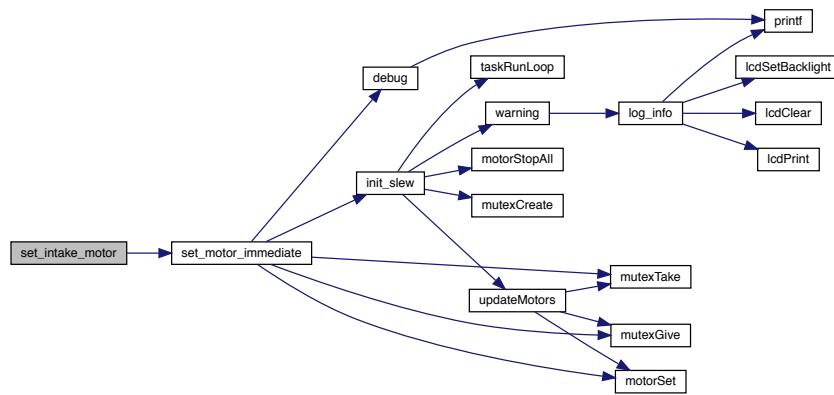
Definition at line 3 of file **mobile_goal_intake.c**.

References **set_motor_immediate()**.

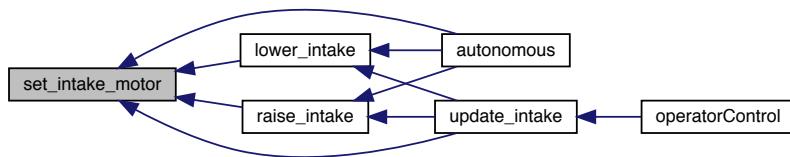
Referenced by **autonomous()**, **lower_intake()**, **raise_intake()**, and **update_intake()**.

```
00003 { set_motor_immediate(MOBILE_INTAKE_MOTOR, n); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.33.1.4 update_intake()

```
void update_intake ( )
```

updates the mobile goal intake in teleop.

Author

Chris Jerrett

Definition at line 12 of file **mobile_goal_intake.c**.

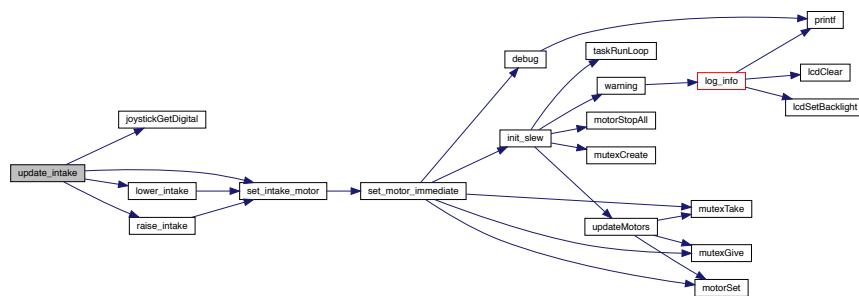
References **joystickGetDigital()**, **lower_intake()**, **raise_intake()**, and **set_intake_motor()**.

Referenced by **operatorControl()**.

```

00012     {
00013     if (joystickGetDigital(MASTER, 8, JOY_UP)) {
00014         raise_intake();
00015     } else if (joystickGetDigital(MASTER, 8, JOY_DOWN)) {
00016         lower_intake();
00017     } else
00018         set_intake_motor(0);
00019 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.34 mobile_goal_intake.h

```
00001 #ifndef _MOBLE_GOAL_INTAKE_
00002 #define _MOBLE_GOAL_INTAKE_
00003
00004 #include "controller.h"
00005 #include "motor_ports.h"
00006 #include "slew.h"
00007
00013 void update_intake();
00014
00019 void set_intake_motor(int n);
00020
00024 void raise_intake();
00025
00029 void lower_intake();
00030
00031 #endif
```

6.35 include/motor_ports.h File Reference

The motor port definitions

Macros for the different motors ports.

6.35.1 Detailed Description

The motor port definitions

Macros for the different motors ports.

Definition in file **motor_ports.h**.

6.36 motor_ports.h

```

00001
00006 #ifndef _MOTOT_PORTS_H_
00007 #define _MOTOR_PORTS_H_
00008
00012 #define MAX_SPEED 127
00013
00017 #define MIN_SPEED -128
00018
00024 #define MOTOR_FRONT_RIGHT 2
00025
00031 #define MOTOR_FRONT_LEFT 7
00032
00038 #define MOTOR_MIDDLE_RIGHT 3
00039
00045 #define MOTOR_MIDDLE_LEFT 6
00046
00052 #define MOTOR_BACK_RIGHT 4
00053
00058 #define MOTOR_BACK_LEFT 5
00059
00063 #define MOTOR_MAIN_LIFTER 9
00064
00068 #define CLAW_MOTOR 10
00069
00073 #define MOTOR_SECONDARY_LIFTER 1
00074
00078 #define MOBILE_INTAKE_MOTOR 8
00079
00080#endif

```

6.37 include/potentiometer.h File Reference

6.38 potentiometer.h

```

00001
00006 #ifndef _POTENTIOMETER_H_
00007 #define _POTENTIOMETER_H_
00008
00009 #define TICK_MAX 4095.0
00010 #define DEG_MAX 250.0
00011
00012#endif

```

6.39 include/routines.h File Reference

Data Structures

- struct **routine_t**

Routine system that allows mapping buttons to actions.

Typedefs

- typedef struct **routine_t routine_t**

Routine system that allows mapping buttons to actions.

Functions

- void **deinit_routines** ()
Stops the routine system.
- void **init_routine** ()
Starts the routine system.
- void **register_routine** (void(*routine)(void *), **button_t** on_buttons, **button_t** *prohibited_buttons)
Registers a routine for the system to use.
- void **routine_task** ()
Task that manages routines.

6.39.1 Typedef Documentation

6.39.1.1 **routine_t**

```
typedef struct routine_t routine_t
```

Routine system that allows mapping buttons to actions.

Author

Chris Jerrett

Date

1/8/17 Struct representing a routine

6.39.2 Function Documentation

6.39.2.1 **deinit_routines()**

```
void deinit_routines ( )
```

Stops the routine system.

Author

Chris Jerrett

Definition at line **47** of file **routines.c**.

References **list_destroy()**.

```
00047 { list_destroy(routine_list); }
```

Here is the call graph for this function:



6.39.2.2 init_routine()

```
void init_routine ( )
```

Starts the routine system.

Author

Chris Jerrett

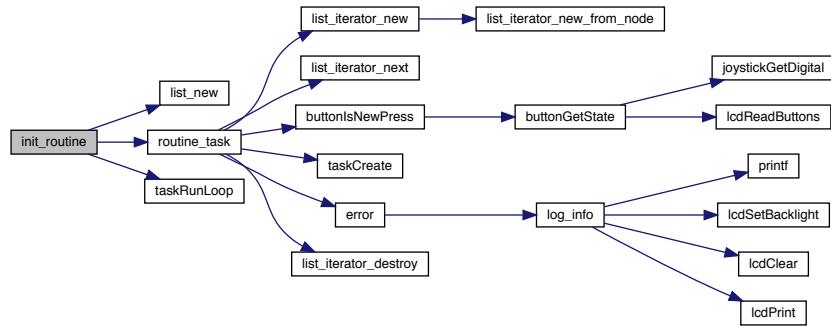
Definition at line **38** of file **routines.c**.

References **list_new()**, **routine_task()**, **routine_task_var**, and **taskRunLoop()**.

Referenced by **operatorControl()**.

```
00038           {
00039     routine_list = list_new();
00040     routine_task_var = taskRunLoop(routine_task, 20);
00041 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.39.2.3 register_routine()

```

void register_routine (
    void(*)(void *) routine,
    button_t on_buttons,
    button_t * prohibited_buttons )
  
```

Registers a routine for the system to use.

Parameters

<i>routine</i>	The routine to register
<i>on_buttons</i>	the trigger button
<i>prohibited_buttons</i>	the buttons it blocks

Todo

Parameters

<i>routine</i>	The routine to register
<i>on_buttons</i>	the trigger button
<i>prohibited_buttons</i>	the buttons it blocks

Todo**Author**

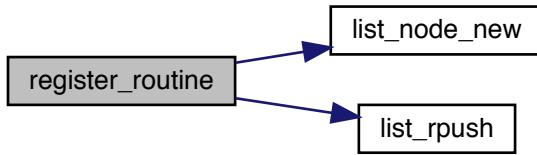
Chris Jerrett

Definition at line **56** of file **routines.c**.References **routine_t::blocked_buttons**, **list_node_new()**, **list_rpush()**, **routine_t::on_button**, **routine_t::routine**, and **list_node::val**.Referenced by **operatorControl()**.

```

00057
00058     struct routine_t *r = (struct routine_t *)malloc(sizeof(routine_t));
00059     r->blocked_buttons = prohibited_buttons;
00060     r->routine = routine;
00061     r->on_button = on_buttons;
00062     list_node_t *node = list_node_new(r);
00063     node->val = r;
00064     list_rpush(routine_list, node);
00065 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.39.2.4 routine_task()

```
void routine_task ( )
```

Task that manages routines.

Author

Chris Jerrett

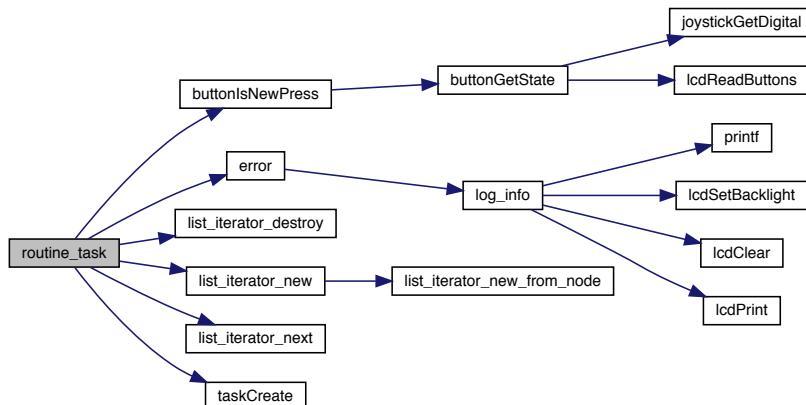
Definition at line 14 of file **routines.c**.

References **buttonIsNewPress()**, **error()**, **LIST_HEAD**, **list_iterator_destroy()**, **list_iterator_new()**, **list_iterator_next()**, **routine_t::on_button**, **routine_t::routine**, **taskCreate()**, and **list_node::val**.

Referenced by **init_routine()**.

```
00014     {
00015     list_node_t *node;
00016     list_iterator_t *it = list_iterator_new(routine_list, LIST_HEAD);
00017     if (it != NULL) {
00018         while (node = list_iterator_next(it)) {
00019             if (node->val != NULL) {
00020                 routine_t *routine = (routine_t *) (node->val);
00021                 if (buttonIsNewPress(routine->on_button)) {
00022                     TaskHandle task =
00023                         taskCreate(routine->routine, TASK_DEFAULT_STACK_SIZE, NULL,
00024                                     TASK_PRIORITY_DEFAULT);
00025                 }
00026             }
00027         }
00028     } else {
00029         error("List iterator was null");
00030     }
00031     list_iterator_destroy(it);
00032 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.40 routines.h

```
00001
00006 #include "controller.h"
00007
00011 typedef struct routine_t {
00012     // the button it triggers on
00013     button_t on_button;
00014     // buttons it disables @todo
00015     button_t *blocked_buttons;
00016
00017     // the routine to start the function on
00018     void (*routine)(void *);
00019 } routine_t;
00020
00024 void init_routine();
00025
00029 void routine_task();
00030
00034 void deinit_routines();
00035
00042 void register_routine(void (*routine)(void *), button_t on_buttons,
00043                         button_t *prohibited_buttons);
```

6.41 include/sensors.h File Reference

Variables

- **Ultrasonic_lifter_ultrasonic**

6.41.1 Variable Documentation

6.41.1.1 lifter_ultrasonic

Ultrasonic_lifter_ultrasonic

Definition at line 24 of file **sensors.h**.

Referenced by **autostack_routine()**, **initialize()**, and **main_lifter_update()**.

6.42 sensors.h

```
00001  
00008 #ifndef _PORTS_H_  
00009 #define _PORTS_H_  
00010  
00011 #include "API.h"  
00012  
00020 #define IME_FRONT_RIGHT 0  
00021 #define LIFTER 2  
00022 #define CLAW_POT 1  
00023  
00024 Ultrasonic_lifter_ultrasonic;  
00025  
00026 #endif
```

6.43 include/slew.h File Reference

Contains the slew rate controller wrapper for the motors.

Functions

- void **deinitslew ()**
Deinitializes the slew rate controller and frees memory.
- void **init_slew ()**
Initializes the slew rate controller.
- void **set_motor_immediate** (int motor, int speed)
Sets the motor speed ignoring the slew controller.
- void **set_motor_slew** (int motor, int speed)
Sets motor speed wrapped inside the slew rate controller.
- void **updateMotors ()**
Closes the distance between the desired motor value and the current motor value by half for each motor.

6.43.1 Detailed Description

Contains the slew rate controller wrapper for the motors.

Author

Chris Jerrett

Date

9/14/17

Definition in file **slew.h**.

6.43.2 Function Documentation

6.43.2.1 deinitslew()

```
void deinitslew ( )
```

Deinitializes the slew rate controller and frees memory.

Author

Chris Jerrett

Date

9/14/17

Definition at line **59** of file **slew.c**.

References **initialized**, **motors_curr_speeds**, **motors_set_speeds**, **slew**, and **taskDelete()**.

Referenced by **autonomous()**.

```
00059     {
00060     taskDelete(slew);
00061     memset(motors_set_speeds, 0, sizeof(int) * 10);
00062     memset(motors_curr_speeds, 0, sizeof(int) * 10);
00063     initialized = false;
00064 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.43.2.2 init_slew()

```
void init_slew ( )
```

Initializes the slew rate controller.

Author

Chris Jerrett, Christian DeSimone

Date

9/14/17

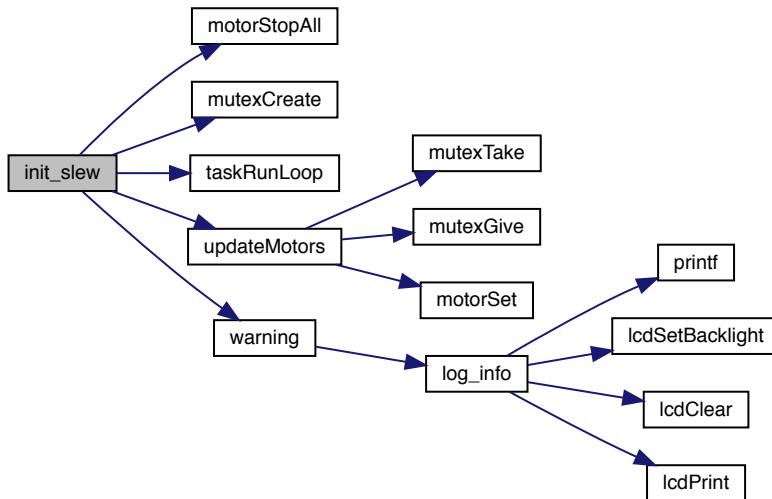
Definition at line 42 of file **slew.c**.

References **initialized**, **motors_curr_speeds**, **motors_set_speeds**, **motorStopAll()**, **mutexCreate()**, **slew**, **speeds_mutex**, **taskRunLoop()**, **updateMotors()**, and **warning()**.

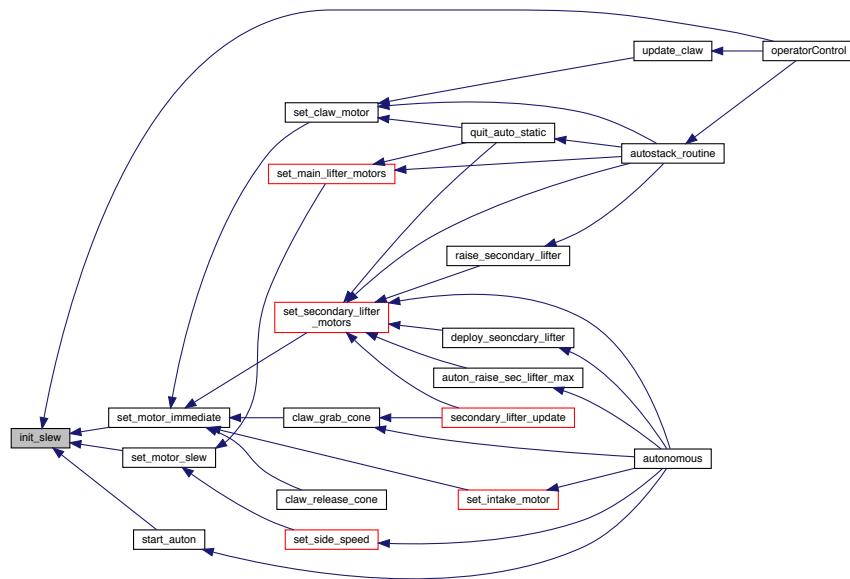
Referenced by **operatorControl()**, **set_motor_immediate()**, **set_motor_slew()**, and **start_auton()**.

```
00042
00043     {
00044     if (initialized) {
00045         warning("Trying to init already init slew");
00046     }
00047     memset(motors_set_speeds, 0, sizeof(int) * 10);
00048     memset(motors_curr_speeds, 0, sizeof(int) * 10);
00049     motorStopAll();
00050     speeds_mutex = mutexCreate();
00051     slew = taskRunLoop(updateMotors, 100);
00052     initialized = true;
00053 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.43.2.3 `set_motor_immediate()`

```
void set_motor_immediate (
    int motor,
    int speed )
```

Sets the motor speed ignoring the slew controller.

Parameters

<code>motor</code>	the motor port to use
<code>speed</code>	the speed to use, between -127 and 127

Author

Chris Jerrett

Date

9/14/17

Definition at line **90** of file **slew.c**.

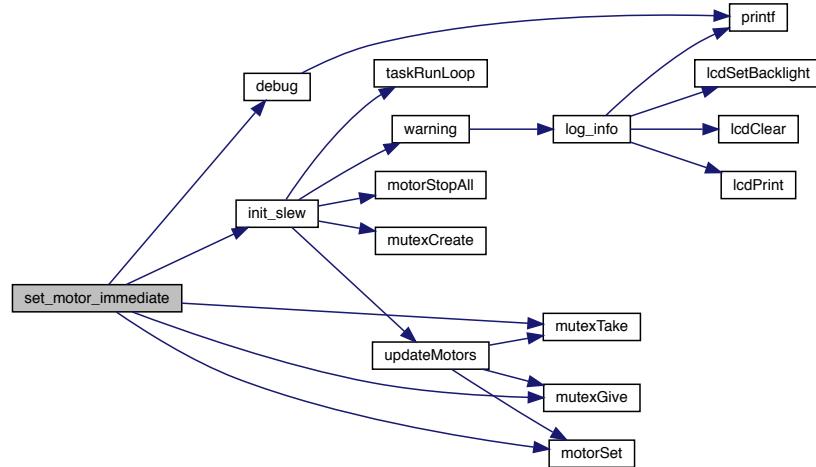
References `debug()`, `init_slew()`, `initialized`, `motors_curr_speeds`, `motors_set_speeds`, `motorSet()`, `mutexGive()`, `mutexTake()`, and `speeds_mutex`.

Referenced by `claw_grab_cone()`, `claw_release_cone()`, `set_claw_motor()`, `set_intake_motor()`, and `set_secondary_lifter_motors()`.

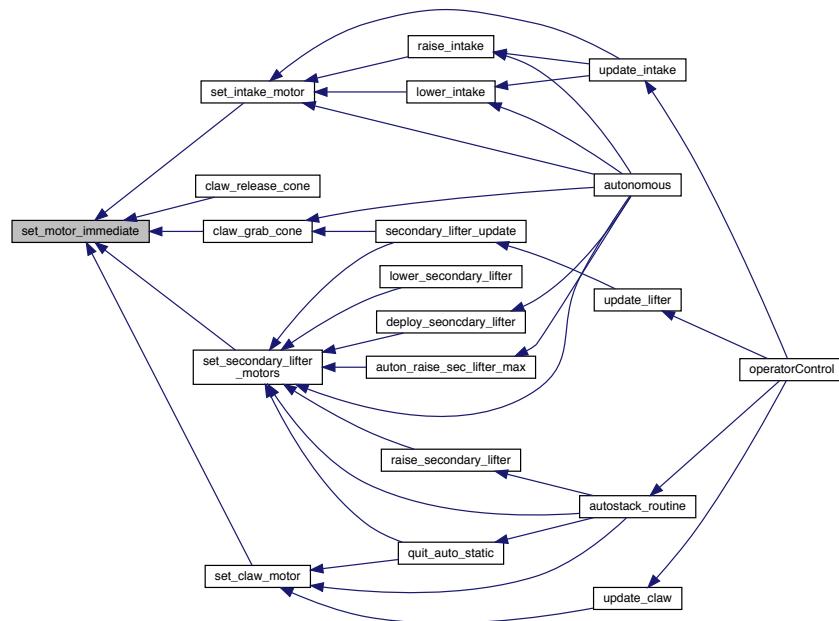
```

00090
00091     if (!initialized) {
00092         debug("Slew Not Initialized! Initializing");
00093         init_slew();
00094     }
00095     motorSet(motor, speed);
00096     mutexTake(speeds_mutex, 10);
00097     motors_curr_speeds[motor - 1] = speed;
00098     motors_set_speeds[motor - 1] = speed;
00099     mutexGive(speeds_mutex);
00100 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.43.2.4 set_motor_slew()

```
void set_motor_slew (
    int motor,
    int speed )
```

Sets motor speed wrapped inside the slew rate controller.

Parameters

<i>motor</i>	the motor port to use
<i>speed</i>	the speed to use, between -127 and 127

Author

Chris Jerrett

Date

9/14/17

Definition at line 73 of file **slew.c**.

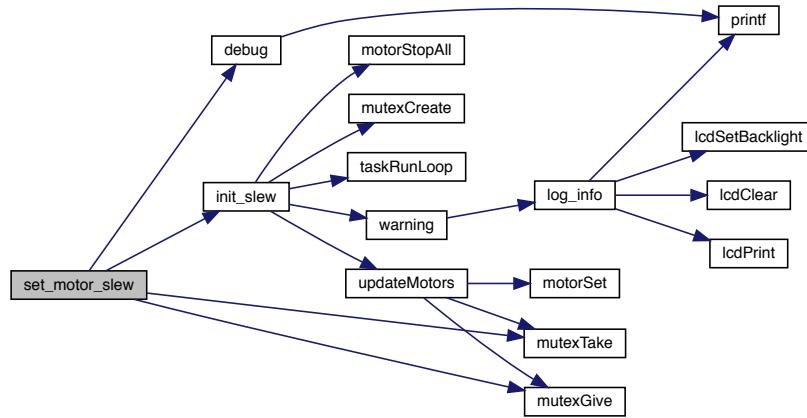
References **debug()**, **init_slew()**, **initialized**, **motors_set_speeds**, **mutexGive()**, **mutexTake()**, and **speeds_← mutex**.

Referenced by **set_main_lifter_motors()**, and **set_side_speed()**.

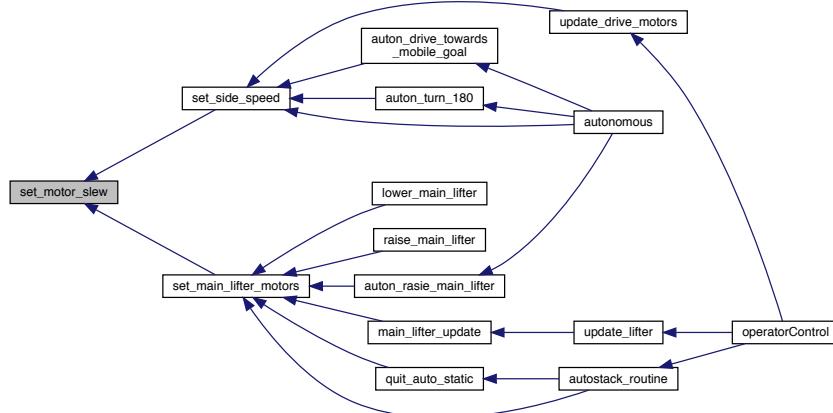
```

00073             {
00074     if (!initialized) {
00075         debug("Slew Not Initialized! Initializing");
00076         init_slew();
00077     }
00078     mutexTake(speeds_mutex, 10);
00079     motors_set_speeds[motor - 1] = speed;
00080     mutexGive(speeds_mutex);
00081 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.43.2.5 updateMotors()

```
void updateMotors ( )
```

Closes the distance between the desired motor value and the current motor value by half for each motor.

Author

Chris Jerrett

Date

9/14/17

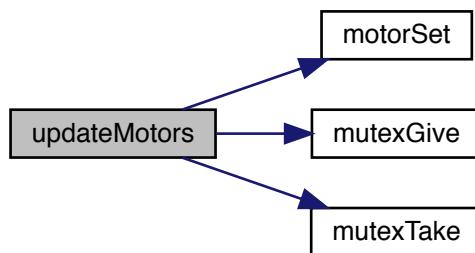
Definition at line **19** of file **slew.c**.

References **motors_curr_speeds**, **motors_set_speeds**, **motorSet()**, **mutexGive()**, **mutexTake()**, and **speeds_mutex**.

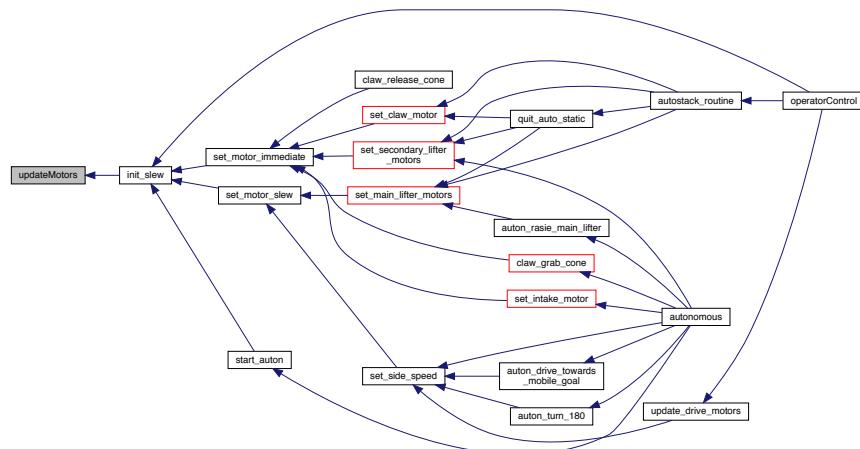
Referenced by **init_slew()**.

```
00019      {
00020      // Take back half approach
00021      // Not linear but equal to setSpeed(1-(1/2)^x)
00022      for (unsigned int i = 0; i < 9; i++) {
00023          if (motors_set_speeds[i] == motors_curr_speeds[i])
00024              continue;
00025          mutexTake(speeds_mutex, 10);
00026          int set_speed = (motors_set_speeds[i]);
00027          int curr_speed = motors_curr_speeds[i];
00028          mutexGive(speeds_mutex);
00029          int diff = set_speed - curr_speed;
00030          int offset = diff;
00031          int n = curr_speed + offset;
00032          motors_curr_speeds[i] = n;
00033          motorSet(i + 1, n);
00034      }
00035 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.44 slew.h

```

00001
00008 #ifndef _SLEW_H_
00009 #define _SLEW_H_
00010
00011 #include <API.h>
00012 #include <math.h>
00013 #include <vlib.h>
00014
00020 #define UPDATE_PERIOD_MS 25
00021
00027 #define MOTOR_PORTS 12
00028
00035 #define RAMP_PROPORTION 1
00036
00043 void updateMotors();
00044
00050 void deinitSlew();
00051
00057 void init_slew();
00058
00066 void set_motor_slew(int motor, int speed);
00067
00075 void set_motor_immediate(int motor, int speed);
00076
00077 #endif

```

6.45 include/toggle.h File Reference

Functions

- **bool buttonGetState (button_t)**
Returns the current status of a button (pressed or not pressed)
- **void buttonInit ()**
Initializes the buttons.
- **bool buttonIsNewPress (button_t)**
Detects if button is a new press from most recent check by comparing previous value to current value.

6.45.1 Function Documentation

6.45.1.1 buttonGetState()

```
bool buttonGetState (
    button_t )
```

Returns the current status of a button (pressed or not pressed)

Parameters

<i>button</i>	The button to detect from the Buttons enumeration.
---------------	--

Returns

true (pressed) or false (not pressed)

Definition at line 27 of file **toggle.c**.

References **joystickGetDigital()**, **LCD_CENT**, **LCD_LEFT**, **LCD_RIGHT**, and **IcdReadButtons()**.

Referenced by **buttonIsNewPress()**.

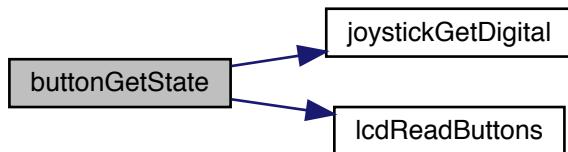
```
00027
00028     bool currentButton = false;
00029
00030     // Determine how to get the current button value (from what function) and
00031     // where it is, then get it.
00032     if (button < LCD_LEFT) {
00033         // button is a joystick button
00034         unsigned char joystick;
00035         unsigned char buttonGroup;
00036         unsigned char buttonLocation;
00037
00038         button_t newButton;
00039         if (button <= 11) {
00040             // button is on joystick 1
00041             joystick = 1;
00042             newButton = button;
00043         } else {
00044             // button is on joystick 2
00045             joystick = 2;
00046             // shift button down to joystick 1 buttons in order to
00047             // detect which button on joystick is queried
00048             newButton = (button_t)(button - 12);
00049         }
00050
00051         switch (newButton) {
00052             case 0:
00053                 buttonGroup = 5;
00054                 buttonLocation = JOY_DOWN;
00055                 break;
00056             case 1:
00057                 buttonGroup = 5;
00058                 buttonLocation = JOY_UP;
00059                 break;
00060             case 2:
00061                 buttonGroup = 6;
00062                 buttonLocation = JOY_DOWN;
00063                 break;
00064             case 3:
```

```

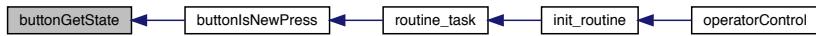
00065     buttonGroup = 6;
00066     buttonLocation = JOY_UP;
00067     break;
00068 case 4:
00069     buttonGroup = 7;
00070     buttonLocation = JOY_UP;
00071     break;
00072 case 5:
00073     buttonGroup = 7;
00074     buttonLocation = JOY_LEFT;
00075     break;
00076 case 6:
00077     buttonGroup = 7;
00078     buttonLocation = JOY_RIGHT;
00079     break;
00080 case 7:
00081     buttonGroup = 7;
00082     buttonLocation = JOY_DOWN;
00083     break;
00084 case 8:
00085     buttonGroup = 8;
00086     buttonLocation = JOY_UP;
00087     break;
00088 case 9:
00089     buttonGroup = 8;
00090     buttonLocation = JOY_LEFT;
00091     break;
00092 case 10:
00093     buttonGroup = 8;
00094     buttonLocation = JOY_RIGHT;
00095     break;
00096 case 11:
00097     buttonGroup = 8;
00098     buttonLocation = JOY_DOWN;
00099     break;
00100 default:
00101     break;
00102 }
00103 currentButton = joystickGetDigital(joystick, buttonGroup, buttonLocation);
00104 } else {
00105 // button is on LCD
00106 if (button == LCD_LEFT)
00107     currentButton = (lcdReadButtons(uart1) == LCD_BTN_LEFT);
00108
00109 if (button == LCD_CENT)
00110     currentButton = (lcdReadButtons(uart1) == LCD_BTN_CENTER);
00111
00112 if (button == LCD_RIGHT)
00113     currentButton = (lcdReadButtons(uart1) == LCD_BTN_RIGHT);
00114 }
00115 return currentButton;
00116 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.45.1.2 buttonInit()

```
void buttonInit ( )
```

Initializes the buttons.

Initializes the buttons.

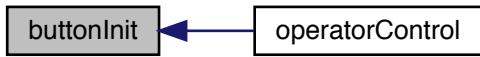
Definition at line **22** of file **toggle.c**.

References **buttonPressed**.

Referenced by **operatorControl()**.

```
00022     {
00023     for (int i = 0; i < 27; i++)
00024         buttonPressed[i] = false;
00025 }
```

Here is the caller graph for this function:



6.45.1.3 buttonIsNewPress()

```
bool buttonIsNewPress (
    button_t button )
```

Detects if button is a new press from most recent check by comparing previous value to current value.

Parameters

<i>button</i>	The button to detect from the Buttons enumeration (see include/buttons.h).
---------------	--

Returns

true or false depending on if there was a change in button state.

Parameters

<i>button</i>	The button to detect from the Buttons enumeration (see include/buttons.h).
---------------	--

Returns

true or false depending on if there was a change in button state.

Example code:

```
...
if(buttonIsNewPress(JOY1_8D))
    digitalWrite(1, !digitalRead(1));
...
```

Definition at line 136 of file **toggle.c**.

References **buttonGetState()**, and **buttonPressed**.

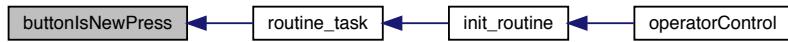
Referenced by **routine_task()**.

```
00136
00137     bool currentButton = buttonGetState(button);
00138
00139     if (!currentButton) // buttons is not currently pressed
00140         buttonPressed[button] = false;
00141
00142     if (currentButton && !buttonPressed[button]) {
00143         // button is currently pressed and was not detected as being pressed during
00144         // last check
00145         buttonPressed[button] = true;
00146         return true;
00147     } else
00148         return false; // button is not pressed or was already detected
00149 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.46 toggle.h

```
00001
00009 #ifndef BUTTONS_H_
00010 #define BUTTONS_H_
00011
00012 #include "controller.h"
00013 #include <API.h>
00014
00018 void buttonInit();
00019
00030 bool buttonIsNewPress(button_t);
00031
00040 bool buttonGetState(button_t);
00041
00042 #endif
```

6.47 include/vlib.h File Reference

Contains misc helpful functions.

Functions

- void **ftoa** (float a, char *buffer, int precision)
converts a float to string.
- int **itoaa** (int a, char *buffer, int digits)
converts a int to string.
- void **reverse** (char *str, int len)
reverses a string 'str' of length 'len'

6.47.1 Detailed Description

Contains misc helpful functions.

Author

Chris Jerrett

Date

9/9/2017

Definition in file **vlib.h**.

6.47.2 Function Documentation

6.47.2.1 ftoa()

```
void ftoa ( 
    float a,
    char * buffer,
    int precision )
```

converts a float to string.

Parameters

<i>a</i>	the float
<i>buffer</i>	the string the float will be written to.
<i>precision</i>	digits after the decimal to write

Author

Christian DeSimone

Date

9/26/2017

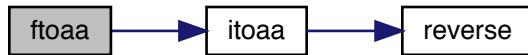
Definition at line **55** of file **vlib.c**.

References [itoaa\(\)](#).

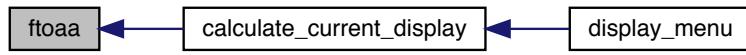
Referenced by [calculate_current_display\(\)](#).

```
00055
00056
00057 // Extract integer part
00058 int ipart = (int)a;
00059
00060 // Extract floating part
00061 float fpart = a - (float)ipart;
00062
00063 // convert integer part to string
00064 int i = itoaa(ipart, buffer, 0);
00065
00066 // check for display option after point
00067 if (precision != 0) {
00068     buffer[i] = '.';
00069     // Get the value of fraction part up to given num.
00070     // of points after dot. The third parameter is needed
00071     // to handle cases like 233.007
00072     fpart = fpart * pow(10, precision);
00073
00074     itoaa((int)fpart, buffer + i + 1, precision);
00075 }
00076 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.47.2.2 itoaa()

```
int itoaa (
    int a,
    char * buffer,
    int digits )
```

converts a int to string.

Parameters

<i>a</i>	the integer
<i>buffer</i>	the string the int will be written to.
<i>digits</i>	the number of digits to be written

Returns

the digits

Author

Chris Jerrett, Christian DeSimone

Date

9/9/2017

Definition at line **30** of file **vlib.c**.References **reverse()**.Referenced by **ftoaa()**.

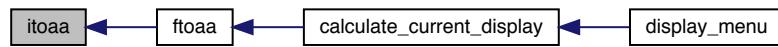
```

00030                               {
00031     int i = 0;
00032     while (a) {
00033         buffer[i++] = (a % 10) + '0';
00034         a = a / 10;
00035     }
00036
00037 // If number of digits required is more, then
00038 // add 0s at the beginning
00039     while (i < digits)
00040         buffer[i++] = '0';
00041
00042     reverse(buffer, i);
00043     buffer[i] = '\0';
00044     return i;
00045 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.47.2.3 reverse()

```
void reverse (
    char * str,
    int len )
```

reverses a string 'str' of length 'len'

Author

Chris Jerrett

Date

9/9/2017

Parameters

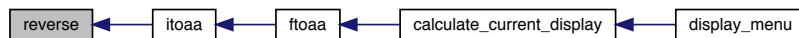
<i>str</i>	the string to reverse
<i>len</i>	the length

Definition at line **10** of file **vlib.c**.

Referenced by **itoaa()**.

```
00010
00011     int i = 0, j = len - 1, temp;
00012     while (i < j) {
00013         temp = str[i];
00014         str[i] = str[j];
00015         str[j] = temp;
00016         i++;
00017         j--;
00018     }
00019 }
```

Here is the caller graph for this function:



6.48 vlib.h

```
00001
00008 #ifndef _VLIB_H_
00009 #define _VLIB_H_
00010
00011 #include <API.h>
```

```

00012 #include <math.h>
00013 #include <string.h>
00014
00022 void reverse(char *str, int len);
00023
00034 int itoaa(int a, char *buffer, int digits);
00035
00044 void ftoaa(float a, char *buffer, int precision);
00045
00046 #endif

```

6.49 include/vmath.h File Reference

Vex Specific Math Functions, includes: Cartesian to polar coordinates.

Data Structures

- struct **cord**
A struct that contains cartesian coordinates.
- struct **polar_cord**
A struct that contains polar coordinates.

Functions

- struct **polar_cord cartesian_cord_to_polar** (struct **cord** cords)
Function to convert x and y 2 dimensional cartesian cordinated to polar coordinates.
- struct **polar_cord cartesian_to_polar** (float x, float y)
Function to convert x and y 2 dimensional cartesian coordinated to polar coordinates.
- int **max** (int a, int b)
the min of two values
- int **min** (int a, int b)
the min of two values
- double **sind** (double angle)
sine of a angle in degrees

6.49.1 Detailed Description

Vex Specific Math Functions, includes: Cartesian to polar coordinates.

Author

Christian Desimone
Chris Jerrett

Date

9/9/2017

Definition in file **vmath.h**.

6.49.2 Function Documentation

6.49.2.1 cartesian_cord_to_polar()

```
struct polar_cord cartesian_cord_to_polar (
    struct cord cords )
```

Function to convert x and y 2 dimensional cartesian cordinated to polar coordinates.

Author

Christian Desimone

Date

9/8/2017

Parameters

<code>cords</code>	the cartesian cords
--------------------	---------------------

Returns

a struct containing the angle and magnitude.

See also

polar_cord (p. 23)
cord (p. 7)

Definition at line 53 of file **vmath.c**.

References **cartesian_to_polar()**.

```
00053
00054     return cartesian_to_polar(cords.x, cords.y);
00055 }
```

{

Here is the call graph for this function:



6.49.2.2 cartesian_to_polar()

```
struct polar_cord cartesian_to_polar (
    float x,
    float y )
```

Function to convert x and y 2 dimensional cartesian coordinates to polar coordinates.

Author

Christian Desimone

Date

9/8/2017

Parameters

<i>x</i>	float value of the x cartesian coordinate.
<i>y</i>	float value of the y cartesian coordinate.

Returns

a struct containing the angle and magnitude.

See also

polar_cord (p. 23)

Definition at line 15 of file **vmath.c**.

References **polar_cord::angle**, and **polar_cord::magnitue**.

Referenced by **cartesian_cord_to_polar()**.

```
00015                                     {
00016     float degree = 0;
00017     double magnitude = sqrt((fabs(x) * fabs(x)) + (fabs(y) * fabs(y)));
00018
00019     if (x < 0) {
00020         degree += 180.0;
00021     } else if (x > 0 && y < 0) {
00022         degree += 360.0;
00023     }
00024
00025     if (x != 0 && y != 0) {
00026         degree += atan((float)y / (float)x);
00027     } else if (x == 0 && y > 0) {
00028         degree = 90.0;
00029     } else if (y == 0 && x < 0) {
00030         degree = 180.0;
00031     } else if (x == 0 && y < 0) {
00032         degree = 270.0;
00033     }
00034
00035     struct polar_cord p;
00036     p.angle = degree;
00037     p.magnitue = magnitude;
00038     return p;
00039 }
```

Here is the caller graph for this function:



6.49.2.3 max()

```
int max (
    int a,
    int b )
```

the min of two values

Parameters

a	the first
b	the second

Returns

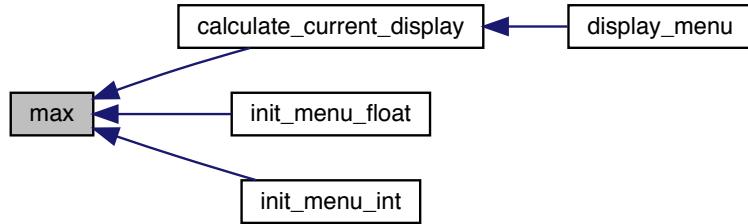
the smaller of a and b

Definition at line **83** of file **vmath.c**.

Referenced by **calculate_current_display()**, **init_menu_float()**, and **init_menu_int()**.

```
00083     {
00084     if (a > b)
00085         return a;
00086     return b;
00087 }
```

Here is the caller graph for this function:



6.49.2.4 min()

```
int min (
    int a,
    int b )
```

the min of two values

Parameters

<code>a</code>	the first
<code>b</code>	the second

Returns

the smaller of a and b

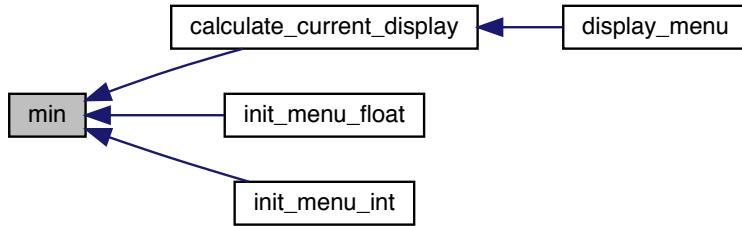
Definition at line 71 of file `vmath.c`.

Referenced by `calculate_current_display()`, `init_menu_float()`, and `init_menu_int()`.

```

00071
00072     if (a < b)
00073         return a;
00074     return b;
00075 }
```

Here is the caller graph for this function:



6.49.2.5 sind()

```
double sind (
    double angle )
```

sine of a angle in degrees

Definition at line **60** of file **vmath.c**.

```
00060             {
00061     double angleradians = angle * M_PI / 180.0f;
00062     return sin(angleradians);
00063 }
```

6.50 vmath.h

```
00001
00009 #ifndef _VMATH_H_
00010 #define _VMATH_H_
00011
00012 #include <math.h>
00017 #define M_PI 3.14159265358979323846
00018
00024 struct polar_cord {
00026     float angle;
00028     float magnitue;
00029 };
00030
00036 struct cord {
00038     float x;
00040     float y;
00041 };
00042
00055 struct polar_cord cartesian_to_polar(float x, float y);
00056
00069 struct polar_cord cartesian_cord_to_polar(struct cord cords);
00070
00077 int min(int a, int b);
00078
00085 int max(int a, int b);
00086
00090 double sind(double angle);
00091 #endif
```

6.51 README.md File Reference

6.52 README.md

```
00001 # InTheZoneA
00002 Team A code for In The Zone
```

6.53 src/auto.c File Reference

File for autonomous code.

Functions

- void **auton_drive_towards_mobile_goal** (int counts_drive, int counts_drive_left, int counts_drive_right)
Drives the robot forward until it reaches the mobile goal.
- void **auton_raise_sec_lifter_max** ()
utility function which raises the second lifter to its maximum height
- void **auton_rasie_main_lifter** ()
utility function to raise the mainlifter to the mobile goal height
- void **auton_turn_180** ()
Rotates the robot 180 degrees clockwise.
- void **autonomous** ()
Runs the user autonomous code.
- void **deploy_seoncdary_lifter** ()
utility function which deploys the secondary lifter at the start of autonomous.
- static void **setup_ime_auton** (int *counts_drive_left, int *counts_drive_right, int *counts_drive)
sets up the IMEs for the autonomous portion.
- static void **start_auton** ()
Starts the auntonomous program.

6.53.1 Detailed Description

File for autonomous code.

This file should contain the user **autonomous()** (p. 180) function and any functions related to it.

Any copyright is dedicated to the Public Domain. <http://creativecommons.org/publicdomain/zero/1.0/>

PROS contains FreeRTOS (<http://www.freertos.org>) whose source code may be obtained from <http://sourceforge.net/projects/freertos/files/> or on request.

Definition in file **auto.c**.

6.53.2 Function Documentation

6.53.2.1 auton_drive_towards_mobile_goal()

```
void auton_drive_towards_mobile_goal (
    int counts_drive,
    int counts_drive_left,
    int counts_drive_right )
```

Drives the robot forward until it reaches the mobile goal.

Author

Christian DeSimone, Chris Jerrett

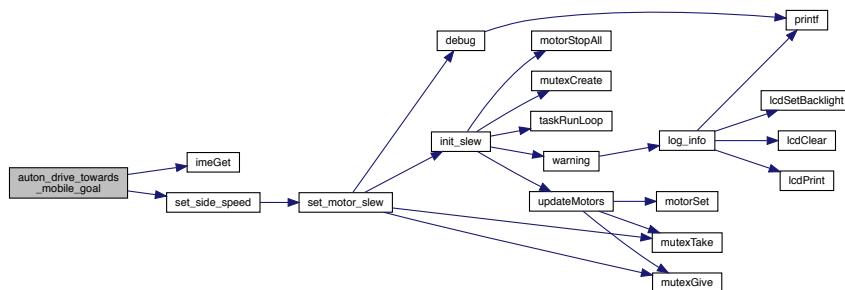
Definition at line 83 of file **auto.c**.

References **BOTH**, **imeGet()**, and **set_side_speed()**.

Referenced by **autonomous()**.

```
00084
00085     while (counts_drive < MOBILE_GOAL_DISTANCE) {
00086         set_side_speed(BOTH, 127);
00087         // Restablish the distance traveled
00088         imeGet(MID_LEFT_DRIVE, &counts_drive_left);
00089         imeGet(MID_RIGHT_DRIVE, &counts_drive_right);
00090         counts_drive = counts_drive_left + counts_drive_right;
00091         counts_drive /= 2;
00092     }
00093 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.53.2.2 auton_raise_sec_lifter_max()

`void auton_raise_sec_lifter_max ()`

utility function which raises the second lifter to its maximum height

Author

Chris Jerrett, Christian DeSimone

See also

`MAX_HEIGHT`

Definition at line **62** of file `auto.c`.

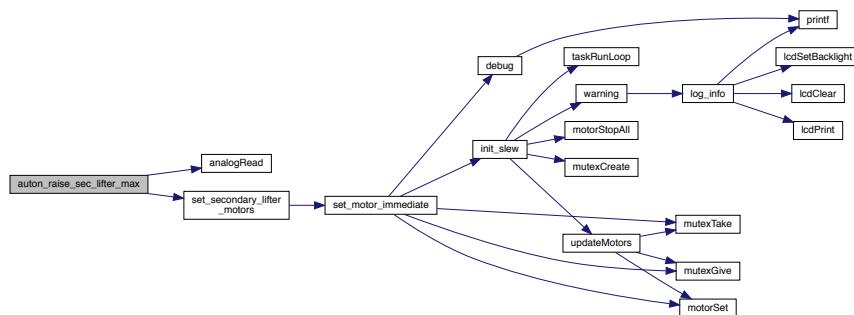
References `analogRead()`, and `set_secondary_lifter_motors()`.

Referenced by `autonomous()`.

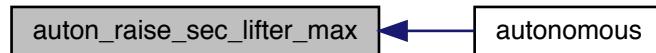
```

00062     {
00063     while (analogRead(SECONDARY_LIFTER_POT_PORT) < MAX_HEIGHT) {
00064         set_secondary_lifter_motors(MAX_SPEED);
00065     }
00066     set_secondary_lifter_motors(0);
00067 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.53.2.3 auton_rasie_main_lifter()

```
void auton_rasie_main_lifter( )
```

utility function to raise the mainlifter to the mobile goal height

Author

Chris Jerrett, Christian DeSimone

Definition at line **73** of file **auto.c**.

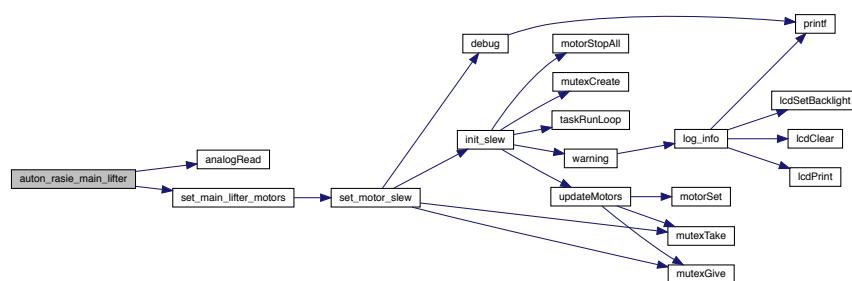
References **analogRead()**, and **set_main_lifter_motors()**.

Referenced by **autonomous()**.

```

00073 {
00074     while (analogRead(MAIN_LIFTER_POT) < MOBILE_GOAL_HEIGHT) {
00075         set_main_lifter_motors(MAX_SPEED);
00076     }
00077     set_main_lifter_motors(0);
00078 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.53.2.4 auton_turn_180()

```
void auton_turn_180( )
```

Rotates the robot 180 degrees clockwise.

Author

Chris Jerrett, Christian DeSimone

Definition at line **98** of file **auto.c**.

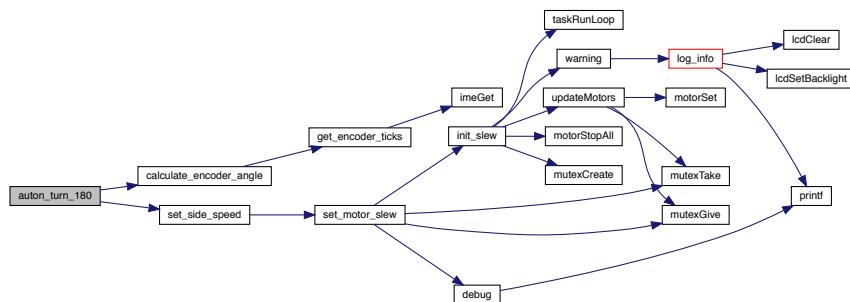
References **BOTH**, **calculate_encoder_angle()**, **LEFT**, **RIGHT**, and **set_side_speed()**.

Referenced by **autonomous()**.

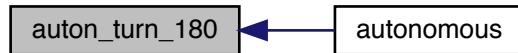
```

00098
00099     int ang = 0;
00100     while (ang < HALF_ROTATE) {
00101         ang += calculate_encoder_angle();
00102         set_side_speed(LEFT, MAX_SPEED);
00103         set_side_speed(RIGHT, MIN_SPEED);
00104     }
00105     set_side_speed(BOTH, 0);
00106 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.53.2.5 autonomous()

```
void autonomous ( )
```

Runs the user autonomous code.

This function will be started in its own task with the default priority and stack size whenever the robot is enabled via the Field Management System or the VEX Competition Switch in the autonomous mode. If the robot is disabled or communications is lost, the autonomous task will be stopped by the kernel. Re-enabling the robot will restart the task, not re-start it from where it left off.

Code running in the autonomous task cannot access information from the VEX Joystick. However, the autonomous function can be invoked from another task if a VEX Competition Switch is not available, and it can access joystick information if called in this way.

The autonomous task may exit, unlike **operatorControl()** (p. 110) which should never exit. If it does so, the robot will await a switch to another mode or disable/enable cycle.

Definition at line 125 of file **auto.c**.

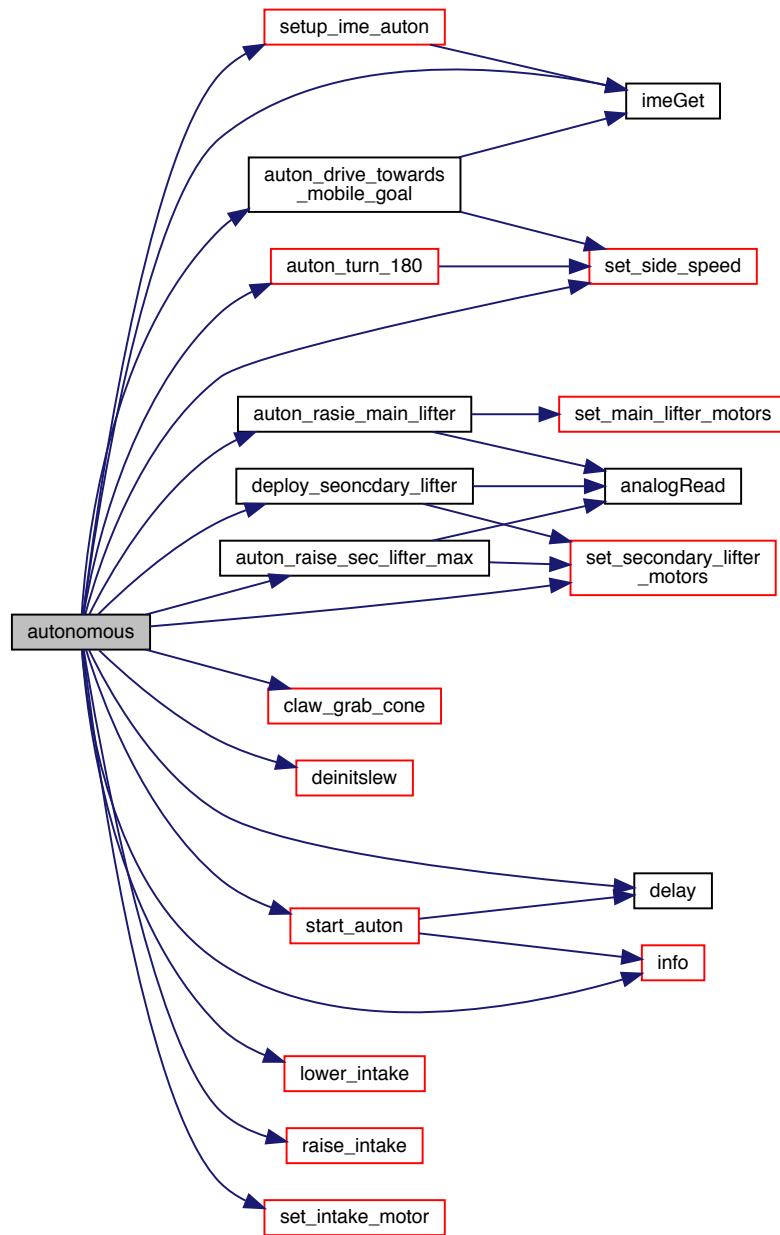
References **auton_drive_towards_mobile_goal()**, **auton_raise_sec_lifter_max()**, **auton_rasie_main_lifter()**, **auton_turn_180()**, **BOTH**, **claw_grab_cone()**, **deinitSlew()**, **delay()**, **deploy_seoncdary_lifter()**, **imeGet()**, **info()**, **lower_intake()**, **raise_intake()**, **set_intake_motor()**, **set_secondary_lifter_motors()**, **set_side_speed()**, **setup_imé_auton()**, and **start_auton()**.

```

00125         {
00126     start_auton();
00127
00128     // How far the left wheels have gone
00129     int counts_drive_left;
00130     // How far the right wheels have gone
00131     int counts_drive_right;
00132     // The average distance traveled forward
00133     int counts_drive;
00134
00135     // Reset the integrated motor controllers
00136     setup_imé_auton(&counts_drive_left, &counts_drive_right, &counts_drive);
00137     info("break 0");
00138     // Deploy claw
00139     deploy_seoncdary_lifter();
00140     info("break 1");
00141
00142     info("break 2");
00143     set_secondary_lifter_motors(0);
  
```

```
00144 // Grab pre-load cone
00145 delay(300);
00147
00148 auton_raise_sec_lifter_max();
00149 // Raise the lifter
00150 auton_rasie_main_lifter();
00151 // Drive towards the goal
00152
00153 lower_intake();
00154 delay(300);
00155 set_intake_motor(0);
00156
00157 auton_drive_towards_mobile_goal(counts_drive, counts_drive_left,
00158                                     counts_drive_right);
00159 // Stop moving
00160 set_side_speed(BOTH, 0);
00161 delay(1000);
00162
00163 raise_intake();
00164 delay(300);
00165 set_intake_motor(0);
00166
00167 // Drop the cone on the goal
00168 claw_grab_cone();
00169 delay(1000);
00170
00171 auton_turn_180();
00172
00173 counts_drive = 0;
00174
00175 while (counts_drive < MOBILE_GOAL_DISTANCE + ZONE_DISTANCE) {
00176     set_side_speed(BOTH, 127);
00177     // Restablish the distance traveled
00178     imeGet(MID_LEFT_DRIVE, &counts_drive_left);
00179     imeGet(MID_RIGHT_DRIVE, &counts_drive_right);
00180     counts_drive = counts_drive_left + counts_drive_right;
00181     counts_drive /= 2;
00182 }
00183
00184 lower_intake();
00185 delay(300);
00186 set_intake_motor(0);
00187
00188 set_side_speed(BOTH, MIN_SPEED);
00189 delay(1000);
00190 set_side_speed(BOTH, 0);
00191
00192 deinitsllew();
00193 }
```

Here is the call graph for this function:



6.53.2.6 `deploy_seoncdary_lifter()`

```
void deploy_seoncdary_lifter( )
```

utility function which deploys the secondary lifter at the start of autonomous.

Author

Christian DeSimone, Chris Jerrett

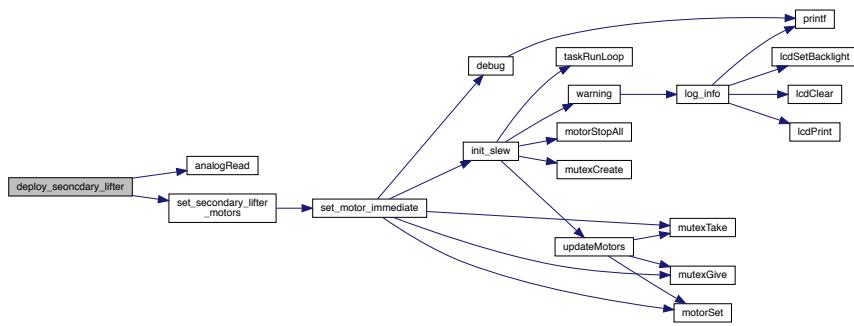
Definition at line **50** of file **auto.c**.

References **analogRead()**, and **set_secondary_lifter_motors()**.

Referenced by **autonomous()**.

```
00050      {
00051      while (analogRead(SECONDARY_LIFTER_POT_PORT) < DEPLOY_HEIGHT) {
00052          set_secondary_lifter_motors(MAX_SPEED);
00053      }
00054      set_secondary_lifter_motors(0);
00055 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

**6.53.2.7 setup_ime_auton()**

```
static void setup_ime_auton (
    int * counts_drive_left,
    int * counts_drive_right,
    int * counts_drive ) [inline], [static]
```

sets up the IMEs for the autonomous portion.

Author

Christian DeSimone, Chris Jerrett

Parameters

<code>counts_drive_left</code>	The encoder value from the left motors
<code>counts_drive_right</code>	The encoder value from the right motors
<code>counts_drive</code>	The average encoder value from both sides

Definition at line 24 of file **auto.c**.

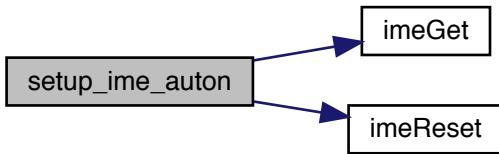
References **imeGet()**, and **imeReset()**.

Referenced by **autonomous()**.

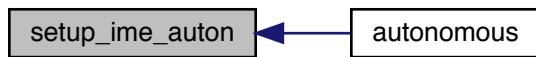
```

00025
00026     imeReset(MID_LEFT_DRIVE);
00027     imeReset(MID_RIGHT_DRIVE);
00028 // Set initial values for how far the wheels have gone
00029     imeGet(MID_LEFT_DRIVE, counts_drive_left);
00030     imeGet(MID_RIGHT_DRIVE, counts_drive_right);
00031     *counts_drive = *counts_drive_left + *counts_drive_right;
00032     *counts_drive /= 2;
00033 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.53.2.8 start_autom()

```
static void start_autom( ) [inline], [static]
```

Starts the autonomous program.

Author

Chris Jerrett, Christian DeSimone

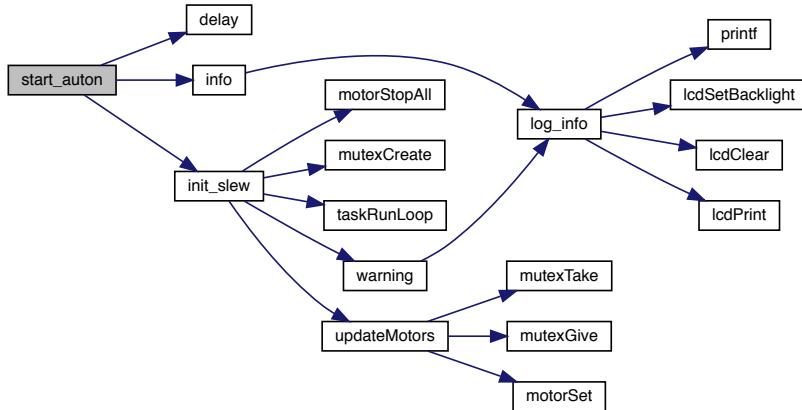
Definition at line **39** of file **auto.c**.

References **delay()**, **info()**, and **init_slew()**.

Referenced by **autonomous()**.

```
00039         { // starts the slew rate controller to prevent ptc trips
00040     init_slew();
00041
00042     delay(10);
00043     info("AUTO");
00044 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.54 auto.c

```

00001
00013 #include "auto.h"
00014 #include "log.h"
00015 #include "slew.h"
00016
00024 static inline void setup_ime_auton(int *counts_drive_left,
00025                                     int *counts_drive_right, int *counts_drive) {
00026     imeReset(MID_LEFT_DRIVE);
00027     imeReset(MID_RIGHT_DRIVE);
00028     // Set initial values for how far the wheels have gone
00029     imeGet(MID_LEFT_DRIVE, counts_drive_left);
00030     imeGet(MID_RIGHT_DRIVE, counts_drive_right);
00031     *counts_drive = *counts_drive_left + *counts_drive_right;
00032     *counts_drive /= 2;
00033 }
00038 static inline void
00039 start_auton() { // starts the slew rate controller to prevent ptc trips
00040     init_slew();
00041
00042     delay(10);
00043     info("AUTO");
00044 }
00045
00050 void deploy_seondary_lifter() {
00051     while (analogRead(SECONDARY_LIFTER_POT_PORT) < DEPLOY_HEIGHT) {
00052         set_secondary_lifter_motors(MAX_SPEED);
00053     }
00054     set_secondary_lifter_motors(0);
00055 }
00056
00062 void auton_raise_sec_lifter_max() {
00063     while (analogRead(SECONDARY_LIFTER_POT_PORT) < MAX_HEIGHT) {
00064         set_secondary_lifter_motors(MAX_SPEED);
00065     }
00066     set_secondary_lifter_motors(0);
00067 }
00068
00073 void auton_rasie_main_lifter() {
00074     while (analogRead(MAIN_LIFTER_POT) < MOBILE_GOAL_HEIGHT) {
00075         set_main_lifter_motors(MAX_SPEED);
00076     }
00077     set_main_lifter_motors(0);
00078 }
00083 void auton_drive_towards_mobile_goal(int counts_drive, int counts_drive_left,
00084                                         int counts_drive_right) {
00085     while (counts_drive < MOBILE_GOAL_DISTANCE) {
00086         set_side_speed(BOTH, 127);
00087         // Restablish the distance traveled
00088         imeGet(MID_LEFT_DRIVE, &counts_drive_left);
00089         imeGet(MID_RIGHT_DRIVE, &counts_drive_right);
00090         counts_drive = counts_drive_left + counts_drive_right;
00091         counts_drive /= 2;
00092     }
00093 }
00098 void auton_turn_180() {
00099     int ang = 0;
00100     while (ang < HALF_ROTATE) {
00101         ang += calculate_encoder_angle();
00102         set_side_speed(LEFT, MAX_SPEED);
00103         set_side_speed(RIGHT, MIN_SPEED);
00104     }
00105     set_side_speed(BOTH, 0);
00106 }
00107
00108 /*
00109  * Runs the user autonomous code. This function will be started in its own task
00110  * with the default priority and stack size whenever the robot is enabled via
00111  * the Field Management System or the VEX Competition Switch in the autonomous
00112  * mode. If the robot is disabled or communications is lost, the autonomous
00113  * task will be stopped by the kernel. Re-enabling the robot will restart the
00114  * task, not re-start it from where it left off.
00115 *
00116  * Code running in the autonomous task cannot access information from the VEX
00117  * Joystick. However, the autonomous function can be invoked from another task
00118  * if a VEX Competition Switch is not available, and it can access joystick
00119  * information if called in this way.
00120 *
00121  * The autonomous task may exit, unlike operatorControl() which should never

```

```

00122 * exit. If it does so, the robot will await a switch to another mode or
00123 * disable/enable cycle.
00124 */
00125 void autonomous() {
00126     start_auton();
00127
00128     // How far the left wheels have gone
00129     int counts_drive_left;
00130     // How far the right wheels have gone
00131     int counts_drive_right;
00132     // The average distance traveled forward
00133     int counts_drive;
00134
00135     // Reset the integrated motor controllers
00136     setup_ime_auton(&counts_drive_left, &counts_drive_right, &counts_drive);
00137     info("break 0");
00138     // Deploy claw
00139     deploy_secondary_lifter();
00140     info("break 1");
00141
00142     info("break 2");
00143     set_secondary_lifter_motors(0);
00144
00145     // Grab pre-load cone
00146     delay(300);
00147
00148     auton_raise_sec_lifter_max();
00149     // Raise the lifter
00150     auton_rasie_main_lifter();
00151     // Drive towards the goal
00152
00153     lower_intake();
00154     delay(300);
00155     set_intake_motor(0);
00156
00157     auton_drive_towards_mobile_goal(counts_drive, counts_drive_left,
00158                                     counts_drive_right);
00159     // Stop moving
00160     set_side_speed(BOTH, 0);
00161     delay(1000);
00162
00163     raise_intake();
00164     delay(300);
00165     set_intake_motor(0);
00166
00167     // Drop the cone on the goal
00168     claw_grab_cone();
00169     delay(1000);
00170
00171     auton_turn_180();
00172
00173     counts_drive = 0;
00174
00175     while (counts_drive < MOBILE_GOAL_DISTANCE + ZONE_DISTANCE) {
00176         set_side_speed(BOTH, 127);
00177         // Restablish the distance traveled
00178         imeGet(MID_LEFT_DRIVE, &counts_drive_left);
00179         imeGet(MID_RIGHT_DRIVE, &counts_drive_right);
00180         counts_drive = counts_drive_left + counts_drive_right;
00181         counts_drive /= 2;
00182     }
00183
00184     lower_intake();
00185     delay(300);
00186     set_intake_motor(0);
00187
00188     set_side_speed(BOTH, MIN_SPEED);
00189     delay(1000);
00190     set_side_speed(BOTH, 0);
00191
00192     deinitslew();
00193 }

```

6.55 src/battery.c File Reference

Functions

- double **backup_battery_voltage ()**

- **bool battery_level_acceptable ()**
gets the backup battery voltage
- **double main_battery_voltage ()**
returns if the batteries are acceptable
- **double main_battery_voltage ()**
gets the main battery voltage

6.55.1 Function Documentation

6.55.1.1 backup_battery_voltage()

```
double backup_battery_voltage ( )
```

gets the backup battery voltage

Author

Chris Jerrett

Definition at line 14 of file **battery.c**.

References **powerLevelBackup()**.

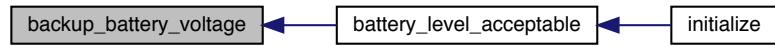
Referenced by **battery_level_acceptable()**.

```
00014 { return powerLevelBackup() / 1000.0; }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.55.1.2 battery_level_acceptable()

```
bool battery_level_acceptable ( )
```

returns if the batteries are acceptable

See also

[MIN_MAIN_VOLTAGE](#)
[MIN_BACKUP_VOLTAGE](#)

Author

Chris Jerrett

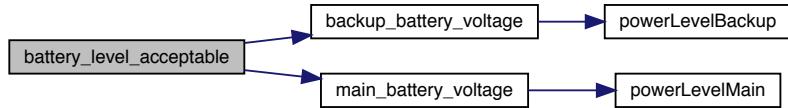
Definition at line **23** of file **battery.c**.

References **backup_battery_voltage()**, and **main_battery_voltage()**.

Referenced by **initialize()**.

```
00023     {
00024     if (main_battery_voltage() < MIN_MAIN_VOLTAGE)
00025         return false;
00026     if (backup_battery_voltage() < MIN_BACKUP_VOLTAGE)
00027         return false;
00028     return true;
00029 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.55.1.3 main_battery_voltage()

```
double main_battery_voltage ( )
```

gets the main battery voltage

Author

Chris Jerrett

Definition at line **8** of file **battery.c**.

References **powerLevelMain()**.

Referenced by **battery_level_acceptable()**.

```
00008 { return powerLevelMain() / 1000.0; }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.56 battery.c

```
00001 #include "battery.h"
00002 #include <API.h>
00003
00008 double main_battery_voltage() { return powerLevelMain() / 1000.0; }
00009
00014 double backup_battery_voltage() { return powerLevelBackup() / 1000.0; }
00015
00023 bool battery_level_acceptable() {
00024     if (main_battery_voltage() < MIN_MAIN_VOLTAGE)
00025         return false;
00026     if (backup_battery_voltage() < MIN_BACKUP_VOLTAGE)
00027         return false;
00028     return true;
00029 }
```

6.57 src/claw.c File Reference

Functions

- void **claw_grab_cone ()**
Drives the motors to grab a cone.
- void **claw_release_cone ()**
Drives the motors to release a cone.
- void **set_claw_motor (const int v)**
sets the claw motor speed
- void **update_claw ()**
Updates the claw motor values.

Variables

- bool **lifter_autostack_running**
- static enum **claw_state state = CLAW_NEUTRAL_STATE**

6.57.1 Function Documentation

6.57.1.1 claw_grab_cone()

```
void claw_grab_cone ( )
```

Drives the motors to grab a cone.

Drives the motors to open the claw.

Author

Chris Jerrett

See also

[CLAW_MOTOR](#)
[MAX_CLAW_SPEED](#)

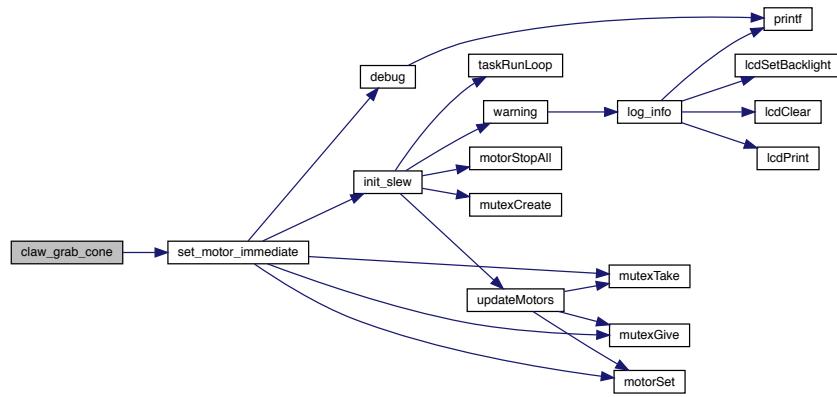
Definition at line **50** of file **claw.c**.

References [set_motor_immediate\(\)](#).

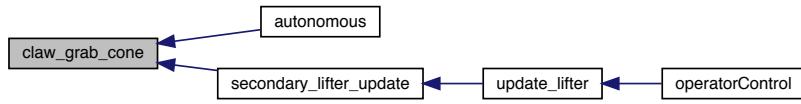
Referenced by [autonomous\(\)](#), and [secondary_lifter_update\(\)](#).

```
00050 { set_motor_immediate(CLAW_MOTOR, MAX_CLAW_SPEED); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.57.1.2 claw_release_cone()

```
void claw_release_cone( )
```

Drives the motors to release a cone.

Drives the motors to close the claw.

Author

Chris Jerrett

See also

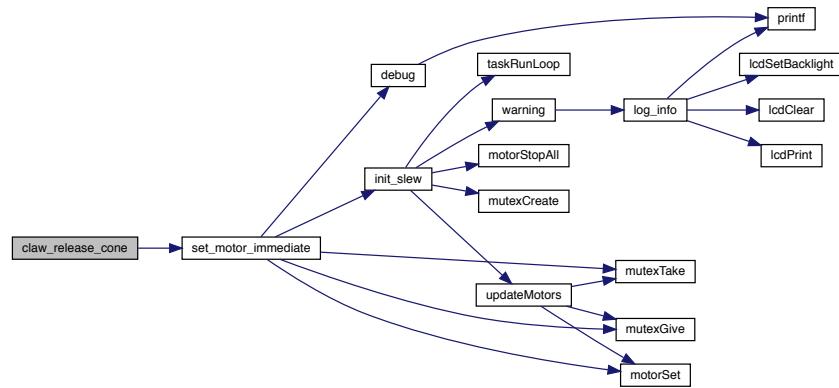
`CLAW_MOTOR`
`MIN_CLAW_SPEED`

Definition at line **58** of file `claw.c`.

References `set_motor_immediate()`.

```
00058 { set_motor_immediate(CLAW_MOTOR, MIN_CLAW_SPEED); }
```

Here is the call graph for this function:

**6.57.1.3 set_claw_motor()**

```
void set_claw_motor (
    const int v )
```

sets the claw motor speed

Author

Chris Jerrett

See also

CLAW_MOTOR

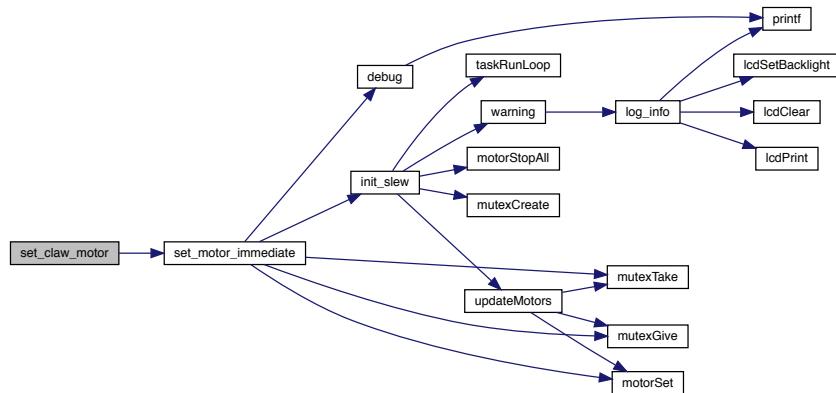
Definition at line 42 of file **claw.c**.

References **set_motor_immediate()**.

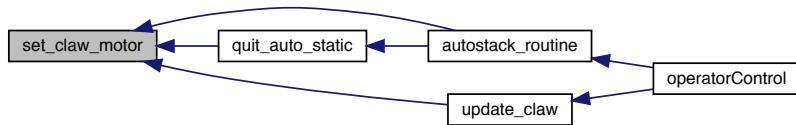
Referenced by **autostack_routine()**, **quit_auto_static()**, and **update_claw()**.

```
00042 { set_motor_immediate(CLAW_MOTOR, v); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.57.1.4 update_claw()

```
void update_claw ( )
```

Updates the claw motor values.

Author

Chris Jerrett

See also

- [CLAW_CLOSE](#)
- [CLAW_CLOSE_STATE](#) (p. 30)
- [CLAW_OPEN](#)
- [CLAW_OPEN_STATE](#) (p. 30)
- [CLAW_NEUTRAL_STATE](#) (p. 30)
- [MAX_CLAW_SPEED](#)
- [MIN_CLAW_SPEED](#)

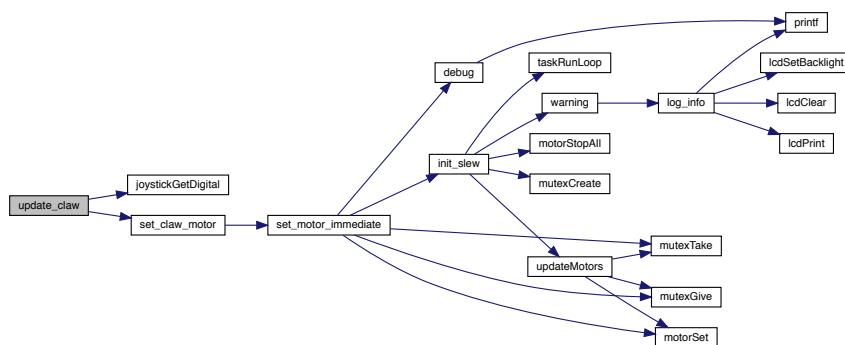
Definition at line 17 of file `claw.c`.

References `CLAW_CLOSE_STATE`, `CLAW_NEUTRAL_STATE`, `CLAW_OPEN_STATE`, `joystickGetDigital()`, `lifter_autostack_running`, `set_claw_motor()`, and `state`.

Referenced by `operatorControl()`.

```
00017      {
00018  if (lifter_autostack_running)
00019    return;
00020  if (joystickGetDigital(CLAW_CLOSE)) {
00021    state = CLAW_CLOSE_STATE;
00022  } else if (joystickGetDigital(CLAW_OPEN)) {
00023    state = CLAW_OPEN_STATE;
00024  } else {
00025    state = CLAW_NEUTRAL_STATE;
00026  }
00027
00028  if (state == CLAW_CLOSE_STATE) {
00029    set_claw_motor(MAX_CLAW_SPEED);
00030  } else if (state == CLAW_OPEN_STATE) {
00031    set_claw_motor(MIN_CLAW_SPEED);
00032  } else {
00033    set_claw_motor(0);
00034  }
00035 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.57.2 Variable Documentation

6.57.2.1 lifter_autostack_running

bool lifter_autostack_running

Definition at line 5 of file **lifter.c**.

Referenced by **autostack_routine()**, **main_lifter_update()**, **quit_auto_static()**, **secondary_lifter_update()**, and **update_claw()**.

6.57.2.2 state

enum **claw_state** state = **CLAW_NEUTRAL_STATE** [static]

Definition at line 2 of file **claw.c**.

Referenced by **update_claw()**.

6.58 claw.c

```

00001 #include "claw.h"
00002 static enum claw_state state = CLAW_NEUTRAL_STATE;
00003
00004 extern bool lifter_autostack_running;
00005
00017 void update_claw() {
00018     if (lifter_autostack_running)
00019         return;
00020     if (joystickGetDigital(CLAW_CLOSE)) {
00021         state = CLAW_CLOSE_STATE;
00022     } else if (joystickGetDigital(CLAW_OPEN)) {
00023         state = CLAW_OPEN_STATE;
00024     } else {
00025         state = CLAW_NEUTRAL_STATE;
00026     }
00027
00028     if (state == CLAW_CLOSE_STATE) {
00029         set_claw_motor(MAX_CLAW_SPEED);
00030     } else if (state == CLAW_OPEN_STATE) {
00031         set_claw_motor(MIN_CLAW_SPEED);
00032     } else {
00033         set_claw_motor(0);
00034     }
00035 }
00036
00042 void set_claw_motor(const int v) { set_motor_immediate(CLAW_MOTOR, v); }
00043
00050 void claw_grab_cone() { set_motor_immediate(CLAW_MOTOR, MAX_CLAW_SPEED); }
00051
00058 void claw_release_cone() { set_motor_immediate(CLAW_MOTOR, MIN_CLAW_SPEED); }
  
```

6.59 src/controller.c File Reference

Functions

- struct **cord get_joystick_cord** (enum **joystick side**, int controller)

Gets the location of a joystick on the controller.

6.59.1 Function Documentation

6.59.1.1 get_joystick_cord()

```
struct cord get_joystick_cord (
    enum joystick side,
    int controller )
```

Gets the location of a joystick on the controller.

Author

Chris Jerrett

See also

RIGHT_JOY (p. 38)
RIGHT_JOY_X
RIGHT_JOY_Y
LEFT_JOY_X
LEFT_JOY_Y

Definition at line 12 of file **controller.c**.

References **joystickGetAnalog()**, **RIGHT_JOY**, **cord::x**, and **cord::y**.

```
00012
00013     int x;
00014     int y;
00015     // Get the joystick value for either the right or left,
00016     // depending on the mode
00017     if (side == RIGHT_JOY) {
00018         y = joystickGetAnalog(controller, RIGHT_JOY_X);
00019         x = joystickGetAnalog(controller, RIGHT_JOY_Y);
00020     } else {
00021         y = joystickGetAnalog(controller, LEFT_JOY_X);
00022         x = joystickGetAnalog(controller, LEFT_JOY_Y);
00023     }
00024     // Define a coordinate for the joystick value
00025     struct cord c;
00026     c.x = x;
00027     c.y = y;
00028     return c;
00029 }
```

Here is the call graph for this function:



6.60 controller.c

```

00001 #include "controller.h"
00002
00012 struct cord get_joystick_cord(enum joystick side, int controller) {
00013     int x;
00014     int y;
00015     // Get the joystick value for either the right or left,
00016     // depending on the mode
00017     if (side == RIGHT_JOY) {
00018         y = joystickGetAnalog(controller, RIGHT_JOY_X);
00019         x = joystickGetAnalog(controller, RIGHT_JOY_Y);
00020     } else {
00021         y = joystickGetAnalog(controller, LEFT_JOY_X);
00022         x = joystickGetAnalog(controller, LEFT_JOY_Y);
00023     }
00024     // Define a coordinate for the joystick value
00025     struct cord c;
00026     c.x = x;
00027     c.y = y;
00028     return c;
00029 }

```

6.61 src/drive.c File Reference

Functions

- int **getThresh ()**
Gets the deadzone threshold on the joystick.
- static float **joystickExp (int joystickVal)**
Applies exponential scale to a joystick value.
- void **set_side_speed (side_t side, int speed)**
sets the speed of one side of the robot.
- void **setThresh (int t)**
Sets the deadzone threshold on the joystick.
- void **update_drive_motors ()**
Updates the drive motors during teleop.

Variables

- static int **thresh = 10**

6.61.1 Function Documentation

6.61.1.1 getThresh()

```
int getThresh ( )
```

Gets the deadzone threshhold on the joystick.

Author

Christian Desimone

Definition at line **12** of file **drive.c**.

References **thresh**.

```
00012 { return thresh; }
```

6.61.1.2 joystickExp()

```
static float joystickExp (
    int joystickVal ) [static]
```

Applies exponential scale to a joystick value.

Author

Christian DeSimone, Chris Jerrett

Parameters

<i>joystickVal</i>	the analog value from the joystick
--------------------	------------------------------------

Date

9/21/2017

Definition at line **73** of file **drive.c**.

References **thresh**.

```

00073                                     {
00074     // make the offset negative if moving backwards
00075     if (abs(joystickVal) < thresh) {
00076         return 0;
00077     }
00078
00079     int offset;
00080     // Use the threshold to ensure the joystick values are significant
00081     if (joystickVal < 0) {
00082         offset = -(thresh);
00083     } else {
00084         offset = thresh;
00085     }
00086     // Apply the function (((x/10)^3)/18) + offset) * 0.8 to the joystick value
00087     return (pow(joystickVal / 10, 3) / 18 + offset) * 0.8;
00088 }
```

6.61.1.3 set_side_speed()

```
void set_side_speed (
    side_t side,
    int speed )
```

sets the speed of one side of the robot.

Author

Christian Desimone

Parameters

<i>side</i>	a side enum which indicates the size.
<i>speed</i>	the speed of the side. Can range from -127 - 127 negative being back and positive forwards

Definition at line 54 of file **drive.c**.

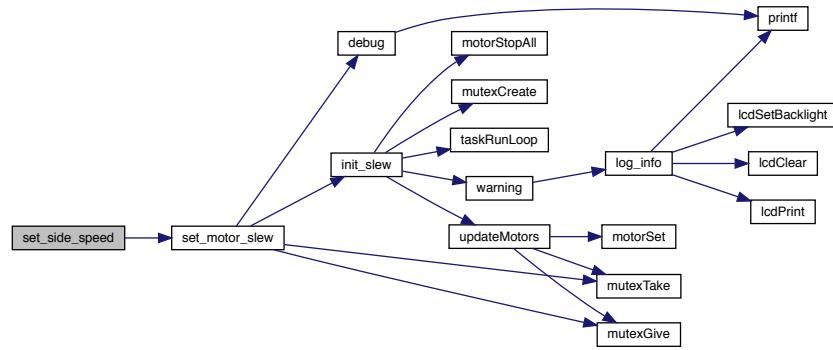
References **BOTH**, **LEFT**, **RIGHT**, and **set_motor_slew()**.

Referenced by **auton_drive_towards_mobile_goal()**, **auton_turn_180()**, **autonomous()**, and **update_drive_motors()**.

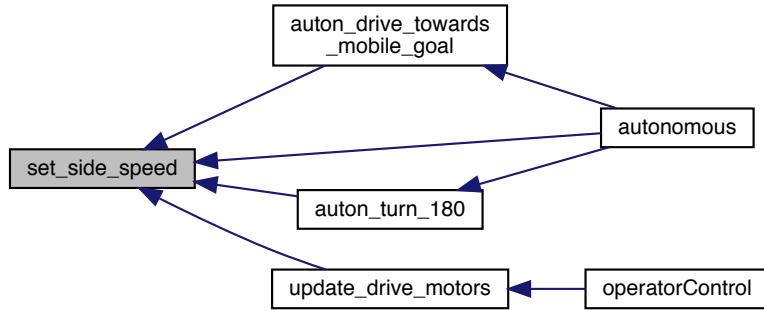
```

00054                                     {
00055     if (side == RIGHT || side == BOTH) {
00056         set_motor_slew(MOTOR_BACK_RIGHT, -speed);
00057         set_motor_slew(MOTOR_FRONT_RIGHT, -speed);
00058         set_motor_slew(MOTOR_MIDDLE_RIGHT, -speed);
00059     }
00060     if (side == LEFT || side == BOTH) {
00061         set_motor_slew(MOTOR_BACK_LEFT, speed);
00062         set_motor_slew(MOTOR_MIDDLE_LEFT, speed);
00063         set_motor_slew(MOTOR_FRONT_LEFT, speed);
00064     }
00065 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.61.1.4 setThresh()

```
void setThresh (
    int t )
```

Sets the deadzone threshhold on the joystick.

Sets the deadzone threshhold on the drive.

Author

Christian Desimone

Definition at line **18** of file **drive.c**.

References **thresh**.

```
00018 { thresh = t; }
```

6.61.1.5 update_drive_motors()

```
void update_drive_motors ( )
```

Updates the drive motors during teleop.

Author

Christian Desimone

Date

9/5/17

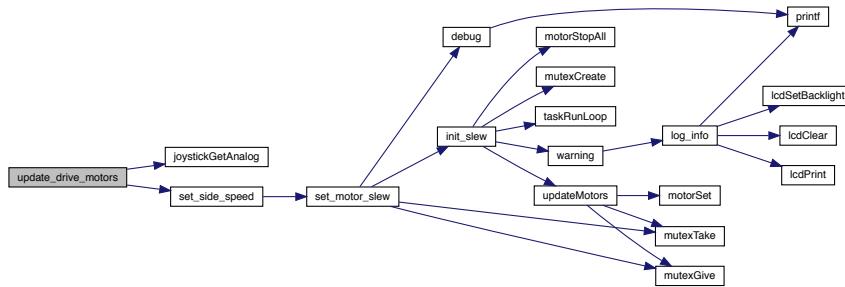
Definition at line **25** of file **drive.c**.

References **joystickGetAnalog()**, **LEFT**, **RIGHT**, **set_side_speed()**, **thresh**, **cord::x**, and **cord::y**.

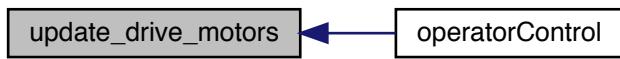
Referenced by **operatorControl()**.

```
00025           {
00026     // Get the joystick values from the controller
00027     int x = 0;
00028     int y = 0;
00029     x = -(joystickGetAnalog(MASTER, 3));
00030     y = (joystickGetAnalog(MASTER, 1));
00031     // Make sure the joystick values are significant enough to change the motors
00032     if (x < thresh && x > -thresh) {
00033       x = 0;
00034     }
00035     if (y < thresh && y > -thresh) {
00036       y = 0;
00037     }
00038     // Create motor values for the left and right from the x and y of the joystick
00039     int r = (x + y);
00040     int l = -(x - y);
00041
00042     // Set the drive motors
00043     set_side_speed(LEFT, l);
00044     set_side_speed(RIGHT, -r);
00045 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.61.2 Variable Documentation

6.61.2.1 thresh

```
int thresh = 10 [static]
```

Definition at line **6** of file **drive.c**.

Referenced by **getThresh()**, **joystickExp()**, **setThresh()**, and **update_drive_motors()**.

6.62 drive.c

```

00001 #include "drive.h"
00002 #include "controller.h"
00003 #include "motor_ports.h"
00004 #include "slew.h"
00005
00006 static int thresh = 10;
00007
00012 int getThresh() { return thresh; }
00013
00018 void setThresh(int t) { thresh = t; }
00019

```

```

00025 void update_drive_motors() {
00026     // Get the joystick values from the controller
00027     int x = 0;
00028     int y = 0;
00029     x = -(joystickGetAnalog(MASTER, 3));
00030     y = (joystickGetAnalog(MASTER, 1));
00031     // Make sure the joystick values are significant enough to change the motors
00032     if (x < thresh && x > -thresh) {
00033         x = 0;
00034     }
00035     if (y < thresh && y > -thresh) {
00036         y = 0;
00037     }
00038     // Create motor values for the left and right from the x and y of the joystick
00039     int r = (x + y);
00040     int l = -(x - y);
00041
00042     // Set the drive motors
00043     set_side_speed(LEFT, l);
00044     set_side_speed(RIGHT, -r);
00045 }
00046
00047 void set_side_speed(side_t side, int speed) {
00048     if (side == RIGHT || side == BOTH) {
00049         set_motor_slew(MOTOR_BACK_RIGHT, -speed);
00050         set_motor_slew(MOTOR_FRONT_RIGHT, -speed);
00051         set_motor_slew(MOTOR_MIDDLE_RIGHT, -speed);
00052     }
00053     if (side == LEFT || side == BOTH) {
00054         set_motor_slew(MOTOR_BACK_LEFT, speed);
00055         set_motor_slew(MOTOR_MIDDLE_LEFT, speed);
00056         set_motor_slew(MOTOR_FRONT_LEFT, speed);
00057     }
00058 }
00059
00060 static float joystickExp(int joystickVal) {
00061     // make the offset negative if moving backwards
00062     if (abs(joystickVal) < thresh) {
00063         return 0;
00064     }
00065 }
00066
00067
00068     int offset;
00069     // Use the threshold to ensure the joystick values are significant
00070     if (joystickVal < 0) {
00071         offset = -(thresh);
00072     } else {
00073         offset = thresh;
00074     }
00075     // Apply the function (((x/10)^3)/18) + offset) * 0.8 to the joystick value
00076     return (pow(joystickVal / 10, 3) / 18 + offset) * 0.8;
00077 }
00078

```

6.63 src/encoders.c File Reference

Functions

- int **get_encoder_ticks** (unsigned char address)

Gets the encoder ticks since last reset.
- int **get_encoder_velocity** (unsigned char address)

Gets the encoder reads.
- bool **init_encoders** ()

Initializes all motor encoders.

6.63.1 Function Documentation

6.63.1.1 get_encoder_ticks()

```
int get_encoder_ticks (
    unsigned char address )
```

Gets the encoder ticks since last reset.

Author

Chris Jerrett

Date

9/15/2017

Definition at line **30** of file **encoders.c**.

References **imeGet()**.

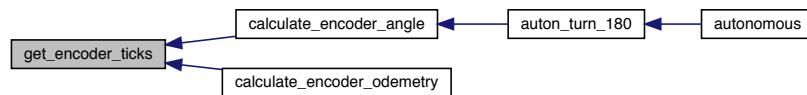
Referenced by **calculate_encoder_angle()**, and **calculate_encoder_odometry()**.

```
00030
00031     int i = 0;
00032     imeGet(address, &i);
00033     return i;
00034 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.63.1.2 get_encoder_velocity()

```
int get_encoder_velocity (
    unsigned char address )
```

Gets the encoder reads.

Author

Chris Jerrett

Date

9/15/2017

Definition at line 41 of file **encoders.c**.

References **imeGetVelocity()**.

```
00041     {
00042     int i = 0;
00043     imeGetVelocity(address, &i);
00044     return i;
00045 }
```

Here is the call graph for this function:



6.63.1.3 init_encoders()

```
bool init_encoders ( )
```

Initializes all motor encoders.

Author

Chris Jerrett

Date

9/9/2017

See also

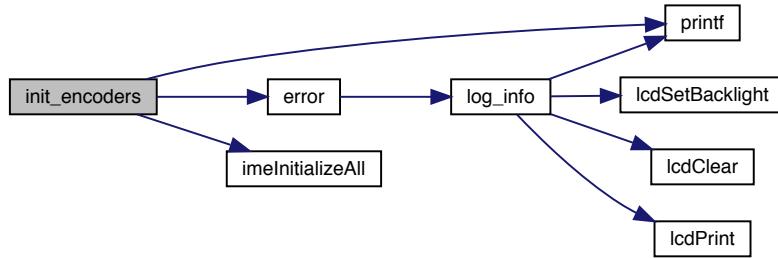
IME_NUMBER

Definition at line 11 of file **encoders.c**.References **error()**, **imeInitializeAll()**, and **printf()**.Referenced by **initialize()**.

```

00011             {
00012 #ifdef IME_NUMBER
00013     int count = imeInitializeAll();
00014     if (count != IME_NUMBER) {
00015         printf("detected only %d\n", count);
00016         error("Wrong Number of IMEs Connected");
00017         return false;
00018     }
00019     return true;
00020 #else
00021     return imeInitializeAll();
00022 #endif
00023 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.64 encoders.c

```
00001 #include "encoders.h"
00002 #include "log.h"
00003 #include <API.h>
00004
00011 bool init_encoders() {
00012 #ifdef IME_NUMBER
00013     int count = imeInitializeAll();
00014     if (count != IME_NUMBER) {
00015         printf("detected only %d\n", count);
00016         error("Wrong Number of IMEs Connected");
00017         return false;
00018     }
00019     return true;
00020 #else
00021     return imeInitializeAll();
00022 #endif
00023 }
00024
00030 int get_encoder_ticks(unsigned char address) {
00031     int i = 0;
00032     imeGet(address, &i);
00033     return i;
00034 }
00035
00041 int get_encoder_velocity(unsigned char address) {
00042     int i = 0;
00043     imeGetVelocity(address, &i);
00044     return i;
00045 }
```

6.65 src/gyro.c File Reference

Functions

- float **get_main_gyro_angluar_velocity ()**
returns the angular velocity directly from the gyro
- bool **init_main_gyro ()**
Initializes the gyro.

Variables

- static **Gyro main_gyro**

6.65.1 Function Documentation

6.65.1.1 get_main_gyro_angluar_velocity()

```
float get_main_gyro_angluar_velocity ( )
```

returns the angular velocity directly from the gyro

Gets the Gyro angular velocity.

Author

Chris Jerrett

See also

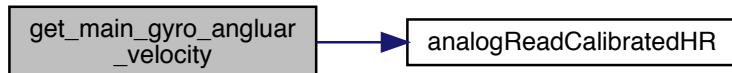
GYRO_PORT

Definition at line **20** of file **gyro.c**.

References **analogReadCalibratedHR()**.

```
00020          {  
00021     uint32_t port = GYRO_PORT;  
00022     int32_t reading = (int32_t)analogReadCalibratedHR(port + 1);  
00023     return 0;  
00024 }
```

Here is the call graph for this function:



6.65.1.2 init_main_gyro()

```
bool init_main_gyro ( )
```

Initializes the gyro.

Initializes the main robot gyroscope/ Only call function when robot still and ready to start autonomous.

Author

Chris Jerrett

See also

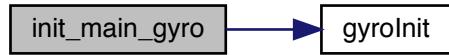
GYRO_PORT
GYRO_MULTIPLIER

Definition at line **10** of file **gyro.c**.

References **gyroInit()**, and **main_gyro**.

```
00010          {  
00011     main_gyro = gyroInit(GYRO_PORT, GYRO_MULTIPLIER);  
00012     return main_gyro != NULL;  
00013 }
```

Here is the call graph for this function:



6.65.2 Variable Documentation

6.65.2.1 main_gyro

Gyro main_gyro [static]

Definition at line **3** of file **gyro.c**.

Referenced by **init_main_gyro()**.

6.66 gyro.c

```

00001 #include "gyro.h"
00002
00003 static Gyro main_gyro;
00010 bool init_main_gyro() {
00011     main_gyro = gyroInit(GYRO_PORT, GYRO_MULTIPLIER);
00012     return main_gyro != NULL;
00013 }
00014
00020 float get_main_gyro_angluar_velocity() {
00021     uint32_t port = GYRO_PORT;
00022     int32_t reading = (int32_t)analogReadCalibratedHR(port + 1);
00023     return 0;
00024 }
```

6.67 src/init.c File Reference

File for initialization code.

Functions

- void **initialize** ()

Runs user initialization code.
- void **initializeIO** ()

Runs pre-initialization code.

Variables

- **Ultrasonic_lifter_ultrasonic**

6.67.1 Detailed Description

File for initialization code.

This file should contain the user **initialize()** (p. 208) function and any functions related to it.

Any copyright is dedicated to the Public Domain. <http://creativecommons.org/publicdomain/zero/1.0/>

PROS contains FreeRTOS (<http://www.freertos.org>) whose source code may be obtained from <http://sourceforge.net/projects/freertos/files/> or on request.

Definition in file **init.c**.

6.67.2 Function Documentation

6.67.2.1 initialize()

```
void initialize ( )
```

Runs user initialization code.

This function will be started in its own task with the default priority and stack size once when the robot is starting up. It is possible that the VEXnet communication link may not be fully established at this time, so reading from the VEX Joystick may fail.

This function should initialize most sensors (gyro, encoders, ultrasonics), LCDs, global variables, and IMEs.

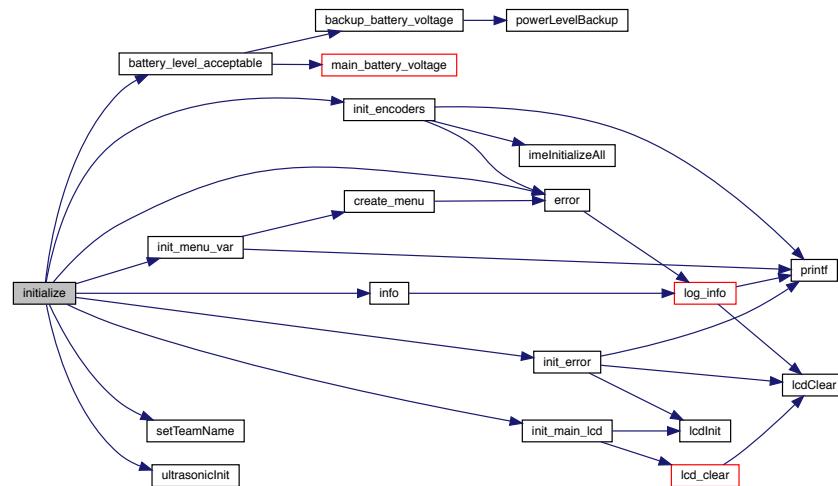
This function must exit relatively promptly, or the **operatorControl()** (p. 110) and **autonomous()** (p. 107) tasks will not start. An autonomous mode selection menu like the `pre_auton()` in other environments can be implemented in this task if desired.

Definition at line 52 of file `init.c`.

References `battery_level_acceptable()`, `error()`, `info()`, `init_encoders()`, `init_error()`, `init_main_lcd()`, `init_menu_var()`, `lifter_ultrasonic`, `setTeamName()`, `STRING_TYPE`, and `ultrasonicInit()`.

```
00052     {
00053     init_main_lcd(uart1);
00054     info("LCD Init");
00055     if (!battery_level_acceptable())
00056         error("Bad main/backup bat");
00057     menu_t *t =
00058         init_menu_var(STRING_TYPE, "TEST Menu", 5, "1", "2", "3", "4", "5");
00059     init_error(true, uart2);
00060     setTeamName("9228A");
00061     init_encoders();
00062     lifter_ultrasonic = ultrasonicInit(4, 5);
00063 }
```

Here is the call graph for this function:



6.67.2.2 initializeIO()

```
void initializeIO ( )
```

Runs pre-initialization code.

This function will be started in kernel mode one time while the VEX Cortex is starting up. As the scheduler is still paused, most API functions will fail.

The purpose of this function is solely to set the default pin modes (**pinMode()** (p. ??)) and port states (**digitalWrite()** (p. ??)) of limit switches, push buttons, and solenoids. It can also safely configure a UART port (**uartOpen()**) but cannot set up an LCD (**lcdInit()** (p. ??)).

Definition at line **36** of file **init.c**.

References **watchdogInit()**.

```
00036 { watchdogInit(); }
```

Here is the call graph for this function:



6.67.3 Variable Documentation

6.67.3.1 lifter_ultrasonic

```
Ultrasonic lifter_ultrasonic
```

Definition at line **24** of file **sensors.h**.

Referenced by **autostack_routine()**, **initialize()**, and **main_lifter_update()**.

6.68 init.c

```

00001
00013 #include "battery.h"
00014 #include "encoders.h"
00015 #include "lcd.h"
00016 #include "lifter.h"
00017 #include "log.h"
00018 #include "main.h"
00019 #include "menu.h"
00020 #include "sensors.h"
00021 #include "slew.h"
00022
00023 extern Ultrasonic lifter_ultrasonic;
00024
00025 /*
00026 * Runs pre-initialization code. This function will be started in kernel mode
00027 * one time while the VEX Cortex is starting up. As the scheduler is still
00028 * paused, most API functions will fail.
00029 *
00030 * The purpose of this function is solely to set the default pin modes
00031 * (pinMode()) and port states (digitalWrite()) of limit switches, push buttons,
00032 * and solenoids. It can also safely configure a UART port (uartOpen()) but
00033 * cannot set up an LCD (lcdInit()).
00034 *
00035 */
00036 void initializeIO() { watchdogInit(); }
00037
00038 /* @brief Initialization code to be run at startup of the cortex
00039 * @author Chris Jerrett
00040 * Runs user initialization code. This function will be started in its own task
00041 * with the default priority and stack size once when the robot is starting up.
00042 * It is possible that the VEXnet communication link may not be fully
00043 * established at this time, so reading from the VEX Joystick may fail.
00044 *
00045 * This function should initialize most sensors (gyro, encoders, ultrasonics),
00046 * LCDs, global variables, and IMEs.
00047 *
00048 * This function must exit relatively promptly, or the operatorControl() and
00049 * autonomous() tasks will not start. An autonomous mode selection menu like the
00050 * pre_automode() in other environments can be implemented in this task if desired.
00051 */
00052 void initialize() {
00053     init_main_lcd(uart1);
00054     info("LCD Init");
00055     if (!battery_level_acceptable())
00056         error("Bad main/backup bat");
00057     menu_t *t =
00058         init_menu_var(STRING_TYPE, "TEST Menu", 5, "1", "2", "3", "4", "5");
00059     init_error(true, uart2);
00060     setTeamName("9228A");
00061     init_encoders();
00062     lifter_ultrasonic = ultrasonicInit(4, 5);
00063 }

```

6.69 src/lcd.c File Reference

Functions

- **void init_main_lcd (FILE *lcd)**
Initializes the lcd screen.
- **static bool lcd_assert ()**
*Asserts the lcd is initialized Works by checking is the File *lcd_port is the default NULL value and thus not set.*
- **void lcd_clear ()**
Clears the lcd.
- **lcd_buttons lcd_get_pressed_buttons ()**
Returns the pressed buttons.
- **void lcd_print (unsigned int line, const char *str)**

- prints a string to a line on the lcd
 - void **lcd_printf** (unsigned int **line**, const char *format_str,...)
prints a formated string to a line on the lcd.
 - void **lcd_set_backlight** (bool **state**)
sets the backlight of the lcd
 - void **prompt_confirmation** (const char *confirm_text)
Prompts the user to confirm a string.

Variables

- static FILE * **lcd_port** = NULL
The port of the initialized lcd.

6.69.1 Function Documentation

6.69.1.1 init_main_lcd()

```
void init_main_lcd (
    FILE * lcd )
```

Initializes the lcd screen.

Also will initialize the lcd_port var. Must be called before any lcd function can be called.

Parameters

<i>lcd</i>	the urart port of the lcd screen
------------	----------------------------------

See also

uart1
uart2

Author

Chris Jerrett

Date

9/9/2017

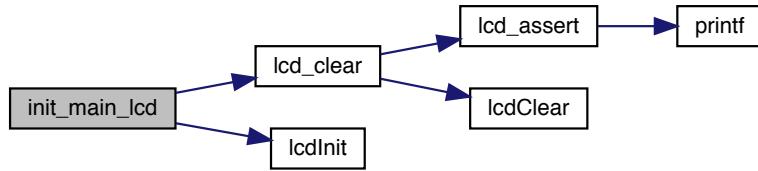
Definition at line **62** of file **lcd.c**.

References **lcd_clear()**, **lcd_port**, and **lcdInit()**.

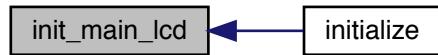
Referenced by **initialize()**.

```
00062     lcd_port = lcd;
00063     lcdInit(lcd);
00064     lcd_clear();
00065
00066 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.69.1.2 lcd_assert()

```
static bool lcd_assert ( ) [static]
```

Asserts the lcd is initialized Works by checking is the File *lcd_port is the default NULL value and thus not set.

Author

Chris Jerrett

Date

9/9/2017

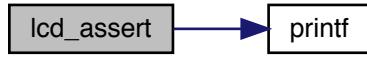
Definition at line 13 of file **lcd.c**.

References **lcd_port**, and **printf()**.

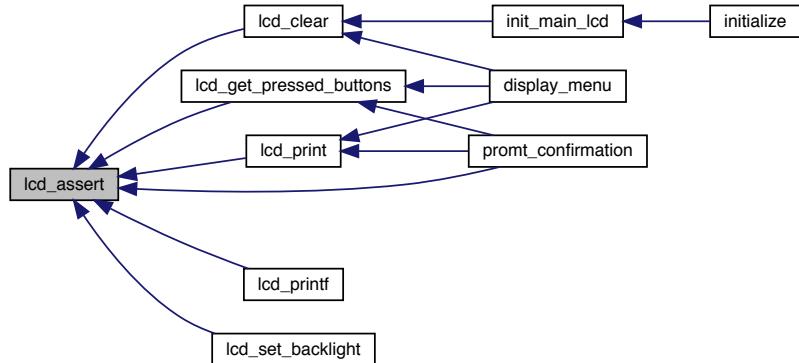
Referenced by **lcd_clear()**, **lcd_get_pressed_buttons()**, **lcd_print()**, **lcd_printf()**, **lcd_set_backlight()**, and **prompt_confirmation()**.

```
00013     {
00014     if (lcd_port == NULL) {
00015         printf("LCD NULL!");
00016         return false;
00017     }
00018     return true;
00019 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.69.1.3 lcd_clear()

```
void lcd_clear ( )
```

Clears the lcd.

Author

Chris Jerrett

Date

9/9/2017

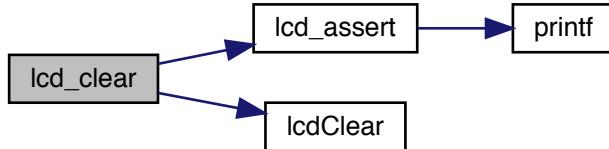
Definition at line 47 of file **lcd.c**.

References **lcd_assert()**, **lcd_port**, and **LcdClear()**.

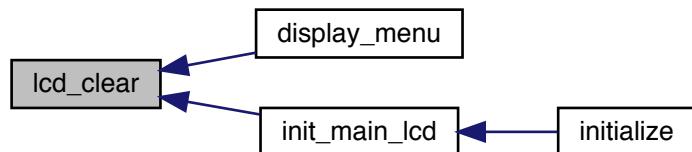
Referenced by **display_menu()**, and **init_main_lcd()**.

```
00047     {  
00048     lcd_assert();  
00049     LcdClear(lcd_port);  
00050 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.69.1.4 `lcd_get_pressed_buttons()`

```
lcd_buttons lcd_get_pressed_buttons ( )
```

Returns the pressed buttons.

Returns

a struct containing the states of all three buttons.

Author

Chris Jerrett

Date

9/9/2017

See also

[lcd_buttons](#) (p. 10)

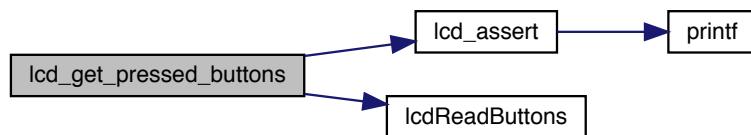
Definition at line **28** of file [lcd.c](#).

References `lcd_assert()`, `lcd_port`, `LcdReadButtons()`, `lcd_buttons::left`, `lcd_buttons::middle`, `PRESSED`, `RELEASED`, and `lcd_buttons::right`.

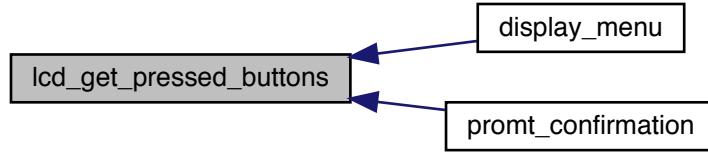
Referenced by `display_menu()`, and `prompt_confirmation()`.

```
00028                                     {
00029     lcd_assert();
00030     unsigned int btn_binary = lcdReadButtons(lcd_port);
00031     bool left = btn_binary & 0x1; // 0001
00032     bool middle = btn_binary & 0x2; // 0010
00033     bool right = btn_binary & 0x4; // 0100
00034     lcd_buttons btns;
00035     btns.left = left ? PRESSED : RELEASED;
00036     btns.middle = middle ? PRESSED : RELEASED;
00037     btns.right = right ? PRESSED : RELEASED;
00038
00039     return btns;
00040 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.69.1.5 `lcd_print()`

```
void lcd_print (
    unsigned int line,
    const char * str )
```

prints a string to a line on the lcd

Parameters

<code>line</code>	the line to print on
<code>str</code>	string to print

Author

Chris Jerrett

Date

9/9/2017

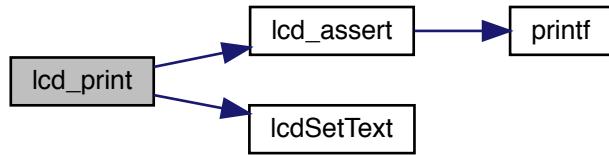
Definition at line **75** of file **lcd.c**.

References `lcd_assert()`, `lcd_port`, and `lcdSetText()`.

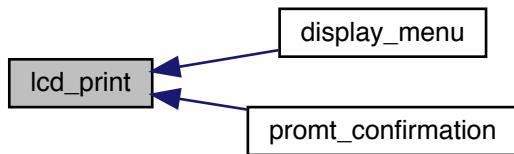
Referenced by `display_menu()`, and `prompt_confirmation()`.

```
00075
00076     lcd_assert();
00077     lcdSetText(lcd_port, line, str);
00078 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.69.1.6 `lcd_printf()`

```
void lcd_printf (
    unsigned int line,
    const char * format_str,
    ... )
```

prints a formated string to a line on the lcd.

Similar to `printf`

Parameters

<code>line</code>	the line to print on
<code>format_str</code>	format string string to print

Author

Chris Jerrett

Date

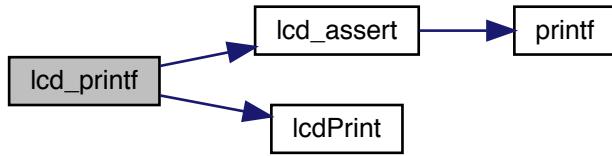
9/9/2017

Definition at line **87** of file **lcd.c**.

References **lcd_assert()**, **lcd_port**, and **lcdPrint()**.

```
00087 {  
00088     lcd_assert();  
00089     lcdPrint(lcd_port, line, format_str);  
00090 }
```

Here is the call graph for this function:



6.69.1.7 lcd_set_backlight()

```
void lcd_set_backlight (  
    bool state )
```

sets the backlight of the lcd

Parameters

<code>state</code>	a boolean representing the state of the backlight. true = on, false = off.
--------------------	--

Author

Chris Jerrett

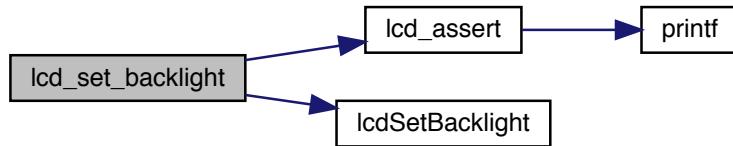
Date

9/9/2017

Definition at line **99** of file **lcd.c**.References **lcd_assert()**, **lcd_port**, and **lcdSetBacklight()**.

```
00099          {  
00100      lcd_assert();  
00101      lcdSetBacklight(lcd_port, state);  
00102 }
```

Here is the call graph for this function:

**6.69.1.8 prompt_confirmation()**

```
void prompt_confirmation (  
    const char * confirm_text )
```

Prompts the user to confirm a string.

User must press middle button to confirm. Function is not thread safe and will stall a thread.

Parameters

<code>confirm_text</code>	the text for the user to confirm.
---------------------------	-----------------------------------

Author

Chris Jerrett

Date

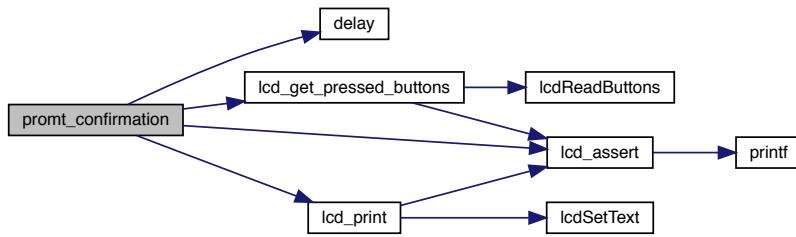
9/9/2017

Definition at line 113 of file **lcd.c**.

References **delay()**, **lcd_assert()**, **lcd_get_pressed_buttons()**, **lcd_print()**, and **PRESSED**.

```
00113     lcd_assert();  
00114     lcd_print(1, confirm_text);  
00115     while (lcd_get_pressed_buttons().middle != PRESSED) {  
00116         delay(200);  
00117     }  
00118 }  
00119 }
```

Here is the call graph for this function:



6.69.2 Variable Documentation

6.69.2.1 lcd_port

```
FILE* lcd_port = NULL [static]
```

The port of the initialized lcd.

Definition at line 4 of file **lcd.c**.

Referenced by `init_main_lcd()`, `lcd_assert()`, `lcd_clear()`, `lcd_get_pressed_buttons()`, `lcd_print()`, `lcd_printf()`, and `lcd_set_backlight()`.

6.70 lcd.c

```

00001 #include "lcd.h"
00002
00004 static FILE *lcd_port = NULL;
00005
00013 static bool lcd_assert() {
00014     if (lcd_port == NULL) {
00015         printf("LCD NULL!");
00016         return false;
00017     }
00018     return true;
00019 }
00020
00028 lcd_buttons lcd_get_pressed_buttons() {
00029     lcd_assert();
00030     unsigned int btn_binary = lcdReadButtons(lcd_port);
00031     bool left = btn_binary & 0x1; // 0001
00032     bool middle = btn_binary & 0x2; // 0010
00033     bool right = btn_binary & 0x4; // 0100
00034     lcd_buttons btns;
00035     btns.left = left ? PRESSED : RELEASED;
00036     btns.middle = middle ? PRESSED : RELEASED;
00037     btns.right = right ? PRESSED : RELEASED;
00038
00039     return btns;
00040 }
00041
00047 void lcd_clear() {
00048     lcd_assert();
00049     lcdClear(lcd_port);
00050 }
00051
00062 void init_main_lcd(FILE *lcd) {
00063     lcd_port = lcd;
00064     lcdInit(lcd);
00065     lcd_clear();
00066 }
00067
00075 void lcd_print(unsigned int line, const char *str) {
00076     lcd_assert();
00077     lcdSetText(lcd_port, line, str);
00078 }
00079
00087 void lcd_printf(unsigned int line, const char *format_str, ...) {
00088     lcd_assert();
00089     lcdPrint(lcd_port, line, format_str);
00090 }
00091
00099 void lcd_set_backlight(bool state) {
00100     lcd_assert();
00101     lcdSetBacklight(lcd_port, state);
00102 }
00103
00113 void prompt_confirmation(const char *confirm_text) {
00114     lcd_assert();
00115     lcd_print(1, confirm_text);
00116     while (lcd_get_pressed_buttons().middle != PRESSED) {
00117         delay(200);
00118     }
00119 }

```

6.71 src/lifter.c File Reference

Functions

- **void autostack_routine (void *param)**
Autostacks a cone once picked up.
- **double getLifterHeight ()**
Gets the height of the lifter in inches.
- **int getLifterTicks ()**

- Gets the value of the lifter pot.
- void **interrupt_auto_stack** (void *param)
 - Stops an autostack in case of an error.*
- float **lifterPotentiometerToDegree** (int x)
 - height of the lifter in degrees from 0 height*
- void **lower_main_lifter** ()
 - Lowers the main lifter.*
- void **lower_secondary_lifter** ()
 - Lowers the secondary lifter.*
- static void **main_lifter_update** ()
 -
- static void **quit_auto_static** ()
 -
- void **raise_main_lifter** ()
 - Raises the main lifter.*
- void **raise_secondary_lifter** ()
 - Raises the main lifter.*
- static void **secondary_lifter_update** ()
 -
- void **set_lifter_pos** (int pos)
 - Sets the lifter positions to the given value.*
- void **set_main_lifter_motors** (const int v)
 - Sets the main lifter motors to the given value.*
- void **set_secondary_lifter_motors** (const int v)
 - Sets the secondary lifter motors to the given value.*
- void **update_lifter** ()
 - Updates the lifter in teleop.*

Variables

- static bool **lifter_autostack_routine_interrupt** = false
- bool **lifter_autostack_running** = false
- static bool **secondary_override** = false

6.71.1 Function Documentation

6.71.1.1 autostack_routine()

```
void autostack_routine (
    void * param )
```

Autostacks a cone once picked up.

Parameters

<i>param</i>	ignored parameter
--------------	-------------------

Definition at line 20 of file `lifter.c`.

References `analogRead()`, `delay()`, `info()`, `lifter_autostack_routine_interrupt`, `lifter_autostack_running`, `lifter_ultrasonic`, `printf()`, `quit_auto_static()`, `raise_secondary_lifter()`, `set_claw_motor()`, `set_main_lifter_motors()`, `set_secondary_lifter_motors()`, and `ultrasonicGet()`.

Referenced by `operatorControl()`.

```

00020
00021     lifter_autostack_routine_interrupt = false;
00022     lifter_autostack_running = true;
00023     raise_secondary_lifter();
00024     while (analogRead(SECONDARY_LIFTER_POT_PORT) < 1600) {
00025         set_secondary_lifter_motors(MIN_SPEED);
00026         if (lifter_autostack_routine_interrupt) {
00027             quit_auto_static();
00028             return;
00029         }
00030         delay(50);
00031         info("1");
00032     }
00033     set_secondary_lifter_motors(0);
00034     bool lifted = false;
00035     int val = ultrasonicGet(lifter_ultrasonic);
00036     printf("%d\n", val);
00037     while (val < 10 && val != ULTRA_BAD_RESPONSE) {
00038         if (lifter_autostack_routine_interrupt) {
00039             quit_auto_static();
00040             return;
00041         }
00042         set_main_lifter_motors(MAX_SPEED);
00043         info("2");
00044         lifted = true;
00045         delay(50);
00046         val = ultrasonicGet(lifter_ultrasonic);
00047         printf("%d\n", val);
00048     }
00049     if (lifter_autostack_routine_interrupt) {
00050         quit_auto_static();
00051         return;
00052     }
00053     delay(200);
00054     if (lifted)
00055         delay(50);
00056     if (lifter_autostack_routine_interrupt) {
00057         quit_auto_static();
00058         return;
00059     }
00060     set_main_lifter_motors(0);
00061     set_secondary_lifter_motors(0);
00062
00063     while (analogRead(SECONDARY_LIFTER_POT_PORT) < 3000) {
00064         if (lifter_autostack_routine_interrupt) {
00065             quit_auto_static();
00066             return;
00067         }
00068         set_secondary_lifter_motors(MIN_SPEED);
00069         delay(50);
00070         info("3");
00071     }
00072
00073     set_main_lifter_motors(MIN_SPEED / 1.333);
00074
00075     while (val > 10) {
00076         if (lifter_autostack_routine_interrupt) {
00077             quit_auto_static();
00078             return;
00079         }
00080         info("2");
00081         lifted = true;
00082         delay(30);
00083         val = ultrasonicGet(lifter_ultrasonic);
00084         printf("%d\n", val);
00085     }
00086
00087     set_main_lifter_motors(0);

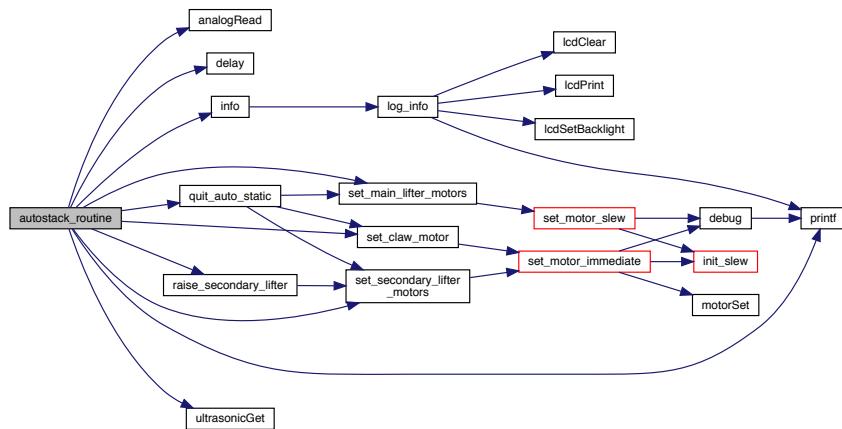
```

```

00088     set_claw_motor(MIN_CLAW_SPEED);
00089     if (lifter_autostack_routine_interrupt) {
00090         quit_auto_static();
00091         return;
00092     }
00093 }
00094 delay(500);
00095 if (lifter_autostack_routine_interrupt) {
00096     quit_auto_static();
00097     return;
00098 }
00099 set_main_lifter_motors(MAX_SPEED);
00100 if (lifter_autostack_routine_interrupt) {
00101     quit_auto_static();
00102     return;
00103 }
00104 delay(300);
00105
00106 set_main_lifter_motors(MIN_SPEED);
00107 set_claw_motor(0);
00108 set_secondary_lifter_motors(0);
00109
00110 lifter_autostack_running = false;
00111 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.71.1.2 getLifterHeight()

```
double getLifterHeight ( )
```

Gets the height of the lifter in inches.

Returns

the height of the lifter.

Author

Chris Jerrett

Date

9/17/2017

Definition at line **306** of file **lifter.c**.

References **getLifterTicks()**.

```
00306             {
00307     unsigned int ticks = getLifterTicks();
00308     return (-2 * pow(10, (-9 * ticks)) + 6 * (pow(10, (-6 * ticks * ticks))) +
00309             0.0198 * ticks + 2.3033);
00310 }
```

Here is the call graph for this function:



6.71.1.3 getLifterTicks()

```
int getLifterTicks ( )
```

Gets the value of the lifter pot.

Returns

the value of the pot.

Author

Chris Jerrett

Date

9/9/2017

Definition at line **297** of file **lifter.c**.

References **analogRead()**.

Referenced by **getLifterHeight()**.

```
00297 { return analogRead(LIFTER); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.71.1.4 interrupt_auto_stack()

```
void interrupt_auto_stack (
    void * param )
```

Stops an autostack in case of an error.

Parameters

<i>param</i>	ignore parameter
--------------	------------------

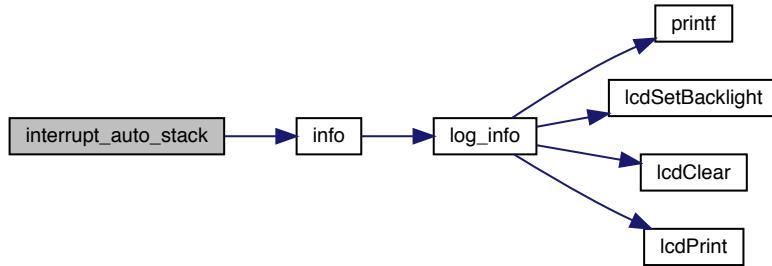
Definition at line **8** of file **lifter.c**.

References **info()**, and **lifter_autostack_routine_interrupt**.

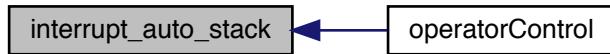
Referenced by **operatorControl()**.

```
00008
00009     info("int");
00010     lifter_autostack_routine_interrupt = true;
00011 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.71.1.5 lifterPotentiometerToDegree()

```
float lifterPotentiometerToDegree (
    int x )
```

height of the lifter in degrees from 0 height

Parameters

x	the pot value
---	---------------

Returns

the positions in degrees

Author

Chris Jerrett

Date

10/13/2017

Definition at line **286** of file **lifter.c**.

```
00286     {
00287     return (x - INIT_ROTATION) / TICK_MAX * DEG_MAX;
00288 }
```

6.71.1.6 lower_main_lifter()

```
void lower_main_lifter ( )
```

Lowers the main lifter.

Author

Christian DeSimone

Date

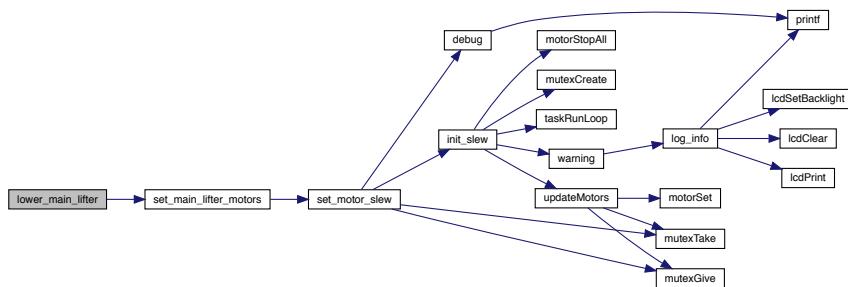
9/12/2017

Definition at line **160** of file **lifter.c**.

References **set_main_lifter_motors()**.

```
00160 { set_main_lifter_motors(MAX_SPEED); }
```

Here is the call graph for this function:



6.71.1.7 lower_secondary_lifter()

```
void lower_secondary_lifter ( )
```

Lowers the secondary lifter.

Author

Christian DeSimone

Date

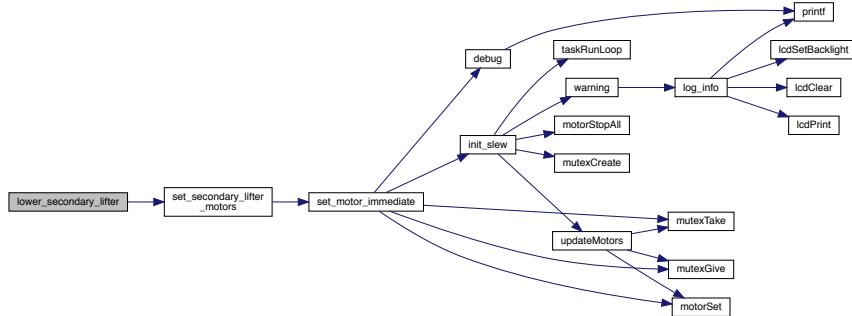
9/12/2017

Definition at line **176** of file **lifter.c**.

References **set_secondary_lifter_motors()**.

```
00176 { set_secondary_lifter_motors(MAX_SPEED); }
```

Here is the call graph for this function:



6.71.1.8 main_lifter_update()

```
static void main_lifter_update ( ) [static]
```

Definition at line **180** of file **lifter.c**.

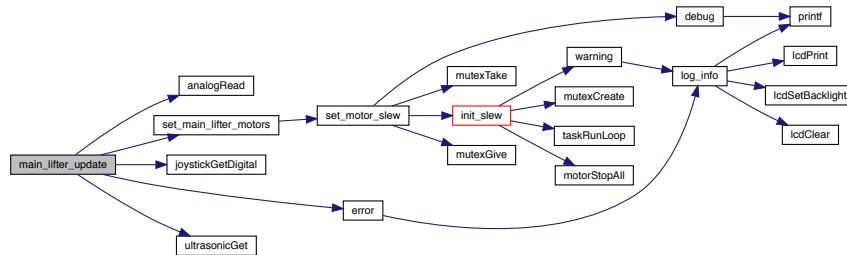
References **analogRead()**, **error()**, **joystickGetDigital()**, **lifter_autostack_running**, **lifter_ultrasonic**, **secondary_override**, **set_main_lifter_motors()**, and **ultrasonicGet()**.

Referenced by **update_lifter()**.

```

00180     if (lifter_autostack_running) {
00181         error("True");
00182         return;
00183     }
00184     error("1");
00185     static int count = 0;
00186     static bool pid_on = false;
00187     static int main_target = 0;
00188     int main_motor_speed = 0;
00189     static long long main_i = 0;
00190     if (count == 20) {
00191         main_target = analogRead(MAIN_LIFTER_POT);
00192     }
00193     if (pid_on && count > 20) {
00194         int curr = analogRead(MAIN_LIFTER_POT);
00195         static int main_last_p = 0;
00196         int main_p = curr - main_target;
00197         main_i += main_p;
00198         int main_d = main_last_p - main_p;
00199         // main_motor_speed = MAIN_LIFTER_P * main_p + MAIN_LIFTER_I * main_i +
00200         // MAIN_LIFTER_D * main_d;
00201         main_last_p = main_p;
00202     } else {
00203         main_i = 0;
00204         count++;
00205     }
00206     error("2");
00207     if (joystickGetDigital(LIFTER_UP)) {
00208         int ultra = ultrasonicGet(lifter_ultrasonic);
00209         main_motor_speed = MAX_SPEED;
00210         count = 0;
00211     } else if (joystickGetDigital(LIFTER_DOWN)) {
00212         main_motor_speed = MIN_SPEED;
00213         count = 0;
00214         secondary_override = false;
00215     } else {
00216         secondary_override = false;
00217     }
00218     set_main_lifter_motors(main_motor_speed);
00219     pid_on = true;
00220 }
00221 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.71.1.9 quit_auto_static()

```
static void quit_auto_static ( ) [inline], [static]
```

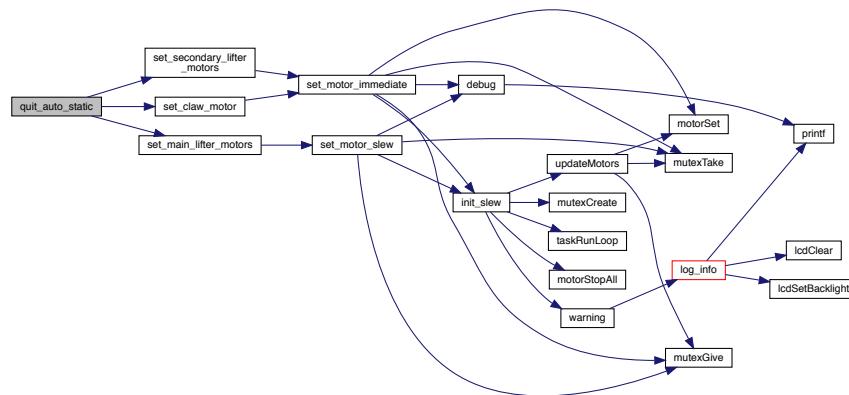
Definition at line 13 of file **lifter.c**.

References **lifter_autostack_running**, **set_claw_motor()**, **set_main_lifter_motors()**, and **set_secondary_lifter_motors()**.

Referenced by **autostack_routine()**.

```
00013
00014     set_main_lifter_motors(0);
00015     set_secondary_lifter_motors(0);
00016     set_claw_motor(0);
00017     lifter_autostack_running = false;
00018 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.71.1.10 raise_main_lifter()

```
void raise_main_lifter ( )
```

Raises the main lifter.

Author

Christian DeSimone

Date

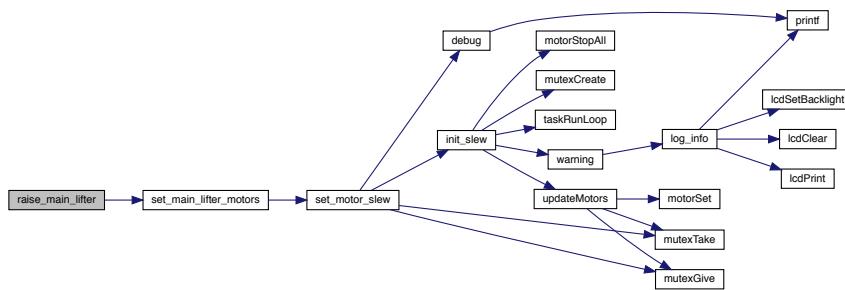
9/12/2017

Definition at line **152** of file **lifter.c**.

References **set_main_lifter_motors()**.

```
00152 { set_main_lifter_motors(MAX_SPEED); }
```

Here is the call graph for this function:



6.71.1.11 raise_secondary_lifter()

```
void raise_secondary_lifter ( )
```

Raises the main lifter.

Author

Christian DeSimone

Date

9/12/2017

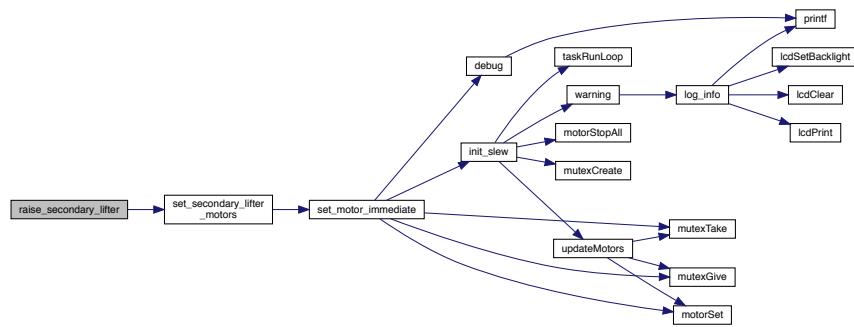
Definition at line **168** of file **lifter.c**.

References **set_secondary_lifter_motors()**.

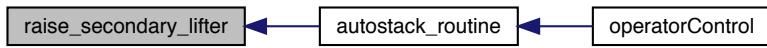
Referenced by **autostack_routine()**.

```
00168 { set_secondary_lifter_motors(MIN_SPEED / 1.5); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.71.1.12 secondary_lifter_update()

```
static void secondary_lifter_update ( ) [static]
```

Definition at line **223** of file **lifter.c**.

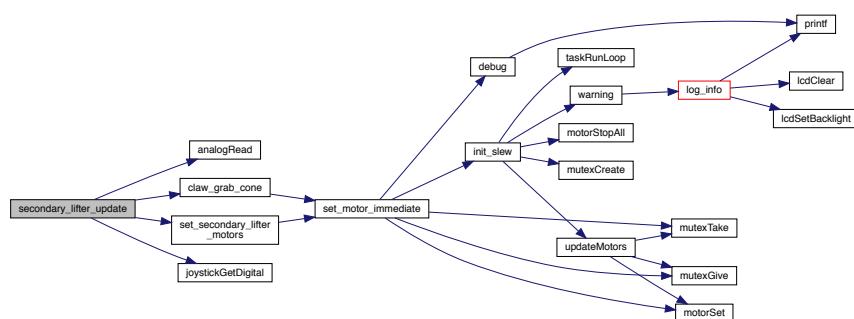
References **analogRead()**, **claw_grab_cone()**, **joystickGetDigital()**, **lifter_autostack_running**, and **set_secondary_lifter_motors()**.

Referenced by **update_lifter()**.

```

00223
00224     if (lifter_autostack_running)
00225         return;
00226     static int count = 0;
00227     // static bool pid_on = false;
00228     static int second_target = 0;
00229     int second_motor_speed = 0;
00230     static long long second_i = 0;
00231
00232     if (count < 10) {
00233         second_target = analogRead(SECONDARY_LIFTER_POT_PORT);
00234         count++;
00235     }
00236
00237     int curr = analogRead(SECONDARY_LIFTER_POT_PORT);
00238     static int second_last_p = 0;
00239     int second_p = curr - second_target;
00240     second_i += second_p;
00241     int second_d = second_last_p - second_p;
00242     second_motor_speed = SECONDARY_LIFTER_P * second_p +
00243                         SECONDARY_LIFTER_I * second_i +
00244                         SECONDARY_LIFTER_D * second_d;
00245     second_last_p = second_p;
00246
00247     if (joystickGetDigital(SECONDARY_LIFTER_DOWN)) {
00248         second_motor_speed = MAX_SPEED;
00249         count = 0;
00250         second_i = 0;
00251         second_target = analogRead(SECONDARY_LIFTER_POT_PORT);
00252         claw_grab_cone();
00253     } else if (joystickGetDigital(SECONDARY_LIFTER_UP)) {
00254         second_motor_speed = MIN_SPEED;
00255         count = 0;
00256         second_i = 0;
00257         second_target =
00258             second_target > 3000 ? 4095 : analogRead(SECONDARY_LIFTER_POT_PORT);
00259     } else {
00260         second_target = second_target > 3000 ? 4095 : second_target;
00261     }
00262     second_motor_speed = abs(second_motor_speed) < 20 ? 0 : second_motor_speed;
00263     set_secondary_lifter_motors(second_motor_speed);
00264 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.71.1.13 set_lifter_pos()

```
void set_lifter_pos ( int pos )
```

Sets the lifter positions to the given value.

Parameters

<i>pos</i>	The height in inches
------------	----------------------

Author

Chris Jerrett

Date

9/12/2017

Definition at line 144 of file **lifter.c**.

00144 {}

6.71.1.14 set_main_lifter_motors()

```
void set_main_lifter_motors ( const int v )
```

Sets the main lifter motors to the given value.

Parameters

<i>v</i>	value for the lifter motor. Between -128 - 127, any values outside are clamped.
----------	---

Author

Chris Jerrett

Date

9/9/2017

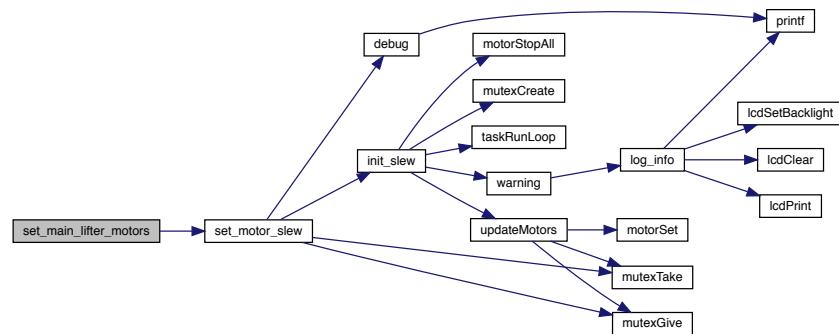
Definition at line 133 of file **lifter.c**.

References **set_motor_slew()**.

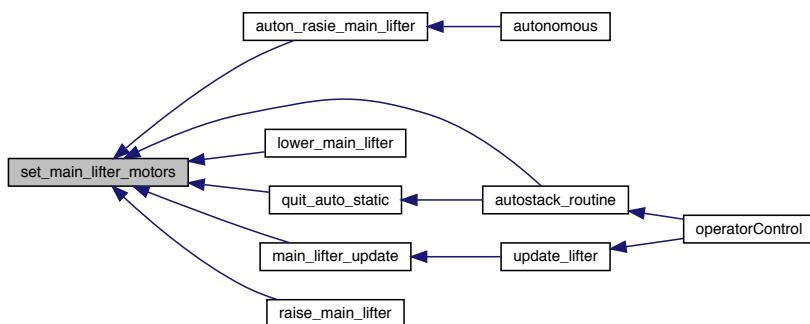
Referenced by **auton_rasie_main_lifter()**, **autostack_routine()**, **lower_main_lifter()**, **main_lifter_update()**, **quit_auto_static()**, and **raise_main_lifter()**.

```
00133     {
00134     set_motor_slew(MOTOR_MAIN_LIFTER, v);
00135 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.71.1.15 set_secondary_lifter_motors()

```
void set_secondary_lifter_motors (
    const int v )
```

Sets the secondary lifter motors to the given value.

Parameters

v	value for the lifter motor. Between -128 - 127, any values outside are clamped.
---	---

Author

Chris Jerrett

Date

1/6/2018

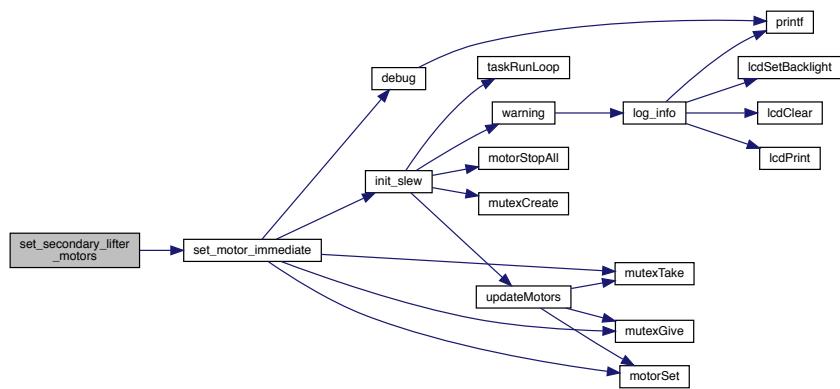
Definition at line 121 of file **lifter.c**.

References **set_motor_immediate()**.

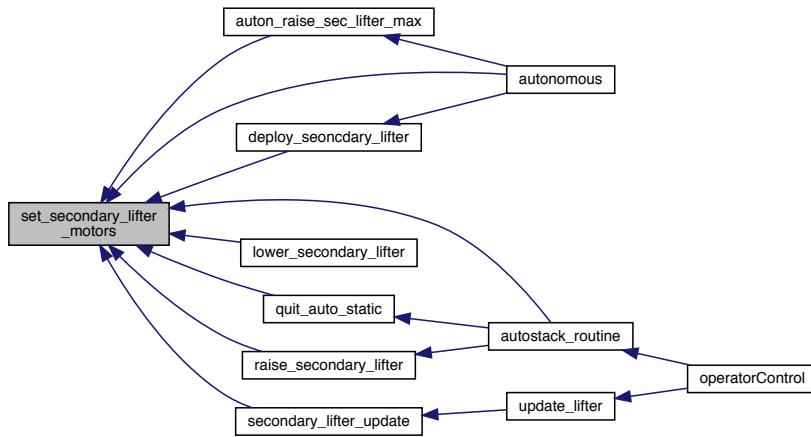
Referenced by **auton_raise_sec_lifter_max()**, **autonomous()**, **autostack_routine()**, **deploy_seoncdary_lifter()**, **lower_secondary_lifter()**, **quit_auto_static()**, **raise_secondary_lifter()**, and **secondary_lifter_update()**.

```
00121             {
00122     set_motor_immediate(MOTOR_SECONDARY_LIFTER, v);
00123 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.71.1.16 update_lifter()

```
void update_lifter( )
```

Updates the lifter in teleop.

Author

Chris Jerrett

Date

9/9/2017

Definition at line 272 of file **lifter.c**.

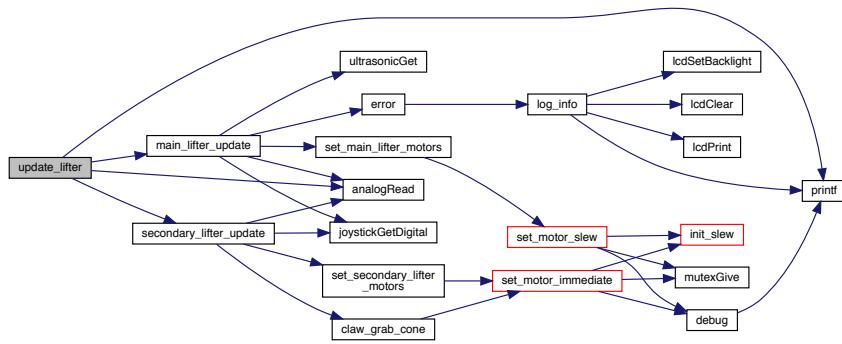
References `analogRead()`, `main_lifter_update()`, `printf()`, `secondary_lifter_update()`, and `secondary_override`.

Referenced by `operatorControl()`.

```

00272
00273     printf("%d \n", analogRead(SECONDARY_LIFTER_POT_PORT));
00274     main_lifter_update();
00275     if (!secondary_override)
00276         secondary_lifter_update();
00277 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.71.2 Variable Documentation

6.71.2.1 lifter_autostack_routine_interrupt

```
bool lifter_autostack_routine_interrupt = false [static]
```

Definition at line 6 of file **lifter.c**.

Referenced by `autostack_routine()`, and `interrupt_auto_stack()`.

6.71.2.2 lifter_autostack_running

```
bool lifter_autostack_running = false
```

Definition at line 5 of file **lifter.c**.

Referenced by `autostack_routine()`, `main_lifter_update()`, `quit_auto_static()`, `secondary_lifter_update()`, and `update_claw()`.

6.71.2.3 secondary_override

```
bool secondary_override = false [static]
```

Definition at line 178 of file **lifter.c**.

Referenced by **main_lifter_update()**, and **update_lifter()**.

6.72 lifter.c

```
00001 #include "lifter.h"
00002 #include "claw.h"
00003 #include "log.h"
00004
00005 bool lifter_autostack_running = false;
00006 static bool lifter_autostack_routine_interrupt = false;
00007
00008 void interrupt_auto_stack(void *param) {
00009     info("int");
0010     lifter_autostack_routine_interrupt = true;
0011 }
0012
0013 static inline void quit_auto_static() {
0014     set_main_lifter_motors(0);
0015     set_secondary_lifter_motors(0);
0016     set_claw_motor(0);
0017     lifter_autostack_running = false;
0018 }
0019
0020 void autostack_routine(void *param) {
0021     lifter_autostack_routine_interrupt = false;
0022     lifter_autostack_running = true;
0023     raise_secondary_lifter();
0024     while (analogRead(SECONDARY_LIFTER_POT_PORT) < 1600) {
0025         set_secondary_lifter_motors(MIN_SPEED);
0026         if (lifter_autostack_routine_interrupt) {
0027             quit_auto_static();
0028             return;
0029         }
0030         delay(50);
0031         info("1");
0032     }
0033     set_secondary_lifter_motors(0);
0034     bool lifted = false;
0035     int val = ultrasonicGet(lifter_ultrasonic);
0036     printf("%d\n", val);
0037     while (val < 10 && val != ULTRA_BAD_RESPONSE) {
0038         if (lifter_autostack_routine_interrupt) {
0039             quit_auto_static();
0040             return;
0041         }
0042         set_main_lifter_motors(MAX_SPEED);
0043         info("2");
0044         lifted = true;
0045         delay(50);
0046         val = ultrasonicGet(lifter_ultrasonic);
0047         printf("%d\n", val);
0048     }
0049     if (lifter_autostack_routine_interrupt) {
0050         quit_auto_static();
0051         return;
0052     }
0053     delay(200);
0054     if (lifted)
0055         delay(50);
0056     if (lifter_autostack_routine_interrupt) {
0057         quit_auto_static();
0058         return;
0059     }
0060     set_main_lifter_motors(0);
0061     set_secondary_lifter_motors(0);
0062     while (analogRead(SECONDARY_LIFTER_POT_PORT) < 3000) {
```

```

00064     if (lifter_autostack_routine_interrupt) {
00065         quit_auto_static();
00066         return;
00067     }
00068     set_secondary_lifter_motors(MIN_SPEED);
00069     delay(50);
00070     info("3");
00071 }
00072
00073 set_main_lifter_motors(MIN_SPEED / 1.333);
00074
00075 while (val > 10) {
00076     if (lifter_autostack_routine_interrupt) {
00077         quit_auto_static();
00078         return;
00079     }
00080     info("2");
00081     lifted = true;
00082     delay(30);
00083     val = ultrasonicGet(lifter_ultrasonic);
00084     printf("%d\n", val);
00085 }
00086
00087 set_main_lifter_motors(0);
00088
00089 set_claw_motor(MIN_CLAW_SPEED);
00090 if (lifter_autostack_routine_interrupt) {
00091     quit_auto_static();
00092     return;
00093 }
00094 delay(500);
00095 if (lifter_autostack_routine_interrupt) {
00096     quit_auto_static();
00097     return;
00098 }
00099 set_main_lifter_motors(MAX_SPEED);
00100 if (lifter_autostack_routine_interrupt) {
00101     quit_auto_static();
00102     return;
00103 }
00104 delay(300);
00105
00106 set_main_lifter_motors(MIN_SPEED);
00107 set_claw_motor(0);
00108 set_secondary_lifter_motors(0);
00109
00110 lifter_autostack_running = false;
00111 }
00112
00121 void set_secondary_lifter_motors(const int v) {
00122     set_motor_immediate(MOTOR_SECONDARY_LIFTER, v);
00123 }
00124
00133 void set_main_lifter_motors(const int v) {
00134     set_motor_slew(MOTOR_MAIN_LIFTER, v);
00135 }
00136
00144 void set_lifter_pos(int pos) {}
00145
00152 void raise_main_lifter() { set_main_lifter_motors(MAX_SPEED); }
00153
00160 void lower_main_lifter() { set_main_lifter_motors(MIN_SPEED); }
00161
00168 void raise_secondary_lifter() { set_secondary_lifter_motors(MIN_SPEED / 1.5); }
00169
00176 void lower_secondary_lifter() { set_secondary_lifter_motors(MAX_SPEED); }
00177
00178 static bool secondary_override = false;
00179
00180 static void main_lifter_update() {
00181     if (lifter_autostack_running) {
00182         error("True");
00183         return;
00184     }
00185     error("1");
00186     static int count = 0;
00187     static bool pid_on = false;
00188     static int main_target = 0;
00189     int main_motor_speed = 0;
00190     static long long main_i = 0;
00191     if (count == 20) {

```

```
00192     main_target = analogRead(MAIN_LIFTER_POT);
00193 }
00194 if (pid_on && count > 20) {
00195     int curr = analogRead(MAIN_LIFTER_POT);
00196     static int main_last_p = 0;
00197     int main_p = curr - main_target;
00198     main_i += main_p;
00199     int main_d = main_last_p - main_p;
00200     // main_motor_speed = MAIN_LIFTER_P * main_p + MAIN_LIFTER_I * main_i +
00201     // MAIN_LIFTER_D * main_d;
00202     main_last_p = main_p;
00203 } else {
00204     main_i = 0;
00205     count++;
00206 }
00207 error("2");
00208 if (joystickGetDigital(LIFTER_UP)) {
00209     int ultra = ultrasonicGet(lifter_ultrasonic);
00210     main_motor_speed = MAX_SPEED;
00211     count = 0;
00212 } else if (joystickGetDigital(LIFTER_DOWN)) {
00213     main_motor_speed = MIN_SPEED;
00214     count = 0;
00215     secondary_override = false;
00216 } else {
00217     secondary_override = false;
00218 }
00219 set_main_lifter_motors(main_motor_speed);
00220 pid_on = true;
00221 }
00222
00223 static void secondary_lifter_update() {
00224     if (lifter_autostack_running)
00225         return;
00226     static int count = 0;
00227     // static bool pid_on = false;
00228     static int second_target = 0;
00229     int second_motor_speed = 0;
00230     static long long second_i = 0;
00231
00232     if (count < 10) {
00233         second_target = analogRead(SECONDARY_LIFTER_POT_PORT);
00234         count++;
00235     }
00236
00237     int curr = analogRead(SECONDARY_LIFTER_POT_PORT);
00238     static int second_last_p = 0;
00239     int second_p = curr - second_target;
00240     second_i += second_p;
00241     int second_d = second_last_p - second_p;
00242     second_motor_speed = SECONDARY_LIFTER_P * second_p +
00243                         SECONDARY_LIFTER_I * second_i +
00244                         SECONDARY_LIFTER_D * second_d;
00245     second_last_p = second_p;
00246
00247     if (joystickGetDigital(SECONDARY_LIFTER_DOWN)) {
00248         second_motor_speed = MAX_SPEED;
00249         count = 0;
00250         second_i = 0;
00251         second_target = analogRead(SECONDARY_LIFTER_POT_PORT);
00252         claw_grab_cone();
00253     } else if (joystickGetDigital(SECONDARY_LIFTER_UP)) {
00254         second_motor_speed = MIN_SPEED;
00255         count = 0;
00256         second_i = 0;
00257         second_target =
00258             second_target > 3000 ? 4095 : analogRead(SECONDARY_LIFTER_POT_PORT);
00259     } else {
00260         second_target = second_target > 3000 ? 4095 : second_target;
00261     }
00262     second_motor_speed = abs(second_motor_speed) < 20 ? 0 : second_motor_speed;
00263     set_secondary_lifter_motors(second_motor_speed);
00264 }
00265
00272 void update_lifter() {
00273     printf("%d \n", analogRead(SECONDARY_LIFTER_POT_PORT));
00274     main_lifter_update();
00275     if (!secondary_override)
00276         secondary_lifter_update();
00277 }
00286 float lifterPotentiometerToDegree(int x) {
```

```

00287     return (x - INIT_ROTATION) / TICK_MAX * DEG_MAX;
00288 }
00289
00297 int getLifterTicks() { return analogRead(LIFTER); }
00298
00306 double getLifterHeight() {
00307     unsigned int ticks = getLifterTicks();
00308     return (-2 * pow(10, (-9 * ticks)) + 6 * (pow(10, (-6 * ticks * ticks))) +
00309             0.0198 * ticks + 2.3033);
00310 }

```

6.73 src/list.c File Reference

Functions

- **list_node_t * list_at (list_t *self, int index)**
Finds a node a given index.
- **void list_destroy (list_t *self)**
Deallocates a list.
- **list_node_t * list_find (list_t *self, void *val)**
Finds a node in a list with a given value.
- **list_node_t * list_lpop (list_t *self)**
removes and returns the start node
- **list_node_t * list_lpush (list_t *self, list_node_t *node)**
Pushed a node to the start of a list.
- **list_t * list_new ()**
Allocated a new list.
- **void list_remove (list_t *self, list_node_t *node)**
removes and returns the a given node from the list
- **list_node_t * list_rpop (list_t *self)**
removes and returns the end node
- **list_node_t * list_rpush (list_t *self, list_node_t *node)**
Pushed a node to the end of a list.

6.73.1 Function Documentation

6.73.1.1 list_at()

```

list_node_t* list_at (
    list_t * self,
    int index )

```

Finds a node a given index.

Parameters

self	the list
index	the index

Returns

the node

Author

Chris Jerrett

Date

1/3/18

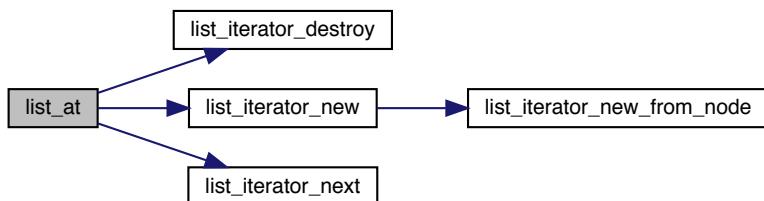
Definition at line 202 of file **list.c**.

References **LIST_HEAD**, **list_iterator_destroy()**, **list_iterator_new()**, **list_iterator_next()**, and **LIST_TAIL**.

```

00202     {
00203     list_direction_t direction = LIST_HEAD;
00204
00205     if (index < 0) {
00206         direction = LIST_TAIL;
00207         index = ~index;
00208     }
00209
00210     if ((unsigned)index < self->len) {
00211         list_iterator_t *it = list_iterator_new(self, direction);
00212         list_node_t *node = list_iterator_next(it);
00213         while (index--)
00214             node = list_iterator_next(it);
00215         list_iterator_destroy(it);
00216         return node;
00217     }
00218
00219     return NULL;
00220 }
```

Here is the call graph for this function:

**6.73.1.2 list_destroy()**

```
void list_destroy (
    list_t * self )
```

Deallocates a list.

Parameters

<i>self</i>	the list
-------------	----------

Author

Chris Jerrett

Date

1/3/18

Definition at line **50** of file **list.c**.References **list_node::next**, and **list_node::val**.Referenced by **deinit_routines()**.

```

00050
00051     unsigned int len = self->len;
00052     list_node_t *next;
00053     list_node_t *curr = self->head;
00054
00055     while (len--) {
00056         next = curr->next;
00057         if (self->free)
00058             self->free(curr->val);
00059         free(curr);
00060         curr = next;
00061     }
00062
00063     free(self);
00064 }
```

Here is the caller graph for this function:

**6.73.1.3 list_find()**

```

list_node_t* list_find (
    list_t * self,
    void * val )

```

Finds a node in a list with a given value.

Parameters

<i>self</i>	the list
<i>val</i>	the value

Returns

the node

Author

Chris Jerrett

Date

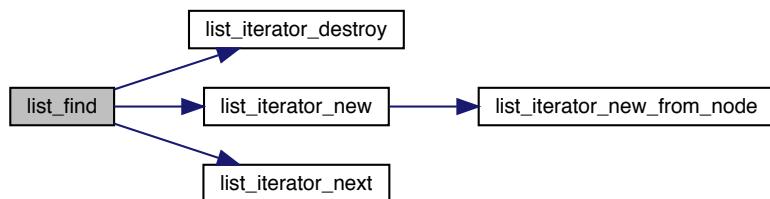
1/3/18

Definition at line 172 of file **list.c**.References **LIST_HEAD**, **list_iterator_destroy()**, **list_iterator_new()**, **list_iterator_next()**, and **list_node::val**.

```

00172
00173     list_iterator_t *it = list_iterator_new(self, LIST_HEAD);
00174     list_node_t *node;
00175
00176     while ((node = list_iterator_next(it))) {
00177         if (self->match) {
00178             if (self->match(val, node->val)) {
00179                 list_iterator_destroy(it);
00180                 return node;
00181             }
00182         } else {
00183             if (val == node->val) {
00184                 list_iterator_destroy(it);
00185                 return node;
00186             }
00187         }
00188     }
00189
00190     list_iterator_destroy(it);
00191     return NULL;
00192 }
```

Here is the call graph for this function:



6.73.1.4 list_lpop()

```
list_node_t* list_lpop (
    list_t * self )
```

removes and returns the start node

Parameters

<code>self</code>	the list
-------------------	----------

Returns

the node removed

Author

Chris Jerrett

Date

1/3/18

Definition at line **122** of file **list.c**.

References **list_node::next**, and **list_node::prev**.

```
00122                                     {
00123     if (!self->len)
00124         return NULL;
00125
00126     list_node_t *node = self->head;
00127
00128     if (--self->len) {
00129         (self->head = node->next)->prev = NULL;
00130     } else {
00131         self->head = self->tail = NULL;
00132     }
00133
00134     node->next = node->prev = NULL;
00135     return node;
00136 }
```

6.73.1.5 list_lpush()

```
list_node_t* list_lpush (
    list_t * self,
    list_node_t * node )
```

Pushed a node to the start of a list.

Parameters

<i>self</i>	the list
<i>node</i>	the node

Returns

the node added

Author

Chris Jerrett

Date

1/3/18

Definition at line **146** of file **list.c**.

References **list_node::next**, and **list_node::prev**.

```
00146 {  
00147     if (!node)  
00148         return NULL;  
00149  
00150     if (self->len) {  
00151         node->next = self->head;  
00152         node->prev = NULL;  
00153         self->head->prev = node;  
00154         self->head = node;  
00155     } else {  
00156         self->head = self->tail = node;  
00157         node->prev = node->next = NULL;  
00158     }  
00159  
00160     ++self->len;  
00161     return node;  
00162 }
```

6.73.1.6 list_new()

```
list_t* list_new ( )
```

Allocated a new list.

Returns

the new list

Author

Chris Jerrett

Date

1/3/18

Definition at line 34 of file **list.c**.

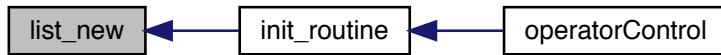
References **list_t::head**.

Referenced by **init_routine()**.

```

00034     {
00035     list_t *self;
00036     if (! (self = (list_t *)malloc(sizeof(list_t))))
00037         return NULL;
00038     self->head = NULL;
00039     self->tail = NULL;
00040     self->free = NULL;
00041     self->match = NULL;
00042     self->len = 0;
00043     return self;
00044 }
```

Here is the caller graph for this function:

**6.73.1.7 list_remove()**

```

void list_remove (
    list_t * self,
    list_node_t * node )
```

removes and returns the a given node from the list

Parameters

<i>self</i>	the list
-------------	----------

Returns

the node removed

Author

Chris Jerrett

Date

1/3/18

Definition at line **229** of file **list.c**.

References **list_node::next**, **list_node::prev**, and **list_node::val**.

```
00229     {
00230     node->prev ? (node->prev->next = node->next) : (self->head = node->next);
00231
00232     node->next ? (node->next->prev = node->prev) : (self->tail = node->prev);
00233
00234     if (self->free)
00235         self->free(node->val);
00236
00237     free(node);
00238     --self->len;
00239 }
```

6.73.1.8 list_rpop()

```
list_node_t* list_rpop (
    list_t * self )
```

removes and returns the end node

Parameters

self	the list
------	----------

Returns

the node removed

Author

Chris Jerrett

Date

1/3/18

Definition at line **99** of file **list.c**.References **list_node::next**, and **list_node::prev**.

```

00099          {
00100      if (!self->len)
00101          return NULL;
00102
00103      list_node_t *node = self->tail;
00104
00105      if (--self->len) {
00106          (self->tail = node->prev)->next = NULL;
00107      } else {
00108          self->tail = self->head = NULL;
00109      }
00110
00111      node->next = node->prev = NULL;
00112      return node;
00113 }
```

6.73.1.9 list_rpush()

```
list_node_t* list_rpush (
    list_t * self,
    list_node_t * node )
```

Pushed a node to the end of a list.

Parameters

<i>self</i>	the list
<i>node</i>	the node

Returns

the node added

Author

Chris Jerrett

Date

1/3/18

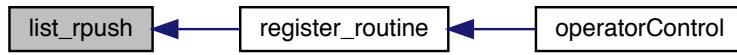
Definition at line **74** of file **list.c**.References **list_node::next**, and **list_node::prev**.Referenced by **register_routine()**.

```

00074     if (!node)
00075         return NULL;
00077
00078     if (self->len) {
00079         node->prev = self->tail;
00080         node->next = NULL;
00081         self->tail->next = node;
00082         self->tail = node;
00083     } else {
00084         self->head = self->tail = node;
00085         node->prev = node->next = NULL;
00086     }
00087
00088     ++self->len;
00089     return node;
00090 }

```

Here is the caller graph for this function:



6.74 list.c

```

00001
00025 #include "list.h"
00026 #include <API.h>
00027
00034 list_t *list_new() {
00035     list_t *self;
00036     if (!(self = (list_t *)malloc(sizeof(list_t))))
00037         return NULL;
00038     self->head = NULL;
00039     self->tail = NULL;
00040     self->free = NULL;
00041     self->match = NULL;
00042     self->len = 0;
00043     return self;
00044 }
00045
00046 /*
00047 * Free the list.
00048 */
00049
00050 void list_destroy(list_t *self) {
00051     unsigned int len = self->len;
00052     list_node_t *next;
00053     list_node_t *curr = self->head;
00054
00055     while (len--) {
00056         next = curr->next;
00057         if (self->free)
00058             self->free(curr->val);
00059         free(curr);
00060         curr = next;
00061     }
00062
00063     free(self);
00064 }
00065
00074 list_node_t *list_rpush(list_t *self, list_node_t *node) {
00075     if (!node)

```

```

00076     return NULL;
00077
00078     if (self->len) {
00079         node->prev = self->tail;
00080         node->next = NULL;
00081         self->tail->next = node;
00082         self->tail = node;
00083     } else {
00084         self->head = self->tail = node;
00085         node->prev = node->next = NULL;
00086     }
00087
00088     ++self->len;
00089     return node;
00090 }
00091
00099 list_node_t *list_rpop(list_t *self) {
00100     if (!self->len)
00101         return NULL;
00102
00103     list_node_t *node = self->tail;
00104
00105     if (--self->len) {
00106         (self->tail = node->prev)->next = NULL;
00107     } else {
00108         self->tail = self->head = NULL;
00109     }
00110
00111     node->next = node->prev = NULL;
00112     return node;
00113 }
00114
00122 list_node_t *list_lpop(list_t *self) {
00123     if (!self->len)
00124         return NULL;
00125
00126     list_node_t *node = self->head;
00127
00128     if (--self->len) {
00129         (self->head = node->next)->prev = NULL;
00130     } else {
00131         self->head = self->tail = NULL;
00132     }
00133
00134     node->next = node->prev = NULL;
00135     return node;
00136 }
00137
00146 list_node_t *list_lpush(list_t *self, list_node_t *node) {
00147     if (!node)
00148         return NULL;
00149
00150     if (self->len) {
00151         node->next = self->head;
00152         node->prev = NULL;
00153         self->head->prev = node;
00154         self->head = node;
00155     } else {
00156         self->head = self->tail = node;
00157         node->prev = node->next = NULL;
00158     }
00159
00160     ++self->len;
00161     return node;
00162 }
00163
00172 list_node_t *list_find(list_t *self, void *val) {
00173     list_iterator_t *it = list_iterator_new(self, LIST_HEAD);
00174     list_node_t *node;
00175
00176     while ((node = list_iterator_next(it))) {
00177         if (self->match) {
00178             if (self->match(val, node->val)) {
00179                 list_iterator_destroy(it);
00180                 return node;
00181             }
00182         } else {
00183             if (val == node->val) {
00184                 list_iterator_destroy(it);
00185                 return node;
00186             }
00187     }
00188 }
```

```

00187      }
00188  }
00189
00190  list_iterator_destroy(it);
00191  return NULL;
00192 }
00193
00194 list_node_t *list_at(list_t *self, int index) {
00195  list_direction_t direction = LIST_HEAD;
00196
00197  if (index < 0) {
00198    direction = LIST_TAIL;
00199    index = ~index;
00200  }
00201
00202  if ((unsigned)index < self->len) {
00203    list_iterator_t *it = list_iterator_new(self, direction);
00204    list_node_t *node = list_iterator_next(it);
00205    while (index--)
00206      node = list_iterator_next(it);
00207    list_iterator_destroy(it);
00208    return node;
00209  }
00210
00211  return NULL;
00212 }
00213
00214 void list_remove(list_t *self, list_node_t *node) {
00215  node->prev ? (node->prev->next = node->next) : (self->head = node->next);
00216
00217  node->next ? (node->next->prev = node->prev) : (self->tail = node->prev);
00218
00219  if (self->free)
00220    self->free(node->val);
00221
00222  free(node);
00223  --self->len;
00224 }
```

6.75 src/list_iterator.c File Reference

Functions

- **void list_iterator_destroy (list_iterator_t *self)**
Destroys the iterator.
- **list_iterator_t * list_iterator_new (list_t *list, list_direction_t direction)**
Creates a new iterator.
- **list_iterator_t * list_iterator_new_from_node (list_node_t *node, list_direction_t direction)**
Creates a new iterator by using the node to start at.
- **list_node_t * list_iterator_next (list_iterator_t *self)**
The next node in the iterator and advances the iterator.

6.75.1 Function Documentation

6.75.1.1 list_iterator_destroy()

```
void list_iterator_destroy (
    list_iterator_t * self )
```

Destroys the iterator.

Parameters

<i>self</i>	the iterator
-------------	--------------

Author

Chris Jerrett

Date

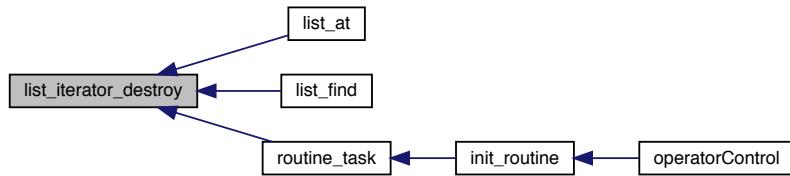
1/3/17

Definition at line **56** of file **list_iterator.c**.Referenced by **list_at()**, **list_find()**, and **routine_task()**.

```

00056
00057     free(self);
00058     self = NULL;
00059 }
```

Here is the caller graph for this function:

**6.75.1.2 list_iterator_new()**

```
list_iterator_t* list_iterator_new (
    list_t * list,
    list_direction_t direction )
```

Creates a new iterator.

Parameters

<i>list</i>	the list
<i>direction</i>	direction the iterator should progress in

Returns

the iterator created

Author

Chris Jerrett

Date

1/3/18

Definition at line 11 of file **list_iterator.c**.

References **list_t::head**, **LIST_HEAD**, **list_iterator_new_from_node()**, and **list_t::tail**.

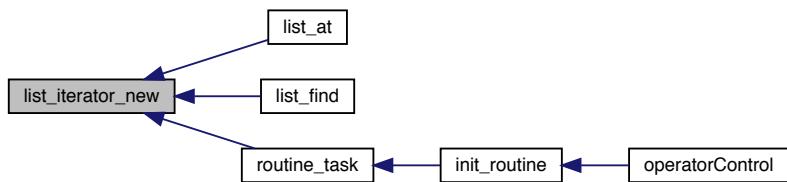
Referenced by **list_at()**, **list_find()**, and **routine_task()**.

```
00011
00012     list_node_t *node = direction == LIST_HEAD ? list->head : list->tail;
00013     return list_iterator_new_from_node(node, direction);
00014 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.75.1.3 **list_iterator_new_from_node()**

```
list_iterator_t* list_iterator_new_from_node (
    list_node_t * node,
    list_direction_t direction )
```

Creates a new iterator by using the node to start at.

Parameters

<i>node</i>	the start node
<i>direction</i>	direction the iterator should progress in

Returns

the iterator created

Author

Chris Jerrett

Date

1/3/18

Definition at line **24** of file **list_iterator.c**.

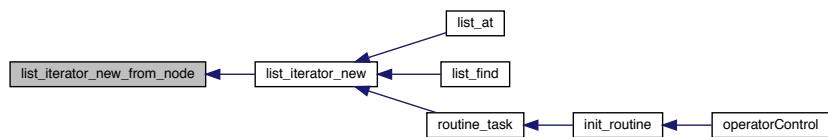
References **list_iterator_t::next**.

Referenced by **list_iterator_new()**.

```

00025
00026     list_iterator_t *self;
00027     if (!(self = (list_iterator_t *)malloc(sizeof(list_iterator_t))))
00028         return NULL;
00029     self->next = node;
00030     self->direction = direction;
00031     return self;
00032 }
```

Here is the caller graph for this function:

**6.75.1.4 list_iterator_next()**

```

list_node_t* list_iterator_next (
    list_iterator_t * self )
```

The next node in the iterator and advances the iterator.

Returns NULL when done.

Parameters

<i>self</i>	the iterator
-------------	--------------

Returns

the next node.

Author

Chris Jerrett

Date

1/3/17

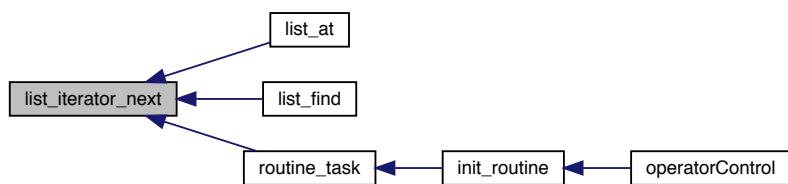
Definition at line **42** of file **list_iterator.c**.

References **LIST_HEAD**, **list_node::next**, and **list_node::prev**.

Referenced by **list_at()**, **list_find()**, and **routine_task()**.

```
00042 {  
00043     list_node_t *curr = self->next;  
00044     if (curr) {  
00045         self->next = self->direction == LIST_HEAD ? curr->next : curr->prev;  
00046     }  
00047     return curr;  
00048 }
```

Here is the caller graph for this function:



6.76 list_iterator.c

```

00001 #include "list.h"
00002 #include <API.h>
00011 list_iterator_t *list_iterator_new(list_t *list, list_direction_t direction) {
00012     list_node_t *node = direction == LIST_HEAD ? list->head : list->tail;
00013     return list_iterator_new_from_node(node, direction);
00014 }
00015
00024 list_iterator_t *list_iterator_new_from_node(list_node_t *node,
00025                                              list_direction_t direction) {
00026     list_iterator_t *self;
00027     if (!(self = (list_iterator_t *)malloc(sizeof(list_iterator_t))))
00028         return NULL;
00029     self->next = node;
00030     self->direction = direction;
00031     return self;
00032 }
00033
00042 list_node_t *list_iterator_next(list_iterator_t *self) {
00043     list_node_t *curr = self->next;
00044     if (curr) {
00045         self->next = self->direction == LIST_HEAD ? curr->next : curr->prev;
00046     }
00047     return curr;
00048 }
00049
00056 void list_iterator_destroy(list_iterator_t *self) {
00057     free(self);
00058     self = NULL;
00059 }
```

6.77 src/list_node.c File Reference

Functions

- **list_node_t * list_node_new (void *val)**

Allocates a new node.

6.77.1 Function Documentation

6.77.1.1 list_node_new()

```
list_node_t* list_node_new (
    void * val )
```

Allocates a new node.

Parameters

val	The value the node contains.
------------	------------------------------

Returns

The newly allocated node.

Author

Chris Jerrett

Date

1/3/18 Node must be freed

Definition at line **12** of file **list_node.c**.

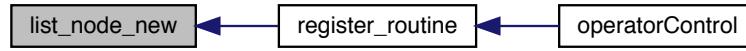
References **list_node::next**, **list_node::prev**, and **list_node::val**.

Referenced by **register_routine()**.

```

00012
00013     list_node_t *self;
00014     if (!(self = (list_node_t *)malloc(sizeof(list_node_t))))
00015         return NULL;
00016     self->prev = NULL;
00017     self->next = NULL;
00018     self->val = val;
00019     return self;
00020 }
```

Here is the caller graph for this function:

**6.78 list_node.c**

```

00001 #include "list.h"
00002 #include <API.h>
00003
00012 list_node_t *list_node_new(void *val) {
00013     list_node_t *self;
00014     if (!(self = (list_node_t *)malloc(sizeof(list_node_t))))
00015         return NULL;
00016     self->prev = NULL;
00017     self->next = NULL;
00018     self->val = val;
00019     return self;
00020 }
```

6.79 src/localization.c File Reference**Data Structures**

- struct **accelerometer_odometry**

Structure for holding an xy position from the accelerometer.

- struct **encoder_odometry**

Structure for holding an xy position and an angle theta from the IMEs.

Functions

- static struct **accelerometer_odometry** **calculate_accelerometer_odometry ()**
calculates the robot's position using the accelerometer
- static double **calculate_angle ()**
- int **calculate_encoder_angle ()**
calculates the current angle using the IMEs
- static void **calculate_encoder_odometry ()**
calculates the x y position of the robot based upon the IMEs
- static double **calculate_gryo_angular_velocity ()**
calculates the angular velocity using the gyro positions
- struct **location get_position ()**
Gets the current position of the robot.
- bool **init_localization** (const unsigned char gyro1, unsigned short multiplier, int start_x, int start_y, int start_theta)
initializes the localization
- static double **integrate_gyro_w** (int new_w)
Increases the stored angle based upon the update frequency and the current angular velocity.
- void **update_position ()**
Updates the position from the localization.

Variables

- static **Gyro g1**
- static int **last_call** = 0
- static **TaskHandle localization_task**
- **matrix * state_matrix**

6.79.1 Function Documentation

6.79.1.1 calculate_accelerometer_odometry()

```
static struct accelerometer_odometry calculate_accelerometer_odometry ( ) [static]
```

calculates the robot's position using the accelerometer

Author

Chris Jerrett, Christian DeSimone

Returns

the xy position of the robot

See also

[accelerometer_odometry](#) (p. 6)

Definition at line **70** of file **localization.c**.

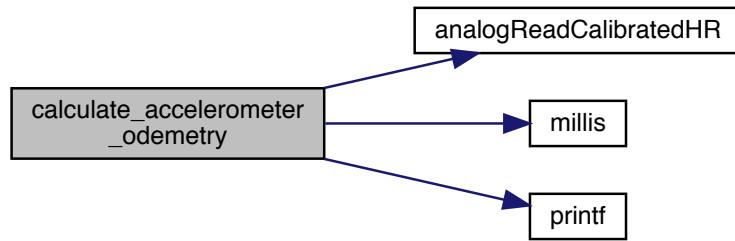
References `analogReadCalibratedHR()`, `last_call`, `millis()`, and `printf()`.

Referenced by `update_position()`.

```

00070
00071     static double vel_acumm_x = 0;
00072     static double vel_acumm_y = 0;
00073
00074     int32_t accel_x_rel = (int32_t)analogReadCalibratedHR(2);
00075     int32_t accel_y_rel = (int32_t)analogReadCalibratedHR(3);
00076
00077     // Ignore atom format string errors
00078     printf("x: %+" PRId32 " y: %+" PRId32 "\n", accel_x_rel, accel_y_rel);
00079
00080     double delta_time = ((millis() - last_call) / 1000.0);
00081     // double accel_x_abs = (accel_x_rel * cos(theta) + accel_y_rel * sin(theta))
00082     // * delta_time;    double accel_y_abs = (accel_y_rel * cos(theta) +
00083     // accel_x_rel
00084     // * sin(theta)) * delta_time;
00085
00086     // vel_acumm_x += accel_x_abs;
00087     // vel_acumm_y += accel_y_abs;
00088
00089     // double new_x = x + vel_acumm_x * delta_time;
00090     // double new_y = y + vel_acumm_y * delta_time;
00091
00092     struct accelerometer_odometry od;
00093     // od.x = new_x;
00094     // od.y = new_y;
00095     return od;
00096 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.79.1.2 calculate_angle()

```
static double calculate_angle ( ) [static]
```

6.79.1.3 calculate_encoder_angle()

```
int calculate_encoder_angle ( )
```

calculates the current angle using the IMEs

Calculates the angle using the encoders.

Returns

the angle of rotation

Author

Chris Jerrett, Christian DeSimone

Definition at line **129** of file **localization.c**.

References **get_encoder_ticks()**.

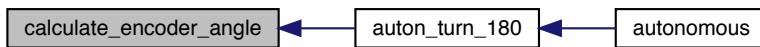
Referenced by **auton_turn_180()**.

```
00129
00130 #define WIDTH 13.5
00131 #define CPR 392.0
00132 #define WHEEL_RADIUS 2
00133     int dist_r = get_encoder_ticks(0) / CPR;
00134     int dist_l = get_encoder_ticks(1) / CPR;
00135     return ((dist_r - dist_l) / WIDTH);
00136 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.79.1.4 calculate_encoder_odometry()

```
static void calculate_encoder_odometry ( ) [static]
```

calculates the x y position of the robot based upon the IMEs

Author

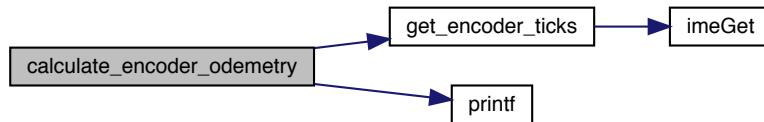
Chris Jerrett, Christian DeSimone

Definition at line 142 of file **localization.c**.

References **get_encoder_ticks()**, **printf()**, and **encoder_odemtry::theta**.

```
00142 {  
00143 #define WIDTH 13.5  
00144 #define CPR 392.0  
00145 #define WHEEL_RADIUS 2  
00146  
00147 int dist_r = get_encoder_ticks(0) / CPR;  
00148 int dist_l = get_encoder_ticks(1) / CPR;  
00149 printf("dist_r: %d dist_l: %d\n", dist_r, dist_l);  
00150 int theta = (dist_l - dist_r) / WIDTH;  
00151 printf("theta: %d\n", theta);  
00152 int arc_length = ((M_PI * theta) * (WIDTH * WIDTH) / (8));  
00153 }
```

Here is the call graph for this function:



6.79.1.5 calculate_gryo-angular_velocity()

```
static double calculate_gryo-angular_velocity ( ) [static]
```

calculates the angular velocity using the gyro positions

Author

Chris Jerrett, Christian DeSimone

Returns

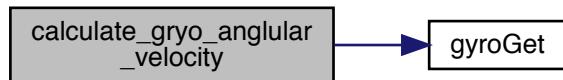
the angular velocity of the robot.

Definition at line **116** of file **localization.c**.

References **g1**, and **gyroGet()**.

```
00116     static int last_gyro = 0;
00117     int current = gyroGet(g1);
00118     // Calculate w (angular velocity in degrees per second)
00119     double w = (current - last_gyro) / (LOCALIZATION_UPDATE_FREQUENCY / 1000.0);
00120     return w;
00121 }
00122 }
```

Here is the call graph for this function:

**6.79.1.6 get_position()**

```
struct location get_position( )
```

Gets the current position of the robot.

Author

Chris Jerrett

Parameters

<i>gyro1</i>	The first gyro
--------------	----------------

Returns

the location of the robot as a struct.

Definition at line **38** of file **localization.c**.

```
00038 { }
```

6.79.1.7 init_localization()

```
bool init_localization (
    const unsigned char gyrol,
    unsigned short multiplier,
    int start_x,
    int start_y,
    int start_theta )
```

initializes the localization

Starts the localization process.

Author

Chris Jerrett

Returns

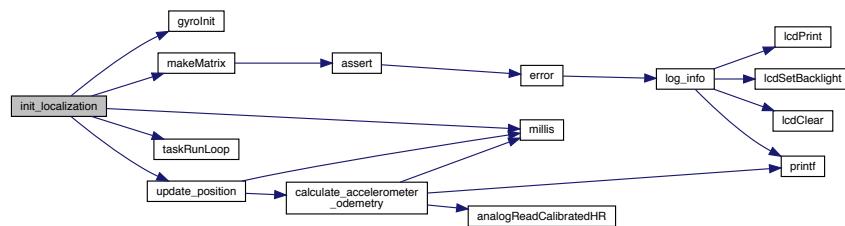
returns true when initialization is complete.

Definition at line **160** of file **localization.c**.

References **g1**, **gyroInit()**, **last_call**, **localization_task**, **makeMatrix()**, **millis()**, **taskRunLoop()**, and **update_position()**.

```
00161                                     {
00162     g1 = gyroInit(gyrol, multiplier);
00163     // init state matrix
00164
00165     // one dimensional vector with x, y, theta, acceleration in x and y
00166     state_matrix = makeMatrix(1, 5);
00167     localization_task =
00168         taskRunLoop(update_position, LOCALIZATION_UPDATE_FREQUENCY * 1000);
00169     last_call = millis();
00170     return true;
00171 }
```

Here is the call graph for this function:



6.79.1.8 integrate_gyro_w()

```
static double integrate_gyro_w (
    int new_w ) [static]
```

Increases the stored angle based upon the update frequency and the current angular velocity.

Author

Chris Jerrett

Returns

returns the angle theta of the robot

Definition at line **104** of file **localization.c**.

References **encoder_odemtry::theta**.

```
00104             {
00105     static double theta = 0;
00106     double delta_theta = new_w * LOCALIZATION_UPDATE_FREQUENCY;
00107     theta += delta_theta;
00108     return theta;
00109 }
```

6.79.1.9 update_position()

```
void update_position ( )
```

Updates the position from the localization.

Author

Chris Jerrett, Christian DeSimone

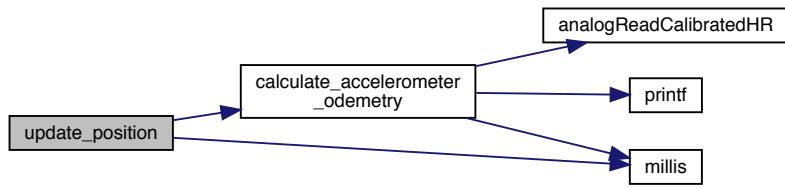
Definition at line **45** of file **localization.c**.

References **calculate_accelerometer_odometry()**, **last_call**, and **millis()**.

Referenced by **init_localization()**.

```
00045             {
00046     // int curr_theta = calculate_angle();
00047
00048     struct accelerometer_odometry oddem = calculate_accelerometer_odometry();
00049     // printf("x: %d y: %d T: %d\n", a.x, a.y, 0);
00050
00051     /*int l = 1;
00052     int vr = get_encoder_velocity(1);
00053     int vl = get_encoder_velocity(2);
00054     int theta_dot = (vr - vl) / l;
00055     int curr_theta = theta + theta_dot;
00056     double dt = LOCALIZATION_UPDATE_FREQUENCY;
00057     double v_tot = (vr+vl)/2.0;
00058     int x_curr = x - v_tot*dt*sin(curr_theta);
00059     int y_curr = y + v_tot*dt*cos(curr_theta);
00060     x = x_curr;
00061     y = y_curr; */
00062     last_call = millis();
00063 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.79.2 Variable Documentation

6.79.2.1 g1

```
Gyro g1 [static]
```

Definition at line 5 of file **localization.c**.

Referenced by `calculate_gryo-angular_velocity()`, and `init_localization()`.

6.79.2.2 last_call

```
int last_call = 0 [static]
```

Definition at line 8 of file **localization.c**.

Referenced by `calculate_accelerometer_odometry()`, `init_localization()`, and `update_position()`.

6.79.2.3 localization_task

```
TaskHandle localization_task [static]
```

Definition at line **6** of file **localization.c**.

Referenced by **init_localization()**.

6.79.2.4 state_matrix

```
matrix* state_matrix
```

Definition at line **10** of file **localization.c**.

6.80 localization.c

```
00001 #include "localization.h"
00002 #include "vmath.h"
00003 #include <inttypes.h>
00004
00005 static Gyro g1;
00006 static TaskHandle localization_task;
00007
00008 static int last_call = 0;
00009
00010 matrix *state_matrix;
00011
00015 struct encoder_odometry {
00016     double x;
00017     double y;
00018     double theta;
00019 };
00020
00024 struct accelerometer_odometry {
00025     double x;
00026     double y;
00027 };
00028
00029 static double calculate_angle();
00030 static struct accelerometer_odometry calculate_accelerometer_odometry();
00031
00038 struct location get_position() {}
00039
00045 void update_position() {
00046     // int curr_theta = calculate_angle();
00047
00048     struct accelerometer_odometry oddem = calculate_accelerometer_odometry();
00049     // printf("x: %d y: %d T: %d\n", a.x, a.y, 0);
00050
00051     /*int l = 1;
00052     int vr = get_encoder_velocity(1);
00053     int vl = get_encoder_velocity(2);
00054     int theta_dot = (vr - vl) / l;
00055     int curr_theta = theta + theta_dot;
00056     double dt = LOCALIZATION_UPDATE_FREQUENCY;
00057     double v_tot = (vr+vl)/2.0;
00058     int x_curr = x - v_tot*dt*sin(curr_theta);
00059     int y_curr = y + v_tot*dt*cos(curr_theta);
00060     x = x_curr;
00061     y = y_curr; */
00062     last_call = millis();
00063 }
00070 static struct accelerometer_odometry calculate_accelerometer_odometry() {
00071     static double vel_acumm_x = 0;
00072     static double vel_acumm_y = 0;
00073 }
```

```

00074     int32_t accel_x_rel = (int32_t)analogReadCalibratedHR(2);
00075     int32_t accel_y_rel = (int32_t)analogReadCalibratedHR(3);
00076
00077     // Ignore atom format string errors
00078     printf("x: %" PRIId32 " y: %" PRId32 "\n", accel_x_rel, accel_y_rel);
00079
00080     double delta_time = ((millis() - last_call) / 1000.0);
00081     // double accel_x_abs = (accel_x_rel * cos(theta) + accel_y_rel * sin(theta))
00082     // * delta_time;    double accel_y_abs = (accel_y_rel * cos(theta) +
00083     // accel_x_rel
00084     // * sin(theta)) * delta_time;
00085
00086     // vel_acumm_x += accel_x_abs;
00087     // vel_acumm_y += accel_y_abs;
00088
00089     // double new_x = x + vel_acumm_x * delta_time;
00090     // double new_y = y + vel_acumm_y * delta_time;
00091
00092     struct accelerometer_odometry od;
00093     // od.x = new_x;
00094     // od.y = new_y;
00095     return od;
00096 }
00097
00104 static double integrate_gyro_w(int new_w) {
00105     static double theta = 0;
00106     double delta_theta = new_w * LOCALIZATION_UPDATE_FREQUENCY;
00107     theta += delta_theta;
00108     return theta;
00109 }
00110
00116 static double calculate_gryo_angular_velocity() {
00117     static int last_gyro = 0;
00118     int current = gyroGet(g1);
00119     // Calculate w (angluar velocity in degrees per second)
00120     double w = (current - last_gyro) / (LOCALIZATION_UPDATE_FREQUENCY / 1000.0);
00121     return w;
00122 }
00123
00129 int calculate_encoder_angle() {
00130 #define WIDTH 13.5
00131 #define CPR 392.0
00132 #define WHEEL_RADIUS 2
00133     int dist_r = get_encoder_ticks(0) / CPR;
00134     int dist_l = get_encoder_ticks(1) / CPR;
00135     return ((dist_r - dist_l) / WIDTH);
00136 }
00137
00142 static void calculate_encoder_odometry() {
00143 #define WIDTH 13.5
00144 #define CPR 392.0
00145 #define WHEEL_RADIUS 2
00146
00147     int dist_r = get_encoder_ticks(0) / CPR;
00148     int dist_l = get_encoder_ticks(1) / CPR;
00149     printf("dist_r: %d dist_l: %d\n", dist_r, dist_l);
00150     int theta = (dist_l - dist_r) / WIDTH;
00151     printf("theta: %d\n", theta);
00152     int arc_length = ((M_PI * theta) * (WIDTH * WIDTH) / (8));
00153 }
00154
00160 bool init_localization(const unsigned char gyrol, unsigned short multiplier,
00161                         int start_x, int start_y, int start_theta) {
00162     g1 = gyroInit(gyrol, multiplier);
00163     // init state matrix
00164
00165     // one dimensional vector with x, y, theta, acceleration in x and y
00166     state_matrix = makeMatrix(1, 5);
00167     localization_task =
00168         taskRunLoop(update_position, LOCALIZATION_UPDATE_FREQUENCY * 1000);
00169     last_call = millis();
00170     return true;
00171 }

```

Functions

- void **debug** (const char *debug_message)
prints a info message
- void **error** (const char *error_message)
prints a error message and displays on lcd.
- void **info** (const char *info_message)
prints a info message
- void **init_error** (bool use_lcd, FILE *lcd)
Initializes the error lcd system Only required if using lcd.
- static void **log_info** (const char *s, const char *mess)
prints a log info message to the lcd
- void **warning** (const char *warning_message)
prints a warning message and displays on lcd.

Variables

- static FILE * **log_lcd** = NULL
- unsigned int **log_level** = INFO

6.81.1 Function Documentation

6.81.1.1 debug()

```
void debug (
    const char * debug_message )
```

prints a info message

Only will print and display if log_level is greater than info

See also

log_level (p. 279)

Parameters

<i>debug_message</i>	the message
----------------------	-------------

Definition at line 83 of file **log.c**.

References **log_level**, and **printf()**.

Referenced by **set_motor_immediate()**, and **set_motor_slew()**.

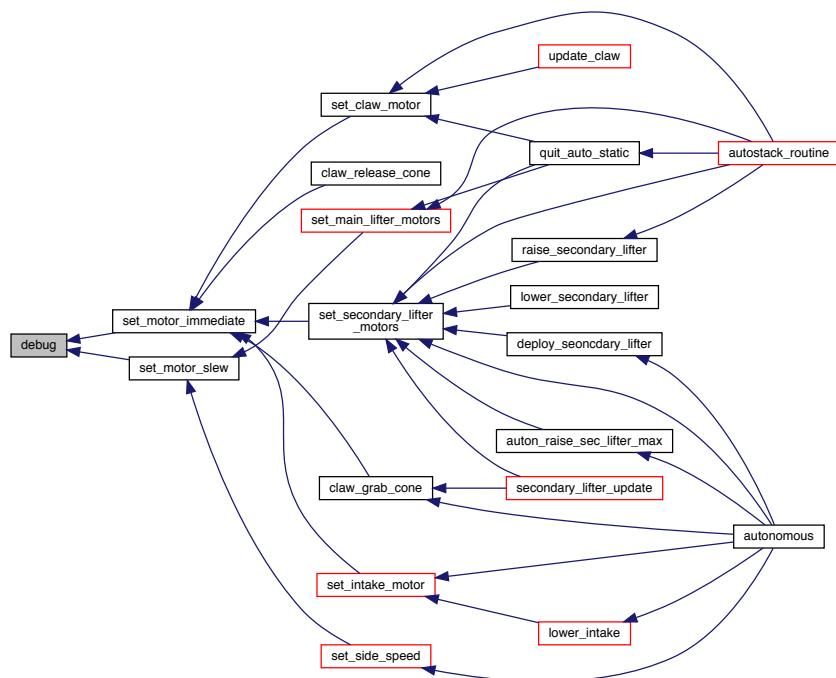
```

00083
00084     if (log_level > ERROR) {
00085         printf("[INFO]: %s\n", debug_message);
00086     }
00087 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.81.1.2 error()

```

void error (
    const char * error_message )
```

prints a error message and displays on lcd.

Only will print and display if log_level is greater than NONE

See also

[log_level](#) (p. 279)

Author

Chris Jerrett

Date

9/10/17

Parameters

<i>error_message</i>	the message
----------------------	-------------

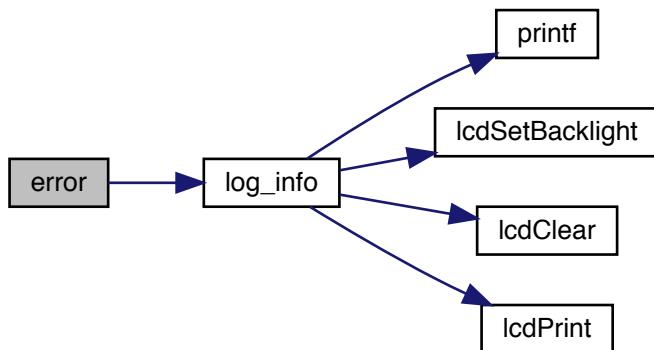
Definition at line **45** of file **log.c**.

References **log_info()**, and **log_level**.

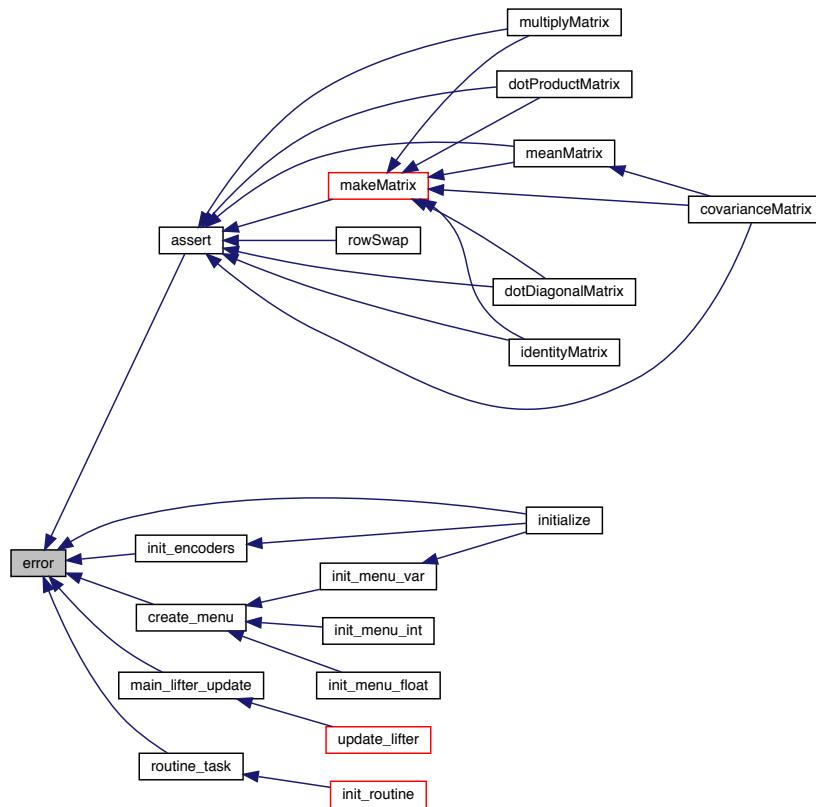
Referenced by **assert()**, **create_menu()**, **init_encoders()**, **initialize()**, **main_lifter_update()**, and **routine_task()**.

```
00045
00046     if (log_level > NONE)
00047         log_info("ERROR", error_message);
00048 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.81.1.3 info()

```
void info (
    const char * info_message )
```

prints a info message

Only will print and display if log_level is greater than ERROR

See also

[log_level](#) (p. 279)

Parameters

<i>info_message</i>	the message
---------------------	-------------

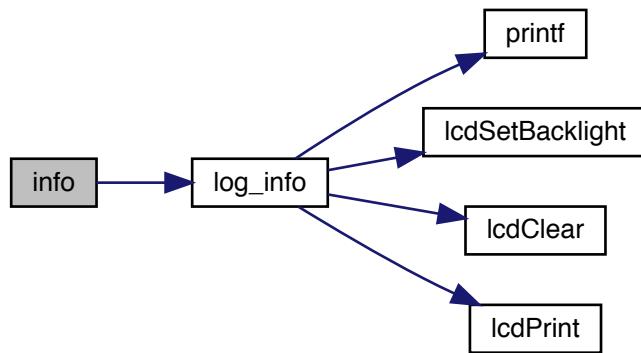
Definition at line **70** of file **log.c**.

References **log_info()**, and **log_level**.

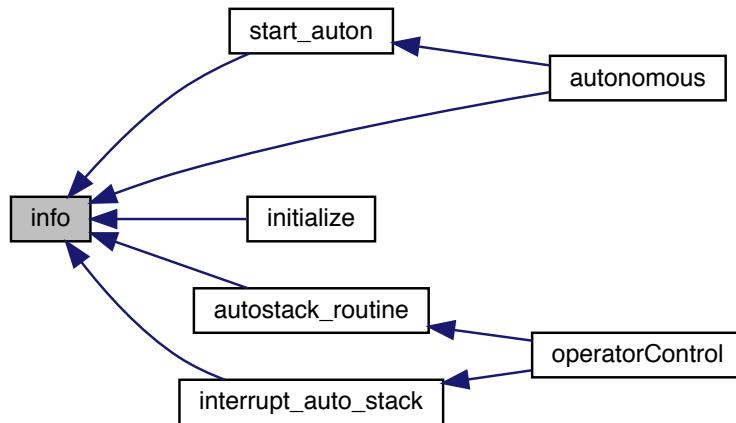
Referenced by **autonomous()**, **autostack_routine()**, **initialize()**, **interrupt_auto_stack()**, and **start_auton()**.

```
00070          {
00071      if (log_level > ERROR) {
00072          log_info("INFO", info_message);
00073      }
00074 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.81.1.4 init_error()

```
void init_error (
    bool use_lcd,
    FILE * lcd )
```

Initializes the error lcd system Only required if using lcd.

Author

Chris Jerrett

Date

9/10/17

Parameters

<i>use_lcd</i>	whether to use the lcd
<i>lcd</i>	the lcd

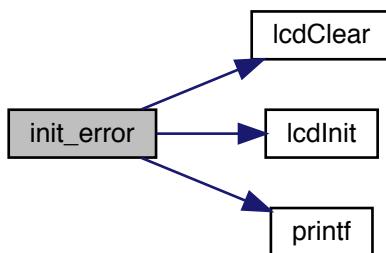
Definition at line 14 of file **log.c**.

References **LcdClear()**, **LcdInit()**, **log_lcd**, and **printf()**.

Referenced by **initialize()**.

```
00014     {
00015     if (use_lcd) {
00016         lcdInit(lcd);
00017         log_lcd = lcd;
00018         lcdClear(log_lcd);
00019         printf("LCD Init\n");
00020     }
00021 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.81.1.5 log_info()

```
static void log_info (
    const char * s,
    const char * mess ) [static]
```

prints a log info message to the lcd

Author

Chris Jerrett

Date

9/10/17

Parameters

<i>s</i>	the string on the top line
<i>mess</i>	the string on the bottom line

Definition at line **29** of file **log.c**.

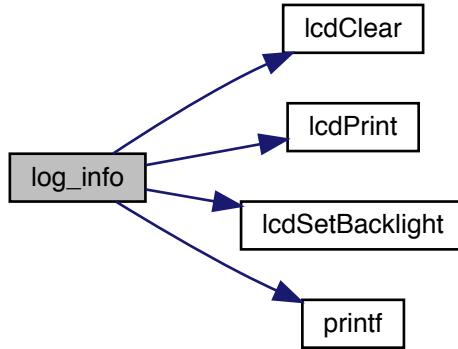
References **IcdClear()**, **IcdPrint()**, **IcdSetBacklight()**, **log_lcd**, and **printf()**.

Referenced by **error()**, **info()**, and **warning()**.

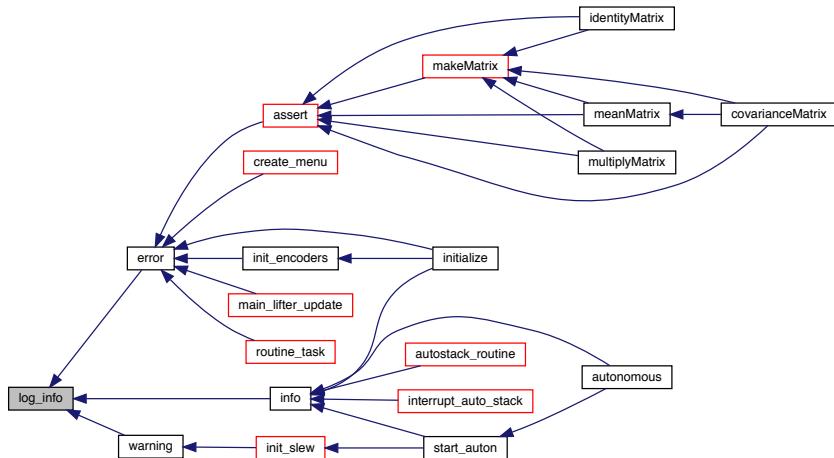
```

00029
00030     printf("[%s]: %s\n", s, mess);
00031     lcdSetBacklight(log_lcd, false);
00032     lcdClear(log_lcd);
00033     lcdPrint(log_lcd, TOP_ROW, s);
00034     lcdPrint(log_lcd, BOTTOM_ROW, mess);
00035 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.81.1.6 warning()

```
void warning (
    const char * warning_message )
```

prints a warning message and displays on lcd.

Only will print and display if log_level is greater than NONE

See also

[log_level](#) (p. 279)

Author

Chris Jerrett

Date

9/10/17

Parameters

<i>warning_message</i>	the message
------------------------	-------------

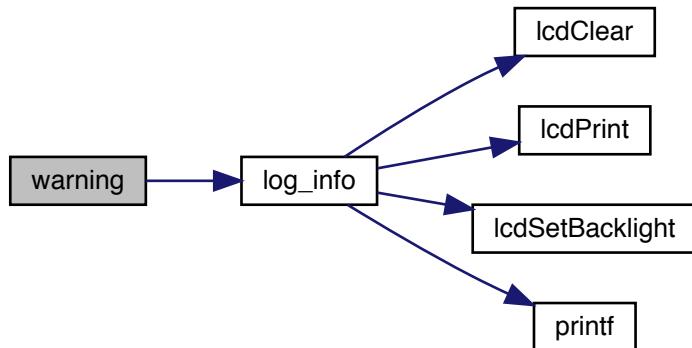
Definition at line **58** of file **log.c**.

References **log_info()**, and **log_level**.

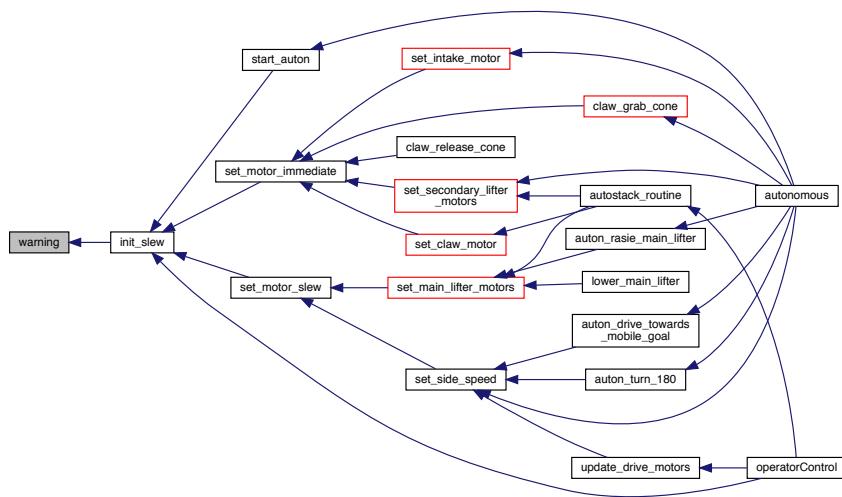
Referenced by **init_slew()**.

```
00058 {  
00059     if (log_level > WARNING)  
00060         log_info("WARNING", warning_message);  
00061 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.81.2 Variable Documentation

6.81.2.1 log_lcd

```
FILE* log_lcd = NULL [static]
```

Definition at line 4 of file **log.c**.

Referenced by **init_error()**, and **log_info()**.

6.81.2.2 log_level

```
unsigned int log_level = INFO
```

Definition at line 3 of file **log.c**.

Referenced by **debug()**, **error()**, **info()**, and **warning()**.

6.82 log.c

```

00001 #include "log.h"
00002
00003 unsigned int log_level = INFO;
00004 static FILE *log_lcd = NULL;
00005
00014 void init_error(bool use_lcd, FILE *lcd) {
00015     if (use_lcd) {
00016         lcdInit(lcd);
00017         log_lcd = lcd;
00018         lcdClear(log_lcd);
00019         printf("LCD Init\n");
00020     }
00021 }
00029 static void log_info(const char *s, const char *mess) {
00030     printf("[%s]: %s\n", s, mess);
00031     lcdSetBacklight(log_lcd, false);
00032     lcdClear(log_lcd);
00033     lcdPrint(log_lcd, TOP_ROW, s);
00034     lcdPrint(log_lcd, BOTTOM_ROW, mess);
00035 }
00036
00045 void error(const char *error_message) {
00046     if (log_level > NONE)
00047         log_info("ERROR", error_message);
00048 }
00049
00058 void warning(const char *warning_message) {
00059     if (log_level > WARNING)
00060         log_info("WARNING", warning_message);
00061 }
00062
00070 void info(const char *info_message) {
00071     if (log_level > ERROR) {
00072         log_info("INFO", info_message);
00073     }
00074 }
00075
00083 void debug(const char *debug_message) {
00084     if (log_level > ERROR) {
00085         printf("[INFO]: %s\n", debug_message);
00086     }
00087 }

```

6.83 src/matrix.c File Reference

Functions

- **void assert (int assertion, const char *message)**
Asserts a condition is true.
- **matrix * copyMatrix (matrix *m)**
Copies a matrix.
- **matrix * covarianceMatrix (matrix *m)**
returns the covariance of the matrix
- **matrix * dotDiagonalMatrix (matrix *a, matrix *b)**
performs a diagonal matrix dot product.
- **matrix * dotProductMatrix (matrix *a, matrix *b)**
returns the matrix dot product.
- **void freeMatrix (matrix *m)**
Frees the resources of a matrix.
- **matrix * identityMatrix (int n)**
Returns an identity matrix of size n by n.
- **matrix * makeMatrix (int width, int height)**

- **matrix * meanMatrix (matrix *m)**
Makes a matrix with a width and height parameters.
- **matrix * multiplyMatrix (matrix *a, matrix *b)**
Given an "m rows by n columns" matrix, return a matrix where each element represents the mean of that full column.
- **matrix * printMatrix (matrix *m)**
Given a two matrices, returns the multiplication of the two.
- **void rowSwap (matrix *a, int p, int q)**
Prints a matrix.
- **void scaleMatrix (matrix *m, double value)**
swaps the rows of a matrix.
- **double traceMatrix (matrix *m)**
scales a matrix.
- **matrix * transposeMatrix (matrix *m)**
Given an "m rows by n columns" matrix returns the sum.
- **matrix * transposeMatrix (matrix *m)**
returns the transpose matrix.

6.83.1 Function Documentation

6.83.1.1 assert()

```
void assert (
    int assertion,
    const char * message )
```

Asserts a condition is true.

If the assertion is non-zero (i.e. true), then it returns. If the assertion is zero (i.e. false), then it display the string and aborts the program. This is ment to act like Python's assert keyword.

Parameters

<i>assertion</i>	the condition, acts like a boolean 0 = false else true
<i>message</i>	the message to print if it fails

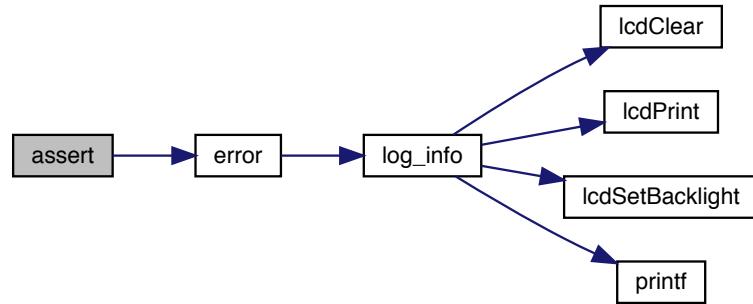
Definition at line 17 of file **matrix.c**.

References **error()**.

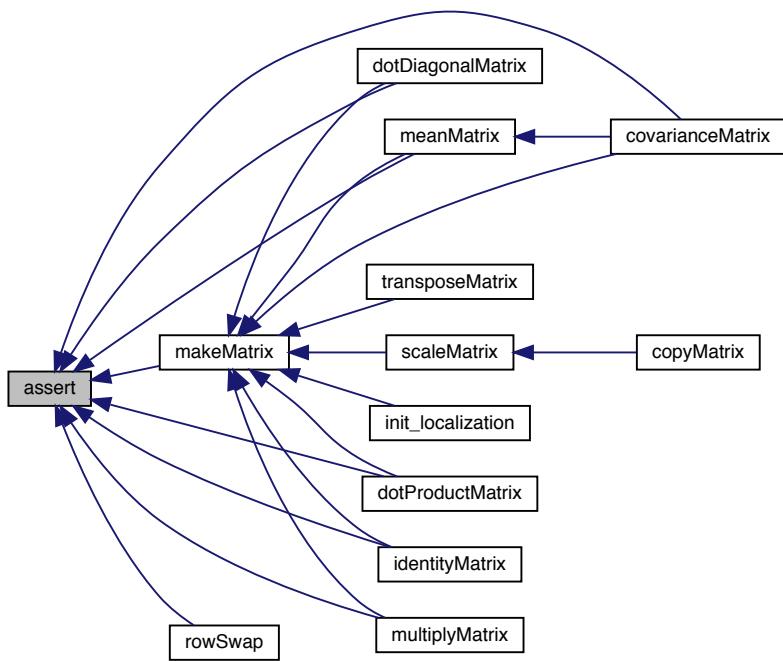
Referenced by **covarianceMatrix()**, **dotDiagonalMatrix()**, **dotProductMatrix()**, **identityMatrix()**, **makeMatrix()**, **meanMatrix()**, **multiplyMatrix()**, and **rowSwap()**.

```
00017
00018     if (assertion == 0) {
00019         error(message);
00020         exit(1);
00021     }
00022 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.83.1.2 copyMatrix()

```
matrix* copyMatrix (
    matrix * m )
```

Copies a matrix.

This function uses scaleMatrix, because scaling matrix by 1 is the same as a copy.

Parameters

<i>m</i>	a pointer to the matrix
----------	-------------------------

Returns

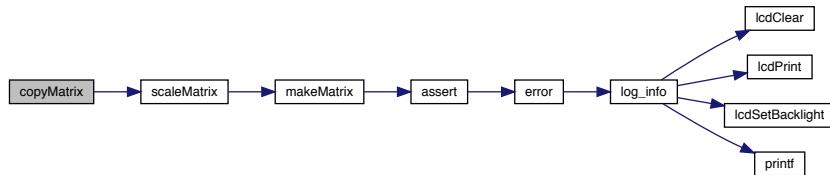
a copied matrix

Definition at line **55** of file **matrix.c**.

References **scaleMatrix()**.

```
00055 { return scaleMatrix(m, 1); }
```

Here is the call graph for this function:



6.83.1.3 covarianceMatrix()

```
matrix* covarianceMatrix (
    matrix * m )
```

returns the covariance of the matrix

Parameters

<i>the</i>	matrix
------------	--------

Returns

a matrix with n row and n columns, where each element represents covariance of 2 columns.

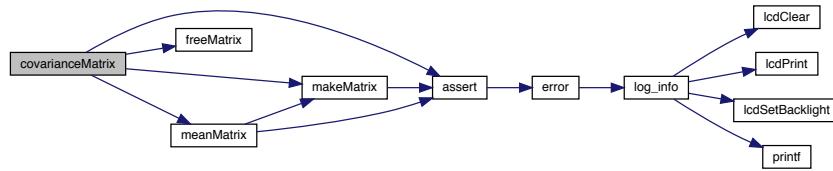
Definition at line 171 of file **matrix.c**.

References **assert()**, **_matrix::data**, **freeMatrix()**, **_matrix::height**, **makeMatrix()**, **meanMatrix()**, and **_matrix::width**.

```

00171      {
00172      int i, j, k = 0;
00173      matrix *out;
00174      matrix *mean;
00175      double *ptrA;
00176      double *ptrB;
00177      double *ptrOut;
00178
00179      assert(m->height > 1, "Height of matrix cannot be zero or one.");
00180
00181      mean = meanMatrix(m);
00182      out = makeMatrix(m->width, m->width);
00183      ptrOut = out->data;
00184
00185      for (i = 0; i < m->width; i++) {
00186          for (j = 0; j < m->width; j++) {
00187              ptrA = &m->data[i];
00188              ptrB = &m->data[j];
00189              *ptrOut = 0.0;
00190              for (k = 0; k < m->height; k++) {
00191                  *ptrOut += (*ptrA - mean->data[i]) * (*ptrB - mean->data[j]);
00192                  ptrA += m->width;
00193                  ptrB += m->width;
00194              }
00195              *ptrOut /= m->height - 1;
00196              ptrOut++;
00197          }
00198      }
00199
00200      freeMatrix(mean);
00201      return out;
00202  }
```

Here is the call graph for this function:



6.83.1.4 dotDiagonalMatrix()

```

matrix* dotDiagonalMatrix (
    matrix * a,
    matrix * b )
```

performs a diagonal matrix dot product.

Given a two matrices (or the same matrix twice) with identical widths and heights, this method returns a 1 by a->height matrix of the cross product of each matrix along the diagonal.

Dot product is essentially the sum-of-squares of two vectors.

If the second parameter is NULL, it is assumed that we are performing a cross product with itself.

Parameters

<i>a</i>	the first matrix
<i>b</i>	the second matrix

Returns

the matrix result

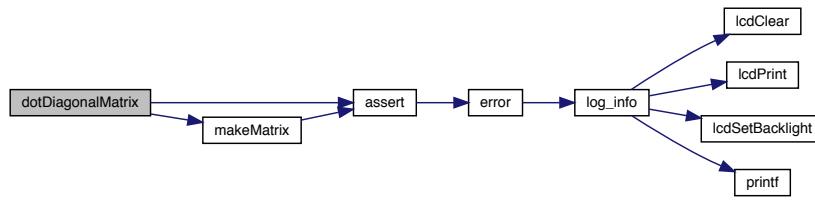
Definition at line 391 of file **matrix.c**.

References **assert()**, **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

```

00391
00392     matrix *out;
00393     double *ptrOut;
00394     double *ptrA;
00395     double *ptrB;
00396     int i, j;
00397
00398     if (b != NULL) {
00399         assert(a->width == b->width && a->height == b->height,
00400                 "Matrices must be of the same dimensionality.");
00401     }
00402
00403     // Are we computing the sum of squares of the same matrix?
00404     if (a == b || b == NULL) {
00405         b = a; // May not appear safe, but we can do this without risk of losing b.
00406     }
00407
00408     out = makeMatrix(1, a->height);
00409     ptrOut = out->data;
00410     ptrA = a->data;
00411     ptrB = b->data;
00412
00413     for (i = 0; i < a->height; i++) {
00414         *ptrOut = 0;
00415         for (j = 0; j < a->width; j++) {
00416             *ptrOut += *ptrA * *ptrB;
00417             ptrA++;
00418             ptrB++;
00419         }
00420         ptrOut++;
00421     }
00422
00423     return out;
00424 }
```

Here is the call graph for this function:



6.83.1.5 dotProductMatrix()

```
matrix* dotProductMatrix (
    matrix * a,
    matrix * b )
```

returns the matrix dot product.

Given a two matrices (or the same matrix twice) with identical widths and different heights, this method returns a $a->\text{height} \times b->\text{height}$ matrix of the cross product of each matrix.

Dot product is essentially the sum-of-squares of two vectors.

Also, if the second parameter is NULL, it is assumed that we are performing a cross product with itself.

Parameters

<i>a</i>	the first matrix
<i>the</i>	second matrix

Returns

the result of the dot product

Definition at line **338** of file **matrix.c**.

References **assert()**, **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

```

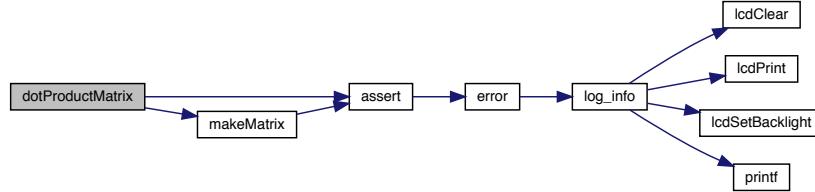
00338
00339     matrix *out;
00340     double *ptrOut;
00341     double *ptrA;
00342     double *ptrB;
00343     int i, j, k;
00344
00345     if (b != NULL) {
00346         assert(a->width == b->width,
  
```

```

00347         "Matrices must be of the same dimensionality.");
00348     }
00349
00350 // Are we computing the sum of squares of the same matrix?
00351 if (a == b || b == NULL) {
00352     b = a; // May not appear safe, but we can do this without risk of losing b.
00353 }
00354
00355 out = makeMatrix(b->height, a->height);
00356 ptrOut = out->data;
00357
00358 for (i = 0; i < a->height; i++) {
00359     ptrB = b->data;
00360
00361     for (j = 0; j < b->height; j++) {
00362         ptrA = &a->data[i * a->width];
00363
00364         *ptrOut = 0;
00365         for (k = 0; k < a->width; k++) {
00366             *ptrOut += *ptrA * *ptrB;
00367             ptrA++;
00368             ptrB++;
00369         }
00370         ptrOut++;
00371     }
00372 }
00373
00374 return out;
00375 }

```

Here is the call graph for this function:



6.83.1.6 freeMatrix()

```

void freeMatrix (
    matrix * m )

```

Frees the resources of a matrix.

Parameters

<i>the</i>	matrix to free
------------	----------------

Definition at line 61 of file **matrix.c**.

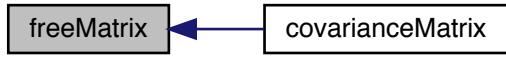
References **_matrix::data**.

Referenced by **covarianceMatrix()**.

```

00061
00062     if (m != NULL) {
00063         if (m->data != NULL) {
00064             free(m->data);
00065             m->data = NULL;
00066         }
00067         free(m);
00068     }
00069     return;
00070 }
```

Here is the caller graph for this function:



6.83.1.7 identityMatrix()

```
matrix* identityMatrix (
    int n )
```

Returns an identity matrix of size n by n.

Parameters

<i>n</i>	the input matrix.
----------	-------------------

Returns

the identity matrix parameter.

Definition at line 95 of file **matrix.c**.

References **assert()**, **_matrix::data**, and **makeMatrix()**.

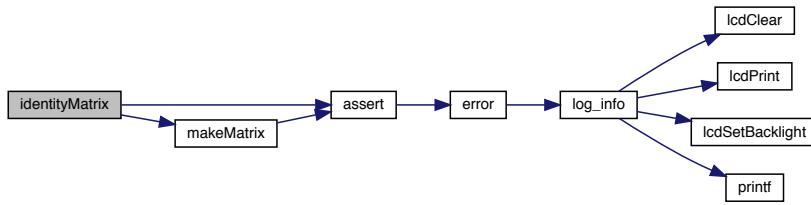
```

00095
00096     int i;
00097     matrix *out;
00098     double *ptr;
00099
00100    assert(n > 0, "Identity matrix must have value greater than zero.");
00101 }
```

```

00102     out = makeMatrix(n, n);
00103     ptr = out->data;
00104     for (i = 0; i < n; i++) {
00105         *ptr = 1.0;
00106         ptr += n + 1;
00107     }
00108
00109     return out;
00110 }
```

Here is the call graph for this function:



6.83.1.8 makeMatrix()

```
matrix* makeMatrix (
    int width,
    int height )
```

Makes a matrix with a width and height parameters.

Parameters

<i>width</i>	The width of the matrix
<i>height</i>	the height of the matrix

Returns

the new matrix

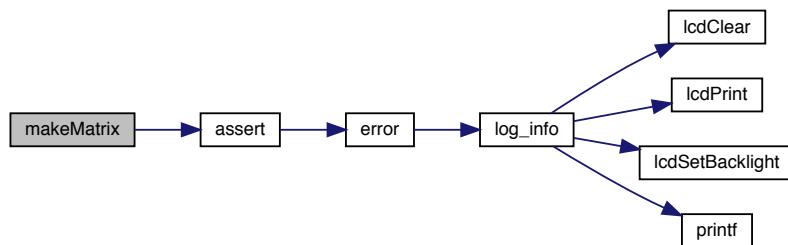
Definition at line **30** of file **matrix.c**.

References **assert()**, **_matrix::data**, **_matrix::height**, and **_matrix::width**.

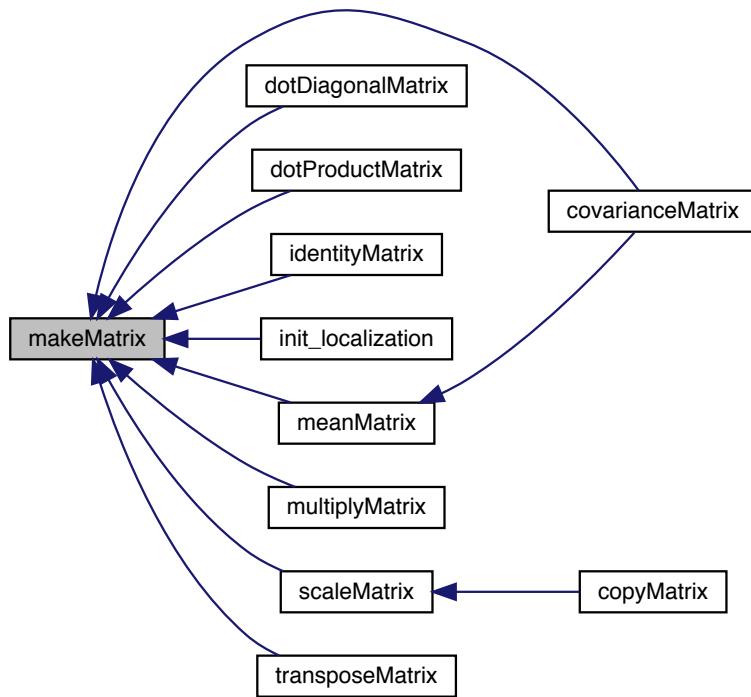
Referenced by **covarianceMatrix()**, **dotDiagonalMatrix()**, **dotProductMatrix()**, **identityMatrix()**, **init_localization()**, **meanMatrix()**, **multiplyMatrix()**, **scaleMatrix()**, and **transposeMatrix()**.

```
00030     matrix *out;
00031     assert(width > 0 && height > 0, "New matrix must be at least a 1 by 1");
00032     out = (matrix *)malloc(sizeof(matrix));
00033
00034     assert(out != NULL, "Out of memory.");
00035
00036     out->width = width;
00037     out->height = height;
00038     out->data = (double *)malloc(sizeof(double) * width * height);
00039
00040     assert(out->data != NULL, "Out of memory.");
00041
00042     memset(out->data, 0.0, width * height * sizeof(double));
00043
00044     return out;
00045 }
00046 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.83.1.9 meanMatrix()

```
matrix* meanMatrix (
    matrix * m )
```

Given an "m rows by n columns" matrix, return a matrix where each element represents the mean of that full column.

the matrix

Returns

matrix with 1 row and n columns each element represents the mean of that full column.

Definition at line 144 of file **matrix.c**.

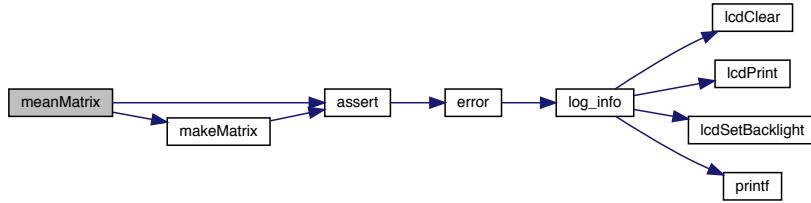
References **assert()**, **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

Referenced by **covarianceMatrix()**.

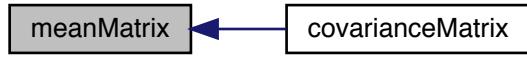
```

00144     int i, j;
00145     matrix *out;
00146
00147     assert(m->height > 0, "Height of matrix cannot be zero.");
00148
00149     out = makeMatrix(m->width, 1);
00150
00151     for (i = 0; i < m->width; i++) {
00152         double *ptr;
00153         out->data[i] = 0.0;
00154         ptr = &m->data[i];
00155         for (j = 0; j < m->height; j++) {
00156             out->data[i] += *ptr;
00157             ptr += out->width;
00158         }
00159     }
00160     out->data[i] /= (double)m->height;
00161 }
00162 return out;
00163 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.83.1.10 multiplyMatrix()

```

matrix* multiplyMatrix (
    matrix * a,
    matrix * b )
```

Given a two matrices, returns the multiplication of the two.

Parameters

<i>a</i>	the first matrix
<i>b</i>	the second matrix return the result of the multiplication

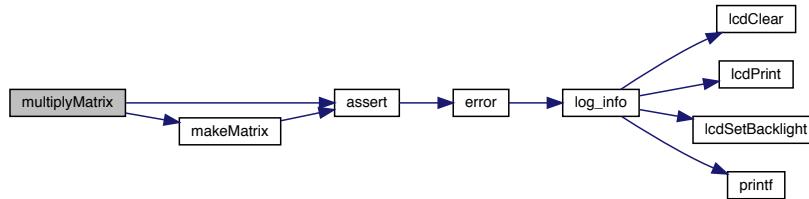
Definition at line 233 of file **matrix.c**.

References **assert()**, **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

```

00233     int i, j, k;
00234     int i, j, k;
00235     matrix *out;
00236     double *ptrOut;
00237     double *ptrA;
00238     double *ptrB;
00239
00240     assert(a->width == b->height,
00241             "Matrices have incorrect dimensions. a->width != b-> height");
00242
00243     out = makeMatrix(b->width, a-> height);
00244     ptrOut = out->data;
00245
00246     for (i = 0; i < a-> height; i++) {
00247
00248         for (j = 0; j < b-> width; j++) {
00249             ptrA = &a->data[i * a-> width];
00250             ptrB = &b->data[j];
00251
00252             *ptrOut = 0;
00253             for (k = 0; k < a-> width; k++) {
00254                 *ptrOut += *ptrA * *ptrB;
00255                 ptrA++;
00256                 ptrB += b-> width;
00257             }
00258             ptrOut++;
00259         }
00260     }
00261
00262     return out;
00263 }
```

Here is the call graph for this function:



6.83.1.11 printMatrix()

```
void printMatrix (
    matrix * m )
```

Prints a matrix.

Parameters

<i>the</i>	matrix
------------	--------

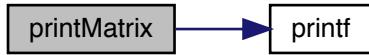
Definition at line **76** of file **matrix.c**.

References **_matrix::data**, **_matrix::height**, **printf()**, and **_matrix::width**.

```

00076
00077     int i, j;
00078     double *ptr = m->data;
00079     printf("%d %d\n", m->width, m->height);
00080     for (i = 0; i < m->height; i++) {
00081         for (j = 0; j < m->width; j++) {
00082             printf(" %.6f", *(ptr++));
00083         }
00084         printf("\n");
00085     }
00086     return;
00087 }
```

Here is the call graph for this function:

**6.83.1.12 rowSwap()**

```

void rowSwap (
    matrix * a,
    int p,
    int q )
```

swaps the rows of a matrix.

This method changes the input matrix. Given a matrix, this algorithm will swap rows p and q, provided that p and q are less than or equal to the height of matrix A and p and q are different values.

Parameters

<i>the</i>	matrix to swap. This method changes the input matrix.
<i>the</i>	first row
<i>the</i>	second row

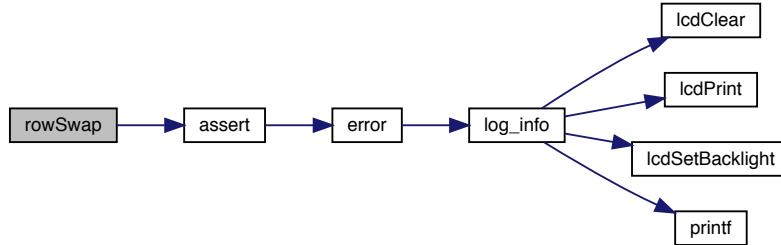
Definition at line 294 of file **matrix.c**.

References **assert()**, **_matrix::data**, **_matrix::height**, and **_matrix::width**.

```

00294     {
00295     int i;
00296     double temp;
00297     double *pRow;
00298     double *qRow;
00299
00300     assert(a->height > 2, "Matrix must have at least two rows to swap.");
00301     assert(p < a->height && q < a->height,
00302             "Values p and q must be less than the height of the matrix.");
00303
00304     // If p and q are equal, do nothing.
00305     if (p == q) {
00306         return;
00307     }
00308
00309     pRow = a->data + (p * a->width);
00310     qRow = a->data + (q * a->width);
00311
00312     // Swap!
00313     for (i = 0; i < a->width; i++) {
00314         temp = *pRow;
00315         *pRow = *qRow;
00316         *qRow = temp;
00317         pRow++;
00318         qRow++;
00319     }
00320
00321     return;
00322 }
```

Here is the call graph for this function:



6.83.1.13 scaleMatrix()

```

matrix* scaleMatrix (
    matrix * m,
    double value )
```

scales a matrix.

Parameters

<i>m</i>	the matrix to scale
<i>the</i>	value to scale by

Returns

a new matrix where each element in the input matrix is multiplied by the scalar value

Definition at line **272** of file **matrix.c**.

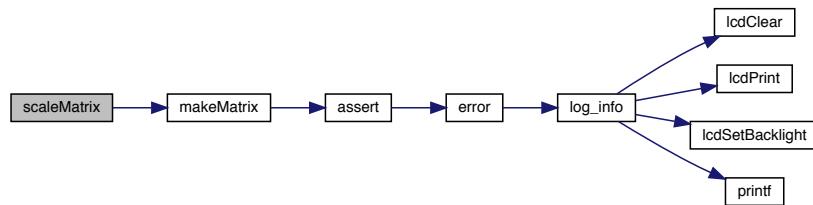
References **_matrix::data**, **_matrix::height**, **makeMatrix()**, and **_matrix::width**.

Referenced by **copyMatrix()**.

```

00272
00273     int i, elements = m->width * m->height;
00274     matrix *out = makeMatrix(m->width, m->height);
00275     double *ptrM = m->data;
00276     double *ptrOut = out->data;
00277
00278     for (i = 0; i < elements; i++) {
00279         * (ptrOut++) = *(ptrM++) * value;
00280     }
00281
00282     return out;
00283 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.83.1.14 traceMatrix()

```
double traceMatrix (
    matrix * m )
```

Given an "m rows by n columns" matrix returns the sum.

Given an "m rows by n columns" matrix.

Returns

the sum of the elements along the diagonal.

Definition at line 117 of file **matrix.c**.

References **_matrix::data**, **_matrix::height**, and **_matrix::width**.

```
00117     int i;
00118     int size;
00119     double *ptr = m->data;
00120     double sum = 0.0;
00121
00122
00123     if (m->height < m->width) {
00124         size = m->height;
00125     } else {
00126         size = m->width;
00127     }
00128
00129     for (i = 0; i < size; i++) {
00130         sum += *ptr;
00131         ptr += m->width + 1;
00132     }
00133
00134     return sum;
00135 }
```

6.83.1.15 transposeMatrix()

```
matrix* transposeMatrix (
    matrix * m )
```

returns the transpose matrix.

Parameters

<i>the</i>	matrix to transpose.
------------	----------------------

Returns

the transposed matrix.

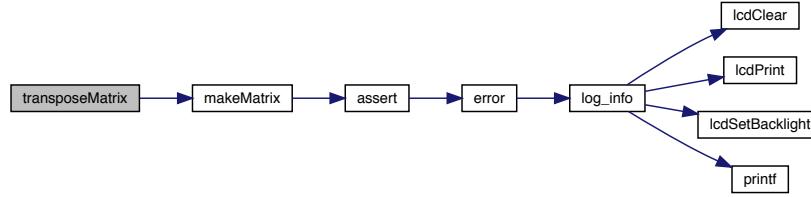
Definition at line 209 of file **matrix.c**.

References `_matrix::data`, `_matrix::height`, `makeMatrix()`, and `_matrix::width`.

```

00209             {
00210     matrix *out = makeMatrix(m->height, m->width);
00211     double *ptrM = m->data;
00212     int i, j;
00213
00214     for (i = 0; i < m->height; height; i++) {
00215         double *ptrOut;
00216         ptrOut = &out->data[i];
00217         for (j = 0; j < m->width; j++) {
00218             *ptrOut = *ptrM;
00219             ptrM++;
00220             ptrOut += out->width;
00221         }
00222     }
00223
00224     return out;
00225 }
```

Here is the call graph for this function:



6.84 matrix.c

```

00001 #include "matrix.h"
00002 #include "log.h"
00003 #include <API.h>
00004 #include <stdlib.h>
00005 #include <string.h>
00006
00017 void assert(int assertion, const char *message) {
00018     if (assertion == 0) {
00019         error(message);
00020         exit(1);
00021     }
00022 }
00023
00030 matrix *makeMatrix(int width, int height) {
00031     matrix *out;
00032     assert(width > 0 && height > 0, "New matrix must be at least a 1 by 1");
00033     out = (matrix *)malloc(sizeof(matrix));
00034
00035     assert(out != NULL, "Out of memory.");
00036
00037     out->width = width;
00038     out->height = height;
00039     out->data = (double *)malloc(sizeof(double) * width * height);
00040
00041     assert(out->data != NULL, "Out of memory.");
00042
00043     memset(out->data, 0.0, width * height * sizeof(double));
00044
00045     return out;
00046 }
```

```
00055 matrix *copyMatrix(matrix *m) { return scaleMatrix(m, 1); }
00056
00061 void freeMatrix(matrix *m) {
00062     if (m != NULL) {
00063         if (m->data != NULL) {
00064             free(m->data);
00065             m->data = NULL;
00066         }
00067         free(m);
00068     }
00069     return;
00070 }
00071
00076 void printMatrix(matrix *m) {
00077     int i, j;
00078     double *ptr = m->data;
00079     printf("%d %d\n", m->width, m->height);
00080     for (i = 0; i < m->height; i++) {
00081         for (j = 0; j < m->width; j++) {
00082             printf(" %.6f", *(ptr++));
00083         }
00084         printf("\n");
00085     }
00086     return;
00087 }
00088
00095 matrix *identityMatrix(int n) {
00096     int i;
00097     matrix *out;
00098     double *ptr;
00099
00100    assert(n > 0, "Identity matrix must have value greater than zero.");
00101
00102    out = makeMatrix(n, n);
00103    ptr = out->data;
00104    for (i = 0; i < n; i++) {
00105        *ptr = 1.0;
00106        ptr += n + 1;
00107    }
00108
00109    return out;
00110 }
00111
00117 double traceMatrix(matrix *m) {
00118     int i;
00119     int size;
00120     double *ptr = m->data;
00121     double sum = 0.0;
00122
00123     if (m->height < m->width) {
00124         size = m->height;
00125     } else {
00126         size = m->width;
00127     }
00128
00129     for (i = 0; i < size; i++) {
00130         sum += *ptr;
00131         ptr += m->width + 1;
00132     }
00133
00134     return sum;
00135 }
00136
00144 matrix *meanMatrix(matrix *m) {
00145     int i, j;
00146     matrix *out;
00147
00148     assert(m->height > 0, "Height of matrix cannot be zero.");
00149
00150     out = makeMatrix(m->width, 1);
00151
00152     for (i = 0; i < m->width; i++) {
00153         double *ptr;
00154         out->data[i] = 0.0;
00155         ptr = &m->data[i];
00156         for (j = 0; j < m->height; j++) {
00157             out->data[i] += *ptr;
00158             ptr += out->width;
00159         }
00160         out->data[i] /= (double)m->height;
00161     }
```

```

00162     return out;
00163 }
00164
00171 matrix *covarianceMatrix(matrix *m) {
00172     int i, j, k = 0;
00173     matrix *out;
00174     matrix *mean;
00175     double *ptrA;
00176     double *ptrB;
00177     double *ptrOut;
00178
00179     assert(m->height > 1, "Height of matrix cannot be zero or one.");
00180
00181     mean = meanMatrix(m);
00182     out = makeMatrix(m->width, m->width);
00183     ptrOut = out->data;
00184
00185     for (i = 0; i < m->width; i++) {
00186         for (j = 0; j < m->width; j++) {
00187             ptrA = &m->data[i];
00188             ptrB = &m->data[j];
00189             *ptrOut = 0.0;
00190             for (k = 0; k < m->height; k++) {
00191                 *ptrOut += (*ptrA - mean->data[i]) * (*ptrB - mean->data[j]);
00192                 ptrA += m->width;
00193                 ptrB += m->width;
00194             }
00195             *ptrOut /= m->height - 1;
00196             ptrOut++;
00197         }
00198     }
00199
00200     freeMatrix(mean);
00201     return out;
00202 }
00203
00209 matrix *transposeMatrix(matrix *m) {
00210     matrix *out = makeMatrix(m->height, m->width);
00211     double *ptrM = m->data;
00212     int i, j;
00213
00214     for (i = 0; i < m->height; i++) {
00215         double *ptrOut;
00216         ptrOut = &out->data[i];
00217         for (j = 0; j < m->width; j++) {
00218             *ptrOut = *ptrM;
00219             ptrM++;
00220             ptrOut += out->width;
00221         }
00222     }
00223
00224     return out;
00225 }
00226
00233 matrix *multiplyMatrix(matrix *a, matrix *b) {
00234     int i, j, k;
00235     matrix *out;
00236     double *ptrOut;
00237     double *ptrA;
00238     double *ptrB;
00239
00240     assert(a->width == b->height,
00241            "Matrices have incorrect dimensions. a->width != b->height");
00242
00243     out = makeMatrix(b->width, a->height);
00244     ptrOut = out->data;
00245
00246     for (i = 0; i < a->height; i++) {
00247
00248         for (j = 0; j < b->width; j++) {
00249             ptrA = &a->data[i * a->width];
00250             ptrB = &b->data[j];
00251
00252             *ptrOut = 0;
00253             for (k = 0; k < a->width; k++) {
00254                 *ptrOut += *ptrA * *ptrB;
00255                 ptrA++;
00256                 ptrB += b->width;
00257             }
00258             ptrOut++;
00259         }

```

```
00260      }
00261
00262     return out;
00263 }
00264
00272 matrix *scaleMatrix(matrix *m, double value) {
00273     int i, elements = m->width * m->height;
00274     matrix *out = makeMatrix(m->width, m->height);
00275     double *ptrM = m->data;
00276     double *ptrOut = out->data;
00277
00278     for (i = 0; i < elements; i++) {
00279         *(ptrOut++) = *(ptrM++) * value;
00280     }
00281
00282     return out;
00283 }
00284
00294 void rowSwap(matrix *a, int p, int q) {
00295     int i;
00296     double temp;
00297     double *pRow;
00298     double *qRow;
00299
00300     assert(a->height > 2, "Matrix must have at least two rows to swap.");
00301     assert(p < a->height && q < a->height,
00302             "Values p and q must be less than the height of the matrix.");
00303
00304     // If p and q are equal, do nothing.
00305     if (p == q) {
00306         return;
00307     }
00308
00309     pRow = a->data + (p * a->width);
00310     qRow = a->data + (q * a->width);
00311
00312     // Swap!
00313     for (i = 0; i < a->width; i++) {
00314         temp = *pRow;
00315         *pRow = *qRow;
00316         *qRow = temp;
00317         pRow++;
00318         qRow++;
00319     }
00320
00321     return;
00322 }
00323
00338 matrix *dotProductMatrix(matrix *a, matrix *b) {
00339     matrix *out;
00340     double *ptrOut;
00341     double *ptrA;
00342     double *ptrB;
00343     int i, j, k;
00344
00345     if (b != NULL) {
00346         assert(a->width == b->width,
00347                 "Matrices must be of the same dimensionality.");
00348     }
00349
00350     // Are we computing the sum of squares of the same matrix?
00351     if (a == b || b == NULL) {
00352         b = a; // May not appear safe, but we can do this without risk of losing b.
00353     }
00354
00355     out = makeMatrix(b->height, a->height);
00356     ptrOut = out->data;
00357
00358     for (i = 0; i < a->height; i++) {
00359         ptrB = b->data;
00360
00361         for (j = 0; j < b->height; j++) {
00362             ptrA = &a->data[i * a->width];
00363
00364             *ptrOut = 0;
00365             for (k = 0; k < a->width; k++) {
00366                 *ptrOut += *ptrA * *ptrB;
00367                 ptrA++;
00368                 ptrB++;
00369             }
00370             ptrOut++;
00370 }
```

```

00371     }
00372 }
00373
00374     return out;
00375 }
00376
00391 matrix *dotDiagonalMatrix(matrix *a, matrix *b) {
00392     matrix *out;
00393     double *ptrOut;
00394     double *ptrA;
00395     double *ptrB;
00396     int i, j;
00397
00398     if (b != NULL) {
00399         assert(a->width == b->width && a->height == b->height,
00400                 "Matrices must be of the same dimensionality.");
00401     }
00402
00403 // Are we computing the sum of squares of the same matrix?
00404 if (a == b || b == NULL) {
00405     b = a; // May not appear safe, but we can do this without risk of losing b.
00406 }
00407
00408 out = makeMatrix(1, a->height);
00409 ptrOut = out->data;
00410 ptrA = a->data;
00411 ptrB = b->data;
00412
00413 for (i = 0; i < a->height; i++) {
00414     *ptrOut = 0;
00415     for (j = 0; j < a->width; j++) {
00416         *ptrOut += *ptrA * *ptrB;
00417         ptrA++;
00418         ptrB++;
00419     }
00420     ptrOut++;
00421 }
00422
00423 return out;
00424 }
```

6.85 src/menu.c File Reference

Functions

- static void **calculate_current_display** (char *rtn, **menu_t** *menu)

Static function that calculates the string from menu.
- static **menu_t** * **create_menu** (enum **menu_type** type, const char *prompt)

Static function that handles creation of menu.
- void **denint_menu** (**menu_t** *menu)

Destroys a menu. Menu must be freed or will cause memory leak
- int **display_menu** (**menu_t** *menu)

Displays a menu context.
- **menu_t** * **init_menu_float** (enum **menu_type** type, float **min**, float **max**, float step, const char *prompt)

Creates a menu context, but does not display.
- **menu_t** * **init_menu_int** (enum **menu_type** type, int **min**, int **max**, int step, const char *prompt)

Creates a menu context, but does not display.
- **menu_t** * **init_menu_var** (enum **menu_type** type, const char *prompt, int nums,...)

Creates a menu context, but does not display.

6.85.1 Function Documentation

6.85.1.1 calculate_current_display()

```
static void calculate_current_display (
    char * rtn,
    menu_t * menu ) [static]
```

Static function that calculates the string from menu.

Parameters

<i>rtn</i>	the string to be written to
<i>menu</i>	the menu for the display to be calculated from

Author

Chris Jerrett

Date

9/8/17

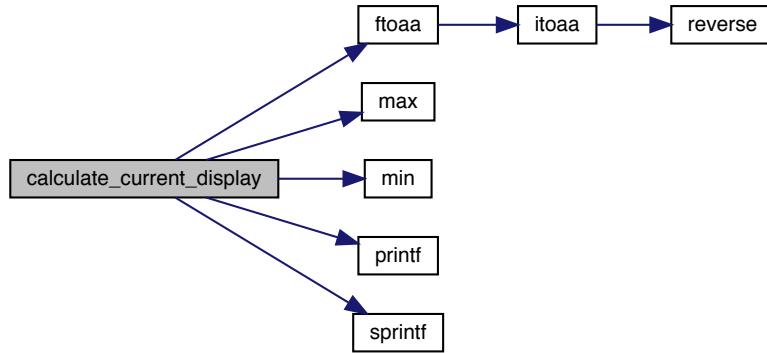
Definition at line **120** of file **menu.c**.

References **menu_t::current**, **FLOAT_TYPE**, **ftoaa()**, **INT_TYPE**, **menu_t::length**, **max()**, **menu_t::max**, **menu_t::max_f**, **min()**, **menu_t::min**, **menu_t::min_f**, **menu_t::options**, **printf()**, **sprintf()**, **menu_t::step**, **menu_t::step_f**, **STRING_TYPE**, and **menu_t::type**.

Referenced by **display_menu()**.

```
00120                                     {
00121     if (menu->type == STRING_TYPE) {
00122         int index = menu->current % menu->length;
00123         sprintf(rtn, "%s", menu->options[index]);
00124         printf("%s\n", rtn);
00125         return;
00126     }
00127     if (menu->type == INT_TYPE) {
00128         int step = (menu->step);
00129         int min = (menu->min);
00130         int max = (menu->max);
00131         int value = menu->current * step;
00132         if (value < min) {
00133             value = min;
00134             menu->current++;
00135         }
00136         if (value > max) {
00137             value = max;
00138             menu->current--;
00139         }
00140         sprintf(rtn, "%d", value);
00141     }
00142     if (menu->type == FLOAT_TYPE) {
00143         float step = (menu->step_f);
00144         float min = (menu->min_f);
00145         float max = (menu->max_f);
00146         float value = menu->current * step;
00147         value = value < min ? min : value;
00148         value = value > max ? max : value;
00149
00150         ftoaa(value, rtn, 5);
00151     }
00152 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.85.1.2 `create_menu()`

```
static menu_t * create_menu (
    enum menu_type type,
    const char * prompt ) [static]
```

Static function that handles creation of menu.

Menu must be freed or will cause memory leak

Author

Chris Jerrett

Date

9/8/17

Definition at line 27 of file **menu.c**.

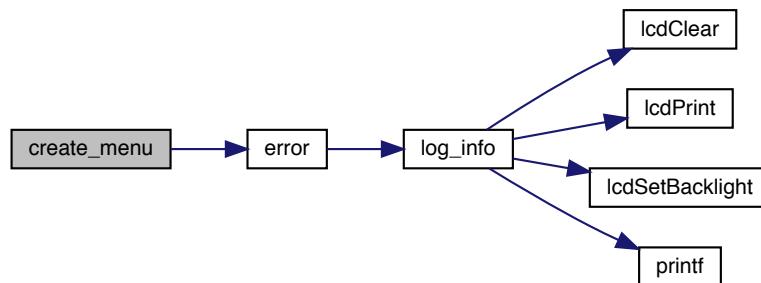
References **menu_t::current**, **error()**, **menu_t::max**, **menu_t::max_f**, **menu_t::min**, **menu_t::min_f**, **menu_t::prompt**, **menu_t::step**, **menu_t::step_f**, and **menu_t::type**.

Referenced by **init_menu_float()**, **init_menu_int()**, and **init_menu_var()**.

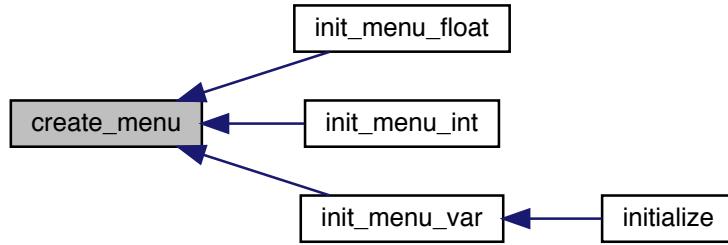
```

00027
00028     menu_t *menu = (menu_t *)malloc(sizeof(menu_t));
00029     if (!menu) {
00030         error("Menu Malloc");
00031     }
00032     menu->type = type;
00033     // Add one for null terminator
00034     size_t strlength = strlen(prompt) + 1;
00035     menu->prompt = (char *)malloc(strlength * sizeof(char));
00036     memcpy(menu->prompt, prompt, strlength);
00037     menu->max = INT_MAX;
00038     menu->min = INT_MIN;
00039     menu->step = 1;
00040     menu->min_f = FLT_MIN;
00041     menu->max_f = FLT_MAX;
00042     menu->step_f = 1;
00043     menu->current = 0;
00044
00045     return menu;
00046 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.85.1.3 denint_menu()

```
void denint_menu (
    menu_t * menu )
```

Destroys a menu *Menu must be freed or will cause memory leak*

Parameters

<code>menu</code>	the menu to free
-------------------	------------------

See also

`menu`

Author

Chris Jerrett

Date

9/8/17

Definition at line 203 of file `menu.c`.

References `menu_t::options`, and `menu_t::prompt`.

```
00203
00204     free(menu->prompt);
00205     if (menu->options != NULL)
00206         free(menu->options);
00207     free(menu);
00208 }
```

6.85.1.4 display_menu()

```
int display_menu (
    menu_t * menu )
```

Displays a menu context.

Menu must be freed or will cause memory leak! Will exit if robot is enabled. This prevents menu from locking up system in even of a reset.

Parameters

menu	the menu to display
------	---------------------

See also

[menu_type](#) (p. 132)

Author

Chris Jerrett

Date

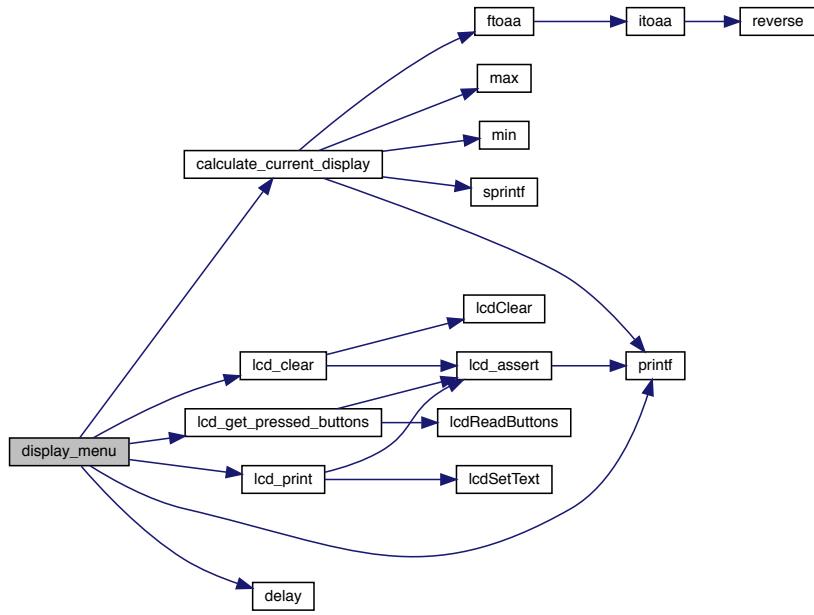
9/8/17

Definition at line 164 of file `menu.c`.

References `calculate_current_display()`, `menu_t::current`, `delay()`, `lcd_clear()`, `lcd_get_pressed_buttons()`, `lcd_print()`, `PRESSED`, `printf()`, `menu_t::prompt`, and `RELEASED`.

```
00164 {
00165     lcd_print(TOP_ROW, menu->prompt);
00166     printf("printed prompt\n");
00167     // Will exit if teleop or autonomous begin. This is extremely important if
00168     // robot disconnects or resets.
00169     char val[16];
00170     while (lcd_get_pressed_buttons().middle == RELEASED) {
00171         calculate_current_display(val, menu);
00172
00173         if (lcd_get_pressed_buttons().right == PRESSED) {
00174             menu->current += 1;
00175         }
00176         if (lcd_get_pressed_buttons().left == PRESSED) {
00177             menu->current -= 1;
00178         }
00179         printf("%s\n", val);
00180         printf("%d\n", menu->current);
00181         lcd_print(2, val);
00182         delay(300);
00183     }
00184     printf("%d\n", menu->current);
00185     printf("return\n");
00186     lcd_clear();
00187     lcd_print(1, "Thk Cm Agn");
00188     lcd_print(2, val);
00189     delay(800);
00190     lcd_clear();
00191     return menu->current;
00192 }
```

Here is the call graph for this function:



6.85.1.5 init_menu_float()

```
menu_t* init_menu_float (
```

enum	menu_type	<i>type</i> ,
float	<i>min</i> ,	
float	<i>max</i> ,	
float	<i>step</i> ,	
const	<i>char</i> *	<i>prompt</i>)

Creates a menu context, but does not display.

Menu must be freed or will cause memory leak!

Parameters

type *the type of menu*

See also

menu_type (p. 132)

Parameters

<code>min</code>	<i>the minimum value</i>
<code>max</code>	<i>the maximum value</i>
<code>step</code>	<i>the step value</i>
<code>prompt</code>	<i>the prompt to display to user</i>

Author

Chris Jerrett

Date

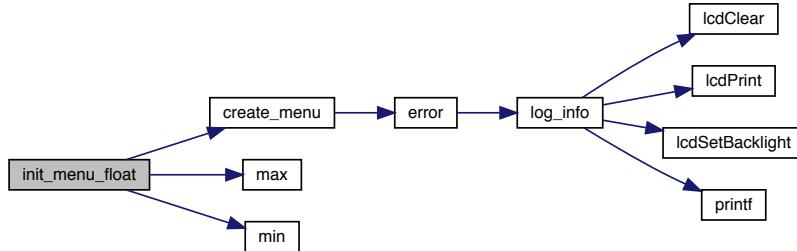
9/8/17

Definition at line 111 of file **menu.c**.

References `create_menu()`, `max()`, `menu_t::max_f`, `min()`, `menu_t::min_f`, and `menu_t::step_f`.

```
00112
00113     menu_t *menu = create_menu(type, prompt);
00114     menu->min_f = min;
00115     menu->max_f = max;
00116     menu->step_f = step;
00117     return menu;
00118 }
```

Here is the call graph for this function:

**6.85.1.6 init_menu_int()**

```
menu_t* init_menu_int (
    enum menu_type type,
    int min,
    int max,
    int step,
    const char * prompt )
```

Creates a menu context, but does not display.

Menu must be freed or will cause memory leak

Parameters

<code>type</code>	<i>the type of menu</i>
-------------------	-------------------------

See also

[menu_type](#) (p. 132)

Parameters

<code>min</code>	<i>the minimum value</i>
<code>max</code>	<i>the maximum value</i>
<code>step</code>	<i>the step value</i>
<code>prompt</code>	<i>the prompt to display to user</i>

Author

Chris Jerrett

Date

9/8/17

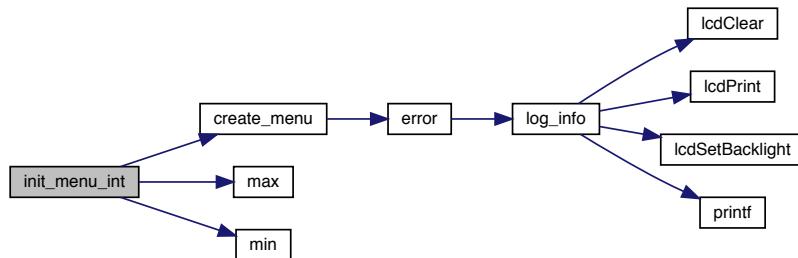
Definition at line 88 of file `menu.c`.

References `create_menu()`, `menu_t::current`, `max()`, `menu_t::max`, `min()`, `menu_t::min`, and `menu_t::step`.

```

00089
00090     menu_t *menu = create_menu(type, prompt);
00091     menu->min = min;
00092     menu->max = max;
00093     menu->step = step;
00094     menu->current = 0;
00095     return menu;
00096 }
```

Here is the call graph for this function:



6.85.1.7 init_menu_var()

```
menu_t* init_menu_var (
    enum menu_type type,
    const char * prompt,
    int nums,
    ...
)
```

Creates a menu context, but does not display.

Menu must be freed or will cause memory leak

Parameters

type	<i>the type of menu</i>
-------------	-------------------------

See also

[menu_type](#) (p. 132)

Parameters

nums	<i>the number of elements passed to function</i>
prompt	<i>the prompt to display to user</i>
options	<i>the options to display for user</i>

Author

Chris Jerrett

Date

9/8/17

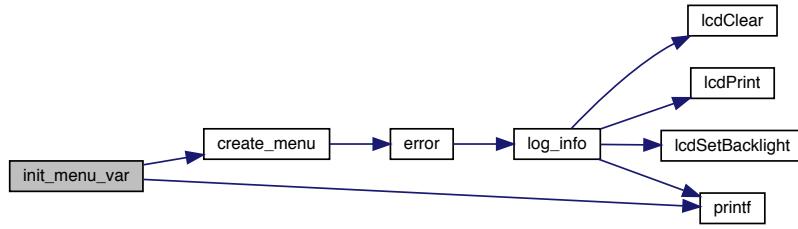
Definition at line **60** of file **menu.c**.

References [create_menu\(\)](#), [menu_t::length](#), [menu_t::options](#), and [printf\(\)](#).

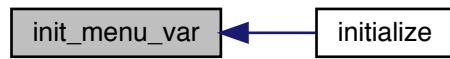
Referenced by [initialize\(\)](#).

```
00060
00061     menu_t *menu = create_menu(type, prompt);
00062     va_list ap;
00063     char **options_array = (char **)calloc(sizeof(char *), nums);
00064     va_start(ap, nums);
00065     for (int i = 0; i < nums; i++) {
00066         options_array[i] = (char *)va_arg(ap, char *);
00067         printf("%s\n", options_array[i]);
00068     }
00069     va_end(ap);
00070     menu->options = options_array;
00071     menu->length = nums;
00072     return menu;
00073 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.86 menu.c

```

00001 #include "menu.h"
00002
00009 static menu_t *create_menu(enum menu_type type, const char *prompt);
00010
00019 static void calculate_current_display(char *rtn, menu_t *menu);
00020
00027 static menu_t *create_menu(enum menu_type type, const char *prompt) {
00028     menu_t *menu = (menu_t *)malloc(sizeof(menu_t));
00029     if (!menu) {
00030         error("Menu Malloc");
00031     }
00032     menu->type = type;
00033     // Add one for null terminator
00034     size_t strlength = strlen(prompt) + 1;
00035     menu->prompt = (char *)malloc(strlength * sizeof(char));
00036     memcpy(menu->prompt, prompt, strlength);
00037     menu->max = INT_MAX;
00038     menu->min = INT_MIN;
00039     menu->step = 1;
00040     menu->min_f = FLT_MIN;
00041     menu->max_f = FLT_MAX;
00042     menu->step_f = 1;
00043     menu->current = 0;
00044
00045     return menu;
00046 }
00047
00060 menu_t *init_menu_var(enum menu_type type, const char *prompt, int nums, ...) {
00061     menu_t *menu = create_menu(type, prompt);
00062     va_list ap;
00063     char **options_array = (char **)calloc(sizeof(char *), nums);
00064     va_start(ap, nums);
00065     for (int i = 0; i < nums; i++) {
  
```

```

00066     options_array[i] = (char *)va_arg(ap, char *);
00067     printf("%s\n", options_array[i]);
00068 }
00069 va_end(ap);
00070 menu->options = options_array;
00071 menu->length = nums;
00072 return menu;
00073 }
00074
00075 menu_t *init_menu_int(enum menu_type type, int min, int max, int step,
00076                         const char *prompt) {
00077     menu_t *menu = create_menu(type, prompt);
00078     menu->min = min;
00079     menu->max = max;
00080     menu->step = step;
00081     menu->current = 0;
00082     return menu;
00083 }
00084
00085 menu_t *init_menu_float(enum menu_type type, float min, float max, float step,
00086                           const char *prompt) {
00087     menu_t *menu = create_menu(type, prompt);
00088     menu->min_f = min;
00089     menu->max_f = max;
00090     menu->step_f = step;
00091     return menu;
00092 }
00093
00094 static void calculate_current_display(char *rtn, menu_t *menu) {
00095     if (menu->type == STRING_TYPE) {
00096         int index = menu->current % menu->length;
00097         sprintf(rtn, "%s", menu->options[index]);
00098         printf("\n", rtn);
00099         return;
00100     }
00101     if (menu->type == INT_TYPE) {
00102         int step = (menu->step);
00103         int min = (menu->min);
00104         int max = (menu->max);
00105         int value = menu->current * step;
00106         if (value < min) {
00107             value = min;
00108             menu->current++;
00109         }
00110         if (value > max) {
00111             value = max;
00112             menu->current--;
00113         }
00114         sprintf(rtn, "%d", value);
00115     }
00116     if (menu->type == FLOAT_TYPE) {
00117         float step = (menu->step_f);
00118         float min = (menu->min_f);
00119         float max = (menu->max_f);
00120         float value = menu->current * step;
00121         value = value < min ? min : value;
00122         value = value > max ? max : value;
00123         ftoa(value, rtn, 5);
00124     }
00125 }
00126
00127 int display_menu(menu_t *menu) {
00128     lcd_print(TOP_ROW, menu->prompt);
00129     printf("printed prompt\n");
00130     // Will exit if teleop or autonomous begin. This is extremely important if
00131     // robot disconnects or resets.
00132     char val[16];
00133     while (lcd_get_pressed_buttons().middle == RELEASED) {
00134         calculate_current_display(val, menu);
00135
00136         if (lcd_get_pressed_buttons().right == PRESSED) {
00137             menu->current += 1;
00138         }
00139         if (lcd_get_pressed_buttons().left == PRESSED) {
00140             menu->current -= 1;
00141         }
00142         printf("\n", val);
00143         printf("\n", menu->current);
00144         lcd_print(2, val);
00145         delay(300);
00146
00147     }
00148 }
```

```

00183     }
00184     printf("%d\n", menu->current);
00185     printf("return\n");
00186     lcd_clear();
00187     lcd_print(1, "Thk Cm Agn");
00188     lcd_print(2, val);
00189     delay(800);
00190     lcd_clear();
00191     return menu->current;
00192 }
00193
00203 void denint_menu(menu_t *menu) {
00204     free(menu->prompt);
00205     if (menu->options != NULL)
00206         free(menu->options);
00207     free(menu);
00208 }
```

6.87 src/mobile_goal_intake.c File Reference

Functions

- **void lower_intake ()**
lowers the intake
- **void raise_intake ()**
raises the intake
- **void set_intake_motor (int n)**
sets the intake motor
- **void update_intake ()**
updates the mobile goal intake in teleop.

6.87.1 Function Documentation

6.87.1.1 lower_intake()

void lower_intake ()

lowers the intake

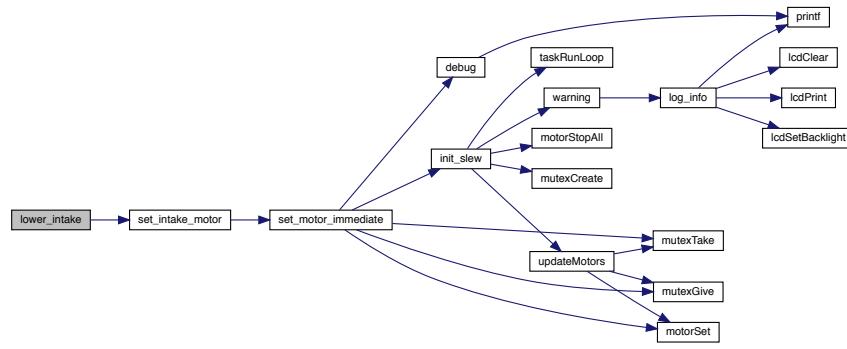
Definition at line 5 of file **mobile_goal_intake.c**.

References **set_intake_motor()**.

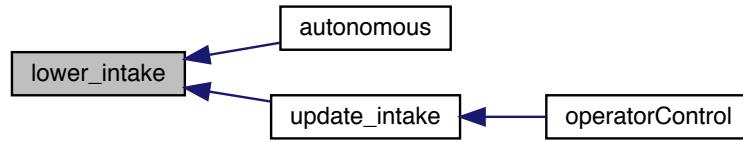
Referenced by **autonomous()**, and **update_intake()**.

00005 { set_intake_motor(-100); }

Here is the call graph for this function:



Here is the caller graph for this function:



6.87.1.2 raise_intake()

```
void raise_intake( )
```

raises the intake

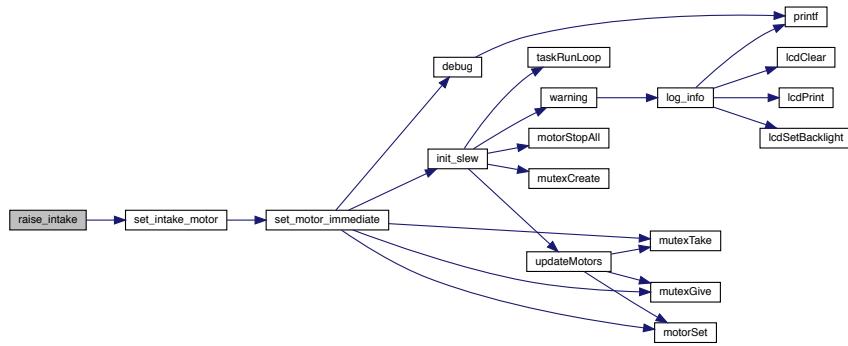
Definition at line 7 of file **mobile_goal_intake.c**.

References **set_intake_motor()**.

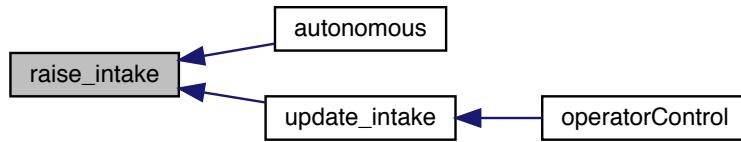
Referenced by **autonomous()**, and **update_intake()**.

```
00007 { set_intake_motor(100); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.87.1.3 set_intake_motor()

```
void set_intake_motor (
    int n )
```

sets the intake motor

Author

Chris Jerrett

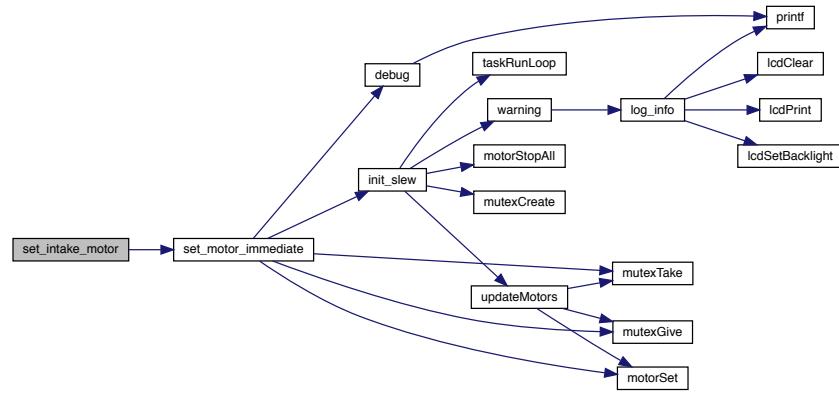
Definition at line 3 of file **mobile_goal_intake.c**.

References **set_motor_immediate()**.

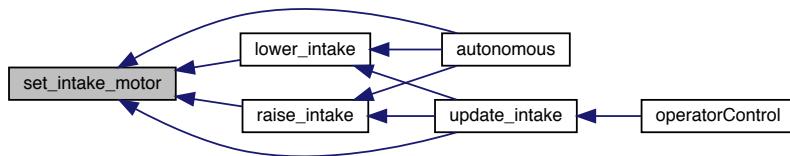
Referenced by **autonomous()**, **lower_intake()**, **raise_intake()**, and **update_intake()**.

```
00003 { set_motor_immediate(MOBILE_INTAKE_MOTOR, n); }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.87.1.4 update_intake()

```
void update_intake( )
```

updates the mobile goal intake in teleop.

Author

Chris Jerrett

Definition at line 12 of file **mobile_goal_intake.c**.

References `joystickGetDigital()`, `lower_intake()`, `raise_intake()`, and `set_intake_motor()`.

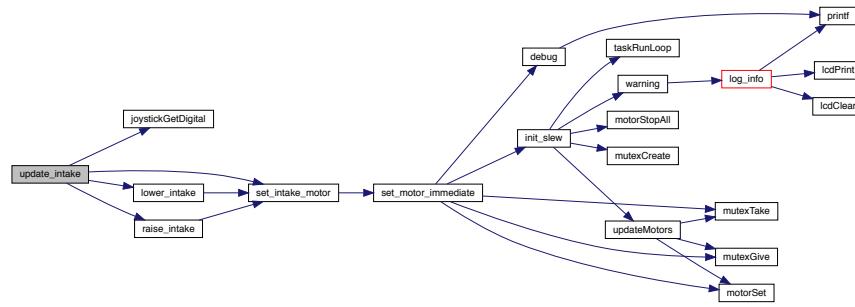
Referenced by `operatorControl()`.

```

00012     {
00013     if (joystickGetDigital(MASTER, 8, JOY_UP)) {
00014         raise_intake();
00015     } else if (joystickGetDigital(MASTER, 8, JOY_DOWN)) {
00016         lower_intake();
00017     } else
00018         set_intake_motor(0);
00019 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.88 mobile_goal_intake.c

```

00001 #include "mobile_goal_intake.h"
00002
00003 void set_intake_motor(int n) { set_motor_immediate(MOBILE_INTAKE_MOTOR, n); }
00004
00005 void lower_intake() { set_intake_motor(-100); }
00006
00007 void raise_intake() { set_intake_motor(100); }
00008
00012 void update_intake() {
00013     if (joystickGetDigital(MASTER, 8, JOY_UP)) {
00014         raise_intake();
00015     } else if (joystickGetDigital(MASTER, 8, JOY_DOWN)) {
00016         lower_intake();
00017     } else
00018         set_intake_motor(0);
00019 }

```

6.89 src/opcontrol.c File Reference

File for operator control code.

Functions

- **void operatorControl ()**

Runs the user operator control code.

6.89.1 Detailed Description

File for operator control code.

This file should contain the user **operatorControl()** (p. 317) function and any functions related to it.

Any copyright is dedicated to the Public Domain. <http://creativecommons.org/publicdomain/zero/1.0/>

PROS contains FreeRTOS (<http://www.freertos.org>) whose source code may be obtained from <http://sourceforge.net/projects/freertos/files/> or on request.

Definition in file **opcontrol.c**.

6.89.2 Function Documentation

6.89.2.1 operatorControl()

```
void operatorControl ( )
```

Runs the user operator control code.

This function will be started in its own task with the default priority and stack size whenever the robot is enabled via the Field Management System or the VEX Competition Switch in the operator control mode. If the robot is disabled or communications is lost, the operator control task will be stopped by the kernel. Re-enabling the robot will restart the task, not resume it from where it left off.

If no VEX Competition Switch or Field Management system is plugged in, the VEX Cortex will run the operator control task. Be warned that this will also occur if the VEX Cortex is tethered directly to a computer via the USB A to A cable without any VEX Joystick attached.

Code running in this task can take almost any action, as the VEX Joystick is available and the scheduler is operational. However, proper use of **delay()** (p. ??) or **taskDelayUntil()** (p. ??) is highly recommended to give other tasks (including system tasks such as updating LCDs) time to run.

This task should never exit; its should end with some kind of infinite loop, even if empty.

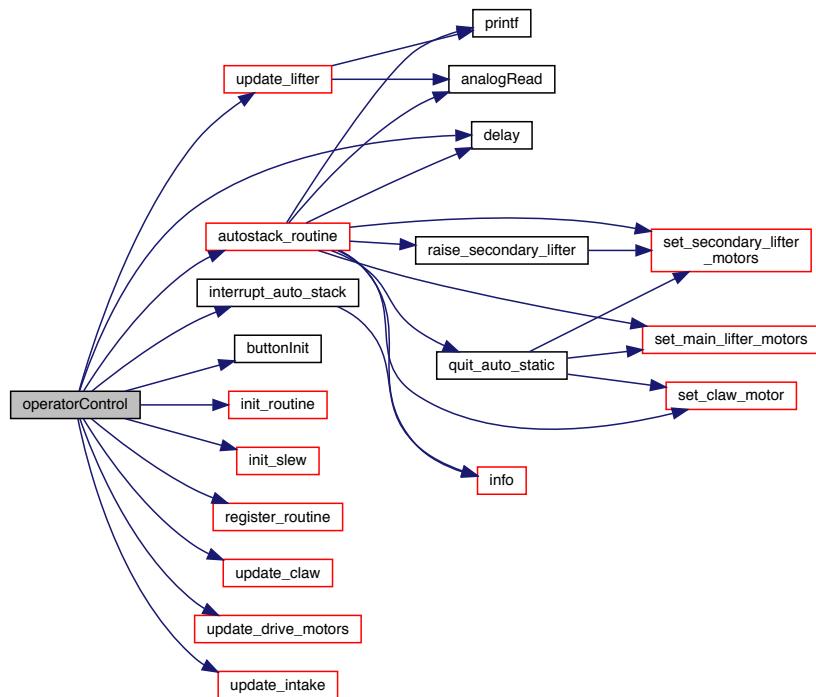
Definition at line **48** of file **opcontrol.c**.

References **autostack_routine()**, **buttonInit()**, **delay()**, **init_routine()**, **init_slew()**, **interrupt_auto_stack()**, **JOY2_7D**, **JOY2_7R**, **register_routine()**, **update_claw()**, **update_drive_motors()**, **update_intake()**, and **update_lifter()**.

```

00048
00049     buttonInit();
00050     init_routine();
00051     init_slew();
00052     register_routine(&autostack_routine, JOY2_7D, NULL);
00053     register_routine(&interrupt_auto_stack, JOY2_7R, NULL);
00054     while (1) {
00055         update_claw();
00056         update_intake();
00057         update_lifter();
00058         update_drive_motors();
00059         delay(20);
00060     }
00061 }
```

Here is the call graph for this function:



6.90 opcontrol.c

```

00001
00014 #include "claw.h"
00015 #include "drive.h"
00016 #include "lifter.h"
00017 #include "localization.h"
00018 #include "log.h"
00019 #include "main.h"
00020 #include "mobile_goal_intake.h"
00021 #include "routines.h"
00022 #include "slew.h"
00023 #include "toggle.h"
00024 #include "vmath.h"
00025
00048 void operatorControl() {
```

```
00049     buttonInit();
00050     init_routine();
00051     init_slew();
00052     register_routine(&autostack_routine, JOY2_7D, NULL);
00053     register_routine(&interrupt_auto_stack, JOY2_7R, NULL);
00054     while (1) {
00055         update_claw();
00056         update_intake();
00057         update_lifter();
00058         update_drive_motors();
00059         delay(20);
00060     }
00061 }
```

6.91 src/routines.c File Reference

Functions

- void **deinit_routines** ()
Stops the routine system.
- void **init_routine** ()
Starts the routine system.
- void **register_routine** (void(*routine)(void *), **button_t** on_buttons, **button_t** *prohibited_buttons)
Registers a routine for the system to use.
- void **routine_task** ()
Task that manages routines.

Variables

- static **list_t** * **routine_list**
- static **TaskHandle** **routine_task_var**

6.91.1 Function Documentation

6.91.1.1 **deinit_routines()**

```
void deinit_routines ( )
```

Stops the routine system.

Author

Chris Jerrett

Definition at line **47** of file **routines.c**.

References **list_destroy()**.

```
00047 { list_destroy(routine_list); }
```

Here is the call graph for this function:



6.91.1.2 init_routine()

```
void init_routine ( )
```

Starts the routine system.

Author

Chris Jerrett

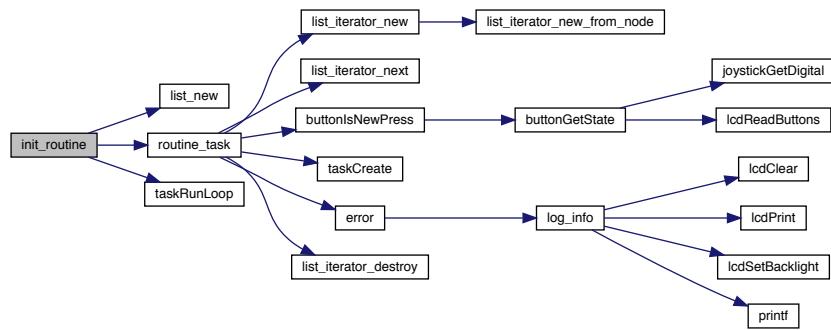
Definition at line **38** of file **routines.c**.

References **list_new()**, **routine_task()**, **routine_task_var**, and **taskRunLoop()**.

Referenced by **operatorControl()**.

```
00038 {
00039     routine_list = list_new();
00040     routine_task_var = taskRunLoop(routine_task, 20);
00041 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.91.1.3 register_routine()

```
void register_routine (
    void(*)(void *) routine,
    button_t on_buttons,
    button_t * prohibited_buttons )
```

Registers a routine for the system to use.

Parameters

<i>routine</i>	The routine to register
<i>on_buttons</i>	the trigger button
<i>prohibited_buttons</i>	the buttons it blocks

Todo

Author

Chris Jerrett

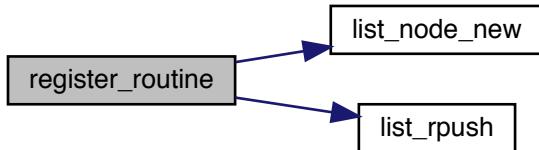
Definition at line 56 of file **routines.c**.

References **routine_t::blocked_buttons**, **list_node_new()**, **list_rpush()**, **routine_t::on_button**, **routine_t::routine**, and **list_node::val**.

Referenced by **operatorControl()**.

```
00057     {
00058     struct routine_t *r = (struct routine_t *)malloc(sizeof(routine_t));
00059     r->blocked_buttons = prohibited_buttons;
00060     r->routine = routine;
00061     r->on_button = on_buttons;
00062     list_node_t *node = list_node_new(r);
00063     node->val = r;
00064     list_rpush(routine_list, node);
00065 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.91.1.4 routine_task()

```
void routine_task ( )
```

Task that manages routines.

Author

Chris Jerrett

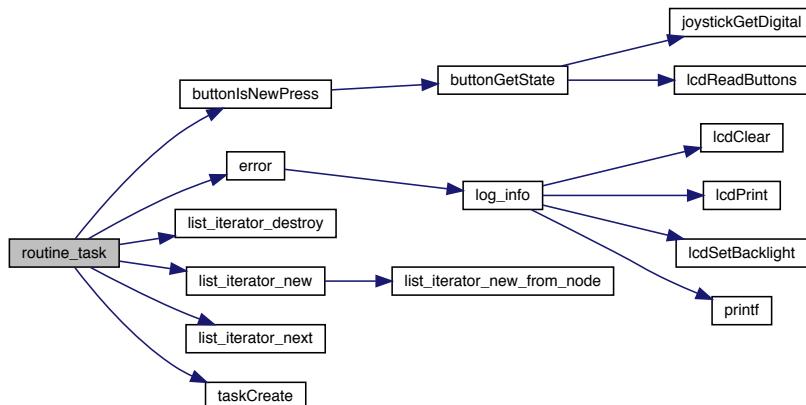
Definition at line 14 of file **routines.c**.

References **buttonIsNewPress()**, **error()**, **LIST_HEAD**, **list_iterator_destroy()**, **list_iterator_new()**, **list_iterator_next()**, **routine_t::on_button**, **routine_t::routine**, **taskCreate()**, and **list_node::val**.

Referenced by **init_routine()**.

```
00014     {
00015     list_node_t *node;
00016     list_iterator_t *it = list_iterator_new(routine_list, LIST_HEAD);
00017     if (it != NULL) {
00018         while (node = list_iterator_next(it)) {
00019             if (node->val != NULL) {
00020                 routine_t *routine = (routine_t *) (node->val);
00021                 if (buttonIsNewPress(routine->on_button)) {
00022                     TaskHandle task =
00023                         taskCreate(routine->routine, TASK_DEFAULT_STACK_SIZE, NULL,
00024                                     TASK_PRIORITY_DEFAULT);
00025                 }
00026             }
00027         }
00028     } else {
00029         error("List iterator was null");
00030     }
00031     list_iterator_destroy(it);
00032 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.91.2 Variable Documentation

6.91.2.1 routine_list

```
list_t* routine_list [static]
```

Definition at line **7** of file **routines.c**.

6.91.2.2 routine_task_var

```
TaskHandle routine_task_var [static]
```

Definition at line **9** of file **routines.c**.

Referenced by **init_routine()**.

6.92 routines.c

```

00001 #include "routines.h"
00002 #include "controller.h"
00003 #include "list.h"
00004 #include "log.h"
00005 #include "toggle.h"
00006
00007 static list_t *routine_list;
00008
00009 static TaskHandle routine_task_var;
00010
00011 void routine_task() {
00012     list_node_t *node;
00013     list_iterator_t *it = list_iterator_new(routine_list, LIST_HEAD);
00014     if (it != NULL) {
00015         while (node = list_iterator_next(it)) {
00016             if (node->val != NULL) {
00017                 routine_t *routine = (routine_t *) (node->val);
00018                 if (buttonIsNewPress(routine->on_button)) {
00019                     TaskHandle task =
00020                         taskCreate(routine->routine, TASK_DEFAULT_STACK_SIZE, NULL,
00021                                     TASK_PRIORITY_DEFAULT);
00022                     taskDelete(task);
00023                 }
00024             }
00025         }
00026     }
00027 }
```

```

00028     } else {
00029         error("List iterator was null");
00030     }
00031     list_iterator_destroy(it);
00032 }
00033
00038 void init_routine() {
00039     routine_list = list_new();
00040     routine_task_var = taskRunLoop(routine_task, 20);
00041 }
00042
00047 void deinit_routines() { list_destroy(routine_list); }
00048
00056 void register_routine(void (*routine)(void *), button_t on_buttons,
00057                         button_t *prohibited_buttons) {
00058     struct routine_t *r = (struct routine_t *)malloc(sizeof(routine_t));
00059     r->blocked_buttons = prohibited_buttons;
00060     r->routine = routine;
00061     r->on_button = on_buttons;
00062     list_node_t *node = list_node_new(r);
00063     node->val = r;
00064     list_rpush(routine_list, node);
00065 }

```

6.93 src/slew.c File Reference

Functions

- **void deinitSlew ()**
Deinitializes the slew rate controller and frees memory.
- **void init_slew ()**
Initializes the slew rate controller.
- **void set_motor_immediate (int motor, int speed)**
Sets the motor speed ignoring the slew controller.
- **void set_motor_slew (int motor, int speed)**
Sets motor speed wrapped inside the slew rate controller.
- **void updateMotors ()**
Closes the distance between the desired motor value and the current motor value by half for each motor.

Variables

- static bool **initialized** = false
- static int **motors_curr_speeds** [10]
- static int **motors_set_speeds** [10]
- static **TaskHandle** **slew** = NULL
- static **Mutex** **speeds_mutex**

6.93.1 Function Documentation

6.93.1.1 deinitslew()

```
void deinitslew ( )
```

Deinitializes the slew rate controller and frees memory.

Author

Chris Jerrett

Date

9/14/17

Definition at line **59** of file **slew.c**.

References **initialized**, **motors_curr_speeds**, **motors_set_speeds**, **slew**, and **taskDelete()**.

Referenced by **autonomous()**.

```
00059     {
00060     taskDelete(slew);
00061     memset(motors_set_speeds, 0, sizeof(int) * 10);
00062     memset(motors_curr_speeds, 0, sizeof(int) * 10);
00063     initialized = false;
00064 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.93.1.2 init_slew()

```
void init_slew ( )
```

Initializes the slew rate controller.

Author

Chris Jerrett, Christian DeSimone

Date

9/14/17

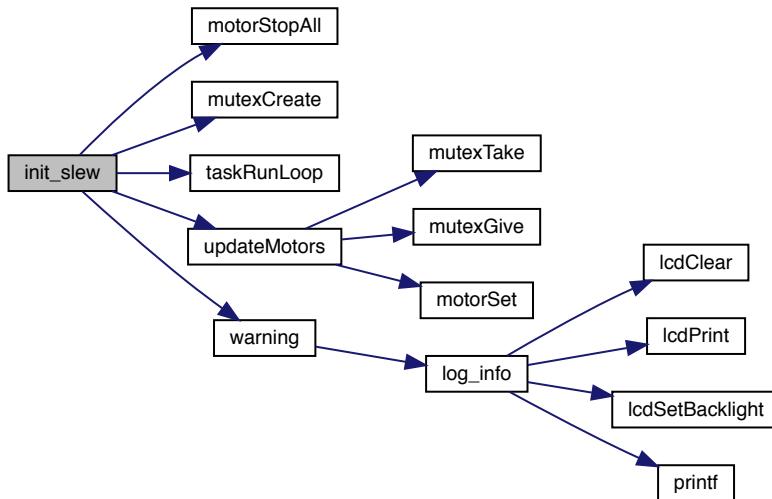
Definition at line 42 of file **slew.c**.

References **initialized**, **motors_curr_speeds**, **motors_set_speeds**, **motorStopAll()**, **mutexCreate()**, **slew**, **speeds_mutex**, **taskRunLoop()**, **updateMotors()**, and **warning()**.

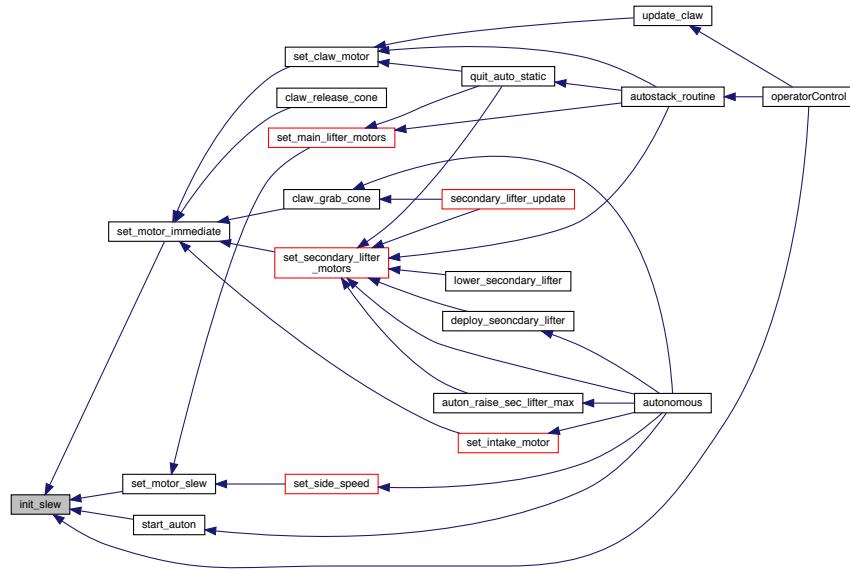
Referenced by **operatorControl()**, **set_motor_immediate()**, **set_motor_slew()**, and **start_auton()**.

```
00042             {
00043     if (initialized) {
00044         warning("Trying to init already init slew");
00045     }
00046     memset(motors_set_speeds, 0, sizeof(int) * 10);
00047     memset(motors_curr_speeds, 0, sizeof(int) * 10);
00048     motorStopAll();
00049     speeds_mutex = mutexCreate();
00050     slew = taskRunLoop(updateMotors, 100);
00051     initialized = true;
00052 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.93.1.3 set_motor_immediate()

```
void set_motor_immediate (
```

Sets the motor speed ignoring the slew controller.

Parameters

<i>motor</i>	the motor port to use
<i>speed</i>	the speed to use, between -127 and 127

Author

Chris Jerrett

Date

9/14/17

Definition at line **90** of file **slew.c**.

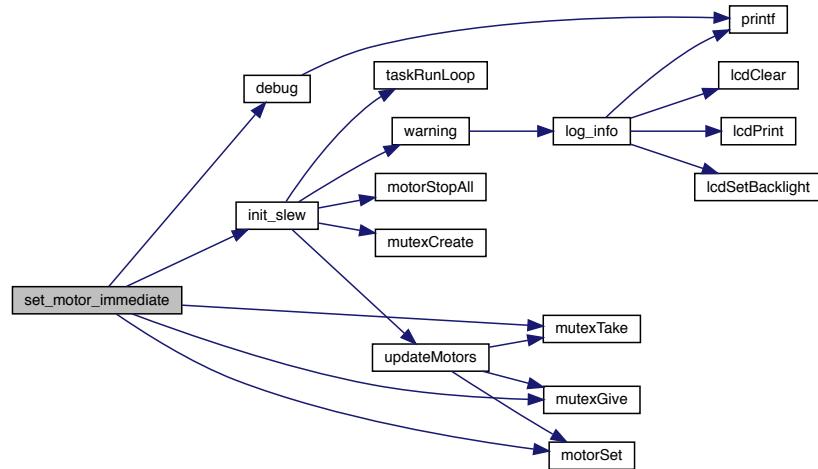
References `debug()`, `init_slew()`, `initialized`, `motors_curr_speeds`, `motors_set_speeds`, `motorSet()`, `mutexGive()`, `mutexTake()`, and `speeds_mutex`.

Referenced by `claw_grab_cone()`, `claw_release_cone()`, `set_claw_motor()`, `set_intake_motor()`, and `set_secondary_lifter_motors()`.

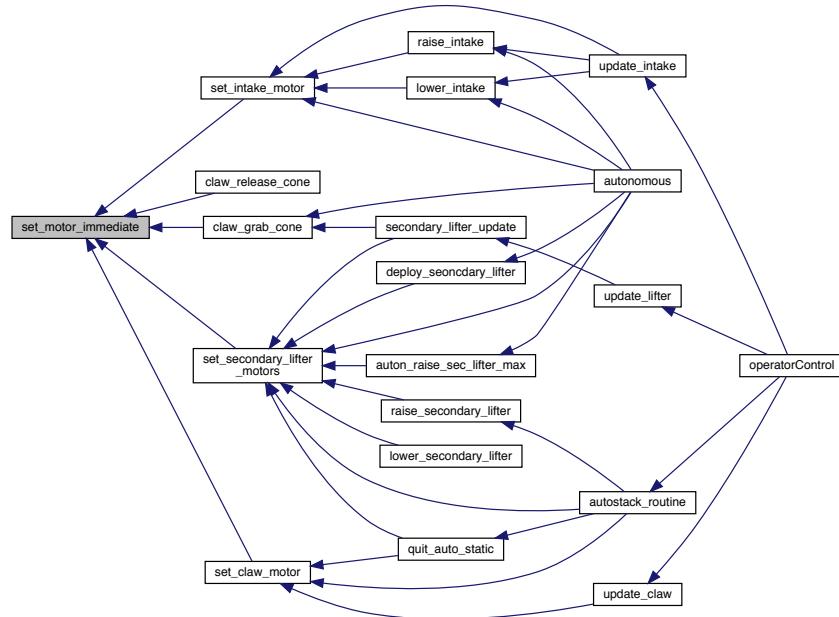
```

00090
00091     if (!initialized) {
00092         debug("Slew Not Initialized! Initializing");
00093         init_slew();
00094     }
00095     motorSet(motor, speed);
00096     mutexTake(speeds_mutex, 10);
00097     motors_curr_speeds[motor - 1] = speed;
00098     motors_set_speeds[motor - 1] = speed;
00099     mutexGive(speeds_mutex);
00100 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.93.1.4 set_motor_slew()

```
void set_motor_slew (
    int motor,
    int speed )
```

Sets motor speed wrapped inside the slew rate controller.

Parameters

<i>motor</i>	the motor port to use
<i>speed</i>	the speed to use, between -127 and 127

Author

Chris Jerrett

Date

9/14/17

Definition at line 73 of file **slew.c**.

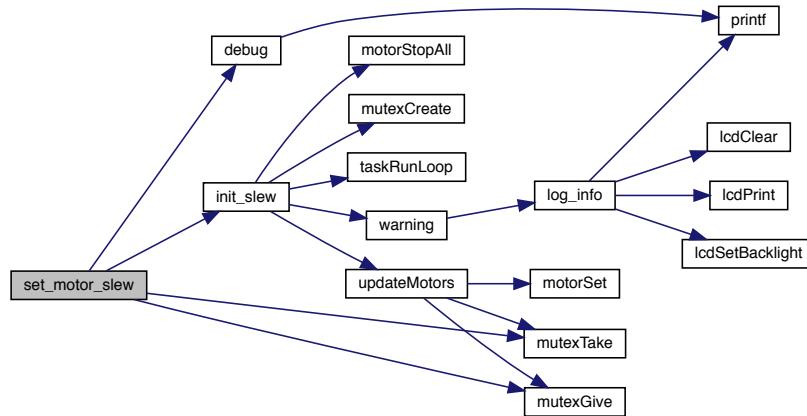
References **debug()**, **init_slew()**, **initialized**, **motors_set_speeds**, **mutexGive()**, **mutexTake()**, and **speeds_← mutex**.

Referenced by **set_main_lifter_motors()**, and **set_side_speed()**.

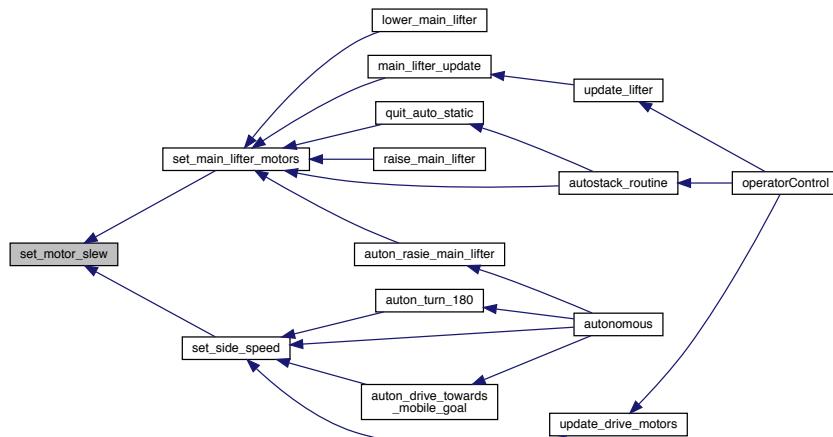
```

00073     {
00074     if (!initialized) {
00075         debug("Slew Not Initialized! Initializing");
00076         init_slew();
00077     }
00078     mutexTake(speeds_mutex, 10);
00079     motors_set_speeds[motor - 1] = speed;
00080     mutexGive(speeds_mutex);
00081 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.93.1.5 updateMotors()

```
void updateMotors ( )
```

Closes the distance between the desired motor value and the current motor value by half for each motor.

Author

Chris Jerrett

Date

9/14/17

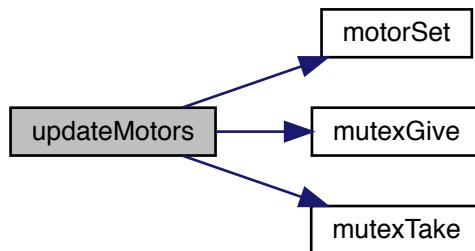
Definition at line **19** of file **slew.c**.

References **motors_curr_speeds**, **motors_set_speeds**, **motorSet()**, **mutexGive()**, **mutexTake()**, and **speeds_mutex**.

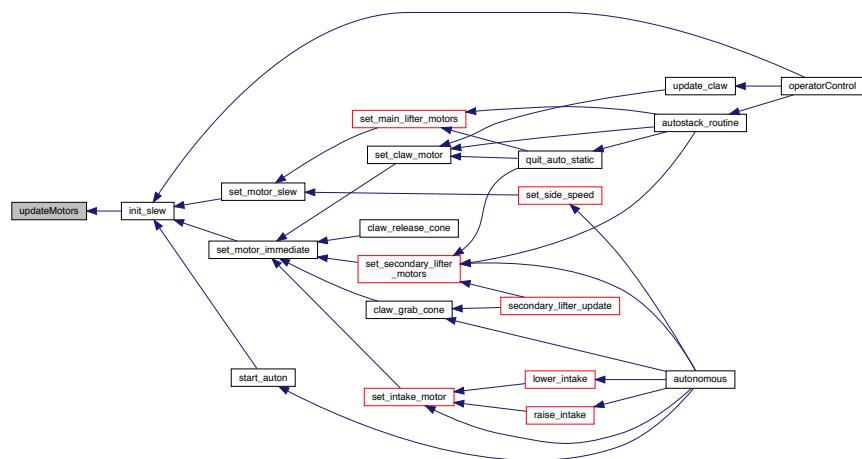
Referenced by **init_slew()**.

```
00019      {
00020      // Take back half approach
00021      // Not linear but equal to setSpeed(1-(1/2)^x)
00022      for (unsigned int i = 0; i < 9; i++) {
00023          if (motors_set_speeds[i] == motors_curr_speeds[i])
00024              continue;
00025          mutexTake(speeds_mutex, 10);
00026          int set_speed = (motors_set_speeds[i]);
00027          int curr_speed = motors_curr_speeds[i];
00028          mutexGive(speeds_mutex);
00029          int diff = set_speed - curr_speed;
00030          int offset = diff;
00031          int n = curr_speed + offset;
00032          motors_curr_speeds[i] = n;
00033          motorSet(i + 1, n);
00034      }
00035 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.93.2 Variable Documentation

6.93.2.1 initialized

```
bool initialized = false [static]
```

Definition at line 11 of file **slew.c**.

Referenced by **deinitSlew()**, **init_slew()**, **set_motor_immediate()**, and **set_motor_slew()**.

6.93.2.2 motors_curr_speeds

```
int motors_curr_speeds[10] [static]
```

Definition at line 7 of file **slew.c**.

Referenced by **deinitSlew()**, **init_slew()**, **set_motor_immediate()**, and **updateMotors()**.

6.93.2.3 motors_set_speeds

```
int motors_set_speeds[10] [static]
```

Definition at line 6 of file **slew.c**.

Referenced by **deinitSlew()**, **init_slew()**, **set_motor_immediate()**, **set_motor_slew()**, and **updateMotors()**.

6.93.2.4 slew

```
TaskHandle slew = NULL [static]
```

Definition at line 9 of file **slew.c**.

Referenced by **deinitSlew()**, and **init_slew()**.

6.93.2.5 speeds_mutex

```
Mutex speeds_mutex [static]
```

Definition at line 4 of file **slew.c**.

Referenced by **init_slew()**, **set_motor_immediate()**, **set_motor_slew()**, and **updateMotors()**.

6.94 slew.c

```
00001 #include "slew.h"
00002 #include "log.h"
00003
00004 static Mutex speeds_mutex;
00005
00006 static int motors_set_speeds[10];
00007 static int motors_curr_speeds[10];
00008
00009 static TaskHandle slew = NULL; // TaskHandle is of type void*
00010
00011 static bool initialized = false;
00012
00013 void updateMotors() {
00014     // Take back half approach
00015     // Not linear but equal to setSpeed(1-(1/2)^x)
00016     for (unsigned int i = 0; i < 9; i++) {
00017         if (motors_set_speeds[i] == motors_curr_speeds[i])
00018             continue;
00019         mutexTake(speeds_mutex, 10);
00020         int set_speed = (motors_set_speeds[i]);
00021         int curr_speed = motors_curr_speeds[i];
00022         mutexGive(speeds_mutex);
00023         int diff = set_speed - curr_speed;
00024         int offset = diff;
00025         int n = curr_speed + offset;
00026         motors_curr_speeds[i] = n;
00027         motorSet(i + 1, n);
00028     }
00029 }
00030
00031 void init_slew() {
00032     if (initialized) {
00033         warning("Trying to init already init slew");
00034     }
00035 }
00036
00037 void deinitSlew() {
00038     taskDelete(slew);
00039     memset(motors_set_speeds, 0, sizeof(int) * 10);
00040     memset(motors_curr_speeds, 0, sizeof(int) * 10);
00041     motorStopAll();
00042     speeds_mutex = mutexCreate();
00043     slew = taskRunLoop(updateMotors, 100);
00044     initialized = true;
00045 }
00046
00047 void taskRunLoop(void (*task)(void), int period) {
00048     while (1) {
00049         task();
00050         delay(period);
00051     }
00052 }
```

```

00064 }
00065
00073 void set_motor_slew(int motor, int speed) {
00074     if (!initialized) {
00075         debug("Slew Not Initialized! Initializing");
00076         init_slew();
00077     }
00078     mutexTake(speeds_mutex, 10);
00079     motors_set_speeds[motor - 1] = speed;
00080     mutexGive(speeds_mutex);
00081 }
00082
00090 void set_motor_immediate(int motor, int speed) {
00091     if (!initialized) {
00092         debug("Slew Not Initialized! Initializing");
00093         init_slew();
00094     }
00095     motorSet(motor, speed);
00096     mutexTake(speeds_mutex, 10);
00097     motors_curr_speeds[motor - 1] = speed;
00098     motors_set_speeds[motor - 1] = speed;
00099     mutexGive(speeds_mutex);
00100 }

```

6.95 src/toggle.c File Reference

Functions

- **bool buttonGetState (button_t button)**
Returns the current status of a button (pressed or not pressed)
- **void buttonInit ()**
Initializes the buttons array.
- **bool buttonIsNewPress (button_t button)**
Detects if button is a new press from most recent check by comparing previous value to current value.

Variables

- **bool buttonPressed [27]**
Represents the array of "wasPressed" for all 27 available buttons.

6.95.1 Function Documentation

6.95.1.1 buttonGetState()

```
bool buttonGetState (
    button_t   )
```

Returns the current status of a button (pressed or not pressed)

Parameters

button	The button to detect from the Buttons enumeration.
---------------	--

Returns

true (pressed) or false (not pressed)

Definition at line **27** of file **toggle.c**.

References **joystickGetDigital()**, **LCD_CENT**, **LCD_LEFT**, **LCD_RIGHT**, and **IcdReadButtons()**.

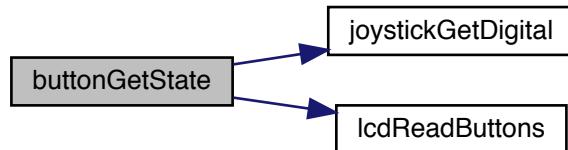
Referenced by **buttonIsNewPress()**.

```
00027         {
00028     bool currentButton = false;
00029
00030     // Determine how to get the current button value (from what function) and
00031     // where it is, then get it.
00032     if (button < LCD_LEFT) {
00033         // button is a joystick button
00034         unsigned char joystick;
00035         unsigned char buttonGroup;
00036         unsigned char buttonLocation;
00037
00038         button_t newButton;
00039         if (button <= 11) {
00040             // button is on joystick 1
00041             joystick = 1;
00042             newButton = button;
00043         } else {
00044             // button is on joystick 2
00045             joystick = 2;
00046             // shift button down to joystick 1 buttons in order to
00047             // detect which button on joystick is queried
00048             newButton = (button_t)(button - 12);
00049         }
00050
00051         switch (newButton) {
00052             case 0:
00053                 buttonGroup = 5;
00054                 buttonLocation = JOY_DOWN;
00055                 break;
00056             case 1:
00057                 buttonGroup = 5;
00058                 buttonLocation = JOY_UP;
00059                 break;
00060             case 2:
00061                 buttonGroup = 6;
00062                 buttonLocation = JOY_DOWN;
00063                 break;
00064             case 3:
00065                 buttonGroup = 6;
00066                 buttonLocation = JOY_UP;
00067                 break;
00068             case 4:
00069                 buttonGroup = 7;
00070                 buttonLocation = JOY_UP;
00071                 break;
00072             case 5:
00073                 buttonGroup = 7;
00074                 buttonLocation = JOY_LEFT;
00075                 break;
00076             case 6:
00077                 buttonGroup = 7;
00078                 buttonLocation = JOY_RIGHT;
00079                 break;
00080             case 7:
00081                 buttonGroup = 7;
00082                 buttonLocation = JOY_DOWN;
00083                 break;
00084             case 8:
00085                 buttonGroup = 8;
00086                 buttonLocation = JOY_UP;
00087                 break;
00088             case 9:
00089                 buttonGroup = 8;
00090                 buttonLocation = JOY_LEFT;
00091                 break;
00092             case 10:
```

```

00093     buttonGroup = 8;
00094     buttonLocation = JOY_RIGHT;
00095     break;
00096 case 11:
00097     buttonGroup = 8;
00098     buttonLocation = JOY_DOWN;
00099     break;
00100 default:
00101     break;
00102 }
00103 currentButton = joystickGetDigital(joystick, buttonGroup, buttonLocation);
00104 } else {
00105 // button is on LCD
00106 if (button == LCD_LEFT)
00107     currentButton = (lcdReadButtons(uart1) == LCD_BTN_LEFT);
00108
00109 if (button == LCD_CENT)
00110     currentButton = (lcdReadButtons(uart1) == LCD_BTN_CENTER);
00111
00112 if (button == LCD_RIGHT)
00113     currentButton = (lcdReadButtons(uart1) == LCD_BTN_RIGHT);
00114 }
00115 return currentButton;
00116 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.95.1.2 buttonInit()

```
void buttonInit ( )
```

Initializes the buttons array.

Initializes the buttons.

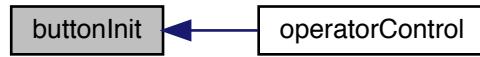
Definition at line 22 of file **toggle.c**.

References **buttonPressed**.

Referenced by **operatorControl()**.

```
00022     {
00023     for (int i = 0; i < 27; i++)
00024         buttonPressed[i] = false;
00025 }
```

Here is the caller graph for this function:



6.95.1.3 buttonIsNewPress()

```
bool buttonIsNewPress (
    button_t button )
```

Detects if button is a new press from most recent check by comparing previous value to current value.

Parameters

<i>button</i>	The button to detect from the Buttons enumeration (see include/buttons.h).
---------------	--

Returns

true or false depending on if there was a change in button state.

Example code:

```
...
if(buttonIsNewPress(JOY1_8D))
    digitalWrite(1, !digitalRead(1));
...
```

Definition at line **136** of file **toggle.c**.

References **buttonGetState()**, and **buttonPressed**.

Referenced by **routine_task()**.

```
00136     {
00137     bool currentButton = buttonGetState(button);
00138
00139     if (!currentButton) // buttons is not currently pressed
00140         buttonPressed[button] = false;
00141
00142     if (currentButton && !buttonPressed[button]) {
00143         // button is currently pressed and was not detected as being pressed during
00144         // last check
00145         buttonPressed[button] = true;
00146         return true;
00147     } else
00148         return false; // button is not pressed or was already detected
00149 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.95.2 Variable Documentation

6.95.2.1 buttonPressed

```
bool buttonPressed[27]
```

Represents the array of "wasPressed" for all 27 available buttons.

Definition at line 17 of file **toggle.c**.

Referenced by **buttonInit()**, and **buttonIsNewPress()**.

6.96 toggle.c

```
00001
00012 #include "toggle.h"
00013
00017 bool buttonPressed[27];
00018
00022 void buttonInit() {
00023     for (int i = 0; i < 27; i++)
00024         buttonPressed[i] = false;
00025 }
00026
00027 bool buttonGetState(button_t button) {
00028     bool currentButton = false;
00029
00030     // Determine how to get the current button value (from what function) and
00031     // where it is, then get it.
00032     if (button < LCD_LEFT) {
00033         // button is a joystick button
00034         unsigned char joystick;
00035         unsigned char buttonGroup;
00036         unsigned char buttonLocation;
00037
00038         button_t newButton;
00039         if (button <= 11) {
00040             // button is on joystick 1
00041             joystick = 1;
00042             newButton = button;
00043         } else {
00044             // button is on joystick 2
00045             joystick = 2;
00046             // shift button down to joystick 1 buttons in order to
00047             // detect which button on joystick is queried
00048             newButton = (button_t)(button - 12);
00049         }
00050
00051         switch (newButton) {
00052             case 0:
00053                 buttonGroup = 5;
00054                 buttonLocation = JOY_DOWN;
00055                 break;
00056             case 1:
00057                 buttonGroup = 5;
00058                 buttonLocation = JOY_UP;
00059                 break;
00060             case 2:
00061                 buttonGroup = 6;
00062                 buttonLocation = JOY_DOWN;
00063                 break;
00064             case 3:
00065                 buttonGroup = 6;
00066                 buttonLocation = JOY_UP;
00067                 break;
00068             case 4:
00069                 buttonGroup = 7;
00070                 buttonLocation = JOY_UP;
00071                 break;
00072             case 5:
00073                 buttonGroup = 7;
00074                 buttonLocation = JOY_LEFT;
00075                 break;
00076             case 6:
00077                 buttonGroup = 7;
00078                 buttonLocation = JOY_RIGHT;
00079                 break;
00080             case 7:
00081                 buttonGroup = 7;
00082                 buttonLocation = JOY_DOWN;
00083                 break;
00084             case 8:
00085                 buttonGroup = 8;
00086                 buttonLocation = JOY_UP;
00087                 break;
00088             case 9:
00089                 buttonGroup = 8;
00090                 buttonLocation = JOY_LEFT;
00091                 break;
00092             case 10:
00093                 buttonGroup = 8;
00094                 buttonLocation = JOY_RIGHT;
```

```

00095     break;
00096 case 11:
00097     buttonGroup = 8;
00098     buttonLocation = JOY_DOWN;
00099     break;
00100 default:
00101     break;
00102 }
00103 currentButton = joystickGetDigital(joystick, buttonGroup, buttonLocation);
00104 } else {
00105 // button is on LCD
00106 if (button == LCD_LEFT)
00107     currentButton = (lcdReadButtons(uart1) == LCD_BTN_LEFT);
00108
00109 if (button == LCD_CENT)
00110     currentButton = (lcdReadButtons(uart1) == LCD_BTN_CENTER);
00111
00112 if (button == LCD_RIGHT)
00113     currentButton = (lcdReadButtons(uart1) == LCD_BTN_RIGHT);
00114 }
00115 return currentButton;
00116 }
00117
00136 bool buttonIsNewPress(button_t button) {
00137     bool currentButton = buttonGetState(button);
00138
00139     if (!currentButton) // buttons is not currently pressed
00140         buttonPressed[button] = false;
00141
00142     if (currentButton && !buttonPressed[button]) {
00143 // button is currently pressed and was not detected as being pressed during
00144 // last check
00145         buttonPressed[button] = true;
00146         return true;
00147     } else
00148         return false; // button is not pressed or was already detected
00149 }
```

6.97 src/vlib.c File Reference

Functions

- void **ftoaa** (float a, char *buffer, int precision)
converts a float to string.
- int **itoaa** (int a, char *buffer, int digits)
converts a int to string.
- void **reverse** (char *str, int len)
reverses a string 'str' of length 'len'

6.97.1 Function Documentation

6.97.1.1 ftoaa()

```
void ftoaa (
    float a,
    char * buffer,
    int precision )
```

converts a float to string.

Parameters

<i>a</i>	the float
<i>buffer</i>	the string the float will be written to.
<i>precision</i>	digits after the decimal to write

Author

Christian DeSimone

Date

9/26/2017

Definition at line **55** of file **vlib.c**.

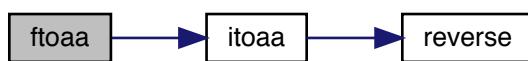
References **itoaa()**.

Referenced by **calculate_current_display()**.

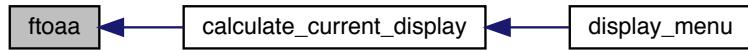
```

00055
00056
00057 // Extract integer part
00058 int ipart = (int)a;
00059
00060 // Extract floating part
00061 float fpart = a - (float)ipart;
00062
00063 // convert integer part to string
00064 int i = itoaa(ipart, buffer, 0);
00065
00066 // check for display option after point
00067 if (precision != 0) {
00068     buffer[i] = '.'; // add dot
00069
00070     // Get the value of fraction part up to given num.
00071     // of points after dot. The third parameter is needed
00072     // to handle cases like 233.007
00073     fpart = fpart * pow(10, precision);
00074
00075     itoaa((int)fpart, buffer + i + 1, precision);
00076 }
00077 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.97.1.2 itoaa()

```
int itoaa (
    int a,
    char * buffer,
    int digits )
```

converts a int to string.

Parameters

<i>a</i>	the integer
<i>buffer</i>	the string the int will be written to.
<i>digits</i>	the number of digits to be written

Returns

the digits

Author

Chris Jerrett, Christian DeSimone

Date

9/9/2017

Definition at line **30** of file **vlib.c**.

References **reverse()**.

Referenced by **ftoaa()**.

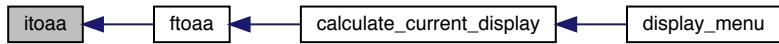
```

00030                                     {
00031     int i = 0;
00032     while (a) {
00033         buffer[i++] = (a % 10) + '0';
00034         a = a / 10;
00035     }
00036
00037     // If number of digits required is more, then
00038     // add 0s at the beginning
00039     while (i < digits)
00040         buffer[i++] = '0';
00041
00042     reverse(buffer, i);
00043     buffer[i] = '\0';
00044     return i;
00045 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.97.1.3 reverse()

```

void reverse (
    char * str,
    int len )
```

reverses a string 'str' of length 'len'

Author

Chris Jerrett

Date

9/9/2017

Parameters

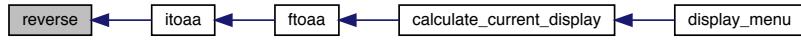
<i>str</i>	the string to reverse
<i>len</i>	the length

Definition at line **10** of file **vlib.c**.

Referenced by **itoaa()**.

```
00010
00011     int i = 0, j = len - 1, temp;
00012     while (i < j) {
00013         temp = str[i];
00014         str[i] = str[j];
00015         str[j] = temp;
00016         i++;
00017         j--;
00018     }
00019 }
```

Here is the caller graph for this function:

**6.98 vlib.c**

```
00001 #include "vlib.h"
00002
00010 void reverse(char *str, int len) {
00011     int i = 0, j = len - 1, temp;
00012     while (i < j) {
00013         temp = str[i];
00014         str[i] = str[j];
00015         str[j] = temp;
00016         i++;
00017         j--;
00018     }
00019 }
00020
00030 int itoaa(int a, char *buffer, int digits) {
00031     int i = 0;
00032     while (a) {
00033         buffer[i++] = (a % 10) + '0';
00034         a = a / 10;
00035     }
00036
00037     // If number of digits required is more, then
00038     // add 0s at the beginning
00039     while (i < digits)
00040         buffer[i++] = '0';
00041
00042     reverse(buffer, i);
00043     buffer[i] = '\0';
00044     return i;
00045 }
00046
00055 void ftoaa(float a, char *buffer, int precision) {
00056 }
```

```

00057 // Extract integer part
00058 int ipart = (int)a;
00059
00060 // Extract floating part
00061 float fpart = a - (float)ipart;
00062
00063 // convert integer part to string
00064 int i = itoa(ipart, buffer, 0);
00065
00066 // check for display option after point
00067 if (precision != 0) {
00068     buffer[i] = '.'; // add dot
00069
00070     // Get the value of fraction part up to given num.
00071     // of points after dot. The third parameter is needed
00072     // to handle cases like 233.007
00073     fpart = fpart * pow(10, precision);
00074
00075     itoa((int)fpart, buffer + i + 1, precision);
00076 }
00077 }
```

6.99 src/vmath.c File Reference

Functions

- struct **polar_cord cartesian_cord_to_polar** (struct **cord** cords)

Function to convert x and y 2 dimensional cartesian cordinated to polar coordinates.
- struct **polar_cord cartesian_to_polar** (float x, float y)

Function to convert x and y 2 dimensional cartesian coordinated to polar coordinates.
- int **max** (int a, int b)

the min of two values
- int **min** (int a, int b)

the min of two values
- double **sind** (double angle)

sine of a angle in degrees

6.99.1 Function Documentation

6.99.1.1 cartesian_cord_to_polar()

```
struct polar_cord cartesian_cord_to_polar (
    struct cord cords )
```

Function to convert x and y 2 dimensional cartesian cordinated to polar coordinates.

Author

Christian Desimone

Date

9/8/2017

Parameters

<code>cords</code>	the cartesian cords
--------------------	---------------------

Returns

a struct containing the angle and magnitude.

See also

[polar_cord](#) (p. 23)

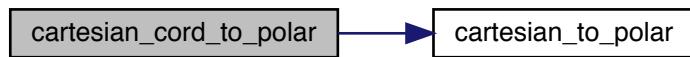
[cord](#) (p. 7)

Definition at line 53 of file **vmath.c**.

References [cartesian_to_polar\(\)](#).

```
00053 {  
00054     return cartesian_to_polar(cords.x, cords.y);  
00055 }
```

Here is the call graph for this function:



6.99.1.2 cartesian_to_polar()

```
struct polar_cord cartesian_to_polar (  
    float x,  
    float y )
```

Function to convert x and y 2 dimensional cartesian coordinates to polar coordinates.

Author

Christian Desimone

Date

9/8/2017

Parameters

x	float value of the x cartesian coordinate.
y	float value of the y cartesian coordinate.

Returns

a struct containing the angle and magnitude.

See also

polar_cord (p.23)

Definition at line 15 of file **vmath.c**.

References **polar_cord::angle**, and **polar_cord::magnitue**.

Referenced by **cartesian_cord_to_polar()**.

```

00015           {
00016     float degree = 0;
00017     double magnitude = sqrt((fabs(x) * fabs(x)) + (fabs(y) * fabs(y)));
00018
00019     if (x < 0) {
00020       degree += 180.0;
00021     } else if (x > 0 && y < 0) {
00022       degree += 360.0;
00023     }
00024
00025     if (x != 0 && y != 0) {
00026       degree += atan((float)y / (float)x);
00027     } else if (x == 0 && y > 0) {
00028       degree = 90.0;
00029     } else if (y == 0 && x < 0) {
00030       degree = 180.0;
00031     } else if (x == 0 && y < 0) {
00032       degree = 270.0;
00033     }
00034
00035     struct polar_cord p;
00036     p.angle = degree;
00037     p.magnitue = magnitude;
00038     return p;
00039   }

```

Here is the caller graph for this function:



6.99.1.3 max()

```
int max (
    int a,
    int b )
```

the min of two values

Parameters

<i>a</i>	the first
<i>b</i>	the second

Returns

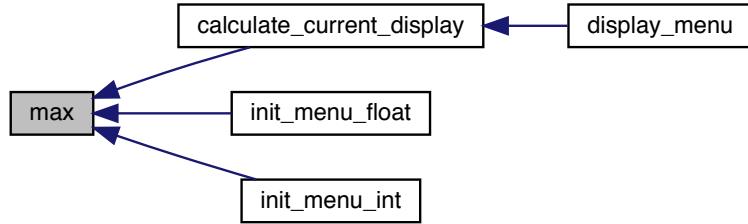
the smaller of *a* and *b*

Definition at line 83 of file **vmath.c**.

Referenced by **calculate_current_display()**, **init_menu_float()**, and **init_menu_int()**.

```
00083          {
00084     if (a > b)
00085         return a;
00086     return b;
00087 }
```

Here is the caller graph for this function:

**6.99.1.4 min()**

```
int min (
    int a,
    int b )
```

the min of two values

Parameters

<i>a</i>	the first
<i>b</i>	the second

Returns

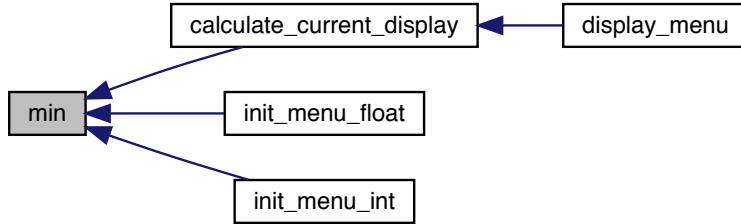
the smaller of a and b

Definition at line **71** of file **vmath.c**.

Referenced by **calculate_current_display()**, **init_menu_float()**, and **init_menu_int()**.

```
00071     if (a < b)           {  
00072         return a;  
00073     return b;  
00075 }
```

Here is the caller graph for this function:

**6.99.1.5 sind()**

```
double sind (  
    double angle )
```

sine of a angle in degrees

Definition at line **60** of file **vmath.c**.

```
00060             {  
00061     double angleradians = angle * M_PI / 180.0f;  
00062     return sin(angleradians);  
00063 }
```

6.100 vmath.c

```
00001 #include "vmath.h"
00002
00015 struct polar_cord cartesian_to_polar(float x, float y) {
00016     float degree = 0;
00017     double magnitude = sqrt((fabs(x) * fabs(x)) + (fabs(y) * fabs(y)));
00018
00019     if (x < 0) {
00020         degree += 180.0;
00021     } else if (x > 0 && y < 0) {
00022         degree += 360.0;
00023     }
00024
00025     if (x != 0 && y != 0) {
00026         degree += atan((float)y / (float)x);
00027     } else if (x == 0 && y > 0) {
00028         degree = 90.0;
00029     } else if (y == 0 && x < 0) {
00030         degree = 180.0;
00031     } else if (x == 0 && y < 0) {
00032         degree = 270.0;
00033     }
00034
00035     struct polar_cord p;
00036     p.angle = degree;
00037     p.magnitude = magnitude;
00038     return p;
00039 }
00040
00053 struct polar_cord cartesian_cord_to_polar(struct cord cords) {
00054     return cartesian_to_polar(cords.x, cords.y);
00055 }
00056
00060 double sind(double angle) {
00061     double angleradians = angle * M_PI / 180.0f;
00062     return sin(angleradians);
00063 }
00064
00071 int min(int a, int b) {
00072     if (a < b)
00073         return a;
00074     return b;
00075 }
00076
00083 int max(int a, int b) {
00084     if (a > b)
00085         return a;
00086     return b;
00087 }
```