2D FSAE simulator

Generated by Doxygen 1.9.1

Chapter 1

2D FSAE Simulation

1.1 Project Overview

This project is a 2D simulation of a Formula SAE (FSAE) vehicle, designed to model vehicle dynamics, perception, trajectory planning, and control systems. It provides a platform for testing and visualizing autonomous driving algorithms in a simplified environment.

1.2 Key Features

- **Vehicle Dynamics:** Simulates basic vehicle movement based on pedal and steering inputs.
- **LiDAR-based Perception:** Implements cone detection using simulated LiDAR measurements and a circle Hough transform.
- **Trajectory Planning:** Generates a driving trajectory by connecting detected cones.
- **Autonomous Control:** Includes an autonomous control system to follow the planned trajectory.
- **Manual Control:** Allows keyboard control of the vehicle.
- **Visualization:** Uses the Allegro library to provide a 2D graphical representation of the simulation, including the car, track, cones, and planned trajectory.
- **Periodic Tasks:** Utilizes a periodic task scheduler for managing different simulation components.

1.3 Project Structure

The project is organized into several modules:

- ∗∗src/main.c:∗∗ Initializes the simulation, creates tasks, and manages the main loop.
- ∗∗src/vehicle.c:∗∗ Implements the vehicle dynamics model.
- ∗∗src/perception.c:∗∗ Handles LiDAR data processing and cone detection.
- ∗∗src/trajectory.c:∗∗ Plans the driving trajectory based on detected cones.
- ∗∗src/control.c:∗∗ Implements control algorithms for both manual and autonomous driving.

- **src/display.c:** Manages the graphical display of the simulation.
- **src/tasks.c:** Defines the periodic tasks for perception, trajectory planning, control, and display.
- **src/utilities.c:** Provides utility functions for cone management and runtime measurement.
- **src/globals.c:** Defines global variables and constants used throughout the simulation.
- **src/ptask.c:** Implements the periodic task scheduler.
- **include/*.h:** Header files declaring the interfaces for each module.

1.4 Global Call Graph

1.5 Task Descriptions

- **Perception Task**: Acquires LiDAR measurements, detects cones, and updates the map.
- **Trajectory Task**: Plans the trajectory based on the detected cones.
- **Control Task**: Controls the vehicle either manually or autonomously.
- **Display Task**: Updates the graphical display of the simulation.

1.6 Building and Running the Simulation

1) Ensure you have Allegro 4 installed *(On Ubuntu/Debian:* `sudo apt-get install liballegro4.2 liballegro4.2-dev*)*.

2) Other dependencies for this project:

- *libyaml*: `git clone https://github.com/yaml/libyaml.git`

3) Use the provided Makefile to build the project.

4) Run the executable `make`.

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

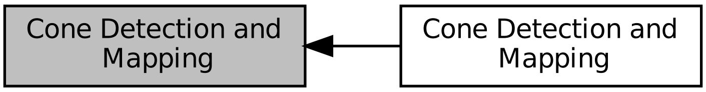
Here is a list of all files with brief descriptions:

Chapter 5

Module Documentation

5.1 Cone Detection and Mapping

Collaboration diagram for Cone Detection and Mapping:



Modules

- Cone Detection and Mapping

Files

- file perception.c

  Implements LiDAR-based cone detection and mapping using circle Hough transform.

Classes

- struct Hough_circle_point_t

  Represents a point in the Hough circle transformation space.

Functions

- void check_nearest_point (int angle, float new_point_x, float new_point_y, int color, cone_border *cone_↩ borders)
- void lidar (float car_x, float car_y, pointcloud_t *measures)
- void init_cone_borders (cone_border *cone_borders)
- void calculate_circle_points (float center_x, float center_y, int color, cone *circle_points)
- void find_closest_points (Hough_circle_point_t *circumference_points, float point_x, float point_↩ y, Hough_circle_point_t *reference_points, int ref_size)
- void find_local_minima (Hough_circle_point_t *points, int *first_min, int *second_min)
- float * find_cone_center (Hough_circle_point_t *possible_centers, int center_count)
- void mapping (float car_x, float car_y, int car_angle, cone *detected_cones)
- void update_map (cone *detected_cones)

Variables

- const int sliding_window = 360
- const int angle_step = 1
- int start_angle = 0
- const float ignore_distance = 0.5f
- const float distance_resolution = 0.01f
- cone detected_cones [MAX_DETECTED_CONES]
- int n_candidates = 0
- candidate_cone candidates [MAX_CANDIDATES]
- int track_map_idx = 0
- cone track_map [MAX_CONES_MAP]

5.1.1 Detailed Description

5.1.2 Function Documentation

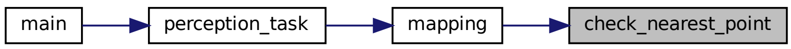
5.1.2.1 calculate_circle_points()

```
void calculate_circle_points (
            float center_x,
            float center_y,
            int color,
            cone * circle_points )
```

5.1.2.2 check_nearest_point()

```
void check_nearest_point (
            int angle,
            float new_point_x,
            float new_point_y,
            int color,
            cone_border * cone_borders )
```

Here is the caller graph for this function:



5.1.2.3 find_closest_points()

```
void find_closest_points (
            Hough_circle_point_t * circumference_points,
            float point_x,
            float point_y,
            Hough_circle_point_t * reference_points,
            int ref_size )
```

5.1.2.4 find_cone_center()

```
float* find_cone_center (
            Hough_circle_point_t * possible_centers,
            int center_count )
```

5.1.2.5 find_local_minima()

```
void find_local_minima (
            Hough_circle_point_t * points,
            int * first_min,
            int * second_min )
```

5.1.2.6 init_cone_borders()

```
void init_cone_borders (
            cone_border * cone_borders )
```

5.1.2.7 lidar()

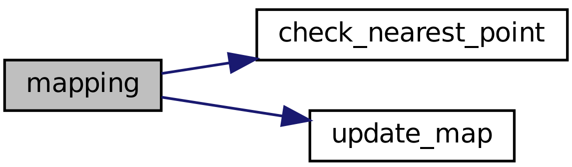
```
void lidar (
            float car_x,
            float car_y,
            pointcloud_t * measures )
```

Here is the caller graph for this function:

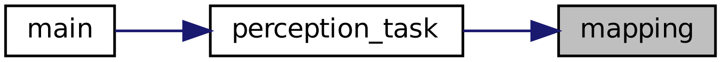
5.1.2.8 mapping()

```
void mapping (
            float car_x,
            float car_y,
            int car_angle,
            cone * detected_cones )
```

Here is the call graph for this function:



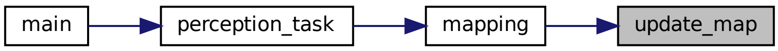
Here is the caller graph for this function:



5.1.2.9 update_map()

```
void update_map (
            cone * detected_cones )
```

Here is the caller graph for this function:



5.1.3 Variable Documentation

5.1.3.1 angle_step

```
const int angle_step = 1
```

5.1.3.2 candidates

```
candidate_cone candidates[MAX_CANDIDATES]
```

5.1.3.3 detected_cones

```
cone detected_cones[MAX_DETECTED_CONES]
```

5.1.3.4 distance_resolution

```
const float distance_resolution = 0.01f
```

5.1.3.5 ignore_distance

```
const float ignore_distance = 0.5f
```

5.1.3.6 n_candidates

```
int n_candidates = 0
```

5.1.3.7 sliding_window

```
const int sliding_window = 360
```

5.1.3.8 start_angle

```
int start_angle = 0
```

5.1.3.9 track_map

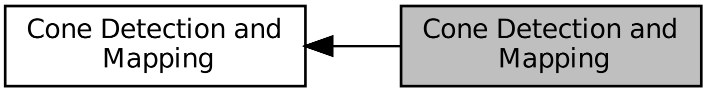
```
cone track_map[MAX_CONES_MAP]
```

5.1.3.10 track_map_idx

```
int track_map_idx = 0
```

5.2 Cone Detection and Mapping

Collaboration diagram for Cone Detection and Mapping:



5.3 Vehicle model and control

Functions

- void vehicle_model (float *car_x, float *car_y, int *car_angle, float pedal, float steering)
  Updates the vehicle's position and orientation based on pedal and steering inputs.

5.3.1 Detailed Description

5.3.2 Function Documentation

5.3.2.1 vehicle_model()

```
void vehicle_model (
            float * car_x,
            float * car_y,
            int * car_angle,
            float pedal,
            float steering )
```

Updates the vehicle's position and orientation based on pedal and steering inputs.

Simulates the vehicle's movement by updating its position and orientation.

This function simulates the movement of a vehicle using a simple bicycle model (Ackermann steering). It updates the current position (car_x, car_y) and the heading (car_angle) of the vehicle according to the pedal input (acceleration or braking) and the steering angle over a fixed time step.

The simulation parameters include:

- A time step (dt) for the simulation.
- A mass of the vehicle, which is used to compute acceleration from the force.

- A fixed wheelbase, representing the distance between the front and rear axles.
- A maximum achievable speed and a maximum braking force.

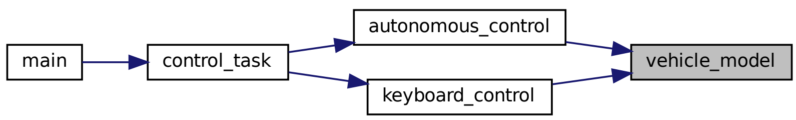
When the pedal input is positive, the vehicle accelerates towards a speed that is a proportion of maxSpeed. If the pedal input is not positive, the vehicle is assumed to be braking, and the deceleration is computed using the current speed and a maximum braking factor.

The vehicle's position is updated based on its current speed and orientation. The steering angle influences the change in the vehicle's heading only if the vehicle is moving above a small threshold speed. The angle conversion between degrees and radians is used to interface with the trigonometric functions.

Parameters

`in,out`	*car_x*	Pointer to the vehicle's x-coordinate. Updated based on the vehicle's motion.
`in,out`	*car_y*	Pointer to the vehicle's y-coordinate. Updated based on the vehicle's motion.
`in,out`	*car_angle*	Pointer to the vehicle's heading in degrees. Updated based on the computed orientation.
`in`	*pedal*	The pedal input; a positive value corresponds to acceleration while a non-positive value initiates braking.
`in`	*steering*	The steering input that determines the rate of change of the vehicle's heading.

Here is the caller graph for this function:



Chapter 6

Class Documentation

6.1 candidate_cone Struct Reference

Represents a candidate cone that might be validated after multiple detections.

```
#include <perception.h>
```

Public Attributes

- float x
- float y
- int color
- int detections

6.1.1 Detailed Description

Represents a candidate cone that might be validated after multiple detections.

This structure is used for maintaining a candidate cone before it is confirmed. It stores the position, color, and the number of detections that have contributed to this candidate.

Members:

- x: X position of the candidate cone.
- y: Y position of the candidate cone.
- color: Color of the candidate cone.
- detections: Count of how many times this candidate has been detected.

6.1.2 Member Data Documentation

6.1.2.1 color

```
int candidate_cone::color
```

6.1.2.2 detections

```
int candidate_cone::detections
```

6.1.2.3 x

```
float candidate_cone::x
```

6.1.2.4 y

```
float candidate_cone::y
```

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

- include/perception.h

6.2 cone Struct Reference

Processes the LiDAR measurements.

```
#include <perception.h>
```

Public Attributes

- float x
- float y
- int color

6.2.1 Detailed Description

Processes the LiDAR measurements.

Represents a cone detected in the environment.

This function simulates or processes a LiDAR scan, updating the point cloud information based on the car's current position.

Parameters

car_x	The X coordinate of the car's position.
car_y	The Y coordinate of the car's position.
measures	Pointer to the point cloud structure to be updated with the LiDAR measurements.

Performs real-time mapping using detected cones.

This function updates the array of detected cones by correlating the car's current position and orientation with cone positions.

Parameters

car_x	The X coordinate of the car's position.
car_y	The Y coordinate of the car's position.
car_angle	The orientation angle of the car (in degrees or radians as defined elsewhere).
detected_cones	Pointer to the array where detected cones are stored.

Checks and updates the nearest point for a given LiDAR scan angle.

This function compares a new detected point against existing data to determine if it belongs to a known cone border, updating the cone border data accordingly.

Parameters

angle	The scan angle index from the LiDAR measurement.
new_point_x	The X coordinate of the new detected point.
new_point_y	The Y coordinate of the new detected point.
color	The color associated with the new detected point.
cone_borders	Pointer to the cone_border structure to be updated.

Updates the global track map with the latest detected cones.

This function takes the newly detected cones and integrates them into the overall track map, which represents the accumulated perception of the environment.

Parameters

detected_cones	Pointer to the array of cones that have been newly detected.

This structure stores the position and color information for a cone. The positions are stored in floating point values in meters and can be converted to pixels when rendering. The color is represented using the Allegro color format.

Members:

- x: X position of the cone.
- y: Y position of the cone.
- color: Color of the cone (using Allegro color format).

6.2.2 Member Data Documentation

6.2.2.1 color

`int cone::color`

Color (in Allegro color format)

6.2.2.2 x

`float cone::x`

X position of the cone (in meters, converted to px when drawn)

6.2.2.3 y

`float cone::y`

Y position of the cone

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

- include/perception.h

6.3 cone_border Struct Reference

Represents a border of a cone as seen in a LiDAR scan.

`#include <perception.h>`

Public Attributes

- int angles [MAX_POINTS_PER_CONE]
- int color

6.3.1 Detailed Description

Represents a border of a cone as seen in a LiDAR scan.

This structure collects indices from the LiDAR scan that correspond to the boundaries of a cone. It also stores the color information associated with that cone border.

Members:

- angles: An array of indices (angles) in the LiDAR scan corresponding to the same cone border.
- color: Color of the cone border.

6.3.2 Member Data Documentation

6.3.2.1 angles

```
int cone_border::angles[MAX_POINTS_PER_CONE]
```

Indices (angles) in the LiDAR scan that map to the same cone border

6.3.2.2 color

```
int cone_border::color
```

Color of the cone

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

- include/perception.h

6.4 Hough_circle_point_t Struct Reference

Represents a point in the Hough circle transformation space.

```
#include <perception.h>
```

Public Attributes

- float x
- float y
- float distance
- int color

6.4.1 Detailed Description

Represents a point in the Hough circle transformation space.

This structure is used to store information about a point in the Hough circle transformation space, including its position, distance, and color.

6.4.2 Member Data Documentation

6.4.2.1 color

```
int Hough_circle_point_t::color
```

6.4.2.2 distance

```
float Hough_circle_point_t::distance
```

6.4.2.3 x

```
float Hough_circle_point_t::x
```

6.4.2.4 y

```
float Hough_circle_point_t::y
```

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

- include/perception.h
- src/perception.c

6.5 pointcloud_t Struct Reference

Contains the measures of distance made by the LiDAR at each angle.

```
#include <globals.h>
```

Public Attributes

- float point_x
- float point_y
- float distance
- int color

6.5.1 Detailed Description

Contains the measures of distance made by the LiDAR at each angle.

6.5.2 Member Data Documentation

6.5.2.1 color

`int pointcloud_t::color`

6.5.2.2 distance

`float pointcloud_t::distance`

6.5.2.3 point_x

`float pointcloud_t::point_x`

6.5.2.4 point_y

`float pointcloud_t::point_y`

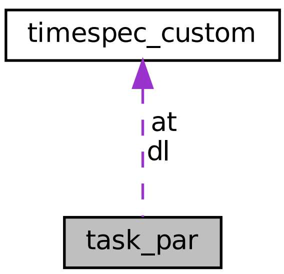
The documentation for this struct was generated from the following file:

- include/globals.h

6.6 task_par Struct Reference

`#include <ptask.h>`

Collaboration diagram for task_par:



Public Attributes

- int arg
- int period
- int deadline
- int prio
- int dmiss
- timespec_custom at
- timespec_custom dl
- pthread_t tid
- sem_t asem

6.6.1 Member Data Documentation

6.6.1.1 arg

```
int task_par::arg
```

6.6.1.2 asem

```
sem_t task_par::asem
```

6.6.1.3 at

```
timespec_custom task_par::at
```

6.6.1.4 deadline

```
int task_par::deadline
```

6.6.1.5 dl

```
timespec_custom task_par::dl
```

6.6.1.6 dmiss

```
int task_par::dmiss
```

6.6.1.7 period

```
int task_par::period
```

6.6.1.8 prio

```
int task_par::prio
```

6.6.1.9 tid

```
pthread_t task_par::tid
```

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

- include/ptask.h

6.7 timespec_custom Struct Reference

```
#include <ptask.h>
```

Public Attributes

- time_t tv_sec
- long tv_nsec

6.7.1 Member Data Documentation

6.7.1.1 tv_nsec

```
long timespec_custom::tv_nsec
```

6.7.1.2 tv_sec

```
time_t timespec_custom::tv_sec
```

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

- include/ptask.h

6.8 waypoint Struct Reference

Plans the trajectory for the car based on its current state and detected obstacles.

```
#include <trajectory.h>
```

Public Attributes

- float x
- float y

6.8.1 Detailed Description

Plans the trajectory for the car based on its current state and detected obstacles.

Represents a point in 2D space.

This function calculates the trajectory by considering the car's position and orientation, as well as the positions of detected cones. The resultant waypoints are stored in the provided trajectory array.

Parameters

car_x	The x-coordinate of the car's current position.
car_y	The y-coordinate of the car's current position.
car_angle	The heading angle of the car in radians.
detected_cones	Pointer to an array of cone structures representing detected cones.
trajectory	Pointer to an array of waypoint structures where the planned trajectory will be stored.

This structure is used to define a waypoint with x and y coordinates.

6.8.2 Member Data Documentation

6.8.2.1 x

```
float waypoint::x
```

6.8.2.2 y

```
float waypoint::y
```

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

- include/trajectory.h

Chapter 7

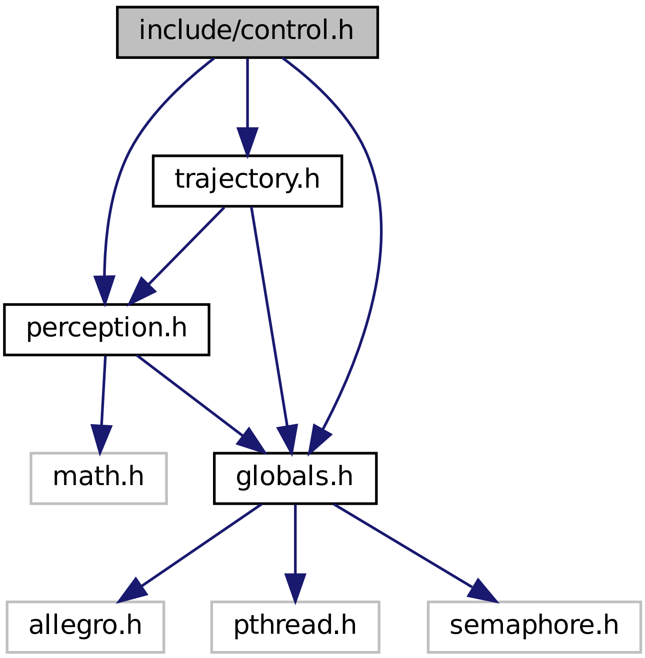
File Documentation

7.1 include/control.h File Reference

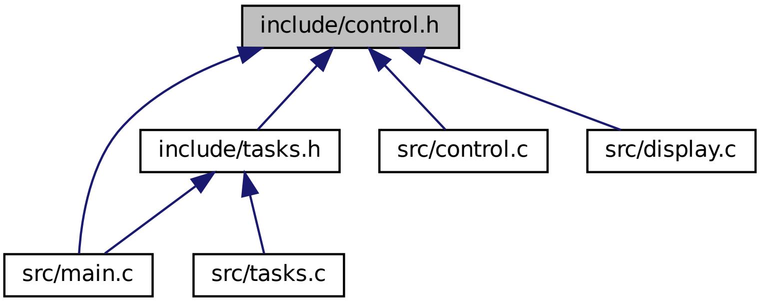
Declarations for vehicle control functions and variables in the 2D simulator.

```
#include <trajectory.h>
#include "globals.h"
#include "perception.h"
```

Include dependency graph for control.h:



This graph shows which files directly or indirectly include this file:



Functions

- void keyboard_control (float ∗car_x, float ∗car_y, int ∗car_angle)

    Adjusts vehicle controls based on keyboard input.
- void autonomous_control (float ∗car_x, float ∗car_y, int ∗car_angle, waypoint ∗trajectory)

    Autonomous control routine using centerline waypoints.

Variables

- float steering

    Controls the vehicle using keyboard input.
- float pedal

7.1.1 Detailed Description

Declarations for vehicle control functions and variables in the 2D simulator.

This header file is part of the 2D simulation project and provides the interface for controlling the simulated vehicle. It supports both manual (keyboard-driven) and autonomous (trajectory-based) modes.

Global Variables:

- steering: Current steering angle in radians.
- pedal: Throttle input as a normalized value in the range [0, 1].

Dependencies:

- trajectory.h: Used for waypoint and trajectory definitions.
- globals.h: Contains global variables and settings.
- perception.h: Provides functionalities related to situational awareness.

7.1.2 Function Documentation

7.1.2.1 autonomous_control()

```
void autonomous_control (
            float * car_x,
            float * car_y,
            int * car_angle,
            waypoint * center_waypoints )
```

Autonomous control routine using centerline waypoints.

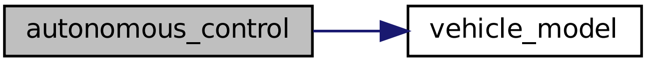
Implements an autonomous control strategy based on a provided centerline represented by an array of waypoints. The function follows these steps:

- Counts the number of valid centerline waypoints (terminated when a waypoint with x $<$ 0.0f is found).
- Filters the centerline to extract waypoints that are located ahead of the vehicle using the is_in_front helper.
- Identifies the closest valid waypoint ahead, and depending on the availability of neighboring points, computes a reference trajectory vector. This is done in one of three ways:
  - If at least three valid ahead waypoints exist, the reference vector is computed from the previous to the next waypoint surrounding the closest ahead waypoint.
  - If there are only one or two ahead points (but at least two total waypoints), the reference vector is derived by combining the vector from the vehicle to the last waypoint and the segment between the last two waypoints.
  - If only one waypoint is available, the vector from the vehicle to that waypoint is used as the reference.
- The computed reference trajectory is normalized. If normalization fails, a default forward direction is used.
- The vehicle's current heading is computed as a unit vector based on car_angle.
- The required steering correction (delta) is obtained by computing the sine of the angle difference through the 2D cross product between the normalized reference vector and the heading vector.
- A constant pedal value is applied.
- Finally, the updated control signals (pedal and delta for steering) are applied to the vehicle by calling vehicle↩ _model.

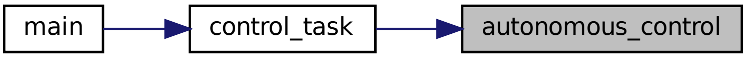
Parameters

in,out	*car_x*	Pointer to the vehicle's x-coordinate (in meters).
in,out	*car_y*	Pointer to the vehicle's y-coordinate (in meters).
in,out	*car_angle*	Pointer to the vehicle's orientation angle (in degrees).
in	*center_waypoints*	Pointer to an array of waypoints representing the desired centerline trajectory. The array should be terminated by a waypoint with x $<$ 0.0f.

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.2.2 keyboard_control()

```
void keyboard_control (
            float * car_x,
            float * car_y,
            int * car_angle )
```

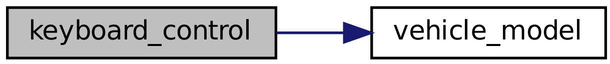
Adjusts vehicle controls based on keyboard input.

This function processes the state of keyboard keys to adjust the vehicle's pedal (speed) and steering angle. It increments or decrements the pedal value and steering angle within pre-defined limits depending on the keys pressed (e.g., KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT). After updating these control signals, the vehicle's state is updated by calling the vehicle_model function.

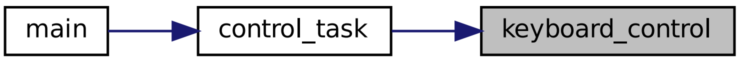
Parameters

in,out	*car_x*	Pointer to the vehicle's x-coordinate (in meters).
in,out	*car_y*	Pointer to the vehicle's y-coordinate (in meters).
in,out	*car_angle*	Pointer to the vehicle's orientation angle (in degrees).

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.3 Variable Documentation

7.1.3.1 pedal

```
float pedal [extern]
```

7.1.3.2 steering

```
float steering [extern]
```

Controls the vehicle using keyboard input.

Processes user keyboard inputs to modify the vehicle's position and orientation. The function updates the car's x and y coordinates, as well as its angular orientation (in degrees).

Parameters

car_x	Pointer to the car's x-coordinate.
car_y	Pointer to the car's y-coordinate.
car_angle	Pointer to the car's current angle (in degrees).

Controls the vehicle autonomously along a given trajectory.

Uses the provided trajectory (a sequence of waypoints) to automatically adjust the vehicle's steering and pedal inputs. This function updates the car's position and orientation by computing the necessary control actions based on the navigation path.

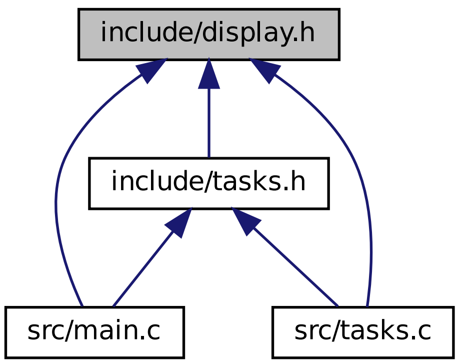
Parameters

car_x	Pointer to the car's x-coordinate.
car_y	Pointer to the car's y-coordinate.
car_angle	Pointer to the car's current orientation angle (in degrees).
trajectory	Pointer to the waypoint structure that defines the trajectory.

7.2 include/display.h File Reference

Header file declaring display rendering functions.

This graph shows which files directly or indirectly include this file:



Functions

- void draw_dir_arrow ()

  Draws a directional arrow.
- void draw_car (float car_x, float car_y, int car_angle)

  Draws a car shape on the display.
- void draw_track ()

  Draws the track.
- void draw_lidar (pointcloud_t ∗measures)

  Draws lidar measurements.
- void draw_detected_cones (cone ∗detected_cones)

  Draws detected cones.
- void draw_cone_map (cone ∗track_map, int track_map_idx)

  Draws the cone map (track map).

- void draw_perception ()

  Draws perception results.
- void draw_trajectory (waypoint *trajectory)

  Draws the planned trajectory.
- void update_display ()

  Updates the display.

7.2.1 Detailed Description

Header file declaring display rendering functions.

This file contains declarations for functions that handle the drawing of various display elements including car representation, lidar measurements, track, perception, trajectory, and more. These functions provide an interface for visualizing simulation data.

Note

This header relies on types and globals defined in "globals.h".

7.2.2 Function Documentation

7.2.2.1 draw_car()

```
void draw_car (
            float car_x,
            float car_y,
            int car_angle )
```

Draws a car shape on the display.

Parameters

car_x	X-coordinate of the car's position.
car_y	Y-coordinate of the car's position.
car_angle	Orientation angle of the car (typically in degrees).

Renders a graphical representation of the car using its position and orientation.

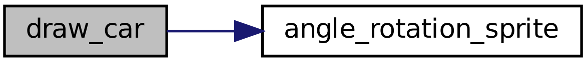
Draws a car shape on the display.

Calculates the proper position for the car sprite based on its coordinates in the world space (scaled by pixels per meter) and draws it using Allegro's rotate_scaled_sprite function. In DEBUG mode, the function also draws view angle lines to indicate the car's field of view.

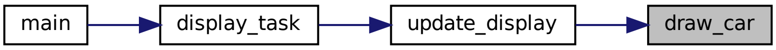
Parameters

car_x	The car's x-coordinate in world units.
car_y	The car's y-coordinate in world units.
car_angle	The car's angle in degrees.

Here is the call graph for this function:



Here is the caller graph for this function:



7.2.2.2 draw_cone_map()

```
void draw_cone_map (
            cone * track_map,
            int track_map_idx )
```

Draws the cone map (track map).

Parameters

track_map	Pointer to an array of cone structures representing the map.
track_map_idx	Index representing the current position or number of cones on the track map.

This function displays the cone-based track map, helping the user to visualize the layout and key points along the track.

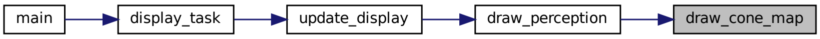
Draws the cone map (track map).

Iterates through the track map data and draws a white filled circle at each cone's position, properly mapping simulation world coordinates to the perception window.

Parameters

track_map	Pointer to an array of cones representing the track map.
track_map_idx	The number of valid entries in the track map array.

Here is the caller graph for this function:



7.2.2.3 draw_detected_cones()

```
void draw_detected_cones (
            cone * detected_cones )
```

Draws detected cones.

Parameters

detected_cones	Pointer to a cone structure that holds information about detected cones.

Renders the detected cones on the display, which could represent position markers or obstacles in the simulation environment.

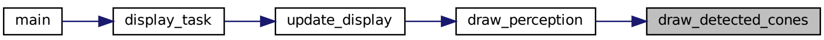
Draws detected cones.

Iterates through the array of detected cones and draws a filled circle at each cone's location, after mapping the coordinates from the simulation world to the perception window.

Parameters

detected_cones	Pointer to an array of detected cone structures.

Here is the caller graph for this function:



7.2.2.4 draw_dir_arrow()

```
void draw_dir_arrow ( )
```

Draws a directional arrow.

This function is used to render an arrow representing direction on the display. The arrow can be used as an indicator for direction in the simulation.

Draws a directional arrow.

Computes the arrow's start (at the car's position) and end points based on the car angle, using cosine and sine functions. The arrow shaft and head are drawn as thick green lines, with the head drawn at two angles to form a head shape.

7.2.2.5 draw_lidar()

```
void draw_lidar (
            pointcloud_t * measures )
```

Draws lidar measurements.

Parameters

measures	Pointer to a pointcloud_t structure containing lidar data.

Processes and graphically displays the lidar data as a set of points or a cloud, allowing visualization of the detected obstacles or environment.

Draws lidar measurements.

Clears the perception bitmap and centers it on the car's position. For each lidar measurement, computes the global-to-perception window mapping for the detected point and draws a line from the car's center to that point. The color of the line depends on whether a cone was detected or not.

Parameters

measures	Pointer to an array of lidar measurements.

Here is the caller graph for this function:

7.2.2.6 draw_perception()

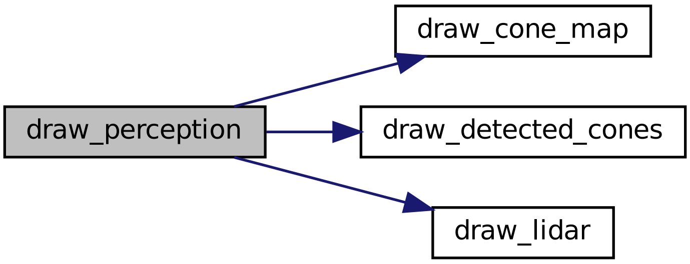
```
void draw_perception ( )
```

Draws perception results.

This function is responsible for visualizing the outcomes of the perception process, which might include sensor fusion results or object detection overlays.

Draws perception results.

Combines the lidar lines, detected cones, and cone map visualizations to create the perception overlay. Afterwards, the perception bitmap is blended onto the main display buffer offset in relation to the car's position. Here is the call graph for this function:



Here is the caller graph for this function:

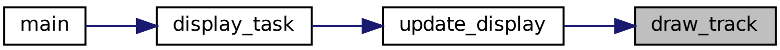
7.2.2.7 draw_track()

```
void draw_track ( )
```

Draws the track.

This function draws the track layout on the display, which might include boundaries, lanes, or other information necessary for simulation or visualization.

Checks that the track bitmap has valid dimensions, and if so, draws the track image onto the display buffer. If the bitmap dimensions are invalid, an error message is shown. Here is the caller graph for this function:



7.2.2.8 draw_trajectory()

```
void draw_trajectory (
            waypoint * trajectory )
```

Draws the planned trajectory.

Parameters

trajectory	Pointer to an array of waypoint structures representing the planned path.

Visualizes the computed trajectory on the display, providing insight into planned routes or maneuvers.

Draws the planned trajectory.

Clears and sets the background for the trajectory bitmap and iterates over each trajectory waypoint, drawing each as a filled circle. In DEBUG mode, the waypoint index is also displayed and an extra line is drawn from the car sprite center.

Parameters

trajectory	Pointer to an array of waypoint structures representing the car's trajectory.

Here is the caller graph for this function:

7.2.2.9 update_display()

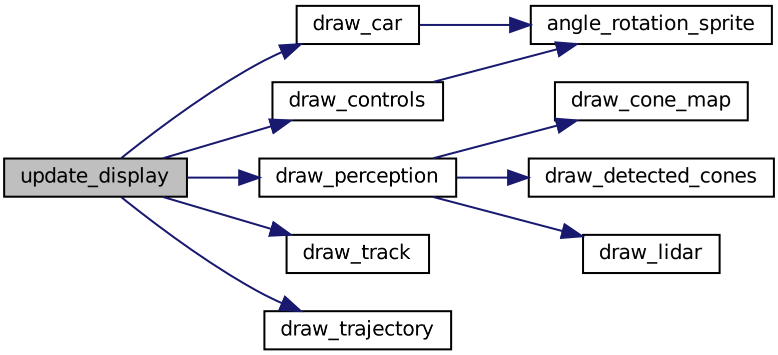
```
void update_display ( )
```

Updates the display.

This function refreshes the display, ensuring that all drawn elements are updated according to the latest simulation data.

Updates the display.

Locks the drawing mutex and sequentially renders the background, track, car, perception, trajectory, title text, and controls on the display buffer. Once all elements are rendered, the display buffer is blitted to the screen, and the mutex is subsequently unlocked. Here is the call graph for this function:



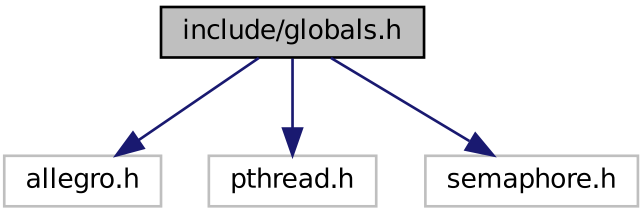
Here is the caller graph for this function:

7.3 include/globals.h File Reference

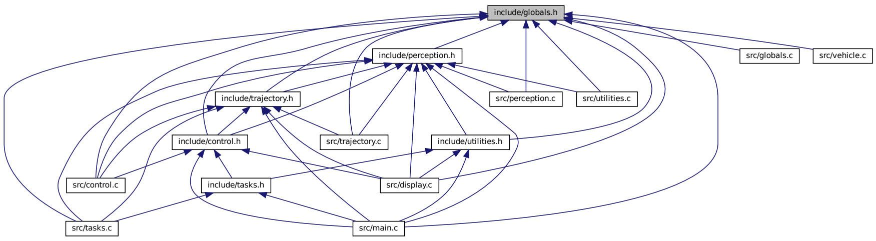
Global constants, variables, and structures for the simulation and visualization system.

```
#include <allegro.h>
#include <pthread.h>
#include <semaphore.h>
```

Include dependency graph for globals.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct pointcloud_t

    Contains the measures of distance made by the LiDAR at each angle.

Macros

- #define PROFILING
- #define PERCEPTION_PERIOD 50
- #define TRAJECTORY_PERIOD 10
- #define CONTROL_PERIOD 10
- #define DISPLAY_PERIOD 17
- #define PERCEPTION_DEADLINE PERCEPTION_PERIOD
- #define TRAJECTORY_DEADLINE TRAJECTORY_PERIOD
- #define CONTROL_DEADLINE CONTROL_PERIOD
- #define DISPLAY_DEADLINE DISPLAY_PERIOD
- #define PERCEPTION_PRIORITY 15
- #define TRAJECTORY_PRIORITY 20
- #define CONTROL_PRIORITY 25
- #define DISPLAY_PRIORITY 30
- #define px_per_meter 100
- #define deg2rad 0.017453292519943295769236907684886f
- #define maxThrottleHeight 100
- #define MAX_DETECTED_CONES 360

Variables

- pthread_mutex_t draw_mutex
- const char ∗ title
- const int X_MAX
- const int Y_MAX
- const float cone_radius
- BITMAP ∗ control_panel
- BITMAP ∗ steering_wheel
- BITMAP ∗ throttle_gauge
- BITMAP ∗ background
- BITMAP ∗ track
- BITMAP ∗ car
- BITMAP ∗ perception

- BITMAP ∗ trajectory_bmp
- BITMAP ∗ display_buffer
- int grass_green
- int asphalt_gray
- int white
- int pink
- int yellow
- int blue
- float car_x
- float car_y
- int car_angle
- pointcloud_t measures [MAX_DETECTED_CONES]
- sem_t lidar_sem

7.3.1 Detailed Description

Global constants, variables, and structures for the simulation and visualization system.

This header file declares all global macros, extern variables, and data types required across multiple modules for task management, visualization, and perception in a 2D simulation environment.

7.3.2 Tasks Constants

- Defines task periods (in milliseconds):
  - PERCEPTION_PERIOD: Period for the perception task.
  - TRAJECTORY_PERIOD: Period for the trajectory planning task.
  - CONTROL_PERIOD: Period for the control task.
  - DISPLAY_PERIOD: Period for the display update task.
- Defines deadlines (in milliseconds), set equal to the corresponding task periods:
  - PERCEPTION_DEADLINE, TRAJECTORY_DEADLINE, CONTROL_DEADLINE, DISPLAY_↩ DEADLINE.
- Defines task priorities (lower number indicates higher priority):
  - PERCEPTION_PRIORITY, TRAJECTORY_PRIORITY, CONTROL_PRIORITY, DISPLAY_PRIORITY.

7.3.3 Drawing Mutex

- Declares the external pthread_mutex_t variable draw_mutex used to synchronize drawing operations across threads.

7.3.4 Conversion Constants

- px_per_meter: Conversion factor from meters to pixels.
- deg2rad: Constant to convert degrees to radians.

7.3.5 Visualization Constants

- Declares external constants and variables for visualization:
  - title: Pointer to a string containing the window title.
  - X_MAX, Y_MAX: Maximum dimensions (in pixels) of the window or display.
  - cone_radius: Radius used for drawing cones.
- Declares external BITMAP pointers for various graphical assets:
  - control_panel: Bitmap for the control panel.
  - steering_wheel: Bitmap for the steering wheel graphic.
  - throttle_gauge: Bitmap for the throttle gauge.
  - background: Background image bitmap.
  - track: Bitmap representing the track.
  - car: Bitmap representing the car.
  - perception: Bitmap used for displaying perception data.
  - trajectory_bmp: Bitmap for trajectory visualization.
  - display_buffer: Bitmap used as a back buffer for display updates.
- Defines maxThrottleHeight: Maximum height for the throttle gauge.
- Declares external color variables for various color schemes used in the display:
  - grass_green, asphalt_gray, white, pink, yellow, blue.

7.3.6 Car Pose

- Declares global variables representing the car's state:
  - car_x, car_y: The car's x and y positions in meters.
  - car_angle: The angle of the car in degrees.

7.3.7 Perception Constants

- MAX_DETECTED_CONES: Macro defining the maximum number of cones that can be detected simultaneously.
- pointcloud_t: Struct representing a detected cone with the following fields:
  - point_x: x-coordinate of the detected point.
  - point_y: y-coordinate of the detected point.
  - distance: Distance from the sensor to the cone.
  - color: Color identifier for the cone.
- Declares an external array 'measures' of pointcloud_t to hold the detected cone data.
- Declares a semaphore 'lidar_sem' used to synchronize the passing of LiDAR data between tasks.

7.3.8 Macro Definition Documentation

7.3.8.1 CONTROL_DEADLINE

```
#define CONTROL_DEADLINE CONTROL_PERIOD
```

7.3.8.2 CONTROL_PERIOD

```
#define CONTROL_PERIOD 10
```

7.3.8.3 CONTROL_PRIORITY

```
#define CONTROL_PRIORITY 25
```

7.3.8.4 deg2rad

```
#define deg2rad 0.017453292519943295769236907684886f
```

7.3.8.5 DISPLAY_DEADLINE

```
#define DISPLAY_DEADLINE DISPLAY_PERIOD
```

7.3.8.6 DISPLAY_PERIOD

```
#define DISPLAY_PERIOD 17
```

7.3.8.7 DISPLAY_PRIORITY

```
#define DISPLAY_PRIORITY 30
```

7.3.8.8 MAX_DETECTED_CONES

```
#define MAX_DETECTED_CONES 360
```

7.3.8.9 maxThrottleHeight

```
#define maxThrottleHeight 100
```

7.3.8.10 PERCEPTION_DEADLINE

```
#define PERCEPTION_DEADLINE PERCEPTION_PERIOD
```

7.3.8.11 PERCEPTION_PERIOD

```
#define PERCEPTION_PERIOD 50
```

7.3.8.12 PERCEPTION_PRIORITY

```
#define PERCEPTION_PRIORITY 15
```

7.3.8.13 PROFILING

```
#define PROFILING
```

7.3.8.14 px_per_meter

```
#define px_per_meter 100
```

7.3.8.15 TRAJECTORY_DEADLINE

```
#define TRAJECTORY_DEADLINE TRAJECTORY_PERIOD
```

7.3.8.16 TRAJECTORY_PERIOD

```
#define TRAJECTORY_PERIOD 10
```

7.3.8.17 TRAJECTORY_PRIORITY

```
#define TRAJECTORY_PRIORITY 20
```

7.3.9 Variable Documentation

7.3.9.1 asphalt_gray

```
int asphalt_gray
```

7.3.9.2 background

```
background [extern]
```

-

7.3.9.3 blue

```
grass_green asphalt_gray white pink yellow blue
```

Integer values representing various colors used for rendering the simulation's elements.

- LiDAR Data:
  -

7.3.9.4 car

```
car [extern]
```

-

7.3.9.5 car_angle

```
car_angle  [extern]
```

The initial orientation angle of the car (in degrees).

- Graphical Resources:
  -

7.3.9.6 car_x

```
car_x  [extern]
```

The initial X-coordinate position of the car within the simulation.

-

7.3.9.7 car_y

```
car_y
```

The initial Y-coordinate position of the car within the simulation.

-

7.3.9.8 cone_radius

```
cone_radius  [extern]
```

The radius (in meters) for cone-related calculations, set to 0.05.

- Car Initial State:
  -

7.3.9.9 control_panel

```
control_panel [extern]
```

-

7.3.9.10 display_buffer

```
display_buffer [extern]
```

Bitmap pointers used for rendering various components of the simulation such as the background, track markings, car image, perception overlays, trajectory visualizations, and the overall display buffer.

- Color Definitions:
  -

7.3.9.11 draw_mutex

```
draw_mutex [extern]
```

A POSIX mutex that ensures mutual exclusion during drawing operations.

Note

The multiplier 'px_per_meter' used for computing X_MAX and Y_MAX is defined elsewhere.

7.3.9.12 grass_green

```
int grass_green [extern]
```

7.3.9.13 lidar_sem

```
lidar_sem [extern]
```

A POSIX semaphore used to synchronize access to LiDAR data.

-

7.3.9.14 measures

```
measures [extern]
```

An array of 360 LiDAR measurements (one per degree) representing the simulated point cloud data.

- Thread Synchronization:
  -

7.3.9.15 perception

```
perception [extern]
```

-

7.3.9.16 pink

```
int pink
```

7.3.9.17 steering_wheel

```
steering_wheel [extern]
```

-

7.3.9.18 throttle_gauge

```
throttle_gauge [extern]
```

Bitmap pointers for UI elements including the control panel, steering wheel, and throttle gauge.

-

7.3.9.19 title

`title [extern]`

The simulation title ("2D FSAE sim by rimaturus").

- Simulation Dimensions:
  –

7.3.9.20 track

`track [extern]`

•

7.3.9.21 trajectory_bmp

`trajectory_bmp [extern]`

•

7.3.9.22 white

`int white`

7.3.9.23 X_MAX

`X_MAX [extern]`

The maximum horizontal dimension of the simulation in pixels, computed as 19 ∗ px_per_meter.

•

7.3.9.24 Y_MAX

```
Y_MAX [extern]
```

The maximum vertical dimension of the simulation in pixels, computed as 12 $*$ px_per_meter.

- Physics and Geometry:
  –

7.3.9.25 yellow

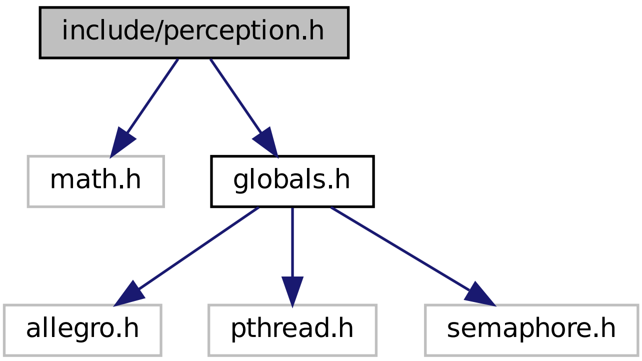
```
int yellow [extern]
```

7.4 include/perception.h File Reference

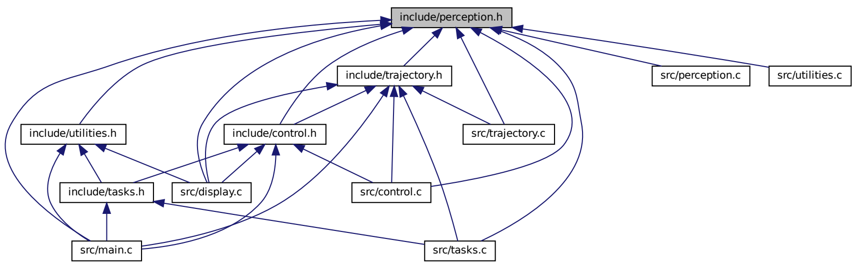
Header file for perception-related types, constants, and function declarations.

```
#include <math.h>
#include "globals.h"
```

Include dependency graph for perception.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct cone
  Processes the LiDAR measurements.
- struct cone_border
  Represents a border of a cone as seen in a LiDAR scan.
- struct candidate_cone
  Represents a candidate cone that might be validated after multiple detections.
- struct Hough_circle_point_t
  Represents a point in the Hough circle transformation space.

Macros

- #define MAX_POINTS_PER_CONE 180
- #define MAX_CONES_MAP 3000
- #define maxRange 10.0f
- #define MAX_CANDIDATES 100000
- #define DETECTIONS_THRESHOLD 10

Functions

- void lidar (float car_x, float car_y, pointcloud_t ∗measures)
- void mapping (float car_x, float car_y, int car_angle, cone ∗detected_cones)
- void check_nearest_point (int angle, float new_point_x, float new_point_y, int color, cone_border ∗cone_↩ borders)
- void update_map (cone ∗detected_cones)

Variables

- const float ignore_distance
- const int sliding_window
- const int angle_step
- int start_angle
- cone detected_cones [MAX_DETECTED_CONES]
- cone track_map [MAX_CONES_MAP]
- int track_map_idx

7.4.1 Detailed Description

Header file for perception-related types, constants, and function declarations.

This file contains the datatypes and function prototypes used for perception in the 2D simulation. It includes structures to represent cones, cone borders, candidate cones, and mapping functionalities such as LiDAR scanning, real-time mapping, and track map updating.

Global constants and configurable parameters:

- MAX_POINTS_PER_CONE: Maximum number of points (angles) that can define the border of a cone.
- MAX_CONES_MAP: Maximum number of cones that can be stored in the track map.

- maxRange: Maximum detection range (in meters) for the LiDAR.
- ignore_distance: Distance threshold below which detected points may be ignored.
- sliding_window, angle_step, start_angle: Parameters that control the LiDAR scanning and cone detection algorithm.

Global arrays:

- detected_cones: Array containing the cones detected by perception.
- track_map: Array representing the global track map of cones.
- track_map_idx: Current index in the track_map array.

Other constants:

- MAX_CANDIDATES: Maximum number of candidate cones tracked.
- DETECTIONS_THRESHOLD: Minimum number of detections required to consider a candidate cone valid.

7.4.2 Macro Definition Documentation

7.4.2.1 DETECTIONS_THRESHOLD

```
#define DETECTIONS_THRESHOLD 10
```

7.4.2.2 MAX_CANDIDATES

```
#define MAX_CANDIDATES 100000
```

7.4.2.3 MAX_CONES_MAP

```
#define MAX_CONES_MAP 3000
```

7.4.2.4 MAX_POINTS_PER_CONE

```
#define MAX_POINTS_PER_CONE 180
```

7.4.2.5 maxRange

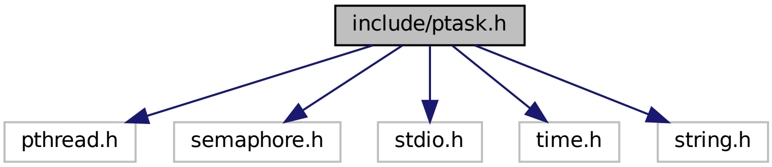
```
#define maxRange 10.0f
```

7.5 include/ptask.h File Reference

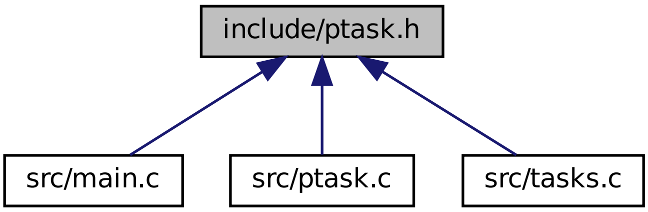
Interface for managing periodic tasks and time utilities.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
```

Include dependency graph for ptask.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct timespec_custom
- struct task_par

Macros

- #define MAX_TASKS 32
- #define MICRO 1
- #define NANO 2
- #define ACT 1
- #define DEACT 0

Typedefs

- typedef struct timespec_custom timespec_custom
- typedef struct task_par task_par

Functions

- void time_copy (timespec_custom ∗td, timespec_custom ts)
- void time_add_ms (timespec_custom ∗t, int ms)
- int time_cmp (timespec_custom t1, timespec_custom t2)
- void ptask_init (int policy)
- long get_systime (int unit)
- int task_create (int i, void ∗(∗task)(void ∗), int period, int drel, int prio, int aflag)
- int get_task_index (void ∗arg)
- void wait_for_activation (int i)
- void task_activate (int i)
- int deadline_miss (int i)
- void wait_for_period (int i)
- void task_set_period (int i, int per)
- void task_set_deadline (int i, int drel)
- int task_period (int i)
- int task_deadline (int i)
- int task_dmiss (int i)
- void task_atime (int i, timespec_custom ∗at)
- void task_adline (int i, timespec_custom ∗dl)
- void wait_for_task_end (int i)

Variables

- task_par tp [MAX_TASKS]
- timespec_custom ptask_t0
- int ptask_policy

7.5.1 Detailed Description

Interface for managing periodic tasks and time utilities.

This header file defines data types, macros, global variables, and function prototypes that collectively support a periodic real-time task management system.

The API supports:

- Configuring a maximum number of tasks via MAX_TASKS.
- Defining time units using MICRO (for microseconds) and NANO (for nanoseconds).
- Controlling task activation state with the ACT and DEACT flags.
- Using a custom time structure (timespec_custom) with the same layout as struct timespec, enabling compatibility with standard POSIX time functions (e.g., clock_gettime, clock_nanosleep).

Two primary data structures are provided:

- timespec_custom: A custom time structure for representing seconds and nanoseconds.
- task_par: A structure representing a periodic task that contains the task identifier, period (in milliseconds), relative deadline (in milliseconds), scheduling priority, a counter for deadline misses, activation and deadline timestamps, a thread identifier, and a semaphore for activation control.

Global Variables:

- tp[MAX_TASKS]: Array of task_par structures holding task parameters.
- ptask_t0: Reference starting time for the system.
- ptask_policy: Global scheduling policy to be applied to all tasks.

Provided Functions:

- time_copy(): Copies a timespec_custom value.
- time_add_ms(): Adds a given number of milliseconds to a timespec_custom value.
- time_cmp(): Compares two timespec_custom values, returning 1 if the first is greater, -1 if it is smaller, and 0 if they are equal.
- ptask_init(): Initializes the periodic task system by setting the scheduling policy, capturing the reference system time, and initializing activation semaphores.
- get_systime(): Returns the elapsed system time since initialization in either microseconds or nanoseconds.
- task_create(): Creates a new periodic task with specified period, relative deadline, priority, and activation flag. If activated immediately, the task's semaphore is posted.
- get_task_index(): Retrieves the task index from the task parameter pointer.
- wait_for_activation(): Blocks the task until its activation semaphore is posted. It then updates the task's activation and deadline times based on its period and deadline.
- task_activate(): Posts the semaphore to activate the task.

- deadline_miss(): Checks if the current time has exceeded the task's deadline; if so, it increments the deadline miss counter and returns 1.
- wait_for_period(): Uses absolute time waiting (clock_nanosleep) to synchronize the task with its defined period, avoiding cumulative time drift.
- Task Parameter Setters and Getters:
  - task_set_period() and task_set_deadline(): Modify a task's period and relative deadline.
  - task_period(), task_deadline(), and task_dmiss(): Retrieve a task's period, relative deadline, and the count of deadline misses.
  - task_atime() and task_adline(): Get the next activation time and current deadline time for a task.
- wait_for_task_end(): Waits (joins) for a task thread to complete execution.

Note

The function implementations are only compiled when PTASK_IMPLEMENTATION is defined.

Author

Your Name or Organization

Date

YYYY-MM-DD

7.5.2 Macro Definition Documentation

7.5.2.1 ACT

```
#define ACT 1
```

7.5.2.2 DEACT

```
#define DEACT 0
```

7.5.2.3 MAX_TASKS

```
#define MAX_TASKS 32
```

7.5.2.4 MICRO

```
#define MICRO 1
```

7.5.2.5 NANO

```
#define NANO 2
```

7.5.3 Typedef Documentation

7.5.3.1 task_par

```
typedef struct task_par task_par
```

7.5.3.2 timespec_custom

```
typedef struct timespec_custom timespec_custom
```

7.5.4 Function Documentation

7.5.4.1 deadline_miss()

```
int deadline_miss (
             int i )
```

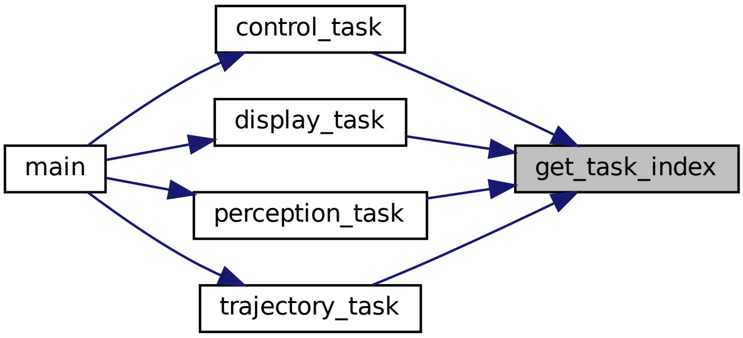
7.5.4.2 get_systime()

```
long get_systime (
             int unit )
```

7.5.4.3 get_task_index()

```
int get_task_index (
            void * arg )
```

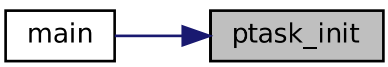
Here is the caller graph for this function:



7.5.4.4 ptask_init()

```
void ptask_init (
            int policy )
```

Here is the caller graph for this function:



7.5.4.5 task_activate()

```
void task_activate (
            int i )
```

7.5.4.6 task_adline()

```
void task_adline (
            int i,
            timespec_custom * dl )
```

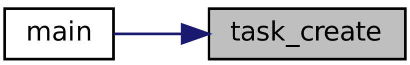
7.5.4.7 task_atime()

```
void task_atime (
            int i,
            timespec_custom * at )
```

7.5.4.8 task_create()

```
int task_create (
            int i,
            void *(*)(void *) task,
            int period,
            int drel,
            int prio,
            int aflag )
```

Here is the caller graph for this function:



7.5.4.9 task_deadline()

```
int task_deadline (
            int i )
```

7.5.4.10 task_dmiss()

```
int task_dmiss (
            int i )
```

7.5.4.11 task_period()

```
int task_period (
            int i )
```

7.5.4.12 task_set_deadline()

```
void task_set_deadline (
            int i,
            int drel )
```

7.5.4.13 task_set_period()

```
void task_set_period (
            int i,
            int per )
```

7.5.4.14 time_add_ms()

```
void time_add_ms (
            timespec_custom * t,
            int ms )
```

7.5.4.15 time_cmp()

```
int time_cmp (
            timespec_custom t1,
            timespec_custom t2 )
```

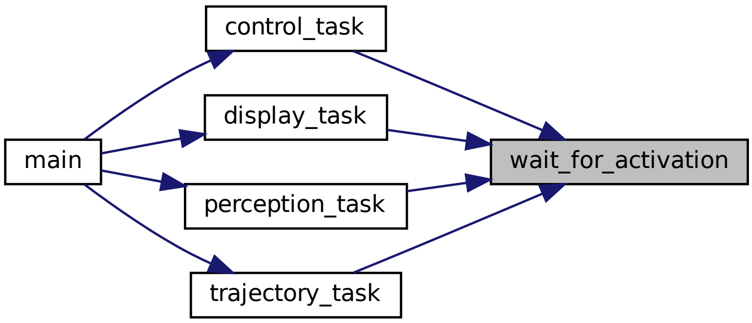
7.5.4.16 time_copy()

```
void time_copy (
            timespec_custom * td,
            timespec_custom ts )
```

7.5.4.17 wait_for_activation()

```
void wait_for_activation (
            int i )
```

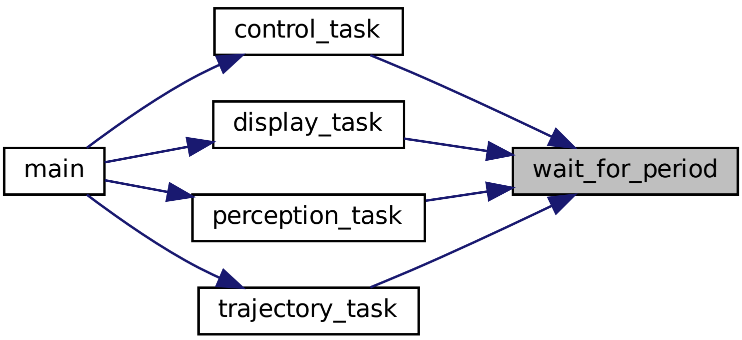
Here is the caller graph for this function:



7.5.4.18 wait_for_period()

```
void wait_for_period (
            int i )
```

Here is the caller graph for this function:



7.5.4.19 wait_for_task_end()

```
void wait_for_task_end (
            int i )
```

Here is the caller graph for this function:

7.5.5 Variable Documentation

7.5.5.1 ptask_policy

```
int ptask_policy [extern]
```

7.5.5.2 ptask_t0

```
timespec_custom ptask_t0 [extern]
```

7.5.5.3 tp

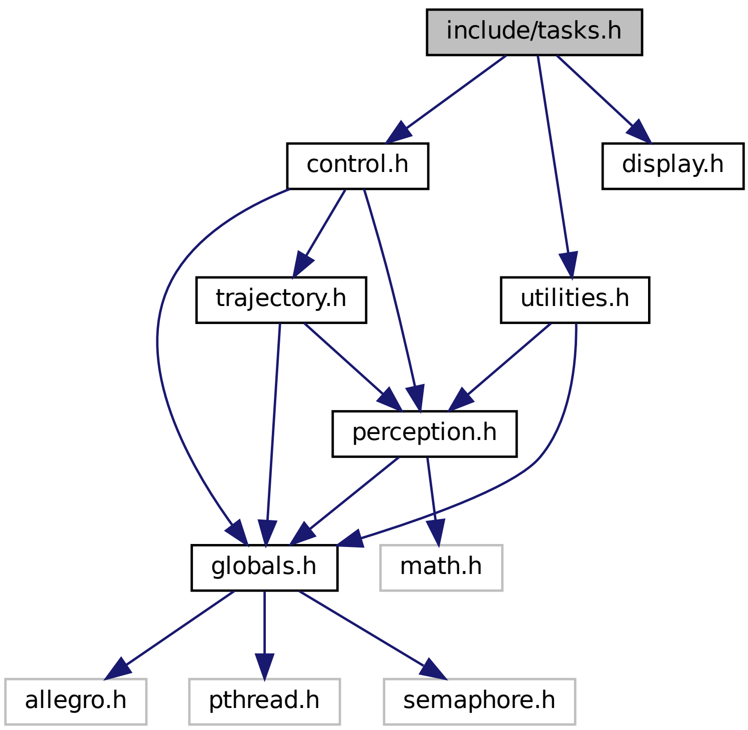
```
task_par tp[MAX_TASKS] [extern]
```

7.6 include/tasks.h File Reference

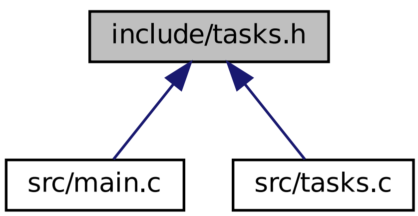
Declaration of task thread functions for the simulation.

```
#include "utilities.h"
#include "control.h"
#include "display.h"
```

Include dependency graph for tasks.h:



This graph shows which files directly or indirectly include this file:



Functions

- void ∗ perception_task (void ∗arg)

  Perception task.
- void ∗ trajectory_task (void ∗arg)
- void ∗ control_task (void ∗arg)
- void ∗ display_task (void ∗arg)

7.6.1 Detailed Description

Declaration of task thread functions for the simulation.

This header file declares thread functions for various tasks in the simulation:

- Perception: Processes sensor or simulation data to interpret the environment.
- Trajectory: Computes and updates planned paths or trajectories.
- Control: Implements control logic that may handle both manual and autonomous inputs.
- Display: Manages rendering of the simulation's visual output.

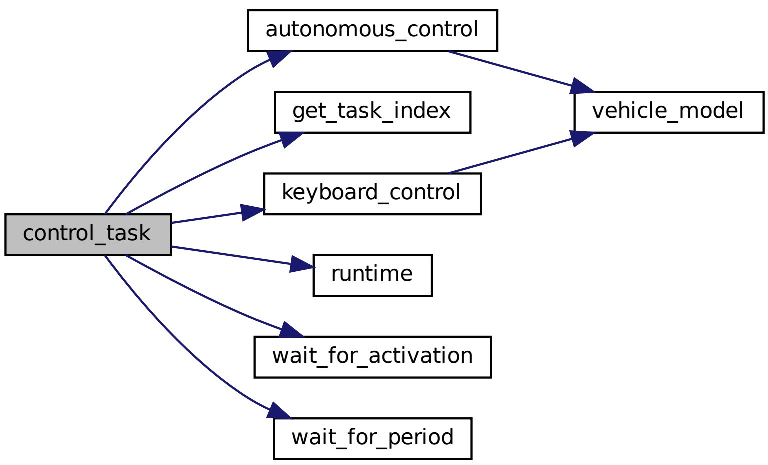
Each task is designed to be executed in its own thread, and the functions follow the prototype required by POSIX thread routines.

7.6.2 Function Documentation

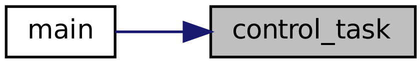
7.6.2.1 control_task()

```
void* control_task (
            void * arg )
```

Here is the call graph for this function:



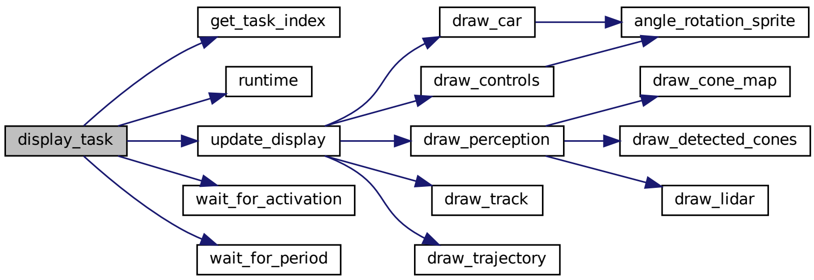
Here is the caller graph for this function:



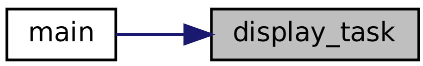
7.6.2.2 display_task()

```
void* display_task (
            void * arg )
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.2.3 perception_task()

```
void* perception_task (
            void * arg )
```

Perception task.

This function executes perception related operations, such as processing sensor data or simulation input to understand the surrounding environment.

Parameters

arg	Pointer to the arguments required by the task. The actual type and usage of this parameter are defined by the implementation context.

Returns

Pointer representing the function's outcome, typically not used.

Trajectory planning task.

This function is responsible for calculating and updating the movement trajectory for the entity in simulation. It handles the logic for path planning based on environmental data and simulation dynamics.

Parameters

arg	Pointer to the arguments specific to the trajectory task.

Returns

Pointer representing the result of the function, usually unused.

Control task.

This function manages the control logic, determining inputs for either keyboard control or autonomous behavior. It integrates various control signals and decisions, ensuring the entity reacts appropriately within the simulation.

Parameters

arg	Pointer to a structure or value containing control parameters.

Returns

Pointer to the result of the control processing, typically not utilized.

Display update task.

This function handles the rendering process for the simulation's trajectory display. It updates the visual output based on the current state of the simulation data.

Parameters

arg	Pointer to display specific parameters or context information.

Returns

Pointer indicating the outcome of the display operation, generally unused.

Perception task.

This task acquires LiDAR measurements, resets the array of detected cones, and performs mapping based on the current position and orientation of the vehicle. After processing the sensor data, it signals the trajectory planning task using a semaphore. The task executes continuously until the ESC key is pressed.

Parameters

in	*arg*	Pointer used to determine the task's index.

Returns

Always returns NULL.

Note

The function uses a runtime measurement call to monitor its execution time.

Periodic trajectory planning task.

This task waits for a signal from the perception task to ensure updated sensor data is available. It then calculates a new trajectory for the vehicle based on its current position, orientation, and the detected cones from the environment. The task runs periodically and terminates when the ESC key is pressed.

Parameters

in	*arg*	Pointer used to determine the task's index.

Returns

Always returns NULL.

Note

The task's operation is encapsulated between runtime measurement calls.

Periodic control task.

This task is responsible for controlling the vehicle's movement. It chooses between keyboard-based control and autonomous control based on the state of the 'A' key. If autonomous mode is active (i.e., 'A' is pressed), it uses the computed trajectory, otherwise it relies on user input to drive the vehicle. Execution continues until the ESC key is pressed.

Parameters

in	*arg*	Pointer used to determine the task's index.

Returns

Always returns NULL.

Note

The function uses runtime measurement calls to monitor execution time per cycle.

Periodic display task.

This task updates the graphical display of the simulation environment. It refreshes the on-screen information such as the vehicle's position, sensor data, and other simulation elements. The task continues running until the ESC key is pressed.

Parameters

in	*arg*	Pointer used to determine the task's index.

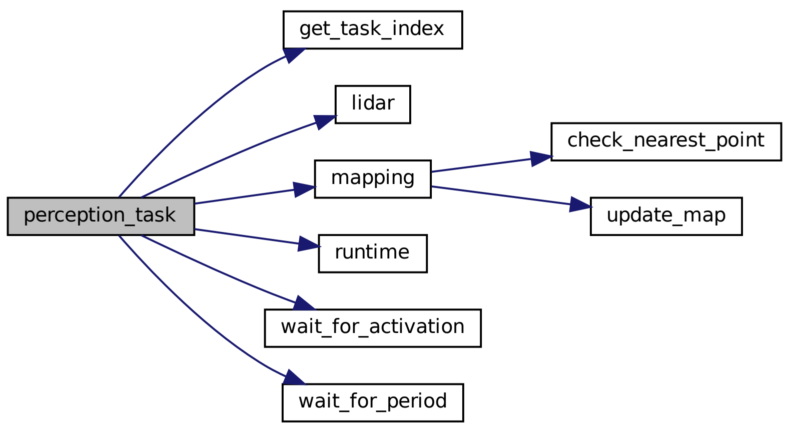
Returns

Always returns NULL.

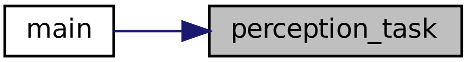
Note

Like the other tasks, execution time is tracked using runtime measurement calls.

Here is the call graph for this function:



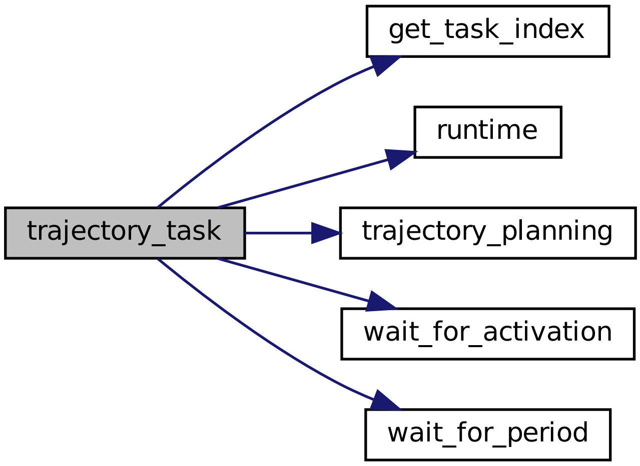
Here is the caller graph for this function:



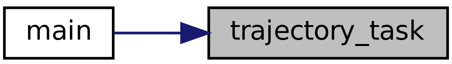
7.6.2.4 trajectory_task()

```
void* trajectory_task (
            void * arg )
```

Here is the call graph for this function:



Here is the caller graph for this function:

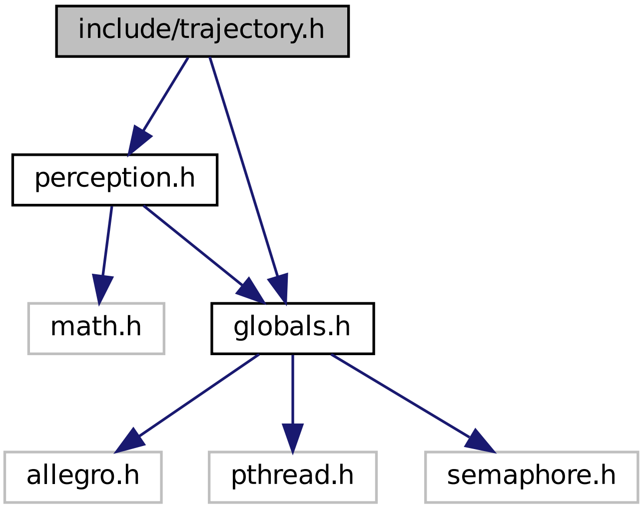


7.7 include/trajectory.h File Reference

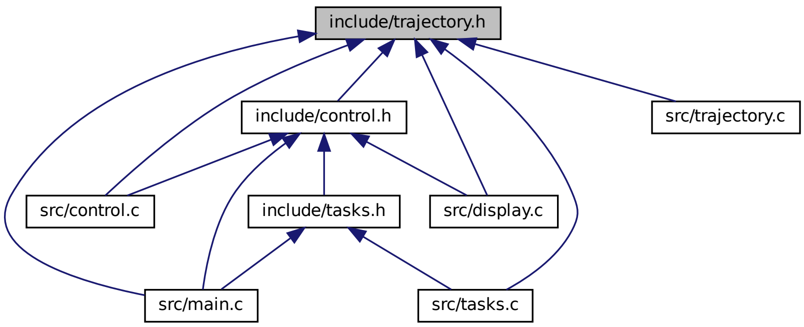
Provides declarations for trajectory planning based on cone detection.

```
#include "perception.h"
#include "globals.h"
```

Include dependency graph for trajectory.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct waypoint

  Plans the trajectory for the car based on its current state and detected obstacles.

Functions

- void trajectory_planning (float car_x, float car_y, float car_angle, cone ∗detected_cones, waypoint ∗trajectory)

Variables

- waypoint trajectory [2 ∗MAX_DETECTED_CONES]

    Global array of waypoints forming the planned trajectory.
- int trajectory_idx

    Global index used to track the number of waypoints currently stored in the trajectory.

7.7.1 Detailed Description

Provides declarations for trajectory planning based on cone detection.

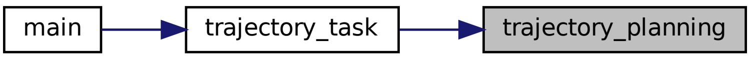
This header defines the data structures and function prototypes necessary for generating trajectories (waypoints) that guide the car through detected cones.

7.7.2 Function Documentation

7.7.2.1 trajectory_planning()

```
void trajectory_planning (
            float car_x,
            float car_y,
            float car_angle,
            cone * detected_cones,
            waypoint * trajectory )
```

Here is the caller graph for this function:



7.7.3 Variable Documentation

7.7.3.1 trajectory

```
trajectory [extern]
```

Global array of waypoints forming the planned trajectory.

This array stores the computed trajectory based on the detected cones. It is sized to accommodate up to twice the number of maximum detectable cones (2 ∗ MAX_DETECTED_CONES).

7.7.3.2 trajectory_idx

```
trajectory_idx  [extern]
```

Global index used to track the number of waypoints currently stored in the trajectory.

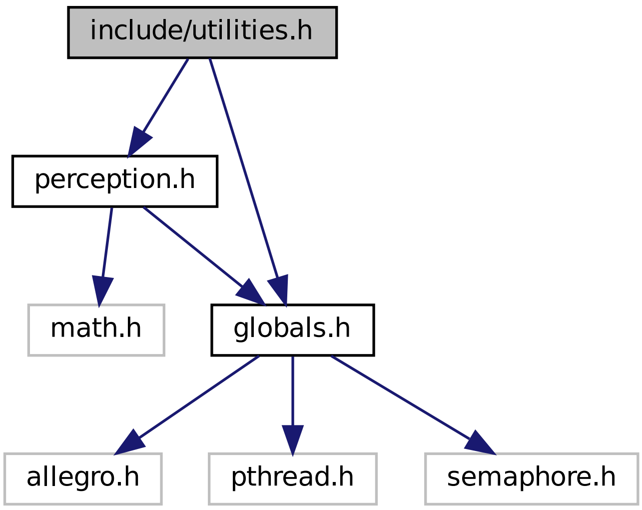
This variable indicates the current position in the trajectory array, facilitating the addition of new waypoints.

7.8 include/utilities.h File Reference

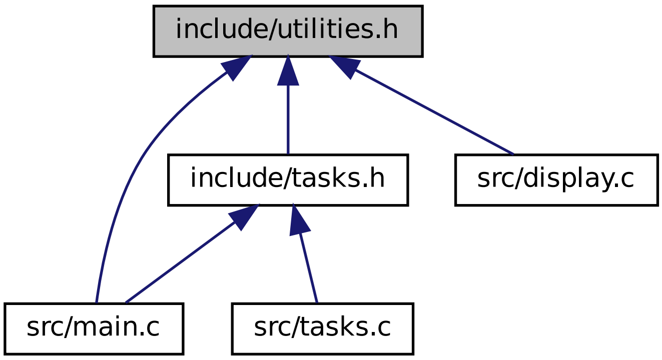
Utility function prototypes for cone management and runtime tasks.

```
#include <perception.h>
#include <globals.h>
```

Include dependency graph for utilities.h:



This graph shows which files directly or indirectly include this file:



Functions

- void init_cones (cone ∗cones)

    Initializes an array of cone structures.
- void load_cones_positions (const char ∗filename, cone ∗cones, int max_cones)
- float angle_rotation_sprite (float angle)
- void runtime (int stop_signal, char ∗task_name)

7.8.1 Detailed Description

Utility function prototypes for cone management and runtime tasks.

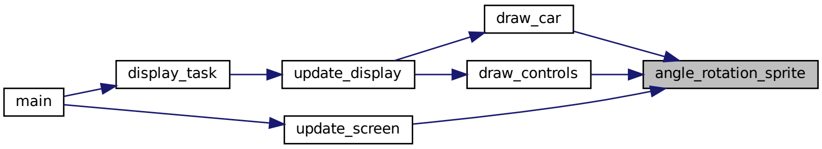
This header declares functions for initializing cone structures, loading cone positions from a file, calculating sprite rotations, and handling runtime processes.

7.8.2 Function Documentation

7.8.2.1 angle_rotation_sprite()

```
float angle_rotation_sprite (
            float angle )
```

Here is the caller graph for this function:



7.8.2.2 init_cones()

```
void init_cones (
            cone * cones )
```

Initializes an array of cone structures.

This function sets up the specified array of cones so that each cone is properly initialized.

Parameters

cones	Pointer to the first element of an array of cone structures.

Loads cone positions from a file.

Reads cone position data from the provided file and populates the cone array accordingly. Only up to `max_cones` entries are loaded, ensuring the array bounds are respected.

Parameters

filename	Path to the file containing cone position data.
cones	Pointer to an array of cone structures where the positions will be stored.
max_cones	The maximum number of cones to load from the file.

Computes the sprite rotation angle.

Calculates and returns a new rotation angle for a sprite based on the input `angle`. The calculation may involve normalization or other transformations suitable for sprite handling.

Parameters

angle	The initial angle value (in degrees or radians, as defined by the application's convention).

Returns

The adjusted angle after applying the rotation transformation.

Handles runtime processes based on a stop signal.

Executes runtime tasks, potentially including logging or task management, until a stop condition defined by `stop_signal` is met. The associated task description is provided via `task_name`.

Parameters

stop_signal	Signal value indicating when to terminate the runtime process.
task_name	A string identifier for the task being executed.

This function initializes each cone in the provided array by setting its x and y coordinates to 0.0f and its color to -1.

Parameters

cones	Pointer to the array of cone structures to be initialized.

Loads cone positions and colors from a YAML file.

This function parses the specified YAML file to extract cone position (x and y) and color data. The YAML file is expected to have a top-level key "cones" containing a sequence of cone mappings. Each cone mapping should provide:

- "x": A string representing the x coordinate (converted to float and scaled).
- "y": A string representing the y coordinate (converted to float and scaled).
- "color": A string representing the cone color (e.g., "yellow" or "blue").

The function prints out diagnostic messages to the standard output regarding the file being loaded, and reports any errors encountered during file opening or YAML parsing.

Parameters

filename	Path to the YAML file containing cone data.
cones	Pointer to the array of cone structures to be populated.
max_cones	Maximum number of cones to load into the array.

Converts an angle (in degrees) to a sprite rotation value.

This helper function transforms an input angle (in degrees) into a corresponding sprite rotation value. The conversion is based on the formula: sprite_rotation = 64.0f - (128.0f $*$ angle / 180.0f).

Parameters

angle	Angle in degrees.

Returns

Sprite rotation value as a floating-point number.

Measures code performance based on a runtime signal.

This function is a helper to measure code execution performance if the PROFILING macro is defined. When called with a start signal (stop_signal = 0), it records the current time and prints a "START" message. When called with a stop signal (stop_signal != 0), it calculates the difference from the start time, prints an "END" message, and optionally prints the elapsed runtime in microseconds.

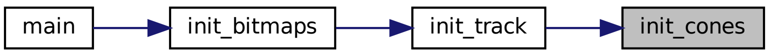
Note

This function depends on the POSIX clock_gettime() function and is active only if PROFILING is defined.

Parameters

stop_signal	A flag indicating whether to start (0) or stop (non-zero) the runtime measurement.
task_name	A string representing the name of the task for which the runtime is being measured.

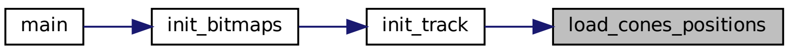
Here is the caller graph for this function:



7.8.2.3 load_cones_positions()

```
void load_cones_positions (
            const char * filename,
            cone * cones,
            int max_cones )
```

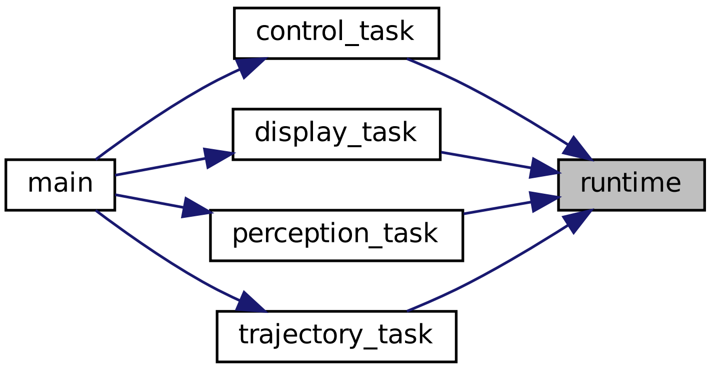
Here is the caller graph for this function:



7.8.2.4 runtime()

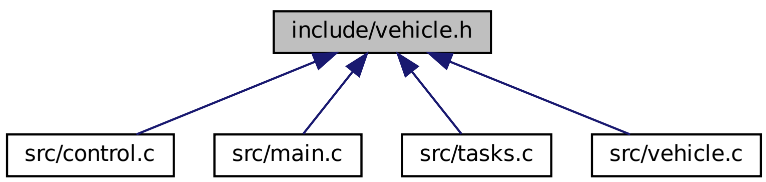
```
void runtime (
            int stop_signal,
            char * task_name )
```

Here is the caller graph for this function:



7.9 include/vehicle.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void vehicle_model (float ∗car_x, float ∗car_y, int ∗car_angle, float pedal, float steering)

  Simulates the vehicle's movement by updating its position and orientation.

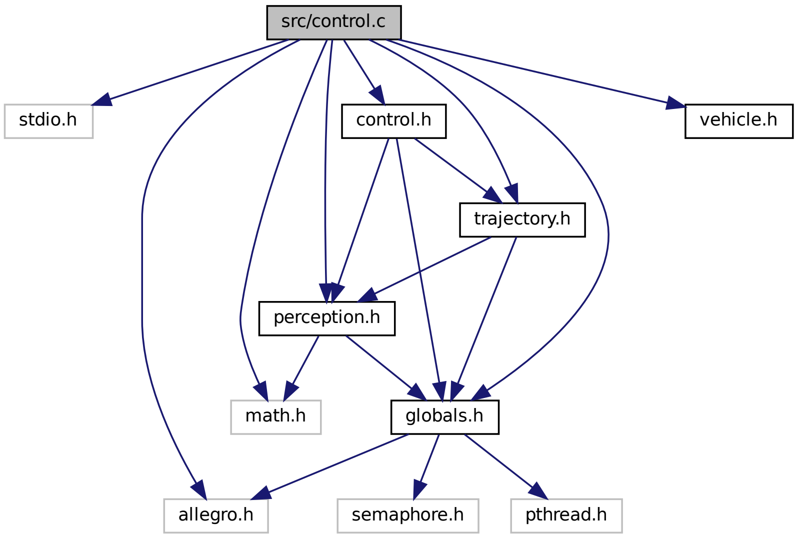
7.10 readme.md File Reference

7.11 src/control.c File Reference

Implements vehicle control functions including both manual (keyboard) and autonomous (centerline based) control.

```
#include <stdio.h>
#include <math.h>
#include <allegro.h>
#include "trajectory.h"
#include "perception.h"
#include "globals.h"
#include "control.h"
#include "vehicle.h"
```

Include dependency graph for control.c:



Functions

- void keyboard_control (float ∗car_x, float ∗car_y, int ∗car_angle)

    Adjusts vehicle controls based on keyboard input.
- void autonomous_control (float ∗car_x, float ∗car_y, int ∗car_angle, waypoint ∗center_waypoints)

    Autonomous control routine using centerline waypoints.

Variables

- float pedal = 0.0f
- float steering = 0.0f

    Controls the vehicle using keyboard input.

7.11.1 Detailed Description

Implements vehicle control functions including both manual (keyboard) and autonomous (centerline based) control.

This file contains methods to update the vehicle's motion state. The manual control routine (keyboard_control) processes user keystrokes to adjust speed (pedal) and steering angle, while the autonomous control routine uses a centerline of waypoints to compute a reference trajectory and derive the appropriate steering correction.

7.11.2 Function Documentation

7.11.2.1 autonomous_control()

```
void autonomous_control (
            float * car_x,
            float * car_y,
            int * car_angle,
            waypoint * center_waypoints )
```

Autonomous control routine using centerline waypoints.

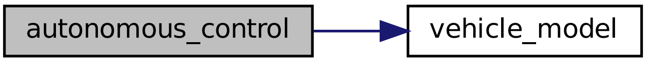
Implements an autonomous control strategy based on a provided centerline represented by an array of waypoints. The function follows these steps:

- Counts the number of valid centerline waypoints (terminated when a waypoint with x $<$ 0.0f is found).
- Filters the centerline to extract waypoints that are located ahead of the vehicle using the is_in_front helper.
- Identifies the closest valid waypoint ahead, and depending on the availability of neighboring points, computes a reference trajectory vector. This is done in one of three ways:
  - If at least three valid ahead waypoints exist, the reference vector is computed from the previous to the next waypoint surrounding the closest ahead waypoint.
  - If there are only one or two ahead points (but at least two total waypoints), the reference vector is derived by combining the vector from the vehicle to the last waypoint and the segment between the last two waypoints.
  - If only one waypoint is available, the vector from the vehicle to that waypoint is used as the reference.
- The computed reference trajectory is normalized. If normalization fails, a default forward direction is used.
- The vehicle's current heading is computed as a unit vector based on car_angle.
- The required steering correction (delta) is obtained by computing the sine of the angle difference through the 2D cross product between the normalized reference vector and the heading vector.
- A constant pedal value is applied.
- Finally, the updated control signals (pedal and delta for steering) are applied to the vehicle by calling vehicle_model.

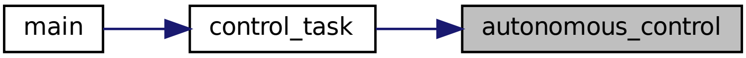
Parameters

in,out	*car_x*	Pointer to the vehicle's x-coordinate (in meters).
in,out	*car_y*	Pointer to the vehicle's y-coordinate (in meters).
in,out	*car_angle*	Pointer to the vehicle's orientation angle (in degrees).
in	*center_waypoints*	Pointer to an array of waypoints representing the desired centerline trajectory. The array should be terminated by a waypoint with x $<$ 0.0f.

Here is the call graph for this function:



Here is the caller graph for this function:



7.11.2.2 keyboard_control()

```
void keyboard_control (
            float * car_x,
            float * car_y,
            int * car_angle )
```

Adjusts vehicle controls based on keyboard input.

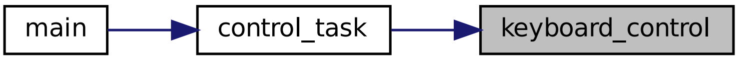
This function processes the state of keyboard keys to adjust the vehicle's pedal (speed) and steering angle. It increments or decrements the pedal value and steering angle within pre-defined limits depending on the keys pressed (e.g., KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT). After updating these control signals, the vehicle's state is updated by calling the vehicle_model function.

Parameters

in,out	*car_x*	Pointer to the vehicle's x-coordinate (in meters).
in,out	*car_y*	Pointer to the vehicle's y-coordinate (in meters).
in,out	*car_angle*	Pointer to the vehicle's orientation angle (in degrees).

Here is the call graph for this function:

Here is the caller graph for this function:



7.11.3 Variable Documentation

7.11.3.1 pedal

```
float pedal = 0.0f
```

7.11.3.2 steering

```
float steering = 0.0f
```

Controls the vehicle using keyboard input.

Processes user keyboard inputs to modify the vehicle's position and orientation. The function updates the car's x and y coordinates, as well as its angular orientation (in degrees).

Parameters

car_x	Pointer to the car's x-coordinate.
car_y	Pointer to the car's y-coordinate.
car_angle	Pointer to the car's current angle (in degrees).

Controls the vehicle autonomously along a given trajectory.

Uses the provided trajectory (a sequence of waypoints) to automatically adjust the vehicle's steering and pedal inputs. This function updates the car's position and orientation by computing the necessary control actions based on the navigation path.

Parameters

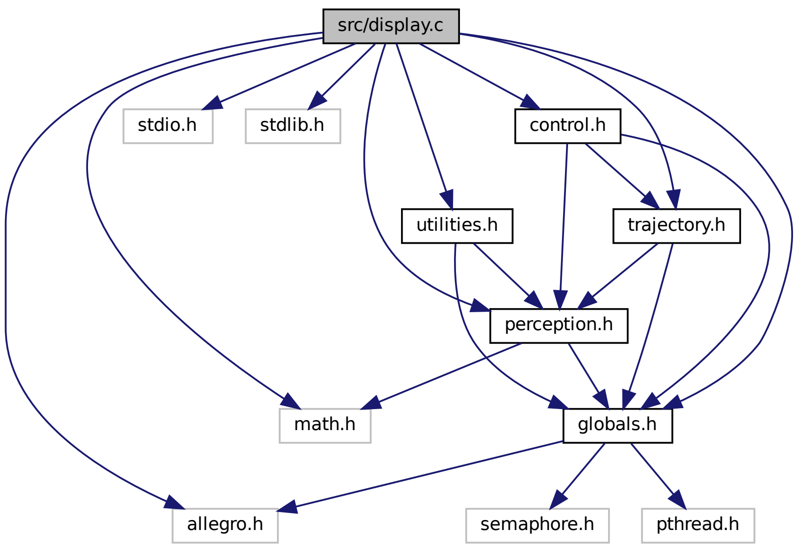
car_x	Pointer to the car's x-coordinate.
car_y	Pointer to the car's y-coordinate.
car_angle	Pointer to the car's current orientation angle (in degrees).
trajectory	Pointer to the waypoint structure that defines the trajectory.

7.12 src/display.c File Reference

Rendering and display update functions for the 2D simulation using Allegro.

```
#include <allegro.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "globals.h"
#include "perception.h"
#include "trajectory.h"
#include "utilities.h"
#include "control.h"
```

Include dependency graph for display.c:



Functions

- void draw_dir_arrow ()

Draws a directional arrow representing the car's orientation.
- void draw_car (float car_x, float car_y, int car_angle)

  Renders the car sprite rotated to its current angle.
- void draw_track ()

  Draws the track.
- void draw_lidar (pointcloud_t *measures)

  Draws the lidar/perception view.
- void draw_detected_cones (cone *detected_cones)

  Renders the detected cones on the perception bitmap.
- void draw_cone_map (cone *track_map, int track_map_idx)

  Draws the cone map overlay.
- void draw_perception ()

  Renders the complete perception layer.
- void draw_trajectory (waypoint *trajectory)

  Draws the trajectory of the car.
- void draw_controls ()

  Draws user control indicators.
- void update_display ()

  Updates the entire display.

7.12.1 Detailed Description

Rendering and display update functions for the 2D simulation using Allegro.

This file contains functions responsible for drawing various elements of the simulation, including the car sprite, direction indicator, track, lidar/perception display, trajectory, and user controls (e.g., steering wheel and pedal gauge). It also handles the final composition of the display buffer and its transfer to the screen.

7.12.2 Function Documentation

7.12.2.1 draw_car()

```
void draw_car (
            float car_x,
            float car_y,
            int car_angle )
```

Renders the car sprite rotated to its current angle.

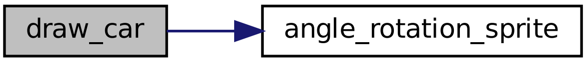
Draws a car shape on the display.

Calculates the proper position for the car sprite based on its coordinates in the world space (scaled by pixels per meter) and draws it using Allegro's rotate_scaled_sprite function. In DEBUG mode, the function also draws view angle lines to indicate the car's field of view.

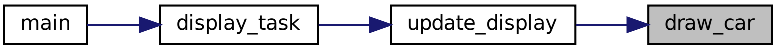
Parameters

car_x	The car's x-coordinate in world units.
car_y	The car's y-coordinate in world units.
car_angle	The car's angle in degrees.

Here is the call graph for this function:



Here is the caller graph for this function:



7.12.2.2 draw_cone_map()

```
void draw_cone_map (
            cone * track_map,
            int track_map_idx )
```

Draws the cone map overlay.

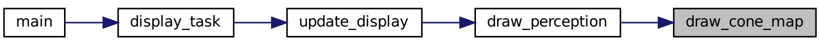
Draws the cone map (track map).

Iterates through the track map data and draws a white filled circle at each cone's position, properly mapping simulation world coordinates to the perception window.

Parameters

track_map	Pointer to an array of cones representing the track map.
track_map_idx	The number of valid entries in the track map array.

Here is the caller graph for this function:

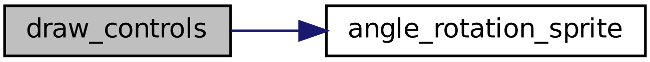


7.12.2.3 draw_controls()

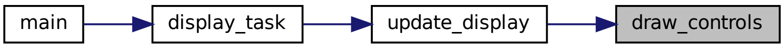
```
void draw_controls ( )
```

Draws user control indicators.

Renders the steering wheel sprite with the correct rotation and draws a gauge for pedal control. The gauge's fill color indicates the pedal level (green for positive, red for non-positive). Here is the call graph for this function:



Here is the caller graph for this function:



7.12.2.4 draw_detected_cones()

```
void draw_detected_cones (
            cone * detected_cones )
```

Renders the detected cones on the perception bitmap.

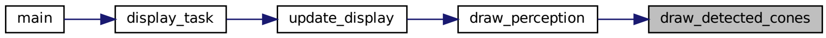
Draws detected cones.

Iterates through the array of detected cones and draws a filled circle at each cone's location, after mapping the coordinates from the simulation world to the perception window.

Parameters

detected_cones	Pointer to an array of detected cone structures.

Here is the caller graph for this function:



7.12.2.5 draw_dir_arrow()

```
void draw_dir_arrow ( )
```

Draws a directional arrow representing the car's orientation.

Draws a directional arrow.

Computes the arrow's start (at the car's position) and end points based on the car angle, using cosine and sine functions. The arrow shaft and head are drawn as thick green lines, with the head drawn at two angles to form a head shape.

7.12.2.6 draw_lidar()

```
void draw_lidar (
            pointcloud_t * measures )
```

Draws the lidar/perception view.

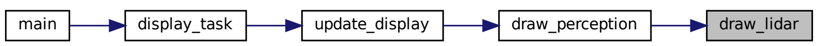
Draws lidar measurements.

Clears the perception bitmap and centers it on the car's position. For each lidar measurement, computes the global-to-perception window mapping for the detected point and draws a line from the car's center to that point. The color of the line depends on whether a cone was detected or not.

Parameters

measures	Pointer to an array of lidar measurements.

Here is the caller graph for this function:



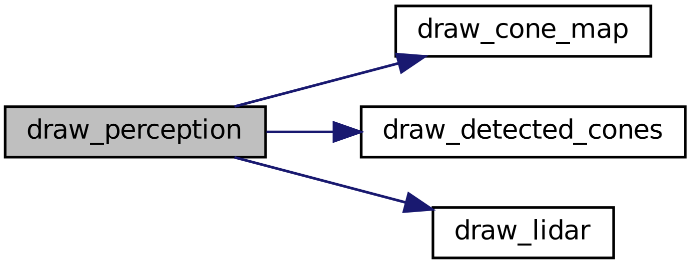
7.12.2.7 draw_perception()

```
void draw_perception ( )
```

Renders the complete perception layer.

Draws perception results.

Combines the lidar lines, detected cones, and cone map visualizations to create the perception overlay. Afterwards, the perception bitmap is blended onto the main display buffer offset in relation to the car's position. Here is the call graph for this function:



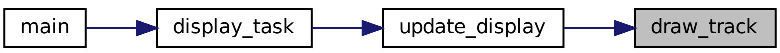
Here is the caller graph for this function:

7.12.2.8 draw_track()

```
void draw_track ( )
```

Draws the track.

Checks that the track bitmap has valid dimensions, and if so, draws the track image onto the display buffer. If the bitmap dimensions are invalid, an error message is shown. Here is the caller graph for this function:



7.12.2.9 draw_trajectory()

```
void draw_trajectory (
            waypoint * trajectory )
```

Draws the trajectory of the car.

Draws the planned trajectory.

Clears and sets the background for the trajectory bitmap and iterates over each trajectory waypoint, drawing each as a filled circle. In DEBUG mode, the waypoint index is also displayed and an extra line is drawn from the car sprite center.

Parameters

trajectory	Pointer to an array of waypoint structures representing the car's trajectory.

Here is the caller graph for this function:

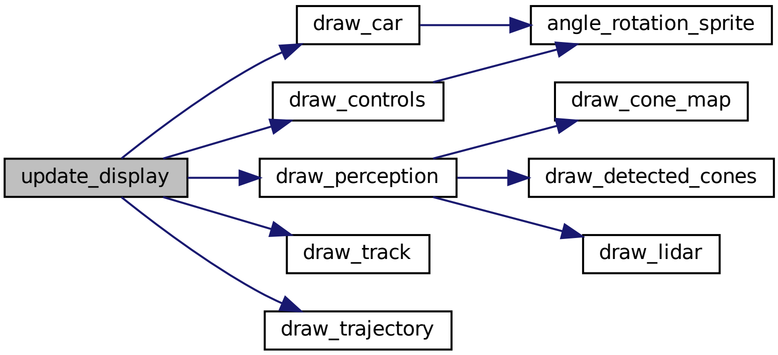
7.12.2.10 update_display()

```
void update_display ( )
```

Updates the entire display.

Updates the display.

Locks the drawing mutex and sequentially renders the background, track, car, perception, trajectory, title text, and controls on the display buffer. Once all elements are rendered, the display buffer is blitted to the screen, and the mutex is subsequently unlocked. Here is the call graph for this function:



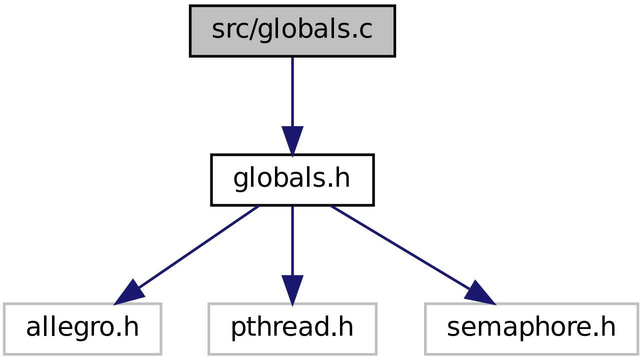
Here is the caller graph for this function:

7.13 src/globals.c File Reference

Global definitions and state variables for the 2D FSAE simulation.

```
#include "globals.h"
```

Include dependency graph for globals.c:



Variables

- const char ∗ title = "2D FSAE sim by rimaturus"
- const int X_MAX = (19 ∗ px_per_meter)
- const int Y_MAX = (12 ∗ px_per_meter)
- const float cone_radius = 0.05f
- float car_x = 4.5f
- float car_y = 3.0f
- int car_angle = 0
- BITMAP ∗ control_panel = NULL
- BITMAP ∗ steering_wheel = NULL
- BITMAP ∗ throttle_gauge = NULL
- BITMAP ∗ background = NULL
- BITMAP ∗ track = NULL
- BITMAP ∗ car = NULL

- BITMAP ∗ perception = NULL
- BITMAP ∗ trajectory_bmp = NULL
- BITMAP ∗ display_buffer = NULL
- int grass_green
- int asphalt_gray
- int white
- int pink
- int yellow
- int blue
- pointcloud_t measures [360]
- sem_t lidar_sem
- pthread_mutex_t draw_mutex = PTHREAD_MUTEX_INITIALIZER

7.13.1 Detailed Description

Global definitions and state variables for the 2D FSAE simulation.

This file declares and initializes global constants, variables, and resources used by the simulation.

- Simulation Information:
  -

7.13.2 Variable Documentation

7.13.2.1 asphalt_gray

```
int asphalt_gray
```

7.13.2.2 background

```
BITMAP* background = NULL
```

-

7.13.2.3 blue

```
int blue
```

7.13.2.4 car

```
BITMAP* car = NULL
```

-

7.13.2.5 car_angle

```
int car_angle = 0
```

The initial orientation angle of the car (in degrees).

- Graphical Resources:
  -

7.13.2.6 car_x

```
float car_x = 4.5f
```

The initial X-coordinate position of the car within the simulation.

-

7.13.2.7 car_y

```
float car_y = 3.0f
```

7.13.2.8 cone_radius

```
const float cone_radius = 0.05f
```

The radius (in meters) for cone-related calculations, set to 0.05.

- Car Initial State:
  -

7.13.2.9 control_panel

```
BITMAP* control_panel = NULL
```

-

7.13.2.10 display_buffer

```
BITMAP* display_buffer = NULL
```

Bitmap pointers used for rendering various components of the simulation such as the background, track markings, car image, perception overlays, trajectory visualizations, and the overall display buffer.

- Color Definitions:
  -

7.13.2.11 draw_mutex

```
pthread_mutex_t draw_mutex = PTHREAD_MUTEX_INITIALIZER
```

A POSIX mutex that ensures mutual exclusion during drawing operations.

Note

The multiplier 'px_per_meter' used for computing X_MAX and Y_MAX is defined elsewhere.

7.13.2.12 grass_green

```
int grass_green
```

7.13.2.13 lidar_sem

```
sem_t lidar_sem
```

A POSIX semaphore used to synchronize access to LiDAR data.

-

7.13.2.14 measures

```
pointcloud_t measures[360]
```

An array of 360 LiDAR measurements (one per degree) representing the simulated point cloud data.

- Thread Synchronization:
  -

7.13.2.15 perception

```
BITMAP* perception = NULL
```

-

7.13.2.16 pink

```
int pink
```

7.13.2.17 steering_wheel

```
BITMAP* steering_wheel = NULL
```

-

7.13.2.18 throttle_gauge

```
BITMAP* throttle_gauge = NULL
```

Bitmap pointers for UI elements including the control panel, steering wheel, and throttle gauge.

-

7.13.2.19 title

```
const char* title = "2D FSAE sim by rimaturus"
```

The simulation title ("2D FSAE sim by rimaturus").

- Simulation Dimensions:
  –

7.13.2.20 track

```
BITMAP* track = NULL
```

-

7.13.2.21 trajectory_bmp

```
BITMAP* trajectory_bmp = NULL
```

-

7.13.2.22 white

```
int white
```

7.13.2.23 X_MAX

```
const int X_MAX = (19 * px_per_meter)
```

The maximum horizontal dimension of the simulation in pixels, computed as 19 ∗ px_per_meter.

-

7.13.2.24 Y_MAX

```
const int Y_MAX = (12 * px_per_meter)
```

The maximum vertical dimension of the simulation in pixels, computed as 12 ∗ px_per_meter.

- Physics and Geometry:
  –

7.13.2.25 yellow

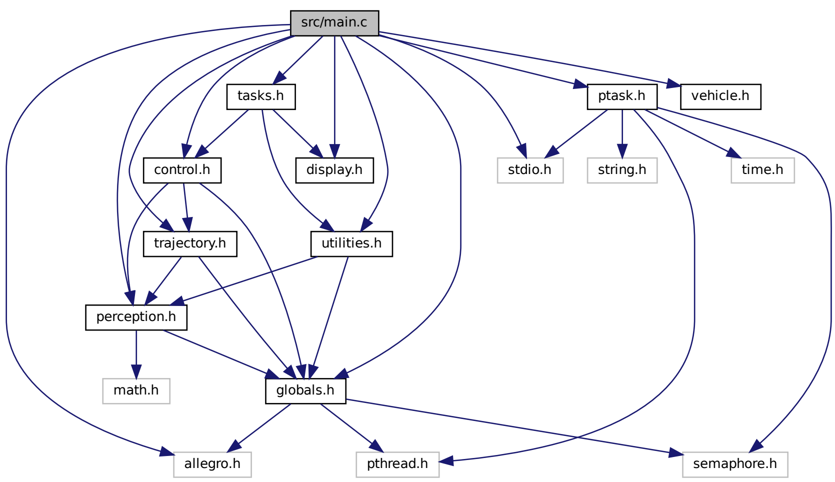
```
int yellow
```

7.14 src/main.c File Reference

Main implementation of the 2D FSAE Simulation.

```
#include <stdio.h>
#include <allegro.h>
#include "control.h"
#include "display.h"
#include "globals.h"
#include "perception.h"
#include "tasks.h"
#include "trajectory.h"
#include "utilities.h"
#include "vehicle.h"
#include "ptask.h"
```

Include dependency graph for main.c:



Functions

- void init_allegro ()
- void init_track ()
- void init_car ()
- void init_perception ()
- void init_visual_controls ()
- void init_bitmaps ()
- void update_screen ()
- int main (void)
- void init_trajectory ()

Variables

- const char filename [100] = "track/cones.yaml"

  Application entry point.
- int car_x_px
- int car_y_px
- int car_bitmap_x
- int car_bitmap_y

7.14.1 Detailed Description

Main implementation of the 2D FSAE Simulation.

This file initializes the Allegro graphics library, sets up various bitmaps for simulation elements, and creates periodic tasks for perception, trajectory, control, and display. It also handles input, such as waiting for the user to press ESC to exit the simulation.

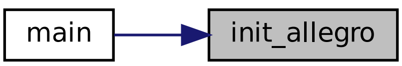
These routines configure the graphical environment, load assets (bitmaps), and initialize the simulation track including cones from an external YAML file. The program makes use of a periodic task system (using SCHED_OTHER) and employs mutexes to synchronize drawing operations.

7.14.2 Function Documentation

7.14.2.1 init_allegro()

```
void init_allegro ( )
```

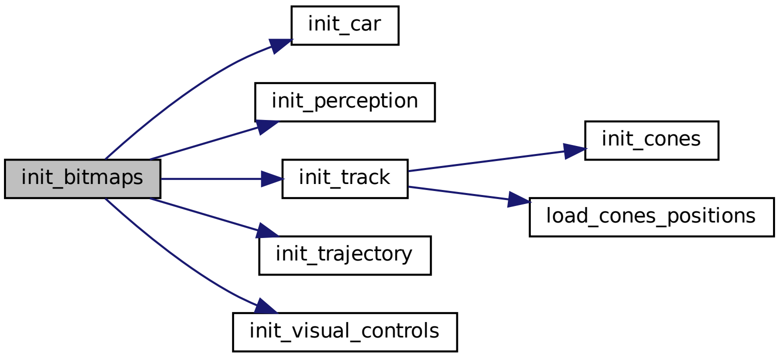
Here is the caller graph for this function:



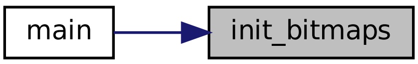
7.14.2.2 init_bitmaps()

```
void init_bitmaps ( )
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.14.2.3 init_car()

```
void init_car ( )
```

Here is the caller graph for this function:

7.14.2.4 init_perception()

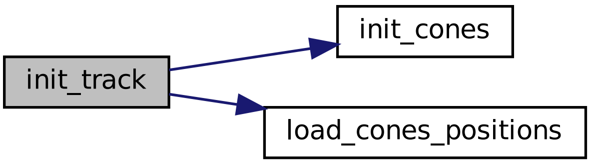
```
void init_perception ( )
```

Here is the caller graph for this function:

7.14.2.5 init_track()

```
void init_track ( )
```

Here is the call graph for this function:



Here is the caller graph for this function:

7.14.2.6 init_trajectory()

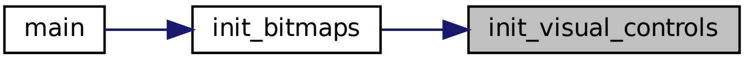
```
void init_trajectory ( )
```

Here is the caller graph for this function:

7.14.2.7 init_visual_controls()

```
void init_visual_controls ( )
```

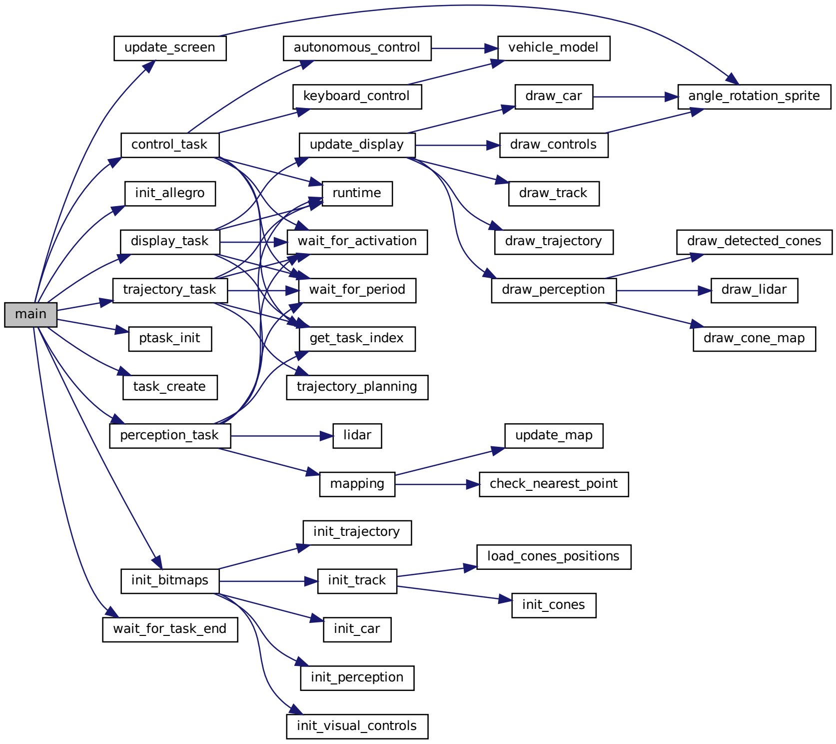
Here is the caller graph for this function:



7.14.2.8 main()

```
int main (
            void   )
```

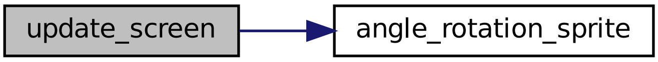
Here is the call graph for this function:



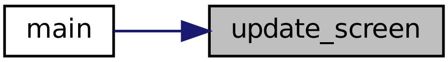
7.14.2.9 update_screen()

```
void update_screen ( )
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.14.3 Variable Documentation

7.14.3.1 car_bitmap_x

```
int car_bitmap_x
```

7.14.3.2 car_bitmap_y

```
int car_bitmap_y
```

7.14.3.3 car_x_px

```
int car_x_px
```

7.14.3.4 car_y_px

```
int car_y_px
```

7.14.3.5 filename

```
const char filename[100] = "track/cones.yaml"
```

Application entry point.

The main() function initializes the graphics system, creates display bitmaps, and sets up periodic tasks for different simulation modules:

- Perception: processes sensor data to detect environment features.
- Trajectory: calculates the planned path for the vehicle.
- Control: manages vehicle control based on perception and trajectory.
- Display: updates the visual representation of the simulation.

After task creation, the main loop waits for all tasks to terminate (typically on ESC key press), cleans up the graphical buffers, and exits the simulation.

See also

init_allegro(), init_bitmaps(), update_screen(), task_create(), wait_for_task_end()

Initializes the Allegro graphics system.

Sets up the graphics data structures, installs keyboard and mouse handlers, and configures the graphics mode. It defines the color depth and initializes several colors (e.g., grass green, asphalt gray, pink) used throughout the simulation. The display window is created with a title and switch mode, and finally the screen is cleared.

See also

allegro_init(), install_keyboard(), install_mouse(), set_gfx_mode(), set_window_title(), clear_to_color()

Initializes the track bitmap.

Creates a bitmap representing the track and clears it to an asphalt color. It loads cone positions from an external YAML configuration file and plots track cones on the track bitmap. Only valid cones (with a valid color) are rendered, using a radius scaled to pixels per meter.

See also

load_cones_positions(), init_cones(), clear_bitmap(), circlefill()

Initializes the car bitmap and positions.

Loads the car sprite bitmap from a file. It computes the car position in pixels, ensuring that the car's graphical representation is centered on its logical position by offsetting the bitmap's top-left coordinates.

See also

load_bitmap(), clear_bitmap(), circlefill()

Initializes the perception bitmap.

Creates a dedicated bitmap for representing the perceptual data (e.g., sensor outputs) of the simulation. The dimensions are based on the maximum range of the sensor (scaled by pixels per meter), ensuring that the bitmap covers the necessary area for sensor information. The bitmap is cleared and set to a transparent color.

See also

create_bitmap(), clear_bitmap(), clear_to_color()

Initializes the trajectory bitmap.

This function creates and configures the bitmap that visualizes the calculated trajectory path. The bitmap is cleared and its background is set transparent (using the designated pink color).

See also

create_bitmap(), clear_bitmap(), clear_to_color()

Initializes visual control elements.

Loads and initializes bitmaps for user visual controls such as the steering wheel (bitmap for steering) and a visual throttle gauge. In case of any loading error, the application exits with an error message.

See also

load_bitmap(), create_bitmap(), clear_bitmap(), clear_to_color()

Initializes all bitmaps used for displaying the simulation.

Creates the display buffer and background bitmaps, clearing them to the proper color. Then, it calls several initialization functions that prepare all simulation-specific bitmaps (track, car, perception, trajectory, and visual controls).

See also

create_bitmap(), init_track(), init_car(), init_perception(), init_trajectory(), init_visual_controls()

Updates the display screen.

Locks a mutex to prevent concurrent drawing modifications, clears the display buffer, and performs a series of drawing operations:

- Renders the steering wheel with rotation and scaling.
- Draws the background and track bitmaps.
- Rotates, scales, and draws the car sprite ensuring correct orientation.
- Draws the perception and trajectory bitmaps. Finally, the updated display buffer is blitted to the actual screen, and the mutex is unlocked.

See also

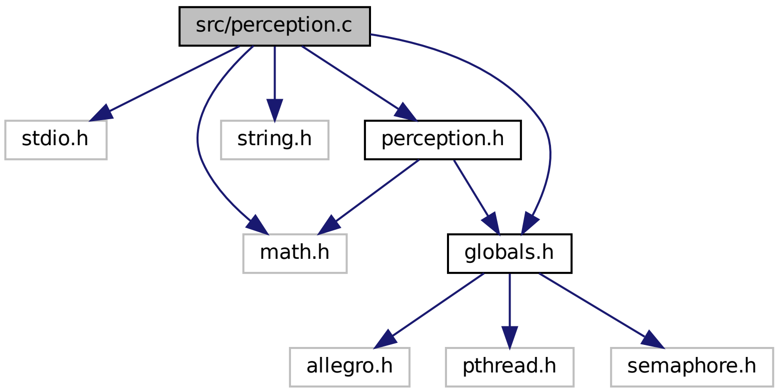
pthread_mutex_lock(), clear_to_color(), rotate_scaled_sprite(), draw_sprite(), blit(), pthread_mutex_unlock()

7.15 src/perception.c File Reference

Implements LiDAR-based cone detection and mapping using circle Hough transform.

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "globals.h"
#include "perception.h"
```

Include dependency graph for perception.c:



Classes

- struct Hough_circle_point_t

    Represents a point in the Hough circle transformation space.

Functions

- void check_nearest_point (int angle, float new_point_x, float new_point_y, int color, cone_border *cone_borders)
- void lidar (float car_x, float car_y, pointcloud_t *measures)
- void init_cone_borders (cone_border *cone_borders)
- void calculate_circle_points (float center_x, float center_y, int color, cone *circle_points)
- void find_closest_points (Hough_circle_point_t *circumference_points, float point_x, float point_y, Hough_circle_point_t *reference_points, int ref_size)
- void find_local_minima (Hough_circle_point_t *points, int *first_min, int *second_min)
- float * find_cone_center (Hough_circle_point_t *possible_centers, int center_count)
- void mapping (float car_x, float car_y, int car_angle, cone *detected_cones)
- void update_map (cone *detected_cones)

Variables

- const int sliding_window = 360
- const int angle_step = 1
- int start_angle = 0
- const float ignore_distance = 0.5f
- const float distance_resolution = 0.01f
- cone detected_cones [MAX_DETECTED_CONES]
- int n_candidates = 0
- candidate_cone candidates [MAX_CANDIDATES]
- int track_map_idx = 0
- cone track_map [MAX_CONES_MAP]

7.15.1 Detailed Description

Implements LiDAR-based cone detection and mapping using circle Hough transform.

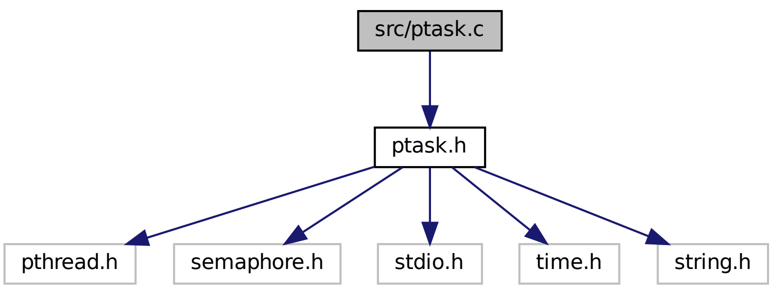
This file contains functions to process LiDAR measurements, cluster points belonging to the same cone, compute cone centers with a Hough circle parameterization, and update a global map of detected cones.

Global variables:

- sliding_window: Number of angles scanned (360 degrees).
- angle_step: Step in degrees for each LiDAR scan.
- start_angle: Starting angle of the LiDAR scan.
- ignore_distance: Minimum distance threshold for detection.
- distance_resolution: Increment step for distances along LiDAR rays.
- detected_cones: Array that holds final detected cone positions and colors.
- n_candidates: Number of candidate cones currently tracked.
- candidates: Array that holds candidate cone detections.
- track_map_idx: Index for the global map where detected cones are stored.
- track_map: Global map of cone positions and colors.

7.16 src/ptask.c File Reference

```
#include "ptask.h"
```
Include dependency graph for ptask.c:



Macros

- #define PTASK_IMPLEMENTATION

7.16.1 Macro Definition Documentation

7.16.1.1 PTASK_IMPLEMENTATION

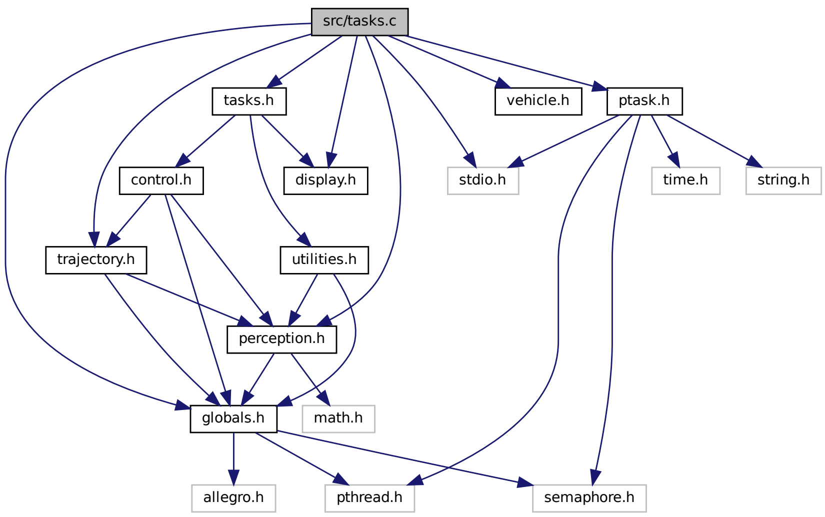
```
#define PTASK_IMPLEMENTATION
```

7.17 src/tasks.c File Reference

Contains periodic task functions for LiDAR perception, trajectory planning, vehicle control, and display update in a 2D simulation environment.

```
#include <stdio.h>
#include "globals.h"
#include "tasks.h"
#include "perception.h"
#include "trajectory.h"
#include "vehicle.h"
#include "display.h"
#include "ptask.h"
```

Include dependency graph for tasks.c:



Functions

- void * perception_task (void *arg)

    Periodic perception task.
- void * trajectory_task (void *arg)
- void * control_task (void *arg)
- void * display_task (void *arg)

7.17.1 Detailed Description

Contains periodic task functions for LiDAR perception, trajectory planning, vehicle control, and display update in a 2D simulation environment.

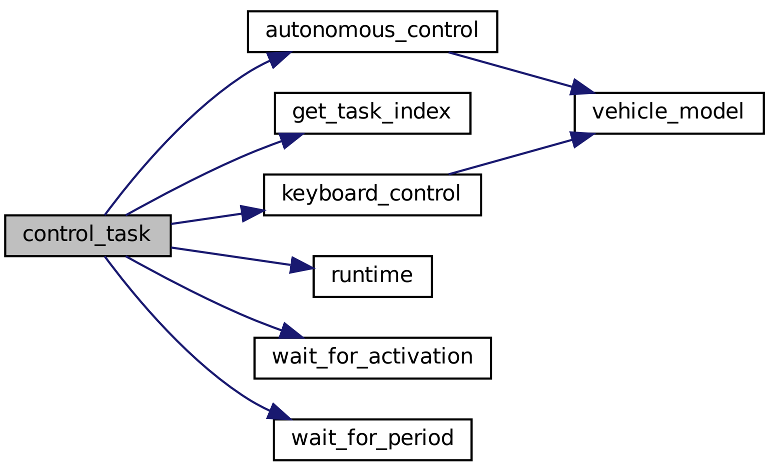
This file implements tasks that run periodically in separate threads. Each task is synchronized with others through semaphores and managed by a periodic task scheduler.

7.17.2 Function Documentation

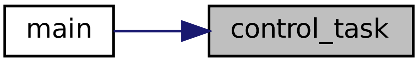
7.17.2.1 control_task()

```
void* control_task (
            void * arg )
```

Here is the call graph for this function:



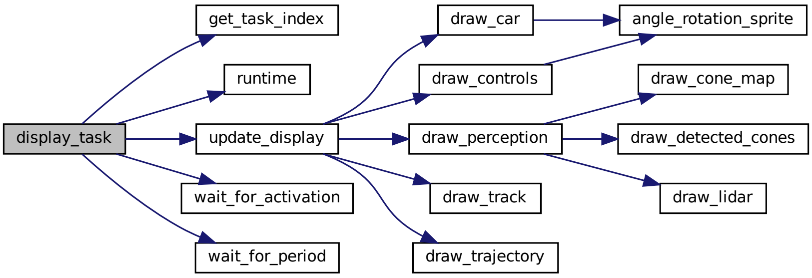
Here is the caller graph for this function:



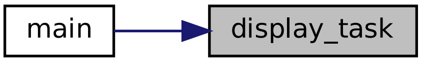
7.17.2.2 display_task()

```
void* display_task (
            void * arg )
```

Here is the call graph for this function:



Here is the caller graph for this function:



7.17.2.3 perception_task()

```
void* perception_task (
            void * arg )
```

Periodic perception task.

Perception task.

This task acquires LiDAR measurements, resets the array of detected cones, and performs mapping based on the current position and orientation of the vehicle. After processing the sensor data, it signals the trajectory planning task using a semaphore. The task executes continuously until the ESC key is pressed.

Parameters

in	*arg*	Pointer used to determine the task's index.

Returns

Always returns NULL.

Note

The function uses a runtime measurement call to monitor its execution time.

Periodic trajectory planning task.

This task waits for a signal from the perception task to ensure updated sensor data is available. It then calculates a new trajectory for the vehicle based on its current position, orientation, and the detected cones from the environment. The task runs periodically and terminates when the ESC key is pressed.

Parameters

in	*arg*	Pointer used to determine the task's index.

Returns

Always returns NULL.

Note

The task's operation is encapsulated between runtime measurement calls.

Periodic control task.

This task is responsible for controlling the vehicle's movement. It chooses between keyboard-based control and autonomous control based on the state of the 'A' key. If autonomous mode is active (i.e., 'A' is pressed), it uses the computed trajectory, otherwise it relies on user input to drive the vehicle. Execution continues until the ESC key is pressed.

Parameters

in	*arg*	Pointer used to determine the task's index.

Returns

Always returns NULL.

Note

The function uses runtime measurement calls to monitor execution time per cycle.

Periodic display task.

This task updates the graphical display of the simulation environment. It refreshes the on-screen information such as the vehicle's position, sensor data, and other simulation elements. The task continues running until the ESC key is pressed.

Parameters

in	*arg*	Pointer used to determine the task's index.

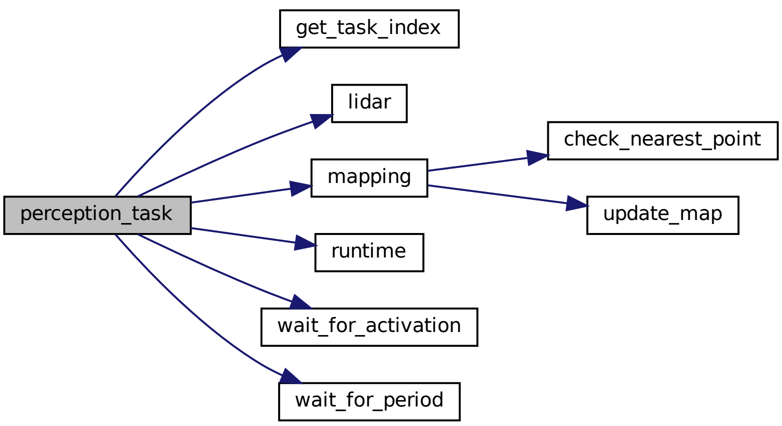
Returns

Always returns NULL.

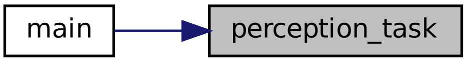
Note

Like the other tasks, execution time is tracked using runtime measurement calls.

Here is the call graph for this function:



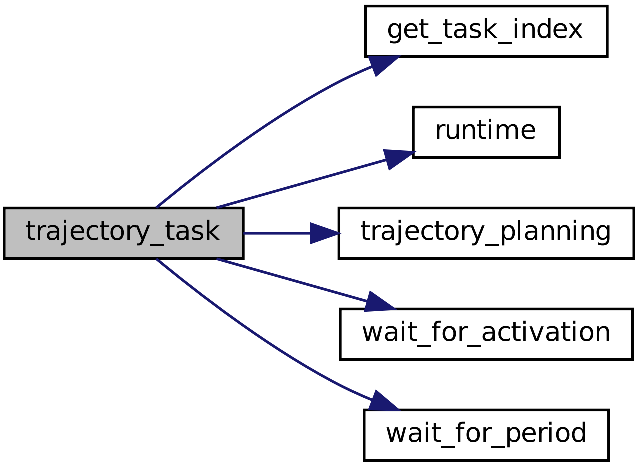
Here is the caller graph for this function:



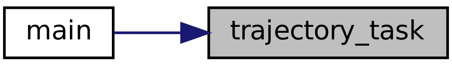
7.17.2.4 trajectory_task()

```
void* trajectory_task (
            void * arg )
```

Here is the call graph for this function:



Here is the caller graph for this function:

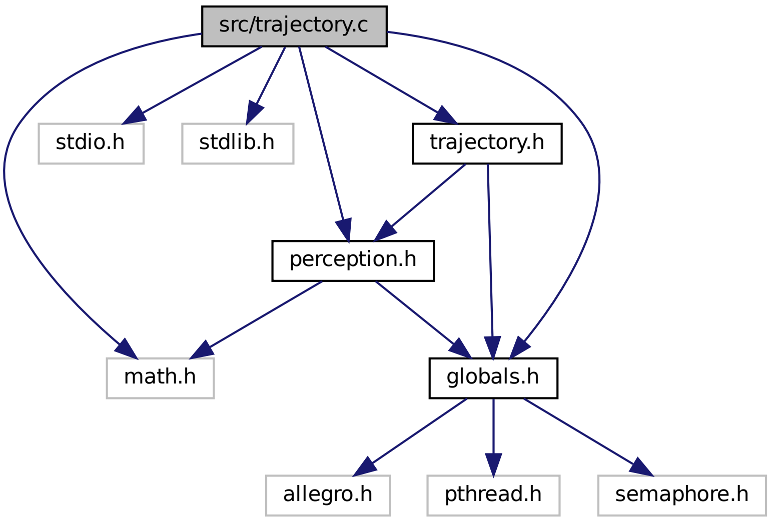


7.18 src/trajectory.c File Reference

Implements trajectory planning based on detected cones.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "trajectory.h"
#include "globals.h"
#include "perception.h"
```

Include dependency graph for trajectory.c:



Functions

- void trajectory_planning (float car_x, float car_y, float car_angle, cone ∗detected_cones, waypoint ∗trajectory)

Variables

- int trajectory_idx = 0

    Global index used to track the number of waypoints currently stored in the trajectory.
- waypoint trajectory [2 ∗MAX_DETECTED_CONES]

    Global array of waypoints forming the planned trajectory.

7.18.1 Detailed Description

Implements trajectory planning based on detected cones.

This file contains the functionality to plan a driving trajectory using cone detection data. The trajectory is generated by connecting a global track map of cones (both blue and yellow) based on their proximity, and then finding midpoints between connected cones to serve as trajectory waypoints. Optionally, in debug builds, the cone connections are drawn for visualization.

Global Variables:

- trajectory_idx: An integer index representing the number of valid points in the generated trajectory.
- trajectory: An array of waypoints that stores the planned trajectory.

The trajectory planning process includes:

- Initializing trajectory points with invalid default values.
- Validating if there are enough cones in the track map to perform planning.
- Computing nearest neighbors for each cone (one for each color) for connection building.
- Drawing connection lines between cones when debugging is enabled.
- Generating trajectory waypoints by calculating the midpoint between connected cones.
- Reordering trajectory points based on proximity to ensure a smooth and sequential path.

Parameters

car_x	The current x-coordinate of the car.
car_y	The current y-coordinate of the car.
car_angle	The current orientation angle of the car in radians.
detected_cones	Pointer to an array of cones detected by the car's perception system.
trajectory	Pointer to an array of waypoints where the computed trajectory will be stored.

Note

- The function assumes that there exists a correctly maintained global track map (track_map) and its size (track_map_idx).
- Debug drawing of cone connections is enclosed in an #ifdef DEBUG block.
- The maximum number of trajectory points is defined by MAX_DETECTED_CONES.

Warning

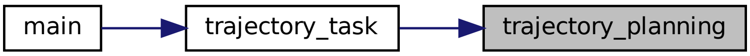
Ensure proper initialization of global variables and suitable allocation of the trajectory array before invoking this function.

7.18.2 Function Documentation

7.18.2.1 trajectory_planning()

```
void trajectory_planning (
            float car_x,
            float car_y,
            float car_angle,
            cone * detected_cones,
            waypoint * trajectory )
```

Here is the caller graph for this function:



7.18.3 Variable Documentation

7.18.3.1 trajectory

```
waypoint trajectory[2 *MAX_DETECTED_CONES]
```

Global array of waypoints forming the planned trajectory.

This array stores the computed trajectory based on the detected cones. It is sized to accommodate up to twice the number of maximum detectable cones (2 ∗ MAX_DETECTED_CONES).

7.18.3.2 trajectory_idx

```
int trajectory_idx = 0
```

Global index used to track the number of waypoints currently stored in the trajectory.

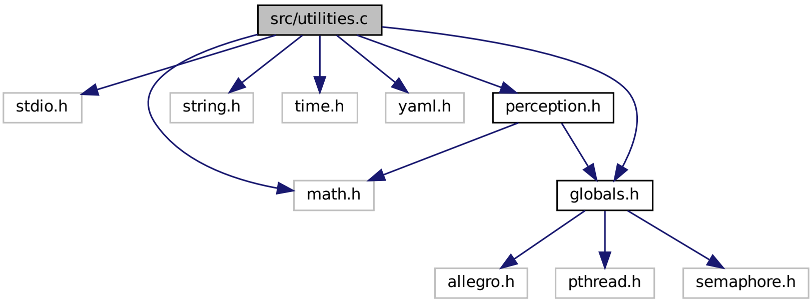
This variable indicates the current position in the trajectory array, facilitating the addition of new waypoints.

7.19 src/utilities.c File Reference

Utility functions for cone initialization, YAML cone loading, angle rotation conversion, and runtime performance measurement.

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <yaml.h>
#include "globals.h"
```

`#include "perception.h"`
Include dependency graph for utilities.c:



Functions

- void init_cones (cone *cones)

  Initializes an array of cone structures.
- void load_cones_positions (const char *filename, cone *cones, int max_cones)
- float angle_rotation_sprite (float angle)
- void runtime (int stop_signal, char *task_name)

Variables

- float tmp_scale = 1.5 / 100

7.19.1 Detailed Description

Utility functions for cone initialization, YAML cone loading, angle rotation conversion, and runtime performance measurement.

This file provides utility functions that include:

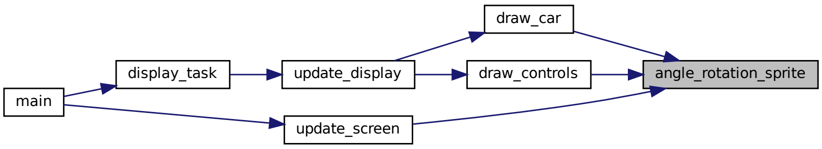
- Initializing an array of cone structures.
- Parsing a YAML file to load cone positions and colors.
- Converting an angle into a sprite rotation value.
- Timing code performance using the POSIX clock.

7.19.2 Function Documentation

7.19.2.1 angle_rotation_sprite()

```
float angle_rotation_sprite (
            float angle )
```

Here is the caller graph for this function:



7.19.2.2 init_cones()

```
void init_cones (
            cone * cones )
```

Initializes an array of cone structures.

This function initializes each cone in the provided array by setting its x and y coordinates to 0.0f and its color to -1.

Parameters

cones	Pointer to the array of cone structures to be initialized.

Loads cone positions and colors from a YAML file.

This function parses the specified YAML file to extract cone position (x and y) and color data. The YAML file is expected to have a top-level key "cones" containing a sequence of cone mappings. Each cone mapping should provide:

- "x": A string representing the x coordinate (converted to float and scaled).
- "y": A string representing the y coordinate (converted to float and scaled).
- "color": A string representing the cone color (e.g., "yellow" or "blue").

The function prints out diagnostic messages to the standard output regarding the file being loaded, and reports any errors encountered during file opening or YAML parsing.

Parameters

filename	Path to the YAML file containing cone data.
cones	Pointer to the array of cone structures to be populated.
max_cones	Maximum number of cones to load into the array.

Converts an angle (in degrees) to a sprite rotation value.

This helper function transforms an input angle (in degrees) into a corresponding sprite rotation value. The conversion is based on the formula: sprite_rotation = 64.0f - (128.0f ∗ angle / 180.0f).

Parameters

angle	Angle in degrees.

Returns

Sprite rotation value as a floating-point number.

Measures code performance based on a runtime signal.

This function is a helper to measure code execution performance if the PROFILING macro is defined. When called with a start signal (stop_signal = 0), it records the current time and prints a "START" message. When called with a stop signal (stop_signal != 0), it calculates the difference from the start time, prints an "END" message, and optionally prints the elapsed runtime in microseconds.

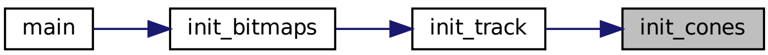
Note

This function depends on the POSIX clock_gettime() function and is active only if PROFILING is defined.

Parameters

stop_signal	A flag indicating whether to start (0) or stop (non-zero) the runtime measurement.
task_name	A string representing the name of the task for which the runtime is being measured.

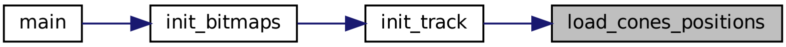
Here is the caller graph for this function:



7.19.2.3 load_cones_positions()

```
void load_cones_positions (
            const char * filename,
            cone * cones,
            int max_cones )
```

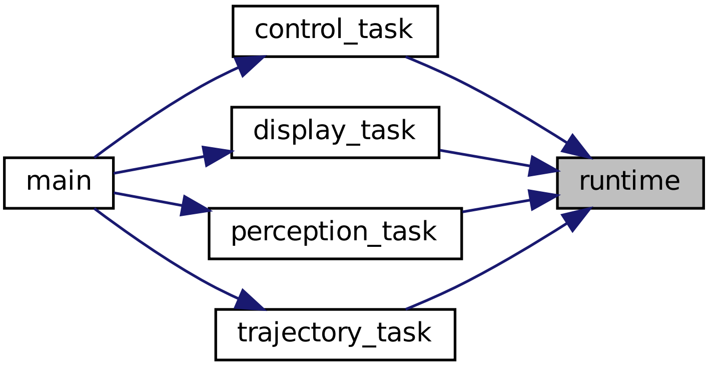
Here is the caller graph for this function:



7.19.2.4 runtime()

```
void runtime (
            int stop_signal,
            char * task_name )
```

Here is the caller graph for this function:



7.19.3 Variable Documentation

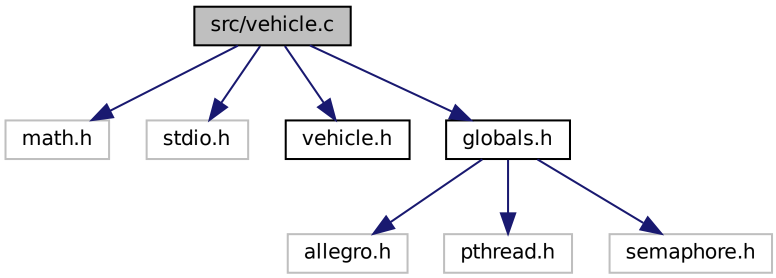
7.19.3.1 tmp_scale

```
float tmp_scale = 1.5 / 100
```

7.20 src/vehicle.c File Reference

```
#include <math.h>
#include <stdio.h>
#include "vehicle.h"
#include "globals.h"
```

Include dependency graph for vehicle.c:



Functions

- void vehicle_model (float *car_x, float *car_y, int *car_angle, float pedal, float steering)

  Updates the vehicle's position and orientation based on pedal and steering inputs.