Saphira Reference Manual

8.3.2

Generated by Doxygen 1.2.10

Sun Sep 7 18:05:49 2003

Contents

Chapter 1

Saphira Module Index

1.1 Saphira Modules

Here is a list of all modules:

Chapter 2

Saphira Hierarchical Index

2.1 Saphira Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Saphira Compound Index

3.1 Saphira Compound List

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

Chapter 4

Saphira Module Documentation

4.1 Gradient Navigation Module

Functions

- void **sfGradInit** (void)
  Initializes the gradient.
- void **sfGradInitRes** (int res, int turnRadius)
  Initializes the gradient,.
- void **sfGradSetMap** (void *p)
  Sets the map with an mcObject.
- void **sfGradUseArtifacts** (int useArtifacts)
  Whether the gradient module should use artifacts (.wld files).
- void **sfGradUseMap** (int useMap)
  Whether the gradient module should use grid maps (.map files).
- void **sfGradUseSonar** (int useSonar)
  Whether the gradient module should use sonar data.
- void **sfGradUseLaser** (int useLaser)
  Whether the gradient module should use laser data.

- void **sfGradSetGoal** (float x, float y)

  Sets the Gradient goal to this global point (in mm). The goal can be changed at any time.

- int **sfGradSetGoalByName** (char *goalName)

  Sets the Gradient goal to the position of this goal (looks it up from map data by name). The goal can be changed at any time. Returns 1 if goal was there and 0 if it wasn't.

- void **sfGradSetSpeed** (int high, int mid, int back)

  Sets the speed the robot travels at.

- void **sfGradDoGoal** (int which)

  Whether the robot should be going to a goal or not.

- void **sfGradStop** (void)

  Stops the robot going to goal and stops the robot.

- int **sfGradStatus** (void)

  Gets the status of the gradient module.

- int **sfGradIsActive** (void)

  Gets if the gradient module is active or not.

- void **sfGradObsParams** (int keepout, int decay)

  Resets keep-out distance and decay distance; in mm.

- void **sfGradSetDone** (int close, int done)

  Set how close we need to be to a goal to slow done or be done.

- void **sfGradSetSonarBuffer** (int which)

  Tell the gradient which sonar buffer to use 1 current 2 cumulative 3 both (2 is the default).

- void **sfGradSetLaserBuffer** (int which)

  Tell the gradient which laser buffer to use 1 current 2 cumulative 3 both (3 is the default).

- int **sfGradGetSonarBuffer** (void)

  Gets which sonar buffer the gradient is using.

- int **sfGradGetLaserBuffer** (void)

  Gets which laser buffer the gradient is using.

- int **sfGradGetCanBack** (void)

  Gets if the grad action can back up, 0 = no, 1 = when appropriate, 2 = always.

- void **sfGradSetCanBack** (int canBack)

  Gets if the grad action can back up, 0 = no, 1 = when appropriate, 2 = always.

- void **sfGradSetTurnRadius** (int turnRadius)

  Sets the turn radius needed to let it turn instead of back, make it back with sfGradSetCanBack.

- int **sfGradGetTurnRadius** (void)

  Gets the turn radius needed to let it turn instead of back, make sure it is allowed to back up with with sfGradGetCanBack.

- void **sfGradSetAcc** (int acc)

  Sets the acceleration used in driving the robot.

- void **sfGradSetPnum** (int n)

  Set number of propagations, mostly for show.

- void **sfGradSetMax** (int width, int height)

  Sets the size of the gradient window.

- void **sfGradUseFinalApproach** (int which)

  Use the final approach action or not.

- void **sfGradSetFinalApproachSpeed** (int speed)

  Sets the final approach speed.

- int **sfGradGetFinalApproachSpeed** (void)

  Gets the final approach speed.

- void **sfGradSetLocalPath** (int dist)

  Sets the global path distance in mm (> 0 turns it on).

- int **sfGradGetLocalPath** ()

  Gets the global path distance in mm.

- void **sfGradSetMaxLaserDist** (int dist)

  Sets the maximum laser distance in mm (> 0 turns it on).

- int **sfGradGetMaxLaserDist** ()

  Gets the maximum laser distance in mm that will be used for path planning.

- void **sfGradSetMaxSonarDist** (int dist)

  Sets the maximum sonar distance in mm (> 0 turns it on).

- int **sfGradGetMaxSonarDist** ()

  Gets the maximum sonar distance in mm that will be used for path planning.

- int **sfGradLoadMap** (char *name)

  Loads a map for gradient, should be the same as localization (0 failure, 1 success).

4.1.1 Detailed Description

For efficient movement based on local obstacles and world maps, Saphira has a realtime path planner based on the gradient method [Konolige, IROS 2000]. For planning paths and moving in a world map, the gradient module is typically used with Markov Localization to keep the robot registered with a map as it moves (sfLoc, sfLocLaser, and sfLocFl libraries).

Gradient Path Planning is a process for determining optimal paths for the robot, in real time. These paths can take into account both local obstacles, sensed by sonars and/or laser range-finder devices; and global map information such as the location of walls and other structural obstacles. At each sync cycle (100 ms), the Gradient module calculates the lowest-cost path from a goal point or set of goal points to the robot. The algorithm starts by considering a local neighborhood connecting the robot and the goal or goals, and then expands its search if no path is found. There is a user-settable limit on the size of the neighborhood considered.

Gradient uses a square-cell grid as a cost field for determining good paths. You can set the grid resolution; a typical resolution is 10 mm. The maximum size of the grid can also be set.

Costs are calculated from a set of obstacles, obtained from pre-existing maps and from local sensor readings. Here are the obstacles sources:

1. Artifacts in a world map. Load a world map, and call sfGradUse-Artifacts(true).

2. A grid map created from the laser navigation software. A grid map is typically loaded into the localization system using **mcLoadScanMap**() (p. **??**). To access this map from the gradient module, use sfGradSetMap(**mcGetObject**() (p. **??**)). Finally, turn on grid map use by calling sfGradUseMap(true).

Grid maps and world maps may both be used at the same time. In this case, usually the grid map will contain the basic geometric information about the world, while the artifacts are special areas for the robot, e.g., keep-out areas.

3. Laser and sonar readings. These can be turned on and off with **sfGradUseLaser**() (p. 15) and **sfGradUseSonar**() (p. 10).

The cost field radiates outward from obstacles, to create a safety cushion for the robot. You can adjust this cushion using **sfGradObsParams**() (p. **??**).

There is a local controller, implemented as an action (SfGradAction), that drives the robot along the gradient path. This action controls the speed of the robot in a parameterized fashion, and can be set with sfGradSetSpeed. A further refinement for rectangular robots is the ability to back up when appropriate; this behavior is controlled by sfGradSetCanBack and sfGradSetTurnRadius. If the robot needs to turn more than 45 degrees to follow the path, it will see if its turn radius is clear. If it isn't, it will try to back up if possible, and turn around when it is clear of obstacles.

Some relevant sample Colbert load files: flgrad.act - basic gradient, without localization for localization with respect to a world, look at flloc.act scan.act - localization and gradient using a grid map (from the Laser Localization/Navigation module).

You can also use the gradient for a final approach to a given position, to do this call sfGradUseFinalApproach... you will also want to adjust the gradient obstacle parameters with sfGradObsParams and the done distance with sfGradSetDone (the close dist is ignored for the final approach). You may want to set the speed with sfGradSetFinalApproachSpeed.

Another refinement on gradient is to use a global path. In this approach, a first global path is calculated to the goal, and then it is fixed. The robot calculates a local gradient to a goal point at some distance ahead on the path. Compute time is saved, because only the local path is updated on each cycle.

Set the global path with the boolean **sfGradSetLocalPath**(int dist) (p. 14). When a new goal is given, the global path is calculated and stored, then used as above. dist is the distance to look ahead on the global path; if 0, no global path is used.

4.1.2 Function Documentation

4.1.2.1 int sfGradGetCanBack (void)

Gets if the grad action can back up, 0 = no, 1 = when appropriate, 2 = always.

Returns:

0 if the gradient action will never back up, 1 if it will back up when appropriate (see sfGradInitRes), and 2 it will back up all the time

4.1.2.2 int sfGradGetLaserBuffer (void)

Gets which laser buffer the gradient is using.

Returns:
1 using current buffer, 2 using cumulative buffer, 3 using both

4.1.2.3 int sfGradGetLocalPath ()

Gets the global path distance in mm.

Sets the local path distance for the gradient module. In local movement, the global path is computed and saved for a goal. A local path is computed to the goal path at every iteration.

Returns:
the distance to look ahead on the global path (mm) Set to 0 to turn off the local path.

4.1.2.4 int sfGradGetSonarBuffer (void)

Gets which sonar buffer the gradient is using.

Returns:
1 using current buffer, 2 using cumulative buffer, 3 using both

4.1.2.5 int sfGradGetTurnRadius (void)

Gets the turn radius needed to let it turn instead of back, make sure it is allowed to back up with with sfGradGetCanBack.

Returns:
the radius that must be clear for the robot to turn more than 45 degrees

4.1.2.6 void sfGradInit (void)

Initializes the gradient.

Initializes the Gradient module. Should be called right after loading the Gradient library.

4.1.2.7 void sfGradInitRes (int *res*, int *turnRadius*)

Initializes the gradient,.

Parameters:

res res in mm for grid cell resolution

turnRadius if turnRadius is 0 then it will use the normal parameters and action for movement, buf it turnRadius is not 0 it will back up if the robot needs to travel backwards and a circle with radius turnRadius is obstructed

4.1.2.8 int sfGradLoadMap (char * *name*)

Loads a map for gradient, should be the same as localization (0 failure, 1 success).

Load a scan map for gradient from a file. Should be the same map as that used for localization, but the resolution set by gradient can be different.

4.1.2.9 void sfGradObsParams (int *keepout*, int *decay*)

Resets keep-out distance and decay distance; in mm.

This sets the distance away from obstacles the gradient algorithm will stay. (These set fcost and fdecay on my_grad, see the header file for SfGrad.h and class SfGradGrid for the information about this in C++).

Parameters:

keepout (mm) the distance from the obstacles not to never drive within

decay (mm) the distance from obstacles to avoid if possible

4.1.2.10 void sfGradSetCanBack (int *canBack*)

Gets if the grad action can back up, 0 = no, 1 = when appropriate, 2 = always.

Parameters:

canBack 0 if the gradient action will never back up, 1 if it will back up when appropriate (see sfGradInitRes), and 2 it will back up all the time

4.1.2.11 void sfGradSetDone (int *close*, int *done*)

Set how close we need to be to a goal to slow done or be done.

Parameters:

close (mm) the distance away the robot is from the goal when it switches to close mode

done (mm) the distance away the robot is from the goal when the gradient decides its done

4.1.2.12 void sfGradSetLaserBuffer (int *which*)

Tell the gradient which laser buffer to use 1 current 2 cumulative 3 both (3 is the default).

Parameters:

which 1 use current buffer, 2 use cumulative buffer, 3 use both

4.1.2.13 void sfGradSetLocalPath (int *dist*)

Sets the global path distance in mm (> 0 turns it on).

Sets the local path distance for the gradient module. In local movement, the global path is computed and saved for a goal. A local path is computed to the goal path at every iteration.

Parameters:

dist the distance to look ahead on the global path (mm) Set to 0 to turn off the local path.

4.1.2.14 void sfGradSetMap (void * *p*)

Sets the map with an mcObject.

Sets the grid map that the gradient will use. Typically the map is taken from the localization module using a call to **mcGetObject**() (p. **??**), so that the gradient will use the same map that localization is. NOTE: The resolution of the map must be the same as that of the gradient routines (sfGradInit call).

4.1.2.15 void sfGradSetMax (int *width*, int *height*)

Sets the size of the gradient window.

Sets the maximum size (in mm) of the neighborhood considered by the Gradient module. The neighborhood will expand until it reaches this size, in searching for a valid path.

Parameters:

width maximum width in mm of neighborhood

height maximum height in mm of neighborhood

4.1.2.16 void sfGradSetMaxLaserDist (int *dist*)

Sets the maximum laser distance in mm (> 0 turns it on).

if this value is less than 0 it turns the maximum distance off, if this value is greater than 0 and a laser reading is further from the robot than this distance the laser reading is not added to the grid for use in path planning

4.1.2.17 void sfGradSetMaxSonarDist (int *dist*)

Sets the maximum sonar distance in mm (> 0 turns it on).

if this value is less than 0 it turns the maximum distance off, if this value is greater than 0 and a sonar reading is further from the robot than this distance the sonar reading is not added to the grid for use in path planning

4.1.2.18 void sfGradSetSonarBuffer (int *which*)

Tell the gradient which sonar buffer to use 1 current 2 cumulative 3 both (2 is the default).

Parameters:

which 1 use current buffer, 2 use cumulative buffer, 3 use both

4.1.2.19 void sfGradSetSpeed (int *high*, int *mid*, int *back*)

Sets the speed the robot travels at.

Sets the speed the gradient module will use for different circumstances.

Parameters:

high the speed to travel at when there are no obstructions (mm/sec)

mid the speed to travel at when there is congestion (mm/sec)

back the maximum speed for backwards travel (see sfGradInitRes)... it will go slower than this if needed and will use the same parameters high and mid above, but simply cap the backwards velocity at the one given here

4.1.2.20 void sfGradSetTurnRadius (int *turnRadius*)

Sets the turn radius needed to let it turn instead of back, make it back with sfGradSetCanBack.

Parameters:

turnRadius the radius that must be clear for the robot to turn more than 45 degrees

4.1.2.21 int sfGradStatus (void)

Gets the status of the gradient module.

Returns:

0 is idle, 1 is active, 2 is done, 3 is failed, 4 is searching

4.2 Module

The irrf module is seperate from the normal Saphira distribution. It comes with either a Irrf or a Irrf Integration kit from ActivMedia Robotics. For information contact `sales@activmedia.com.`

If you have purchased either of these you should be able to download new versions of this module from `http://robots.activmedia.com/sicklrf`

Do make sure your Saphira/versionIrrf file is the same version as your Saphira/version.txt file. There is dependency checking in Linux already, but not yet in the Windows installer. Its critical you're using the same versions for Saphira and its modules.

This module allows Saphira to use the ArIrrfDevice class in ARIA, for both display and the gradient module. The normal localization does not use the irrf as its an entirely different process, check out the irrf localization/navigation module. The display of the irrf is in green dots scattered on the screen where the irrf readings are. Note that by default Saphira readings close to each other are filtered out.

Even if you do not have this module you can use the ArIrrfDevice class in Saphira just like you use any of the other ARIA code (ie just compile it).

4.3 Laser Module

Functions

- void **sfStartLaser** (char *port)

  Connects the laser (NULL or "" port means simulator).

- void **sfStartLaserTcp** (char *host, int port)

  Connects the laser on a tcp port (NOT for the simulator).

- void **sfStopLaser** ()

  Disconnects the laser.

4.3.1 Detailed Description

The laser module is seperate from the normal Saphira distribution. It comes with either a Laser or a Laser Integration kit from ActivMedia Robotics. For information contact `sales@activmedia.com`.

If you have purchased either of these you should be able to download new versions of this module from `http://robots.activmedia.com/Laser`

Do make sure your Saphira/versionLaser file is the same version as your Saphira/version.txt file. There is dependency checking in Linux already, but not yet in the Windows installer. Its critical you're using the same versions for Saphira and its modules.

This module allows Saphira to use the ArSick (SICK Laser) class in ARIA, for both display and the gradient module. The normal localization does not use the laser as its an entirely different process, check out the laser localization/navigation module. The display of the laser is in green dots scattered on the screen where the laser readings are. Note that by default Saphira readings close to each other are filtered out.

Even if you do not have this module you can use the ArSick class in Saphira just like you use any of the other ARIA code (ie just compile it).

4.4 Localization Module

Functions

- void **mcSonarInit** (void)

  Initializes the localization.

- void **mcSonarInitRes** (int res)

  Initiailizes the localization using a particular grid size.

- void * **mcGetObject** (void)

  Gets an mc object (mostly for use with sfGradSetMap).

- void **mcSetMove** (int da, int ds, int tm)

  Sets the delta angle (da) delta distance (ds) or delta time (tm) before loc fires again.

- void **mcUpdateRobotPose** (int on)

  call this with true (non-zero) to make the localization update the robot position.

- void **mcPrintDuringUpdates** (int on)

  If true this will print during localization updates, false it won't.

- void **mcSetGain** (int pct)

  Sets the gain of the sensor information in the update step.

- void **mcSetGauss** (float dx, float dth)

  Centers the sample distribution on the center of the robot.

- void **mcSetNumSamples** (int n)

  Sets the number of samples.

4.4.1 Detailed Description

This module is used localization. By itself it can only be used with the sonar to localize in a vector (line) map. With the Laser Localization/Navigation module it can be used with the laser in a grid map generated by a robot.

You can look at/load flloc.act to see how to use this module.

4.4.2 Function Documentation

4.4.2.1 void mcSetGain (int *pct*)

Sets the gain of the sensor information in the update step.

Parameters:

pct sets the gain of the sensor information in the update step to the percent pct. If pct is 0, no sensor information is used. Reasonable values range from 10 to 50 percent, depending on the environment, the application, and the sensors.

4.4.2.2 void mcSetGauss (float *dx*, float *dth*)

Centers the sample distribution on the center of the robot.

This centers the sample distribution onto the center of where the robot is. You should probably change the robot position with sfJumpRobotAbs and then call this.

Parameters:

dx the length of the size of the square to put the samples in

dth the difference in angle to put the samples within

4.4.2.3 void mcSetMove (int *da*, int *ds*, int *tm*)

Sets the delta angle (da) delta distance (ds) or delta time (tm) before loc fires again.

Parameters:

da the amount turned before relocalizing (degrees)

ds the distance moved before relocalizing (mm)

tm the number of cycles to go after the robot stops before relocalizing (it won't localize after the robot stops if this parameter is 0) (cycles)

4.4.2.4 void mcSetNumSamples (int *n*)

Sets the number of samples.

All sample poses are reset to zero, and mcSetGauss may be called to re-center the sample set on the robot.

Parameters:

n number of samples to use

4.5 Laser Navigation And Localization

Functions

- void **mcLrfInit** ()

  Initializes the localization.

- void **mcLrfInitRes** (int res)

  Initializes the localization with a particular size of grid.

- void **mcLrfScanInit** ()

  Adds in the cumulative LRF buffer to map.

- int **mcLoadScanMap** (char *name)

  Loads a scan map (returns 0 if failed to load, 1 if loaded).

4.5.1 Detailed Description

The laser Navigation and Localization module is seperate from the normal Saphira distribution. It comes with the complete Laser Mapping and Navigation package ActivMedia Robotics. For information contact `sales@activmedia.com`.

If you have purchased this module you should be able to download new versions of this module from `http://robots.activmedia.com/LaserNavigation`

Do make sure your Saphira/versionNavigation file is the same version as your Saphira/version.txt file. There is dependency checking in Linux already, but not yet in the Windows installer. Its critical you're using the same versions for Saphira and its modules.

This module allows Saphira to use the the laser for Localization.

In addition to the documentation here, look at scan.act to see how to use these.

Chapter 5

Saphira Class Documentation

5.1 Sf Class Reference

Saphira system definition.

`#include <SfSystem.h>`

Static Public Methods

- void **init** ()

  initializes Saphira system, including Aria; called by the standard Saphira client on startup.

- **SfRobot** * **robot** ()

  Current robot object; Saphira 8.0 has only one User programs can access this with the SfROBOT macro.

- **SfSonarDevice** * **sonar** ()

  sonar device object.

- **SfLaserDevice** * **laser** ()

  laser device object.

- **SfLaserDevice** * **laser2** ()

  laser device object.

- **SfIrrfDevice** * **irrf** ()
  IR rangefinder device object.
- double **getX** ()
  Gets the robot current X value, in mm.
- double **getY** ()
  Gets the robot current Y value, in mm.
- double **getTh** ()
  Gets the robot current heading, in degrees.
- ArPose **getRwPose** ()
  Gets the robot current pose object.

Static Public Attributes

- **SfArtifactList** * **artList**
  artifact list.
- SfColbertStream * **ourColbert**
  main Colbert stream for reading/writing Colbert commands.
- SfFr * **frame**
  display frame; user programs can access this with the SfFRAME macro.

5.1.1 Detailed Description

Saphira system definition.

The Saphira system class is a static class that holds basic information about the single robot server for which Saphira is the client. It has useful functions to get the robot object, device buffers, artifact list, connections to the robot server, and so on.

On startup of the standard Saphira client, all the items in the Sf class are initialized by calling **init**() (p. 23).

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

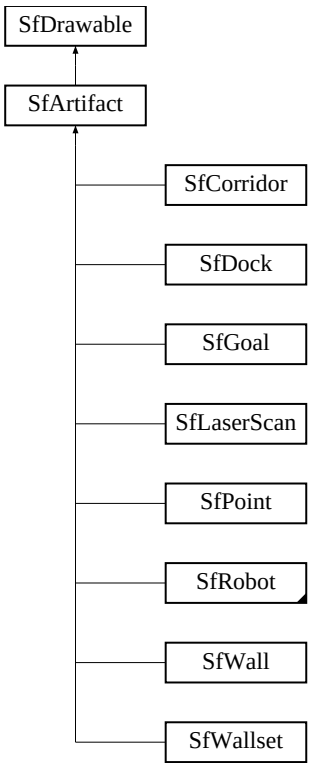
- **SfSystem.h**
- SfSystem.cpp

5.2 SfArtifact Class Reference

SfArtifact objects are Local Perceptual Space objects Inherit from this class if you want to be drawn on the LPS graphics window.

`#include <SfLps.h>`

Inheritance diagram for SfArtifact::



Public Types

- enum **Type** { **Robot**, **Point**, **Wall**, **Wallset**, **Corridor**, **Goal** }

Public Methods

- void **draw** (**SfWin** *w)

  This function is overridden by the artifact subclass to draw a particular artifact.

- **SfArtifact** ()

  Constructor, adds the artifact object to the artifact list.

- virtual ~**SfArtifact** ()

  Destructor, removes the artifact from the artifact list.

Public Attributes

- ArPose **p**

  Current pose of the object.

5.2.1 Detailed Description

SfArtifact objects are Local Perceptual Space objects Inherit from this class if you want to be drawn on the LPS graphics window.

The SfArtifact class is the standard way to draw objects on the LPS graphcis window. Inheriting from this class lets a subclass define the **draw**(SfWin *) (p. 25) function, which is called every time the graphics window is refreshed. The SfArtifact class adds its objects to the artifact list, and removes them on destruction. To turn off drawing of an artifact, use the visible flag (inherited from the **SfDrawable** (p. 32) class).

See also:
SfWin (p. 48) , **SfArtifactList** (p. 28)

5.2.2 Member Enumeration Documentation

5.2.2.1 enum SfArtifact::Type

Enumeration values:

Robot Robot object.

Point Point object, draws as a circle.

Wall Wall object, draws as a line.

Wallset Wall set object, not currently used.

Corridor Corridor object, draws as a parallel pair of double lines.

Goal Goal object, draws as a circle for now.

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

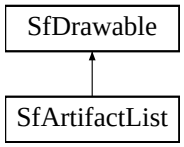
- **SfLps.h**
- SfObjects.cpp

5.3 SfArtifactList Class Reference

SfArtifactList is a static class that holds the set of current artifacts.

`#include <SfLps.h>`

Inheritance diagram for SfArtifactList::



Public Methods

- **SfVector** & **Bounds** (void)

  Returns the bounds of the artifacts in the artifact list The bounds are set from the most recent world file read in.

Static Public Methods

- SfArtifactList * **current** ()

  Artifact list.

5.3.1 Detailed Description

SfArtifactList is a static class that holds the set of current artifacts.

All artifacts objects, when created, put themselves onto the global artifact list; and they remove themselves when destroyed. User programs should not explicitly add or delete artifacts from this list. To stop an artifact from drawing, using the visible flag. User programs can request several facts about the artifact list, including its current bounds, which is set when a world is read in. They can also request the artifact list itself, to cycle through the artifacts.

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

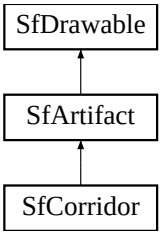
- **SfLps.h**
- SfObjects.cpp

5.4 SfCorridor Class Reference

Corridor artifacts: center point, width and length.

`#include <SfLps.h>`

Inheritance diagram for SfCorridor::



Public Methods

- **SfCorridor** (double x, double y, double th, double w, double l)

  Constructor using all parameters.

- **SfCorridor** ()

  Constructor using default of zero.

- virtual ~**SfCorridor** ()

  Destructor.

- void **draw** (**SfWin** *w)

  This function is overridden by the artifact subclass to draw a particular artifact.

Public Attributes

- double **width**

  Width and length of the corridor, can be reset.

- double **length**

  Width and length of the corridor, can be reset.

5.4.1 Detailed Description

Corridor artifacts: center point, width and length.

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

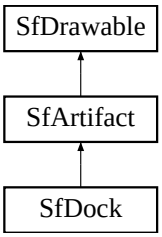
- **SfLps.h**
- SfObjects.cpp

5.5 SfDock Class Reference

Dock artifact, which is a pose and a name Draws as an orange circle.

`#include <SfLps.h>`

Inheritance diagram for SfDock::



Public Methods

- **SfDock** (double x, double y, double th, const char *name)

  Constructor using all parameters.

- **SfDock** ()

  Constructor with default 0,0,0.

- void **draw** (**SfWin** *w)

  This function is overridden by the artifact subclass to draw a particular artifact.

- virtual ~**SfDock** ()

  Destructor.

5.5.1 Detailed Description

Dock artifact, which is a pose and a name Draws as an orange circle.

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

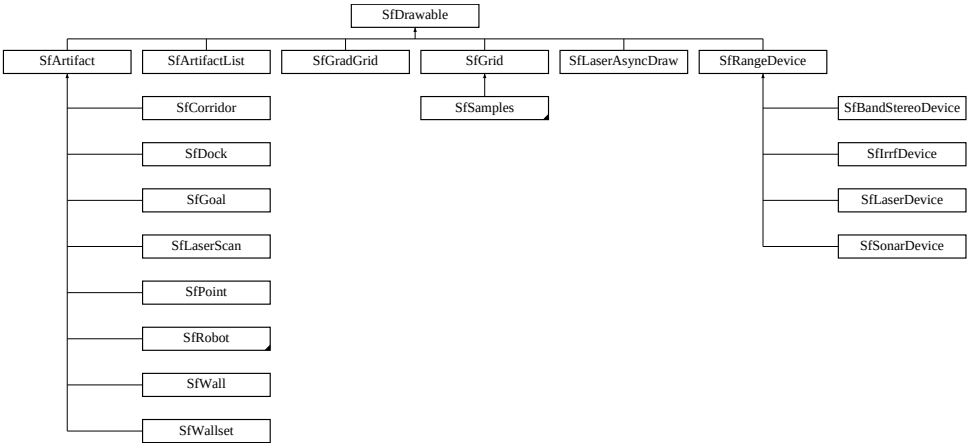
- **SfLps.h**
- SfObjects.cpp

5.6 SfDrawable Class Reference

SfDrawable class To draw on the LPS graphics window, inherit from the subclass **SfArtifact** (p. 25) Members variables here can be used to turn drawing on or off, change the color.

`#include <SfClass.h>`

Inheritance diagram for SfDrawable::



Public Attributes

- bool **visible**

  True if the object is to be drawn.

- int **color**

  Color of the object; not yet implemented...

5.6.1 Detailed Description

SfDrawable class To draw on the LPS graphics window, inherit from the subclass **SfArtifact** (p. 25) Members variables here can be used to turn drawing on or off, change the color.

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

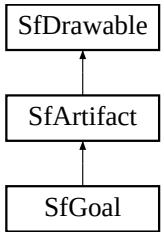
- **SfClass.h**

5.7 SfGoal Class Reference

Goal artifact, which is a pose and a name Draws as a green circle (with a line if it cares about heading).

`#include <SfLps.h>`

Inheritance diagram for SfGoal::



Public Methods

- **SfGoal** (double x, double y, double th, const char *name, bool useHeading)

  Constructor using all parameters.

- **SfGoal** ()

  Constructor with default 0,0,0.

- void **draw** (**SfWin** *w)

  This function is overridden by the artifact subclass to draw a particular artifact.

- virtual ~**SfGoal** ()

  Destructor.

5.7.1 Detailed Description

Goal artifact, which is a pose and a name Draws as a green circle (with a line if it cares about heading).

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

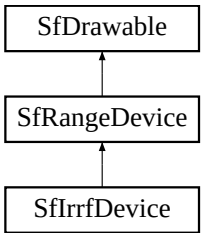
- **SfLps.h**
- SfObjects.cpp

5.8 SfIrrfDevice Class Reference

Irrf device class. Created by initialization of the **Sf** (p. 23) static class.

`#include <SfDevices.h>`

Inheritance diagram for SfIrrfDevice::



Public Methods

- double **getStartAngle** ()
  Start angle of laser scan wrt robot.
- double **getEndAngle** ()
  End angle of laser scan wrt robot.
- bool **isIrrfFlipped** ()
  Gets whether the laser is flipped over or not.
- double **getDegrees** ()
  Gets the degrees the laser is scanning.
- double **getIncrement** ()
  Gets the amount each scan increments.

5.8.1 Detailed Description

Irrf device class. Created by initialization of the **Sf** (p. 23) static class.

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

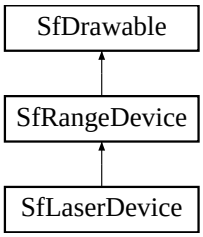
- **SfDevices.h**
- SfIrrfDev.cpp

5.9 SfLaserDevice Class Reference

Laser device class. Created by initialization of the **Sf** (p. 23) static class.

`#include <SfDevices.h>`

Inheritance diagram for SfLaserDevice::



Public Methods

- bool **start** (char *port)

  Start up the device.

- bool **stop** ()

  Stop the device.

- double **getStartAngle** ()

  Start angle of laser scan wrt robot.

- double **getEndAngle** ()

  End angle of laser scan wrt robot.

- bool **isLaserFlipped** ()

  Gets whether the laser is flipped over or not.

- double **getDegrees** ()

  Gets the degrees the laser is scanning.

- double **getIncrement** ()

  Gets the amount each scan increments.

5.9.1 Detailed Description

Laser device class. Created by initialization of the **Sf** (p. 23) static class.

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

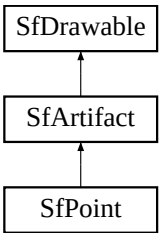
- **SfDevices.h**
- SfLaserDev.cpp

5.10 SfPoint Class Reference

Point artifact, which is a position and direction Draws as a circle.

`#include <SfLps.h>`

Inheritance diagram for SfPoint::



Public Methods

- **SfPoint** (double x, double y, double th)

  Constructor using all parameters.

- **SfPoint** ()

  Constructor with default 0,0,0.

- void **draw** (**SfWin** *w)

  This function is overridden by the artifact subclass to draw a particular artifact.

- virtual ~**SfPoint** ()

  Destructor.

5.10.1 Detailed Description

Point artifact, which is a position and direction Draws as a circle.

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

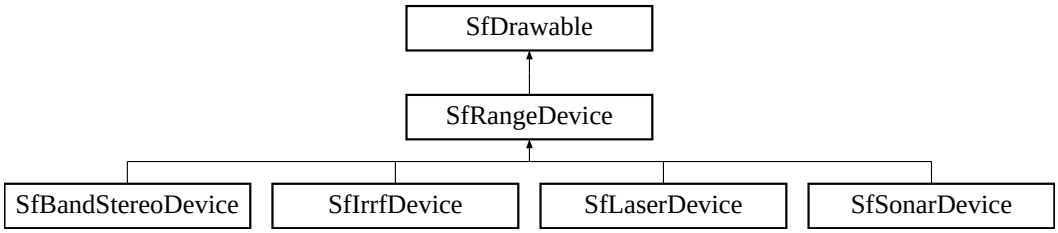
- **SfLps.h**
- SfObjects.cpp

5.11 SfRangeDevice Class Reference

SfRangeDevice is the Saphira class for range devices, encapsulating the ArRangeDevice class.

`#include <SfDevices.h>`

Inheritance diagram for SfRangeDevice::



Public Methods

- ArRangeBuffer * **getCurrent** ()

  Returns the current history buffer for the range device.

- ArRangeBuffer * **getAccum** ()

  Returns the accumulated history buffer for the range device.

- void **lockDevice** ()

  Lock the range buffers while we fool with them.

- void **unlockDevice** ()

  Unlock the buffers.

- const char * **getName** ()

  Name of the device.

Public Attributes

- ArRangeDevice * **ourRangeDevice**

  Aria range device class.

- bool **accumDraw**

  Set this to draw the accum buffer.

- bool **curDraw**

  Set this to draw the current buffer.

5.11.1 Detailed Description

SfRangeDevice is the Saphira class for range devices, encapsulating the ArRangeDevice class.

Instead of subclassing ArRangeDevice, we include a pointer to one as part of the class data. This way, the subclasses of ArRangeDevice can be accomodated.

User programs can access the current and accum buffers easily. Range device buffers are made available in the **Sf** (p. 23) class.

See also:
Sf (p. 23)

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

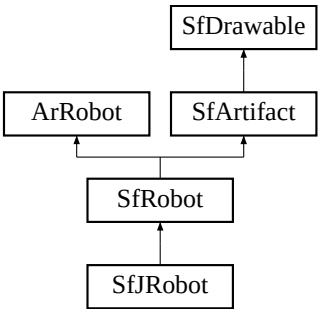
- **SfDevices.h**

5.12 SfRobot Class Reference

SfRobot inherits the basic ArRobot, plus is an artifact so it can be drawn.

`#include <SfLps.h>`

Inheritance diagram for SfRobot::



Public Methods

- void **draw** (**SfWin** *w)

  This function is overridden by the artifact subclass to draw a particular artifact.

5.12.1 Detailed Description

SfRobot inherits the basic ArRobot, plus is an artifact so it can be drawn.

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

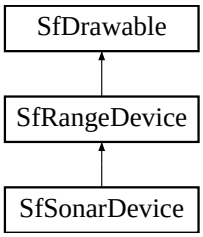
- **SfLps.h**
- SfObjects.cpp
- SfSystem.cpp

5.13 SfSonarDevice Class Reference

Sonar device class. Created by initialization of the **Sf** (p. 23) static class.

`#include <SfDevices.h>`

Inheritance diagram for SfSonarDevice::



5.13.1 Detailed Description

Sonar device class. Created by initialization of the **Sf** (p. 23) static class.

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

- **SfDevices.h**
- SfSonarDev.cpp

5.14 SfTime Class Reference

Unility timing class.

`#include <SfClass.h>`

Public Methods

- **SfTime** ()
  Constructor, sets time to zero for this object.
- void **Reset** ()
  Reset time to zero.
- int **TimeMS** ()
  returns current time in milliseconds.
- int **TimeUS** ()
  returns current time in microseconds.

5.14.1 Detailed Description

Unility timing class.

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

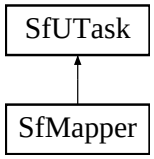
- **SfClass.h**
- SfUtil.cpp

5.15 SfUTask Class Reference

Saphira class that encapsulates the Aria synchronous task.

`#include <SfUTask.h>`

Inheritance diagram for SfUTask::



Public Methods

- **SfUTask** (char *name, int priority)

  Constructor, must be chained to by the subclass.

- virtual ~**SfUTask** ()

  Destructor.

- void **suspendTask** ()

  Suspend the task.

- void **resumeTask** ()

  Resume the task.

- virtual void **process** ()

  Task main body, called every sync cycle.

Public Attributes

- int **processState**

  Process state. Can be written to during normal processing, to set a user-defined state. Should not be used to suspend or resume the uTask.

5.15.1 Detailed Description

Saphira class that encapsulates the Aria synchronous task.

This Saphira micro-task class is a wrapper for the Aria synchronous task facility. Each SfUTask object is has its **process**() (p. 43) function called during the 100 ms synchronous cycle.

There are functions for suspending and resuming the uTask, which can be called from within **process**() (p. 43) or outside of it.

There is a tutorial program in the directory Saphira/tutor/task.

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

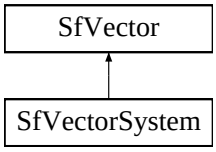
- **SfUTask.h**
- SfUTask.cpp

5.16 SfVector Class Reference

SfVector objects represent a vector by its endpoints.

`#include <SfClass.h>`

Inheritance diagram for SfVector::



Public Methods

- **SfVector** ()
  Default constructor, all zero coords.
- **SfVector** (double xx1, double yy1, double xx2, double yy2)
  Constructor, using endpoints.

5.16.1 Detailed Description

SfVector objects represent a vector by its endpoints.

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

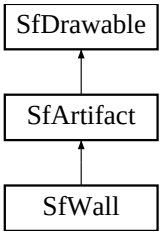
- **SfClass.h**

5.17 SfWall Class Reference

Wall artifacts: center point, length.

`#include <SfLps.h>`

Inheritance diagram for SfWall::



Public Methods

- **SfWall** (double x1, double y1, double x2, double y2)

  Constructor, using endpoint arguments.

- **SfWall** ()

  Constructor, using defaults of zero.

- virtual ~**SfWall** ()

  Destructor.

- void **draw** (**SfWin** *w)

  This function is overridden by the artifact subclass to draw a particular artifact.

Public Attributes

- double **length**

  Length of wall, should not be reset.

- **SfVector v**

  Vector representation, should not be reset.

5.17.1 Detailed Description

Wall artifacts: center point, length.

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

- **SfLps.h**
- SfObjects.cpp

5.18 SfWin Class Reference

Abstract window class Purpose of this class is to make drawing fns available without specifying any implementation E.g., could use FLTK, or a null implementation for no drawing.

`#include <SfClass.h>`

Public Types

- enum { **FIRST**, **LAST**, **REMOVE** }

Public Methods

- virtual void **Vector** (double, double, double, double)
  draws a line.
- virtual void **Vector** (double, double, double, double, ArPose ∗)
  draws a line relative to a pose; use NULL for the robot.
- virtual void **Rectangle** (double, double, double, double)
  draws a rectangle.
- virtual void **CRectangle** (double, double, double, double)
  draws a centered rectangle.
- virtual void **CRectangle** (double, double, double, double, ArPose ∗)
  draws a centered rectangle relative to a pose; use NULL for the robot.
- virtual void **Point** (double, double)
  draws a point.
- virtual void **Point** (double, double, ArPose ∗)
  draws a point relative to a pose; use NULL for the robot.
- virtual void **Polygon** (int, double ∗, double ∗)
  draws a polygon, using an array of points.
- virtual void **Circle** (double x, double y, double r)
  draws a circle at x,y, with radius r.
- virtual void **Circle** (double x, double y, double r, ArPose ∗)

draws a circle relative to a pose; use NULL for the robot.

- virtual void **Text** (char *str, double x, double y)

  draws a text string at location x,y.

- virtual void **Text** (char *str, double x, double y, ArPose *)

  draws a text string at location x,y relative to a pose; use NULL for the robot.

- virtual void **PenColor** (int)

  Sets the drawing pen color This holds until another pen color is set.

- virtual void **Coords** (double *x, double *y, int i, int j)

  returns RW coords from screen i,j.

- void **AddKeyHandler** (int(*fn)(int, int, SfWin *), int which=FIRST)

  adds a keystroke handler to the window.

- void **AddButtonHandler** (int(*fn)(int, int, int, int, SfWin *), int which=FIRST)

  adds a button press handler.

5.18.1 Detailed Description

Abstract window class Purpose of this class is to make drawing fns available without specifying any implementation E.g., could use FLTK, or a null implementation for no drawing.

5.18.2 Member Enumeration Documentation

5.18.2.1 anonymous enum

Enumeration values:

FIRST Put a button or key handler at the beginning of the callback list.

LAST At the end.

REMOVE Remove it.

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

- **SfClass.h**
- SfStream.cpp

Index