Player/Stage project

USC Robotics Laboratory
University of Southern California
Los Angeles, California, USA

libplayerc

Version 1.5 Reference Manual

Andrew Howard
ahoward@usc.edu

May 31, 2004

Contents

Chapter 1

Introduction

`libplayerc` is a client library for the Player robot device server. It is written in C to maximize portability, and in the expectation that users will write bindings for other languages (such as Java) against this library; Python bindings for `libplayerc` are already available.

Users reading this manual this manual should also read the *Player User Manual* included in the standard Player distribution.

1.1 Getting `libplayerc`

`libplayerc` is included with the standard Player distribution, starting with version 1.2. The Player homepage is:

```
http://playerstage.sourceforge.net
```

Check there for the latest versions of the Player distributon and this document. The default Player installation will build and install the following files:

```
$(HOME)/player-1.3/include/playerc/playerc.h
$(HOME)/player-1.3/lib/playerc/libplayerc.a
```

Make sure these directories are in your include and library paths, respectively.

1.2 Differences between versions 1.2 and 1.3

(Users who are familiar with version 1.2 of `libplayerc` should read this section; new users may safely ignore it).

The Player server has been substantially re-written to support a more general interface/driver model for devices. A device *interface* describes the kinds of interactions a device allows, while a device *driver* handles the low-level hardware interaction. Thus one may use the same interface to control many different pieces of hardware, each of which has a specific driver.

This change has two important consequences for the `libplayerc` client library:

- Client-side *proxies* now correspond to server-side *interfaces*. Thus, for example, a `position` proxy on the client connects to a `position` interface on the server.
- Some proxy names have changed to bring them into line with the new server naming conventions.

See the *Player User Manual* for more details on the new interface/driver model.

1.3 Bugs

This software is provided WITHOUT WARRANTY. Nevertheless, if you find something that doesn't work, or there is some feature you would like to see, you can submit a bug report/feature request through the Player/Stage homepage:

```
http://playerstage.sourceforge.net
```

Include a detailed description of you problem and/or feature request, and information such as the Player version and operating system. Make sure you also select the "`libplayerc` " category when reporting bugs.

1.4 Licence

1.5 Acknowledgements

This work is supported by DARPA grant DABT63-99-1-0015 (MARS) and NSF grant ANI-9979457 (SCOWR), and possibly others.

Chapter 2

General Usage

libplayerc is based on a device "proxy" model, in which the client maintains a local proxy for each of the devices on the remote server. Thus, for example, one can create local proxies for the position and laser devices. There is also a special client proxy, used to control the Player server itself.

Programs using libplayerc will generally the following structure:

```
#include <stdio.h>
#include "playerc.h"

int main(int argc, const char **argv)
{
  int i;
  playerc_client_t *client;
  playerc_position_t *position;

  client = playerc_client_create(NULL, "localhost", 6665);
  playerc_client_connect(client);

  position = playerc_position_create(client, 0);
  playerc_position_subscribe(position, PLAYER_ALL_MODE);

  playerc_position_enable(position, 1);
  playerc_position_set_speed(position, 0, 0, 0.1);

  for (i = 0; i < 200; i++)
  {
    playerc_client_read(client);
    printf("position : %f %f %f\n",
           position->px, position->py, position->pa);
  }

  playerc_position_unsubscribe(position);
  playerc_position_destroy(position);
  playerc_client_disconnect(client);
  playerc_client_destroy(client);

  return 0;
}
```

Note that error checking has been omitted from this example for the sake of clarity (for an example with full error checking, see simple.c in the examples/libplayerc directory). This example can be built using the commands:

```
$ gcc -c simple.c -o simple.o
$ gcc -lm -lplayerc simple.o -o simple
```

Make sure that `playerc.h` is in you include path, and that `libplayerc.a` is in your library path.

The above program can be broken into six steps, as follows.

Create and connect a client proxy.

```
client = playerc_client_create(NULL, "localhost", 6665);
playerc_client_connect(client);
```

The `create` function creates a new client proxy and returns a pointer to be used in future function calls (`localhost` should be replaced with the network host name of the robot). The `connect` function notifies the Player server that a new client wishes to recieve data.

Create and subscribe a device proxy.

```
position = playerc_position_create(client, 0);
playerc_position_subscribe(position, PLAYER_ALL_MODE);
```

The `create` function creates a new position device proxy and returns a pointer to be used in future function calls. The `subscribe` function notifies the Player server that the client is using the position device, and that the client expects to both send commands and recieve data (`PLAYER_MODE_ALL`).

Configure the device, send commands.

```
playerc_position_enable(position, 1);
playerc_position_set_speed(position, 0, 0, 0.1);
```

The `enable` function sends a configuration request to the server, changing the robot's motor state from `off` to `on`, thereby allowing the robot to move. The `setspeed` function sends a new motor speed, in this case commanding the robot to turn on the spot.

Note that most Player devices will accept both asynchronous *command* and synchronous *configuration* requests. Sending commands is analogous using the standard Unix `write` device interface, while sending configuration requests is analogous to using the `ioctl` interface. For the most part, `libplayerc` hides the distinction between these two interfaces. Users should be aware, however, that while commands are always handled promptly by the server, configuration requests may take significant time to complete. If possible, configuration requests should therefore be restricted to the initialization phase of the program.

Read data from the device.

```
playerc_client_read(client);
printf("position : %f %f %f\n", position->px, ... );
```

The `read` function blocks until new data arrives from the Player server. This data may be from one of the subscribed devices, or it may be from the server itself (which sends regular synchronization messages to all of its clients). The `read` function inspects the incoming data and automatically updates the elements in the appropriate device proxy. This function also returns a pointer to the proxy that was updated, so that user programs may, if desired, trigger appropriate events on the arrival of different kinds of data.

Unsubscribe and destroy the device proxy.

```
playerc_position_unsubscribe(position);
playerc_position_destroy(position);
```

The `unsubscribe` function tells the Player server that the client is no longer using this device. The `destroy` function then frees the memory associated with the device proxy; the `device` pointer is now invalid and should be not be re-used.

Disconnect and destroy the client proxy.

```
playerc_client_disconnect(client);
playerc_client_destroy(client);
```

The `disconnect` function tells the server that the client is shutting down. The `destroy` function then frees the memory associated with the client proxy; the `client` pointer is now invalid and should be not be re-used.

Chapter 3

Library Reference

This chapter contains information on the various non-device and generic proxies offered by `libplayerc` ; see Chapter 4 for information on specific device proxies.

3.1 client

The client proxy provides an interface to the Player server itself, and therefore has somewhat different functionality from the *device* proxies. The `create` function returns a pointer to an `playerc_client_t` structure to be used in other function calls.

struct playerc_client_t : Client data.

char *host;
int port;

Server address.

int device_count;
struct _playerc_device_t *device[32];

List of subscribed devices

double datatime;

Data time stamp on the last SYNC packet

playerc_client_t *playerc_client_create(playerc_mclient_t *mclient, const char *host, int port);

Create a single-client object. Set mclient to NULL if this is a stand-alone client.

void playerc_client_destroy(playerc_client_t *client);

Destroy a single-client object.

int playerc_client_connect(playerc_client_t *client); int playerc_client_disconnect(playerc_client_t *client);

Connect/disconnect to the server.

int playerc_client_get_devlist(playerc_client_t *client);

Get the list of available device ids. The data is written into the proxy structure rather than returned to the caller.

int playerc_client_peek(playerc_client_t *client, int timeout);

Test to see if there is pending data. Returns -1 on error, 0 or 1 otherwise.

void *playerc_client_read(playerc_client_t *client);

Read data from the server (blocking). For data packets, will return a pointer to the device proxy that got the data; for synch packets, will return a pointer to the client itself; on error, will return NULL.

3.2 device

The `device` proxy provides a 'generic' interface to the functionality that is shared by all devices (i.e., a base class, in OOP parlance). This proxy can be accessed through the `info` element present in each of the device proxies. In general, this proxy should not be used directly.

struct playerc_device_t : Generic device info.

playerc_client_t *client;

Pointer to the client proxy.

int code, index;

Device code, index.

char drivername[PLAYER_MAX_DEVICE_STRING_LEN];

The driver name.

int subscribed;

The subscribe flag is non-zero if the device has been successfully subscribed (read-only).

double datatime;

Data timestamp, i.e., the time at which the data was generated (s).

int fresh;

Freshness flag. Set to 1 whenever data is dispatched to this proxy. Useful with the mclient, but the user must manually set it to 0 after using the data

3.3 Errors

Synopsis

Most functions in `libplayerc` will return 0 on success and non-zero value on error. A descriptive error message will also be written into the error string.

extern const char *playerc_error_str(void);

Retrieve the last error (as a descriptive string).

Chapter 4

Device Reference

This chapter contains information on the various types of proxies offered by `libplayerc` : each type of proxy corresponds to one of the *interface* types offered by the Player server. Note that this section is not intended to provide a complete reference for all of the devices offered by Player, and should be read in conjunction with the Player User Manual.

4.1 Units

Unless otherwise specified, proxies use SI units (meters, radians, seconds) for all measurements.

4.2 Device Summary

`libplayerc` has proxies for the following devices:

`blobfinder`	Color-blob detector: an interface to a color blob detectors such as the ACTS vision system.
`comms`	Broadcast communications device: sends and receives packets between Player clients via the server.
`laser`	Laser device: interface for scanning laser range-finders such as the SICK LMS200.
`localize`	Localization device: returns the global robot pose.
`fiducial`	Fiducial detector: detects fiducials (beacons) placed in the environment.
`position`	Position device: for control of a mobile robot platform.
`power`	Power device: query batter levels.
`ptz`	Pan-tilt-zoom device: an interface to a pan-tilt-zoom camera head such as the Sony EVID30
`sonar`	Sonar device: an interface to an array of sonar range finders, such as those built into a mobile robot base.
`truth`	Truth device: allows clients to get and set the pose of objects in the Stage simulator. **Simulator only.**
`wifi`	WiFi device: queries link status on wireless networks.

4.3 blobfinder

Synopsis

The `blobfinder` proxy provides an interface to color blob detectors such as the ACTS vision system. See the Player User Manual for a complete description of the drivers that support this interface.

Data

struct playerc_blobfinder_blob_t : Description of a single blob.

int channel;

The blob ”channel”; i.e. the color class this blob belongs to.

uint32_t color;

A descriptive color for the blob. Stored as packed RGB 32, i.e.: 0x00RRGGBB.

int x, y;

Blob centroid (image coordinates).

int area;

Blob area (pixels).

int left, top, right, bottom;

Bounding box for blob (image coordinates).

struct playerc_blobfinder_t : Blobfinder device data.

playerc_device_t info;

Device info; must be at the start of all device structures.

int width, height;

Image dimensions (pixels).

int blob_count;
playerc_blobfinder_blob_t blobs[PLAYERC_BLOBFINDER_MAX_BLOBS];

A list of detected blobs.

Functions

playerc_blobfinder_t *playerc_blobfinder_create(playerc_client_t *client, int index);

Create a blobfinder proxy.

void playerc_blobfinder_destroy(playerc_blobfinder_t *device);

Destroy a blobfinder proxy.

int playerc_blobfinder_subscribe(playerc_blobfinder_t *device, int access);

Subscribe to the blobfinder device.

int playerc_blobfinder_unsubscribe(playerc_blobfinder_t *device);

Un-subscribe from the blobfinder device.

4.4 comms

Synposis

The `comms` proxy provides an interface to the network broadcast device. This device broadcasts any message sent to it onto the local network, and returns the messages broadcast by other robots. This device use broadcast UDP sockets, and therefore offers no guarantee that messages will be delivered, or that they will be delivered in the order in which they are transmitted.

Data

struct playerc_comms_t : Comms proxy data.

playerc_device_t info;

Device info; must be at the start of all device structures.

size_t msg_len;
uint8_t msg[PLAYER_MAX_MESSAGE_SIZE];

The most recent incoming messages.

Methods

playerc_comms_t *playerc_comms_create(playerc_client_t *client, int index);

Create a comms proxy.

void playerc_comms_destroy(playerc_comms_t *device);

Destroy a comms proxy.

int playerc_comms_subscribe(playerc_comms_t *device, int access);

Subscribe to the comms device.

int playerc_comms_unsubscribe(playerc_comms_t *device);

Un-subscribe from the comms device.

int playerc_comms_send(playerc_comms_t *device, void *msg, int len);

Send a comms message.

4.5 fiducial

Synopsis

The `fiducial` proxy provides an interface to a fiducial detector. This device looks for fiducials (markers or beacons) in the laser scan, and determines their identity, range, bearing and orientation. See the Player User Manual for a complete description of the various drivers that support the fiducial interface.

Data

struct playerc_fiducial_item_t : Description for a single fiducial.

int id;

Id (0 if fiducial cannot be identified).

double range, bearing, orient;

Beacon range, bearing and orientation.

typedef struct

Fiducial finder data.

playerc_device_t info;

Device info; must be at the start of all device structures.

double pose[3];
double size[2];
double fiducial_size[2];

Geometry in robot cs. These values are filled in by playerc_fiducial_get_geom(). [pose] is the detector pose in the robot cs, [size] is the detector size, [fiducial_size] is the fiducial size.

int fiducial_count;
playerc_fiducial_item_t fiducials[PLAYERC_FIDUCIAL_MAX_SAMPLES];

List of detected beacons.

Methods

playerc_fi ducial_t *playerc_fi ducial_create(playerc_client_t *client, int index);

Create a fiducial proxy.

void playerc_fi ducial_destroy(playerc_fi ducial_t *device);

Destroy a fiducial proxy.

int playerc_fi ducial_subscribe(playerc_fi ducial_t *device, int access);

Subscribe to the fiducial device.

int playerc_fiducial_unsubscribe(playerc_fiducial_t *device);

Un-subscribe from the fiducial device.

int playerc_fiducial_get_geom(playerc_fiducial_t *device);

Get the fiducial geometry. The writes the result into the proxy rather than returning it to the caller.

4.6 laser

Synopsis

The `laser` proxy provides an interface to a scanning laser range finder such as the SICK LMS200. See the Player User Manual for a complete description of this device.

Data

struct playerc_laser_t : Laser proxy data.

> **playerc_device_t info;**
>
> > Device info; must be at the start of all device structures.
>
> **double pose[3];**
> **double size[2];**
>
> > Laser geometry in the robot cs: pose gives the position and orientation, size gives the extent. These values are filled in by playerc_laser_get_geom().
>
> **int scan_count;**
>
> > Number of points in the scan.
>
> **double scan_res;**
>
> > Angular resolution of the scan (radians).
>
> **int range_res;**
>
> > Range resolution multiplier
>
> **double scan[PLAYERC_LASER_MAX_SAMPLES][2];**
>
> > Scan data; range (m) and bearing (radians).
>
> **double point[PLAYERC_LASER_MAX_SAMPLES][2];**
>
> > Scan data; x, y position (m).
>
> **int intensity[PLAYERC_LASER_MAX_SAMPLES];**
>
> > Scan reflection intensity values (0-3). Note that the intensity values will only be filled if intensity information is enabled (using the `set_config` function).

Methods

playerc_laser_t *playerc_laser_create(playerc_client_t *client, int index);

> Create a laser proxy.

void playerc_laser_destroy(playerc_laser_t *device);

Destroy a laser proxy.

int playerc_laser_subscribe(playerc_laser_t *device, int access);

Subscribe to the laser device.

int playerc_laser_unsubscribe(playerc_laser_t *device);

Un-subscribe from the laser device.

int playerc_laser_set_config(playerc_laser_t *device, double min_angle, double max_angle, int resolution, int range_res, int intensity);

Configure the laser. min_angle, max_angle : Start and end angles for the scan. resolution : Resolution in 0.01 degree increments. Valid values are 25, 50, 100. range_res : Range resolution. Valid: 1, 10, 100. intensity : Intensity flag; set to 1 to enable reflection intensity data.

int playerc_laser_get_config(playerc_laser_t *device, double *min_angle, double *max_angle, int *resolution, int *range_res, int *intensity);

Get the laser configuration min_angle, max_angle : Start and end angles for the scan. resolution : Resolution is in 0.01 degree increments. range_res : Range Resolution. Valid: 1, 10, 100. intensity : Intensity flag; set to 1 to enable reflection intensity data.

int playerc_laser_get_geom(playerc_laser_t *device);

Get the laser geometry. The writes the result into the proxy rather than returning it to the caller.

4.7 localize

Synopsis

The `localize` proxy provides an interface to localization drivers. Generally speaking, these are abstract drivers that attempt to localize the robot by matching sensor observations (odometry, laser and/or sonar) against a prior map of the environment (such as an occupancy grid). Since the pose may be ambiguous, multiple hypotheses may returned; each hypothesis specifies one possible pose estimate for the robot. See the Player manual for details of the `localize interface`, and and drivers that support it (such as the `amcl` driver).

Data

struct playerc_localize_hypoth_t : Hypothesis data.

double mean[3];

Pose estimate (x, y, theta) in (m, m, radians).

double cov[3][3];

Covariance.

double weight;

Weight associated with this hypothesis.

struct playerc_localize_t : Localization device data.

playerc_device_t info;

Device info; must be at the start of all device structures.

int map_size_x, map_size_y;

Map dimensions (cells).

double map_scale;

Map scale (m/cell).

int map_tile_x, map_tile_y;

Next map tile to read.

int8_t *map_cells;

Map data (empty = -1, unknown = 0, occupied = +1).

int pending_count;

The number of pending (unprocessed) sensor readings.

double pending_time;

The timestamp on the last reading processed.

int hypoth_count;
playerc_localize_hypoth_t hypoths[PLAYER_LOCALIZE_MAX_HYPOTHS];

List of possible poses.

Methods

playerc_localize_t *playerc_localize_create(playerc_client_t *client, int index);

Create a localize proxy.

void playerc_localize_destroy(playerc_localize_t *device);

Destroy a localize proxy.

int playerc_localize_subscribe(playerc_localize_t *device, int access);

Subscribe to the localize device.

int playerc_localize_unsubscribe(playerc_localize_t *device);

Un-subscribe from the localize device.

int playerc_localize_set_pose(playerc_localize_t *device, double pose[3], double cov[3][3]);

Set the the robot pose (mean and covariance).

int playerc_localize_get_map_info(playerc_localize_t *device);

Retrieve the occupancy map info. The info is written into the proxy structure.

int playerc_localize_get_map_tile(playerc_localize_t *device);

Retrieve a tile from occupancy map. The map is written into the proxy structure.

int playerc_localize_get_map(playerc_localize_t *device);

Retrieve the entire occupancy map. The map is written into the proxy structure.

int playerc_localize_get_config(playerc_localize_t *device, player_localize_config_t *config);

Get the current configuration.

int playerc_localize_set_config(playerc_localize_t *device, player_localize_config_t config);

Modify the current configuration.

4.8 position

Synopsis

The `position` proxy provides an interface to a mobile robot base, such as the ActiveMedia Pioneer series. The proxy supports both differential drive robots (which are capable of forward motion and rotation) and omni-drive robots (which are capable of forward, sideways and rotational motion).

Data

struct playerc_position_t : Position device data.

playerc_device_t info;

Device info; must be at the start of all device structures.

double pose[3];
double size[2];

Robot geometry in robot cs: pose gives the position and orientation, size gives the extent. These values are filled in by playerc_position_get_geom().

double px, py, pa;

Odometric pose (m, m, radians).

double vx, vy, va;

Odometric velocity (m, m, radians).

int stall;

Stall flag [0, 1].

Methods

playerc_position_t *playerc_position_create(playerc_client_t *client, int index);

Create a position device proxy.

void playerc_position_destroy(playerc_position_t *device);

Destroy a position device proxy.

int playerc_position_subscribe(playerc_position_t *device, int access);

Subscribe to the position device

int playerc_position_unsubscribe(playerc_position_t *device);

Un-subscribe from the position device

int playerc_position_enable(playerc_position_t *device, int enable);

Enable/disable the motors

int playerc_position_get_geom(playerc_position_t *device);

Get the position geometry. The writes the result into the proxy rather than returning it to the caller.

int playerc_position_set_cmd_vel(playerc_position_t *device, double vx, double vy, double va, int state);

Set the target speed. vx : forward speed (m/s). vy : sideways speed (m/s); this field is used by omni-drive robots only. va : rotational speed (radians/s). All speeds are defined in the robot coordinate system.

int playerc_position_set_cmd_pose(playerc_position_t *device, double gx, double gy, double ga, int state);

Set the target pose (gx, gy, ga) is the target pose in the odometric coordinate system.

4.9 power

Synopsis

The `power` proxy provides an interface through which battery levels can be monitored.

Data

struct playerc_power_t : Power device data.

> **playerc_device_t info;**
>
> > Device info; must be at the start of all device structures.
>
> **double charge;**
>
> > Battery charge (volts).

Methods

playerc_power_t *playerc_power_create(playerc_client_t *client, int index);

> Create a power device proxy.

void playerc_power_destroy(playerc_power_t *device);

> Destroy a power device proxy.

int playerc_power_subscribe(playerc_power_t *device, int access);

> Subscribe to the power device.

int playerc_power_unsubscribe(playerc_power_t *device);

> Un-subscribe from the power device.

4.10 ptz

Synopsis

The `ptz` proxy provides an interface to pan-tilt units such as the Sony PTZ camera.

Data

struct playerc_ptz_t : PTZ device data.

> **playerc_device_t info;**
>
> > Device info; must be at the start of all device structures.
>
> **double pan, tilt;**
>
> > The current ptz pan and tilt angles. pan : pan angle (+ve to the left, -ve to the right). tilt : tilt angle (+ve upwrds, -ve downwards).
>
> **double zoom;**
>
> > The current zoom value (field of view angle).

Methods

playerc_ptz_t *playerc_ptz_create(playerc_client_t *client, int index);

> Create a ptz proxy.

void playerc_ptz_destroy(playerc_ptz_t *device);

> Destroy a ptz proxy.

int playerc_ptz_subscribe(playerc_ptz_t *device, int access);

> Subscribe to the ptz device.

int playerc_ptz_unsubscribe(playerc_ptz_t *device);

> Un-subscribe from the ptz device.

int playerc_ptz_set(playerc_ptz_t *device, double pan, double tilt, double zoom);

> Set the pan, tilt and zoom values.

int playerc_ptz_set_ws(playerc_ptz_t *device, double pan, double tilt, double zoom, double panspeed, double tiltspeed);

> Set the pan, tilt and zoom values. (and speed)

4.11 sonar

Synopsis

The `sonar` proxy provides an interface to the sonar range sensors built into robots such as the ActiveMedia Pioneer series.

Data

struct playerc_sonar_t : Sonar proxy data.

> **playerc_device_t info;**
>
> > Device info; must be at the start of all device structures.
>
> **int pose_count;**
>
> > Number of pose values.
>
> **double poses[PLAYERC_SONAR_MAX_SAMPLES][3];**
>
> > Pose of each sonar relative to robot (m, m, radians). This structure is filled by calling playerc_sonar_get_geom().
>
> **int scan_count;**
>
> > Number of points in the scan.
>
> **double scan[PLAYERC_SONAR_MAX_SAMPLES];**
>
> > Scan data: range (m).

Methods

playerc_sonar_t *playerc_sonar_create(playerc_client_t *client, int index);

> Create a sonar proxy.

void playerc_sonar_destroy(playerc_sonar_t *device);

> Destroy a sonar proxy.

int playerc_sonar_subscribe(playerc_sonar_t *device, int access);

> Subscribe to the sonar device.

int playerc_sonar_unsubscribe(playerc_sonar_t *device);

> Un-subscribe from the sonar device.

int playerc_sonar_get_geom(playerc_sonar_t *device);

> Get the sonar geometry. The writes the result into the proxy rather than returning it to the caller.

4.12 truth

Synposis

The `truth` proxy can be used to get and set the pose of objects in the Stage simulator.

Data

struct playerc_truth_t : Truth proxy data.

playerc_device_t info;

Device info; must be at the start of all device structures.

double px, py, pa;

The object pose (world cs).

Methods

playerc_truth_t *playerc_truth_create(playerc_client_t *client, int index);

Create a truth proxy.

void playerc_truth_destroy(playerc_truth_t *device);

Destroy a truth proxy.

int playerc_truth_subscribe(playerc_truth_t *device, int access);

Subscribe to the truth device.

int playerc_truth_unsubscribe(playerc_truth_t *device);

Un-subscribe from the truth device.

int playerc_truth_get_pose(playerc_truth_t *device, double *px, double *py, double *pa);

Get the object pose. px, py, pa : the pose (global coordinates).

int playerc_truth_set_pose(playerc_truth_t *device, double px, double py, double pa);

Set the object pose. px, py, pa : the new pose (global coordinates).

4.13 wifi

Synopsis

The `wifi` proxy is used to query the state of a wireless network. It returns information such as the link quality and signal strength of access points or of other wireless NIC's on an ad-hoc network.

Data

struct playerc_wifi_link_t : Individual link info.

char ip[32];

Destination IP address.

int qual, level, noise;

Link properties.

struct player_wifi_t : Wifi device proxy.

playerc_device_t info;

Device info; must be at the start of all device structures.

int link_count;
playerc_wifi_link_t links[PLAYERC_WIFI_MAX_LINKS];

A list containing info for each link.

Methods

playerc_wifi_t *playerc_wifi_create(playerc_client_t *client, int index);

Create a wifi proxy.

void playerc_wifi_destroy(playerc_wifi_t *device);

Destroy a wifi proxy.

int playerc_wifi_subscribe(playerc_wifi_t *device, int access);

Subscribe to the wifi device.

int playerc_wifi_unsubscribe(playerc_wifi_t *device);

Un-subscribe from the wifi device.

Appendix A

Complete Header Listing

This is a complete listing of `playerc.h`.

```
/*
 *  libplayerc : a Player client library
 *  Copyright (C) Andrew Howard 2002-2003
 *
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  as published by the Free Software Foundation; either version 2
 *  of the License, or (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
 *
 */
/*
 *  Player - One Hell of a Robot Server
 *  Copyright (C) Andrew Howard 2003
 *
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU Lesser General Public
 *  License as published by the Free Software Foundation; either
 *  version 2.1 of the License, or (at your option) any later version.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */
/***************************************************************************
 * Desc: Player client
 * Author: Andrew Howard
 * Date: 24 Aug 2001
 # CVS: $Id: playerc.h,v 1.72 2004/05/14 18:26:28 inspectorg Exp $
 **************************************************************************/

#ifndef PLAYERC_H
#define PLAYERC_H

#include <stdio.h>

// Get the message structures from Player
#include "player.h"

#ifndef MIN
  #define MIN(a,b) ((a < b) ? a : b)
#endif
#ifndef MAX
  #define MAX(a,b) ((a > b) ? a : b)
#endif

#ifdef __cplusplus
extern "C" {
#endif


/**************************************************************************
 * Array sizes
 **************************************************************************/

#define PLAYERC_LASER_MAX_SAMPLES         PLAYER_LASER_MAX_SAMPLES
#define PLAYERC_FIDUCIAL_MAX_SAMPLES      PLAYER_FIDUCIAL_MAX_SAMPLES
#define PLAYERC_SONAR_MAX_SAMPLES         PLAYER_SONAR_MAX_SAMPLES
#define PLAYERC_BUMPER_MAX_SAMPLES                PLAYER_BUMPER_MAX_SAMPLES
#define PLAYERC_IR_MAX_SAMPLES               PLAYER_IR_MAX_SAMPLES
#define PLAYERC_BLOBFINDER_MAX_BLOBS      64
#define PLAYERC_WIFI_MAX_LINKS            PLAYER_WIFI_MAX_LINKS


/**************************************************************************
 * Declare all of the basic types
 **************************************************************************/

// Forward declare types
struct _playerc_client_t;
struct _playerc_device_t;

// Items in incoming data queue.
typedef struct
{
  player_msghdr_t header;
  int len;
  void *data;
} playerc_client_item_t;


// Typedefs for proxy callback functions
typedef void (*playerc_putdata_fn_t) (void *device, char *header, char *data, size_t len);
typedef void (*playerc_callback_fn_t) (void *data);
```

```
// forward declaration to avoid including <sys/poll.h>, which may not be
// available when people are building clients against this lib
struct pollfd;

// Multi-client data
typedef struct
{
  // List of clients being managed
  int client_count;
  struct _playerc_client_t *client[128];

  // Poll info
  struct pollfd* pollfd;

} playerc_mclient_t;


/***************************************************************************
 ** begin section Errors
 **************************************************************************/

/** [Synopsis] Most functions in \libplayerc will return 0 on success
and non-zero value on error. A descriptive error message will also be
written into the error string. */

/** Retrieve the last error (as a descriptive string). */
extern const char *playerc_error_str(void);

/***************************************************************************
 ** end section
 **************************************************************************/


/***************************************************************************
 * Utility functions
 **************************************************************************/

// Get the name for a given device code.
extern const char *playerc_lookup_name(int code);

// Get the device code for a give name.
extern int playerc_lookup_code(const char *name);


/***************************************************************************
 * Multi-client functions
 **************************************************************************/

// Create a multi-client object
playerc_mclient_t *playerc_mclient_create(void);

// Destroy a multi-client object
void playerc_mclient_destroy(playerc_mclient_t *mclient);

// Add a client to the multi-client (private).
int playerc_mclient_addclient(playerc_mclient_t *mclient, struct _playerc_client_t *client);
```

```
// Test to see if there is pending data.
// Returns -1 on error, 0 or 1 otherwise.
int playerc_mclient_peek(playerc_mclient_t *mclient, int timeout);

// Read incoming data.  The timeout is in ms.  Set timeout to a
// negative value to wait indefinitely.
int playerc_mclient_read(playerc_mclient_t *mclient, int timeout);


/***************************************************************************
 ** begin section client
 **************************************************************************/

/** The client proxy provides an interface to the Player server
    itself, and therefore has somewhat different functionality from
    the {\em device} proxies. The {\tt create} function returns a
    pointer to an {\tt playerc\_client\_t} structure to be used in
    other function calls.  */

/** Client data. */
typedef struct _playerc_client_t
{
  /** Server address. */
  char *host;
  int port;

  // Socket descriptor
  int sock;

  // List of ids for available devices.  This list is filled in by
  // playerc_client_get_devlist().
  int id_count;
  player_device_id_t ids[PLAYER_MAX_DEVICES];
  char drivernames[PLAYER_MAX_DEVICES][PLAYER_MAX_DEVICE_STRING_LEN];

  /** List of subscribed devices */
  int device_count;
  struct _playerc_device_t *device[32];

  // A circular queue used to buffer incoming data packets.
  int qfirst, qlen, qsize;
  playerc_client_item_t qitems[128];

  /** Data time stamp on the last SYNC packet */
  double datatime;

} playerc_client_t;


/** Create a single-client object.
    Set mclient to NULL if this is a stand-alone client.*/
playerc_client_t *playerc_client_create(playerc_mclient_t *mclient,
                                        const char *host, int port);

/** Destroy a single-client object. */
void playerc_client_destroy(playerc_client_t *client);

/** Connect/disconnect to the server. */
```

```
int playerc_client_connect(playerc_client_t *client);
int playerc_client_disconnect(playerc_client_t *client);

// Change the server's data delivery mode.
int playerc_client_datamode(playerc_client_t *client, int mode);
// Change the server's data delivery frequency (freq is in Hz)
int playerc_client_datafreq(playerc_client_t *client, int freq);

// Add/remove a device proxy (private)
int playerc_client_adddevice(playerc_client_t *client, struct _playerc_device_t *device);
int playerc_client_deldevice(playerc_client_t *client, struct _playerc_device_t *device);

// Add/remove user callbacks (called when new data arrives).
int  playerc_client_addcallback(playerc_client_t *client, struct _playerc_device_t *device,
                                playerc_callback_fn_t callback, void *data);
int  playerc_client_delcallback(playerc_client_t *client, struct _playerc_device_t *device,
                                playerc_callback_fn_t callback, void *data);

/** Get the list of available device ids. The data is written into the
    proxy structure rather than returned to the caller. */
int playerc_client_get_devlist(playerc_client_t *client);

// Subscribe/unsubscribe a device from the sever (private)
int playerc_client_subscribe(playerc_client_t *client, int code, int index,
                             int access, char *drivername, size_t len);
int playerc_client_unsubscribe(playerc_client_t *client, int code, int index);

// Issue a request to the server and await a reply (blocking).
// Returns -1 on error and -2 on NACK. (private)
int playerc_client_request(playerc_client_t *client, struct _playerc_device_t *device,
                           void *req_data, int req_len, void *rep_data, int rep_len);

/* Enable these if Brian changes to server to accept multiple requests.
// Issue request only; use in conjunction with
// playerc_client_request_recv() to issue multiple requests and get
// multiple replies.
int playerc_client_request_send(playerc_client_t *client, playerc_device_t *deviceinfo,
                                void *req_data, int req_len);

// Wait for a reply; use in conjunction with
// playerc_client_request_send() to issue multiple requests and get
// multiple replies.
int playerc_client_request_recv(playerc_client_t *client, playerc_device_t *deviceinfo,
                                void *rep_data, int rep_len);
*/

/** Test to see if there is pending data.
    Returns -1 on error, 0 or 1 otherwise. */
int playerc_client_peek(playerc_client_t *client, int timeout);

/** Read data from the server (blocking). For data packets, will
    return a pointer to the device proxy that got the data; for synch
    packets, will return a pointer to the client itself; on error, will
    return NULL. */
void *playerc_client_read(playerc_client_t *client);

// Write data to the server (private).
int playerc_client_write(playerc_client_t *client, struct _playerc_device_t *device,
                         void *cmd, int len);

/***************************************************************************
 ** end section
 **************************************************************************/


/***************************************************************************
 ** begin section device
 **************************************************************************/

/** The {\tt device} proxy provides a `generic' interface to the
    functionality that is shared by all devices (i.e., a base class,
    in OOP parlance).  This proxy can be accessed through the {\tt
    info} element present in each of the device proxies.  In general, this
    proxy should not be used directly. */

/** Generic device info. */
typedef struct _playerc_device_t
{
  /** Pointer to the client proxy. */
  playerc_client_t *client;

  /** Device code, index. */
  int code, index;

  /** The driver name. */
  char drivername[PLAYER_MAX_DEVICE_STRING_LEN];

  /** The subscribe flag is non-zero if the device has been
      successfully subscribed (read-only). */
  int subscribed;

  /** Data timestamp, i.e., the time at which the data was generated (s). */
  double datatime;

  /** Freshness flag.  Set to 1 whenever data is dispatched to this proxy.
      Useful with the mclient, but the user must manually set it to 0 after
      using the data */
  int fresh;

  // Standard callbacks for this device (private).
  playerc_putdata_fn_t putdata;

  // Extra user data for this device (private).
  void *user_data;

  // Extra callbacks for this device (private).
  int callback_count;
  playerc_callback_fn_t callback[4];
  void *callback_data[4];

} playerc_device_t;


/* Initialise the device (private). */
void playerc_device_init(playerc_device_t *device, playerc_client_t *client,
                         int code, int index, playerc_putdata_fn_t putdata);
```

```
/* Finalize the device (private). */
void playerc_device_term(playerc_device_t *device);

/* Subscribe the device (private). */
int playerc_device_subscribe(playerc_device_t *device, int access);

/* Unsubscribe the device (private). */
int playerc_device_unsubscribe(playerc_device_t *device);

/***************************************************************************
 ** end section
 **************************************************************************/


/***************************************************************************
 ** begin section blobfinder
 **************************************************************************/

/** [Synopsis] The {\tt blobfinder} proxy provides an interface to
 color blob detectors such as the ACTS vision system.  See the Player
 User Manual for a complete description of the drivers that support
 this interface. */

/** [Data] */

/** Description of a single blob. */
typedef struct _playerc_blobfinder_blob_t
{
  /** The blob "channel"; i.e. the color class this blob belongs to. */
  int channel;

  /** A descriptive color for the blob.  Stored as packed RGB 32, i.e.:
      0x00RRGGBB. */
  uint32_t color;

  /** Blob centroid (image coordinates). */
  int x, y;

  /** Blob area (pixels). */
  int area;

  /** Bounding box for blob (image coordinates). */
  int left, top, right, bottom;

} playerc_blobfinder_blob_t;


/** Blobfinder device data. */
typedef struct _playerc_blobfinder_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** Image dimensions (pixels). */
  int width, height;

  /** A list of detected blobs. */
  int blob_count;
  playerc_blobfinder_blob_t blobs[PLAYERC_BLOBFINDER_MAX_BLOBS];

} playerc_blobfinder_t;


/** [Functions] */

/** Create a blobfinder proxy. */
playerc_blobfinder_t *playerc_blobfinder_create(playerc_client_t *client, int index);

/** Destroy a blobfinder proxy. */
void playerc_blobfinder_destroy(playerc_blobfinder_t *device);

/** Subscribe to the blobfinder device. */
int playerc_blobfinder_subscribe(playerc_blobfinder_t *device, int access);

/** Un-subscribe from the blobfinder device. */
int playerc_blobfinder_unsubscribe(playerc_blobfinder_t *device);

/***************************************************************************
 ** end section
 ***************************************************************************/


/***************************************************************************
 ** begin section bps
 ***************************************************************************/

// BPS device data
typedef struct
{
  // Device info; must be at the start of all device structures.
  playerc_device_t info;

  // Robot pose estimate (global coordinates).
  double px, py, pa;

  // Robot pose uncertainty (global coordinates).
  double ux, uy, ua;

  // Residual error in estimates pose.
  double err;

} playerc_bps_t;


// Create a bps proxy
playerc_bps_t *playerc_bps_create(playerc_client_t *client, int index);

// Destroy a bps proxy
void playerc_bps_destroy(playerc_bps_t *device);

// Subscribe to the bps device
int playerc_bps_subscribe(playerc_bps_t *device, int access);

// Un-subscribe from the bps device
int playerc_bps_unsubscribe(playerc_bps_t *device);
```

```
// Set the pose of a beacon in global coordinates.
// id : the beacon id.
// px, py, pa : the beacon pose (global coordinates).
// ux, uy, ua : the uncertainty in the beacon pose.
int  playerc_bps_set_beacon(playerc_bps_t *device, int id,
                            double px, double py, double pa,
                            double ux, double uy, double ua);

// Get the pose of a beacon in global coordinates.
// id : the beacon id.
// px, py, pa : the beacon pose (global coordinates).
// ux, uy, ua : the uncertainty in the beacon pose.
int  playerc_bps_get_beacon(playerc_bps_t *device, int id,
                            double *px, double *py, double *pa,
                            double *ux, double *uy, double *ua);

/***************************************************************************
 ** end section
 ***************************************************************************/


/***************************************************************************
 ** begin section camera
 ***************************************************************************/

/** [Synposis] The {\tt camera} proxy can be used to get images from a
    camera. */

/** [Data] */

/** Camera proxy data. */
typedef struct _playerc_camera_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** Image dimensions (pixels). */
  uint16_t width, height;

  /** Image depth (8, 16, 24). */
  uint8_t depth;

  /** Size of image data (bytes) */
  uint32_t image_size;

  /** Image data (packed format). */
  uint8_t image[PLAYER_CAMERA_IMAGE_SIZE];

} playerc_camera_t;


/** [Methods] */

/** Create a camera proxy. */
playerc_camera_t *playerc_camera_create(playerc_client_t *client, int index);

/** Destroy a camera proxy. */
```

```
void playerc_camera_destroy(playerc_camera_t *device);

/** Subscribe to the camera device. */
int playerc_camera_subscribe(playerc_camera_t *device, int access);

/** Un-subscribe from the camera device. */
int playerc_camera_unsubscribe(playerc_camera_t *device);


/***************************************************************************
 ** end section
 ***************************************************************************/


/***************************************************************************
 ** begin section comms
 ***************************************************************************/

/** [Synposis] The {\tt comms} proxy provides an interface to the
network broadcast device.  This device broadcasts any message sent to
it onto the local network, and returns the messages broadcast by other
robots.  This device use broadcast UDP sockets, and therefore offers
no guarantee that messages will be delivered, or that they will be
delivered in the order in which they are transmitted. */

/** [Data] */

/** Comms proxy data. */
typedef struct _playerc_comms_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** The most recent incoming messages. */
  size_t msg_len;
  uint8_t msg[PLAYER_MAX_MESSAGE_SIZE];

} playerc_comms_t;

/** [Methods] */

/** Create a comms proxy. */
playerc_comms_t *playerc_comms_create(playerc_client_t *client, int index);

/** Destroy a comms proxy. */
void playerc_comms_destroy(playerc_comms_t *device);

/** Subscribe to the comms device. */
int playerc_comms_subscribe(playerc_comms_t *device, int access);

/** Un-subscribe from the comms device. */
int playerc_comms_unsubscribe(playerc_comms_t *device);

/** Send a comms message. */
int playerc_comms_send(playerc_comms_t *device, void *msg, int len);

/***************************************************************************
 ** end section
 ***************************************************************************/


/***************************************************************************
 ** begin section fiducial
 ***************************************************************************/

/** [Synopsis] The {\tt fiducial} proxy provides an interface to a
fiducial detector.  This device looks for fiducials (markers or
beacons) in the laser scan, and determines their identity, range,
bearing and orientation.  See the Player User Manual for a complete
description of the various drivers that support the fiducial
interface. */

/** [Data] */

/** Description for a single fiducial. */
typedef struct _playerc_fiducial_item_t
{
  /** Id (0 if fiducial cannot be identified). */
  int id;

  /** Beacon range, bearing and orientation. */
  double range, bearing, orient;

} playerc_fiducial_item_t;


/** Fiducial finder data. */
typedef struct
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** Geometry in robot cs.  These values are filled in by
      playerc_fiducial_get_geom().  [pose] is the detector pose in the
      robot cs, [size] is the detector size, [fiducial_size] is the
      fiducial size. */
  double pose[3];
  double size[2];
  double fiducial_size[2];

  /** List of detected beacons. */
  int fiducial_count;
  playerc_fiducial_item_t fiducials[PLAYERC_FIDUCIAL_MAX_SAMPLES];

} playerc_fiducial_t;

/** [Methods] */

/** Create a fiducial proxy. */
playerc_fiducial_t *playerc_fiducial_create(playerc_client_t *client, int index);

/** Destroy a fiducial proxy. */
void playerc_fiducial_destroy(playerc_fiducial_t *device);

/** Subscribe to the fiducial device. */
int playerc_fiducial_subscribe(playerc_fiducial_t *device, int access);
```

```
/** Un-subscribe from the fiducial device. */
int playerc_fiducial_unsubscribe(playerc_fiducial_t *device);

/** Get the fiducial geometry.  The writes the result into the proxy
    rather than returning it to the caller. */
int playerc_fiducial_get_geom(playerc_fiducial_t *device);

/***************************************************************************
 ** end section
 ***************************************************************************/


/***************************************************************************
 ** begin section gps
 ***************************************************************************/

/** [Synopsis] The {\tt gps} proxy provides an interface to a
GPS-receiver. */

/** [Data] */

/** GPS proxy data. */
typedef struct _playerc_gps_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** UTC time (seconds since the epoch) */
  double utc_time;

  /** Latitude and logitude (degrees).  Latitudes are positive for
      north, negative for south.  Logitudes are positive for east,
      negative for west. */
  double lat, lon;

  /** Altitude (meters).  Positive is above sea-level, negative is
      below. */
  double alt;

  /** UTM easting and northing (meters). */
  double utm_e, utm_n;

  /** Horizontal dilution of precision. */
  double hdop;

  /** Horizontal and vertical error (meters). */
  double err_horz, err_vert;

  /** Quality of fix 0 = invalid, 1 = GPS fix, 2 = DGPS fix */
  int quality;

  /** Number of satellites in view. */
  int sat_count;

} playerc_gps_t;

/** [Methods] */
/** Create a gps proxy. */
playerc_gps_t *playerc_gps_create(playerc_client_t *client, int index);

/** Destroy a gps proxy. */
void playerc_gps_destroy(playerc_gps_t *device);

/** Subscribe to the gps device. */
int playerc_gps_subscribe(playerc_gps_t *device, int access);

/** Un-subscribe from the gps device. */
int playerc_gps_unsubscribe(playerc_gps_t *device);

/***************************************************************************
 ** end section
 ***************************************************************************/


/***************************************************************************
 ** begin section laser
 ***************************************************************************/

/** [Synopsis] The {\tt laser} proxy provides an interface to a
    scanning laser range finder such as the SICK LMS200.  See the
    Player User Manual for a complete description of this device. */

/** [Data] */

/** Laser proxy data. */
typedef struct _playerc_laser_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** Laser geometry in the robot cs: pose gives the position and
      orientation, size gives the extent.  These values are filled in by
      playerc_laser_get_geom(). */
  double pose[3];
  double size[2];

  /** Number of points in the scan. */
  int scan_count;

  /** Angular resolution of the scan (radians). */
  double scan_res;

  /** Range resolution multiplier */
  int range_res;

  /** Scan data; range (m) and bearing (radians). */
  double scan[PLAYERC_LASER_MAX_SAMPLES][2];

  /** Scan data; x, y position (m). */
  double point[PLAYERC_LASER_MAX_SAMPLES][2];

  /** Scan reflection intensity values (0-3).  Note that the intensity
      values will only be filled if intensity information is enabled
      (using the {\tt set\_config} function). */
  int intensity[PLAYERC_LASER_MAX_SAMPLES];

} playerc_laser_t;


/** [Methods] */

/** Create a laser proxy. */
playerc_laser_t *playerc_laser_create(playerc_client_t *client, int index);

/** Destroy a laser proxy. */
void playerc_laser_destroy(playerc_laser_t *device);

/** Subscribe to the laser device. */
int playerc_laser_subscribe(playerc_laser_t *device, int access);

/** Un-subscribe from the laser device. */
int playerc_laser_unsubscribe(playerc_laser_t *device);

/** Configure the laser.
    min_angle, max_angle : Start and end angles for the scan.
    resolution : Resolution in 0.01 degree increments.  Valid values are 25, 50, 100.
    range_res : Range resolution.  Valid: 1, 10, 100.
    intensity : Intensity flag; set to 1 to enable reflection intensity data. */
int  playerc_laser_set_config(playerc_laser_t *device, double min_angle,
                              double max_angle, int resolution,
                              int range_res, int intensity);

/** Get the laser configuration
    min_angle, max_angle : Start and end angles for the scan.
    resolution : Resolution is in 0.01 degree increments.
    range_res : Range Resolution.  Valid: 1, 10, 100.
    intensity : Intensity flag; set to 1 to enable reflection intensity data. */
int  playerc_laser_get_config(playerc_laser_t *device, double *min_angle,
                              double *max_angle, int *resolution,
                              int *range_res, int *intensity);

/** Get the laser geometry.  The writes the result into the proxy
    rather than returning it to the caller. */
int playerc_laser_get_geom(playerc_laser_t *device);

/***************************************************************************
 ** end section
 ***************************************************************************/



/***************************************************************************
 ** begin section localize
 ***************************************************************************/

/** [Synopsis] The {\tt localize} proxy provides an interface to
    localization drivers.  Generally speaking, these are abstract
    drivers that attempt to localize the robot by matching sensor
    observations (odometry, laser and/or sonar) against a prior map of
    the environment (such as an occupancy grid).  Since the pose may
    be ambiguous, multiple hypotheses may returned; each hypothesis
    specifies one possible pose estimate for the robot.  See the
    Player manual for details of the {\tt localize interface}, and and
    drivers that support it (such as the {\tt amcl} driver). */

/** [Data] */

/** Hypothesis data. */
typedef struct _playerc_localize_hypoth_t
{
  /** Pose estimate (x, y, theta) in (m, m, radians). */
  double mean[3];

  /** Covariance. */
  double cov[3][3];

  /** Weight associated with this hypothesis. */
  double weight;

} playerc_localize_hypoth_t;


/** Localization device data. */
typedef struct _playerc_localize_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** Map dimensions (cells). */
  int map_size_x, map_size_y;

  /** Map scale (m/cell). */
  double map_scale;

  /** Next map tile to read. */
  int map_tile_x, map_tile_y;

  /** Map data (empty = -1, unknown = 0, occupied = +1). */
  int8_t *map_cells;

  /** The number of pending (unprocessed) sensor readings. */
  int pending_count;

  /** The timestamp on the last reading processed. */
  double pending_time;

  /** List of possible poses. */
  int hypoth_count;
  playerc_localize_hypoth_t hypoths[PLAYER_LOCALIZE_MAX_HYPOTHS];

} playerc_localize_t;

/** [Methods] */

/** Create a localize proxy. */
playerc_localize_t *playerc_localize_create(playerc_client_t *client, int index);

/** Destroy a localize proxy. */
void playerc_localize_destroy(playerc_localize_t *device);
```

```
/** Subscribe to the localize device. */
int playerc_localize_subscribe(playerc_localize_t *device, int access);

/** Un-subscribe from the localize device. */
int playerc_localize_unsubscribe(playerc_localize_t *device);

/** Set the the robot pose (mean and covariance). */
int playerc_localize_set_pose(playerc_localize_t *device, double pose[3], double cov[3][3]);

/** Retrieve the occupancy map info.  The info is written into the proxy
    structure. */
int playerc_localize_get_map_info(playerc_localize_t *device);

/** Retrieve a tile from occupancy map.  The map is written into the proxy
    structure. */
int playerc_localize_get_map_tile(playerc_localize_t *device);

/** Retrieve the entire occupancy map.  The map is written into the proxy
   structure. */
int playerc_localize_get_map(playerc_localize_t *device);

/** Get the current configuration. */
int playerc_localize_get_config(playerc_localize_t *device, player_localize_config_t *config);

/** Modify the current configuration. */
int playerc_localize_set_config(playerc_localize_t *device, player_localize_config_t config);

/***************************************************************************
 ** end section
 ***************************************************************************/



/***************************************************************************
 ** begin section position
 ***************************************************************************/

/** [Synopsis] The {\tt position} proxy provides an interface to a
mobile robot base, such as the ActiveMedia Pioneer series.  The proxy
supports both differential drive robots (which are capable of forward
motion and rotation) and omni-drive robots (which are capable of
forward, sideways and rotational motion). */

/** [Data] */

/** Position device data. */
typedef struct _playerc_position_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** Robot geometry in robot cs: pose gives the position and
      orientation, size gives the extent.  These values are filled in
      by playerc_position_get_geom(). */
  double pose[3];
  double size[2];

  /** Odometric pose (m, m, radians). */
  double px, py, pa;

  /** Odometric velocity (m, m, radians). */
  double vx, vy, va;

  /** Stall flag [0, 1]. */
  int stall;

} playerc_position_t;

/** [Methods] */

/** Create a position device proxy. */
playerc_position_t *playerc_position_create(playerc_client_t *client, int index);

/** Destroy a position device proxy. */
void playerc_position_destroy(playerc_position_t *device);

/** Subscribe to the position device */
int playerc_position_subscribe(playerc_position_t *device, int access);

/** Un-subscribe from the position device */
int playerc_position_unsubscribe(playerc_position_t *device);

/** Enable/disable the motors */
int playerc_position_enable(playerc_position_t *device, int enable);

/** Get the position geometry.  The writes the result into the proxy
    rather than returning it to the caller. */
int playerc_position_get_geom(playerc_position_t *device);

/** Set the target speed.  vx : forward speed (m/s).  vy : sideways
    speed (m/s); this field is used by omni-drive robots only.  va :
    rotational speed (radians/s).  All speeds are defined in the robot
    coordinate system. */
int playerc_position_set_cmd_vel(playerc_position_t *device,
                                 double vx, double vy, double va, int state);

/** Set the target pose (gx, gy, ga) is the target pose in the
    odometric coordinate system. */
int playerc_position_set_cmd_pose(playerc_position_t *device,
                                  double gx, double gy, double ga, int state);

/***************************************************************************
 ** end section
 **************************************************************************/


/***************************************************************************
 ** begin section position3d
 **************************************************************************/

/** [Synopsis] The {\tt position3d} proxy provides an interface to a
mobile robot base, such as the Segway RMP series.  The proxy
supports both differential drive robots (which are capable of forward
motion and rotation) and omni-drive robots (which are capable of
forward, sideways and rotational motion). */
```

```c
/** [Data] */

/** Position3d device data. */
typedef struct _playerc_position3d_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** Robot geometry in robot cs: pose gives the position3d and
      orientation, size gives the extent.  These values are filled in
      by playerc_position3d_get_geom(). */
  double pose[3];
  double size[2];

  /** Device position (m). */
  double pos_x, pos_y, pos_z;

  /** Device orientation (radians). */
  double pos_roll, pos_pitch, pos_yaw;

  /** Linear velocity (m/s). */
  double vel_x, vel_y, vel_z;

  /** Angular velocity (radians/sec). */
  double vel_roll, vel_pitch, vel_yaw;

  /** Stall flag [0, 1]. */
  int stall;

} playerc_position3d_t;

/** [Methods] */

/** Create a position3d device proxy. */
playerc_position3d_t *playerc_position3d_create(playerc_client_t *client, int index);

/** Destroy a position3d device proxy. */
void playerc_position3d_destroy(playerc_position3d_t *device);

/** Subscribe to the position3d device */
int playerc_position3d_subscribe(playerc_position3d_t *device, int access);

/** Un-subscribe from the position3d device */
int playerc_position3d_unsubscribe(playerc_position3d_t *device);

/** Enable/disable the motors */
int playerc_position3d_enable(playerc_position3d_t *device, int enable);

/** Get the position3d geometry.  The writes the result into the proxy
    rather than returning it to the caller. */
int playerc_position3d_get_geom(playerc_position3d_t *device);

/** Set the target speed.  vx : forward speed (m/s).  vy : sideways
    speed (m/s); this field is used by omni-drive robots only.  va :
    rotational speed (radians/s).  All speeds are defined in the robot
    coordinate system. */
int playerc_position3d_set_speed(playerc_position3d_t *device,
                                 double vx, double vy, double va);
```

```
/** Set the target pose (gx, gy, ga) is the target pose in the
    odometric coordinate system. */
int playerc_position3d_set_cmd_pose(playerc_position3d_t *device,
                                    double gx, double gy, double ga);

/***************************************************************************
 ** end section
 **************************************************************************/


/***************************************************************************
 ** begin section power
 **************************************************************************/

/** [Synopsis] The {\tt power} proxy provides an interface through
    which battery levels can be monitored. */

/** [Data] */

/** Power device data. */
typedef struct _playerc_power_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** Battery charge (volts). */
  double charge;

} playerc_power_t;

/** [Methods] */

/** Create a power device proxy. */
playerc_power_t *playerc_power_create(playerc_client_t *client, int index);

/** Destroy a power device proxy. */
void playerc_power_destroy(playerc_power_t *device);

/** Subscribe to the power device. */
int playerc_power_subscribe(playerc_power_t *device, int access);

/** Un-subscribe from the power device. */
int playerc_power_unsubscribe(playerc_power_t *device);

/***************************************************************************
 ** end section
 **************************************************************************/


/***************************************************************************
 ** begin section ptz
 **************************************************************************/

/** [Synopsis] The {\tt ptz} proxy provides an interface to pan-tilt
    units such as the Sony PTZ camera. */

/** [Data] */
/** PTZ device data. */
typedef struct _playerc_ptz_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** The current ptz pan and tilt angles.  pan : pan angle (+ve to
      the left, -ve to the right).  tilt : tilt angle (+ve upwrds, -ve
      downwards). */
  double pan, tilt;

  /** The current zoom value (field of view angle). */
  double zoom;

} playerc_ptz_t;

/** [Methods] */

/** Create a ptz proxy. */
playerc_ptz_t *playerc_ptz_create(playerc_client_t *client, int index);

/** Destroy a ptz proxy. */
void playerc_ptz_destroy(playerc_ptz_t *device);

/** Subscribe to the ptz device. */
int playerc_ptz_subscribe(playerc_ptz_t *device, int access);

/** Un-subscribe from the ptz device. */
int playerc_ptz_unsubscribe(playerc_ptz_t *device);

/** Set the pan, tilt and zoom values. */
int playerc_ptz_set(playerc_ptz_t *device, double pan, double tilt, double zoom);

/** Set the pan, tilt and zoom values. (and speed) */
int playerc_ptz_set_ws(playerc_ptz_t *device, double pan, double tilt, double zoom, double panspeed,

/***************************************************************************
 ** end section
 ***************************************************************************/


/***************************************************************************
 ** begin section sonar
 ***************************************************************************/

/** [Synopsis] The {\tt sonar} proxy provides an interface to the
sonar range sensors built into robots such as the ActiveMedia Pioneer
series. */

/** [Data] */

/** Sonar proxy data. */
typedef struct _playerc_sonar_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;
  /** Number of pose values. */
  int pose_count;

  /** Pose of each sonar relative to robot (m, m, radians).  This
      structure is filled by calling playerc_sonar_get_geom(). */
  double poses[PLAYERC_SONAR_MAX_SAMPLES][3];

  /** Number of points in the scan. */
  int scan_count;

  /** Scan data: range (m). */
  double scan[PLAYERC_SONAR_MAX_SAMPLES];

} playerc_sonar_t;

/** [Methods] */

/** Create a sonar proxy. */
playerc_sonar_t *playerc_sonar_create(playerc_client_t *client, int index);

/** Destroy a sonar proxy. */
void playerc_sonar_destroy(playerc_sonar_t *device);

/** Subscribe to the sonar device. */
int playerc_sonar_subscribe(playerc_sonar_t *device, int access);

/** Un-subscribe from the sonar device. */
int playerc_sonar_unsubscribe(playerc_sonar_t *device);

/** Get the sonar geometry.  The writes the result into the proxy
    rather than returning it to the caller. */
int playerc_sonar_get_geom(playerc_sonar_t *device);

/***************************************************************************
 ** end section
 ***************************************************************************/


/***************************************************************************
 ** begin section bumper
 ***************************************************************************/

/** [Synopsis] The {\tt bumper} proxy provides an interface to the
bumper sensors built into robots such as the RWI B21R. */

/** [Data] */

/** Bumper proxy data. */
typedef struct _playerc_bumper_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** Number of pose values. */
  int pose_count;

  /** Pose of each bumper relative to robot (mm, mm, deg, mm, mm).  This
      structure is filled by calling playerc_bumper_get_geom().
```

```
         values are x,y (of center) ,normal,length,curvature */
  double poses[PLAYERC_BUMPER_MAX_SAMPLES][5];

  /** Number of points in the scan. */
  int bumper_count;

  /** Bump data: unsigned char, either boolean or code indicating corner. */
  double bumpers[PLAYERC_BUMPER_MAX_SAMPLES];

} playerc_bumper_t;

/** [Methods] */

/** Create a bumper proxy. */
playerc_bumper_t *playerc_bumper_create(playerc_client_t *client, int index);

/** Destroy a bumper proxy. */
void playerc_bumper_destroy(playerc_bumper_t *device);

/** Subscribe to the bumper device. */
int playerc_bumper_subscribe(playerc_bumper_t *device, int access);

/** Un-subscribe from the bumper device. */
int playerc_bumper_unsubscribe(playerc_bumper_t *device);

/** Get the bumper geometry.  The writes the result into the proxy
    rather than returning it to the caller. */
int playerc_bumper_get_geom(playerc_bumper_t *device);

/***************************************************************************
 ** end section
 **************************************************************************/

/***************************************************************************
 ** begin section ir
 **************************************************************************/

/** [Synopsis] The {\tt ir} proxy provides an interface to the
ir sensors built into robots such as the RWI B21R. */

/** [Data] */



/** Ir proxy data. */
typedef struct _playerc_ir_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  // data
  player_ir_data_t ranges;

  // config
  player_ir_pose_t poses;

} playerc_ir_t;
```

```
/** [Methods] */

/** Create a ir proxy. */
playerc_ir_t *playerc_ir_create(playerc_client_t *client, int index);

/** Destroy a ir proxy. */
void playerc_ir_destroy(playerc_ir_t *device);

/** Subscribe to the ir device. */
int playerc_ir_subscribe(playerc_ir_t *device, int access);

/** Un-subscribe from the ir device. */
int playerc_ir_unsubscribe(playerc_ir_t *device);

/** Get the ir geometry.  The writes the result into the proxy
    rather than returning it to the caller. */
int playerc_ir_get_geom(playerc_ir_t *device);

/***************************************************************************
 ** end section
 ***************************************************************************/



/***************************************************************************
 ** begin section truth
 ***************************************************************************/

/** [Synposis] The {\tt truth} proxy can be used to get and set the
    pose of objects in the Stage simulator. */

/** [Data] */

/** Truth proxy data. */
typedef struct _playerc_truth_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** The object pose (world cs). */
  double px, py, pa;

} playerc_truth_t;


/** [Methods] */

/** Create a truth proxy. */
playerc_truth_t *playerc_truth_create(playerc_client_t *client, int index);

/** Destroy a truth proxy. */
void playerc_truth_destroy(playerc_truth_t *device);

/** Subscribe to the truth device. */
int playerc_truth_subscribe(playerc_truth_t *device, int access);

/** Un-subscribe from the truth device. */
int playerc_truth_unsubscribe(playerc_truth_t *device);
```

```
/** Get the object pose.
    px, py, pa : the pose (global coordinates). */
int playerc_truth_get_pose(playerc_truth_t *device, double *px, double *py, double *pa);

/** Set the object pose.  px, py, pa : the new pose (global
    coordinates). */
int playerc_truth_set_pose(playerc_truth_t *device, double px, double py, double pa);

/***************************************************************************
 ** end section
 ***************************************************************************/


/***************************************************************************
 ** begin section wifi
 ***************************************************************************/

/** [Synopsis] The {\tt wifi} proxy is used to query the state of a
wireless network.  It returns information such as the link quality and
signal strength of access points or of other wireless NIC's on an
ad-hoc network. */

/** [Data] */

/** Individual link info. */
typedef struct _playerc_wifi_link_t
{
  /** Destination IP address. */
  char ip[32];

  /** Link properties. */
  int qual, level, noise;

} playerc_wifi_link_t;


/** Wifi device proxy. */
typedef struct _player_wifi_t
{
  /** Device info; must be at the start of all device structures. */
  playerc_device_t info;

  /** A list containing info for each link. */
  int link_count;
  playerc_wifi_link_t links[PLAYERC_WIFI_MAX_LINKS];

} playerc_wifi_t;

/** [Methods] */

/** Create a wifi proxy. */
playerc_wifi_t *playerc_wifi_create(playerc_client_t *client, int index);

/** Destroy a wifi proxy. */
void playerc_wifi_destroy(playerc_wifi_t *device);

/** Subscribe to the wifi device. */
```

```
int playerc_wifi_subscribe(playerc_wifi_t *device, int access);

/** Un-subscribe from the wifi device. */
int playerc_wifi_unsubscribe(playerc_wifi_t *device);

/***************************************************************************
 ** end section
 ***************************************************************************/

#ifdef __cplusplus
}
#endif

#endif
```