Player/Stage project

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

Information Sciences Laboratory
HRL Laboratories
Malibu, California, USA

Player C++ Client Library

Version 1.5 Reference Manual

Brian P. Gerkey Richard T. Vaughan Andrew Howard

This document may not contain the most current documentation on Player. For the latest documentation, consult the Player homepage:
`http://playerstage.sourceforge.net/`

May 31, 2004

Contents

Chapter 1

Overview

The C++ client (`client_libs/c++`) is generally the most comprehensive library, since it is used to test new features as they are implemented in the server. It is also the most widely used client library and thus the best debugged. Having said that, this client is not perfect, but should be straightforward to use by anyone familiar with C++.

The C++ library is built on a "service proxy" model in which the client maintains local objects that are proxies for remote services. There are two kinds of proxies: the special server proxy `PlayerClient` and the various device-specific proxies. Each kind of proxy is implemented as a separate class. The user first creates a `PlayerClient` proxy and uses it to establish a connection to a Player server. Next, the proxies of the appropriate device-specific types are created and initialized using the existing `PlayerClient` proxy. To make this process concrete, consider the following simple example (for clarity, we omit some error-checking):

```
int main(int argc, char **argv)
{
  /* Connect to Player server on "localhost" at default port */
  PlayerClient robot("localhost");
  /* Request access to the ptz camera */
  PtzProxy zp(&robot,0,'a');

  int dir = 1;
  for(;;)
  {
    if(robot.Read())
      exit(1);

    // print out current camera state
    zp.Print();

    if(zp.pan > 80 || zp.pan < -80)
      dir = -dir;

    zp.SetCam(zp.pan + dir * 5,zp.tilt,zp.zoom);
  }

  // won't actually get here, but...
  robot.Disconnect();
}
```

This program will continuously pan the pan-tilt-zoom camera unit left and right[1]. First, a `PlayerClient` proxy is created, using the default constructor to connect to the server listening at `localhost:6665`. Next, a `PtzProxy` is created to control the camera. The constructor for this object uses the existing `PlayerClient` proxy to establish "all" (`'a'`) access to the 0th pan-tilt-zoom camera. Finally, we enter a simple loop that reads the current camera state and writes back a new state with the pan angle changed slightly.

With that simple example in mind, we now describe the full functionality of the C++ client library.

[1]Well, not exactly. A little extra code is required to handle the physical camera because it pans rather slowly; see `examples/c++/ptz.cc` for the details.

Chapter 2

Class Reference

2.1 PlayerClient

One `PlayerClient` object is used to control each connection to a Player server. Contained within this object are methods for changing the connection parameters and obtaining access to devices, which we explain next.

Attributes

```
bool fresh;
```

Flag set if data has just been read into this client. If you use it, you must set it to false yourself after examining the data.

```
char hostname[256];
```

The host of the Player server to which we are connected.

```
int port;
```

The port of the Player server to which we are connected.

```
struct timeval timestamp;
```

The latest time received from the server.

Methods

```
PlayerClient(const char* hostname=NULL,
const int port=PLAYER_PORTNUM,
const int protocol=PLAYER_TRANSPORT_TCP)
```

Make a client and connect it as indicated. If `hostname` is omitted (or NULL) then the client will *not* be connected. In that cast, call `Connect()` yourself later.

```
PlayerClient(const struct in_addr* hostaddr,
const int port,
const int protocol=PLAYER_TRANSPORT_TCP)
```

Make a client and connect it as indicated, using a binary IP instead of a hostname

```
int Connect(const char* hostname="localhost", int port=PLAYER_PORTNUM)
```

Connect to the indicated host and port.
Returns 0 on success; -1 on error.

```
int Connect(const struct in_addr* addr, int port)
```

Connect to the indicated host and port, using a binary IP.
Returns 0 on success; -1 on error.

```
int ConnectRNS(const char* robotname, const char* hostname="localhost",
int port=PLAYER_PORTNUM)
```

Connect to a robot, based on its name, by using the Player robot name service (RNS) on the indicated host and port. Returns 0 on success; -1 on error.

```
int Disconnect()
```

Disconnect from server.
Returns 0 on success; -1 on error.

```
bool Connected()
```

Check if we are connected.

```
int Read()
```

This method will read one round of data; that is, it will read until a SYNC packet is received from the server. Depending on which data delivery mode is in use, new data may or may not be received for each open device. The data that is received for each device device will be processed by the appropriate device proxy and stored there for access by your program. If no errors occurred 0 is returned. Otherwise, -1 is returned and diagnostic information is printed to `stderr` (you should probably close the connection!).

```
int Write(player_device_id_t device_id,
const char* command, size_t commandlen)
```

Write a command to the server. This method is **not** intended for direct use. Rather, device proxies should implement higher-level methods atop this one. Returns 0 on success, -1 otherwise.

```
int Request(player_device_id_t device_id,
const char* payload,
size_t payloadlen,
player_msghdr_t* replyhdr,
char* reply, size_t replylen)
```

Send a request to the server. This method is **not** intended for direct use. Rather, device proxies should implement higher-level methods atop this one. Returns 0 on success, -1 otherwise.

```
int Request(player_device_id_t device_id,
const char* payload,
size_t payloadlen)
```

Another form of Request(), this one can be used if the caller is not interested in the reply. This method is **not** intended for direct use. Rather, device proxies should implement higher-level methods atop this one. Returns 0 if an ACK is received, -1 otherwise.

```
int RequestDeviceAccess(player_device_id_t device_id,
unsigned char req_access,
unsigned char* grant_access,
char* driver_name = NULL,
int driver_name_len = 0)
```

Request access to a device; meant mostly for use by client-side device proxy constructors. `req_access` is requested access. `grant_access`, if non-NULL, will be filled with the granted access. Returns 0 if everything went OK or -1 if something went wrong.

```
int SetFrequency(unsigned short freq)
```

You can change the rate at which your client receives data from the server with this method. The value of `freq` is interpreted as Hz; this will be the new rate at which your client receives data (when in continuous mode). On error, -1 is returned; otherwise 0.

```
int SetDataMode(unsigned char mode)
```

You can toggle the mode in which the server sends data to your client with this method. The `mode` should be one of:

- `PLAYER_DATAMODE_PUSH_ALL` (all data at fixed frequency)
- `PLAYER_DATAMODE_PULL_ALL` (all data on demand)
- `PLAYER_DATAMODE_PUSH_NEW` (only new new data at fixed freq)
- `PLAYER_DATAMODE_PULL_NEW` (only new data on demand)

On error, -1 is returned; otherwise 0.

```
int RequestData()
```

When in a `PULL` data delivery mode, you can request a single round of data using this method. On error -1 is returned; otherwise 0.

```
int Authenticate(char* key)
```

Attempt to authenticate your client using the provided key. If authentication fails, the server will close your connection.

```
int LookupPort(const char* name)
```

Documentation on LookupPort goes here

2.2 PlayerMultiClient

The PlayerMultiClient makes it easy to control multiple Player connections within one thread. You can connect to any number of Player servers and read from all of them with a single `Read()`.

Attributes

Methods

`PlayerMultiClient()`

Uninteresting constructor.

`int GetNumClients(void)`

How many clients are currently being managed?

`PlayerClient* GetClient( char* host, int port )`

Return a pointer to the client associated with the given host and port, or NULL if none is connected to that address.

`PlayerClient* GetClient( struct in_addr* addr, int port )`

Return a pointer to the client associated with the given binary host and port, or NULL if none is connected to that address.

`void AddClient(PlayerClient* client)`

After creating and connecting a PlayerClient object, you should use this method to hand it over to the PlayerMultiClient for management.

`void RemoveClient(PlayerClient* client)`

Remove a client from PlayerMultiClient management.

`int Read()`

Read on one of the client connections. This method will return after reading from the server with first available data. It will **not** read data from all servers. You can use the `fresh` flag in each client object to determine who got new data. You should then set that flag to false. Returns 0 if everything went OK, -1 if something went wrong.

`int ReadLatest( int max_reads )`

Same as Read(), but reads everything off the socket so we end up with the freshest data, subject to N maximum reads.

2.3 ClientProxy

Base class for all proxy devices. Access to a device is provided by a device-specific proxy class. These classes all inherit from the `ClientProxy` class which defines an interface for device proxies. As such, a few methods are common to all devices and we explain them here.

Attributes

```
PlayerClient* client;
```

The controlling client object.

```
bool valid;
```

Have we yet received any data from this device?

```
char driver_name[PLAYER_MAX_DEVICE_STRING_LEN]; // driver in use
```

The name of the driver used to implement this device in the server.

```
struct timeval timestamp;
```

Time at which this data was generated by the device.

```
struct timeval senttime;
```

Time at which this data was sent by the server.

```
struct timeval receivedtime;
```

Time at which this data was received by the client.

Methods

```
ClientProxy(PlayerClient* pc,
unsigned short req_device,
unsigned short req_index,
unsigned char req_access = 'c')
```

This constructor will try to get access to the device, unless `req_device` is 0 or `req_access` is 'c'. The pointer `pc` must refer to an already connected `PlayerClient` proxy. The `index` indicates which one of the devices to use (usually 0). Note that a request executed by this the constructor can fail, but the constructor cannot indicate the failure. Thus, if you request a particular access mode, you should verify that the current access is identical to your requested access using `GetAccess()`. In any case, you can use `ChangeAccess()` later to change your access mode for the device.

```
unsigned char GetAccess()
```

Returns the current access mode for the device.

```
int ChangeAccess(unsigned char req_access,
unsigned char* grant_access=NULL )
```

Request different access for the device. If `grant_access` is non-NULL, then it is filled in with the granted access. Returns 0 on success, -1 otherwise.

`int Close()`

Convenience method for requesting 'c' access.

`virtual void FillData(player_msghdr_t hdr, const char* buffer)`

All proxies must provide this method. It is used internally to parse new data when it is received.

`void StoreData(player_msghdr_t hdr, const char* buffer)`

This method is used internally to keep a copy of the last message from the device *

`virtual void Print()`

All proxies SHOULD provide this method, which should print out, in a human-readable form, the device's current state.

2.4 BlobfinderProxy

The `BlobfinderProxy` class is used to control a `blobfinder` device. It contains no methods. The latest color blob data is stored in `blobs`, a dynamically allocated 2-D array, indexed by color channel.

Attributes

`unsigned short width, height;`

Dimensions of the camera image, in pixels

`char num_blobs[PLAYER_BLOBFINDER_MAX_CHANNELS];`

Array containing the number of blobs detected on each channel

`Blob* blobs[PLAYER_BLOBFINDER_MAX_CHANNELS];`

Array containing arrays of the latest blob data. Each blob contains the following information:

- `unsigned int color` (in packed RGB)
- `unsigned int area` (blob area, in square pixels)
- `unsigned short x, y` (blob center, in pixels)
- `unsigned short left, right, top, bottom` (blob bounding box, in pixels)
- `double range` (range to blob center, in m)

For example, to access the area of the 0^{th} blob on channel 2, you would refer to: `blobs[2][0].area`.

Methods

```
BlobfinderProxy(PlayerClient* pc, unsigned short index,
unsigned char access='c')
```

Constructor. Leave the access field empty to start unconnected.

`void Print()`

Print out current blob information.

2.5 BumperProxy

The `BumperProxy` class is used to read from a `bumper` device.

Attributes

```
uint8_t bumper_count;
uint8_t bumpers[PLAYER_BUMPER_MAX_SAMPLES];
```

array representing bumped state.

Methods

```
BumperProxy (PlayerClient* pc, unsigned short index,
unsigned char access = 'c')
```

Constructor. Leave the access field empty to start unconnected.

```
bool Bumped (const unsigned int i)
```

Returns 1 if the specified bumper has been bumped, 0 otherwise.

```
bool BumpedAny ()
```

Returns 1 if any bumper has been bumped, 0 otherwise.

```
int GetBumperGeom( player_bumper_geom_t* bumper_defs )
```

Requests the geometries of the bumpers. Returns -1 if anything went wrong, 0 if OK

```
uint8_t BumperCount ()
```

Returns the number of bumper readings.

```
void Print ()
```

Print out the current bumper state.

2.6 CommsProxy

The `CommsProxy` class controls a `comms` device. Data may be written one message at a time using the `Write` method. Incoming data are stored in a set of parallel lists:

- `uint8_t** msg` : list of pointers to message data
- `size_t* msg_len` : list of message lengths
- `struct timeval* msg_ts` : list of message timestamps

The current number of valid messages is stored in `msg_num`, and so each list should be considered to be of length `msg_num`. To delete a message from the lists, use `Delete()`.

Attributes

`uint8_t** msg;`

List of received messages.

`size_t* msg_len;`

List of lengths of the received messages.

`struct timeval* msg_ts;`

List of timestamps of the received messages.

`size_t msg_num;`

Number of received messages

Methods

```
CommsProxy(PlayerClient* pc, unsigned short index,
unsigned char access ='c')
```

Proxy constructor. Leave the access field empty to start unconnected.

`int Write(void *msg, int len)`

Write a message to the outgoing queue. Returns the number of bytes written, or -1 on error.

`int Delete(int index)`

Delete the given message. Returns 0 on success and -1 on error.

`void Print()`

Print out current message.

Attributes

`unsigned short count;`

The number of beacons detected

`double pose[3];`

The pose of the sensor [x,y,theta] in [m,m,rad]

`double size[2];`

The size of the sensor [x,y] in [m,m]

`double fiducial_size[2];`

The size of the most recently detected fiducial

`double min_range;`

the minimum range of the sensor in meters (partially defines the FOV)

`double max_range;`

the maximum range of the sensor in meters (partially defines the FOV)

`double view_angle;`

the receptive angle of the sensor in degrees (partially defines the FOV)

`FiducialItem beacons[PLAYER_FIDUCIAL_MAX_SAMPLES];`

The latest laser beacon data. Each beacon has the following information:

- int id (-1 for unidentified)
- double pose[3] (pose of the beacon)
- double upose[3] (uncertainty in the pose of the beacon)

Where each pose array is composed of:

- range (m)
- bearing (radians)
- orient (radians)

Methods

```
FiducialProxy(PlayerClient* pc, unsigned short index,
unsigned char access='c')
```

Constructor. Leave the access field empty to start unconnected.

```
void Print()
```

Print out latest beacon data.

```
int PrintFOV()
```

Print the latest FOV configuration

```
int PrintGeometry()
```

Print the latest geometry configuration

```
int GetConfigure()
```

Get the sensor's geometry configuration

```
int GetFOV()
```

Get the field of view

2.7 GpsProxy

The `GpsProxy` class is used to control a `gps` device. The latest pose data is stored in three class attributes.

Attributes

```
double latitude;
double longitude;
```

Latitude and longitude, in degrees.

```
double altitude;
```

Altitude, in meters.

```
int satellites;
```

Number of satellites in view.

```
int quality;
```

Fix quality

```
double hdop;
```

Horizontal dilution of position (HDOP)

```
struct timeval time;
```

Time, since the epoch

Methods

```
GpsProxy(PlayerClient* pc, unsigned short index,
unsigned char access='c')
```

Constructor. Leave the access field empty to start unconnected.

```
void Print()
```

Print out current pose information.

2.8 GripperProxy

The `GripperProxy` class is used to control a `gripper` device. The latest gripper data held in a handful of class attributes. A single method provides user control.

Attributes

```
unsigned char state,beams;
```

The latest raw gripper data.

```
bool outer_break_beam,inner_break_beam,
paddles_open,paddles_closed,paddles_moving,
gripper_error,lift_up,lift_down,lift_moving,
lift_error;
```

These boolean variables indicate the state of the gripper

Methods

```
GripperProxy(PlayerClient* pc, unsigned short index,
unsigned char access='c')
```

The client calls this method to make a new proxy. Leave access empty to start unconnected.

```
int SetGrip(unsigned char cmd, unsigned char arg=0)
```

Send a gripper command. Look in the Player user manual for details on the command and argument. Returns 0 if everything's ok, and -1 otherwise (that's bad).

```
void Print()
```

Print out current gripper state.

2.9 IRProxy

The `IRProxy` class is used to control an `ir` device. Right now, it is particular to the `reb_ir` driver.

Attributes

`unsigned short ranges[PLAYER_IR_MAX_SAMPLES];`

Latest range readings

`unsigned short voltages[PLAYER_IR_MAX_SAMPLES];`

Latest voltage readings

`double stddev[PLAYER_IR_MAX_SAMPLES];`

Standard deviations

`double params[PLAYER_IR_MAX_SAMPLES][2];`

Distance regression params

`double sparams[PLAYER_IR_MAX_SAMPLES][2];`

Standard deviation regression params

`player_ir_pose_t ir_pose;`

Poses of the IRs. Contains:

- short poses[PLAYER_IR_MAX_SAMPLES][3];

Where each pose element contains: (x,y,theta) in (mm,mm,degrees).

Methods

```
IRProxy(PlayerClient *pc, unsigned short index,
unsigned char access = 'c')
```

Constructor. Leave the access field empty to start unconnected.

`int SetIRState(unsigned char state)`

Enable/disable the IRs.

`int GetIRPose()`

Request the poses of the IRs.

`void SetRangeParams(int which, double m, double )`

Set range parameters.

`void SetStdDevParams(int which, double m, double b)`

Set standard deviation parameters.

`double CalcStdDev(int w, unsigned short range)`

Calculate standard deviations.

`unsigned short operator [](unsigned int index)`

Range access operator. This operator provides an alternate way of access the range data. For example, given a `IRProxy` named `ip`, the following expressions are equivalent: `ip.ranges[0]` and `ip[0]`.

`void Print()`

Print out current IR data.

2.10 LaserProxy

The `LaserProxy` class is used to control a `laser` device. The latest scan data is held in two arrays: `ranges` and `intensity`. The laser scan range, resolution and so on can be configured using the `Configure()` method.

Attributes

`int scan_count;`

Number of points in scan

`double scan_res;`

Angular resolution of scan (radians)

`double min_angle, max_angle;`

Scan range for the latest set of data (radians)

`double range_res;`

Range resolution of scan (mm)

`bool intensity;`

Whether or not reflectance (i.e., intensity) values are being returned.

`double scan[PLAYER_LASER_MAX_SAMPLES][2];`

Scan data (polar): range (m) and bearing (radians)

`double point[PLAYER_LASER_MAX_SAMPLES][2];`

Scan data (Cartesian): x,y (m)

`unsigned char intensities[PLAYER_LASER_MAX_SAMPLES];`

The reflected intensity values (arbitrary units in range 0-7).

Methods

```
LaserProxy(PlayerClient* pc, unsigned short index,
unsigned char access='c')
```

Constructor. Leave the access field empty to start unconnected.

`int SetLaserState(const unsigned char state)`

Enable/disable the laser. Set `state` to 1 to enable, 0 to disable. Note that when laser is disabled the client will still receive laser data, but the ranges will always be the last value read from the laser before it was disabled. Returns 0 on success, -1 if there is a problem.
Note: The `sicklms200` driver currently does not implement this feature.

```
int Configure(double min_angle,
double max_angle,
unsigned int scan_res,
unsigned int range_res,
bool intensity)
```

Configure the laser scan pattern. Angles `min_angle` and `max_angle` are measured in radians. `scan_res` is measured in units of $0.01°$; valid values are: 25 ($0.25°$), 50 ($0.5°$) and $100(1°)$. `range_res` is measured in mm; valid values are: 1, 10, 100. Set `intensity` to `true` to enable intensity measurements, or `false` to disable. Returns the 0 on success, or -1 of there is a problem.

```
int GetConfigure()
```

Get the current laser configuration; it is read into the relevant class attributes. Returns the 0 on success, or -1 of there is a problem.

```
int RangeCount()
```

Get the number of range/intensity readings.

```
double Ranges (int index)
```

An alternate way to access the range data.

```
double operator [] (unsigned int index)
```

Range access operator. This operator provides an alternate way of access the range data. For example, given an `LaserProxy` named `lp`, the following expressions are equivalent: `lp.ranges[0]`, `lp.Ranges(0)`, and `lp[0]`.

```
void Print()
```

Print out the current configuration and laser range/intensity data.

```
void PrintConfig()
```

Print out the current configuration

2.11 LocalizeProxy

The `LocalizeProxy` class is used to control a `localize` device, which can provide multiple pose hypotheses for a robot.

Attributes

`unsigned int map_size_x, map_size_y;`
Map dimensions (cells)

`double map_scale;`
Map scale (m/cell)

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

`int pending_count;`
Number of pending (unprocessed) sensor readings

`int hypoth_count;`
Number of possible poses

`localize_hypoth hypoths[PLAYER_LOCALIZE_MAX_HYPOTHS];`
Array of possible poses. Each pose contains the following information:

- `double mean[3]` (pose estimate, in m, m, radians)
- `double cov[3][3]` (covariance, in m^2 and $radians^2$)
- `double weight` (weight associated with this estimate)

Methods

`LocalizeProxy(PlayerClient* pc, unsigned short index,`
`unsigned char access = 'c')`
Constructor. Leave the access field empty to start unconnected.

`int SetPose(double pose[3], double cov[3][3])`
Set the current pose hypothesis (m, m, radians). Returns 0 on success, -1 on error.

`int GetNumParticles()`
Get the number of particles (for particle filter-based localization systems). Returns the number of particles, or -1 on error.

`int GetMap()`
Get the map from the server. It's stored in map_size_x, map_size_y, map_scale, and map_cells. Returns 0 on success, -1 on error.

`void Print()`
Print out current hypotheses.

2.12 MComProxy

The `MComProxy` class is used to exchange data with other clients connected with the same server, through a set of named "channels". For some useful (but optional) type and constant definitions that you can use in your clients, see <playermcomtypes.h>.

Attributes

```
player_mcom_data_t data;
int type;
char channel[MCOM_CHANNEL_LEN];
```

These members contain the results of the last command. Note: It's better to use the LastData() method.

Methods

`int Pop(int type, char channel[MCOM_CHANNEL_LEN])`

Read and remove the most recent buffer in 'channel' with type 'type'. The result can be read with LastData() after the next call to PlayerClient::Read(). @return 0 if no error @return -1 on error, the channel does not exist, or the channel is empty.

`int Read(int type, char channel[MCOM_CHANNEL_LEN])`

Read the most recent buffer in 'channel' with type 'type'. The result can be read with LastData() after the next call to PlayerClient::Read(). @return 0 if no error @return -1 on error, the channel does not exist, or the channel is empty.

`int Push(int type, char channel[MCOM_CHANNEL_LEN], char dat[MCOM_DATA_LEN])`

Push a message 'dat' into channel 'channel' with message type 'type'.

`int Clear(int type, char channel[MCOM_CHANNEL_LEN])`

Clear all messages of type 'type' on channel 'channel'

`int SetCapacity(int type, char channel[MCOM_CHANNEL_LEN], unsigned char cap)`

Set the capacity of the buffer using 'type' and 'channel' to 'cap'. Note that 'cap' is an unsigned char and must be < MCOM_N_BUFS

`char* LastData()`

Get the results of the last command (Pop or Read). Call PlayerClient::Read() before using.

`int LastMsgType()`

Get the results of the last command (Pop or Read). Call PlayerClient::Read() before using.

`char* LastChannel()`

Get the channel of the last command (Pop or Read). Call PlayerClient::Read() before using.

2.13 PositionProxy

The `PositionProxy` class is used to control a `position` device. The latest position data is contained in the attributes `xpos, ypos`, etc.

Attributes

`double xpos,ypos,theta;`

Robot pose (according to odometry) in m, m, radians.

`double speed, sidespeed, turnrate;`

Robot speeds in m/sec, m/sec, radians/sec.

`unsigned char stall;`

Stall flag: 1 if the robot is stalled and 0 otherwise.

Methods

```
PositionProxy(PlayerClient* pc, unsigned short index,
unsigned char access ='c')
```

Constructor. Leave the access field empty to start unconnected.

`int SetSpeed(double speed, double sidespeed, double turnrate)`

Send a motor command for velocity control mode. Specify the forward, sideways, and angular speeds in m/sec, m/sec, and radians/sec, respectively. Returns: 0 if everything's ok, -1 otherwise.

`int SetSpeed(double speed, double turnrate)`

Same as the previous SetSpeed(), but doesn't take the sideways speed (so use this one for non-holonomic robots).

`int GoTo(double x, double y, double t)`

Send a motor command for position control mode. Specify the desired pose of the robot in m, m, radians. Returns: 0 if everything's ok, -1 otherwise.

`int SetMotorState(unsigned char state)`

Enable/disable the motors. Set `state` to 0 to disable or 1 to enable. Be VERY careful with this method! Your robot is likely to run across the room with the charger still attached. Returns: 0 if everything's ok, -1 otherwise.

`int SelectVelocityControl(unsigned char mode)`

Select velocity control mode. For the the p2os_position driver, set `mode` to 0 for direct wheel velocity control (default), or 1 for separate translational and rotational control.
For the reb_position driver: 0 is direct velocity control, 1 is for velocity-based heading PD controller (uses DoDesiredHeading()).
Returns: 0 if everything's ok, -1 otherwise.

`int ResetOdometry()`

Reset odometry to (0,0,0). Returns: 0 if everything's ok, -1 otherwise.

`int SelectPositionMode(unsigned char mode)`

Select position mode on the reb_position driver. Set `mode` for 0 for velocity mode, 1 for position mode. Returns: 0 if OK, -1 else

`int SetOdometry(double x, double y, double t)`

Sets the odometry to the pose `(x, y, theta)`. Note that `x` and `y` are in m and `theta` is in radians. Returns: 0 if OK, -1 else

`int SetSpeedPID(int kp, int ki, int kd)`

Only supported by the reb_position driver.

`int SetPositionPID(short kp, short ki, short kd)`

Only supported by the reb_position driver.

`int SetPositionSpeedProfile(short spd, short acc)`

Only supported by the reb_position driver.

`int DoStraightLine(int mm)`

Only supported by the reb_position driver.

`int DoRotation(int deg)`

Only supported by the reb_position driver.

`int DoDesiredHeading(int theta, int xspeed, int yawspeed)`

Only supported by the reb_position driver.

`int SetStatus(uint8_t cmd, uint16_t value)`

Only supported by the segwayrmp driver

`int PlatformShutdown()`

Only supported by the segwayrmp driver

`double  Xpos ()`

Accessor method

`double  Ypos ()`

Accessor method

`double Theta ()`

Accessor method

`double  Speed ()`

Accessor method

`double  SideSpeed ()`

Accessor method

`double  TurnRate ()`

Accessor method

`unsigned char Stall ()`

Accessor method

`void Print()`

Print current position device state.

2.14 Position3DProxy

The `Position3DProxy` class is used to control a `position3d` device. The latest position data is contained in the attributes `xpos, ypos`, etc.

Attributes

```
double xpos,ypos,zpos;
double roll,pitch,yaw;
```

Robot pose (according to odometry) in m, radians.

```
double xspeed, yspeed, zspeed;
double rollspeed, pitchspeed, yawspeed;
```

Robot speeds in m/sec, rad/sec

```
unsigned char stall;
```

Stall flag: 1 if the robot is stalled and 0 otherwise.

Methods

```
Position3DProxy(PlayerClient* pc, unsigned short index,
unsigned char access ='c')
```

Constructor. Leave the access field empty to start unconnected.

```
int SetSpeed(double xspeed, double yspeed, double yawspeed)
```

Send a motor command for a planar robot. Specify the forward, sideways, and angular speeds in mm/s, mm/s, and degrees/sec, respectively. Returns: 0 if everything's ok, -1 otherwise.

```
int SetSpeed(double xspeed, double yawspeed)
```

Same as the previous SetSpeed(), but doesn't take the sideways speed (so use this one for non-holonomic robots).

```
int SetMotorState(unsigned char state)
```

Enable/disable the motors

```
void Print()
```

Print current position device state.

```
double  Xpos()
```

Accessor method

```
double  Ypos()
```

Accessor method

```
double  Zpos()
```

Accessor method

`double  Roll()`
Accessor method

`double  Pitch()`
Accessor method

`double  Yaw()`
Accessor method

`double  XSpeed()`
Accessor method

`double  YSpeed()`
Accessor method

`double  ZSpeed()`
Accessor method

`double  RollSpeed()`
Accessor method

`double  PitchSpeed()`
Accessor method

`double  YawSpeed()`
Accessor method

`unsigned char Stall ()`
Accessor method

2.15 PowerProxy

The `PowerProxy` class controls a `power` device.

Attributes

Methods

```
PowerProxy (PlayerClient* pc, unsigned short index,
unsigned char access ='c')
```

Constructor. Leave the access field empty to start unconnected.

```
double Charge ()
```

Returns the current charge.

```
void Print ()
```

Print the current data.

2.16 PtzProxy

The `PtzProxy` class is used to control a `ptz` device. The state of the camera can be read from the `pan, tilt, zoom` attributes and changed using the `SetCam()` method.

Attributes

```
double pan, tilt, zoom;
```

Pan, tilt, and field of view values (all radians).

```
double panspeed, tiltspeed;
```

Pan and tilt speeds (rad/sec)

Methods

```
PtzProxy(PlayerClient* pc, unsigned short index,
unsigned char access='c')
```

Constructor. Leave the access field empty to start unconnected.

```
int SetCam(double pan, double tilt, double zoom)
```

Change the camera state. Specify the new `pan`, `tilt`, and `zoom` values (all degrees). Returns: 0 if everything's ok, -1 otherwise.

```
int SetSpeed(double panspeed, double tiltspeed)
```

Specify new target velocities

```
int SendConfig(uint8_t *bytes, size_t len, uint8_t *reply = NULL,
size_t reply_len = 0)
```

Send a camera-specific config

```
int SelectControlMode(uint8_t mode)
```

Select new control mode. Use either PLAYER_PTZ_POSITION_CONTROL or PLAYER_PTZ_VELOCITY_CONTROL.

```
void Print()
```

Print out current ptz state.

2.17 SonarProxy

The `SonarProxy` class is used to control a `sonar` device. The most recent sonar range measuremts can be read from the `range` attribute, or using the the `[]` operator.

Attributes

```
unsigned short range_count;
```

The number of sonar readings received.

```
double ranges[PLAYER_SONAR_MAX_SAMPLES];
```

The latest sonar scan data. Range is measured in m.

```
int pose_count;
```

Number of valid sonar poses

```
double poses[PLAYER_SONAR_MAX_SAMPLES][3];
```

Sonar poses (m,m,radians)

Methods

```
SonarProxy(PlayerClient* pc, unsigned short index,
unsigned char access = 'c')
```

Constructor. Leave the access field empty to start unconnected.

```
int SetSonarState(unsigned char state)
```

Enable/disable the sonars. Set `state` to 1 to enable, 0 to disable. Note that when sonars are disabled the client will still receive sonar data, but the ranges will always be the last value read from the sonars before they were disabled.
Returns 0 on success, -1 if there is a problem.

```
int GetSonarGeom()
```

Request the sonar geometry.

```
double operator [](unsigned int index)
```

Range access operator. This operator provides an alternate way of access the range data. For example, given a `SonarProxy` named `sp`, the following expressions are equivalent: `sp.ranges[0]` and `sp[0]`.

```
void Print()
```

Print out current sonar range data.

2.18 SoundProxy

The `SoundProxy` class is used to control a `sound` device, which allows you to play pre-recorded sound files on a robot.

Attributes

Methods

```
SoundProxy(PlayerClient* pc, unsigned short index,
unsigned char access='c')
```

The client calls this method to make a new proxy. Leave access empty to start unconnected.

```
int Play(int index)
```

Play the sound indicated by the index. Returns 0 on success, -1 on error.

```
void Print()
```

Does nothing.

2.19 SpeechProxy

The `SpeechProxy` class is used to control a `speech` device. Use the `say` method to send things to say.

Attributes

Methods

```
SpeechProxy(PlayerClient* pc, unsigned short index,
unsigned char access='c')
```

Constructor. Leave the access field empty to start unconnected.

```
int Say(char* str)
```

Send a phrase to say. The phrase is an ASCII string. Returns the 0 on success, or -1 of there is a problem.

2.20 TruthProxy

The `TruthProxy` gets and sets the *true* pose of a truth device [worldfile tag: truth()]. This may be different from the pose returned by a device such as GPS or Position. If you want to log what happened in an experiment, this is the device to use. Setting the position of a truth device moves its parent, so you can put a truth device on robot and teleport it around the place.

Attributes

```
double x, y, a;
```

These vars store the current device pose (x,y,a) as (m,m,radians). The values are updated at regular intervals as data arrives. You can read these values directly but setting them does NOT change the device's pose!. Use `TruthProxy::SetPose()` for that.

Methods

```
TruthProxy(PlayerClient* pc, unsigned short index,
unsigned char access = 'c')
```

Constructor. Leave the access field empty to start unconnected.

```
void Print()
```

Print out current pose info in a format suitable for data logging.

```
int GetPose( double *px, double *py, double *pa )
```

Query Player about the current pose - requests the pose from the server, then fills in values for the arguments (m,m,radians). Usually you'll just read the `x,y,a` attributes but this function allows you to get pose direct from the server if you need too. Returns 0 on success, -1 if there is a problem.

```
int SetPose( double px, double py, double pa )
```

Request a change in pose (m,m,radians). Returns 0 on success, -1 if there is a problem.

```
int SetPoseOnRoot( double px, double py, double pa )
```

???

```
int GetFiducialID( int16_t* id )
```

Request the value returned by a fiducialfinder (and possibly a foofinser, depending on its mode), when detecting this object.

```
int SetFiducialID( int16_t id )
```

Set the value returned by a fiducialfinder (and possibly a foofinser, depending on its mode), when detecting this object.

2.21 WiFiProxy

The `WiFiProxy` class controls a `wifi` device.

Attributes

```
int link_count;
player_wifi_link_t links[PLAYER_WIFI_MAX_LINKS];
uint32_t throughput;
uint8_t op_mode;
int32_t bitrate;
uint16_t qual_type, maxqual, maxlevel, maxnoise;
```

The current wifi data.

Methods

```
WiFiProxy(PlayerClient *pc, unsigned short index,
unsigned char access = 'c')
```

Constructor. Leave the access field empty to start unconnected.

```
void Print()
```

Print out current data.