Robotics Laboratory
Stanford University
Stanford, California, USA

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

Autonomy Laboratory
Simon Frazer University
Vancouver, British Columbia, Canada

Player/Stage project

Player

Version 1.5 User Manual

Brian P. Gerkey Richard T. Vaughan Andrew Howard

This document may not contain the most current documentation on Player. For the latest documentation, visit the Player/Stage project online:
`http://playerstage.sourceforge.net`

June 2, 2004

Contents

List of Figures

List of Tables

Chapter 0

Metadata

0.1 How to Read this Manual

We know that you are dying to read this entire document, but let us give you some advice that may save you some time. If you are only planning to use Player with the Stage simulator, then you should only need to read Chapters 1 and 3 and the manual appropriate to the language in which you will write your programs. For example, if you plan to use the C++ client utilities, then read *Player C++ Client Library Reference Manual*. The manual that you are currently reading only includes documentation for the C reference client. Language-specific manuals are provided with the distribution (and are available from the homepage) for C++, Tcl, and LISP. Client libraries have also been contributed by users for other languages, including Python, Java, and Visual C++. These other client libraries are distributed separately; see the contributed clients page for details:

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

If you intend to use Player with physical hardware, then you should also read Chapters 2 & 4 and consult Chapter 7 in order to familiarize yourself with the details of connecting the hardware and telling Player where it is. If you are interested in modifying an existing client library or writing your own, then you should also read Chapters 5 & 6, which describe the message protocol and data formats between client and server. Finally, if you want to hack on the server in any way (e.g., add a new device driver, write your own Player server for a different language/platform), then you should also read Chapter 8, which will (hopefully) provide all the information that you will need.

0.2 A Note on Versions

This document describes the Player robot server version 1.5. It applies to Player versions >= 1.5 (at least until we put out another manual). Since the previous version, many things have likely changed. Thus, this manual does *not* apply to older versions of Player. You can find an older manual at the Player homepage:

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

In addition to providing access to physical hardware, Player is also the interface to the robot simulators Stage and Gazebo. The three software packages evolve independently, and so their version numbers have no meaningful correspondence (it used to be the case that Player and Stage versions were matched exactly; this is no longer true). For information on which versions of Player and the simulators are compatible, see the online FAQ: `http://playerstage.sourceforge.net/faq.html`

Chapter 1

Introduction

Player was originally developed by Brian Gerkey and Kasper Støy; the Stage simulator interface was originally written by Richard T. Vaughan and Andrew Howard. Gazebo and its Player interface were written by Nate Koenig and Andrew Howard. Now a part of the larger Player/Stage project, Player's development is administered by Brian Gerkey, Andrew Howard, and Richard T. Vaughan.

Player has historically been developed primarily at the Robotics Research Lab of the University of Southern California, but the current state of Player is the result of contributions from a great many developers and users in unversities, companies, and government labs around the world. For a list of significant contributors, see:

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

For a list of known users, see:

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

We have tried to make this documentation as complete as possible. Hopefully there is sufficient information here for you to use Player and the provided clients as well as write your clients in your language of choice.

What do you like? What do you hate? How do you use it? Questions and comments regarding Player should be directed to our mailing lists:

`http://sourceforge.net/mail/?group_id=42445`

and bug / feature request tracker:

`http://sourceforge.net/tracker/?group_id=42445`

1.1 License

The included Player client libraries are also simultaneously released under the GNU Lesser General Public License (GNU LGPL):

> 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

1.2 Description

What is Player? Player is a robot device server. It gives you simple and complete control over the physical sensors and actuators on your mobile robot. When Player is running on your robot, your client control program connects to it via a standard TCP socket, and communication is accomplished by the sending and receiving of some of a small set of simple messages.

Player is designed to be language and platform independent. Your client program can run on any machine that has network connectivity to your robot, and it can be written in any language that can open and control a TCP socket. Client-side utilities are currently available in C, C++, Tcl, LISP, Java, and Python. Further, Player makes no assumptions about how you might want to structure your robot control programs. In this way, it is much more "minimal" than other robot interfaces. If you want your client to be a highly concurrent multi-threaded program, write it like that. If you like a simple read-think-act loop, do that. If you like to control your robot interactively, try our Tcl client (or write your own client utilities in your favorite interactive language).

Player is also designed to support virtually any number of clients. Have you ever wanted your robots to "see" through each others' eyes? Now they can. Any client can connect to and read sensor data from (and even write motor commands to) any instance of Player on any robot. Aside from distributed sensing for control, you can also use Player for monitoring of experiments. For example, while your C++ client controls a robot, you can run a Tk GUI client elsewhere that shows you current sensor data, and a Python client that logs data for later analysis. Also, on-the-fly device requests allow your clients to gain access to different sensors and actuators as needed for the task at hand.

In addition to controlling the physical hardware, Player can be used to interface to the robot simulators Stage and Gazebo.

Last but not least, Player is Free Software. If you don't like how something works, change it. And please send us your patch!

Example of Player Operation

As a simple example of the use of Player, consider Figure 1.1 (note that for clarity, we leave out several protocol-level interactions). The server is executing locally on the computer to which the devices of interest are connected. In many cases, this computer is the robot itself, but it could also be, for example, a desktop machine attached to a SICK laser range-finder. The client can execute anywhere that has network connectivity to the machine hosting the server.

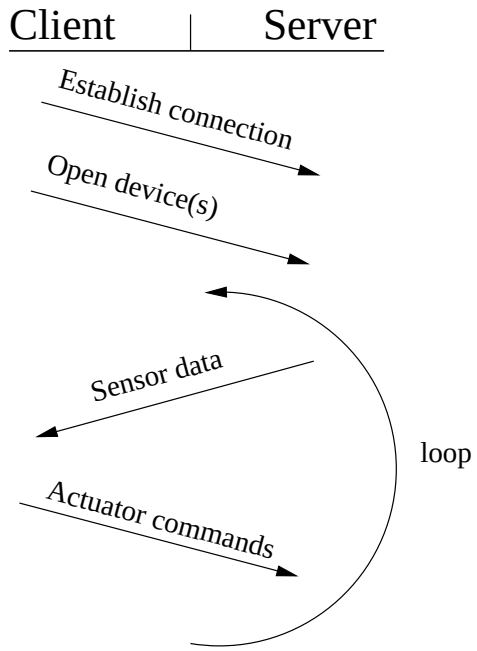


Figure 1.1: Example client/server interaction

First, the client establishes a TCP socket connection to the server. The client next sends some messages to the server to *open* the devices in which the client is interested. After that, the server continuously feeds data from those devices to the client, and the client exerts control by sending appropriate commands back to the server. Very simple.

1.3 System Requirements

Player was originally developed on x86/Linux systems, and it is still used primarily on that platform. However, with the help of the GNU Autotools (which rule), Player now builds and runs on most POSIX platforms, including a number of cross-compiler configurations. A notable exception is that Player does **not** run on Windows, and we have **no** plans to port it to Windows (however, a Cygwin port should be doable; if you're interested in doing this, let us know). The current status of Player for the platforms where it is known to build and run can be found on the project website. If you have an addition or correction, please let us know.

To build Player "out of the box", you need one of the supported systems and a recent version of GNU `gcc/g++` (we use some GNU extensions to C, so other compilers will **not** work).

1.4 Getting Player

The Player homepage is:

`http://playerstage.sourceforge.net`

Check there for the latest versions of the server and this document.

1.5 Bugs

Depite our diligent testing, Player is bound to contain some bugs. If you manage to break something, or if some aspect of Player's behavior seems wrong or non-intuitive, let us know.

To report a bug or request a feature, **please** do **not** send mail directly to the developers. Instead, use the tracking facilities provided at SourceForge; you can find a link on the Player homepage (see Section 1.4). Include as much information as possible, including at least Player version and OS version. A detailed description of what happened will enable us (hopefully) to repeat and analyze the problem.

1.6 On the Name Player

Player was originally named Golem, and the Stage simulator was originally named Arena. However, we soon discovered that many, many, many pieces of robotics-related software already use those names. So, we had to make a change. We needed names that capture the now-integral relationship between the server and simulator, so we chose Player and Stage, as suggested by a living Englishman, in reference to a very dead Englishman.

From *As You Like It* Act II, Scene 7:

> "All the world's a stage,
> And all the men and women merely players:
> They have their exits and their entrances;
> And one man in his time plays many parts,"

From *Macbeth* Act V, Scene 5:

> "Life's but a walking shadow, a poor player
> That struts and frets his hour upon the stage
> And then is heard no more: it is a tale
> Told by an idiot, full of sound and fury,
> Signifying nothing."

1.7 Acknowledgements

This work is supported in part by the Intel Foundation, DARPA grant DABT63-99-1-0015 (MARS), NSF grant ANI-9979457 (SCOWR), DARPA contract DAAE07-98-C-L028 (TMR), ONR Grants N00014-00-1-0140 and N0014-99-1-0162, and JPL Contract No. 1216961.

We thank the many developers and users who have contributed so much to the success of the project, especially: Maxim Batalin, Josh Bers, Matt Brewer, Brendan Burns, Jason Douglas, Jakob Fredslund, Ben Grocholsky, Kim Jinsuck, Chris Jones, Boyoon Jung, Marin Kobilarov, Nathan Koenig, James McKenna, Alex Makarenko, Andy Martignoni III, Nik Melchior, Dave Naffin, Esben Østergård, Dylan Shell, Gabe Sibley, Pranav Srivastava, Kasper Støy, John Sweeney and Doug Vail.

Thanks also to SourceForge.net for project hosting.

1.8 Citations

If you find Player useful in your work, we would greatly appreciate your mentioning that fact in papers that you publish. We have presented papers on Player in peer-reviewed conferences; the following papers are the definitive references when citing Player [8, 5, 4]:

- Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 2121–2427, Las Vegas, Nevada, October 2003.
- Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proc. of the Intl. Conf. on Advanced Robotics (ICAR)*, pages 317–323, Coimbra, Portugal, July 2003.
- Brian P. Gerkey, Richard T. Vaughan, Kasper Støy, Andrew Howard, Maja J Matarić and Gaurav S Sukhatme. Most Valuable Player: A Robot Device Server for Distributed Control. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 1226–1231, Wailea, Hawaii, October 2001.

If you have space (and are feeling generous), you can also insert a footnote similar to the following:

> Player is freely available under the GNU General Public License from http://playerstage.sourceforge.net.

By including such acknowledgements, you do more than feed our egos and further our academic careers. You spread the word about Player, which will bring more users and developers, as well as please our funders, ensuring that we will continue to be allowed to hack on the software.

Chapter 2

Running Player

2.1 Building and Installing Player

System requirements, including platforms on which Player is known to build and run, are given in Section 1.3. If you've got one of those systems and the necessary tools (e.g., `g++`), then the build is pretty simple.

First, open your source tarball. Generic instructions are given in `INSTALL`, and more information is in `README`. If you don't feel like reading those files, the following should suffice:

```
$ ./configure
$ make
$ make install
```

Player will be installed in the default location: `/usr/local/`. **Note that this location differs from older versions, which were installed in** `$HOME`. The `configure` script accepts a number of options for customizing the build process, including changing the install location and adding/removing device drivers. For example, to change the install location for Player, do:

```
$ ./configure --prefix <path>
```

To see a complete list of such options (e.g., to enable drivers that aren't built by default), do:

```
$ ./configure --help
```

2.2 Command Line Arguments

Player is executed as follows:

```
$ player [-p <port>] [-s <path>] [-g <id>] [-r <path>] [-d <shlib>]
         [-k <key>] [<configfile>]
```

The command-line arguments are summarized in Table 2.1, and the configuration file syntax is described in Chapter 4. Note that you can specify at most one of `-s`, `-g`, or `-r`; if you specify none of these options, the default is to connect to physical hardware.

Argument	Meaning
-h	Print out usage statement.
-p <port>	The Player server should listen on TCP port <port>. Default is `6665`.
-s <path>	Run Player as a child process under the Stage simulator, and use memory-mapped IO through the files found in the directory <path> to communicate with the simulator. Note that `-s` should only be given manually when you are debugging Player; it is really intended for use only when Player is spawned by Stage.
-g <id>	Connect to a Gazebo simulation engine with the given id.
-r <logfile>	Read data from a stored log file instead of real sensors. For use with the `readlog` driver.
-d <shlib>	The Player server should load the shared object <shlib>. This option is generally used to load device drivers from dynamic libraries. See Section 8.2.12 for information on building such a library.
-k <key>	The Player server should enable authentication. Clients will be required to send an authentication request containing <key> before Player will service them. This option is usually only specified through Stage; Default is to disable authentication.
<configfile>	Player should load and configure device drivers according to the indicated configuration file. See Chapter 4 for details.

Table 2.1: Player command-line arguments

2.3 Visualization tools

Once you have Player running, either on a real robot or under Stage, the first thing that you might want to do is have a look at what your robot "sees." For this purpose, we provide a graphical visualization tool that we have found most useful in our work. This tool, `playerv`, can be found in the Player distribution, and is installed alongside `player` itself in the subdirectory `bin`. Enjoy.

Chapter 3

Device Overview

In Player, we use the notion of a *device* in much the same way as most UNIX systems. That is, a device is an abstract entity that provides a standard interface to some service. Devices behave similarly to files in that they must be *opened* with the correct access mode before use, and *closed* afterward. Once open, a device can be read from, written to, and configured (note the similarity to UNIX's `read()`, `write()`, and `ioctl()`), although not every device supports all three operations. A conceptual overview of these device operations is given in Section 3.1.

As in UNIX, devices in Player do **not** have a one-to-one mapping to physical hardware components, and with good reason. For example, it so happens that when retrieving odometry data from the ActivMedia Pioneer 2-DX robot, one also receives sonar range data. A client program that only wants to log the robot's current position should not also receive unwanted sonar range data; in fact, the client should be completely unaware of the coupling that exists inside the robot, because it is irrelevant. In order to present an intuitive interface to the client, Player controls one physical piece of hardware, the P2OS microcontroller, but implements two different devices: `position` and `sonar`. These two devices can be opened, closed, and controlled independently, relieving the client of the burden of remembering details about the internals of the robot. In addition to making such logical divisions, the device abstraction allows us to implement more sophisticated devices that do not simply return sensor data but rather filter or process it in some way.

Besides controlling different kinds of devices, Player can control multiple instances of a given kind of device. For example, if you have 2 SICK laser range-finders attached to your robot, then you can access both of them through the use of indices (see Chapter 4 for how to tell Player about the devices and Section 5.4 for protocol-level details of indexing). In fact, all device access is made by index; it just happens that most of the time, the index is 0 because there is only one of each device.

NOTE: Player implements *no device locking*. That is, many clients can have concurrent access to a given device, and they can concurrently command it. Player makes no attempt to arbitrate among the clients, and the command that is actually sent to the device will be determined by the rate at which the clients are sending commands, as well as some subtle timing issues. We purposefully chose not to implement any device locking, as it results in a more flexible system in which interesting ideas such as large-scale collaborative control can be explored (e.g., [3]).

3.1 Device access: data, command, configuration

Data

When a client `read`s from a device, the client receives the device's current data. For a Player device, the term *data* is used to refer to the salient state of the device. Of course, this definition is not objective, and for

some devices, there may be more than one reasonable choice at to what constitutes the *data*. However we find that for most devices we can readily define the data as the information that is of interest to clients and that changes (relatively) frequently. For example, when interacting with a `sonar` device, which controls an array of sonar tranducers, the data is simply a list of range readings. For a `laser` device, which controls a scanning laser range-finder, the data includes not only the range readings from the laser but also some current settings of the laser, such as scanning aperture and angular resolution. We originally included only the range readings, but found that the auxillary configuration data is comparatively small and is in fact required in order to correctly interpret the range data. In the same way, the contents of a device's data will in general represent a trade-off between information content and communication overhead. Keep in mind that a device's data will usually be transmitted to the client at 10Hz (or faster).

Command

When a client `write`s to a device, the client sends a new command to the device. For a Player device, the term *command* is used to refer to the salient *controllable* state of the device. That is, a device's command will generally include those parameters of the device that are most often changed in the course of using the device. For example, when interacting with a `position` device, which controls a mobile robot, the command is a set of translational and angular velocities. Although many other aspects of the robot are user-configurable, most of the time one need only change the translational and angular velocities, as they determine the physical position of the robot, which is the main point of interest. For a `speech` device, which controls the a speech synthesizer, we chose to make the command the ASCII string that is to be synthesized; we believe that this choice presents a natural interface to the underlying synthesizer. Again, command specification will be different for each device, and keep in mind that a device's command will usually be transmitted to the server at 10Hz (or faster).

Configuration

All device interaction that does not qualify as data or command must be implemented as *configuration*. Whereas data and command are asynchronous, one-way, (pseudo-)continuous streams, the Player configuration mechanism provides a synchronous, two-way, request-reply interaction between client and device. When a client sends a configuration request to the server, the request is added to an incoming queue for the appropriate device. At its leisure, the device will service the request, generate an appropriate reply and add it to the device's outgoing queue. This reply will then be transmitted by the server to the waiting client. Thus another distinction of configuration, as compared with data or command, is that configuration requests and replies are *not* overwritten and so are guaranteed to be received by the device and the client, respectively[1].

In general, any kind of device interaction can be implemented as configuration, and a device can accept more than one kind of configuration request. Usually, configuration is used to assign or query some aspect of the device's state. For example, when interacting with a `fiducial` device, which finds special beacons in the environment, there are configuration requests that can be used to set parameters used in finding and identifying the beacons. Compared to data and command frequency, configurations are made relatively rarely; if for a particular device a particular configuration change is made very often, then the variables being configured should probably be part of the device's command.

[1] This is mostly true, at least until the incoming and/or outgoing queues fill. If the incoming queue fills, then the client will be notified by the server (see Section 5.4). If the outgoing queue fills, then there is not really anthing to do; anyway this should not happen.

3.2 Interfaces vs. Drivers

In order to support different kinds of hardware, Player makes a distinction between device *interfaces* and device *drivers*. A device interface, such as `ptz`, specifies the format of the data, command, and configuration interactions that a device allows. A device driver, such as `sonyevid30`, specifies how the low-level device control will be carried out. In general, more than one driver may support a given interface[2], though not all drivers will support all configuration requests. Thus we extend in Player the analogy of UNIX devices, where, for example, a wide variety of joysticks all present the same "joystick" API to the programmer.

As an example, consider the two drivers `p2os_position` and `rwi_position`, which control Pioneer mobile robots and RWI mobile robots, respectively. They both support the `position` interface and thus they both accept commands and generate data in the same format, allowing a client program to treat them identically, ignoring the details of the underlying hardware. They also accept configuration requests in the same format, but not all configuration requests are supported by both drivers. For example, motor power can be toggled from software with Pioneer robots but not with RWI robots. Thus the `p2os_position` driver supports the configuration request to toggle motor power, while the `rwi_position` driver does not.

All client/server interaction is done by interface, with no reference to the underlying driver[3]. So, for example, if Player has been configured to control a single `position` device with index 0 and driver `p2os_position` (see Chapter 4 for how specify this information), then the client opens and controls the 0^{th} `position` device. Player could also be configured to control a second `position` device with index 1 and driver `rwi_position`; to access it, the client would open the 1^{st} `position` device.

Details on Player's device interfaces and drivers are given in Chapters 6 & 7, respectively.

3.3 Supported Hardware & Software

For a table of the currently supported hardware and software, check the FAQ:

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

[2]Conversely, a given driver may support multiple interfaces; the `amcl` and `segwayrmp` drivers demonstrate this idea.

[3]This is almost true, except that the driver name is passed back to the client when a device is opened, just in case the client wants to do something driver-specifi c.

Chapter 4

Configuration Files

Player needs to know where and how your devices are connected/configured. For example, if you are controlling a SICK LMS laser, then Player needs to know to which serial port the laser is connected. The configuration file is used to tell Player this information.
By convention, configuraton files for Player have the extension `.cfg`. Some example configuraton files are included in the distribution; they are installed in the subdirectory `config`.

4.1 Basic Syntax

Player's configuration file syntax is very similar to that of Stage (in fact, they use the same parser and much of the following text is adapted from the Stage User Manual). A simple configuration file might look like this:

```
# The file configures Player to control a Pioneer 2-DX equipped
# with a gripper and a Sony pan-tilt-zoom camera.

position:0 ( driver "p2os_position" )
sonar:0 ( driver "p2os_sonar" )
gripper:0 ( driver "p2os_gripper" )
ptz:0 ( driver "sonyevid30" )
```

This example shows the basic syntactic features of the configuration file format: comments, devices, indices, and properties.

Comments are indicated by the # symbol; they may be placed anywhere in the file and continue to the end of the line. For example:

```
# This file configures Player for a Pioneer robot
```

Devices are indicated using `interface ( ... )` entries; each such entry instantiates a device with interface `interface`. For example:

```
position:0 ( ... )
```

creates a device with a position interface (e.g., a mobile robot). The list of available interfaces is given in Chapter 6.

Player can concurrently control multiple devices with the same interface, and they are differentiated by indices. When instantiating a device, its index is indicated with a colon and number syntax. For example

```
ptz:1 ( ... )
```

creates a pan-tilt-zoom device that will be identified with index 1. If the index is not specified, then an index of 0 is assumed. Indices need not be consecutive, but for a given interface, they must be unique. If multiple devices are declared with the same interface and index, then the one that is declared last will replace the others.

Devices have properties, indicated using `name value` pairs:

```
position:0 ( driver "p2os_position" port "/dev/ttyS0" )
```

This entry creates a position device using the driver `p2os_position`; that driver will control the underlying robot hardware via the serial port `/dev/ttyS0`. Property values can be either numbers (`6665`), strings (indicated by double quotes `"robot1"`) or tuples (indicated by brackets `[1 1 0]`).

There are two special properties, which can be supplied for any device:

Name	Type	Default	Meaning
`driver`	string	varies	Selects which driver will be loaded for this interface. If no `driver` is given for a device, then the default driver for the interface will be used. Default drivers for each interface are given in Chapter 6.
`alwayson`	integer	0	Tells the server to subscribe to the device when the server starts up. You might use this to frontload startup time for drivers that take a while to start (e.g., `acts`), to start a device that will never have any direct clients (e.g., a name service), or for testing a driver without the need to connect a client.

The remaining driver-specific properties and their defaults are given in Chapter 7.

4.2 Defining new device types

The `define` statement can be used to define new types of devices. New devices are defined using the syntax:

```
define newdevice olddevice (...)
```

For example, the line:

```
define pioneer2 position (driver "p2os_position" port "/dev/ttyS0")
```

defines a new `pioneer2` device type composed of the primitive `position` device, appropriately configured for a Pioneer robot attached to the first serial port. This device may be instantiated using the standard syntax:

```
pioneer2 ()
```

4.3 Using include files

The `include` statement can be used to include device definitions (or declarations) from another file. The definitions are included with the following syntax:

```
include "filename"
```

4.4 Units

The default units for length and angles are meters and degrees respectively. Units may be changed using the following global properties:

Name	Values	Description
`unit_length`	`"m"`, `"cm"`, `"mm"`	Set the unit length to meters, centimeters or millimeters.
`unit_angle`	`"degrees"`, `"radians"`	Set the unit angle to degrees or radians.

Be warned that the length specfication applies to the include files as well, so choose a unit length early and stick to it.

Chapter 5

Client/Server Protocol

This section describes the TCP/IP socket interface to the Player server. Only device-independent information is given here. For interface-specific payload formats, see Chapter 6. For driver-specific information, see Chapter 7. For language-specific examples, consult the documentation for the appropriate client library.

5.1 A Note on Data Types

We are about to describe the protocol-level details of the socket interface to Player. As such, it is worth making clear two details regarding data types. First, the various messages that are sent between client and server are composed of fields of three different sizes, as listed in Table 5.1. They may be signed or unsigned, but they will always be the same size.

Type	size (in bytes)
byte/character	1
short	2
int	4

Table 5.1: Data types and their sizes

The second important detail is that all data on the network is in network byte-order (big-endian)[1]. So, before sending a message to the server, the client must ensure that all multibyte fields (i.e., **short**s and **int**s) are in network byte-order. Analogously, before interpreting any messages from the server, the client must ensure that all multibyte fields are in the native byte-order. Single characters require no special processing.

Most programming languages provide some method for converting from network to native byte-order and back. For example, in C you can use library functions like `ntohs()` and `htons()`. On the other hand, Java handles byteswapping on data streams automatically, and Tcl offers a choice of byte-order when using the `binary` command to marshal and demarshal binary strings.

5.2 A Note on Time

As explained below, Player messages often contain one or more time values, and it is important that the client be able to interpret them properly. Player measures time in the same way as many operating systems (`struct timeval`). Each time value is represented as two **int**s; one gives the number of seconds elapsed

[1] x86 machines are little-endian; thus clients running on them must byte-swap.

since the epoch (00:00:00 January 1, 1970) and the other gives the number of microseconds since the last second elapsed. Note that the time fields are only ever set by the server when sending a message to the client; the client should set them to zero when sending messages to the server.

5.3 Connecting to the Server

First a connection to Player needs to be established. This is done by creating a TCP socket and connecting to Player on port number[2] 6665. Immediately after connection, Player will respond with a 32-character NULL-terminated string that identifies its version; if the version string is less than 32 characters in length, NULL characters will be added to lengthen it. When Player is interfacing to real devices, the string will be something like:

```
Player v.1.2.3
```

When Player is running under Stage, the string will be something like:

```
Player v.1.2.3 (stage)
```

The client must consume these 32 bytes; whether or not they are used in any way is up to the client. After the version string, the server is waiting for direction from the client.

5.4 Message Formats

Player clients and servers communicate with a simple symmetric message protocol. Each message is composed of two parts: a header and a payload. We now describe the format of the header and of the payloads of the 4 different message types.

5.4.1 Header

Every Player message has a 32-byte header that contains information about how to interpret the payload of the message. The header format is shown in Table 5.2.

Byte 0 Byte 31

STX	type	device	index	t_sec	t_usec	ts_sec	ts_usec	reserved	size
short	**short**	**short**	**short**	**int**	**int**	**int**	**int**	**int**	**int**

Table 5.2: Message header fi elds and types

The fields in the header have the following meanings:

- `STX` - This **short** is a special symbol that signals the start of a message. It always has the same value: `0x5878`. *Remember that this number, like all other* **short***s and* **int***s, is transmitted in network byte-order.*
- `type` - This **short** designates the type of the message to follow. There are 4 message types in the Player protocol. Their type codes are given in Table 5.3 and they are described in detail in Sections 5.4.2–5.4.8.

[2]This is the default port; Player can be confi gured to listen on a different port through a command-line option at startup. See Table 2.1.

Value	Meaning
0x0001	Data message
0x0002	Command message
0x0003	Request message
0x0004	Acknowledgement response message
0x0005	Synchronization message
0x0006	Negative acknowledgment response message
0x0007	Error response message

Table 5.3: Message type codes

- device - This **short** designates the interface to which the message pertains. The currently available interface types are given in Table 5.4. Descriptions of the interfaces can found in Chapter 6.
- index - This **short** designates the particular device of type device to which the message pertains. For example, if your robot is equipped with two laser range-finders, an index of 0x0000 addresses the first and an index of 0x0001 addresses the second.
- t_sec - *(Only set by server)* This **int** is the seconds portion of the server's current time.
- t_usec - *(Only set by server)* This **int** is the microseconds portion of the server's current time.
- ts_sec - *(Only set by server on data messages)* This **int** is the seconds portion of the timestamp supplied by the device from which the data originated. It can be interpreted as the time at which the device "sensed" the phenomenom represented by the data.
- ts_usec - *(Only set by server on data messages)* This **int** is the microseconds portion of the timestamp supplied by the device from which the data originated. It can be interpreted as the time at which the device "sensed" the phenomenom represented by the data.
- reserved - This field is reserved for future use.
- size - This **int** is the size in bytes of the payload to follow (it does not include the size of the header).

5.4.2 Data Messages

When read access has been granted for a device, sensor data from that device it is sent to the client in a data message (type 0x0001). By default, the server continuously sends sensor data at 10Hz. The device and index fields designate the device from which the data comes and the ts_sec and ts_usec fields give the time at which the device generated the data. The t_sec and t_usec fields give the server's current time the payload contains the sensor data, the format of which is device-specific. Data formats are given in Chapter 6.

5.4.3 Command Messages

When write permission has been granted for a device, the client can command the device by sending a command message (type 0x0002) to the server. The device and index fields designate the device for which the command is intended. The t_sec, t_usec, ts_sec, and ts_usec fields are unused. The payload contains the actuator command, the format of which is device-specific. In the interest of simplifying the protocol, the server does NOT respond to command messages. Badly formatted commands and

commands to devices for which write permission was never established will only cause errors to be printed on the console from which Player was launched. Command formats are given in Chapter 6.

5.4.4 Request Messages

Request messages (`type 0x0003`) are sent by the client to the server to make configuration changes to devices. Although stricly speaking the device need not be open in order for the client to configure it, the client should always open it first to ensure that the configuration change is actually made (the `player` device is always "open"). The `device` and `index` fields designate the device for which the configuration is intended. The `t_sec`, `t_usec`, `ts_sec`, and `ts_usec` fields are unused. The payload contains the configuration request, the format of which is device-specific. The server will respond to each request message with an appropriate response message (`type 0x0004`, `0x0006`, or `0x0007`).

5.4.5 Acknowledgement Response Messages

Acknowledgement response messages (`type 0x0004`) are generated as a result of request messages (`type 0x0003`) that were successfully received, interpreted, and executed by a device. Acknowledgement response messages are only sent by the server to the client. The `device` and `index` fields designate the device for which the configuration is intended. The `t_sec` and `t_usec` fields give the server's current time; the `ts_sec` and `ts_usec` fields are time at which the device generated the reply. The payload contains the response, the format of which is device-specific.

5.4.6 Synchronization Messages

After each round of data, the server sends a single synchronization message (type `0x0005`), with a zero-length body. This message lets the client know that all data for this cycle has been sent. The synchronization message is sent even when no data was sent, as can occur if the client is using one of the data delivery modes that only send *new* data.

5.4.7 Negative Acknowledgement Response Messages

Negative acknowledgement response messages (`type 0x0006`) are generated as a result of request messages (`type 0x0003`) that were successfully received by a device, but which could not be properly interpreted or executed. Negative acknowledgement response messages are only sent by the server to the client. The `device` and `index` fields designate the device for which the configuration is intended. The `t_sec` and `t_usec` fields give the server's current time; the `ts_sec` and `ts_usec` fields are time at which the device generated the reply. The payload contains the response, the format of which is device-specific.

5.4.8 Error Acknowledgement Response Messages

Error acknowledgement response messages (`type 0x0007`) are generated as a result of request messages (`type 0x0003`) that could not be handed off to the target device, usually because the device's incoming configuration queue is full. Error acknowledgement response messages are only sent by the server to the client. The `device` and `index` fields designate the device for which the configuration is intended. The `t_sec` and `t_usec` fields give the server's current time; the `ts_sec` and `ts_usec` fields are unused. The payload will be empty.

Value	Interface	Description
0x0001	player	The player server itself
0x00FF	null	Null interface
0x0002	power	Power subsytem
0x0003	gripper	Simple robotic gripper
0x0004	position	Mobile robot base
0x0005	sonar	Array of fi xed acoustic range-fi nders
0x0006	laser	Single-origin scanning range-fi nder
0x0007	blobfinder	Visual color segmentation system
0x0008	ptz	Pan-tilt-zoom camera unit
0x0009	audio	Fixed-tone generation and detection
0x000A	fiducial	Fiducial (e.g., landmark) detector
0x000B	comms	General-purpose communication system
0x000C	speech	Speech synthesis/recognition system
0x000D	gps	Global positioning system
0x000E	bumper	Tactile bumper array
0x000F	truth	Ground truth (only available in Stage)
0x0010	idarturret	Collection of IDAR sensors
0x0011	idar	IDAR (Infrared Data and Ranging) sensor
0x0012	descartes	The Descartes mobile robot base
0x0014	dio	Digitial I/O
0x0015	aio	Analog I/O
0x0016	ir	Array of fi xed infrared range-fi nders
0x0017	wifi	Wireless Ethernet card
0x0018	waveform	Raw digital data (e.g., audio)
0x0019	localize	Multi-hypothesis localization system
0x001A	mcom	Inter-robot stack-based communication
0x001B	sound	Play pre-recorded sound fi les
0x001C	audiodsp	Fixed-tone generation and detection
0x001D	audiomixer	Control sound levels
0x001E	position3d	Robot base that moves in 3D
0x001F	simulation	Interface for controlling simulator
0x0020	service_adv	Service discovery
0x0021	blinkenlight	Blinking lights
0x0022	camera	Camera images

Table 5.4: Device interface type codes

Chapter 6

Device Interfaces

In this chapter, we define the various interface-specific payload formats. The Player protocol itself is described in Chapter 5. Although this section is generally up-to-date, the best place to look for the "real" message formats is in the header file `player.h`; that file defines the C `structs` that are manipulated by Player and the various device drivers.

6.1 Player

Synopsis

The `player` device represents the server itself, and is used in configuring the behavior of the server. This device is always open.

Constants

#define PLAYER_READ_MODE 'r'
#define PLAYER_WRITE_MODE 'w'
#define PLAYER_ALL_MODE 'a'
#define PLAYER_CLOSE_MODE 'c'
#define PLAYER_ERROR_MODE 'e'

The device access modes

#define PLAYER_DATAMODE_PUSH_ALL 0
#define PLAYER_DATAMODE_PULL_ALL 1
#define PLAYER_DATAMODE_PUSH_NEW 2
#define PLAYER_DATAMODE_PULL_NEW 3

The valid data delivery modes

#define PLAYER_PLAYER_DEVLIST_REQ ((uint16_t)1)
#define PLAYER_PLAYER_DRIVERINFO_REQ ((uint16_t)2)
#define PLAYER_PLAYER_DEV_REQ ((uint16_t)3)
#define PLAYER_PLAYER_DATA_REQ ((uint16_t)4)
#define PLAYER_PLAYER_DATAMODE_REQ ((uint16_t)5)
#define PLAYER_PLAYER_DATAFREQ_REQ ((uint16_t)6)
#define PLAYER_PLAYER_AUTH_REQ ((uint16_t)7)
#define PLAYER_PLAYER_NAMESERVICE_REQ ((uint16_t)8)

The request subtypes

Data

This interface accepts no commands.

Commands

This interface accepts no commands.

Configuration: Get device list

struct player_device_id : A device identifier; devices are differentiated internally in Player by these identifiers, and some messages contain them.

uint16_t code;

The interface provided by the device

uint16_t index;

The index of the device

uint16_t port;

The TCP port of the device (only useful with Stage)

struct player_device_devlist : Get the list of available devices from the server. It's useful for applications such as viewer programs and test suites that tailor behave differently depending on which devices are available. To request the list, set the subtype to PLAYER_PLAYER_DEVLIST_REQ and leave the rest of the fields blank. Player will return a packet with subtype PLAYER_PLAYER_DEVLIST_REQ with the fields filled in.

uint16_t subtype;

Subtype; must be PLAYER_PLAYER_DEVLIST_REQ.

uint16_t device_count;

The number of devices

player_device_id_t devices[PLAYER_MAX_DEVICES];

The list of available devices.

Configuration: Get driver name

struct player_device_driverinfo : Get the driver name for a particular device. To get a name, set the subtype to PLAYER_PLAYER_DRIVERINFO_REQ and set the id field. Player will return the driver info.

uint16_t subtype;

Subtype; must be PLAYER_PLAYER_DRIVERINFO_REQ.

player_device_id_t id;

The device identifier.

char driver_name[PLAYER_MAX_DEVICE_STRING_LEN];

The driver name (returned)

Configuration: Request device access

struct player_device_req : *This is the most important request!* Before interacting with a device, the client must request appropriate access. The format of this request is:

uint16_t subtype;

Subtype; must be PLAYER_PLAYER_DEV_REQ

uint16_t code;

The interface for the device

uint16_t index;

The index for the device

uint8_t access;

The requested access

struct player_device_resp : The format of the server's reply is:

uint16_t subtype;

Subtype; will be PLAYER_PLAYER_DEV_REQ

uint16_t code;

The interface for the device

uint16_t index;

The index for the device

uint8_t access;

The granted access

uint8_t driver_name[PLAYER_MAX_DEVICE_STRING_LEN];

The name of the underlying driver

The access codes, which are used in both the request and response, are given above. **Read** access means that the server will start sending data from the specified device. For instance, if read access is obtained for the sonar device Player will start sending sonar data to the client. **Write** access means that the client has permission to control the actuators of the device. There is no locking mechanism so different clients can have concurrent write access to the same actuators. **All** access is both of the above and finally **close** means that there is no longer any access to the device. Device request messages can be sent at any time, providing on the fly reconfiguration for clients that need different devices depending on the task at hand.
Of course, not all of the access codes are applicable to all devices; for instance it does not make sense to write to the sonars. However, a request for such access will not generate an error; rather, it will be granted, but any commands actually sent to that device will be ignored. In response to such a device request, the server will send a reply indicating the *actual* access that was granted for the device. The granted access may be different from the requested access; in particular, if there was some error in initializing the device the granted access will be 'e', and the client should not try to read from or write to the device.

Configuration: Request data

struct player_device_data_req : When the server is in a *pull* data delivery mode (see next request for information on data delivery modes), the client can request a single round of data by sending a zero-argument request with type code `0x0003`. The response will be a zero-length acknowledgement.

uint16_t subtype;

Subtype; must be PLAYER_PLAYER_DATA_REQ

Configuration: Change data delivery mode

struct player_device_datamode_req : The Player server supports four data modes, described above. By default, the server operates in `PLAYER_DATAMODE_PUSH_NEW` mode at a frequency of 10Hz. To switch to a different mode send a request with the format given below. The server's reply will be a zero-length acknowledgement.

uint16_t subtype;

Subtype; must be PLAYER_PLAYER_DATAMODE_REQ

uint8_t mode;

The requested mode

Configuration: Change data delivery frequency

struct player_device_datafreq_req : By default, the fixed frequency for the *push* data delivery modes is 10Hz; thus a client which makes no configuration changes will receive sensor data approximately every 100ms. The server can send data faster or slower; to change the frequency, send a request of the format:

uint16_t subtype;

Subtype; must be PLAYER_PLAYER_DATAFREQ_REQ

uint16_t frequency;

requested frequency in Hz

Configuration: Authentication

struct player_device_auth_req : If server authentication has been enabled (by providing `-key <key>` on the command-line; see Section 2.2), then each client must authenticate itself before otherwise interacting with the server. To authenticate, send a request of the format:

uint16_t subtype;

Subtype; must by PLAYER_PLAYER_AUTH_REQ

uint8_t auth_key[PLAYER_KEYLEN];

The authentication key

If the key matches the server's key then the client is authenticated, the server will reply with a zero-length acknowledgement, and the client can continue with other operations. If the key does not match, or if the client attempts any other server interactions before authenticating, then the connection will be closed immediately. It is only necessary to authenticate each client once.
Note that this support for authentication is **NOT** a security mechanism. The keys are always in plain text, both in memory and when transmitted over the network; further, since the key is given on the command-line, there is a very good chance that you can find it in plain text in the process table (in Linux try `ps -ax | grep player`). Thus you should not use an important password as your key, nor should you rely on Player authentication to prevent bad guys from driving your robots (use a firewall instead). Rather, authentication was introduced into Player to prevent accidentally connecting one's client program to someone else's robot. This kind of accident occurs primarily when Stage is running in a multi-user environment. In this case it is very likely that there is a Player server listening on port 6665, and clients will generally connect to that port by default, unless a specific option is given. Check the Stage documentation for how to specify a Player authentication key in your `.world` file.

struct player_device_nameservice_req : Documentation about nameservice goes here

uint16_t subtype;

Subtype; must by PLAYER_PLAYER_NAMESERVICE_REQ

uint8_t name[PLAYER_MAX_DEVICE_STRING_LEN];

The robot name

uint16_t port;

The corresponding port

6.2 null

Synopsis

The `null` interface produces no data, and accepts no commands or configuration requests. It is the Player analogue to `/dev/null`.

6.3 aio

Synopsis

The `aio` interface provides access to an analog I/O device.

Constants

#define PLAYER_AIO_MAX_SAMPLES 8

The maximum number of analog I/O samples

Data

struct player_aio_data : The `aio` interface returns data regarding the current state of the analog inputs; the format is:

uint8_t count;

number of valid samples

int32_t anin[PLAYER_AIO_MAX_SAMPLES];

the samples

Commands

This interface accepts no commands.

6.4 audio

Synopsis

The `audio` interface is used to control sound hardware, if equipped.

Data

struct player_audio_data : The `audio` interface reads the audio stream from `/dev/audio` (which is assumed to be associated with a sound card connected to a microphone) and performs some analysis on it. Five frequency/amplitude pairs are then returned as data; the format is:

uint16_t frequency0, amplitude0;

Hz, db ?

uint16_t frequency1, amplitude1;

Hz, db ?

uint16_t frequency2, amplitude2;

Hz, db ?

uint16_t frequency3, amplitude3;

Hz, db ?

uint16_t frequency4, amplitude4;

Hz, db ?

Command

struct player_audio_cmd : The `audio` interface accepts commands to produce fixed-frequency tones through `/dev/dsp` (which is assumed to be associated with a sound card to which a speaker is attached); the format is:

uint16_t frequency;

Frequency to play (Hz?)

uint16_t amplitude;

Amplitude to play (dB?)

uint16_t duration;

Duration to play (sec?)

6.5 audiodsp

Synopsis

The `audiodsp` interface is used to control sound hardware, if equipped.

Data

struct player_audiodsp_data : The `dsp` interface reads the audio stream from `/dev/dsp` (which is assumed to be associated with a sound card connected to a microphone) and performs some analysis on it. Five frequency/amplitude pairs are then returned as data; the format is:

uint16_t freq[5];

Hz

uint16_t amp[5];

Db ?

Command

struct player_audiodsp_cmd : The `audiodsp` interface accepts commands to produce fixed-frequency tones or binary phase shift keyed(BPSK) chirps through `/dev/dsp` (which is assumed to be associated with a sound card to which a speaker is attached); the format is:

uint8_t subtype;

The packet subtype. Set to PLAYER_AUDIODSP_PLAY_TONE to play a single frequency; bitString and bitStringLen do not need to be set. Set to PLAYER_AUDIODSP_PLAY_CHIRP to play a BPSKeyed chirp; bitString should contain the binary string to encode, and bitStringLen set to the length of the bitString. Set to PLAYER_AUDIODSP_REPLAY to replay the last sound. *

uint16_t frequency;

Frequency to play (Hz)

uint16_t amplitude;

Amplitude to play (dB?)

uint32_t duration;

Duration to play (msec)

unsigned char bitString[PLAYER_MAX_DEVICE_STRING_LEN];

BitString to encode in sine wave

uint16_t bitStringLen;

Length of the bit string

Configuration: get/set audio properties

The audiodsp configuration can be queried using the PLAYER_AUDIODSP_GET_CONFIG request and modified using the PLAYER_AUDIODSP_SET_CONFIG request.

The sample format is defined in sys/soundcard.h, and defines the byte size and endian format for each sample.

The sample rate defines the Hertz at which to sample.

Mono or stereo sampling is defined in the channels parameter where 1==mono and 2==stereo.

struct player_audiodsp_config : Request/reply packet for getting and setting the audio configuration.

uint8_t subtype;

The packet subtype. Set this to PLAYER_AUDIODSP_SET_CONFIG to set the audiodsp configuration; or set to PLAYER_AUDIODSP_GET_CONFIG to get the audiodsp configuration.

int16_t sampleFormat;

Format with which to sample

uint16_t sampleRate;

Sample rate in Hertz

uint8_t channels;

Number of channels to use. 1=mono, 2=stereo

6.6 audiomixer

Synopsis

The `audiomixer` interface is used to control sound levels.

Configuration: get levels

struct player_audiomixer_config : The `audiomixer` interface provides accepts a configuration request which returns the current state of the mixer levels.

uint8_t subtype;
uint16_t masterLeft, masterRight;
uint16_t pcmLeft, pcmRight;
uint16_t lineLeft, lineRight;
uint16_t micLeft, micRight;
uint16_t iGain, oGain;

documentation for these fields goes here

Command

struct player_audiomixer_cmd : The `audiomixer` interface accepts commands to set the left and right volume levels of various channels. The channel may be PLAYER_AUDIOMIXER_MASTER for the master volume, PLAYER_AUDIOMIXER_PCM for the PCM volume, PLAYER_AUDIOMIXER_LINE for the line in volume, PLAYER_AUDIOMIXER_MIC for the microphone volume, PLAYER_AUDIOMIXER_IGAIN for the input gain, and PLAYER_AUDIOMIXER_OGAIN for the output gain.

uint8_t subtype;
uint16_t left;
uint16_t right;

documentation for these fields goes here

6.7 blobfinder

Synopsis

The blobfinder interface provides access to devices that detect colored blobs.

#define PLAYER_BLOBFINDER_SET_COLOR_REQ ((uint8_t)1)
#define PLAYER_BLOBFINDER_SET_IMAGER_PARAMS_REQ ((uint8_t)2)

[Constants]

#define PLAYER_BLOBFINDER_MAX_CHANNELS 32

The maximum number of unique color classes.

#define PLAYER_BLOBFINDER_MAX_BLOBS_PER_CHANNEL 10

The maximum number of blobs for each color class.

#define PLAYER_BLOBFINDER_MAX_BLOBS PLAYER_BLOBFINDER_MAX_CHANNELS * PLAYER_BLOBFINDER_MAX_BLOBS_PER_CHANNEL

The maximum number of blobs in total.

Data

The format of the `blobfinder` data packet is very similar to the ACTS v1.2/2.0 format, but a bit simpler. The packet length is variable, with each packet containing both a list of blobs and a header that provides an index into that list. For each channel, the header entry tells you which blob to start with and how many blobs there are.

struct player_blobfinder_header_elt : Blob index entry.

uint16_t index;

Offset of the first blob for this channel.

uint16_t num;

Number of blobs for this channel.

struct player_blobfinder_blob_elt : Structure describing a single blob.

uint32_t color;

A descriptive color for the blob (useful for gui's). The color is stored as packed 32-bit RGB, i.e., 0x00RRGGBB.

uint32_t area;

The blob area (pixels).

uint16_t x, y;

The blob centroid (image coords).

uint16_t left, right, top, bottom;

Bounding box for the blob (image coords).

uint16_t range;

Range (mm) to the blob center

struct player_blobfinder_data : The list of detected blobs.

uint16_t width, height;

The image dimensions.

player_blobfinder_header_elt_t header[PLAYER_BLOBFINDER_MAX_CHANNELS];

An index into the list of blobs (blobs are indexed by channel).

player_blobfinder_blob_elt_t blobs[PLAYER_BLOBFINDER_MAX_BLOBS];

The list of blobs.

Configuration: Set tracking color

struct player_blobfinder_color_config : For some sensors (ie CMUcam), simple blob tracking tracks only one color. To set the tracking color, send a request with the format below, including the RGB color ranges (max and min). Values of -1 will cause the track color to be automatically set to the current window color. This is useful for setting the track color by holding the tracking object in front of the lens.

uint8_t subtype;

Must be PLAYER_BLOBFINDER_SET_COLOR_REQ.

int16_t rmin, rmax;
int16_t gmin, gmax;
int16_t bmin, bmax;

RGB minimum and max values (0-255) *

Configuration: Set imager params

struct player_blobfinder_imager_config : Imaging sensors that do blob tracking generally have some sorts of image quality parameters that you can tweak. The following ones are implemented here: brightness (0-255) contrast (0-255) auto gain (0=off, 1=on) color mode (0=RGB/AutoWhiteBalance Off, 1=RGB/AutoWhiteBalance On, 2=YCrCB/AWB Off, 3=YCrCb/AWB On) To set the params, send a request with the format below. Any values set to -1 will be left unchanged.

uint8_t subtype;

Must be PLAYER_BLOBFINDER_SET_IMAGER_PARAMS_REQ.

Command

This device accepts no commands.

6.8 bumper

Constants

#define PLAYER_BUMPER_MAX_SAMPLES 32

Maximum number of bumper samples

#define PLAYER_BUMPER_GET_GEOM_REQ ((uint8_t)1)

The request subtypes

Data

struct player_bumper_data : The `gps` interface gives current global position and heading information; the format is:

uint8_t bumper_count;

the number of valid bumper readings

uint8_t bumpers[PLAYER_BUMPER_MAX_SAMPLES];

array of bumper values

Commands

This interface accepts no commands.

Configuration: Query geometry

To query the geometry of a bumper array, give the following request, filling in only the subtype. The server will repond with the other fields filled in.

struct player_bumper_define : The geometry of a single bumper

int16_t x_offset, y_offset, th_offset;

the local pose of a single bumper in mm

uint16_t length;

length of the sensor in mm

uint16_t radius;

radius of curvature in mm - zero for straight lines

struct player_bumper_geom : The geometry for all bumpers.

uint8_t subtype;

Packet subtype. Must be PLAYER_BUMPER_GET_GEOM_REQ.

uint16_t bumper_count;

The number of valid bumper definitions.

player_bumper_define_t bumper_def[PLAYER_BUMPER_MAX_SAMPLES];

geometry of each bumper

6.9 comms

Synopsis

The comms interface allows clients to communicate with each other through the Player server.

Data

This interface returns unstructured, variable length binary data. The packet size must be less than or equal to PLAYER_MAX_MESSAGE_SIZE. Note that more than one data packet may be returned in any given cycle (i.e. in the interval between two SYNCH packets).

Command

This interface accepts unstructred, variable length binary data. The packet size must be less than or equal to PLAYER_MAX_MESSAGE_SIZE.

6.10 camera

Synopsis

EXPERIMENTAL. The `camera` interface is used to see what the camera sees. It is intended primarily for server-side (i.e., driver-to-driver) data transfers, rather than server-to-client transfers.

Data

struct player_camera_data : The `camera` interface returns the image seen by the camera; the format is:

uint16_t width, height;

Image dimensions (pixels).

uint8_t depth;

Image depth (8, 16, 24).

uint32_t image_size;

Size of image data (bytes)

uint8_t image[PLAYER_CAMERA_IMAGE_SIZE];

Image data (packed format).

6.11 dio

Synopsis

The `dio` interface provides access to a digital I/O device.

Data

struct player_dio_data : The `dio` interface returns data regarding the current state of the digital inputs; the format is:

uint8_t count;

number of samples

uint32_t digin;

bitfield of samples

Commands

This interface accepts no commands.

6.12 fiducial

Synopsis

The fiducial interface provides access to devices that detect coded fiducials (markers) placed in the environment.

Constants

#define PLAYER_FIDUCIAL_MAX_SAMPLES 32

The maximum number of fiducials that can be detected at one time.

#define PLAYER_FIDUCIAL_MAX_MSG_LEN 32

The maximum size of a data packet exchanged with a fiducial at one time.

#define PLAYER_FIDUCIAL_GET_GEOM 0x01
#define PLAYER_FIDUCIAL_GET_FOV 0x02
#define PLAYER_FIDUCIAL_SET_FOV 0x03
#define PLAYER_FIDUCIAL_SEND_MSG 0x04
#define PLAYER_FIDUCIAL_RECV_MSG 0x05
#define PLAYER_FIDUCIAL_EXCHANGE_MSG 0x06

Request packet subtypes

Data

The fiducial data packet contains a list of the detected fiducials. Each fiducial is described by the player_fiducial_item structure listed below.

struct player_fiducial_item : The fiducial data packet (one fiducial).

int16_t id;

The fiducial id. Fiducials that cannot be identified get id -1.

int16_t pose[3];

Fiducial pose relative to the detector (range, bearing, orient) in units (mm, degrees, degrees).

int16_t upose[3];

Uncertainty in the measured pose (range, bearing, orient) in units of (mm, degrees, degrees).

struct player_fiducial_data : The fiducial data packet (all fiducials).

uint16_t count;

The number of detected fiducials

player_fiducial_item_t fiducials[PLAYER_FIDUCIAL_MAX_SAMPLES];

List of detected fiducials

Command

This device accepts no commands.

Configuration: get geometry

The geometry (pose and size) of the fiducial device can be queried using the PLAYER_FIDUCIAL_GET_GEOM request. The request and reply packets have the same format.

struct player_fiducial_geom : Fiducial geometry packet.

uint8_t subtype;

Packet subtype. Must be PLAYER_FIDUCIAL_GET_GEOM.

uint16_t pose[3];

Pose of the detector in the robot cs (x, y, orient) in units if (mm, mm, degrees).

uint16_t size[2];

Size of the detector in units of (mm, mm)

uint16_t fiducial_size[2];

Dimensions of the fiducials in units of (mm, mm).

Configuration: sensor field of view

The field of view of the fiducial device can be set using the PLAYER_FIDUCIAL_SET_FOV request, and queried using the PLAYER_FIDUCIAL_GET_FOV request. The device replies to a SET request with the actual FOV achieved. In both cases the request and reply packets have the same format.

struct player_fiducial_fov : Fiducial geometry packet.

uint8_t subtype;

Packet subtype. PLAYER_FIDUCIAL_GET_FOV or PLAYER_FIDUCIAL_SET_FOV.

uint16_t min_range;

The minimum range of the sensor in mm

uint16_t max_range;

The maximum range of the sensor in mm

uint16_t view_angle;

The receptive angle of the sensor in degrees.

Configuration: fiducial messaging.

NOTE: These configs are currently supported only by the Stage fiducial driver (stg_fidicial), but are intended to be a general interface for addressable, peer-to-peer messaging.

The fiducial sensor can attempt to send a message to a target using the PLAYER_FIDUCIAL_SEND_MSG request. If target_id is -1, the message is broadcast to all targets. The device replies with an ACK if the message was sent OK, but receipt by the target is not guaranteed. The intensity field sets a transmit power in device-dependent units. If the consume flag is set, the message is transmitted just once. If it is unset, the message may transmitted repeatedly (at device-dependent intervals, if at all).

Send a PLAYER_FIDUCIAL_RECV_MSG request to obtain the last message recieved from the indicated target. If the consume flag is set, the message is deleted from the device's buffer, if unset, the same message can be retreived multiple times until a new message arrives. The power field indicates the intensity of the recieved messag, again in device-dependent units.

Similarly, the PLAYER_FIDUCIAL_EXCHANGE_MSG request sends a message, then returns the most recently received message. Depending on the device and the situation, this could be a reflection of the sent message, a reply from the target of the sent message, or a message received from an unrelated sender.

Fiducial exchange mesaage request. The device sends the message, then replies with the last message received, which may be (but is not guaranteed to be) be a reply to the sent message. NOTE: this is not yet supported by Stage-1.4.

6.13 gps

Synopsis

The `gps` interface provides access to an absolute position system, such as GPS.

Data

struct player_gps_data : The `gps` interface gives current global position and heading information; the format is:

uint32_t time_sec;
uint32_t time_usec;

GPS (UTC) time, in seconds and microseconds since the epoch.

int32_t latitude;

Latitude, in 1/60 of an arc-second (i.e., 1/216000 of a degree). Positive is north of equator, negative is south of equator.

int32_t longitude;

Longitude, in 1/60 of an arc-second (i.e., 1/216000 of a degree). Positive is east of prime meridian, negative is west of prime meridian.

int32_t altitude;

Altitude, in millimeters. Positive is above reference (e.g., sea-level), and negative is below.

int32_t utm_e, utm_n;

UTM WGS84 coordinates, easting and northing (cm).

uint8_t quality;

Quality of fix 0 = invalid, 1 = GPS fix, 2 = DGPS fix

uint8_t num_sats;

Number of satellites in view.

uint16_t hdop;

Horizontal dilution of position (HDOP), times 10

uint32_t err_horz;

Horizonal error (mm)

uint32_t err_vert;

Vertical error (mm)

Commands

This interface accepts no commands.

6.14 gripper

Synopsis

The `gripper` interface provides access to a robotic gripper.

Data

struct player_gripper_data : The `gripper` interface returns 2 bytes that represent the current state of the gripper; the format is given below. Note that the exact interpretation of this data may vary depending on the details of your gripper and how it is connected to your robot (e.g., General I/O vs. User I/O for the Pioneer gripper).

uint8_t state, beams;

The current gripper lift and breakbeam state

The following table defines how the data can be interpreted for some Pioneer robots and Stage:

Field	Type	Meaning
`state`	**unsigned byte**	bit 0: Paddles open
		bit 1: Paddles closed
		bit 2: Paddles moving
		bit 3: Paddles error
		bit 4: Lift is up
		bit 5: Lift is down
		bit 6: Lift is moving
		bit 7: Lift error
`beams`	**unsigned byte**	bit 0: Gripper limit reached
		bit 1: Lift limit reached
		bit 2: Outer beam obstructed
		bit 3: Inner beam obstructed
		bit 4: Left paddle open
		bit 5: Right paddle open

Commands

struct player_gripper_cmd : The `gripper` interface accepts 2-byte commands, the format of which is given below. These two bytes are sent directly to the gripper; refer to Table 3-3 page 10 in the Pioneer 2 Gripper Manual[1] for a list of commands. The first byte is the command. The second is the argument for the LIFTcarry and GRIPpress commands, but for all others it is ignored.

uint8_t cmd, arg;

the command and optional argument

6.15 ir

Synopsis

The `ir` interface provides access to an array of infrared (IR) range sensors.

Constants

#define PLAYER_IR_MAX_SAMPLES 32

Maximum number of samples

#define PLAYER_IR_POSE_REQ ((uint8_t)1)
#define PLAYER_IR_POWER_REQ ((uint8_t)2)

config requests

Data

struct player_ir_data : The `ir` interface returns range readings from the IR array; the format is:

uint16_t range_count;

number of samples

uint16_t voltages[PLAYER_IR_MAX_SAMPLES];

voltages (units?)

uint16_t ranges[PLAYER_IR_MAX_SAMPLES];

ranges (mm)

Commands

This interface accepts no commands.

Configuration: Query pose

To query the pose of the IRs, use the following request, filling in only the subtype. The server will respond with the other fields filled in.

struct player_ir_pose : gets the pose of the IR sensors on a robot

uint16_t pose_count;

the number of ir samples returned by this robot

int16_t poses[PLAYER_IR_MAX_SAMPLES][3];

the pose of each IR detector on this robot (mm, mm, degrees)

struct player_ir_pose_req : ioctl struct for getting IR pose of a robot

uint8_t subtype;

subtype; must be PLAYER_IR_POSE_REQ

player_ir_pose_t poses;

the poses

Configuration: IR power

struct player_ir_power_req : To turn IR power on and off, use this request. The server will reply with a zero-length acknowledgement

uint8_t subtype;

must be PLAYER_IR_POWER_REQ

uint8_t state;

0 for power off, 1 for power on

6.16 laser

Synopsis

The laser interface provides access to a single-origin scanning range sensor, such as a SICK laser range-finder.

Constants

#define PLAYER_LASER_MAX_SAMPLES 401

The maximum number of laser range values

#define PLAYER_LASER_GET_GEOM 0x01
#define PLAYER_LASER_SET_CONFIG 0x02
#define PLAYER_LASER_GET_CONFIG 0x03
#define PLAYER_LASER_POWER_CONFIG 0x04

Laser request subtypes.

Data

Devices supporting the `laser` interface can be configured to scan at different angles and resolutions. As such, the data returned by the `laser` interface can take different forms. To make interpretation of the data simple, the `laser` data packet contains some extra fields before the actual range data. These fields tell the client the starting and ending angles of the scan, the angular resolution of the scan, and the number of range readings included. Scans proceed counterclockwise about the laser, and 0° is forward. The laser can return a maximum of 401 readings; this limits the valid combinations of scan width and angular resolution.

struct player_laser_data : The laser data packet.

int16_t min_angle, max_angle;

Start and end angles for the laser scan (in units of 0.01 degrees).

uint16_t resolution;

Angular resolution (in units of 0.01 degrees).

uint16_t range_res;

range resolution. ranges should be multipled by this.

uint16_t range_count;

Number of range/intensity readings.

uint16_t ranges[PLAYER_LASER_MAX_SAMPLES];

Range readings (mm).

uint8_t intensity[PLAYER_LASER_MAX_SAMPLES];

Intensity readings.

Command

This device accepts no commands.

Configuration: get geometry

The laser geometry (position and size) can be queried using the PLAYER_LASER_GET_GEOM request. The request and reply packets have the same format.

struct player_laser_geom : Request/reply packet for getting laser geometry.

uint8_t subtype;

The packet subtype. Must be PLAYER_LASER_GET_GEOM.

int16_t pose[3];

Laser pose, in robot cs (mm, mm, degrees).

int16_t size[2];

Laser dimensions (mm, mm).

Configuration: get/set scan properties

The scan configuration can be queried using the PLAYER_LASER_GET_CONFIG request and modified using the PLAYER_LASER_SET_CONFIG request.

The sicklms200 driver, for example, is usually configured to scan a swath of 180° with a resolution of 0.5°, to generate a total of 361 readings. At this aperture, the laser generates a new scan every 200ms or so, for a data rate of 5Hz. This rate can be raised by reducing the aperture to encompass less than the full 180°, or by lowering the resolution to 1°.

Read the documentation for your driver to determine what configuration values are permissible.

struct player_laser_config : Request/reply packet for getting and setting the laser configuration.

uint8_t subtype;

The packet subtype. Set this to PLAYER_LASER_SET_CONFIG to set the laser configuration; or set to PLAYER_LASER_GET_CONFIG to get the laser configuration.

int16_t min_angle, max_angle;

Start and end angles for the laser scan (in units of 0.01 degrees). Valid range is -9000 to +9000.

uint16_t resolution;

Scan resolution (in units of 0.01 degrees). Valid resolutions are 25, 50, 100.

uint16_t range_res;

Range Resolution. Valid: 1, 10, 100 (For mm, cm, dm).

uint8_t intensity;

Enable reflection intensity data.

struct player_laser_power_config : Turn the laser power on or off.

uint8_t subtype;

Must be PLAYER_LASER_POWER_CONFIG.

uint8_t value;

0 to turn laser off, 1 to turn laser on

6.17 localize

Synopsis

The `localize` interface provides pose information for the robot. Generally speaking, localization drivers will estimate the pose of the robot by comparing observed sensor readings against a pre-defined map of the environment. See, for the example, the `regular_mcl` and `adaptive_mcl` drivers, which implement probabilistic Monte-Carlo localization algorithms.

Constants

#define PLAYER_LOCALIZE_MAX_HYPOTHS 10

The maximum number of pose hypotheses.

#define PLAYER_LOCALIZE_SET_POSE_REQ ((uint8_t)1)
#define PLAYER_LOCALIZE_GET_CONFIG_REQ ((uint8_t)2)
#define PLAYER_LOCALIZE_SET_CONFIG_REQ ((uint8_t)3)
#define PLAYER_LOCALIZE_GET_MAP_INFO_REQ ((uint8_t)4)
#define PLAYER_LOCALIZE_GET_MAP_DATA_REQ ((uint8_t)5)

Request/reply packet subtypes

Data

struct player_localize_hypoth : Since the robot pose may be ambiguous (i.e., the robot may at any of a number of widely spaced locations), the `localize` interface is capable of returning more that one hypothesis. The format for each such hypothesis is as follows:

int32_t mean[3];

The mean value of the pose estimate (mm, mm, arc-seconds).

int64_t cov[3][3];

The covariance matrix pose estimate (mm^2, $arc\text{-}seconds^2$).

uint32_t alpha;

The weight coefficient for linear combination (alpha * 1e6).

struct player_localize_data : The `localize` interface returns a data packet containing an an array of hypotheses, defined as follows:

uint16_t pending_count;

The number of pending (unprocessed observations)

uint32_t pending_time_sec, pending_time_usec;

The time stamp of the last observation processed.

uint32_t hypoth_count;

The number of pose hypotheses.

player_localize_hypoth_t hypoths[PLAYER_LOCALIZE_MAX_HYPOTHS];

The array of the hypotheses.

Commands

This interface accepts no commands.

Configuration: Set the robot pose estimate

struct player_localize_set_pose : Set the current robot pose hypothesis. The server will reply with a zero length response packet.

uint8_t subtype;

Request subtype; must be PLAYER_LOCALIZE_SET_POSE_REQ.

int32_t mean[3];

The mean value of the pose estimate (mm, mm, arc-seconds).

int64_t cov[3][3];

The covariance matrix pose estimate (mm^2, arc-$seconds^2$).

Configuration: Get/Set configuration

struct player_localize_config : To retrieve the configuration, set the subtype to PLAYER_LOCALIZE_GET_CONFIG_REQ and leave the other fields empty. The server will reply with the following configuaration fields filled in. To change the current configuration, set the subtype to PLAYER_LOCALIZE_SET_CONFIG_REQ and fill the configuration fields.

uint8_t subtype;

Request subtype; must be either PLAYER_LOCALIZE_GET_CONFIG_REQ or PLAYER_LOCALIZE_SET_CO

uint32_t num_particles;

Maximum number of particles (for drivers using particle * filters).

Configuration: Get map information

struct player_localize_map_info : Retrieve the size and scale information of a current map. This request is used to get the size information before you request the actual map data. Set the subtype to PLAYER_LOCALIZE_GET_MAP_INFO_REQ; the server will reply with the size information filled in.

uint8_t subtype;

Request subtype; must be PLAYER_LOCALIZE_GET_MAP_INFO_REQ

uint32_t scale;

The scale of the map (pixels per kilometer).

uint32_t width, height;

The size of the map (pixels).

Configuration: Get map data

struct player_localize_map_data : Retrieve the map data. Beacause of the limited size of a request-replay messages, the map data is tranfered in tiles. In the request packet, set the column and row index of a specific tile; the server will reply with the requested map data filled in.

uint8_t subtype;

Request subtype; must be PLAYER_LOCALIZE_MAP_DATA_REQ.

uint32_t col, row;

The tile origin (pixels).

uint32_t width, height;

The size of the tile (pixels).

int8_t data[PLAYER_MAX_REQREP_SIZE - 17];

Cell occupancy value (empty = -1, unknown = 0, occupied = +1).

6.18 mcom

Synopsis

The mcom interface is designed for exchanging information between clients. A client sends a message of a given ”type” and ”channel”. This device stores adds the message to that channel’s stack. A second client can then request data of a given ”type” and ”channel”. Push, Pop, Read, and Clear operations are defined, but their semantics can vary, based on the stack discipline of the underlying driver. For example, the lifo_mcom driver enforces a last-in-first-out stack.

Constants

#define MCOM_DATA_LEN 128
#define MCOM_COMMAND_BUFFER_SIZE (sizeof(player_mcom_config_t))
#define MCOM_DATA_BUFFER_SIZE 0

size of the data field in messages

#define MCOM_N_BUFS 10

number of buffers to keep per channel

#define MCOM_CHANNEL_LEN 8

size of channel name

#define MCOM_EMPTY_STRING ”(EMPTY)”

returns this if empty

#define PLAYER_MCOM_PUSH_REQ 0
#define PLAYER_MCOM_POP_REQ 1
#define PLAYER_MCOM_READ_REQ 2
#define PLAYER_MCOM_CLEAR_REQ 3
#define PLAYER_MCOM_SET_CAPACITY_REQ 4

request ids

Data

The mcom interface returns no data.

Command

The mcom interface accepts no commands.

Configuration

struct player_mcom_data : A piece of data.

char full;

a flag

char data[MCOM_DATA_LEN];

the data

struct player_mcom_config : Config requests sent to server.

uint8_t command;

Which request. Should be one of the defined request ids.

uint16_t type;

The ”type” of the data.

char channel[MCOM_CHANNEL_LEN];

The name of the channel.

player_mcom_data_t data;

The data.

struct player_mcom_return : Config replies from server.

uint16_t type;

The ”type” of the data

char channel[MCOM_CHANNEL_LEN];

The name of the channel.

player_mcom_data_t data;

The data.

6.19 position

Synopsis

The `position` interface is used to control a planar mobile robot base.

Constants

#define PLAYER_POSITION_GET_GEOM_REQ ((uint8_t)1)
#define PLAYER_POSITION_MOTOR_POWER_REQ ((uint8_t)2)
#define PLAYER_POSITION_VELOCITY_MODE_REQ ((uint8_t)3)
#define PLAYER_POSITION_RESET_ODOM_REQ ((uint8_t)4)
#define PLAYER_POSITION_POSITION_MODE_REQ ((uint8_t)5)
#define PLAYER_POSITION_SPEED_PID_REQ ((uint8_t)6)
#define PLAYER_POSITION_POSITION_PID_REQ ((uint8_t)7)
#define PLAYER_POSITION_SPEED_PROF_REQ ((uint8_t)8)
#define PLAYER_POSITION_SET_ODOM_REQ ((uint8_t)9)

The various configuration request types.

#define PLAYER_POSITION_RMP_VELOCITY_SCALE ((uint8_t)51)
#define PLAYER_POSITION_RMP_ACCEL_SCALE ((uint8_t)52)
#define PLAYER_POSITION_RMP_TURN_SCALE ((uint8_t)53)
#define PLAYER_POSITION_RMP_GAIN_SCHEDULE ((uint8_t)54)
#define PLAYER_POSITION_RMP_CURRENT_LIMIT ((uint8_t)55)
#define PLAYER_POSITION_RMP_RST_INTEGRATORS ((uint8_t)56)
#define PLAYER_POSITION_RMP_SHUTDOWN ((uint8_t)57)

These are possible Segway RMP config commands; see the status command in the RMP manual

#define PLAYER_POSITION_RMP_RST_INT_RIGHT 0x01
#define PLAYER_POSITION_RMP_RST_INT_LEFT 0x02
#define PLAYER_POSITION_RMP_RST_INT_YAW 0x04
#define PLAYER_POSITION_RMP_RST_INT_FOREAFT 0x08

These are used for resetting the Segway RMP's integrators.

Data

struct player_position_data : The `position` interface returns data regarding the odometric pose and velocity of the robot, as well as motor stall information; the format is:

int32_t xpos, ypos;

X and Y position, in mm

int32_t yaw;

Yaw, in degrees

int32_t xspeed, yspeed;

X and Y translational velocities, in mm/sec

int32_t yawspeed;

Angular velocity, in degrees/sec

uint8_t stall;

Are the motors stalled?

Commands

struct player_position_cmd : The `position` interface accepts new positions and/or velocities for the robot's motors (drivers may support position control, speed control, or both); the format is

int32_t xpos, ypos;

X and Y position, in mm

int32_t yaw;

Yaw, in degrees

int32_t xspeed, yspeed;

X and Y translational velocities, in mm/sec

int32_t yawspeed;

Angular velocity, in degrees/sec

uint8_t state;

Motor state (zero is either off or locked, depending on the driver).

uint8_t type;

Command type; 0 = velocity, 1 = position.

Configuration: Query geometry

struct player_position_geom : To request robot geometry, set the subtype to PLAYER_POSITION_GET_GEOM_REQ and leave the other fields empty. The server will reply with the pose and size fields filled in.

uint8_t subtype;

Packet subtype. Must be PLAYER_POSITION_GET_GEOM_REQ.

uint16_t pose[3];

Pose of the robot base, in the robot cs (mm, mm, degrees).

uint16_t size[2];

Dimensions of the base (mm, mm).

Configuration: Motor power

struct player_position_power_config : On some robots, the motor power can be turned on and off from software. To do so, send a request with the format given below, and with the appropriate `state` (zero for motors off and non-zero for motors on). The server will reply with a zero-length acknowledgement.

Be VERY careful with this command! You are very likely to start the robot running across the room at high speed with the battery charger still attached.

uint8_t request;

subtype; must be PLAYER_POSITION_MOTOR_POWER_REQ

uint8_t value;

0 for off, 1 for on

Configuration: Change velocity control

struct player_position_velocitymode_config : Some robots offer different velocity control modes. It can be changed by sending a request with the format given below, including the appropriate mode. No matter which mode is used, the external client interface to the `position` device remains the same. The server will reply with a zero-length acknowledgement

uint8_t request;

subtype; must be PLAYER_POSITION_VELOCITY_MODE_REQ

uint8_t value;

driver-specific

The `p2os_position` driver offers two modes of velocity control: separate translational and rotational control and direct wheel control. When in the separate mode, the robot's microcontroller internally computes left and right wheel velocities based on the currently commanded translational and rotational velocities and then attenuates these values to match a nice predefined acceleration profile. When in the direct mode, the microcontroller simply passes on the current left and right wheel velocities. Essentially, the separate mode offers smoother but slower (lower acceleration) control, and the direct mode offers faster but jerkier (higher acceleration) control. Player's default is to use the direct mode. Set `mode` to zero for direct control and non-zero for separate control.

For the `reb_position` driver, 0 is direct velocity control, 1 is for velocity-based heading PD controller.

Configuration: Reset odometry

struct player_position_resetodom_config : To reset the robot's odometry to $(x, y, \theta) = (0, 0, 0)$, use the following request. The server will reply with a zero-length acknowledgement.

uint8_t request;

subtype; must be PLAYER_POSITION_RESET_ODOM_REQ

Configuration: Change position control

struct player_position_position_mode_req :

uint8_t subtype;

subtype; must be PLAYER_POSITION_POSITION_MODE_REQ

uint8_t state;

0 for velocity mode, 1 for position mode

Configuration: Set odometry

struct player_position_set_odom_req : To set the robot's odometry to a particular state, use this request:

uint8_t subtype;

subtype; must be PLAYER_POSITION_SET_ODOM_REQ

int32_t x, y;

X and Y (in mm?)

int32_t theta;

Heading (in degrees)

Configuration: Set velocity PID parameters

struct player_position_speed_pid_req :

uint8_t subtype;

subtype; must be PLAYER_POSITION_SPEED_PID_REQ

int32_t kp, ki, kd;

PID parameters

Configuration: Set position PID parameters

struct player_position_position_pid_req :

uint8_t subtype;

subtype; must be PLAYER_POSITION_POSITION_PID_REQ

int32_t kp, ki, kd;

PID parameters

Configuration: Set speed profile parameters

struct player_position_speed_prof_req :

uint8_t subtype;

subtype; must be PLAYER_POSITION_SPEED_PROF_REQ

int16_t speed;

max speed

int16_t acc;

max acceleration

Configuration: Segway RMP-specific configuration

struct player_rmp_config :

uint8_t subtype;

subtype: must be of PLAYER_RMP_*

uint16_t value;

holds various values depending on the type of config. See the ”Status” command in the Segway manual.

6.20 position3d

Synopsis

The `position3d` interface is used to control a 3-D mobile robot base.

Constants

Supported config requests

Data

struct player_position3d_data : The `position3d` interface returns data regarding the odometric pose and velocity of the robot, as well as motor stall information; the format is:

int32_t xpos, ypos, zpos;

X, Y, and Z position, in mm

int32_t roll, pitch, yaw;

Roll, pitch, and yaw, in arc-seconds (1/3600 of a degree)

int32_t xspeed, yspeed, zspeed;

X, Y, and Z translational velocities, in mm/sec

int32_t rollspeed, pitchspeed, yawspeed;

Angular velocities, in arc-seconds / second

uint8_t stall;

Are the motors stalled?

Commands

struct player_position3d_cmd : The `position3d` interface accepts new positions and/or velocities for the robot's motors (drivers may support position control, speed control, or both); the format is

int32_t xpos, ypos, zpos;

X, Y, and Z position, in mm

uint32_t roll, pitch, yaw;

Roll, pitch, and yaw, in arc-seconds (1/3600 of a degree)

int32_t xspeed, yspeed, zspeed;

X, Y, and Z translational velocities, in mm/sec

int32_t rollspeed, pitchspeed, yawspeed;

Angular velocities, in arc-seconds / second

Configuration: Motor power

struct player_position3d_power_config : On some robots, the motor power can be turned on and off from software. To do so, send a request with the format given below, and with the appropriate `state` (zero for motors off and non-zero for motors on). The server will reply with a zero-length acknowledgement.

Be VERY careful with this command! You are very likely to start the robot running across the room at high speed with the battery charger still attached.

uint8_t request;

subtype; must be PLAYER_POSITION_MOTOR_POWER_REQ

uint8_t value;

0 for off, 1 for on

6.21 power

Synopsis

The `power` interface provides access to a robot's power subsystem.

Constants

#define PLAYER_MAIN_POWER_REQ ((uint8_t)14)

what does this do?

Data

struct player_power_data : The `power` device returns data in the format:

uint16_t charge;

Battery voltage, in decivolts

Commands

This interface accepts no commands

Configuration: Request power

struct player_power_config : Packet for requesting power config - replies with a player_power_data_t

uint8_t subtype;

Packet subtype. Must be PLAYER_MAIN_POWER_REQ.

6.22 ptz

Synopsis

The `ptz` interface is used to control a pan-tilt-zoom unit.

Constants

#define PLAYER_PTZ_GENERIC_CONFIG_REQ ((uint8_t)1)
#define PLAYER_PTZ_CONTROL_MODE_REQ ((uint8_t)2)

Configuration request codes

#define PLAYER_PTZ_MAX_CONFIG_LEN 32

Maximum command length for use with PLAYER_PTZ_GENERIC_CONFIG_REQ, based on the Sony EVID30 camera right now.

#define PLAYER_PTZ_VELOCITY_CONTROL 0
#define PLAYER_PTZ_POSITION_CONTROL 1

Control modes, for use with PLAYER_PTZ_CONTROL_MODE_REQ

Data

struct player_ptz_data : The `ptz` interface returns data reflecting the current state of the Pan-Tilt-Zoom unit; the format is:

int16_t pan;

Pan (degrees)

int16_t tilt;

Tilt (degrees)

int16_t zoom;

Field of view (degrees).

int16_t panspeed, tiltspeed;

Current pan/tilt velocities (deg/sec)

Command

struct player_ptz_cmd : The `ptz` interface accepts commands that set change the state of the unit; the format is given below. Note that the commands are *absolute*, not relative.

int16_t pan;

Desired pan angle (degrees)

int16_t tilt;

Desired tilt angle (degrees)

int16_t zoom;

Desired field of view (degrees).

int16_t panspeed, tiltspeed;

Desired pan/tilt velocities (deg/sec)

Configuration: Set/Get unit-specific config

struct player_ptz_generic_config : This ioctl allows the client to send a unit-specific command to the unit. Whether data is returned depends on the command that was sent. The server may fill in "config" with a reply if applicable.

uint8_t subtype;

Must be set to PLAYER_PTZ_GENERIC_CONFIG_REQ

uint16_t length;

Length of data in config buffer

uint8_t config[PLAYER_PTZ_MAX_CONFIG_LEN];

Buffer for command/reply

Configuration: Change control mode

struct player_ptz_controlmode_config : This ioctl allows the client to switch between position and velocity control, for those drivers that support it. Note that this request changes how the driver interprets forthcoming commands from *all* clients.

uint8_t subtype;

Must be set to PLAYER_PTZ_CONTROL_MODE_REQ

uint8_t mode;

Mode to use: must be either PLAYER_PTZ_VELOCITY_CONTROL or PLAYER_PTZ_POSITION_CONTROL.

6.23 sonar

Synopsis

The `sonar` interface provides access to a collection of fixed range sensors, such as a sonar array.

Constants

#define PLAYER_SONAR_MAX_SAMPLES 64

maximum number of sonar samples in a data packet

#define PLAYER_SONAR_GET_GEOM_REQ ((uint8_t)1)
#define PLAYER_SONAR_POWER_REQ ((uint8_t)2)

request types

Data

struct player_sonar_data : The `sonar` interface returns up to 32 range readings from a robot's sonars. The format is:

uint16_t range_count;

The number of valid range readings.

uint16_t ranges[PLAYER_SONAR_MAX_SAMPLES];

The range readings

Commands

This interface accepts no commands.

Configuration: Query geometry

struct player_sonar_geom : To query the geometry of the sonar transducers, use the request given below, but only fill in the subtype. The server will reply with the other fields filled in.

uint8_t subtype;

Subtype. Must be PLAYER_SONAR_GET_GEOM_REQ.

uint16_t pose_count;

The number of valid poses.

int16_t poses[PLAYER_SONAR_MAX_SAMPLES][3];

Pose of each sonar, in robot cs (mm, mm, degrees).

Configuration: Sonar power

struct player_sonar_power_config : On some robots, the sonars can be turned on and off from software. To do so, issue a request of the form given below. The server will reply with a zero-length acknowledgement.

uint8_t subtype;

Packet subtype. Must be PLAYER_P2OS_SONAR_POWER_REQ.

uint8_t value;

Turn power off (0) or on (¿0)

6.24 sound

Synopsis

The sound interface allows playback of a pre-recorded sound (e.g., on an Amigobot).

Data

This interface provides no data.

Commands

struct player_sound_cmd : The sound interface accepts an index of a pre-recorded sound file to play.

uint16_t index;

Index of sound to be played.

6.25 speech

Synopsis

The speech interface provides access to a speech synthesis system.

Constants

#define PLAYER_SPEECH_MAX_STRING_LEN 256
#define PLAYER_SPEECH_MAX_QUEUE_LEN 4

incoming command queue parameters

Data

The speech interface returns no data.

Command

struct player_speech_cmd : The speech interface accepts a command that is a string to be given to the speech synthesizer. The command packet is simply 256 bytes that are interpreted as ASCII, and so the maximum length of each string is 256 characters.

uint8_t string[PLAYER_SPEECH_MAX_STRING_LEN];

The string to say

6.26 truth

Synopsis

The `truth` interface provides access to the absolute state of entities. Note that, unless your robot has superpowers, `truth` devices are only avilable in Stage.

Constants

#define PLAYER_TRUTH_GET_POSE 0x00
#define PLAYER_TRUTH_SET_POSE 0x01
#define PLAYER_TRUTH_SET_POSE_ON_ROOT 0x02
#define PLAYER_TRUTH_GET_FIDUCIAL_ID 0x03
#define PLAYER_TRUTH_SET_FIDUCIAL_ID 0x04

Request packet subtypes.

Data

struct player_truth_data : The `truth` interface returns data concerning the current state of an entity; the format is:

int32_t px, py, pa;

Object pose in world cs (mm, mm, degrees).

Commands

This interface accepts no commands.

Configuration: Get/set pose

struct player_truth_pose : To get the pose of an object, use the following request, filling in only the subtype with PLAYER_TRUTH_GET_POSE. The server will respond with the other fields filled in. To set the pose, set the subtype to PLAYER_TRUTH_SET_POS and fill in the rest of the fields with the new pose.

uint8_t subtype;

Packet subtype. Must be either PLAYER_TRUTH_GET_POSE or PLAYER_TRUTH_SET_POSE or PLAYER_TRUTH_SET_POSE_ON_ROOT. The last option places the object on the background and sets its pose. Great for repositioning pucks that have been picked up.

int32_t px, py, pa;

Object pose in world cs (mm, mm, degrees).

Configuration: Get/set fiducial ID number

struct player_truth_fiducial_id : To get the fiducial ID of an object, use the following request, filling in only the subtype with PLAYER_TRUTH_GET_FIDUCIAL_ID. The server will respond with the ID field filled in. To set the fiducial ID, set the subtype to PLAYER_TRUTH_SET_FIDUCIAL_ID and fill in the ID field with the desired value

uint8_t subtype;

Packet subtype. Must be either PLAYER_TRUTH_GET_FIDUCIAL_ID or PLAYER_TRUTH_SET_FIDUCIAL_I

int16_t id;

the fiducial ID

6.27 waveform

Synopsis

The `waveform` interface is used to receive arbitrary digital samples, say from a digital audio device.

Data

struct player_waveform_data : The `waveform` interface reads a digitized waveform from the target device.

uint32_t rate;

Bit rate - bits per second

uint16_t depth;

Depth - bits per sample

uint32_t samples;

Samples - the number of bytes of raw data

uint8_t data[PLAYER_WAVEFORM_DATA_MAX];

data - an array of raw data

6.28 wifi

Synopsis

The `wifi` interface provides access to the state of a wireless network interface.

Constants

#define PLAYER_WIFI_MAX_LINKS 16

The maximum number of remote hosts to report on

#define PLAYER_WIFI_QUAL_DBM 1

link quality is in dBm

#define PLAYER_WIFI_QUAL_REL 2

link quality is relative

#define PLAYER_WIFI_QUAL_UNKNOWN 3

link quality is unknown

#define PLAYER_WIFI_MODE_UNKNOWN 0

unknown operating mode

#define PLAYER_WIFI_MODE_AUTO 1

driver decides the mode

#define PLAYER_WIFI_MODE_ADHOC 2

ad hoc mode

#define PLAYER_WIFI_MODE_INFRA 3

infrastructure mode (multi cell network, roaming)

#define PLAYER_WIFI_MODE_MASTER 4

access point, master mode

#define PLAYER_WIFI_MODE_REPEAT 5

repeater mode

#define PLAYER_WIFI_MODE_SECOND 6

secondary/backup repeater

#define PLAYER_WIFI_MAC_REQ ((uint8_t)1)

config requests

Data

struct player_wifi_link : The `wifi` interface returns data regarding the signal characteristics of remote hosts as perceived through a wireless network interface; the format of the data for each host is:

char ip[32];

IP address of destination.

// these could be uint8_t instead, ¡linux/wireless.h¿ will only
// return that much. maybe some other architecture needs larger??
uint16_t qual, level, noise;

Link quality, level and noise information

struct player_wifi_data : The complete data packet format is:

player_wifi_link_t links[PLAYER_WIFI_MAX_LINKS];
uint16_t link_count;

A list of links

uint32_t throughput;

mysterious throughput calculated by driver

int32_t bitrate;

current bitrate of device

uint8_t mode;

operating mode of device

uint8_t qual_type;

Indicates type of link quality info we have

uint16_t maxqual, maxlevel, maxnoise;

Maximum values for quality, level and noise.

char ap[32];

MAC address of current access point/cell

Commands

This interface accepts no commands.

Chapter 7

Device Drivers

In this chapter we describe the device drivers included in Player. For each driver, the supported interfaces, configuration requests, and configuration file options are given. The syntax for specifying configuration file options is given in Chapter 4.

7.1 acts

Authors

Brian P. Gerkey gerkey(at)stanford.edu

Synopsis

ACTS is a fast color segmentation system written by Paul Rybski and sold by ActivMedia; see:

```
http://www.activrobots.com
```

After training, ACTS finds colored blobs in a single camera image. Player's `acts` driver provides access to ACTS.

Interfaces

Supported interfaces:

- `blobfinder`

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

Note: In the table below, a default value of (none) indicates that the associated option will not be passed to ACTS. As a result, ACTS's own internal default for that parameter will be used. Consult the ACTS manual to determine what those defaults are.

Name	Type	Default	Meaning
path	string	""	Path to the ACTS executable (leave empty to search the user's PATH for acts).
configfile	string	"/usr/local/acts/actsconfig"	Path to the ACTS configuration file to be used.
version	string	"2.0"	The version of ACTS in use (should be "1.0", "1.2", or "2.0").
width	integer	160	Width of the camera image (in pixels).
height	integer	120	Height of the camera image (in pixels).
pixels	integer	(none)	Minimum area required to call a blob a blob (in pixels).
port	integer	5001	TCP port by which Player should connect to ACTS.
fps	integer	(none)	Frame per second of the camera.
drivertype	string	(none)	Type of framegrabber driver in use (e.g., "bttv", "bt848", "matrox").
invert	integer	(none)	Is the camera inverted?
devicepath	string	(none)	Path to the device file for the framegrabber (e.g., "/dev/fg0").
channel	integer	(none)	Which channel to select on the framegrabber.
norm	string	(none)	Normalization??
pxc200	integer	(none)	Is the framegrabber a PXC200?
brightness	fbat	(none)	Brightness level??
contrast	fbat	(none)	Contrast level??

Notes

- The acts driver supports ACTS versions 1.0, 1.2, and 2.0.
- PXC200 framegrabbers, when accessed through the bttv module, may cause the machine to hang. A workaround is to first read a frame from a framegrabber channel on which there is no video signal, and then start reading from the right channel. (This problem is unrelated to Player's acts driver)

7.2 acoustics

Authors

Nate Koenig nkoenig(at)usc.edu

Synopsis

TODO

Interfaces

Supported interfaces:

- audiodsp

Required devices:

- None.

Supported configuration requests:

- TODO

Configuration file options

TODO

Notes

TODO

7.3 amcl

Authors

Andrew Howard ahoward(at)usc.edu,Boyoon Jung boyoon(at)robotics.usc.edu

Synopsis

The amcl driver implements the Adaptive Monte-Carlo Localization algorithm described by Fox [2]. At the conceptual level, the amcl driver maintains a probability distribution over the set of all possible robot poses, and updates this distribution using data from odometry, sonar and/or laser range-finders. The driver also requires a pre-defined map of the environment against which to compare observed sensor values. At the implementation level, the amcl driver represents the probability distribution using a particle filter. The filter is "adaptive" because it dynamically adjusts the number of particles in the filter: when the robot's pose is highly uncertain, the number of particles is increased; when the robot's pose is well determined, the number of particles is decreased. The driver is therefore able make a trade-off between processing speed and localization accuracy.

As an example, consider the sequence of images shown in Figure 7.1. This sequence shows the filter converging from an initial configuration in which the pose of the robot is entirely unknown to a final configuration in which the pose of the robot is well determined. At the same time, the number of particles in the filter decreases from 100,000 to less than 100.

The amcl driver has the some of the usual features – and failures – associated with simple Monte-Carlo Localization techniques:

- If the robot's initial pose is specified as being completely unknown, the driver's estimate will usually converge to correct pose. This assumes that the particle filter starts with a large number of particles (to cover the space of possible poses), and that the robot is driven some distance through the environment (to collect observations).
- If the robot's initial pose is specified accurately, but incorrectly, or if the robot becomes lost (e.g., by picking it up and replacing it elsewhere) the driver's estimate will not converge on the correct pose. Such situations require the use of more advanced techniques that have not yet been implemented (see [6], for example).

The amcl driver also has some slightly unusual temporal behavior:

- When the number of particles in the filter is large, data may arrive from the sensors faster than it can be processed. When this happens, data is queued up for later processing, but the driver continues to generate an up-to-date estimate for the robot pose. Thus, for example, at time $t = 10$ sec, the driver may have only processed sensor readings up until time $t = 5$ sec, but will nevertheless generate an estimate (prediction) of where the robot is at $t = 10$ sec. The adaptive nature of the algorithm more-or-less guarantees that the driver will eventual "catch up": as more sensor readings are processed, the number of particles will generally decrease, and the sensor update step of the algorithm will run faster.

Caveats

At the time of writing, this driver is still evolving. The sensor models, in particular, are currently over-simplified and under-parameterized (there are lots of magic numbers lurking about the place). Consequently, while this driver is known to work for certain hardware configurations (think Pioneer2DX with a SICKLMS200 laser range-finder), other configurations may require some refinement of the sensor models.

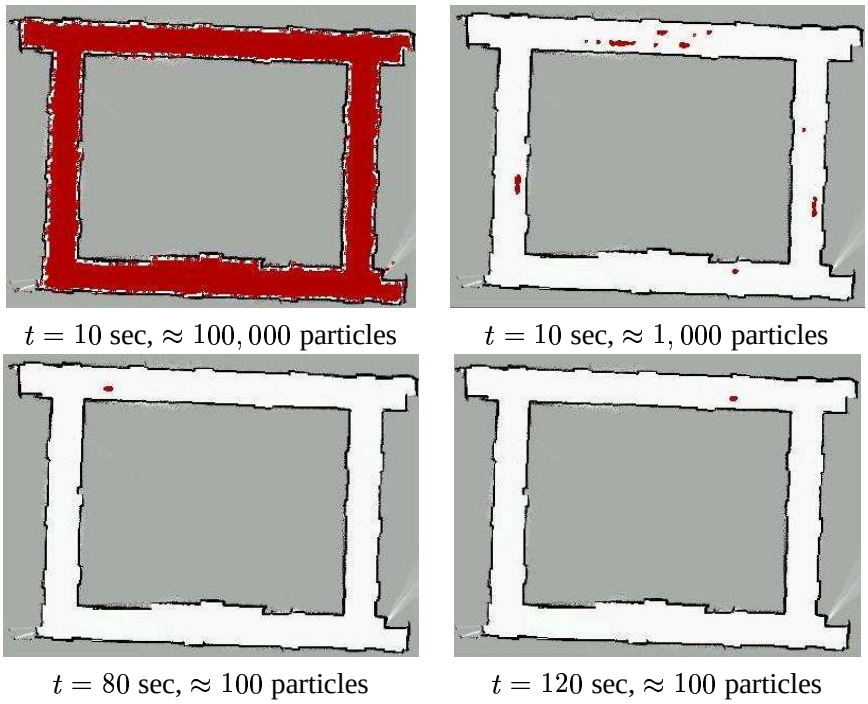


Figure 7.1: Snap-shots showing the `amcl` driver in action; convergence in this case is relatively slow.

Interfaces

Supported interfaces:

- `localize`

Required devices:

- `position`
- Any combination of: `sonar`, `laser` and `wifi`.

Supported configuration requests:

- None.

Configuration file options

Name	Type	Default	Meaning
`position_index`	integer	0	Index of the position device to use (this will usually be an odometric device of some sort).
`sonar_index`	integer	-1	Index of the sonar ranging device to use; set this to -1 if you dont wish to use sonar.
`laser_index`	integer	-1	Index of the laser ranging device to use; set this to -1 if you dont wish to use laser.
`wifi_index`	integer	-1	Index of the WiFi signal-strength device to use; set this to -1 if you dont wish to use WiFi signal-strength.
`map_file`	filename	NULL	Name of the file containing the occupancy map; see notes below for more on the map format.
`map_scale`	length	0.05	Scale of the map (meters/pixel).
`map_negate`	integer	0	Invert the states in the map (occupied becomes empty and empty becomes occupied); see notes below.
`robot_radius`	length	0.20	Effective radius of the robot (meters); this value will be used to eliminate hypotheses that imply that the robot is co-located with an obstacle.
`laser_max_samples`	integer	5	The maximum number of laser range readings to use when updating the filter.
`wifi_beacon_N`	tuple	none	A tuple `[ "hostname" "mapfilename" ]` describing the N^{th} WiFi beacon. `hostname` specifies the name or IP address of the beacon; `mapfilename` points to the WiFi signal strength map for this beacon.
`pf_min_samples`	integer	100	Lower bound on the number of samples to maintain in the particle filter.
`pf_max_samples`	integer	10000	Upper bound on the number of samples to maintain in the particle filter.
`pf_err`	fbat	0.01	Control parameter for the particle set size. See notes below.
`pf_z`	fbat	3	Control parameter for the particle set size. See notes below.
`init_pose`	vector	[0 0 0]	Initial pose estimate (mean value) for the robot (meters, meters, degrees).
`init_pose_var`	vector	$[10^3 10^3 10^2]$	Uncertainty in the initial pose estimate (meters, meters, degrees).
`enable_gui`	integer	0	Set this to 1 to enable the built-in driver GUI (useful for debugging). Player must also be build with `configure --enable-rtkgui` for this option to have any effect.

Notes

- **Maps**

  The odometric, sonar and laser sensor models make use of a common occupancy grid map. This map is a regular grid in which cells are in one of three states: occupied, empty or unknown (although the behavior for unknown cells is currently undefined). Maps are stored as (uncompressed) images in PGM or PNM/grayscale format: black pixels are treated as occupied cells, white pixels are treated as empty cells, and the remaining colors are treated as unknown. The interpretation of these colors may be reversed (white is occupied, black is unknown) by setting the `map_negate` flag in the configuration file. This flag is particularly handy if you wish to use the same image file as both a map and a Stage bitmap.

  TODO: WiFi maps

- **Coordinate System**

  The origin of the global coordinate system corresponds to the center of occupancy grid map. Standard coordinate orientation is used; i.e., positive x is towards the right of the map, positive y towards the top of the map.

- **Number of particles**

  The number of particles in the filter can be controlled using the configuration file parameters `pf_err` and `pf_z`. Specifically, `pf_err` is the maximum allowed error between the true distribution and the estimated distribution, while `pf_z` is the upper standard normal quantile for $(1 - p)$, where p is the probability that the error on the estimated distribution will be less than `pf_err`. If you dont know what that means, dont worry, I'm not exactly sure either. See [2] for a more meaningful explanation.

- **Speed**

  Many factors affect the speed at which the `amcl` driver runs, but the following tips might be helpful:

  – Reducing the number of laser range readings being used (`laser_max_samples` in the configuration file) will significantly increase driver speed, but may also lead to slower convergence and/or less accurate localization.

  – Increasing the allowed error `pf_err` and reducing the quantile `pf_z` will lead to smaller particle sets and will hence increase driver speed. This may also lead, however, to over-convergence.

  As a benchmark, this driver has been successfully deployed on a Pioneer2DX equipped with a SICK LMS200 and a 266MHz Mobile Pentium with 32Mb of RAM.

- **Memory**

  The two key factors affecting memory usage are:

  – The size and resolution of the map.

  – The maximum number of particles.

  As currently configured, the `amcl` driver will typically use 10 to 20Mb of memory. On embedded systems, where memory is at a premium, users may have to decrease the map resolution or the maximum number of particles to achieve acceptable preformance.

Example: Using the `amcl` driver with a Pioneer robot

The following configuration file illustrates the use of the `amcl` driver on a Pioneer robot equipped with a SICK LMS200 scanning laser range finder:

```
position:0
(
   driver "p2os_position"
   port "/dev/ttyS1"
)

laser:0
(
  driver "sicklms200"
  port "/dev/ttyS2"
)
```

```
localize:0
(
  driver "amcl"
  position_index 0
  laser_index 0
  map_file "mymap.pgm"
  map_scale 0.05
)
```

Naturally, the `port`, `map_file` and `map_scale` values should be changed to match your particular configuration.

Example: Using the `amcl` driver with Stage

The `amcl` driver is not supported natively in Stage. Users must therefore employ a second Player server configured to use the `passthrough` driver (see Section 7.31). The basic procedure is as follows.

- Start Stage with a world file something like this:

  ```
  ...
  position (port 6665 laser ())
  ...
  ```

  Stage will create one robot (position device) with a laser, and create a Player server on port 6665.

- Start another Player server using the command

  ```
  player -p 7000 amcl.cfg
  ```

  where the configuration file `amcl.cfg` looks like this:

  ```
  position:0
  (
    driver "passthrough"
    port 6665 index 0
  )

  laser:0
  (
    driver "passthrough"
    port 6665
    index 0
  )

  localize:0
  (
    driver "amcl"
    position_index 0
    laser_index 0
  ```

```
  map_file "cave.pnm"
  map_scale 0.03
  map_negate 1
)
```

The second Player server will listen on port 7000; clients connecting to this server will see a robot with `position`, `laser` and `localize` devices. The map file `cave.pnm` can be the same file used by Stage to create the world bitmap.

Example: Using WiFi signal strength

TODO

7.4 amtecpowercube

Authors

Brian P. Gerkey gerkey(at)stanford.edu

Synopsis

The amtecpowercube driver provides control of an industrial-strength pan-tilt unit called the PowerCube, made by Amtec.

Interfaces

Supported interfaces:

- ptz

Required devices:

- None.

Supported configuration requests:

- PLAYER_PTZ_CONTROL_MODE_REQ

Configuration file options

Name	Type	Default	Meaning
port	string	"/dev/ttyS0"	The serial port to be used
home	integer	0	Whether to home the unit before commanding it
speed	integer	40	Maximum pan/tilt speed (deg/sec)

Notes

- This driver is new and not thoroughly tested.
- For constant swiveling, the PowerCube works better under velocity control.

7.5 cmucam2

Authors

Pouya Bastani

Synopsis

The `cmucam2` driver connects over a serial port to a CMUCam2. Presents a blobfinder interface and can track multiple color blobs. Provides data only; no commands or configs. This driver is rudimentary but working. Color tracking parameters are defined in Player's config file; for example:

```
blobfinder(
  driver "cmucam2"
  devicepath "/dev/ttyS1"
  num_blobs 2
  # values must be between 40 and 240 (!)
  color0 [  red_min red_max blue_min blue_max green_min green_max] )
  # values must be between 40 and 240 (!)
  color1 [  red_min red_max blue_min blue_max green_min green_max] )
)
```

Interfaces

Supported interfaces:

- `blobfinder`

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

TODO

Notes

7.6 cmvision

Authors

Andy Martignoni III ajm7(at)cs.wustl.edu,Brian P. Gerkey gerkey(at)stanford.edu,Brendan Burns bburns(at)cs.umass.edu,Ben Grocholsky bpg(at)grasp.upenn.edu

Synopsis

CMVision (Color Machine Vision) is a fast color-segmentation (aka blob-finding) software library. CMVision was written by Jim Bruce at CMU and is Freely available under the GNU GPL:

http://www-2.cs.cmu.edu/~jbruce/cmvision/

But you don't have to download CMVision yourself, because Player's cmvision driver includes the CMVision code. The cmvision driver provides a stream of camera images to the CMVision code and assembles the resulting blob information into Player's blobfinder data format.

The frame-grabbing portion of the cmvision driver is modular, allowing the user to select the source of camera images. Currently, the following sources are supported (see below for how to select the capture source). Note that support for each source is compiled only if the required libraries and/or kernel features are detected.

- IEEE 1394 (aka Firewire) cameras; requires the libraw1394 and libdc1394 development packages (both are Freely available)
- Video4Linux (aka V4L) cameras; requires V4L headers
- Video4Linux2 (aka V4L2) cameras; requires V4L2 support in your kernel (*currently disabled*)
- A video source that supports Player's internal camera interface, such as the Gazebo camera driver

Interfaces

Supported interfaces:

- blobfinder

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

Name	Type	Default	Meaning
capture	string	"1394"	Capture source (should be "1394", "V4L2", "V4L", or "camera")
colorfile	string	""	(absolute?) path to the CMVision configuration file.
height	integer	240	Height of the camera images (pixels).
width	integer	320	Width of the camera images (pixels).

Notes

- This driver (or at least its underlying video capture code) only works in Linux.
- Consult the CMVision documentation for details on writing a CMVision configuration file.

7.7 er1_position

Authors

David Feil-Seifer dfseifer(at)cs.usc.edu

Synopsis

The er1 driver provides position control of the Evolution Robotics' ER1 and ERSDK robots.

Interfaces

Supported interfaces:

- position

Required devices:

- None.

Supported configuration requests:

- PLAYER_POSITION_GET_GEOM_REQ
- PLAYER_POSITION_MOTOR_POWER_REQ

Configuration file options

Name	Type	Default	Meaning
port	string	"/dev/usb/ttyUSB1"	The serial port to be used
axle	fbat	0.38	The distance between the motorized wheels
motor_dir	-1,1	1	Direction of the motors, if the left motor is plugged in to the motor 1 port on the RCM, put -1 here instead
debug	0,1	0	Put a 1 here if you want to see debug messages

Notes

- This driver is new and not thoroughly tested. The odometry cannot be trusted to give accurate readings.
- You will need a kernel driver to allow the serial port to be seen. This driver, and news about the player driver can be found at "http://www-robotics.usc.edu/ dfseifer/project-erplayer.php".
- TODO: split this driver similar to the way that p2os is split, one main body device, and sub-devices like position, and IR which inherit from this main class. Implement power interface.
- NOT DOING: I don't have a gripper, if someone has code for a gripper, by all means contribute it. It would be welcome to the mix.

7.8 festival

Authors

Brian P. Gerkey gerkey(at)stanford.edu

Synopsis

The festival driver provides access to the Festival speech synthesis system. Festival is available separately (also under the GNU GPL) at: http://www.cstr.ed.ac.uk/projects/festival/. Unlike most drivers, the festival driver queues incoming commands, rather than overwriting them. When the queue is full, new commands are discarded.

Interfaces

Supported interfaces:

- speech

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

Name	Type	Default	Meaning
port	integer	1314	The TCP port on which Player should communicate with Festival.
libdir	string	"/usr/local/festival/lib"	Festival's library directory.
queuelen	integer	4	The length of the incoming command queue.

Notes

7.9 fixedtones

Authors

Esben Østergård,Brian P. Gerkey gerkey(at)stanford.edu

Synopsis

The fixedtones driver provides access to sound hardware, via the Linux OSS interface. Incoming sound is put through a Discrete Fourier Transform, and the frequencies and amplitudes of the five highest peaks in the frequency domain are determined. Note that the FFTW library is required; this package is available (also under the GNU GPL) from: http://www.fftw.org. The fixedtones driver can also produce fixed-tone sounds of given frequency, amplitude, and duration.

Interfaces

Supported interfaces:

- audio

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

No configuration file options are supported.

Notes

- This driver is not widely used and may not function properly.

7.10 flockofbirds

Authors

Toby Collett t.collett(at)auckland.ac.nz

Synopsis

The flockofbirds driver provides a basic interface to the ascension Flock of Birds 6DOF position tracker.

This driver ignores all commands and configuration requests and simple provides a continuous stream of position updates from a single flock of birds controller.

There is currently no support for multiple trackers.

Interfaces

Supported interfaces:

- position3d

Supported configuration requests:

- None.

Configuration file options

- None.

7.11 garminnmea

Authors

Brian P. Gerkey gerkey(at)stanford.edu,Andrew Howard ahoward(at)usc.edu

Synopsis

The garminnmea driver controls a Garmin handheld GPS unit, via a RS232 link. The driver was developed using the Garmin Geko 201, but it should work with other Garmin units. It is unlikely to work with non-Garmin GPS units, as at least one proprietary Garmin NMEA sentences is being used.

This driver is also capable of operating the unit in DGPS mode using RTCM corrections. The driver listens on a network socket for RTCM packets generated by a remote DGPS base station; the packets are forwarded over the serial port to the GPS unit, which responds by switching into DGPS mode. The dgps_server utility found in the utils/dgps_server directory may be used to generate RTCM corrections; corrections are transmitted to clients using UDP multicast.

Interfaces

Supported interfaces:

- gps

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

Name	Type	Default	Meaning
port	string	"/dev/ttyS0"	The serial port to be used
dgps_enable	integer	1	Enable DGPS RTCM forwarding
dgps_group	string	225.0.0.43	Multicast group for RTCM corrections
dgps_port	integer	7778	Port number for RTCM corrections

Notes

7.12 gz_camera

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The gz_camera driver is used to access Gazebo models that support the camera interface (such as the SonyVID model).

Interfaces

Supported interfaces:

- camera

Required devices:

- None.

Configuration file options

Name	Type	Default	Meaning
gz_id	string	NULL	ID of the Gazebo model.

Notes

Consult the Gazebo manual for more information on the Gazebo simulation package.

7.13 gz_gripper

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The gz_gripper driver is used to access Gazebo models that support the gripper interface (such as the SonyVID model).

Interfaces

Supported interfaces:

- gripper

Required devices:

- None.

Configuration file options

Name	Type	Default	Meaning
gz_id	string	NULL	ID of the Gazebo model.

Notes

Consult the Gazebo manual for more information on the Gazebo simulation package.

7.14 gz_laser

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The gz_laser driver is used to access Gazebo models that support the laser interface (such as the SickLMS200 model).

Interfaces

Supported interfaces:

- laser

Required devices:

- None.

Supported configuration requests:

- PLAYER_LASER_GET_GEOM

Configuration file options

Name	Type	Default	Meaning
gz_id	string	NULL	ID of the Gazebo model.

Notes

Consult the Gazebo manual for more information on the Gazebo simulation package.

7.15 gz_position

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The gz_postion driver is used to access Gazebo models that support the position interface (generally speaking, these are robots such as the Pioneer2AT).

Interfaces

Supported interfaces:

- position

Required devices:

- None.

Configuration file options

Name	Type	Default	Meaning
gz_id	string	NULL	ID of the Gazebo model.

Notes

Consult the Gazebo manual for more information on the Gazebo simulation package.

7.16 gz_position3d

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The gz_postion3d driver is used to access Gazebo models that support the position interface (generally speaking, these are robots such as the Pioneer2AT).

Interfaces

Supported interfaces:

- position3d

Required devices:

- None.

Configuration file options

Name	Type	Default	Meaning
gz_id	string	NULL	ID of the Gazebo model.

Notes

Consult the Gazebo manual for more information on the Gazebo simulation package.

7.17 gz_power

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The gz_power driver is used to access Gazebo models that support the power interface.

Interfaces

Supported interfaces:

- power

Required devices:

- None.

Configuration file options

Name	Type	Default	Meaning
gz_id	string	NULL	ID of the Gazebo model.

Notes

Consult the Gazebo manual for more information on the Gazebo simulation package.

7.18 gz_ptz

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The gz_ptz driver is used to access Gazebo models that support the ptz interface.

Interfaces

Supported interfaces:

- ptz

Required devices:

- None.

Configuration file options

Name	Type	Default	Meaning
gz_id	string	NULL	ID of the Gazebo model.

Notes

Consult the Gazebo manual for more information on the Gazebo simulation package.

7.19 gz_sonar

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The gz_sonar driver is used to access Gazebo models that support the sonars interface.

Interfaces

Supported interfaces:

- sonar

Required devices:

- None.

Configuration file options

Name	Type	Default	Meaning
gz_id	string	NULL	ID of the Gazebo model.

Notes

Consult the Gazebo manual for more information on the Gazebo simulation package.

7.20 gz_truth

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The gz_truth driver is used to access Gazebo models that support the truth interface.

Interfaces

Supported interfaces:

- truth

Required devices:

- None.

Configuration file options

Name	Type	Default	Meaning
gz_id	string	NULL	ID of the Gazebo model.

Notes

Consult the Gazebo manual for more information on the Gazebo simulation package.

7.21 khepera

Authors

Toby Collett t.collett(at)auckland.ac.nz

Synopsis

The khepera_* family of drivers are used to interface to the K-Team khepera robot.

This driver is experimental and should be treated with caution. At this point it supports the position and ir interfaces.

Interfaces / Configuration requests

Like the P2OS device, one thread handles 2 separate devices: position and IR.

- khepera_position
  - Interface: position (see Section 6.19)
  - Configurations: GET_GEOM, MOTOR_POWER, VELOCITY_MODE, RESET_ODOM
- khepera_ir
  - Interface: ir (see Section 6.15)
  - Configurations: POSE

Configuration file options

Table 7.1 lists the available configuration file options for the Khepera device. If an option is specified more than once in the config file, then only the first value will be used.

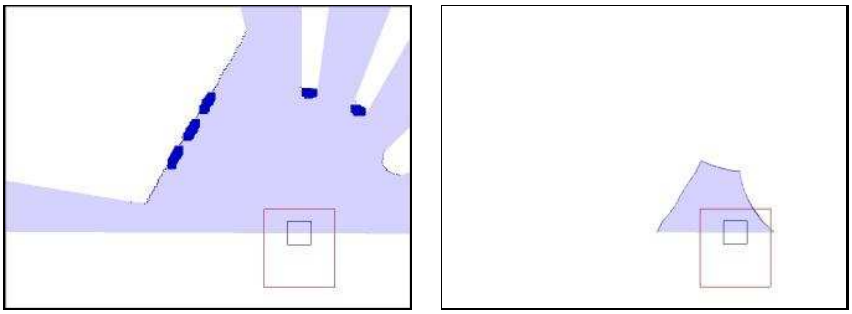
Name	Type	Default	Supported by	Values	Meaning
port	string	/dev/ttyUSB0	khepera_*		This port connects to the Khepera.
scale	fbat	10	khepera_*		As the khepera is so small the actual geom
encoder_res	fbat	1/12	khepera_position		The wheel encoder resolution.
pose	fbat tuple	[0 0 0]	khepera_position		The pose of the robot in player coordinate
size	fbat tuple	[57 57]	khepera_position		The size of the robot approximated to a re
pose_count	int	8	khepera_ir		The number of ir poses.
poses	fbat tuple	Khepera poses	khepera_ir		The pose of each ir sensor [mm,mm,deg]

Table 7.1: Configuration file options for the khepera_* *drivers.*

7.22 lasercspace

Authors

Andrew Howard ahoward(at)usc.edu



(a) (b)

Figure 7.2: (a) Standard laser scan. (b) The corresponding C-space scan for a robot of radius 0.05 m

Synopsis

The lasercspace driver processes a laser scan to compute the configuration space ('C-space') boundary. That is, it shortens the range of each laser scan such that the resultant scan delimits the obstacle-free portion of the robot's configuration space. This driver is particular useful for writing obstacle avoidance algorithms, since the robot may safely move to any point in the obstacle-free portion of the configuration space.

Note that driver computes the configuration space for a robot of some fixed radius; this radius may be set in the configuration file.

Interfaces

Supported interfaces:

- laser

Required devices:

- laser

Supported configuration requests:

- PLAYER_LASER_GET_GEOM

Configuration file options

Name	Type	Default	Meaning
laser	integer	0	Index of the laser device to use.
radius	length	0.50	Robot radius.

Notes

7.23 laserbar

Authors

Andrew Howard ahoward(at)usc.edu

Figure 7.3: A sample laser bar (ignore the colored bands).

Synopsis

The laser bar detector searches for retro-reflective targets in the laser range finder data. Targets can be either planar or cylindrical, as shown in Figure 7.3. For planar targets, the range, bearing and orientation will be determined; for cylindrical targets, only the range and bearing will be determined. The target size and shape can be set in the configuration file.

The range at which targets can be detected is dependant on the target size, the angular resolution of the laser and the quality of the retro-reflective material used on the target.

See also the laserbarcode and laservisualbarcode drivers.

Interfaces

Supported interfaces:

- fiducial

Required devices:

- laser

Supported configuration requests:

- PLAYER_FIDUCIAL_GET_GEOM

Configuration file options

Name	Type	Default	Meaning
laser	integer	0	Index of the laser device to be used.
shape	string	"cylinder"	Target shape: "plane" or "cylinder". Planar fi ducials are currently not supported.
width	length	0.08	Target width (m).

Notes

7.24 laserbarcode

Authors

Andrew Howard ahoward(at)usc.edu

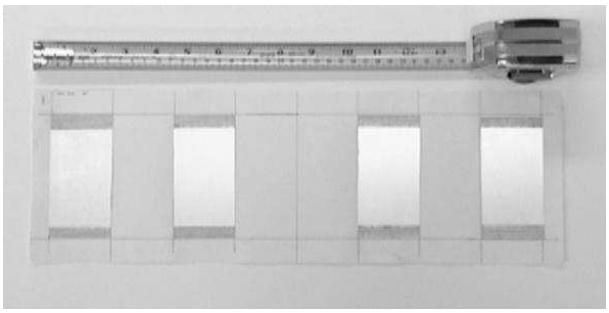


Figure 7.4: A sample laser barcode. This barcode has 8 bits, each of which is 50mm wide.

Synopsis

The laser barcode detector searches for specially constructed barcodes in the laser range finder data. An example laser barcode is shown in Figure 7.4. The barcode is constructed using strips of retro-reflective paper. Each retro-reflective strip represents a '1' bit; each non-reflective strip represents a '0' bit. By default, the laserbarcode driver searches for barcodes containing 8 bits, each of which is exactly 50mm wide (the total barcode width is thus 400m). The first and last bits are used as start and end markers, and the remaining bits are used to determine the identity of the barcode; with an 8-bit barcode there are 64 unique IDs. The number of bits and the width of each bit can be set in the configuration file.

The range at which barcodes can be detected *and identified* is dependent on the bit width and the angular resolution of the laser. With 50mm bits and an angular resolution of 0.5°, barcodes can be detected and identified at a range of about 2.5m. With the laser resolution set to 0.25°, this distance is roughly doubled to about 5m.

See also the laserbar and laservisualbarcode drivers.

Interfaces

Supported interfaces:

- fiducial

Required devices:

- laser

Supported configuration requests:

- PLAYER_FIDUCIAL_GET_GEOM

Configuration file options

Name	Type	Default	Meaning
laser	integer	0	Index of the laser device to be used.
bit_count	integer	8	The number of bits in the barcodes.
bit_width	length	0.05	The width of each bit in the barcode (m).

Notes

For more information on the laserbarcode driver, ask Andrew Howard: ahoward@usc.edu.

7.25 laservisualbarcode

Authors

Andrew Howard ahoward(at)usc.edu

Figure 7.5: A sample laser visual barcode.

Synopsis

The laser visual barcode detector uses both searches for fiducials that are both retro-reflective and color-coded. Fiducials can be either planar or cylindical, as shown in Figure 7.5. For planar targets, the range, bearing, orientation and identity will be determined; for cylindrical targets, the orientation will be undefined. The target size and shape can be set in the configuration file.

The laser visual barcode detector searches the laser range data to find retro-reflective targets, points the camera at each of these targets in turn, then uses color information to determine the presence and identity of fiducials. Thus, this detector makes use of three underlying devices: a laser range finder, a pan-tilt-zoom camera and a color blob detector. Note that the laser is used to determine the geometry of the fidicual (range, bearing and orientation), while the camera is used to determine its identity.

The range at which fiducials can be both detected and identified depends on a number of factors, including the size of the fiducial and the angular resolution of the laser. Generally speaking, however, this detector has better range than the laserbarcode detector, but produces fewer observations.

See also the laserbar and laserbarcode drivers.

Interfaces

Supported interfaces:

- fiducial

Required devices:

- laser
- ptz
- blobfinder

Supported configuration requests:

- PLAYER_FIDUCIAL_GET_GEOM

Configuration file options

Name	Type	Default	Meaning
laser	integer	0	Index of the laser device to be used.
ptz	integer	0	Index of the ptz device to be used.
blobfinder	integer	0	Index of the blobfinder device to be used.
shape	string	"cylinder"	Target shape: "plane" or "cylinder". Planar fiducials are currently not supported.
bit_count	integer	8	The number of bits in the barcode.
bit_width	length	0.05	The width of each bit in the barcode (m).
bit_height	length	0.05	The height of each bit in the barcode (m).

Notes

Setting up the laser-visual barcode detector can be a bit tricky, since it involves so many underlying devices. Users should first check that the laser, ptz and blobfinder devices are working before attempting to use the laservisualbarcode driver. Note that the blobfinder device must be calibrated to detect the particular colors used in the fiducials, and that the identity assigned to each fiducial is determined by the color-to-channel mapping chosen during this configuration.

For more information on the laserbarcode driver, ask Andrew Howard: ahoward@usc.edu.

7.26 lifo-mcom

Authors

Matt Brewer mbrewer(at)andrew.cmu.edu,Reed Hedges reed(at)zerohour.net

Synopsis

The lifo_mcom driver provides a last-in-first-out (LIFO) multi-stack communication system with which clients can exchange data through an instance of Player.

If Pop is called, the last piece of data that was Pushed to the named channel is returned and then deleted. If Read is called the last piece of data added is returned, and left there. Since this is a LIFO stack, if we're reading drive commands, for example, we can be sure to get a "STOP" and interrupt a "FWD" before it's been read.

Interfaces

Supported interfaces:

- mcom

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

No configuration file options are supported.

7.27 linuxwifi

Authors

John Sweeney sweeney(at)cs.umass.edu

Synopsis

The linux_wifi driver provides access to information about wireless Ethernet cards in Linux.

Interfaces

Supported interfaces:

- wifi

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

No configuration file options are accepted.

Notes

This driver simply parses the contents of /proc/net/wireless.

7.28 mixer

Authors

Nate Koenig nkoenig(at)usc.edu

Synopsis

TODO

Interfaces

Supported interfaces:

- audiomixer

Required devices:

- None.

Supported configuration requests:

- TODO

Configuration file options

TODO

Notes

TODO

7.29 nomad

Authors

Richard T. Vaughan vaughan(at)sfu.com,Pawel Zebrowski pzebrows(at)sfu.ca

Synopsis

The nomad driver connects over a serial port to a Nomad 200 robot running Nomadics' "robotd" daemon. Data and commands are handled via the custom nomad interface.

The nomad_position and nomad_sonar drivers provide data in the standard position and sonar interfaces. Position interface provides data, takes commands and supports geometry config. Sonar interface provides data and supports geometry config. This driver is rudimentary but tested and working on the Nomad 200. It may also work on other Nomads that run robotd.

Interfaces

Supported interfaces:

- nomad
- position
- sonar

Required devices:

- None.

Supported configuration requests:

- GET_GEOM (position and sonar)

Configuration file options

nomad:

Name	Type	Default	Meaning
serial_device	string	""	The serial port to be used
serial_speed	int	-1	The baud rate to use

nomad_position:

Name	Type	Default	Meaning
nomad_index	int	0	The index of the underlying nomad device.

nomad_sonar:

Name	Type	Default	Meaning
nomad_index	int	0	The index of the underlying nomad device.

Notes

7.30 p2os

Authors

Brian P. Gerkey gerkey(at)stanford.edu,Kasper Støy,James McKenna

Synopsis

Many robots made by ActivMedia, such as the Pioneer series and the AmigoBot, are controlled by a microcontroller that runs a special embedded operating system called P2OS (on older robots it is called PSOS). The host computer talks to the P2OS microcontroller over a standard RS232 serial line. Player includes a driver that offer access to the various P2OS-mediated devices, logically splitting up the devices' functionality.

Interfaces / Configuration requests

Although all the P2OS interaction is actually done in a single thread, the different P2OS devices are accessed through different Player drivers, each supporting a different interface and supporting some subset of configuration requests:

- p2os_aio:
  - Interface: aio (see Section 6.3)
  - Configurations: none
  - Notes: Provides access to analog User I/O.
- p2os_bumper:
  - Interface: bumper (see Section 6.8)
  - Configurations: none
  - Notes: Provides access to Pioneer bumpers, for those robots so equipped.
- p2os_cmucam:
  - Interface: blobfinder (see Section 6.7)
  - Configurations: SET_COLOR, SET_IMAGER_PARAMS
  - Notes: Controls a CMUCam that is connected to the AUX2 port on the P2OS board. Use the SET_COLOR request to tell the camera which color to track.
- p2os_compass:
  - Interface: position (see Section 6.19)
  - Configurations: none
  - Notes: Fills the compass heading into the yaw field of the position data packet. Accepts no commands.
- p2os_dio:
  - Interface: dio (see Section 6.11)

  - Configurations: none
  - Notes: Provides access to digital User I/O.

- `p2os_gripper`:
  - Interface: `gripper` (see Section 6.14)
  - Configurations: none
  - Notes: Provides access to a Pioneer gripper, for those robots so equipped.

- `p2os_position`:
  - Interface: `position` (see Section 6.19)
  - Configurations: `GET_GEOM`, `MOTOR_POWER`, `VELOCITY_MODE`, `RESET_ODOM`
  - Notes: Provides access to a differential wheelbase. Only speed control is supported.

- `p2os_power`:
  - Interface: `power` (see Section 6.21)
  - Configurations: none
  - Notes: Provides access to battery charge information.

- `p2os_sonar`:
  - Interface: `sonar` (see Section 6.23)
  - Configurations: `GET_GEOM`, `POWER`
  - Notes: Provides access to a sonar array. Sonar indices start with 0 at the front left and increase clockwise.

- `p2os_sound`:
  - Interface: `sound` (see Section 6.24)
  - Configurations: none
  - Notes: Allows you to play pre-recorded sound files on an Amigobot (and other robots?).

Configuration file options

The configuration file options listed in Table 7.2 control how Player communicates with P2OS. Any option can be specified for any of the drivers listed in the previous section; if an option is specified for more than one driver, the value given last will be used.

Name	Type	Default	Meaning
`port`	string	`"/dev/ttyS0"`	The serial port to be used
`radio`	integer	`0`	Nonzero if a radio modem is being used; zero for a direct serial link

Table 7.2: Configuration file options for the `p2os *` *drivers.*

Notes

- The connection to the P2OS microcontroller is only kept open while at least one client has at least one of the P2OS-mediated devices open. When the last P2OS device is closed, the connection to P2OS is also closed. Implications include: odometry is reset to (0,0,0), motors might be turned off.
- Since the P2OS driver uses static C++ class members, only one P2OS robot can be controlled by Player at any given time. If you want to control more than one P2OS robots, you'll need to run a separate instance of Player for each.
- This driver can usually initiate a connection to P2OS even when P2OS was not properly shut down last time. However, if the connection to P2OS is interrupted (e.g., the serial cable is pulled out), then the driver is not likely to recover.
- This driver likely only works in Linux.

7.31 passthrough

Authors

Brian P. Gerkey gerkey(at)stanford.edu

Synopsis

The passthrough driver acts as a *client* to another Player server; it returns data generated by the remote server to client programs, and send commands from the client programs to the remote server. In this way, one Player server can pretend to have devices that are actually located at some other location in the network (i.e., owned by some other Player server). Thus, the passthrough driver makes possible two important capabilities:

- Data from multiple robots can be aggregated in a single Player server; client programs can then talk to more than one robot through a single connection.
- Computationally intensive drivers can be moved off the robot and onto a workstation. Client programs connect to the workstation rather that the robot, but are otherwise unchanged.

See the below for some examples of the passthrough driver in action.

Interfaces

The passthrough driver will support any of Player's interfaces, and can connect to any Player device.

Configuration file options

Name	Type	Default	Meaning
host	string	localhost	Host name for the machine running the remote Player server.
port	integer	6665	Port number for remote server.
index	integer	0	Index of the device on the remote server.

Example: Controlling multiple robots through a single connection

The passthrough driver can be used to aggregate devices from multiple robots into a single server. The following example illustrates the general method for doing

- Imagine that we have to robots named bee and bug. On each robot, start a Player server with the following configuration file:

```
position:0 (driver "p2os_position")
laser:0 (driver "sicklms200")
```

  In this example, the robots are assumed to be Pioneer's with SICK laser range-finders.

- Now imagine that we have a workstation named orac. On this workstation, start another instance of Player with the following configuration file:

```
position:0 (driver "passthrough" host "bee" port 6665 index 0)
laser:0 (driver "passthrough" host "bee" port 6665 index 0)
position:1 (driver "passthrough" host "bug" port 6665 index 0)
laser:1 (driver "passthrough" host "bug" port 6665 index 0)
```

A client connecting to `orac` will see four devices: two `position` devices and two `laser` devices. Both robots can now be controlled through a single connection to `orac`.

Example: Shifting computation

Computationally expensive drivers (such as `adaptive_mcl`) can be shifted off the robot and onto a workstation. The basic method is a straight-forward variant of the example given above.

- Imagine that we have a robot named `bee`. On `bee`, run the Player server with this configuration file:

  ```
  position:0 (driver "p2os_position")
  laser:0 (driver "sicklms200")
  ```

  The robot is assumed to be a Pioneer with a SICK laser range-finder.

- Now imagine that we have a workstation named `orac`. On this workstation, start another instance of Player with the following configuration file:

  ```
  position:0 (driver "passthrough" host "bee" port 6665 index 0)
  laser:0 (driver "passthrough" host "bee" port 6665 index 0)
  localize:0 (driver "adaptive_mcl" position_index 0 laser_index 0 ...)
  ```

  (see Section 7.3 for a detailed description of the additional setings for the `adaptive_mcl` driver). Clients connecting to this server will see a robot with `position`, `laser` and `localize` devices, but all of the heavy computation will be done on the workstation.

Example: Using the `adaptive_mcl` driver with Stage

Some newer drivers, such as the `adaptive_mcl` driver, are not supported natively in Stage. For these drivers users must employ a second Player server configured to use the `passthrough` driver. The basic procedure is as follows.

- Start Stage with a world file something like this:

  ```
  ...
  position (port 6665 laser ())
  ...
  ```

  Stage will create one robot (position device) with a laser, and will start a Player server that listens on port 6665.

- Start another Player server using the command

```
player -p 7000 amcl.cfg
```

  where the configuration file `amcl.cfg` looks like this (see Section 7.3 for a detailed description of the setings for the `adaptive_mcl` driver):

```
position:0 (driver "passthrough" port 6665 index 0)
laser:0 (driver "passthrough" port 6665 index 0)
localize:0 (driver "adaptive_mcl" position_index 0 laser_index 0 ...)
```

  The second Player server will start up and listen on port 7000; clients connecting to this server will see a robot with `position`, `laser` and `localize` devices.

7.32 ptu46

Authors

Toby Collett t.collett(at)auckland.ac.nz

Synopsis

The ptu46 driver provides control of the PTU-46-17.5 pan-tilt unit from directed perceptions through its text interface (This unit is standard on the RWI b21r robot). This driver will probably work with other directed perceptions pan tilt units, please let me know if you have tested it.

Interfaces

Supported interfaces:

- ptz

Required devices:

- None.

Supported configuration requests:

- PLAYER_PTZ_CONTROL_MODE_REQ

Configuration file options

Name	Type	Default	Meaning
port	string	"/dev/ttyS0"	The serial port to be used

Notes

- This driver is new and not thoroughly tested.

7.33 reb

Authors

John Sweeney sweeney(at)cs.umass.edu

Synopsis

The reb_* family of drivers are used to control robots using the K-Team Kameleon 376SBC with Robotics Extension Board (REB). The Kameleon, (or Kam), has a Motorola MC68376 microcontroller that can perform velocity and position control and odometry for up to four motors, using the REB. It can also access a number of A/D inputs, which we have connected to Sharp GP2D12 IR proximity detectors.

In its default setting, a host computer can communicate with the Kam using the K-Team SerCom program, which uses a simple protocol to send commands and read data back. At UMass, we found that the default SerCom did not offer enough performance, so we developed our own, LPRSerCom, which uses the same protocol, but with some enhancements, such as letting the Kam do the odometry updates and IR synchronization. The bottom line is that you need to modifiy these drivers to work with the K-Team SerCom, which is not very difficult (mainly removing the LPRSerCom specific code). We can also send you a copy of LPRSerCom if you'd like. Email John Sweeney (sweeney@cs.umass.edu) for information.

Interfaces / Configuration requests

Like the P2OS device, one thread handles 3 separate devices: position, IR, and power.

- reb_position
  - Interface: position (see Section 6.19)
  - Configurations: GET_GEOM, MOTOR_POWER, VELOCITY_MODE, RESET_ODOM, POSITION_MODE, SPEED_PID, POSITION_PID, SPEED_PROF, SET_ODOM
  - Notes: Provides access to differential wheelbase. Position mode is supported, but experimental. Velocity mode has two operating modes: direct and heading-based. In direct mode, the translational and rotational desired velocities are given as commands. In heading-based, a desired heading an limits on translation and rotational velocities are given.
- reb_ir
  - Interface: ir (see Section 6.15)
  - Configurations: POSE, POWER
  - Notes: Accesses an array of IR proximity detectors. The device returns voltages from the detector, which the client must decode into ranges (usually done in IRProxy). The 8 sensors are arranged in a counterclockwise octagon around the robot, with sensor 0 oriented with the robocentric positive $\hat{x}$ axis, and sensor 2 oriented robocentrically at positive $\hat{y}$.
- reb_power
  - Interface: power (see Section 6.21)
  - Configurations: none
  - Notes: Accesses the current battery voltage information, from the REB.

Configuration file options

Table 7.3 lists the available configuration file options for the REB device. In an option is specified more than once in the config file, then only the last value will be used. Note that the "subclass" option is very UMass specific, since are using two different chassis with different gear ratios.

Name	Type	Default	Supported by	Values	Meaning
`port`	string	`/dev/ttySA1`	`reb_*`		This port connects to the REB.
`subclass`	string	`slow`	`reb_position`	fast, slow	The type of robot.

Table 7.3: Configuration file options for the `reb *` *drivers.*

Notes

- The `reb_position` driver sets some default PID parameters and resets the odometry to $(0, 0, 0)$ when the first client subscribes. Likewise, the IR sensors are only turned on when an IR client has subscribed.
- Position mode is very finicky. This seems to be a problem with the REB itself, which may lose bytes on the serial port while performing position mode actions. This causes the driver to time out, and quite possibly lose a connection to the REB.
- The LPRSerCom protocol running on the REB will sometimes lose a byte over the port, which can cause the driver to time out on a read call to the port. The driver will attempt to retry the call, but there is no guarantee that the REB will be able to handle it. The best solution is to reset the REB. Hopefully this should be a relatively rare occurrence.
- As mentioned above, for this driver to function properly, the REB needs to be running the LPRSerCom program.
- Much of the code for this driver was originally adapted from the `p2os` driver, which we have appreciated.

7.34 rflex

Authors

Matt Brewer mbrewer(at)andrew.cmu.edu, Toby Collett t.collett(at)auckland.ac.nz

Synopsis

The rflex_* family of drivers are used to control RWI robots by directly communicating with RFLEX onboard the robot (i.e., Mobility is bypassed). To date, these drivers have been tested on an ATRV-Jr, but they *should* work with other RFLEX-controlled robots: you will have to determine some parameters to set in the config file, however.

As of March 2003 these drivers have been modified to support the b21r robot, Currently additional support has been added for the power interface and bumper interface. For the pan tilt unit on the b21r please refer to the ptu46 driver. – Toby Collett

Interfaces / Configuration requests

Although all the RFLEX interaction is actually done in a single thread, the different devices are accessed through different Player drivers:

- rflex_position:
  - Interface: position (see Section 6.19)
  - Configurations: VELOCITY_MODE, SET_ODOM, GET_GEOM, MOTOR_POWER, RESET_ODOM
- rflex_sonar:
  - Interface: sonar (see Section 6.23)
  - Configurations: GET_GEOM, POWER
- rflex_power:
  - Interface: power (see Section 6.21)
  - Configurations: none
  - Notes: The power driver seems to report a different value than that on the LCD on the robot so an offset can be added in the configuration file.
- rflex_bumper:
  - Interface: bumper (see Section 6.8)
  - Configurations: GET_GEOM
- rflex_ir:
  - Interface: ir (see Section 6.15)
  - Configurations: POSE, POWER

Some generic devices (e.g. aio and dio) may be available, but are untested.

Configuration file options

For example configuration files, see `umass_ATRVJr.cfg`, `umass_ATRVMini.cfg` and `b21r_rflex_lms200.cfg`.

IMPORTANT: Due to a number of initilisation issues relating to the multipart nature of the rflex driver the configuration option *rflex_done* must be set to 1 in the last rflex driver in the config file. This will cause the server to wait until all the rflex driver options have been parsed before launching its main thread. *The driver will hang if you do not speify this value*

rflex_position

Name	Type	Default	Meaning
`rflex_serial_port`	string	**none**	Serial port connected to RFlex device. See note 5.
`mm_length`	string	**none**	Length of the robot in millimeters
`mm_width`	string	**none**	Width of the robot in millimeters
`odo_distance_conversion`	string	**none**	Odometry conversion. See Note 1.
`odo_angle_conversion`	string	**none**	Odometry conversion. See Note 2.
`default_trans_acceleration`	string	**none**	Set translational acceleration, in mm.
`default_rot_acceleration`	string	**none**	Set rotational acceleration, in radians.

rflex_sonar

Name	Type	Default	Meaning
`rflex_serial_port`	string	**none**	Serial port connected to RFlex device. See note 5.
`range_distance_conversion`	string	**none**	Sonar range conversion factor. See Note 7.
`sonar_age`	string	**none**	Prefi ltering parameter. See Note 3.
`max_num_sonars`	string	**none**	See Note 4.
`num_sonars`	string	**none**	See Note 4.
`num_sonar_banks`	string	**none**	See Note 4.
`num_sonars_possible_per_bank`	string	**none**	See Note 4.
`num_sonars_in_bank`	string	**none**	See Note 4.
`mmrad_sonar_poses`	string	**none**	Sonar positions and directions. See Note 6.

rflex_bumper

Name	Type	Default	Meaning
`rflex_serial_port`	string	**none**	Serial port connected to RFlex device. See note 5.
`bumper_count`	int	**none**	Number of bumper panels
`bumper_def`	fbat tuple	**none**	Tuple of geometry for each bumper
`rflex_bumper_address`	int	**64**	The base address of fi rst bumper in the DIO address range

rflex_ir

Name	Type	Default	Meaning
`rflex_serial_port`	string	**none**	Serial port connected to RFlex device. See note 5.
`rflex_base_bank`	int	**0**	Base IR Bank
`rflex_bank_count`	int	**0**	Number of banks in use
`rflex_banks`	int tuple	**[0]**	Number of IR sensors in each bank
`pose_count`	int	**0**	Total Number of IR sensors
`ir_min_range`	int	**0**	Min range of ir sensors (mm) (Any range below this is returned as 0)
`ir_max_range`	int	**0**	Max range of ir sensors (mm) (Any range above this is returned as max)
`rflex_ir_calib`	fbat tuple	**[1 1]**	IR Calibration data (see Note 8)
`poses`	fbat tuple	**[0]**	x,y,theta of sensors (mm, mm, deg)

rflex_power

Name	Type	Default	Meaning
`rflex_serial_port`	string	**none**	Serial port connected to RFlex device. See note 5.
`rflex_power_offset`	int	**0**	The calibration constant for the power calculation in decivolts

Notes

1. Since the units used by the Rflex for odometry appear to be completely arbitrary, this coefficient is needed to convert to millimeters: mm = (rflex units) / (odo_distance_conversion). These arbitrary units also seem to be different on each robot model. I'm afraid you'll have to determine your robot's conversion factor by driving a known distance and observing the output of the RFlex.

2. Conversion coefficient for rotation odometry: see `odo_distance_conversion`. Note that heading is re-calculated by the Player driver since the RFlex is not very accurate in this respect. See also Note 1.

3. Used for prefiltering: the standard Polaroid sensors never return values that are closer than the closest obstacle, thus we can buffer locally looking for the closest reading in the last "sonar_age" readings. Since the servo tick here is quite small, you can still get pretty recent data in the client.

4. These values are all used for remapping the sonars from Rflex indexing to player indexing. Individual sonars are enumerated 0-15 on each board, but at least on my robots each only has between 5 and 8 sonar actually attached. Thus we need to remap all of these indexes to get a contiguous array of N sonars for Player.

   - `max_num_sonars` is the maximum enumeration value+1 of all sonar meaning if we have 4 sonar boards this number is 64.
   - `num_sonars` is the number of physical sonar sensors - meaning the number of ranges that will be returned by Player.
   - `num_sonar_banks` is the number of sonar boards you have.
   - `num_sonars_possible_per_bank` is probobly 16 for all robots, but I included it here just in case. this is the number of sonar that can be attached to each sonar board, meaning the maximum enumeration value mapped to each board.
   - `num_sonars_in_bank` is the nubmer of physical sonar attached to each board in order - you'll notice on each the sonar board a set of dip switches, these switches configure the enumeration of the boards (ours are 0-3)

5. The first RFlex device (position, sonar or power) in the config file must include this option, and only the first device's value will be used.

6. This is about the ugliest way possible of telling Player where each sonar is mounted. Include in the string groups of three values:`"x1 y1 th1 x2 y2 th2 x3 y3 th3 ..."`. x and y are mm and theta is radians, in Player's robot coordinate system.

7. Used to convert between arbitrary sonar units to millimeters: mm = sonar units / range_distance_conversion.

8. Calibration is in the form Range = $(Voltage/a)^b$ and stored in the tuple as [a1 b1 a2 b2 ...] etc for each ir sensor.

7.35 rwi

Authors

Andy Martignoni III ajm7(at)cs.wustl.edu,Nik Melchior

Synopsis

The rwi_* family of drivers are used to control RWI robots using RWI's Mobility software. To date, these drivers have been tested on the B21r, but they *should* work with other Mobility-controlled robots.

Note:	This documentation may be incomplete and/or wrong, because I (brian) didn't write the driver. Comments and corrections from users are welcome.

Interfaces / Configuration requests

Although all the Mobility interaction is actually done in a single thread, the different Mobility devices are accessed through different Player drivers, each supporting a different interface and supporting some subset of configuration requests:

- rwi_bumper:
  - Interface: bumper (see Section 6.8)
  - Configurations: none
  - Notes: Provides access to RWI bumpers, for those robots so equipped.
- rwi_laser:
  - Interface: laser (see Section 6.16)
  - Configurations: none
  - Notes: Provides access to Mobility-controlled laser range-finder, for those robots so equipped.
- rwi_position:
  - Interface: position (see Section 6.19)
  - Configurations: GET_GEOM, MOTOR_POWER, RESET_ODOM
  - Notes: Provides access to a synchro-drive wheelbase. Only speed control is supported.
- rwi_power:
  - Interface: power (see Section 6.21)
  - Configurations: none
  - Notes: Provides access to battery charge information.
- rwi_sonar:
  - Interface: sonar (see Section 6.23)
  - Configurations: GET_GEOM, POWER
  - Notes: Provides access to a sonar array.

Configuration file options

Name	Type	Default	Meaning
name	string	"B21R"	The name of the robot to which Player should connect

7.36 readlog

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The readlog driver can be used to "replay" data stored in a log file. This is particularly useful for debugging client programs, since users may run their clients against the same data set over and over again. Suitable log files can be generated using the writelog driver, or they may be downloaded from the Robotics Data Set Repository (Radish):

```
http://radish.sourceforge.net
```

Note that, to make use of log file data, Player must be started in a special mode:

```
$ player -r <logfile> <configfile>
```

The -r switch instructs Player to load the given log file, and replay the data according the configuration specified in <configfile>. See the below for some usage examples of the readlog driver.

Interfaces

The readlog driver currently supports the following interfaces: laser, position, wifi.

Configuration file options

Name	Type	Default	Meaning
index	integer	0	Device index in the log file.

Example: Replay Odometry and Laser Data

The following configuration file foo.cfg will read odometry and laser data from a log file:

```
position:0 (driver "readlog" index 0)
laser:0 (driver "readlog" index 0)
```

The Player server must also be started in the appropriate mode:

```
$ player -r foo.log foo.cfg
```

where foo.log contains the data to be replayed. See Section 7.48 for an example that shows how to generate a suitable log file using the writelog driver.

Example: Post-hoc Localization

A particularly useful feature of the `readlog` driver is that it can be used to generate localization information a robot the *after* the experiment has been performed. The following configuration file `bar.cfg` will read odometry and laser data from a log file, and pass it to the `amcl` driver to generate robot pose estimates.

```
position:0 (driver "readlog" index 0)
laser:0 (driver "readlog" index 0)
localize:0
(
  driver "amcl"
  position_index 0
  laser_index 0
  map_file "mymap.pnm"
  map_scale 0.05
)
```

The Player server must also be started in the appropriate mode:

```
$ player -r foo.log bar.cfg
```

7.37 segwayrmp

Authors

John Sweeney sweeney(at)cs.umass.edu,Brian P. Gerkey gerkey(at)stanford.edu,Andrew Howard ahoward(at)usc.edu

Synopsis

The segwayrmp driver provides control of a Segway RMP (Robotic Mobility Platform), which is an experimental robotic version of the Segway HT (Human Transport), a kind of two-wheeled, self-balancing electric scooter.

Interfaces

Supported interfaces:

- position
- position3d

Required devices:

- None

Supported configuration requests:

- For the position interface:
  - PLAYER_POSITION_MOTOR_POWER_REQ
  - PLAYER_POSITION_GET_GEOM_REQ
  - PLAYER_POSITION_RESET_ODOM_REQ
  - PLAYER_POSITION_RMP_VELOCITY_SCALE
  - PLAYER_POSITION_RMP_ACCEL_SCALE
  - PLAYER_POSITION_RMP_TURN_SCALE
  - PLAYER_POSITION_RMP_GAIN_SCHEDULE
  - PLAYER_POSITION_RMP_CURRENT_LIMIT
  - PLAYER_POSITION_RMP_RST_INTEGRATORS
  - PLAYER_POSITION_RMP_SHUTDOWN
- For the position3d interface:
  - PLAYER_POSITION_MOTOR_POWER_REQ

Configuration file options

Name	Type	Default	Meaning
canio	string	"kvaser"	The kind of underlying CAN I/O to be used.
max_xspeed	int	500	Maximum allowed translational velocity (mm/sec).
max_yawspeed	int	40	Maximum allowed angular velocity (deg/sec).

Notes

- Because of its power, weight, height, and dynamics, the Segway RMP is a potentially dangerous machine. Be **very** careful with it.
- This driver is **experimental**, as has **not** been widely used or extensively tested. Use at your own risk.
- Although the RMP does not actually support motor power control from software, for safety you must explicitly enable the motors using a `PLAYER_POSITION_MOTOR_POWER_REQ` request (this request is supported in both `position` and `position3d` modes).
- For safety, this driver will stop the RMP (i.e., send zero velocities) if no new command has been received from a client in the previous 400ms or so. Thus, even if you want to continue moving at a constant velocity, you must continuously send your desired velocities.
- Most of the configuration requests have not been tested.
- The `position3d` interface is entirely new and its use with this driver has not been tested.
- Currently, the only supported type of CAN I/O is `"kvaser"`, which uses Kvaser, Inc.'s CANLIB interface library. This library provides access to CAN cards made by Kvaser, such as the LAPcan II. However, the CAN I/O subsystem within this driver is modular, so that it should be pretty straightforward to add support for other CAN cards.

7.38 serviceadv-lsd

Authors

Reed Hedges reed(at)zerohour.net

Synopsis

The service_adv_lsd driver provides local network broadcast service advertisement using the simple Library for Service Discovery available at http://interreality.org/software/servicediscovery. Clients can link against this library as well, and listen for service advertisements. The LSD library should be built and installed before building Player. Note that there is no client proxy in Player for this device: use the LSD library.

The service URL will have the form: player://*host*:*port*

Interfaces

Supported interfaces:

- service_adv

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

Name	Type	Default	Meaning
name	string	"robot"	The "title" of the service
description	string	"Player Robot Server"	The "description" of the service
url	string	(Automatically detected)	Manually override the service URL
tags	tuple	(Device tags only)	Set additional "type" tags. In addition to these, a tag is added for each device on the robot with the form: device:*device name*#*index*(*driver name*)

7.39 sicklms200

Authors

Andrew Howard ahoward(at)usc.edu,Richard T. Vaughan vaughan(at)sfu.com,Kasper Støy

Synopsis

The sicklms200 driver is used to control a SICK LMS-200 laser range-finder.

Interfaces

Supported interfaces:

- laser

Required devices:

- None.

Supported configuration requests:

- PLAYER_LASER_GET_GEOM
- PLAYER_LASER_GET_CONFIG
- PLAYER_LASER_SET_CONFIG

Configuration file options

Name	Type	Default	Meaning
pose	tuple	[0.0 0.0 0.0]	The mounted pose of the laser (in mm, mm, degrees)
delay	integer	0	Startup delay on the laser, in seconds (set this to 35 if you have a newer generation Pioneer whose laser is switched on when the serial port is open).
port	string	"/dev/ttyS1"	The serial port to be used
rate	integer	38400	Baud rate to use when talking to laser; should be one of 9600, 38400, 500000.
resolution	integer	50	The angular scan resolution to be used (in units of 0.01 degrees)
range_res	integer	1	The range resolution mode of the laser. Set to 1 to get 1 mm resolution with 8 m max range; set to 10 to get 1 cm resolution with 80 m range; set to 100 to get 10 cm resolution with 800 m max range.
invert	integer	0	Set this flag if the laser is mounted upside-down; the range and bearing results will be flipped so the laser appears to be right-way-up.

Notes

- This driver likely only works in Linux.

7.40 sickpls

Authors

Yannick Brosseau yannick.brosseau(at)usherbrooke.ca,Andrew Howard ahoward(at)usc.edu

Synopsis

The sickpls driver is used to control a SICK PLS laser range-finder. This driver will soon be merged into the sicklms200 driver.

Interfaces

Supported interfaces:

- laser

Required devices:

- None.

Supported configuration requests:

- PLAYER_LASER_GET_GEOM
- PLAYER_LASER_GET_CONFIG
- PLAYER_LASER_SET_CONFIG

Configuration file options

Name	Type	Default	Meaning
pose	tuple	[0.0 0.0 0.0]	The mounted pose of the laser (in mm, mm, degrees)
delay	integer	0	Startup delay on the laser, in seconds (set this to 35 if you have a newer generation Pioneer whose laser is switched on when the serial port is open).
port	string	"/dev/ttyS1"	The serial port to be used
rate	integer	9600	Baud rate to use when talking to laser; should be one of 9600, 38400, 500000.
resolution	integer	50	The angular scan resolution to be used (in units of 0.01 degrees)
invert	integer	0	Set this flag if the laser is mounted upside-down; the range and bearing results will be flipped so the laser appears to be right-way-up.

Notes

- This driver likely only works in Linux.

7.41 sonyevid30

Authors

Brian P. Gerkey gerkey(at)stanford.edu

Synopsis

The sonyevid30 driver provides control of a Sony EVID30 pan-tilt-zoom camera unit.

Interfaces

Supported interfaces:

- ptz

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

Name	Type	Default	Meaning
port	string	"/dev/ttyS2"	The serial port to be used
fov	tuple	[3 30]	The minimum and maximum fi elds of view (in degrees) Half-angle??

Notes

- The sonyevid30 operates over a direct serial link, **not** through the P2OS microcontroller's AUX port, as is the normal configuration for ActivMedia robots. You may have to make or buy a cable to connect your camera to a normal serial port.
- This driver likely only works in Linux.

7.42 trogdor

Authors

Brian P. Gerkey gerkey(at)stanford.edu

Synopsis

The trogdor driver provides control of a small, fast, mobile robot made by Botrics. It's called both the *O-Bot* and the *Trogdor*.

Interfaces

Supported interfaces:

- position

Required devices:

- None.

Supported configuration requests:

- PLAYER_POSITION_GET_GEOM_REQ
- PLAYER_POSITION_MOTOR_POWER_REQ

Configuration file options

Name	Type	Default	Meaning
port	string	"/dev/usb/ttyUSB1"	The serial port to be used

Notes

- This driver is new and not thoroughly tested.

7.43 udpbroadcast

Authors

Andrew Howard ahoward(at)usc.edu,Brian P. Gerkey gerkey(at)stanford.edu

Synopsis

The udpbroadcast driver provides a mechanism whereby Player clients can communicate with other Player clients, even when those clients are connected to different Player servers. Any message sent to a udpbroadcast device will be received by all udpbroadcast devices that are on the same physical network (including the sending device). The underlying transport mechanism is based on broadcast UDP sockets (see Notes below).

Interfaces

Supported interfaces:

- comms

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

Name	Type	Default	Meaning
addr	string	"10.255.255.255"	The broadcast address to be used.
port	integer	6013	The broadcast port to be used.

Notes

- Make sure your network supports broadcasting, and that your sys-admin wont cut you off for trying! Broadcasting is best done on private networks.
- The default broadcast address is 10.255.255.255, port 6013 (i.e. it assumes you have a 10.*.*.* network with netmask 255.0.0.0). The broadcast address and port are configurable in Player's configuration file.
- There is no "simulated" udpbroadcast driver in Stage; the real udpbroadcast driver is always used. In this case, the default broadcast address is 127.255.255.255 (the loopback device). At present, there is no way of changing the default value.
- There is no guarantee that messages will be delivered, or that they will be delivered in the exact order they were transmitted.
- The udpbroadcast driver is meant for transmitting small packets only: dont try to use it for passing full-color images around at 30Hz! You will flood your network and overflow both incoming and outgoing queues in the udpbroadcast driver.

7.44 upcbarcode

Authors

Andrew Howard ahoward(at)usc.edu

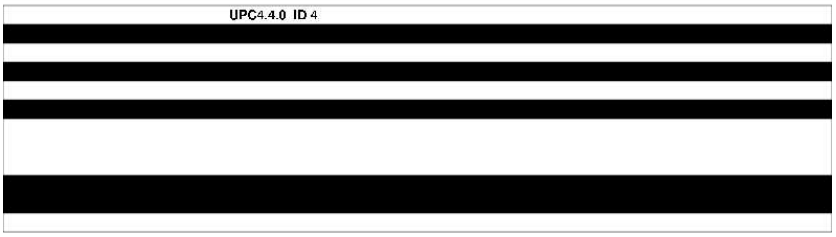


Figure 7.6: A sample UPC barcode (one digit).

Synopsis

The UPC barcode detector searches for standard, single-digit UPC barcodes in a camera image (a sample barcode is shown above).

Interfaces

Supported interfaces:

- blobfinder

Required devices:

- camera

Configuration file options

TODO

Name	Type	Default	Meaning
camera	integer	0	Index of the camera device to be used.

Notes

For more information on the upcbarcode driver, ask Andrew Howard: ahoward@usc.edu.

7.45 vfh

Authors

Chris Jones cvjones(at)robotics.usc.edu

Synopsis

The VFH driver implements the Vector Field Histogram Plus local navigation method by Ulrich and Borenstein [7]. VFH+ provides real-time obstacle avoidance and path following capabilities for fast mobile robots.

Interfaces

Supported interfaces:

- position

Required devices:

- position
- laser

Supported configuration requests:

- None.

Configuration file options

Name	Type	Default	Meaning
position_index	integer	0	Index of the underlying position device.
laser_index	integer	0	Index of the laser device.
cell_size	length	0.1	Local occupancy map grid size (m).
window_diameter	integer	61	Dimensions of occupancy map (map consists of window_diameter X window_diameter cells).
sector_angle	integer	5	Histogram angular resolution, in degrees.
robot_radius	length	0.25	The radius of the robot (m).
safety_dist	length	0.1	The minimum distance the robot is allowed to get to obstacles (m).
max_speed	integer	200	The maximum allowable speed of the robot, in millimeters/second, the robot.
max_turnrate	integer	40	The maximum allowable turnrate of the robot.
free_space_cutoff	fbat	2000000.0	Unitless value. The higher the value, the closer the robot will get to obstacles before avoiding.
weight_desired_dir	fbat	5.0	Bias for the robot to turn to move toward goal position.
weight_current_dir	fbat	3.0	Bias for the robot to continue moving in current direction of travel.
distance_epsilon	length	0.5	Planar distance from the target position that will be considered acceptable.
angle_epsilon	angle	10 degrees	Angular difference from target angle that will considered acceptable.

Notes

The primary parameters to tweak to get reliable performance are `safety_dist` and `free_space_cutoff`. In general, `safety_dist` determines how close the robot will come to an obstacle while turning (around a corner for instance) and `free_space_cutoff` determines how close a robot will get to an obstacle in the direction of motion before turning to avoid. From experience, it is recommeded that `max_turnrate` should be at least 15% of `max_speed`.

To get initiated to VFH, I recommend setting `robot_radius` to the radius of your robot and starting with the default values for other parameters and experimentally adjusting `safety_dist` and `free_space_cutoff` to get a feeling for how the parameters affect performance. Once comfortable, increase `max_speed` and `max_turnrate`. Unless you are familiar with the VFH algorithm, I don't recommend deviating from the default values for `cell_size`, `window_diameter`, or `sector_angle`.

For more information on the `VFH` driver, ask Chris Jones: `cvjones@robotics.usc.edu`.

7.46 waveaudio

Authors

Richard T. Vaughan vaughan(at)sfu.com

Synopsis

The waveaudio driver captures audio from /dev/dsp on systems which support the OSS sound driver. The data is exported using the waveform generic sample data interface. Currently data is captured at 8bit, mono, 16KHz.

Interfaces

Supported interfaces:

- waveform

Required devices:

- None.

Supported configuration requests:

- None.

Configuration file options

No configuration file options are accepted.

Notes

None.

7.47 wavefront

Authors

Brian P. Gerkey gerkey(at)stanford.edu,Andrew Howard ahoward(at)usc.edu

Synopsis

The wavefront driver implements a simple path planner for a planar mobile robot. The underlying planner, which uses wavefront propagation, was written by Andrew Howard; the integration into Player was done by Brian Gerkey.

This driver works in the following way: upon receiving a new position target, a path is planned from the robot's current pose, as reported by the underlying localize device. The waypoints in this path are handed down, in sequence, to the underlying position device.

By tying everything together in this way, this driver offers the mythical "global goto" for your robot.

Interfaces

Supported interfaces:

- position

Required devices:

- position
- localize

Supported configuration requests:

- None.

Configuration file options

Name	Type	Default	Meaning
position_index	integer	0	Index of the underlying position device.
localize_index	integer	0	Index of the underlying localize device.
map_filename	string	none	Bitmap file containing map in which to plan.
map_scale	fbat	none	Meters per pixels in map.
robot_radius	length	0.15	Radius of robot.
safety_dist	length	robot_radius	Distance to keep between robot and obstacles.
max_radius	length	1.0	Distance beyond which is considered infinite for planning purposes (?)
dist_penalty	fbat	1.0	Fudge factor to discourage cutting corners.
dist_epsilon	length	3*robot_radius	Distance from target position that will be considered acceptable.
angle_epsilon	angle	10 degrees	Angular difference from target angle that will be considered acceptable.

Notes

- This driver is new, not widely tested, and non-trivial to configure and use.
- There is currently no way to get feedback from the planner, such as: the current list of waypoints, the lack of a feasible path, or the achievement of the final goal.
- The underlying position device must be capable of doing position control (i.e., not velocity control), preferably with local obstacle avoidance (a very good candidate is the `vfh` driver).
- The underlying localize device must have already converged to a reasonable estimate of the robot's pose before targets are sent to this driver (otherwise results will be unpredictable at best).
- The target thresholds (`dist_epsilon` and `angle_epsilon`) should be *greater* than those thresholds in the underlying position device, assuming it's `vfh`. Otherwise, the underlying driver thinks the robot has reached a target, while the `wavefront` driver is still waiting.

7.48 writelog

Authors

Andrew Howard ahoward(at)usc.edu

Synopsis

The writelog driver can be used to store data from other devices within the Player server. The log files generated in this way can be "replayed" using the readlog driver. See the below for some usage examples of the writelog driver.

Interfaces

The writelog driver currently supports the null interface, meaning that it generates no data, and accepts no commands or configuration requests.

Currently, the writelog driver is capable of storing data from the following interfaces: laser, position, wifi.

Configuration file options

Name	Type	Default	Meaning
filename	string	writelog.log	Output log file name
devices	list	none	A list of strings specifying the devices whose data should be stored. Each device is specified as ''device:index''.

Example: Storing Odometry and Laser Data

The following configuration file will odometry from a Pioneer and laser data from a SickLMS200 to a log file.

```
position:0 (driver "p2os_position")
laser:0 (driver "sicklms200")
null:0
(
  driver "writelog"
  filename "foo.log"
  devices ["position:0" "laser:0"]
  alwayson 1
)
```

Note that the alwayson flag will cause the driver to start logging data as soon as Player is started, and will continue logging data until Player is stopped.

Look in Section 7.36 for an example that shows how to replay this data using the readlog driver.

Chapter 8

Architecture

Player was designed from the beginning to be easily extended by adding new devices and by adding new functionality to existing devices. In fact, Player is really a general-purpose device server; we just happen to use it for controlling our robots. You could use it to provide a simple, clean interface to any sensors or actuators you have. We describe in this chapter the overall system architecture and how you would go about adding your own devices. After reading this chapter, you should consult the code for the existing devices as examples for writing your own. Also, keep in mind that the code may change faster than this document, so the details given here may not always be up to date.

8.1 Server Structure

Player is implemented in C++ and makes use of the POSIX-compliant pthread interface for writing multi-threaded programs. Initially, Player was written with a very large number of threads (2 per client + 1 per device); we found this model to be rather inefficient (especially with LinuxThreads) and poorly scalable due to scheduler delay and context switch time. Thus we have eliminated many threads, keeping the total thread count constant in the number of clients. To support code modularity and reusability there is still generally one thread per device, though some light-weight devices (e.g., the `laserbeacon` device) do not require their own threads.

One thread services all clients, doing the following: listen for new client connections on the selected TCP port(s), read commands and requests from all current clients, and write data and replies to all clients.

When the server receives a request for a device that is not already setup, it calls the proper method, `Setup()`, in the object which controls the indicated device. The invocation of `Setup()` involves spawning another thread to communicate with that device[1]. So, in total, we have 1 server thread and 1 thread per open device.

The overall system structure of Player is shown in Figure 8.1. The center portion of the figure is Player itself; on the left are the physical devices and on the right are the clients. As described above, each client has a TCP socket connection to Player. If the client is executing on the same host as Player, then this socket is simply a loopback connection; otherwise, there is a physical network in between the two. At the other end, Player connects to each device by whatever method is appropriate for that device. For most devices, including the laser, camera, and robot microcontroller, Player makes this connection via an RS-232 serial line. However, connections to the ACTS vision server and Festival speech synthesizer are via a TCP socket.

Within Player, the various threads communicate through a shared global address space. As indicated in Figure 8.1, each device has associated with it a command buffer and a data buffer. These buffers, which are

[1] Most, but not all devices have their own threads.

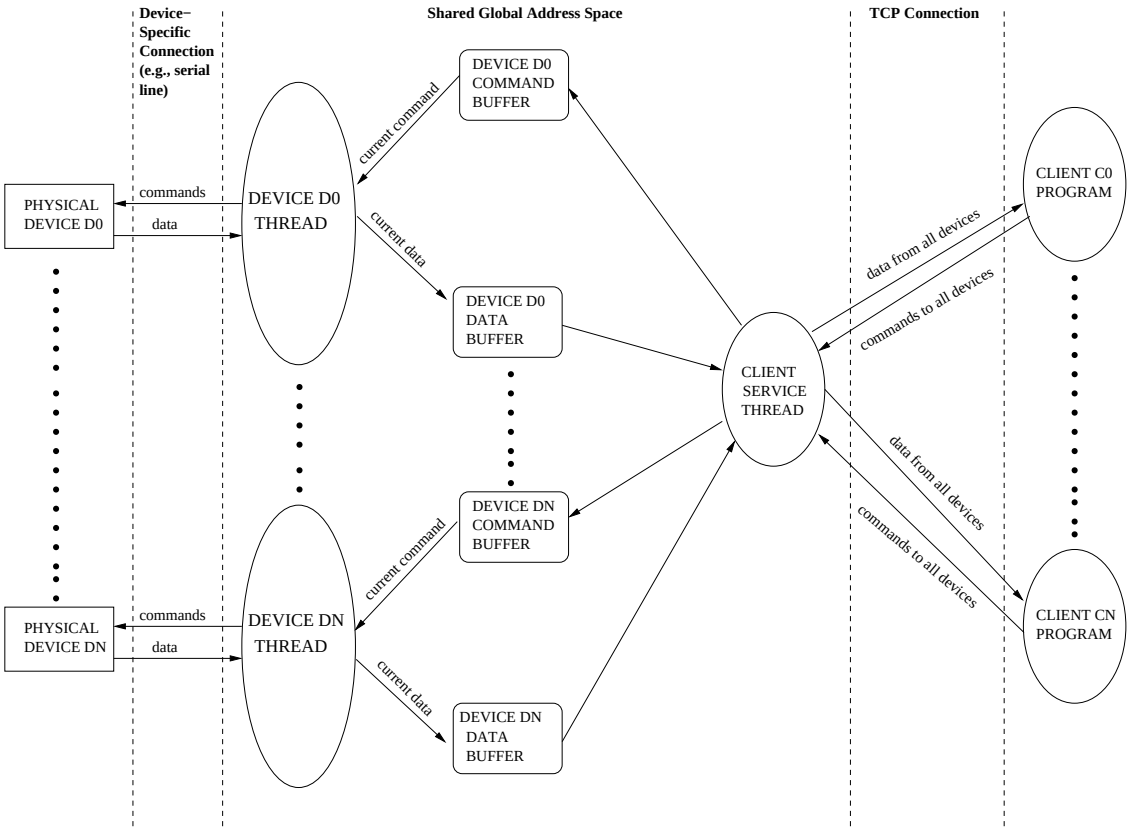


Figure 8.1: Overall system architecture of Player

each protected by mutual exclusion locks, provide an asynchronous communication channel between the device threads and the client reader and writer threads. For example, when the client reader thread receives a new command for a device, it writes the command into the command buffer for that device. At some later point in time, when the device thread is ready for a new command, it will read the command from its command buffer and send it on to the device. Analogously, when a device thread receives new data from its device, it writes the data into its data buffer. Later, when the client writer thread is ready to send new data from that device to a particular client, it reads the data from the data buffer and passes it on to the client. In this way, the client service thread is decoupled from the device service threads (and thus the clients are decoupled from the devices). Also, just by the nature of threads, the devices are decoupled from each other.

8.1.1 Device data

By default, each client will receive new data from each device to which it is subscribed at 10Hz. Of course, receiving data at 10Hz may not be reasonable for all clients; thus we provide a method for changing the frequency, and also for placing the server in a request/reply mode. It is important to remember that even when a client receives data slowly, there is no backlog and it always receives the most current data; it has simply missed out on some intervening information. Also, these frequency changes affect the server's behavior with respect to each client individually; the client at 30Hz and the client at 5Hz can be connected simultaneously, and the server will feed each one data at its preferred rate.

There are four (per-client) modes of data delivery, as follows:

- `PLAYER_DATAMODE_PUSH_ALL` : periodically send to the client current data from all devices cur-

rently opened for reading

- PLAYER_DATAMODE_PULL_ALL : on request from the client, send it current data from all devices currently opened for reading
- PLAYER_DATAMODE_PUSH_NEW : periodically send to the client current data only from those devices that are opened for reading and have generated new data since the last cycle
- PLAYER_DATAMODE_PULL_NEW : on request from the client, send it current data only from those devices that are opened for reading and have generated new data since the last cycle

The default mode is currently PLAYER_DATAMODE_PUSH_NEW, which many clients will find most useful. In general, the *PUSH* modes, which essentially provide continuous streams of data, are good when implementing simple (or multi-threaded) client programs in which the process will periodically block on the socket to wait for new data. Likewise, the *PULL* modes are good for client programs that are very slow and/or aperiodic. Along the other dimension, the *NEW* modes are most efficient, as they never cause "old" data to be sent to the client. However, if a client program does not cache sensor data locally, then it might prefer to use one of the *ALL* modes in order to receive all sensor data on every cycle; in this way the client can operate each cycle on the sensor data in place as it is received.

Of course, it is possible for a device to generate new data faster than the client is reading from the server. In particular, there is no method by which a device can throw an interrupt to signal that it has data ready. Thus the data received by the client will always be slightly older for having sat inside the shared data buffer inside Player. This "buffer sit" time can be minimized by increasing the frequency with which the server is sending data to the client[2]. In any case, all data is timestamped by the originating device driver, preferably as close to the time when the data was gathered from the device. This data timestamp is generally a very close approximation to the time at which the sensed phenomenon occurred and can be used by client programs requiring (fairly) precise timing information.

8.1.2 Device commands

Analogous to the issue of data freshness is the fact that there is no guarantee that a command given by a client will ever be sent to the intended physical device. Player does not implement any device locking, so when multiple clients are connected to a Player server, they can simultaneously write into a single device's command buffer. There is no queuing of commands, and each new command will overwrite the old one; the service thread for the device will only send to the device itself whatever command it finds each time it reads its command buffer. We chose not to implement locking in order to provide maximal power and flexibility to the client programs. In our view, if multiple clients are concurrently controlling a single device, such as a robot's wheels, then those clients are probably cooperative, in which case they should implement their own arbitration mechanism at a higher level than Player. If the clients are not cooperative, then the subject of research is presumably the interaction of competitive agents, in which case device locking would be a hindrance.

8.1.3 Device configurations

Whereas the data and command for each device are stored in simple buffers that are successively overwritten, configuration requests and replies are stored in queues. Configuration requests, which are sent from client to server, are stored in the device's incoming queue. Configuration replies, which are sent from server to client,

[2] On Linux, due to the 10ms scheduler granularity, the effective upper limit on data rate is 50Hz.

are stored in the device's outgoing queue. These queues are fixed-size: queue element size is currently fixed at 1KB for all devices and queue length is determined at compile-time by each device's contructor.

8.2 Adding a new device driver

Having described the internal workings of Player, we now give a short tutorial on how you would go about extending the server by adding a new device driver. As mentioned earlier, in lieu of a more complete prescription for creating drivers, an examination of the code for the existing drivers should provide you with sufficient examples. You should be familiar with C++, class inheritance, and thread programming.

The first step in adding a new driver to Player is to decide which interface(s) it will support. The existing interfaces are described in Chapter 6 and their various message structures and constants are defined in `include/player.h`. Although you can create a new interface, you should try to fit your driver to an existing interface, of which there are many. By deciding to support an existing interface, you'll have less work to do in the server, and will likely have instant client support for your driver in several languages.

To create a new driver, you should create a new class for the driver, which should inherit from `CDevice`, declared in `server/device.h` and implemented in `server/device.cc`. That base class defines a standard interface, part of which the new driver must implement (other parts it may choose to override). We now describe the salient aspects of the `CDevice` class.

8.2.1 Constructors

There are two `CDevice` constructors available. Most drivers will use the more expressive of the two:

```
CDevice::CDevice(size_t datasize, size_t commandsize,
                 int reqqueuelen, int repqueuelen);}
```

Arguments are:

- `datasize` : the size (in bytes) of the buffer to be allocated to hold the current data from the driver
- `commandsize` : the size (in bytes) of the buffer to be allocated to hold the current command for the driver
- `reqqueuelen` : the length (in number of elements) of the queue to be allocated to hold incoming configuration requests for the driver
- `repqueuelen` : the length (in number of elements) of the queue to be allocated to hold outgoing configuration replies from the driver

All requested buffers and queues will be allocated by the `CDevice` constructor (we describe below where the pointers are stored in the object). This constructor should be invoked in the preamble to the driver's own constructor; for example, the `sicklms200` driver, which produces fixed-length data, accepts no commands, and uses incoming and outgoing configuration queues both of length 1, has a constructor that begins:

```
CLaserDevice::CLaserDevice(int argc, char** argv) :
  CDevice(sizeof(player_laser_data_t),0,1,1)
```

Now, you may want to allocte your own buffers and/or queues (e.g., CStageDevice does its own memory management in static `mmap()`ed segments). If so, then your driver should not invoke a `CDevice` constructor; the "default" zero-argument constructor will be invoked for you and will properly initialize some class

members. Even if you do allocate your own buffers, you might benefit from letting `CDevice` know where they are, in that you could still use the standard `CDevice` methods (described below) to interface with your driver. You can do this by calling (usually in your own constructor) `SetupBuffers()`:

```
void CDevice::SetupBuffers(unsigned char* data, size_t datasize,
                           unsigned char* command, size_t commandsize,
                           unsigned char* reqqueue, int reqqueuelen,
                           unsigned char* repqueue, int repqueuelen);
```

Arguments are:

- `data` : the buffer allocated to hold the current data from the driver
- `datasize` : the size (in bytes) of `data`
- `command` : the buffer allocated to hold the current command for the driver
- `commandsize` : the size (in bytes) of `command`
- `reqqueue` : the buffer allocated to hold incoming configuration requests; it should be an allocated as an array of `playerqueue_elt_t`s
- `reqqueuelen` : the length (in number of elements) of `reqqueue`
- `repqueue` : the buffer allocated to hold outgoing configuration replies; it should be an allocated as an array of `playerqueue_elt_t`s
- `repqueuelen` : the length (in number of elements) of `repqueue`

In this case, `SetupBuffers()` will allocate `PlayerQueue` objects to handle configurations; they will operate on the memory segments that you provide.

Whether you let the `CDevice` constructor allocate your buffers or do it yourself and then call `SetupBuffers()`, the relevant pointers and sizes are stored in the following protected members of `CDevice`:

```
// buffers for data and command
unsigned char* device_data;
unsigned char* device_command;

// maximum sizes of data and command buffers
size_t device_datasize;
size_t device_commandsize;

// amount at last write into each respective buffer
size_t device_used_datasize;
size_t device_used_commandsize;

// queues for incoming requests and outgoing replies
PlayerQueue* device_reqqueue;
PlayerQueue* device_repqueue;
```

8.2.2 Locking access to buffers

Because Player is a multi-threaded program, all access to shared buffers must be protected by mutual exclusion locks. For this purpose, `CDevice` contains a (private) `pthread_mutex_t`, and provides (protected) `Lock()` and `Unlock()` methods that call `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively (these methods are virtual; thus a driver can override them in order to use a different mutual exclusion mechanism). You should surround all accesses to any of a driver's shared buffers or queues with calls to `Lock()` and `Unlock()`. The default interface methods described in the following sections do exactly this.

8.2.3 Instantiation

Because Player supports multiple indexed instances of devices, your driver should be prepared to be multiply instantiated (e.g., you generally should not use global or other static variables) and you must provide a function for instantiating it. When a new instance of a driver is required, Player will call an appropriate instantiation function (see Section 8.2.10 for how to register your instantiation method). This function should return (as a `CDevice*`) a pointer to a new instance of your device class. Since an object has not yet been created when this function is called, it must be declared outside of the class (or static within the class). This function should match the following prototype:

```
CDevice* Foo_Init(char* interface, ConfigFile* cf, int section);
```

The arguments are:

- `interface`: the string name of the interface that the driver has been requested to support
- `cf`: a object containing information gleaned from the user's configuration file
- `section`: in which section of the configuration file your driver was requested

You should check the requested `interface` to be sure that your driver can support it. If you cannot, then return `NULL`. You should look in the configuration file object `cf` for any options that may have been specified for you driver (look at existing drivers and `server/configfile.h` for how to get options out).

8.2.4 Setup

When the first client subscribes to a device, the driver's `Setup()` method is called. This method is set to NULL in CDevice:

```
virtual int CDevice::Setup() = 0;
```

Thus every driver *must* implement this method. After doing whatever is required to initialize the device (e.g., open a serial port and spawn a thread to interact with it), `Setup()` should return either zero to indicate that the device was successfully setup, or non-zero to indicate that setup failed. Since clients may immediately request data and since they may never send commands, a driver's data buffer and command buffer should be sensibly "zeroed" in `Setup()`.

8.2.5 Shutdown

When the last client unsubscribes from a device, the driver's `Shutdown()` method is called. This method is set to NULL in CDevice:

```
virtual int CDevice::Shutdown() = 0;
```

Thus every driver *must* implement this method. After doing whatever is required to stop the device (e.g., kill a thread and close a serial port), `Shutdown()` should return either zero to indicate that device was successfully shutdown, or non-zero to indicate that shutdown failed.

8.2.6 Thread management

In order to leverage parallelism, most (but not all) devices use separate threads to do their work. Because thread creation is not intuitively compatible with C++ object context, some support is provided in `CDevice` for starting and stopping threads. You are not required to use this support, but you might find it useful.

The first step is to define in your class a public method `Main()` that overrides the virtual declaration:

```
virtual void CDevice::Main();
```

Presumably your definition of `Main()` will contain a loop that executes all device interaction. When you want to start your thread (probably in `Setup()`), call `StartThread()`. A new thread will be created; in that thread your driver's `Main()` method will be invoked, with the proper context of your driver's object. When you want to stop your thread (probably in `Shutdown()`), call `StopThread()`, which will cancel and join your thread; thus your thread should respond to cancellation requests (even if it initially defers them) and should be in a joinable state (i.e., it should *not* be detached).

8.2.7 Data access methods

Most drivers can use the default `CDevice` *implementations of the following methods; however, they are virtual and can be overridden if necessary.*

PutData

Whenever your driver has new data, it should call `PutData()`:

```
virtual void CDevice::PutData(unsigned char* src, size_t len,
                      uint32_t timestamp_sec, uint32_t timestamp_usec);
```

Arguments are:

- `src` : pointer to the new data
- `len` : length (in bytes) of the new data
- `timestamp_sec` : the time at which the new data was produced
- `timestamp_usec` : the time at which the new data was produced

The default implementation of `PutData()` will `Lock()` access, `memcpy()` your new data into `device_data`, save your `len` and timestamp, and `Unlock()` access. If `ts` is `NULL`, then `PutData()` will use the current time (either system or simulator, as appropriate).

GetNumData

When a client wants to read data from your driver, the server will first call `GetNumData()`:

```
virtual size_t GetNumData(void* client);
```

Arguments are:

- `client` : a unique id for who wants the data

This method should return the number of data packets that are ready for the given client at this time. The server will then call `GetData()` that many times. The default implementation of `GetNumData()` simply returns 1, which is almost always the right thing. However, your driver can override this method if you want.

GetData

When a client wants to read data from your driver, the server will invoke `GetData()`:

```
virtual size_t GetData(void* client, unsigned char* dest, size_t len,
                       uint32_t* timestamp_sec, uint32_t* timestamp_usec);
```

Arguments are:

- `client` : a unique id for who wants the data
- `dest` : pointer to a buffer into which to copy the data
- `len` : length (in bytes) of `dest`
- `timestamp_sec` : buffer into which to copy the time at which the data was
- `timestamp_usec` : buffer into which to copy the time at which the data was produced

The default implementation of `GetData()` will `Lock()` access, `memcpy()` data from `device_data` into `dest` (up to `len` bytes), retrieve timestamp information, `Unlock()` access, and return how many bytes of data were copied.

8.2.8 Command access methods

Most devices can use the default `CDevice` *implementations of the following methods; however, they are virtual and can be overridden if necessary.*

PutCommand

When a client sends a new command for your device, the server will invoke `PutCommand()`:

```
virtual void PutCommand(void* client, unsigned char* src, size_t len);
```

Arguments are:

- `client` : a unique id for the source of the command
- `src` : pointer to the new command
- `len` : length (in bytes) of the new command

The default implementation of `PutCommand()` will `Lock()` access, `memcpy()` the new command into `device_command`, save `len`, and `Unlock()` access.

GetCommand

When you want the current command for your device, you should call `GetCommand()`:

```
virtual size_t CDevice::GetCommand(unsigned char* dest, size_t len);
```

Arguments are:

- `dest` : pointer to a buffer into which to copy the command
- `len` : length (in bytes) of `dest`

The default implementation of `GetCommand()` will `Lock()` access, `memcpy()` the command from `device_command` into `dest` (up to `len` bytes), `Unlock()` access, and return how many bytes of command were copied.

8.2.9 Configuration access methods

Most drivers can use the default `CDevice` *implementations of the following methods; however, they are virtual and can be overridden if necessary.*

PutConfig

When a new configuration request arrives for your device, the server will invoke `PutConfig()`:

```
virtual int CDevice::PutConfig(player_device_id_t* device,
                               void* client,
                               void* data,
                               size_t len);
```

Arguments are:

- `device` : an identifier that indicates the device for whom the request is intended
- `client` : a tag that will be used to route the reply back to the right client (or other device)
- `data` : buffer containing the new request
- `len` : length (in bytes) of the new request

The default implementation of `PutConfig` will `Lock()` access, push the request onto `device_reqqueue`, and `Unlock()` access. If all is well, then zero is returned; otherwise (e.g., if the queue is full) non-zero is returned and the server will send an error response message to the waiting client.

GetConfig

To check for new configuration requests in your device, you should call `GetConfig()`:

```
virtual size_t CDevice::GetConfig(player_device_id_t* device,
                                  void** client,
                                  void *data,
                                  size_t len);
```

Arguments are:

- `device` : an identifier that indicates the device for whom the request is intended (useful when one queue is used for multiple devices, as is the case with P2OS)
- `client` : a place to store a tag that will be used to route the reply back to the right client (or other device)
- `data` : buffer into which to copy the new request
- `len` : length (in bytes) of `data`

For convenience, there is also a short form of `GetConfig()`:

```
virtual size_t CDevice::GetConfig(void** client,
                                  void* data,
                                  size_t len);
```

The default implementation of `GetConfig` will `Lock()` access, pop a request off `device_reqqueue`, and `Unlock()` access. If there was a request to be popped then the size of the request is returned; otherwise zero is returned, indicating that there are no pending requests. If there is request then hang onto `client` because you will need to pass it back in `PutReply()`.

PutReply

After servicing a request, you must generate an appropriate reply; you do this by calling `PutReply()`:

```
virtual int CDevice::PutReply(player_device_id_t* device,
                              void* client,
                              unsigned short type,
                              struct timeval* ts,
                              void* data,
                              size_t len);
```

Arguments are:

- `device` : an identifier that indicates from which the device the reply comes (useful when one queue is used for multiple devices, as is the case with P2OS)
- `client` : the tag that you received in `GetConfig`
- `type` : the type of the reply (see below)
- `ts` : pointer to time at which the configuration was executed
- `data` : the reply itself (if any)
- `len` : length (in bytes) of `data`

There are also two short forms of `PutReply()`:

```
virtual int CDevice::PutReply(void* client,
                              unsigned short type,
                              struct timeval* ts,
                              void* data,
```

```
                                size_t len);

virtual int CDevice::PutReply(void* client,
                                unsigned short type);
```

The first short form assumes that the caller is the originator of the reply. The second short form further assumes that the reply is zero-length and that it should be stamped with the current time.

The default implementation of `PutReply` will `Lock()` access, push the reply onto `device_repqueue`, and `Unlock()` access. If the reply queue is full (which should not happen in practice) then -1 is returned; otherwise a non-negative integer is returned. The given `type` will be used as the message type for the reply that will be sent to the client; it should be one of:

- `PLAYER_MSGTYPE_RESP_ACK` : the configuration was successful
- `PLAYER_MSGTYPE_RESP_NACK` : the configuration failed

If `ts` is `NULL`, then the current time is filled in. Zero-length replies are valid (and frequent).

GetReply

The server will periodically check for replies from your device by calling `GetReply()`:

```
virtual int CDevice::GetReply(player_device_id_t* device,
                                void* client,
                                unsigned short* type,
                                struct timeval* ts,
                                void* data,
                                size_t len);
```

Arguments are:

- `device` : an identifier indicating from which the device the reply has come
- `client` : a tag to be matched
- `type` : place to copy the type of the reply
- `ts` : place to copy the time at which the configuration was executed
- `data` : buffer into which to copy the reply
- `len` : length (in bytes) of `data`

The default implementation of `GetReply()` will `Lock()` access, pop a reply off `device_repqueue`, `Unlock()` access, and return the length of the reply. Because zero-length replies are valid, `GetReply()` will return -1 to indicate that no reply is available.

8.2.10 Registering your device

In order to inform the server about the availability of your driver, you must add it to `driverTable`, a global table of drivers that may be instantiated. You should add your driver by calling `AddDevice()` in the function `register_devices()`, declared in `server/deviceregistry.cc`:

```
int AddDriver(char* name, char access,
              CDevice* (*initfunc)(char*,ConfigFile*,int));
```

Arguments are:

- `name` : the name of your driver
- `access` : the allowable access mode for your driver; should be one of:
  - `PLAYER_READ_MODE`
  - `PLAYER_WRITE_MODE`
  - `PLAYER_ALL_MODE`
- `initfunc` : a function that can be used to create a new instance of your device (see Section 8.2.3)

You may find it convenient to write a registration function, e.g.:

```
void SickLMS200_Register(DriverTable* table)
{
  table->AddDriver("sicklms200", PLAYER_READ_MODE, SickLMS200_Init);
}
```

You should also `#include` your device's class header in `deviceregistry.cc`. To encourage modularity of the server by allowing drivers to be left out at compile-time, it is customary to make both the `#include` and `AddDriver()` conditioned on compiler directives. For example:

```
#ifdef INCLUDE_SICK
void SickLMS200_Register(DriverTable* table);
#endif

...


void
register_devices()
{
  ...

  #ifdef INCLUDE_SICK
    SickLMS200_Register(driverTable);
  #endif

  ...
}
```

8.2.11 Compiling your device

Finally, you need to compile your device and link it into the server binary. You really need to know something about GNU Autotools to do this. As such, look at how the existing drivers are linked in.

8.2.12 Building a shared library

As an alternative to statically linking your device driver directly into the Player binary, you can build your driver as a shared object and have Player load it at run-time. If you choose to take this path, then you should still follow most of the directions given in the previous sections, except for the registration and compilation details in Sections 8.2.10–8.2.11.

Instead of registering your device in `deviceregistry.cc`, you should do so in an initialization function that will be invoked by the loader. You must declare this initialization function, as well as a finalization function, in your driver code. For example, in order to build the `sicklms200` driver as a shared object, the following code is added to `sicklms200.cc`:

```
#include <drivertable.h>
extern DriverTable* driverTable;

/* need the extern to avoid C++ name-mangling  */
extern "C" {
void _init(void)
{
  driverTable->AddDriver("sicklms200", PLAYER_READ_MODE, SickLMS200_Init);
}

void _fini(void)
{
  /* probably don't need any code here; the destructor for the device
   * will be called when Player shuts down.  this function is only useful
   * if you want to dlclose() the shared object before Player exits
   */
}
}
```

The `_init()` function will be invoked by the loader when Player calls `dlopen()` to load your driver. The `_fini()` function will be invoked when the library is `dlclose()`ed; however, Player never closes your library explicitly, so `_fini()` will be called when Player exits.

The details of building a shared object vary from system to system, but the following example, which works with g++ on Linux, should get you started:

```
$ g++ -Wall -DPLAYER_LINUX -g3 -I$PLAYER_DIR/server -c sicklms200.cc
$ g++ -shared -nostartfiles -o sicklms200.so sicklms200.o
```

Having built your driver library, tell Player on the command-line to load it (as described in Section 2.2), e.g.:

```
$ player -d sicklms200.so
```

Note that the dynamic loading functionality is still somewhat experimental, and is not currently used by any core Player drivers. However, it should work. If you use shared libraries, please let us know about your experiences.

Bibliography

[1] ActivMedia Robotics. *Pioneer 2 Gripper Manual*. http://robots.activmedia.com/docs/all_docs/gripmanP2_3.pdf.

[2] D. Fox. KLD-sampling: Adaptive particle filters. In *Advances in Neural Information Processing Systems 14*. MIT Press, 2001.

[3] Brian P. Gerkey, Maja J Matarić, and Gaurav S Sukhatme. Exploiting physical dynamics for concurrent control of a mobile robot. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 3467–3472, Washington D.C., May 2002.

[4] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proc. of the Intl. Conf. on Advanced Robotics (ICAR)*, pages 317–323, Coimbra, Portugal, July 2003.

[5] Brian P. Gerkey, Richard T. Vaughan, Kasper Støy, Andrew Howard, Gaurav S Sukhtame, and Maja J Matarić. Most Valuable Player: A Robot Device Server for Distributed Control. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 1226–1231, Wailea, Hawaii, October 2001.

[6] Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert. Robust Monte Carlo localization for mobile robots. *Artificial Intelligence Journal*, 128(1–2):99–141, 2001.

[7] Iwan Ulrich and Johann Borenstein. Vfh+: Reliable obstacle avoidance for fast mobile robots. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 1572–1577, Leuven, Belgium, May 1998.

[8] Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 2121–2427, Las Vegas, Nevada, October 2003.

Appendix A

The C Client Interface

Included with Player is a simple, no-frills client interface library written in ANSI C (`client_libs/c`). This client is intentionally primitive and most users will find it inconvenient for writing anything more that the simplest control program. Rather than direct use, the C client should be considered the reference implementation of a Player client library and should be consulted for networking details when writing new clients in other languages (the C client can also be used directly as a low-level substrate for other clients; the C++ client is implemented in this way). In the file `playerccclient.c` are defined the 5 device-neutral functions necessary in any client:

- `player_connect()` : connect to the server
- `player_disconnect()` : disconnect from the server
- `player_read()` : read one data packet from the server
- `player_write()` : write one command packet to the server
- `player_request()` : send a request packet and wait for the reply

In addition, the *very* useful helper function `player_request_device_access()` (a special case of `player_request()` that obtains access to devices) is defined.

In the files `print.c` and `helpers.c` are defined some device-specific functions that simplify direct use of the C client; we do not document them here.

A.1 Debug Information

Before getting to the core functions of the C client, we note that they can each generate some debug information to `stderr`. This information is generally helpful, especially in diagnosing problems; you should probably not disable it. However, a function is provided for adjusting the amount of debug information that is printed:

```
/*
 * use this to turn off debug ouput.
 *
 * higher numbers are more output, 0 is none.
 *
 * incidentally, if <level> is -1, it returns the current level and
 * the current level is unchanged
```

```
 */
int player_debug_level(int level);
```

The default debug level is 5, which prints all messages. This same function is used to vary the debug output from the C++ client, since it is built on top of the C client.

A.2 Connecting to the Server

The C client makes no use of static data structures for maintaining connection information; rather the user must always supply a pointer to a properly initialized connection structure, of type `player_connection_t`. This structure is initialized by `player_connect()`:

```
/*
 * connects to server listening at host:port.  conn is filled in with
 * relevant information, and is used in subsequent player function
 * calls
 *
 * Returns:
 *    0 if everything is OK (connection opened)
 *   -1 if something went wrong (connection NOT opened)
 */
int player_connect(player_connection_t* conn, const char* host, int port);
```

Note that the connection will be blocking. A simple example:

```
...
player_connection_t conn;

/* Connect to Player server */
if(player_connect(&conn, "localhost", PLAYER_PORTNUM))
  exit(1);
...
```

A.3 Requesting Device Access

To ease the common process of requesting read and write access to devices, the C client includes the function `player_request_device_access()`:

```
...
/*
 * issue a single device request (special case of player_request())
 *
 *
 * if grant_access is non-NULL, then the actual granted access will
 * be written there.
 *
 *   Returns:
 *      0 if everything went OK
 *     -1 if something went wrong (you should probably close the
 *     connection!)
 */
```

```
int player_request_device_access(player_connection_t* conn,
                                 uint16_t device,
                                 uint16_t device_index,
                                 uint8_t req_access,
                                 uint8_t* grant_access);
```

The following code fragment obtains 'all' (`'a'`) access to the `ptz` device.

```
...
/* Request 'all' access to the ptz device */
if(player_request_device_access(&conn, PLAYER_PTZ_CODE, 0, 'a',NULL) == -1)
  exit(1);
...
```

A.4 Reading Data

For obtaining data from devices, the C client provides the rather simple function `player_read()`:

```
/*
 * read from the indicated connection.  put the data in buffer, up to
 * bufferlen.
 *
 * Returns:
 *    0 if everything went OK
 *   -1 if something went wrong (you should probably close the connection!)
 */
int player_read(player_connection_t* conn, player_msghdr_t* hdr,
                char* payload, size_t payloadlen);
```

This function will read one data packet from the server, blocking if no packet is available. It is the caller's responsibility to provide sufficient storage for the header and payload, and `player_read()` will not overrun the provided buffers. After calling `player_read()`, the user will presumably examine the fields of the header in order to know how to process the payload. Note that `player_read()` will appropriately byte-swap all fields in the header, but will not transform the contents of the payload in any way. A simple example:

```
...
player_msghdr_t hdr;
char data[PLAYER_MAX_MESSAGE_SIZE];

if(player_read(&conn, &hdr, data, sizeof(data)) == -1)
  exit(1);
...
```

A.5 Writing Commands

For writing commands to devices, the C client provides the function `player_write()`:

```
/*
 * write commands to the indicated connection. writes the data contained
 * in command, up to commandlen.
```

```
 *
 * Returns:
 *    0 if everything goes OK
 *   -1 if something went wrong (you should probably close the connection!)
 */
int player_write(player_connection_t* conn,
                 uint16_t device, uint16_t device_index,
                 const char* command, size_t commandlen);
```

This function will build the appropriate message header, including appropriate byte-swapping of the fields. The first `commandlen` bytes of `command` will be copied in as the payload of a messge that will be sent to the server. Note that the caller must byte-swap the contents of the command itself. A simple example that tells the 0th `position` device to spin in place:

```
...
player_position_cmd_t cmd;
cmd.speed = htons(0);
cmd.turnrate = htons(40);
if(player_write(&conn, PLAYER_POSITION_CODE, 0, (char*)&cmd,
                sizeof(player_position_cmd_t)) == -1)
  exit(1);
...
```

A.6 Requesting Configuration Changes

For requesting configuration changes to devices, the C client provides the function `player_request()`:

```
/*
 * issue some request to the server. requestlen is the length of the
 * request.  reply, if non-NULL, will be used to hold the reply; replylen
 * is the size of the buffer (player_request() will not overrun your buffer)
 *
 *   Returns:
 *      0 if everything went OK
 *     -1 if something went wrong (you should probably close the connection!)
 */
int player_request(player_connection_t* conn,
                   uint16_t device, uint16_t device_index,
                   const char* payload, size_t payloadlen,
                   player_msghdr_t* replyhdr, char* reply, size_t replylen);
```

This function will build the proper message header, including appropriately byte-swapping the header fields. The caller is responsible for byte-swapping the contents of the `payload`, which will be copied in as the payload of a message that will be sent to the server. After sending the request, `player_request()` will wait for the matching reply (consuming and discarding all intervening messages) before returning. If the caller wants to examine the reply, then appropriate buffers should be supplied as `replyhdr` and `reply`.

A.7 Disconnecting from the Server

To disconnect from the server, use the function `player_disconnect()`:

```
/*
 * close a connection. conn should be a value that was previously returned
 * by a call to player_connect()
 *
 * Returns:
 *    0 if everything is OK (connection closed)
 *   -1 if something went wrong (connection not closed)
 */
int player_disconnect(player_connection_t* conn);
```