

User Manual

for *ERSP Player Driver Source Package*

From the project: **Player/Stage - Player driver implementation for ERSP**

Bue Petersen <buep@diku.dk>
Jonas Fonseca <fonseca@diku.dk>

Department of Computer Science
University of Copenhagen
Winter 2006

Contents

Contents	2
List of Figures	2
List of Tables	2
Listings	2
1 User Manual	3
1.1 Getting Started	3
1.2 A Brief Overview of Player/Stage	4
1.3 The Scorpion Robots	5
1.4 Taking the Scorpion Robot for a Spin	6
1.5 Using the Sensors	7
1.6 Running on the Physical Robots	8
1.7 Supported Interfaces	8
1.8 Advanced Sensor Usage	8
1.9 The Player Toolbox	10
1.10 Known Limitations and Problems	10
1.11 Other Topics	12
1.11.1 Other Languages	12
1.11.2 Virtual Drivers	12
1.11.3 Further Reading	12

List of Figures

1 The Player Framework	4
2 Stage Scorpion IR range sensors	5

List of Tables

1 Supported Player interfaces	9
---	---

Listings

Drive example	6
IR range sensor example	7
Front bumper example	9

1 User Manual

This manual for using Player/Stage on the Scorpion robots will help you get started writing robot control programs. It assumes that Player version 2.0.3 or greater and Stage version 2.0.1 or greater have already been installed on the local system and that the source package, in the following referred to as the *source package*, has been checked out from the DIKU ImageLab SVN repository. ERSP 3.1 software (Evolution Robotics Software Platform) is required to be installed for using the physical Scorpion robots.

The manual will first give you a brief introduction to Player/Stage and the features of the Scorpion robots. Following are examples on using the various sensors and actuators supported by the Scorpion robots. For the first few examples Stage, the 2D simulator for Player, will be used. Later examples will move on to run programs on the physical Scorpion robots.

If you encounter any errors while going through this manual, please refer to Section 1.10, which has details on some of the most common errors.

1.1 Getting Started

To give you an idea of how the basic system works, here is a quick 3-step guide to getting started. It uses the `Makefile` in the root directory of the source package to simplify initialization of the Player server.

1. From the root folder of the checked out source package, build the drivers and test programs by running:

```
% make
```

2. Start the Player server. To make Player use the Stage simulator pass it the cave world configuration file:

```
% make stage/cave/cave.cfg
```

3. Start a test program. A good and simple example is the random walk test program that uses the IR range sensors to navigate. Start it by running:

```
% ./randomwalk
```

The robot in the window of the Stage simulator should now be exploring the cave world. To stop the test program press **Control-C** in the same console you started the program. The robot in the simulator should stop moving. You can resume control over the robot by simply restarting the test program.

Normally, there is no reason to restart the Player server unless you have changed some of its configuration files. If you want the robot to return to its starting position, simply drag it to the desired position in the Stage window. Stage has several settings worth exploring, see Section 1.9 for a brief overview.

The Player server can be shutdown by running the following command in the same terminal it was started:

```
% make stop
```

This will also shutdown the simulator window and any robot control programs connected to the Player server.

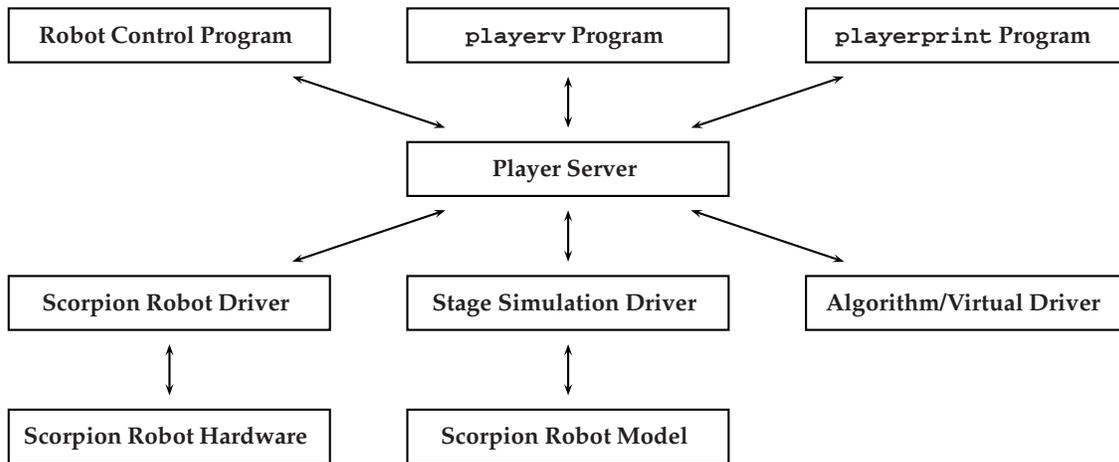


Figure 1: **The different components making up the Player Framework.** At the top of the figure are different types of Player clients, which besides robot control programs can be one of the many Player utility programs. They all connect to the Player server, which then takes care of providing access to the underlying hardware, simulator, or algorithm implementation. All access is done via drivers.

1.2 A Brief Overview of Player/Stage

Before continuing, you should know the basics of what components make up the Player Framework and how it works. As noted above, Player uses a client-server architecture, where the Player server takes care of virtualizing access to various robot devices by providing a set of generic interfaces. A wide range of different interfaces are defined, such as an interface for controlling motors. By using generic interfaces, the same robot control program can be used to control two widely different robots with little or no changes as long as the driver for the robots' devices provide the interfaces required by the program. More importantly, the use of generic interfaces makes it possible to simulate these devices. The configuration file given to the Player server program tells it which drivers to load. The Stage 2D simulator can be thought of as just a basic driver providing simulated devices. Figure 1 illustrates the relation between the components of the Player Framework.

In the Player system, robot control programs act as clients, connecting to the Player server on startup. It then subscribes to devices via the interfaces provided by the server, after which it enters the basic feedback loop of reading and processing sensor input followed by controlling actuators, like the motors. To summarize, the basic flow of a robot control is the following:

- Establish connection to Player server(s)
- Subscribe to device(s)
- Read sensory data from device(s)
- Processing
- Command robot (motors)

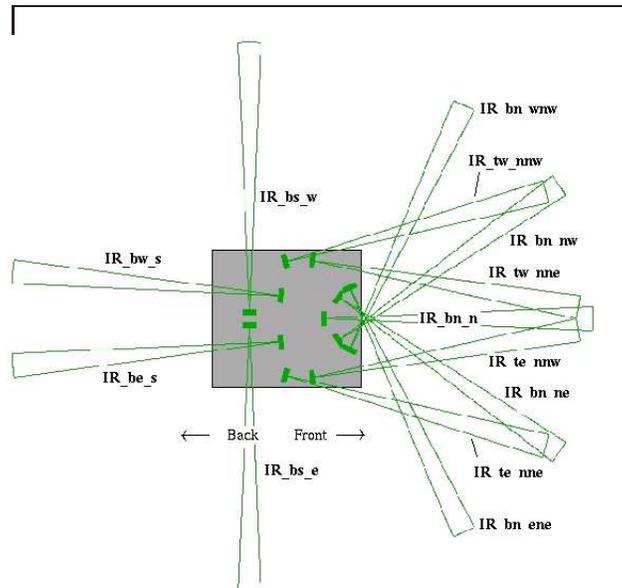


Figure 2: The IR range sensors in Stage. The Scorpion robots IR range sensors as modelled in Stage.

1.3 The Scorpion Robots

The robots that you will be programming are equipped with the following features.

2 contact bumpers are, as the name suggests, situated on the front of the robot to protect it from collisions. There are two sensors, one on each side.

7 vertical bumpers are short range digital IR sensors. 4 are used for ledge detection to avoid hazardous falls or to detect if the robot has been lifted up from the floor and the remaining 3 up-facing sensor for overhang detection to avoid camera damage. As the sensors are digital they only sense if obstacles are within range or not. They all trigger on distances from 1 cm - 24 cm.

13 analog IR range sensors are placed all around the robot with horizontal facing, as depicted in Figure 2. They have a narrow beam and their range is 10 cm - 80 cm. They are positioned on the robot so the first 10 cm of the range are "on" the robot. That is if they detect obstacle within 15 cm the obstacle is 5 cm from the robot.

2 servo motors are available for controlling the front wheels. The rear-end wheel is simply for support.

The robots also have a camera mounted on top, which can be used either via Player or the OpenCV library. How to do this is beyond the scope of this manual. Additionally in the simulator, odometry data from the front wheels can be used to track the movement of the robot.

1.4 Taking the Scorpion Robot for a Spin

To give you an idea of how to write a robot control program the first simple program will make the robot drive forward, stop it, and then turn it left and finally right. You can find the program in the `test/move-turn.cc` file in the source package.

First, include the header file of the Player C++ client library.

```
1 #include <libplayerc++/playerc++.h>
2 using namespace PlayerCc;
```

Next, establish a connection to a Player server running on the local machine by constructing a client object and use this object to subscribe to the position device used for controlling the motors on the Scorpion robot.

```
3 PlayerClient robot("localhost");
4 Position2dProxy position(&robot);
```

Like any other C++ program, the starting point is the `main` function.

```
5 int
6 main(int argc, char *argv[])
7 {
```

Since there is no processing of sensor data taking place in this example, it moves on to commanding the robot. The first command will move the robot forward with a speed of 20 centimeters per second. Player by default uses meters as the basic length unit, thus it is specified as 0.20. The robot will continue executing this order until another command is issued. To give the robot time to move forward the program will sleep for 3 seconds.

```
8     position.SetSpeed(0.20, 0.0);
9     sleep(3);
```

To turn the robot pass a non-zero value as the second argument to the `SetSpeed` method. For 3 seconds turn the robot rightward with an angle of -0.50 radian per second.

```
10     position.SetSpeed(0.0, -0.5);
11     sleep(3);
```

Alternatively, you can use the `DTOR` macro to convert angles in degrees to radians. For another 3 seconds, turn the robot leftward with an angle of 30 degrees per second.

```
12     position.SetSpeed(0.0, DTOR(30));
13     sleep(3);
```

When done, stop the motors so the robot is in a safe state when the program terminates.

```
14     position.SetSpeed(0.0, 0.0);
15
16     return 0;
17 }
```

Compile the robot control program by using the following command:

```
% g++ -o move-turn $(pkg-config --cflags --libs playerc++) test/move-turn.cc
```

Before running the `move-turn` program, remember to restart the Player server with the cave world configuration file if you stopped it in the previous section.

1.5 Using the Sensors

The Scorpion robots are equipped with three different types of sensors. Two of them are “bumper” sensors dedicated to hazard avoidance and the last provides information about ranges to near-by obstacles. In the following example, the range sensors will be used to avoid obstacles in front of the robot by turning the robot before moving forward again.

The program preamble is similar to the previous example, however, this time the range sensors will also be used. As a result, the `scorpion.h` header file is included and an IR device declared.

```

1 #include <libplayerc++/playerc++.h>
2 #include <scorpion.h>
3 using namespace PlayerCc;
4
5 PlayerClient robot("localhost");
6 Position2dProxy position(&robot);
7 IrProxy ir(&robot);

```

Sensor devices can be thought of as providing a continuous source of sensor readings, which are buffered in the robot control program. Consequently, it is necessary for the Player client to continuously synchronize and process these readings so it can update the robot device objects. This is done by periodically calling the `Read` method. A good way to achieve this is to call it in the start of the program’s main loop.

```

8 int
9 main(int argc, char *argv[])
10 {
11     while (true) {
12         robot.Read();

```

To see if there are any obstacles in front, three of the IR front sensors are checked. If either of them are reporting ranges less than 50 centimeters, turn the robot.

```

13         if (ir[SCORPION_IR_TW_NNW] < 0.50 ||
14             ir[SCORPION_IR_BN_N] < 0.50 ||
15             ir[SCORPION_IR_TE_NNE] < 0.50) {
16             position.SetSpeed(0.0, -0.5);
17             continue;
18         }

```

As can be seen, the IR device provides access to its readings as if it was an array. The somewhat cryptic names of the array indices are defined in the `scorpion.h` header file and can optionally be used to refer to the individual sensors. The names are derived from the sensor names used in the resource configuration file for the Scorpion robots¹.

Finally, when no obstacles are found, move the robot forward.

```

19             position.SetSpeed(0.10, 0.0);
20         }
21
22     return 0;
23 }

```

You can find the above example in the `test/ir-avoid.cc` file in the source package. Compile it using:

¹See the `resource-config.xml`-file, which is typically found in the local installed Evolution Robotics documentation, eg. `/opt/evolution-robotics/config/` on DIKUs robot laptop computers

```
% g++ -o ir-avoid $(pkg-config --cflags --libs playerc++)  
-Idriver/include test/ir-avoid.cc
```

1.6 Running on the Physical Robots

Until now you have only run the example programs on the Scorpion robot modeled in the Stage simulator. Before continuing introducing more examples, you should try to run the previous examples on the physical robots.

First, shutdown any currently running Player servers and simulators.

```
% make stop
```

Then, bring up a new Player server, this time using the configuration file for the physical Scorpion robot.

```
% make player/scorpion.cfg
```

Running the robot control programs on the real robots is the same procedure as with the simulator. There is no need to recompile any of them.

1.7 Supported Interfaces

One of the benefits of using the Player system is to be able to quickly prototype and test an initial version of a robot control program in the simulator and then seamlessly move the program to run on the physical robot with little or no modifications. Since Player/Stage is a work in progress this is not entirely the case.

Consequently, before starting to write your own program you should first find out what Player interfaces are provided by the robot for which you want to program—be it a simulated or real-world physical robot—and which interfaces you want to use in your program. Stage, as of version 2.0.1, lacks support for simulating certain Player interfaces, it might therefore be necessary to take this into account by conditionally using interfaces only available on the physical robot.

A summary of the supported interfaces is available in Table 1. As can be seen, there are several differences between the interfaces provided by Stage and those provided by the ERSP Player driver. The basic difference is that all bumper sensors dedicated to hazard avoidance are missing in Stage.

For more advanced robot control programs the incompatibilities can become a problem hindering easy switching between running in Stage and on the physical robots. One way to work around this problem is to probe interfaces inside a `try/catch` construct and in the rest of the program only conditionally access the proxy objects in question. See some of the programs in the `test` directory of the source package for examples on how to accomplish this.

1.8 Advanced Sensor Usage

As mentioned above, sensors readings must periodically be processed. This can be a problem if you are going to use sleep or do a lot of heavy computations in the robot control program, since the buffer will quickly fill up. This results in new readings being dropped and old stale sensor values being reported by the devices. To avoid this problem, you should set up the client to “filter message” and only use the most recent sensor values.

Interface	Feature set	Example usage	Driver	Stage
Drive (position2d)	Motor commands	Position2DProxy pp(robot); pp.SetSpeed(speed, angle); pp.SetCarlike(speed, angle);	yes	yes
	Odometry	pp.Goto(x, y, yax); pp.GetXPos(); pp.GetYPos(); pp.GetYaw(); pp.SetOdometry(); pp.ResetOdometry();	no	yes
	Power (enable/disable)		no	no
IR range (ir)	Range data	IrProxy ir(robot); ir.GetCount(); ir.GetRange(index); ir[index];	yes	yes
	Pose data	ir.RequestGeom(); ir.GetPoseCount(); ir.GetPose();	yes	yes
IR bumper (bumper)	Bump data	BumperProxy irbump(robot); irbump.IsBumped(index); irbump[index]; irbump.IsAnyBumped();	yes	no
Front bumper (bumper)	Bump data	BumperProxy front(robot); front.IsBumped(index); front[index]; front.IsAnyBumped();	yes	no

Table 1: The interfaces supported by the ERSP Player driver and Stage. For each interface, relevant feature sets are listed along with examples on how to use them. The two rightwards columns tells if the feature set is supported by the driver and Stage.

Below is an example of accomplishing this. It checks the front bumpers once every second to see if something is blocking the path of the robot. If the path is clear the robot is moved forward, else it is stopped.

```

1 #include <libplayerc++/playerc++.h>
2 #include <scorpion.h>
3 using namespace PlayerCc;
4
5 int
6 main(int argc, char *argv[])
7 {
8     PlayerClient robot("localhost");
9     Position2dProxy position(&robot);
10    BumperProxy bumper(&robot);
11
12    robot.SetDataMode(PPLAYER_DATAMODE_PULL);
13    robot.SetReplaceRule(-1, -1, PPLAYER_MSGTYPE_DATA, -1, 1);
14
15    while (true) {
16        robot.Read();
17
18        if (bumper.IsAnyBumped()) {
19            position.SetSpeed(0.0, 0);
20            sleep(1);
21        } else {
22            position.SetSpeed(0.10, 0);

```

```

23     }
24 }
25
26     return 0;
27 }

```

The calls to the robot object's `SetDataMode` and `SetReplaceRule` methods configures the client so that similar data messages are replaced with the newest version.

Compile it using:

```
% g++ -o bump-stop $(pkg-config --cflags --libs playerc++)
-Idriver/include test/bump-stop.cc
```

Note, this example can only be run on the physical Scorpion robots as the bumper device is available in the simulator.

This concludes the program examples in this manual. More examples are available in the driver source package's `test` directory. Here you will find the output program, which dumps data from all sensors, including the vertical bumper sensors, for which no example has been given.

1.9 The Player Toolbox

Player/Stage comes with a rich set of tools, some of which are worth getting acquainted with.

playerv also known as *the Player Viewer*, is a tool for examining (and in some circumstances controlling) devices provided by a running Player server. The basic idea is that you can subscribe to the devices, in which you are interested. For example, by subscribing to the `position2d:0` device in the **Devices** menu you will be able to see the current velocity and pose of the robot.

playerprint is a simple alternative to `playerv` which dumps sensor data to the console.

stage has several helpful features for debugging your program. For example, you can visualize the IR range sensors, by enabling the `ranger` config entry in the **View** menu. The same menu also contains **Show trails**, which over time will color the path of the robot. The **File** menu in the Stage GUI also allows you to grab a screenshot or capture a movie.

1.10 Known Limitations and Problems

Some of the most common problems you might run into are listed below. Among them are some Player error messages that might need clarification.

- To localize the install path of Player header files and libraries the `pkg-config` program is used. If it fails to find the installed Player C++ library package configuration file, called `playerc++.pc`, it report that:

```
Package playerc++ was not found in the pkg-config search path.
Perhaps you should add the directory containing 'playerc++.pc'
to the PKG_CONFIG_PATH environment variable
No package 'playerc++' found
```

To solve the problem, add an entry to your the configuration file of your shell. For example, if you are using Bash and installed Player into `/usr/local/`, add this line to `~/.bashrc`:

```
% export PKG_CONFIG_PATH="/usr/local/lib/pkgconfig:$PKG_CONFIG_PATH"
```

Before continuing, remember to source it or open a new terminal to make it take effect. Use the `env` command to check if it is set.

- It can be hard to determine when the Player server has completed its initialization and are ready to be used. The server will first look at the `PLAYERPATH` environment variable to look for plugin drivers. It will then try to load any plugins. In the example below, the ERSP Player driver is loaded and the driver prints that it has registered itself. Finally, the server prints the port it is listening on, in this case uses the default port. After the port number has been printed the server is ready.

```
PLAYERPATH: /home/diku/PlayerStage/driver/lib :
trying to load /home/diku/PlayerStage/driver/lib/libersp ...
success
invoking player_driver_init() ...
Registering ERSP driver.
success
Listening on ports: 6665
```

- The following error is reported when a robot control program is started and no Player server is running.

```
playerc error : connect call on [localhost:6665] failed with error [
Connection refused]
```

See Section 1.1 or 1.6 for information on starting the Player server with or without the Stage simulator backend.

- A failure to subscribe to a device will generate the following error messages.

```
warning : skipping subscription to unknown device 14:0
playerc error : got NACK from request
playerc error : failed to get response
Shutting stage driver down
stage driver has been shutdown
BumperProxy::BumperProxy()(-1) : could not subscribe
closing connection to client 0 on port 6665
```

See the end of Section 1.7 for an example of how to avoid this error by conditionally subscribing to devices.

- If for any reason the USB cable connecting the laptop and the robot is unplugged, the ERSP library will start printing a lot of warnings in the terminal window where the Player server was started.

```
WARN - Failed to read batteries.
WARN - Unable to communicate with resource driver 'Battery'. Please check
hardware connection.
WARN - returning comm write error
WARN - RCM4 command 40 failed with result 32724.
```

```
WARN - Unable to communicate with resource driver 'IR_bn_ene'. Please
      check hardware connection.
WARN - returning comm write error
WARN - RCM4 command f9 failed with result 32724.
WARN - Unable to communicate with resource driver 'IR_tn_edown'. Please
      check hardware connection.
WARN - returning comm write error
WARN - RCM4 command c0 failed with result 32724.
```

To fix the issue you need to shutdown the Player server. Use the following command from any terminal:

```
% killall player
```

You can then reconnect the laptop and robot, and restart the Player server to continue.

1.11 Other Topics

The manual has only touched briefly the world of Player/Stage. This section will try to give you a taste of other interesting topics and where to find more information.

1.11.1 Other Languages

All examples in this manual so far have used C++ for programming the robots. However, Player/Stage supports a variety of other languages for writing robot control programs. Besides the C++ and C client libraries bundled with Player, are clients for Python, Scheme, LISP, and Java to name a few. See the Player Wiki page² for a list of supported languages and how to get started using them.

Note that some languages do not yet provide a complete implementation of the Player/Stage 2.0 API, so before considering using a language other than C++ or C be sure to check the project page of the client library you want to use.

1.11.2 Virtual Drivers

One of the interesting features of the Player/Stage system is the notion of virtual drivers. An example is the `amcl` driver implementing the Adaptive Monte-Carlo Localization algorithm. It uses odometry data from a `position2d` interface, a `laser` interface for scanning surroundings, and a `map` interface holding a map in which to localize the robot. In return, it provides a `localize` and a `position2d` interface.

Virtual drivers usually take input from a collection of existing “raw” Player interfaces, process this input, and provide a higher-level interface. If you are implementing a generic algorithm that you would like to use in several projects or make available for others to use, consider making it a virtual driver.

1.11.3 Further Reading

If you are interested in reading more about how to use Player/Stage here are some relevant online resources:

²<http://playerstage.sourceforge.net/wiki/PlayerClientLibraries>

- The Player/Stage website: <http://playerstage.sourceforge.net/>
- The Player wiki: <http://playerstage.sourceforge.net/wiki>
- The Player Manual: <http://playerstage.sourceforge.net/doc/Player-2.0.0/player/>
- The C++ Player client library: http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__player__clientlib__cplusplus.html
- Manuals for Player utilities: http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__utils.html