Writing Player/Stage Drivers

a howto 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

Listings

1 Writing Player/Stage Drivers

In this tutorial, we will go through the process of writing a simple driver for Player. The goal is to help you quickly get started on your own driver by introducing you to the key concepts. The tutorial will first raise some of the considerations you should make before starting your driver development. This is followed by a walk-through of how a driver interacts with the Player server and which requirements it poses on your driver. Finally, it provides some tips and tricks that might be useful when writing drivers.

The Player driver that will be used as an example is a driver for interacting with Hard Drive Active Protection System (HDAPS) available on various IBM ThinkPads models. HDAPS uses a gyroscope to detect sudden movements; an operating system can use these measurements to avoid the harddrive being damaged. The driver will use information from the Linux HDAPS driver to provide a `joystick` interface, which can be used to control a robot by moving the laptop.

The source for the HDAPS driver is available in the following SVN repository: `http://image.diku.dk/svn/robot/trunk/erspplayerdriver/driver/hdaps`.

1.1 Initial Considerations

Before starting to write a driver you need to decide how the driver will work with Player. There are two type of drivers:

Static drivers are statically linked into the Player server. These drivers are usually distributed together with Player and typically only used by the core Player developers.

Plugin drivers are shared objects that are dynamically loaded into the Player server at runtime. As we will see this makes them more flexible compared to static drivers.

Although the choice of *static* versus *plugin* does not have much affect on the resulting driver code and driver usage, it does affect the development cycle. The more flexible nature of plugin drivers has several advantages, where the most noteworthy are a much faster code/compile/test cycle and the ability to maintain them out-of-tree.

For the purpose of this tutorial, plugin drivers are ideal since they are simpler, because no knowledge of the Player server internal is required. Thus, this tutorial will show how to create the HDAPS driver as a Player server plugin.

1.2 Feature Set and Configuration File

After decided what driver type to use, it is time to consider which features the driver should support. In the world of Player, this translates to what interfaces the driver should implement. There already exists a rich set of interfaces that cover many of the most common robotics hardware as well as other more high-level interfaces for more advanced drivers.

It is possible to create new interfaces, should that be necessary. However, if your needs can be fulfilled by an existing interface this is preferable. In the long run, it will be less work for you and you will likely be able to use some of the many existing Player utilities for debugging your driver.

As already mentioned, the driver in this tutorial will provide a `joystick` interface. Player already has a `joystick` interface, which means that the HDAPS driver will not have to worry about defining new interfaces.

When the feature set of the driver has been decided you should write a configuration file for the driver. Later, this configuration file can be used to start up the Player server, when you will have to test the driver. Below the configuration file for the HDAPS driver is shown.

```
driver
(
        name "hdaps"
        plugin "libhdaps"
        provides ["joystick:0"]
)
```

Briefly explained, it declares a new driver with the name `hdaps`. The driver will be a plugin, which should be loaded from a shared object file starting with the name `libhdaps`. Finally, the interfaces provided by the driver are listed. The HDAPS driver will only supports one `joystick` interface, however, in case of many supported interfaces the appended index notation (`:0`) allows multiple instances of the same interface type to be provided.

1.3 Creating a Driver Class

The first step when starting to implement a driver is to create a new class for the driver. This is usually done in the header file of the driver. All Player drivers must inherit from the `Driver` class, which acts as the layer between the Player server and the driver by defining a standard API. The base class, however, also defines several mandatory methods that your driver should implement.

Before defining the driver class in the header file, it is necessary to first include the `libplayercore/playercore.h` file, which contains driver specific types and utilities. An excerpt of the HDAPS header file is shown below. It defines a class for the HDAPS driver, which, besides the mandatory (*virtual*) public methods, defines some private members for managing its state as well as a private utility method.

```
#include <libplayercore/playercore.h>

class HDAPS : public Driver
{
  private:
        // Joystick state data
        player_devaddr_t joystick_id;
        player_joystick_data_t data, prev_data;

        // Publish state data.
        void PutData(void);

  public:
        HDAPS(ConfigFile *cf, int section);
        ~HDAPS(void);

        // Thread life-cycle
        virtual void Main();
        virtual int Setup();
        virtual int Shutdown();

```

```
22          // Message handling
23          virtual int Subscribe(player_devaddr_t id);
24          virtual int Unsubscribe(player_devaddr_t id);
25          virtual int ProcessMessage(MessageQueue *queue, player_msghdr *msghdr,
                void *data);
26  };
```

As can be seen, the driver defines a member called `joystick_id`. This member holds the device address for the joystick device that the Player server makes available. The driver can use it internally to match incoming messages and manage client subscriptions.

The driver exclusively uses Player defined types to store its state. The main reason is that it simplifies a lot of the driver code, since the interface-specific data types are also used when publishing data. In our example, the HDAPS driver will be able to publish its joystick data by simply handing a reference to its `data` member to a function that will take care of sending it to all subscribed clients.

1.4 Driver Methods

Before explaining the purpose of the mandatory methods, we will first give a short overview of the basic driver methods. The driver methods can be grouped into 4 basic categories:

- Driver setup and shutdown.
- Driver main loop.
- Subscriptions and publishing.
- Message processing.

Besides the above categories, there is a server-specific hook for registering the driver. Plugin drivers must also define some plugin-specific hooks.

Additionally, the driver may declare its own driver specific methods. For example, the HDAPS driver defines a method that is used to poll joystick data from the HDAPS device in the laptop. This method is used in the main loop when updating data. How this method works is not important, so it is left out of this tutorial. If you are interest in how it works, check the driver source code.

The following sections will go into more depth with the basic categories and the various methods belonging to each category.

1.5 Driver Setup and Shutdown

These methods takes care of the overall life-cycle of the driver. They are divided into two parts: setup and shutdown taking place when the Player server starts and shutdowns and the setup and shutdown done when the driver get subscribers. The first part is handled by the constructor and destructor of the driver class, while the other is done by the `Setup` and `Shutdown` methods.

1.5.1 Driver Class Constructor

Called when the server starts up, the driver constructor is responsible for initializing private members and register the devices it provides with the server. For the HDAPS driver, this involves clearing the joystick device address followed by the registration of the joystick device. Any failure to do this will cause an error to be flagged by a call to `SetError`.

```
HDAPS::HDAPS(ConfigFile* cf, int section)
    : Driver(cf, section, true, PLAYER_MSGQUEUE_DEFAULT_MAXLEN)
{
        // zero ids, so that we'll know later which interfaces were requested
        memset(&joystick_id, 0, sizeof(joystick_id));

        if (cf->ReadDeviceAddr(&joystick_id, section, "provides",
                               PLAYER_JOYSTICK_CODE, -1, NULL) == 0) {
                if (AddInterface(joystick_id) != 0) {
                        SetError(-1);
                        return;
                }
        }
}
```

1.5.2 `Setup`

This method is called every time a the driver goes from having no subscriber to receiving the first subscription. It allows the driver to perform device-specific initialization, such as opening a serial port for communication. The method is also responsible for starting the driver thread. On success it should return zero, while any errors should be reported by returning `-1`.

The HDAPS driver does not strictly require any device initialization. However, to ensure a working joystick device it performs a check of whether the HDAPS position can be acquired. If the check fails an error is reported.

```
int HDAPS::Setup()
{
        int xpos, ypos;

        if (hdaps_position(&xpos, &ypos)) {
                PLAYER_ERROR("Failed to read HDAPS position data\n"
                             "Maybe you need to run: modprobe hdaps");
                return -1;
        }

        StartThread();
        return 0;
}
```

1.5.3 `Shutdown`

Being the counter-part to `Setup`, this method is called when the last client unsubscribes from a device provided by the driver. The driver can use this to close down and release device-specific resources. This method must also take care of stopping the driver thread.

Since the HDAPS driver, as already mentioned, does not maintain any device-specific resources, it will simply stop the driver thread when this method is called.

```
int HDAPS::Shutdown()
{
        StopThread();
        return 0;
}
```

1.5.4 Driver Class Destructor

When the Player server is shutdown, the driver's destructor will be called. It can then take care of releasing any long-lived resources. The HDAPS driver has a minimum of state, so does not do anything in its destructor.

```
HDAPS::~HDAPS (void)
{
}
```

1.6 Driver Main Loop

The main loop is where the driver spends most of its time continuously updating and publishing data as well as handling incoming requests. The loop runs as long as there are clients subscribed to the devices of the driver.

1.6.1 `Main`

This method is called when the driver thread is started by `Setup` and primarily consists of a loop. The loop must first call `pthread_testcancel` to check if execution should be cancelled. This is the case when `Shutdown` has been called and stopped the driver thread. Otherwise, the loop usually has 3 steps: first sensor and state data are collected, then collected data is published to the relevant devices, and finally, incoming messages are processed.

The HDAPS uses the above form for its main loop, however, before publishing its joystick data it checks if it has changed and only conditionally sends it. Furthermore, the driver shows how `Lock` and `Unlock` can be used to ensure exclusive access to certain driver class members.

```
void
HDAPS::Main()
{
        int xpos, ypos;

        for(;;) {
                pthread_testcancel();

                memset(&data, 0, sizeof(data));
                Lock();
                if (hdaps_position(&xpos, &ypos) == 0) {
                        data.xpos = (uint32_t) xpos;
                        data.ypos = (uint32_t) ypos;
                }
                Unlock();

```

```
17                  if (memcmp(&prev_data, &data, sizeof(data))) {
18                          PutData();
19                          memcpy(&prev_data, &data, sizeof(data));
20                  }
21
22                  if (InQueue->Empty() == false) {
23                          ProcessMessages();
24                  }
25          }
26  }
```

1.7 Subscriptions and Publishing

Concurrently with the main loop, the driver will receive requests from clients to subscribe and unsubscribe to devices. Each request will generate a call to either `Subscribe` or `Unsubscribe`. The driver can use these to maintain a count of how many clients are currently subscribed to a device. This information can be used in `PubData` to optimize publishing of data.

1.7.1 `Subscribe`

When a client subscribes to a device maintained by a driver, the server calls this method. If the driver provides multiple devices, it may use the passed device address to find which device the subscription is for. The method should return zero on success and `-1` otherwise. The HDAPS driver only has one device so it simply passes the subscription on to the driver super class.

```
1  int
2  HDAPS::Subscribe(player_devaddr_t id)
3  {
4          return Driver::Subscribe(id);
5  }
```

1.7.2 `Unsubscribe`

After a client has told the server that it no-longer is interested in a device, this method is called. As with the `Subscribe` method, the passed device address allows the driver to match among its supported devices. The method should return zero on success and `-1` otherwise. The HDAPS driver also passes unsubscriptions on to the driver super class.

```
1  int
2  HDAPS::Unsubscribe(player_devaddr_t id)
3  {
4          return Driver::Unsubscribe(id);
5  }
```

1.7.3 `PutData`

Periodically, the driver needs to publish data to subscribed clients. It is done via the main loop calling this method. Using the `Publish` method, all the low-level handling of messages is taken care of. The driver only needs to inform it of the device address of the message (`joystick_id`), the message type and subtype (`PLAYER_MSGTYPE_DATA` and `PLAYER_JOYSTICK_STATE`), and finally the data to send (`data` and `sizeof(data)`).

```
void
HDAPS::PutData(void)
{
        Publish(joystick_id, NULL, PLAYER_MSGTYPE_DATA,
                PLAYER_JOYSTICK_DATA_STATE,
                (void*)&data, sizeof(data));
}
```

1.8 Message Processing

The main loop will periodically check if new messages has been queued. Whenever, the queue is not empty, messages on the queue must be processed. This is mainly done by the `ProcessMessage` method.

1.8.1 `ProcessMessage`

This method can use `Message::MatchMessage` to match information found in the message header passed in `hdr`. The driver can use this to filter message and conditionally handle them. If a message is supported, the driver should return zero after it has handled it. Unknown messages should cause it to return `-1`.

Since the HDAPS only supports the `joystick` interface, which does not define any messages that clients can send to the driver, this method simply returns failure.

```
int
HDAPS::ProcessMessage(MessageQueue *queue, player_msghdr *hdr, void *data)
{
        PLAYER_WARN("unknown message");
        return -1;
}
```

1.9 Server and Plugin Specific Hooks

The Player server maintains a registry of all supported drivers. When a driver is loaded as a plugin it must add itself to this registry and provide a factory function. The server can use this to generically create drivers without caring about the specific driver class.

1.9.1 Driver Class Factory: `HDAPS_Init`

When adding a driver to the Player server's registry a driver class factory function must be provided. The function takes arguments that the drivers constructor can use for looking up driver specific settings from the Player server configuration file. The factory function should return a newly created driver in the form of the driver base class.

The HDAPS driver simply passes the arguments on to its constructor and casts the result of calling the constructor to a `Driver` pointer.

```
Driver*
HDAPS_Init(ConfigFile* cf, int section)
{
        return (Driver*)(new HDAPS(cf, section));
}
```

1.9.2 Driver Register Hook: `HDAPS_Register`

The registry of the server is dynamically maintained, which means that all drivers should register themselves on start up. It is accomplished via this method, which in addition to the above driver class factory function also takes the name of the driver as specified in the server configuration file. The HDAPS driver registers its factory function and declares its name to be `hdaps`, which was the name used in the driver configuration file above.

```
1 void HDAPS_Register(DriverTable* table)
2 {
3         table->AddDriver("hdaps", HDAPS_Init);
4 }
```

1.9.3 Driver Load Hook: `player_driver_init`

Upon loading a plugin driver, the Player server will call this method. Its main purpose is to allow the driver to hook into the servers driver table. The function must be declared inside an `extern "c"` section to avoid C++ name-mangling by the compiler. The HDAPS will simply print an informational message and register itself.

```
1 extern "C" {
2         int player_driver_init(DriverTable* table)
3         {
4                 PLAYER_MSG0(1, "Registering HDAPS driver.");
5                 HDAPS_Register(table);
6                 return 0;
7         }
8
9 }
```

1.10 Tips and Tricks

This section gives some tips and tricks that can be useful to consider when writing a driver. The first tips is to use the Player defined print macros for providing diagnostic output to the console. The print macros allows messages to be tagged with severity, such as error, warning, and informational.

When writing a driver that has support for sensors, it is a good idea to first make the driver handle messages related to sensor poses and the robot geometry. This will allow you to use the Player utilities to debug your driver. For example, the Player Viewer program (`playerv`) can visualize the sensor readings that your driver reports.

Drivers that are more advanced than the driver in this tutorial can experience problems related to concurrency, such as race conditions. The issue arises because class members may be accessed concurrently by the main loop and the methods for subscribing and unsubscribing. The solution is to guard all access to such members by using `Lock` and `Unlock` to ensure that access to the protected members is exclusive.