FluentRealSense – The First Steps to a Simpler RealSense

Pete O'Hanlon, 9 Mar 2018

★★★★★ 5.00 (1 vote) Rate this:

Those who know me are aware that I have a long term association (nay, let&#8217;s say it for what it is, love affair) with the RealSense platforms from Intel.

Those who know me are aware that I have a long term association (nay, let's say it for what it is, love affair) with the RealSense platforms from Intel. If you've been following developments in that space for any length of time, you're probably aware that things have moved on a lot and the cameras are available in all sorts of devices and are no longer just limited to the Windows platform. This shift of emphasis means that Intel have moved away from concentrating on the old monolithic RealSense SDK and have moved to supporting a new open source API (available here: https://github.com/IntelRealSense/librealsense).

I have spent a period of time working with this API as it has evolved and have started "wrapping" it to make it a little bit friendlier to use (in my opinion) for people who don't want to get bogged down with the "nitty gritty" of how to get devices, and so on. This work has been in C++, the language the API is available in, and I am going to cover, over a series of posts, how this library evolved. I'll show you the code I've written at each step so that you can also see how I have gone about relearning modern C++ (bearing in mind it has been 18 years since I last worked with it in anger) so you will definitely find that early iterations of the code are naive, at best. My aim with this library was to make it as fluent as possible, hence it's name, FluentRealSense.

An upfront note. Early iterations of this codebase have been done entirely on Windows running Visual Studio 2017. I chose this particular environment because, a) Visual Studio is, to my mind, the finest IDE around and b) because 2017 supports porting your C++ code directly over to Linux. This lets me leverage the rapid turnaround time I'm used to in my dev environment and the ease of deploying to my intended targets later on. Thank you Microsoft for keeping this old developer happy.

Let's start with the first iteration. The first thing I wanted the code to do was to provide me with the ability to iterate over all the cameras on my system and print out some diagnostic information. This meant that my code would be broken down initially into three classes:

- `information`: This class reads information from the API about a single camera and returns it as a `string`.
- `camera`: This is a single device. Initially, it's just going to instantiate and provide access to the `information` class.
- `cameras`: The entry point for consuming applications, this class is responsible for finding out about and managing access to each `camera`; this class is also enumerable so that the calling code can quickly iterate over each `camera`.

Let's start off by looking at the `information` class. First, we're going to add some `include`s and `namespace`s:

Hide Copy Code

```
#pragma once
#include "hpp/rs_context.hpp"

using namespace std;
using namespace rs2;

class information
{
public:
class information() {}
~information() {}
}
```

That's all very straightforward and nothing you won't have seen before. Let's start fleshing this out.

The first thing we're going to need is a RealSense device to get information from – this is going to be passed in so let's add a member to store it and replace our constructor with this:

Hide Copy Code

```
explicit information(const device device) : device_(device) {}
:::
private:
device device_;
```

At this stage, our code looks like this:

Hide Copy Code

```
#pragma once
#include "hpp/rs_context.hpp"

using namespace std;
using namespace rs2;

class information
{
public:
explicit information(const device device) : device_(device) {}
~information() {}

private:
device device_;
}
```

I have to say, at this point, I love the improvements to instantiating members that C++ now provides. This is a wonderful little innovation.

Now, in order to get the information out of the API, I'm going to add a handy little `private` helper method. This makes use of the fact that the API exposes this information via an `rs2_camera_info` enumeration:

Hide Copy Code

```
const char* get_info(const rs2_camera_info info) const
{
if (!device_.supports(info))
{
return "Not supported";
}
return device_.get_info(info);
}
```

This is the first point that our code is going to add any value. Whenever we call `get_info` against the underlying device, we have to check that that particular type of info can be retrieved for that device. It's very easy to write code that makes assumptions that each camera supports exactly the same set of properties here so the underlying call could throw an exception. To get around this, our code checks to see if the device supports that particular `rs2_camera_info` type and returns "`Not supported`" if it doesn't. It is so easy to forget that you should always pair the `get_info` with the supports, so this helper method removes the need to remember that.

So, how do we use this? Well, with some `public` methods like these:

Hide Shrink ▲ Copy Code

```
const char* name() const
{
return get_info(RS2_CAMERA_INFO_NAME);
}

const char* serial_number() const
{
return get_info(RS2_CAMERA_INFO_SERIAL_NUMBER);
}

const char* port() const
{
return get_info(RS2_CAMERA_INFO_PHYSICAL_PORT);
}

const char* firmware_version() const
{
return get_info(RS2_CAMERA_INFO_FIRMWARE_VERSION);
}

const char* debug_opCode() const
{
return get_info(RS2_CAMERA_INFO_DEBUG_OP_CODE);
}

const char* advanced_mode() const
{
return get_info(RS2_CAMERA_INFO_ADVANCED_MODE);
}

const char* product_id() const
{
return get_info(RS2_CAMERA_INFO_PRODUCT_ID);
}

const char* camera_locked() const
{
return get_info(RS2_CAMERA_INFO_CAMERA_LOCKED);
}

string dump_diagnostic() const
{
string text = "\nDevice Name: ";
text += name();
text += "\n Serial number: ";
text += serial_number();
text += "\n Port: ";
text += port();
text += "\n Firmware version: ";
text += firmware_version();
text += "\n Debug op code: ";
text += debug_opCode();
text += "\n Advanced Mode: ";
text += advanced_mode();
text += "\n Product id: ";
text += product_id();
text += "\n Camera locked: ";
text += camera_locked();

return text;
}
```

We now have our complete information class. It looks like this:

Hide Shrink ▲ Copy Code

```
#pragma once
#include "hpp/rs_context.hpp"
```

```
using namespace std;
using namespace rs2;

class information
{
public:
explicit information(const device device) : device_(device) {}
~information()
{
}

const char* name() const
{
  return get_info(RS2_CAMERA_INFO_NAME);
}

const char* serial_number() const
{
  return get_info(RS2_CAMERA_INFO_SERIAL_NUMBER);
}

const char* port() const
{
  return get_info(RS2_CAMERA_INFO_PHYSICAL_PORT);
}

const char* firmware_version() const
{
  return get_info(RS2_CAMERA_INFO_FIRMWARE_VERSION);
}

const char* debug_opCode() const
{
  return get_info(RS2_CAMERA_INFO_DEBUG_OP_CODE);
}

const char* advanced_mode() const
{
  return get_info(RS2_CAMERA_INFO_ADVANCED_MODE);
}

const char* product_id() const
{
  return get_info(RS2_CAMERA_INFO_PRODUCT_ID);
}

const char* camera_locked() const
{
  return get_info(RS2_CAMERA_INFO_CAMERA_LOCKED);
}

string dump_diagnostic() const
{
  string text = "\nDevice Name: ";
  text += name();
  text += "\n Serial number: ";
  text += serial_number();
  text += "\n Port: ";
  text += port();
  text += "\n Firmware version: ";
  text += firmware_version();
  text += "\n Debug op code: ";
  text += debug_opCode();
  text += "\n Advanced Mode: ";
  text += advanced_mode();
  text += "\n Product id: ";
  text += product_id();
  text += "\n Camera locked: ";
  text += camera_locked();
  return text;
}

private:
  const char* get_info(const rs2_camera_info info) const
  {
    if (!device_.supports(info))
    {
      return "Not supported";
    }
    return device_.get_info(info);
  }
  device device_;
};
```

The next thing we have to do is write a class that represents a single `RealSense` camera. There's not much to this class, at the moment, so let's look at it in its entirety.

Hide Copy Code

```
#pragma once
#include "hpp/rs_context.hpp"
#include "information.h"

using namespace std;
using namespace rs2;

class camera
{
public:
  explicit camera(const device dev) : information_(make_shared(dev)) {}

  ~camera()
  {
  }

  shared_ptr get_information() const
  {
    return information_;
  }

private:
  shared_ptr information_;
};
```

I did say this class was pretty light at the moment. The class accepts a single RealSense device which is used to instantiate the `information` class. We provide one method which is used to get the instance of the `information` class. That's it so far.

Finally, we come to the entry point of our code, the `cameras` class. This class lets us enumerate all of the `cameras` on our system and access the functions inside. As usual, we'll start off with the definition:

Hide Copy Code

```
#pragma once
#include
#include
#include "camera.h"
#include "hpp/rs_context.hpp"

using namespace std;
using namespace rs2;

class cameras
{
public:
  cameras() {}
  ~cameras() {}
}
```

As you will remember, I said that I wanted the cameras to be enumerable so I need to do some upfront declarations:

Hide Copy Code

```
using cameras_t = vector<shared_ptr>;
using iterator = cameras_t::iterator;
```

`using const_iterator = cameras_t::const_iterator;`
[/source]

With these in place, I can now start to add the ability to enumerate over the `camera` instances. Before I do that, though, it's time to introduce something new. In the preceding code, we saw that the RealSense camera was represented as a device that we passed into the relevant constructors. The question is, how did we get that device in the first place? Well, that's down to the API providing a context that allows us to access these devices. So, let's add a member to store a vector of `camera` instances and then build in the method to get the list of devices.

Hide Copy Code

```
cameras() : cameras_(make_shared())
{
  context context;
  // Iterate over the devices;
  auto devices = context.query_devices();
  for (const auto dev : devices)
  {
    const auto cam = make_shared(dev);
    cameras_->push_back(cam);
  }
}

private:
  shared_ptr cameras_;
```

There's nothing complicated in that code. We get the devices from the context using `query_devices` and iterate over each one, adding a new camera instance to our vector. We have reached the point where we can now add the ability to enumerate over our vector. All the scaffolding is in place so let's add that capability.

Hide Copy Code

```
int capacity() const
{
  return cameras_->capacity();
}

iterator begin() { return cameras_->begin(); }
iterator end() { return cameras_->end(); }

const_iterator begin() const { return cameras_->begin(); }
const_iterator end() const { return cameras_->end(); }
const_iterator cbegin() const { return cameras_->cbegin(); }
const_iterator cend() const { return cameras_->cend(); }
```

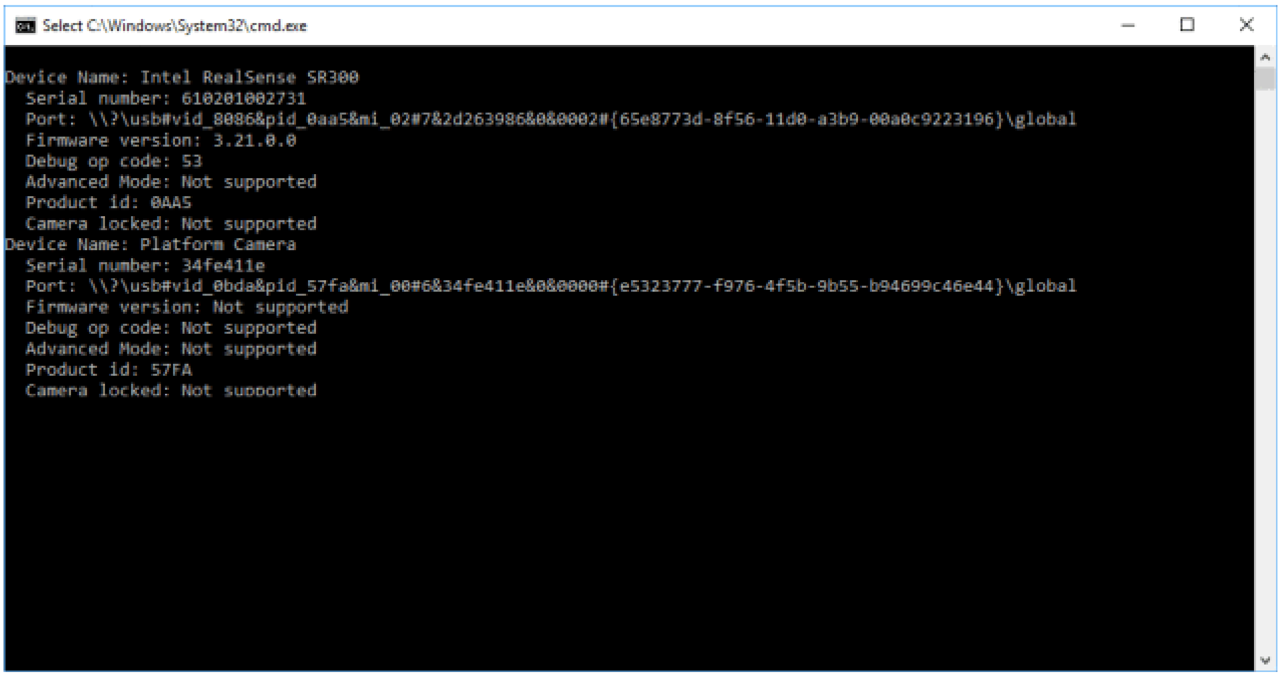
That's it. We now have the ability to build and iterate over the devices on our system. Let's see how we would go about using that. To test it, I created a little console application that I used to call my code. It's as easy as this:

Hide Copy Code

```
#include "stdafx.h"
#include "cameras.h"
#include

int main()
{
const auto devices = std::make_shared();
for (auto &dev : *devices)
{
cout <get_information()->dump_diagnostic();
}
return 0;
}
```

This is what it looks like on my system (running a web camera and a separate RealSense camera).



Select C:\Windows\System32\cmd.exe

```
Device Name: Intel RealSense SR300
  Serial number: 610201002731
  Port: \\?\usb#vid_8086&pid_0aa5&mi_02#7&2d263986&0&0002#{65e8773d-8f56-11d0-a3b9-00a0c9223196}\global
  Firmware version: 3.21.0.0
  Debug op code: 53
  Advanced Mode: Not supported
  Product id: 0AA5
  Camera locked: Not supported
Device Name: Platform Camera
  Serial number: 34fe411e
  Port: \\?\usb#vid_0bda&pid_57fa&mi_00#6&34fe411e&0&0000#{e5323777-f976-4f5b-9b55-b94699c46e44}\global
  Firmware version: Not supported
  Debug op code: Not supported
  Advanced Mode: Not supported
  Product id: 57FA
  Camera locked: Not supported
```

License

Share

TWITTER

About the Author

Pete O'Hanlon

CEO
United
Kingdom

A developer for over 30 years, I've been lucky enough to write articles and applications for Code Project as well as the Intel Ultimate Coder - Going Perceptual challenge. I live in the North East of England with 2 wonderful daughters and a wonderful wife.

I am not the Stig, but I do wish I had Lotus Tuned Suspension.

You may also be interested in...

Connecting Intel® RealSense™ 3D Camera With the Intel® Edison - JavaScript

Window Tabs (WndTabs) Add-In for DevStudio

Intel® RealSense™ SDK Architecture

Introduction to D3DImage

SAPrefs - Netscape-like Preferences Dialog

Creating alternate GUI using Vector Art

Comments and Discussions

You must Sign In to use this message board.

Search Comments

Spacing Relaxed Layout Normal Per page 25 Update

-- There are no messages in this forum --