Walkthrough

Here is an example to walk you through the API.
See `examples/Protonect.cpp` for the full source.

Headers

First, include necessary headers. **`registration.h`** and **`logger.h`** are optional if you don't use them.

```
#include <libfreenect2/libfreenect2.hpp>
#include <libfreenect2/frame_listener_impl.h>
#include <libfreenect2/registration.h>
#include <libfreenect2/packet_pipeline.h>
#include <libfreenect2/logger.h>
```

Logging

This shows how to set up the logger and logging level.

```
       libfreenect2::setGlobalLogger(libfreenect2::createCons
       oleLogger(libfreenect2::Logger::Debug));
```

Though libfreenect2 will have an initial global logger created with **createConsoleLoggerWithDefaultLevel()**. You do not have to explicitly call this if the default is already what you want.

You can implement a custom **Logger** and redirect libfreenect2's log messages to desired places.

Here is an example to save log messages to a file.

```
#include <fstream>
#include <cstdlib>
class MyFileLogger: public libfreenect2::Logger
{
private:
  std::ofstream logfile_;
public:
  MyFileLogger(const char *filename)
  {
    if (filename)
      logfile_.open(filename);
    level_ = Debug;
  }
  bool good()
  {
    return logfile_.is_open() && logfile_.good();
  }
  virtual void log(Level level, const std::string &message)
  {
    logfile_ << "[" <<
       libfreenect2::Logger::level2str(level) << "] " <<
       message << std::endl;
  }
};
```

And use it

```
   MyFileLogger *filelogger = new
        MyFileLogger(getenv("LOGFILE"));
   if (filelogger->good())
     libfreenect2::setGlobalLogger(filelogger);
   else
     delete filelogger;
```

libfreenect2 uses a single global logger regardless of number of contexts and devices. You may have to implement thread safety measure in **log()**, which is called from multiple threads. Console loggers are thread safe because `std::cout` and `std::cerr` are thread safe.

Initialize and Discover Devices

You need these structures for all operations. Here it uses only one device.

```
   libfreenect2::Freenect2 freenect2;
   libfreenect2::Freenect2Device *dev = 0;
   libfreenect2::PacketPipeline *pipeline = 0;
```

You must enumerate all Kinect v2 devices before doing anything else related to devices.

```
   if(freenect2.enumerateDevices() == 0)
   {
     std::cout << "no device connected!" << std::endl;
     return -1;
   }

   if (serial == "")
   {
     serial = freenect2.getDefaultDeviceSerialNumber();
   }
```

Also, you can create a specific **PacketPipeline** instead using the default one for opening the device. Alternatives include **OpenGLPacketPipeline**, **OpenCLPacketPipeline**, etc.

```
         pipeline = new libfreenect2::CpuPacketPipeline();
```

使用如cpu、gpu、opencl、等

Open and Configure the Device

Now you can open the device by its serial number, and using the specific pipeline.

```
     dev = freenect2.openDevice(serial, pipeline);
```

You can also open the device without providing a pipeline, then a default is used. There are a few alternative ways to **openDevice()**.

After opening, you need to attach **Framelisteners** to the device to receive images frames.

This **SyncMultiFrameListener** will wait until all specified types of frames are received once. Like loggers, you may also implement your own frame listeners using the same interface.

```
int types = 0;
if (enable_rgb)
  types |= libfreenect2::Frame::Color;
if (enable_depth)
  types |= libfreenect2::Frame::Ir |
     libfreenect2::Frame::Depth;
libfreenect2::SyncMultiFrameListener listener(types);
libfreenect2::FrameMap frames;

dev->setColorFrameListener(&listener);
dev->setIrAndDepthFrameListener(&listener);
```

You cannot configure the device after starting it.

Start the Device

After finishing configuring the device, you can start the device. You must start the device before querying any information of the device.

```
if (enable_rgb && enable_depth)
{
  if (!dev->start())
    return -1;
}
else
{
  if (!dev->startStreams(enable_rgb, enable_depth))
    return -1;
}

std::cout << "device serial: " << dev->getSerialNumber()
     << std::endl;
std::cout << "device firmware: " << dev-
     >getFirmwareVersion() << std::endl;
```

You can **setIrCameraParams()** after start if you have your own depth calibration parameters.

Otherwise you can also use the factory preset parameters for **Registration**. You can also provide your own depth calibration parameterss (though not color camera calibration parameters right now). Registration is optional.

```
libfreenect2::Registration* registration = new
     libfreenect2::Registration(dev->getIrCameraParams(),
        dev->getColorCameraParams());
  libfreenect2::Frame undistorted(512, 424, 4),
        registered(512, 424, 4);
```

At this time, the processing has begun, and the data flows through the pipeline towards your frame listeners.

Receive Image Frames

This example uses a loop to receive image frames.

```
  while(!protonect_shutdown && (framemax == (size_t)-1 ||
        framecount < framemax))
  {
    if (!listener.waitForNewFrame(frames, 10*1000)) // 10
        sconds
    {
      std::cout << "timeout!" << std::endl;
      return -1;
    }
    libfreenect2::Frame *rgb =
        frames[libfreenect2::Frame::Color];
    libfreenect2::Frame *ir =
        frames[libfreenect2::Frame::Ir];
    libfreenect2::Frame *depth =
        frames[libfreenect2::Frame::Depth];
```

waitForNewFrame() here will block until required frames are all received, and then you can extract `Frame` according to the type.

See **libfreenect2::Frame** for details about pixel format, dimensions, and metadata.

You can do your own things using the frame data. You can feed it to OpenCV, PCL, etc. Here, you can perform registration:

```
      registration->apply(rgb, depth, &undistorted,
       &registered);
```

After you are done with this frame, you must release it.

```
    listener.release(frames);
  }
```

Stop the Device

If you are finished and no longer need to receive more frames, you can stop the device and exit.

```
  dev->stop();
  dev->close();
```

Pause the Device

You can also temporarily pause the device
with **stop()** and **start()**.

```
  if (protonect_paused)
    devtopause->start();
  else
    devtopause->stop();
  protonect_paused = !protonect_paused;
```

Doing this during `waitForNewFrame()` should be thread safe, and tests also show well. But a guarantee of thread safety has not been checked yet.

THE END.