

Developing with the DLP® Structured Light SDK

Developing the DLP® Structured Light SDK

- Setting up the development environment
 - Install the DLP Structured Light SDK
 - Install and compile OpenCV
- Scalable solutions with object oriented programming (OOP) in C++
 - Case Study: Consider that the 3D Scanner Demo software is practically identical for the DLP LightCrafter™ 4500 EVM and DLP LightCrafter 6500 EVM (and now the DLP LightCrafter 3000 EVM)
 - What are the primary abstract modules?
 - How to use abstracted modules?
- Creating new modules
 - Where should source code go?
 - How to creating a new camera module?


Setting up the development environment

- DLP® Structured Light SDK
 - Currently included with 3D Machine Vision Reference Design
 - As new DLP evaluation modules are added (DLP LightCrafter, LightCrafter 6500 EVM, etc.) the DLP Structured Light SDK source code will move to its own tool page
 - This is to prevent duplication of code on ti.com
- OpenCV
 - Download the source code from www.opencv.org
 - Brief instructions
 - Install CMake
 - Use CMake to create Makefile
 - Compile with make
 - Detailed compilation instructions are available in the Machine Vision Reference Design User's Guide

Scalable Solutions with OOP

- Consider the DLP® LightCrafter™ 4500 and LightCrafter 6500 EVMs
 - Both chipsets have different API, resolutions, speeds, etc.
 - Did I need to rewrite each application with all of the chipset specific API ?
 - NO! Only a single line of code needed to change

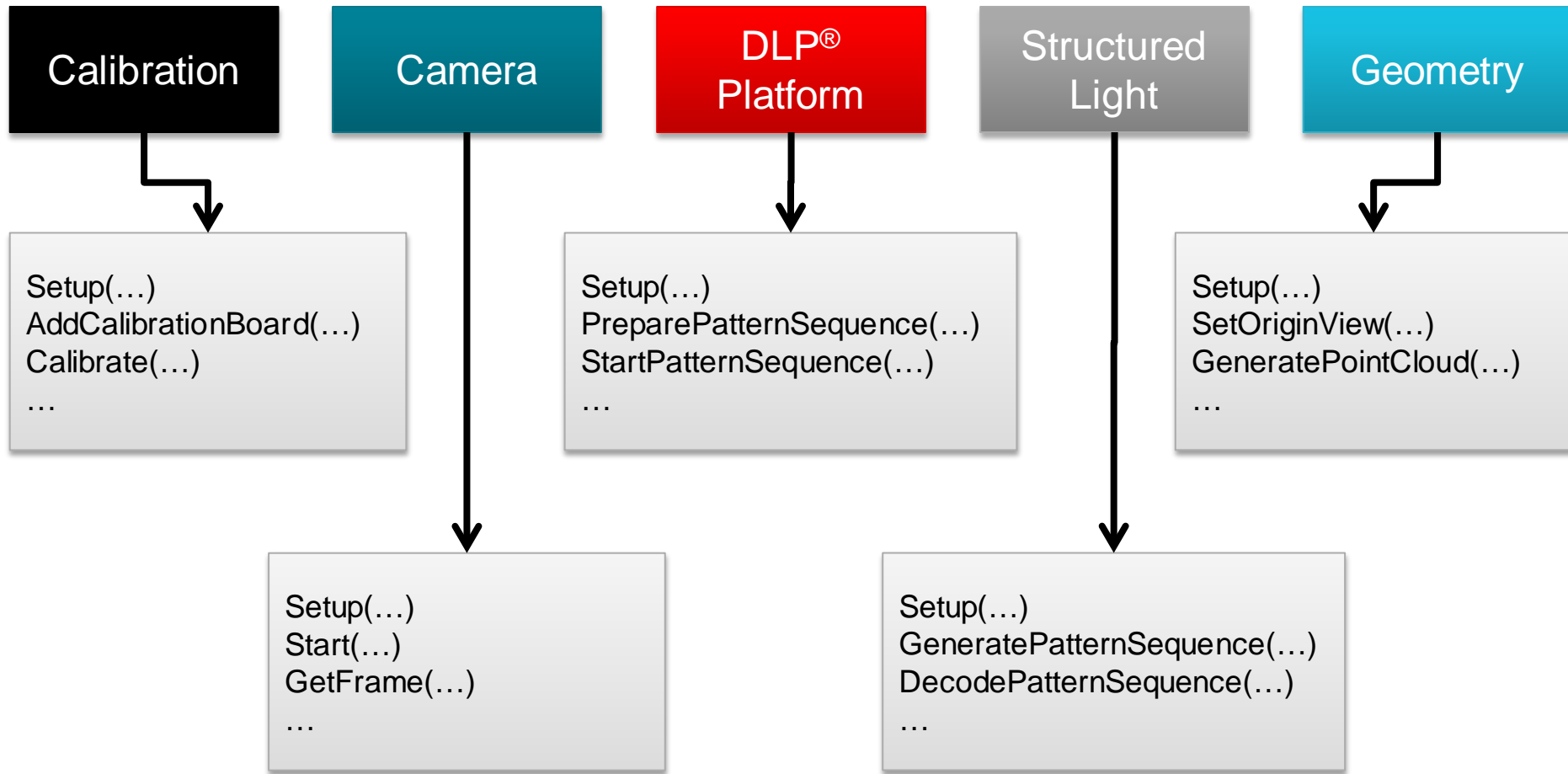
```
1412 // System Variables
1413 dlp::PG_FlyCap2_C camera;
1414 dlp::LCr4500 projector;
```



```
1412 // System Variables
1413 dlp::PG_FlyCap2_C camera;
1414 dlp::LCr6500 projector;
```

- **The DLP LightCrafter 3000 EVM has recently been added also!**
- How is this possible?
 - DLP Structured Light SDK contains modules which define interfaces
 - C++ allows you to reference sub-classes as their parent class

What are the primary abstract modules?



- Each module (base-class) defines an interface which all sub-modules (sub-classes) must follow

How to use abstracted modules?

- Function Declaration uses the abstracted base-class modules

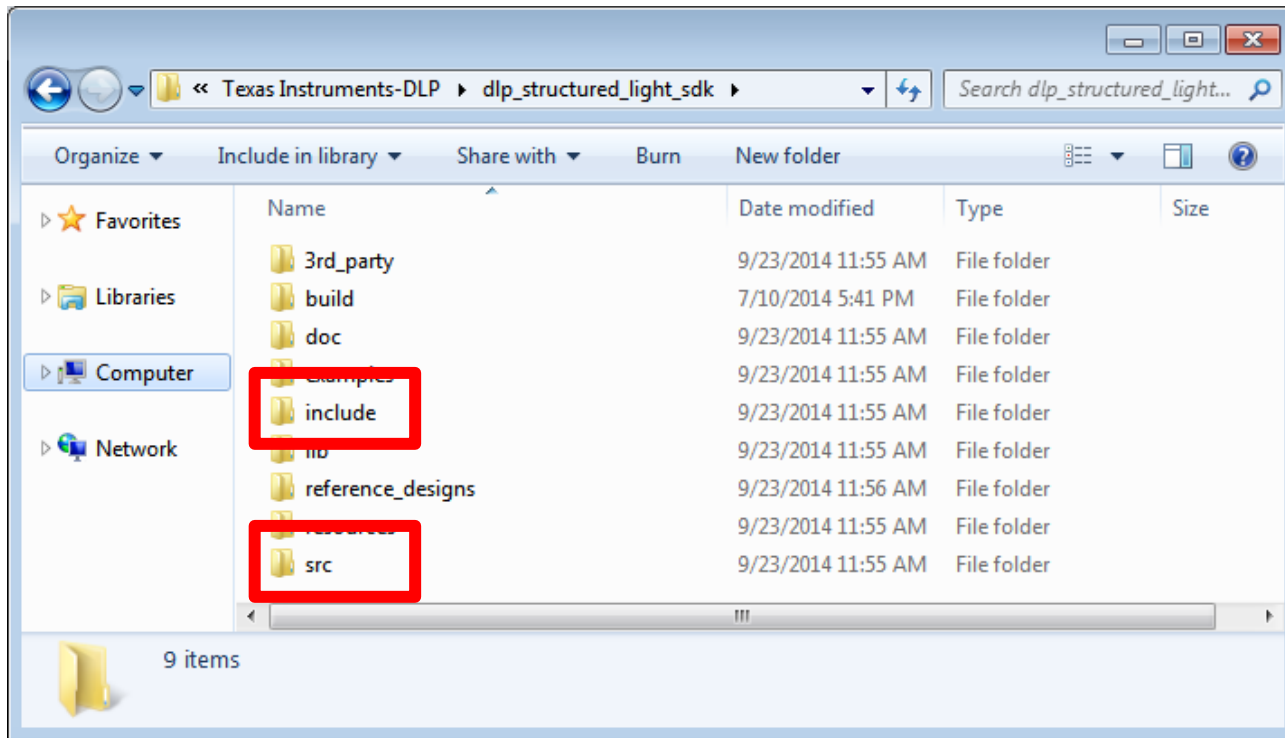
```
void ScanObject(dlp::Camera      *camera,  
               const std::string &camera_calib_data_file,  
               dlp::DLP_Platform *projector,  
               const std::string &projector_calib_data_file,  
               dlp::StructuredLight *structured_light_vertical,  
               dlp::StructuredLight *structured_light_horizontal,  
               const bool        &use_vertical,  
               const bool        &use_horizontal,  
               const std::string &geometry_settings_file){
```

- Specific sub-modules can be passed as the parent type though

```
// System Variables  
dlp::PG_FlyCap2_C camera;  
dlp::LCr6500      projector;  
  
ScanObject(&camera,  
          calibration_data_file_camera,  
          &projector,  
          calibration_data_file_projector,  
          &structured_light_vertical,  
          &structured_light_horizontal,  
          true,  
          true,  
          geometry_settings_file);
```

Where should source code go?

- Use the current sub-modules for reference
 - Header files should be located in the /include directory
 - Source files should be located in the /src directory



- Add new source files to QT PRO file or CMakeLists.txt

How to creating a new camera module?

- Reference the module base-class header files to identify what functions need to be written for a sub-class
 - All virtual functions must be written by the sub-class!

```
class Camera: public dlp::Module{  
    public:  
  
    // Define by subclass  
    virtual ReturnCode Connect(int camera_id) = 0;  
    virtual ReturnCode Disconnect() = 0;  
  
    virtual ReturnCode Start() = 0;  
    virtual ReturnCode Stop() = 0;  
  
    virtual ReturnCode GetFrame(Image* ret_frame) = 0;  
}
```

NOTE: This is only an example!! Please reference camera.hpp for the complete camera module declaration

```
class PG_FlyCap2_C : public Camera  
{  
    public:  
  
    // Define pure virtual functions  
    ReturnCode Connect(int camera_id);  
    ReturnCode Disconnect();  
  
    ReturnCode Start();  
    ReturnCode Stop();  
  
    ReturnCode GetFrame(Image* ret_frame);  
}
```

Notice that the sub-class contains the exact same methods