# Intel® RealSense™ Product Family D400 Series Calibration Tools and API

## Programmer's Guide

*July 2021*

*Version 2.11.1.0*

Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

Intel technologies may require enabled hardware, specific software, or services activation. Check with your system manufacturer or retailer.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or visit www.intel.com/design/literature.htm.

Intel and the Intel logo Xeon, Core trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

intel REALSENSE™

# Contents


intel
REALSENSE™

# Figures

# Revision History

| Revision Number | Description | Revision Date |
|---|---|---|
| 2.5.2.0 | First document for Calibration Tool version 2.5.2.0 | Jan 2018 |
| 2.6.4.0 | Support RGB calibration | March 2018 |
| 2.6.8.0 | Support D435i<br>Custom Calibration Data R/W API supports IMU custom data R/W<br>Support calibration raw data and fisheye custom data read/write | December 2018 |
| 2.8.0.0 | Advanced calibration processes for robotics | March 2019 |
| 2.11.0.0 | Support Intel® RealSense™ Depth Camera D455<br>New DC algo compatible with production calibration algo v5.1<br>Support Linux (Ubuntu 16.04 and Ubuntu 18.04) on ARM platforms (Nvidia Jetson TX2, Xavier, and Firefly RK3399)<br>Fix libpng dependency issue on Ubuntu 18.04<br>Supports Ubuntu 18.04 kernel 5.x<br>Calibration API is combined into Calibration Tool package | July 2020 |
| 2.11.1.0 | Support Thermal Compensation on D455 | July 2021 |

§§

# 1 Scope

This document is the programmer's guide of the Calibration API for Intel® RealSense™ technology. The API is an extension of the Calibration Tools to enable users developing their own calibration applications based on Intel calibration algorithms or user's own custom algorithms on Intel® RealSense™ Product Family D400 Series devices.

The Calibration API for Intel® RealSense™ technology includes the following:

- **Dynamic Calibration API**
  This is a set of interfaces and libraries to calibrate the D400 devices with the same Intel algorithms in the Intel® RealSense™ Dynamic Calibrator. The API supports both targeted and target-less calibration. Users can use the API to build their own calibration application to fit into their specific usage needs.

  The API also includes custom calibration data read/write interfaces to enable calibration with user chosen non-Intel calibration algorithms.

  On very limited device SKU, this API also supports custom calibration data read/write for fisheye.

- **Examples**
  Several working examples are provided with source code as reference to use the Calibration API, including sources for the Intel® RealSense™ Dynamic Calibrator and CustomRW for Intel® RealSense™ technology tools.

  **Important Notes**: Users are free to use the example code, but the Intel calibration algorithms contained in the API library is not open source and only available as binary form. The primary goal is to enable users to develop their calibration application with the embedded Intel calibration algorithm or their custom calibration algorithms.

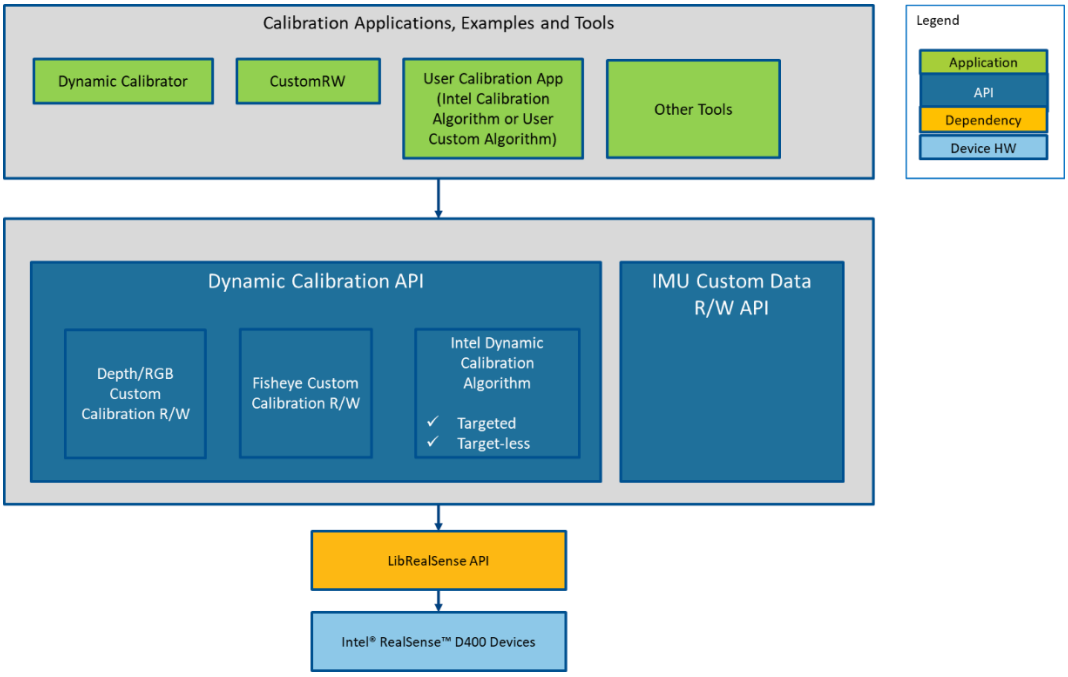
  §§

# 2 Software Architecture and Calibration Flow

## 2.1 Software Stack

The following diagram illustrates the calibration software stack and its components.

**Figure 2-1 Calibration Software Stack**




**Dynamic Calibration API**: A set of APIs that provide interfaces for application to use dynamic calibration algorithms in real time re-calibration. It also contains interfaces for custom calibration R/W for Depth/RGB/Fisheye.

**Dynamic Calibration Algorithms**: A software library that contains the real time re-calibration algorithms, for example, rectification and scale calibration.

**LibRealSense API**: open source camera access API for capture images, read/write to device and other camera controls.

**Camera Device**: Any supported Intel RealSense D400 series Depth Cameras with proper FW updated.

## 2.2 Calibration API

The Calibration API is part of the Calibration Tool package and can be copied from the installed directory. Please refer to chapter ***Developing Custom Apps with Calibration API*** in the User's Guide for details.

## 2.3 API Package Location

A compressed package of the interface and examples is included in the tool installation.

On **Linux**:

```
/usr
  |
  |----- share
  |         |---- doc
  |                |..... librscalibrationtool
  |                          |
  |                          |---- api
  |                                  |---- DynamicCalibrationAPI-Linux-2.11.0.0.tar.gz
```

On **Windows**:

```
C:\Program Files\Intel\CalibrationToolAPI
                              |-------- 2.11.0.0
                              |            |
                              |            |-------- api
                              |            |          |-------- DynamicCalibrationAPI.Zip
```

## 2.4 API Package Structure

Copy DynamicCalibrationAPI-Linux-2.11.0.0.tar.gz or DynamicCalibrationAPI.Zip to your working folder and unzip it into similar directory structure as below:

```
DynamicCalibrationAPI
        |
        |---- 2.11.0.0
        |         |---- CMakeLists.txt
        |         |
        |         |---- lib
```

```
|       |      |---- libDSDynamicCalibrationAPI.so or
DSDynamicCalibrationAPI.dll
|       |
|       |..... examples
|       |          |---- DynamicCalibrator
|       |          |---- CustomRW
|       |          |---- CustomCalibration
|       |          |---- rs-calibration-converter
|       |          |---- simple-rw
|       |          |---- simple-fe
|       |
|       |----Include
|       |      |---- DSDynamicCalibration.h
|       |      |---- DSOSUtils.h
|       |      |---- DSShared.h
|       |      |---- DSCalData.h
|       |
|       |..... attributions.txt
|       |..... License.txt
|       |..... README.md
|       |..... target
|       |---- librealsense
|       |---- 3rdparty
```

Notes:

- API - libDSDynamicCalibrationAPI.so or DSDynamicCalibrationAPI.DLL is the calibration API library and DSDynamicCalibration.h is its main header file
- Under examples folder
  - ✓ DynamicCalibrator contains the source code for the Dynamic Calibrator
  - ✓ CustomRW contains the source code for the CustomRW tool
  - ✓ CustomCalibration contains sources for an example custom calibration of depth and RGB sensors with non-Intel algorithms, the example is described in detail in the *Intel RealSense Depth Module D400 Series Custom Calibration w*hite paper

## 2.5 Compiling Examples

The API package support cmake.

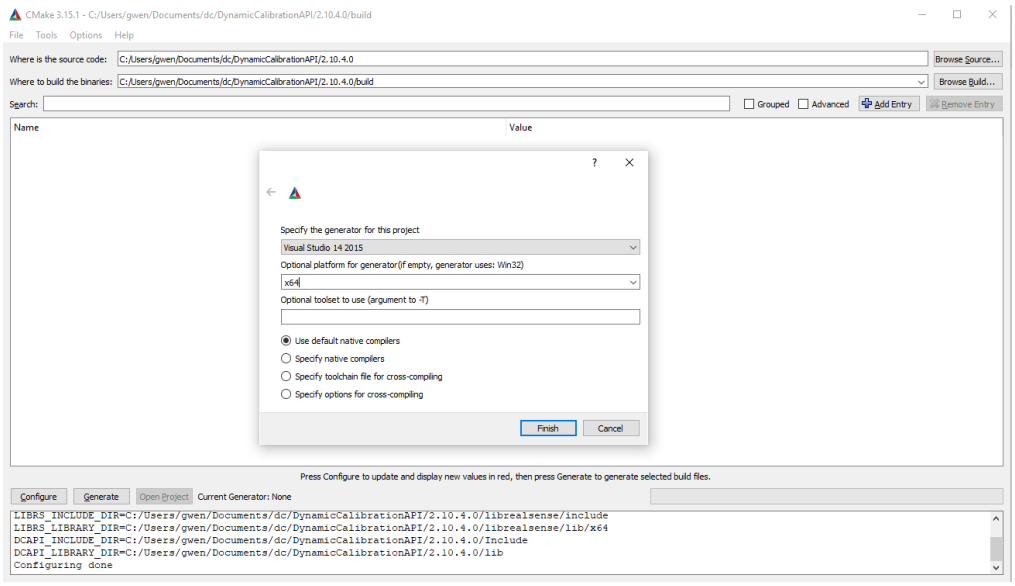
On **Linux**:

```
cd DynamicCalibrationAPI/2.11.0.0
mkdir build
```

```
cd build
cmake ..
        -- The C compiler identification is GNU 5.4.0
        -- The CXX compiler identification is GNU 5.4.0
        -- Check for working C compiler: /usr/bin/cc
        -- Check for working C compiler: /usr/bin/cc -- works
        -- Detecting C compiler ABI info
        -- Detecting C compiler ABI info - done
        -- Detecting C compile features
        -- Detecting C compile features - done
        -- Check for working CXX compiler: /usr/bin/c++
        -- Check for working CXX compiler: /usr/bin/c++ -- works
        -- Detecting CXX compiler ABI info
        -- Detecting CXX compiler ABI info - done
        -- Detecting CXX compile features
        -- Detecting CXX compile features - done
        -- CMAKE_SYSTEM_PROCCESSOR=x86_64
        -- BUILD_SHARED_LIBS=ON
        -- LIBRS_INCLUDE_DIR=/home/gwen/dc/test/DynamicCalibrationAPI/2.10.4.0/librealsense/include
        -- LIBRS_LIBRARY_DIR=/home/gwen/dc/test/DynamicCalibrationAPI/2.10.4.0/librealsense/lib/linux/x86_64
        -- DCAPI_INCLUDE_DIR=/home/gwen/dc/test/DynamicCalibrationAPI/2.10.4.0/Include
        -- DCAPI_LIBRARY_DIR=/home/gwen/dc/test/DynamicCalibrationAPI/2.10.4.0/lib
        -- Configuring done
        -- Generating done
        -- Build files have been written to: /home/gwen/dc/test/DynamicCalibrationAPI/2.10.4.0/build
make -j8
```

On **Windows**:

Config cmake for compilaing CalibrationToolAPI\2.11.0.0 and generate project files for Visual Studio 2015 and x64 platform.




Open rs-dynamic-calibration-api-samples.sln under build directory to compile.

## 2.6 Calibration Parameters

A set of calibration parameters are used to describe the basic characteristics of the device.

**Intrinsic** includes

- Focal length - specified as [fx; fy] in pixels for left, right, and RGB cameras
- Principal point - specified as [px; py] in pixels for left, right, and RGB cameras
- Distortion - specified as Brown's distortion model [k1; k2; p1; p2; k3] for left, right, and RGB cameras

**Extrinsic** includes

- RotationLeftRight - rotation from right camera coordinate system to left camera coordinate system, specified as a 3x3 rotation matrix
- TranslationLeftRight - translation from right camera coordinate system to left camera coordinate system, specified as a 3x1 vector in millimeters
- RotationLeftRGB - rotation from RGB camera coordinate system to left camera coordinate system, specified as a 3x3 rotation matrix
- TranslationLeftRGB - translation from RGB camera coordinate system to left camera coordinate system, specified as a 3x1 vector in millimeters

The left camera is the reference camera and is located at world origin. RGB parameters only apply to depth modules/cameras with RGB sensor for color, e.g., D415, D435, D435i, and D455.

## 2.7 Dynamic Calibration with Intel Algorithms

Dynamic Calibration optimizes extrinsic parameters, i.e., they refer to calibration done in the field at the user environment with minimal or no user intervention. They are ONLY extrinsic parameters (translation and rotation) of the camera image with regard to the main axis system (the axis between the left and right). Intrinsic parameters such as distortion, field of view, and principal point are not dynamically calibrated.

Dynamic calibration is run under the assumption that it is the re-calibration of the depth modules/cameras after factory calibration or Intel OEM Calibration, or at least that the nominal parameters are known.

Two different types of dynamic calibrations algorithms are supported:

- Rectification calibration: aligning the epipolar line to enable the depth pipeline to work correctly and reduce the holes in the depth image
- Depth scale calibration: aligning the depth frame due to changes in position of the optical elements

Dynamic Calibration API supports these algorithms in two distinctive operating modes based on these algorithms: **targeted** and **target-less,** each with a different use flow.

**Important Notes**:

- The **Intel® RealSense™ Dynamic Calibrator** in the Calibration Tool package supports only targeted calibration.
- Targeted calibration is the recommended approach because it includes both rectification and depth scale calibrations and will give more accurate and consistent results than rectification only calibration done in target-less calibration.

### 2.7.1 Target-less

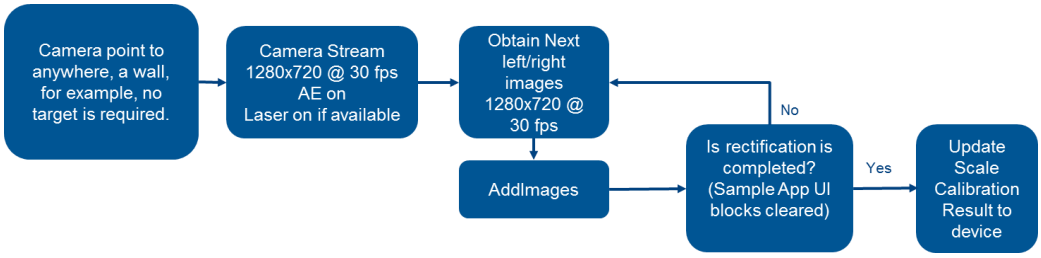
In target-less mode, Dynamic Calibration API supports rectification calibration without the need of any target.

Its basic flow is summarized as following:

1. Take the rectified images from L and R camera in real-time
2. Extract features from the images
3. Match the features between L and R camera

4. The image is binned into 6x8 grid. Check whether each bin contains enough corresponding points.
5. The user sees the panel status and moves the device around (bins without features are blue).
6. Steps 1-6 are iterated until all bins have enough feature points
7. Check rectification error (absolute Y difference). If too large, run a solver to optimize extrinsic parameters to minimize it.

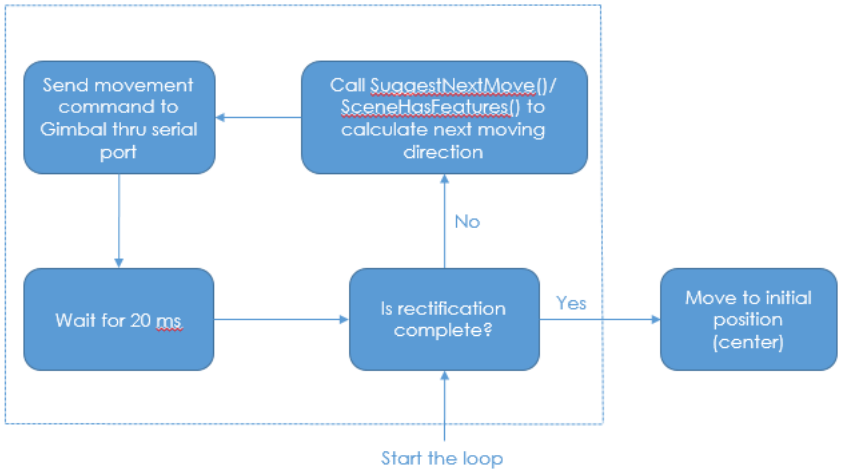
**Figure 2-2 Target-less Calibration Flow**




By default, target-less calibration supports 1280x720 resolution images. On devices with wide angle lenses such as D420, D430, D435, and D435i additional native resolution 1280x800 is also supported.

If Gimbal is used with target-less calibration, it will operate independently of the above flow (Figure 7-1) to move the camera automatically based on scene detection results:

**Figure 2-3 Gimbal Flow with Target-less Calibration**



Gimbal is supported only for target-less calibration at experimental basis. Targeted calibration is not supported.

**Important Notes**:

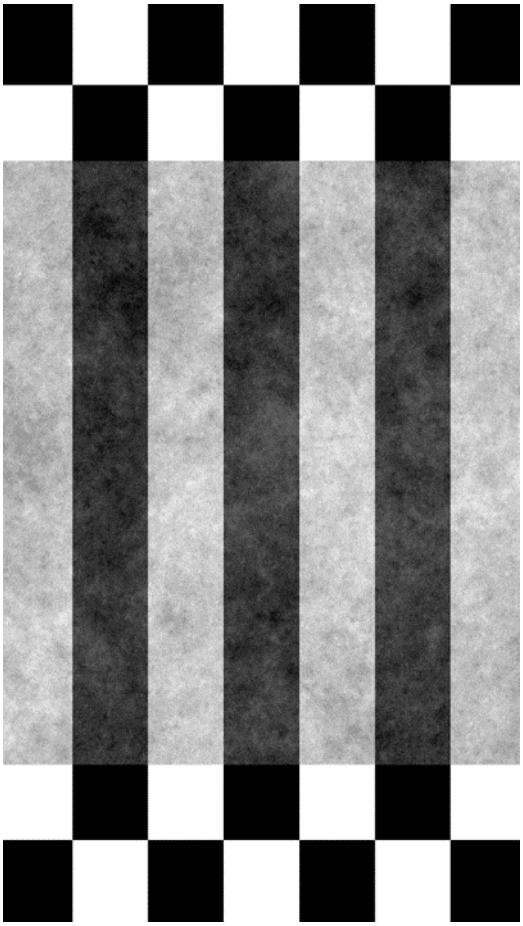
- Targeted calibration is likely to give more accurate results and is therefore the recommended approach.
- The advantage of target-less calibration is simplicity, no calibration target is needed. It fits with user cases where calibrating with a target may not be feasible. The disadvantage is that it calibrates left/right depth but not RGB and generally less accurate and less consistent than targeted calibration.

### 2.7.2 Target-based

In targeted mode, Dynamic Calibration API supports depth scale calibration and a target is required. The target is predefined and can be printed on legal size paper or displayed on a smartphone through phone app.

An example of the target image is shown below. The printed target image is included in print-target-fixed-width.pdf which can be downloaded from Intel® RealSense™ product website or Intel® RealSense™ Calibration Tool package. On phone devices, the actual image is device dependent and dynamically generated at runtime through the Intel® RealSense™ Dynamic Target Tool phone app. The app is available on Apple app store or Google Play.

**Figure 2-4 Target Image**



As part of the process, the application acquires images of the calibration target and feed into dynamic calibration algorithm. The process is divided into two consecutive steps: rectification phase and scale phase, as shown below. The rectification phase fixes rectification issues so the device stream is corrected with sufficient depth quality for the next phase to fix scaling issues. The process requires images from simultaneous depth, rectified left and right images at 1280x720 resolution.

Figure 2-5 Target-based Dynamic Calibration

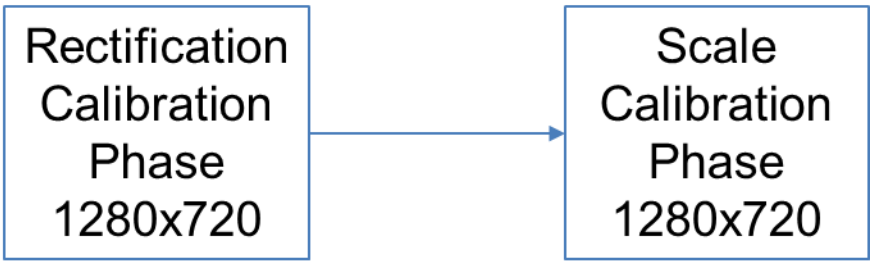
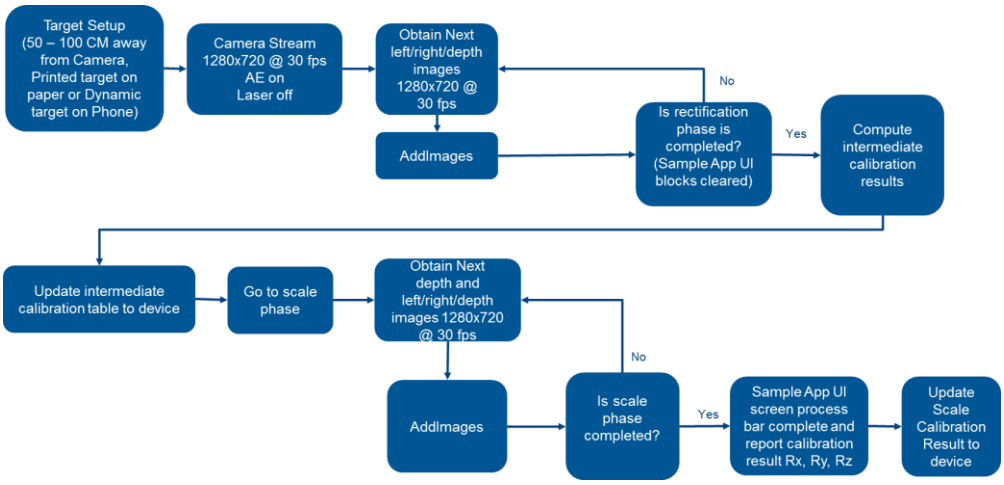



Figure 2-4 highlights a high level flow and shows how targeted dynamic calibration is implemented based on the two-phase algorithm:

1) Take the images from L and R camera, including the depth stream (in real-time)
2) Detect the calibration target in both images
3) User moves the printed target or phone so that it covered most of the image, repeating steps 1-2
4) Once done, user just keeps taking images by positioning the phone anywhere in the image but must move the phone every time
5) After taking 15 images in step 4, the process is complete
6) The process checks for rectification error (absolute Y difference) but also compares the measured pattern size with ground truth

Figure 2-6 Targeted Dynamic Calibration Integrated Two Phase Flow




The advantage of targeted calibration is that it's accurate and consistent. It calibrates left/right depth as well as RGB (on devices with RGB). The disadvantage is that it

requires calibration target which means it cannot be used in cases where calibrating with a target is not feasible.

### 2.7.3 Hybrid

Hybrid Calibration combines best of Target-less Calibration (convenience) and Targeted Calibration (accuracy) into a single process. It starts with a target-less calibration to rectifies the left/right images and then followed with a targeted scale only calibration. It provides flexibility as well as accuracy.

**Figure 2-7 Hybrid Dynamic Calibration Integrated Two (or Three) Phase Flow**

### 2.7.4 Image Formats Used in Dynamic Calibration

Targeted Dynamic Calibration requires simultaneous images from depth, rectified left and right imagers in ZR8L8 format which is only available when the device is running under advanced mode.

Target-less Dynamic Calibration requires simultaneous rectified images from left and right imagers in ZR8L8 format.

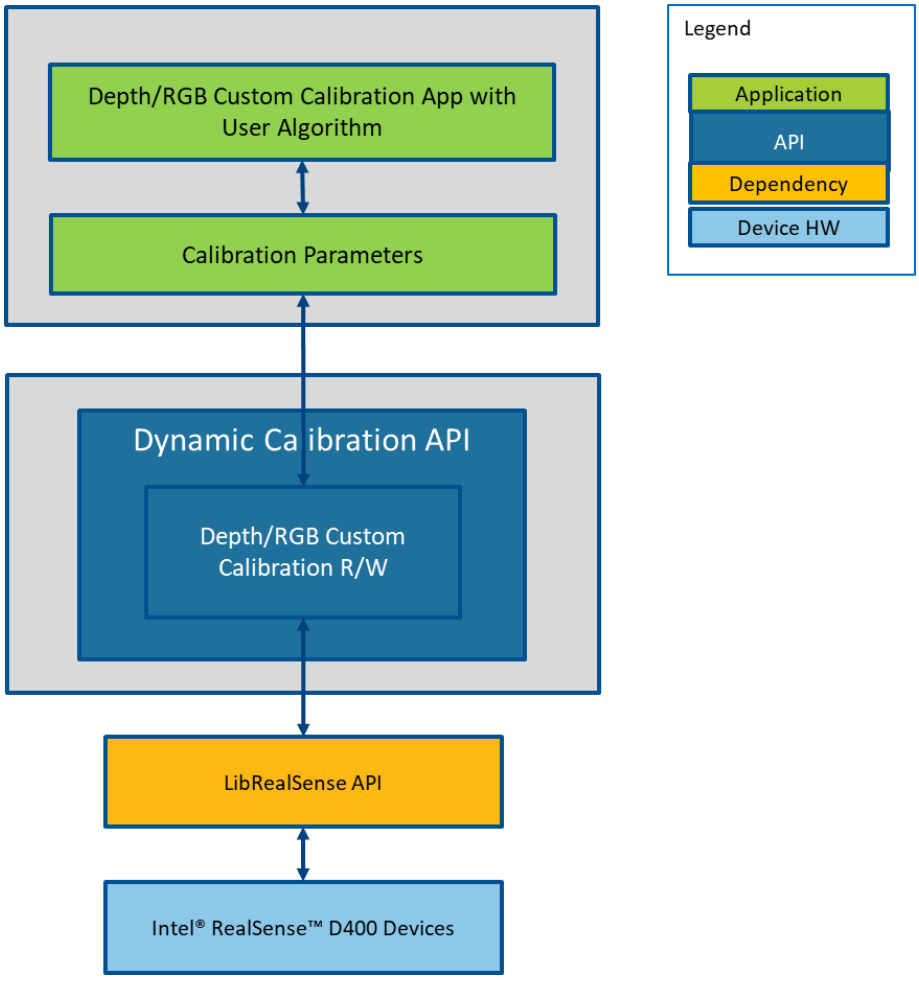
| Format | SKU | Used | Dynamic Calibration Type |
|---|---|---|---|
| ZR8L8 | D400<br>D410<br>D415 | Dynamic Calibration<br>Targeted Calibration:<br>1280x720 @ 30 fps | Targeted Dynamic Calibration |
| | D420<br>D430<br>D435<br>D435i | Dynamic Calibration<br>Targeted Calibration:<br>1280x720 @ 30 fps | |
| RY8_LY8 [16 bits] | D400<br>D410<br>D415 | Dynamic Calibration<br>Target-less Calibration:<br>1280x720 @ 30 fps or<br>1920x1080 @ 25 fps | Target-less Dynamic Calibration |
| | D420<br>D430<br>D435<br>D435i | Dynamic Calibration<br>Target-less Calibration:<br>1280x720 @ 30 fps or<br>1280x800 @ 30 fps | |

## 2.8 Depth/RGB Custom Calibration with User Algorithms

In some cases, user may prefer to calibrate the D400 devices with their own algorithms instead of the Intel algorithms, however, the calibration data will still need to be updated to the device for the optimization to take effect in actual device performance.

The Custom Calibration R/W interfaces in the Dynamic Calibration API provide mechanisms to read and write the high-level calibration parameters described in section 2.2 from/to the device. The API converts the high-level parameters into internal formats that the device can understand so user only works with those high-level parameters.

Figure 2-8 Depth/RGB Custom Calibration R/W with Dynamic Calibration API




A detailed example is described in the white paper *Intel® RealSense™ Product Family D400 Series Custom Calibration* which is available on Intel® RealSense™ product website:

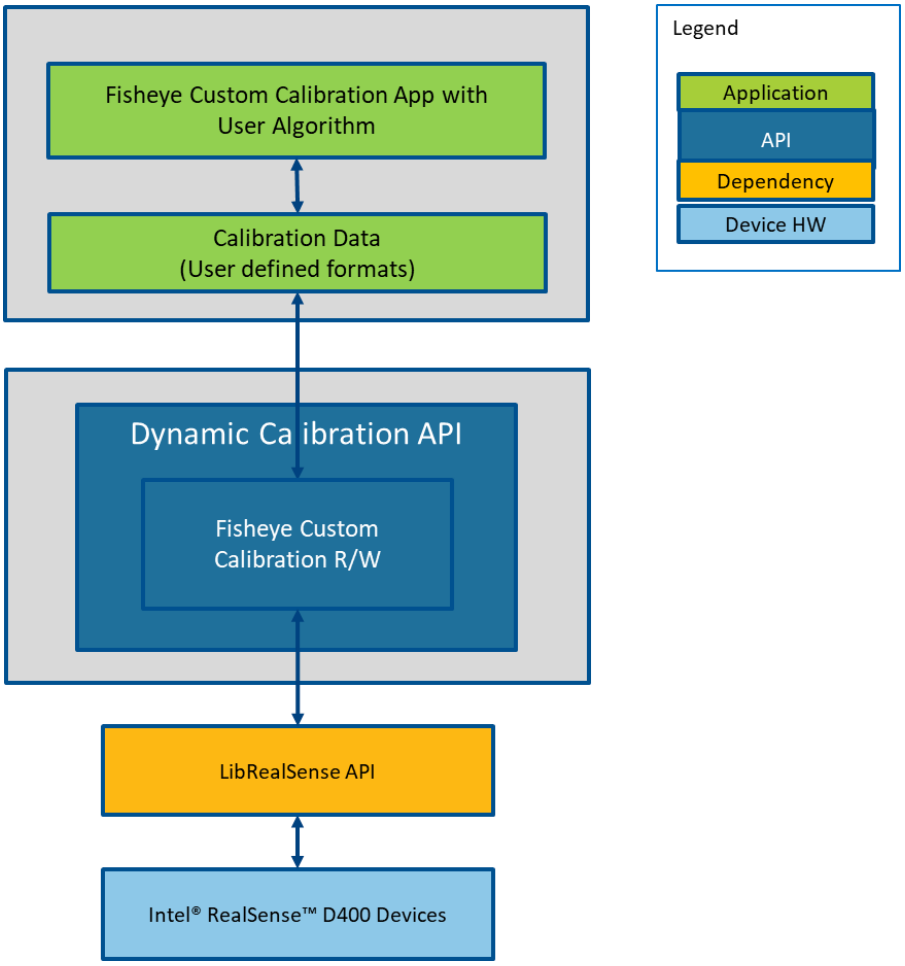
https://www.intel.com/content/www/us/en/support/articles/000026725/emerging-technologies/intel-realsense-technology.html

## 2.9 Fisheye Custom Calibration with User Algorithms

**(Supported only on limited devices with a fisheye)**

Devices in very limited SKU contains a fisheye sensor but it's not calibrated. User will need to calibrate the fisheye with their own methods and store the calibration data on the device and retrieve it later. The fisheye custom data R/W interfaces in Dynamic Calibration API provides such ability for the data storage on device. The API simply takes a byte buffer and does not restrict the exact memory layout. User is responsible for the calibration data format to write to and interpret it after read from the device through the API.

**Figure 2-9 Fisheye Custom Calibration R/W with Dynamic Calibration API**



# 3 Calibrating Depth/RGB with Intel Algorithm through Dynamic Calibration API

Dynamic Calibration API supports three usage modes: Intel target-less calibration algorithm, Intel targeted calibration algorithm, and user custom calibration algorithm.

```
/** Calibration Modes */
enum CalibrationMode
{
        /** Intel target-less calibration (target-less depth rectification only)*/
        CAL_MODE_INTEL_TARGETLESS = 0,
        /** Intel targeted calibration (targeted depth rectification and targeted depth scale)*/
        CAL_MODE_INTEL_TARGETED = 1,
        /** Intel targeted depth scale calibration only */
        CAL_MODE_INTEL_TARGETED_SCALE_ONLY = 3,
        /** Intel targeted RGB calibration */
        CAL_MODE_INTEL_RGB_CALIB = 4,
        /** User custom algorithm */
        CAL_MODE_USER_CUSTOM = 99                    };
```

Intel target-less calibration, Intel targeted calibration, Intel targeted scale only and RGB calibration algorithms are built into the library which user adjust parameters but cannot change the algorithm itself.

## 3.1 Targeted Calibration

The following calling sequence illustrates a simplified flow for Linux based targeted calibration which is a two-phase process: rectification and scale phase in sequential order. Please refer to API documentation and Sample app source code for details

```
// declare variable for Calibration Tool API and camera access api
DSDynamicCalibration g_Dyncal;

// get camera access handle from librealsense, for example, g_rHal
// initialize camera
// turn off laser projector (laser interferes with target images on phone screen)
// setup camera streaming resolution and frame rate for left/right/depth images
```

```
// initialize Calibration Tool API for targeted calibration
g_Dyncal.Initialize(g_rHal,
DynamicCalibrationAPI::DSDynamicCalibration::CAL_MODE_INTEL_TARGETED,
g_width, g_height, false);

// start capture with a callback function

// first phase - rectification
// in the callback function, check if calibration is completed and add images to
Calibration Tool API for processing
while (!g_Dyncal.IsGridFull())
{
   g_Dyncal.AddImages(leftImage, rightImage, depthImage, timeStamp);
}

// once first phase is completed, get the intermediate result and apply it to current
stream on device
g_Dyncal. SetIntermediateRectificationCalibration ()

// switch to second phase – scale
g_Dyncal.GoToScalePhase();
// add left/right/depth images to Calibration Tool API until scale phase is completed
while(!g_Dyncal.IsScalePhaseComplete())
{
   g_Dyncal.AddImages((uint8_t *)leftImage, (uint8_t *)rightImage, (uint16_t
*)depthImage, timeStamp);
}

// when calibration process is completed, stop capture

// get the updated calibration parameters and write to device
g_Dyncal->UpdateCalibrationTables ();

// release handles
```

## 3.2 Target-less Calibration

The following calling sequence illustrates a simplified flow for Linux based target-less calibration. Please refer to API documentation and sample app for details.

```
// declare variable for dynamic calibration api
DSDynamicCalibration g_Dyncal;

// get camera access handle g_rsHal
```

```
// initialize camera
// enable laser projector if available
// setup camera streaming resolution and frame rate for left/right images

// initialize Calibration Tool API for target-less calibration with laser projector on
g_Dyncal.Initialize(g_rsHal,
DynamicCalibrationAPI::DSDynamicCalibration::CAL_MODE_INTEL_TARGETLESS,
g_width, g_height, true);

// start capture with a callback function

// in the callback function, check if calibration is completed and add images to
Calibration Tool API for processing
while (!g_Dyncal.IsGridFull())
{
   g_Dyncal.AddImages(leftImage, rightImage, depthImage, timeStamp);
}

// when calibration is completed, stop capture

// get the updated calibration parameters and write to device
g_Dyncal->UpdateCalibrationTables ();

// release handles
```

## 3.3 Hybrid Calibration

The following calling sequence illustrates a simplified flow for Linux based hybrid calibration. Please refer to API documentation and sample app for details.

```
// declare variable for dynamic calibration api
DSDynamicCalibration g_Dyncal;

// get camera access handle g_rsHal

// initialize camera
// enable laser projector if available
// setup camera streaming resolution and frame rate for left/right images

// initialize Calibration Tool API for target-less calibration with laser projector on
```

```
g_Dyncal.Initialize(g_rsHal,
DynamicCalibrationAPI::DSDynamicCalibration::CAL_MODE_INTEL_TARGETLESS,
g_width, g_height, true);

// start capture with a callback function

// in the callback function, check if calibration is completed and add images to
Calibration Tool API for processing
while (!g_Dyncal.IsGridFull())
{
   g_Dyncal.AddImages(leftImage, rightImage, depthImage, timeStamp);
}

// when calibration is completed, stop capture

// get the updated calibration parameters and write to device
g_Dyncal->UpdateCalibrationTables ();

// release handles

// continues to targeted scale calibration

// initialize camera
// enable laser projector if available
// setup camera streaming resolution and frame rate for depth/left/right images

// initialize Calibration Tool API for targeted scale calibration with laser projector off
g_Dyncal.Initialize(g_rsHal,
DynamicCalibrationAPI::DSDynamicCalibration::CAL_MODE_INTEL_SCALE_ONLY,
g_width, g_height, false);

// start capture with a callback function

// in the callback function, check if calibration is completed and add images to
Calibration Tool API for processing
while (!g_Dyncal.IsGridFull())
{
   g_Dyncal.AddImages(leftImage, rightImage, depthImage, timeStamp);
}

// when calibration is completed, stop capture

// get the updated calibration parameters and write to device
g_Dyncal->UpdateCalibrationTables ();
```

// release handles

## 3.4 Sample Dynamic Calibrator Application

A fully operational sample application Intel® RealSense™ Dynamic Calibrator is provided to show case usage of the Dynamic Calibration API. Source code is part of the installation package under the examples directory. The application supports

CalibrationToolAPI
|-------- 2.8.0.0
| |-------- bin
| |--------examples
| | |
| | |----Common
| | | |-------- Rs400Device.h
| | | |-------- Rs400Device.cpp
| | | |-------- ...
| | |
| | |----DynamicCalibrator
| | |-------- Main.cpp
| | |-------- MainView.cpp
| | |-------- ...
| | |-------- **makefile (Linux)**
| | |-------- **...**
| | |-------- **RS400DynamicCalibrationSample.sln (Windows)**

To compile the sample app

- On Windows, open RS400DynamicCalibrationSample.sln and build the project in **VisualStudio 2015 Update 3**.
- On Linux, use make.

The executables DynamicCalibrator and DynamicCalibratorCLI will be located under bin directory after the sample successful compiled.

# 4 Depth/RGB Custom Calibration R/W

In cases where user calibrates the D400 device with their own algorithm, Custom Calibration R/W interfaces in Dynamic Calibration API helps to write the data to the device and read it back later. This is useful if user determines an optimized calibration algorithm that would better fit their needs than the Intel calibration algorithm can provide.

In custom calibration mode, the user is responsible for the calibration algorithm and come up with the calibration parameters for the device, the library will provide interfaces to read/write the calibration parameters to the device. The API handles all internal data formats and device operations.

Please refer to the “Intel® RealSense™ Product Family D400 Series Custom Calibration” white paper for more details.

## 4.1 Interfaces

Dynamic Calibration API provides interfaces to facilitate the depth/RGB calibration data R/W for custom calibration.

### 4.1.1 Initialize

In custom mode, the Dynamic Calibration API library should be initialized with CalibrationMode = CAL_MODE_USER_CUSTOM. For example,

Initialize(prs2dev,
DynamicCalibrationAPI::DSDynamicCalibration::CAL_MODE_USER_CUSTOM)

where prs2dev is a pointer to librealsense rs2device handle

### 4.1.2 WriteCustomCalibrationParameters

Write calibration parameters to device

int WriteCustomCalibrationParameters(const int resolutionLeftRight[2], const double focalLengthLeft[2], const double principalPointLeft[2], const double distortionLeft[5], const double focalLengthRight[2], const double principalPointRight[2], const double distortionRight[5], const double rotationRight[9], const double translationRight[3], const bool hasRGB, const int resolutionRGB[2], const double focalLengthRGB[2], const double principalPointRGB[2], const double distortionRGB[5], const double rotationRGB[9], const double translationRGB[3]);

where
calibration parameters assumes that left camera is the reference camera and is located at world origin.

- resolutionLeftRight: The resolution of the left and right camera, specified as [width; height]
- focalLengthLeft: The focal length of the left camera, specified as [fx; fy] in pixels

- param principalPointLeft: The principal point of the left camera, specified as [px; py] in pixels
- distortionLeft: The distortion of the left camera, specified as Brown's distortion model [k1; k2; p1; p2; k3]
- focalLengthRight: The focal length of the right camera, specified as [fx; fy] in pixels
- principalPointRight: The principal point of the right camera, specified as [px; py] in pixels
- distortionRight: The distortion of the right camera, specified as Brown's distortion model [k1; k2; p1; p2; k3]
- rotationLeftRight: The rotation from the right camera coordinate system to the left camera coordinate system, specified as a 3x3 row-major rotation matrix
- translationLeftRight: The translation from the right camera coordinate system to the left camera coordinate system, specified as a 3x1 vector in millimeters
- hasRGB: Whether RGB camera calibration parameters are supplied
- resolutionRGB: The resolution of the RGB camera, specified as [width; height]
- focalLengthRGB: The focal length of the RGB camera, specified as [fx; fy] in pixels
- principalPointRGB: The principal point of the RGB camera, specified as [px; py] in pixels
- distortionRGB: The distortion of the RGB camera, specified as Brown's distortion model [k1; k2; p1; p2; k3]
- rotationLeftRGB: The rotation from the RGB camera coordinate system to the left camera coordinate system, specified as a 3x3 row-major rotation matrix
- translationLeftRGB: The translation from the RGB camera coordinate system to the left camera coordinate system, specified as a 3x1 vector in millimeters

returns When DC_SUCCESS then the tables were created successfully, otherwise, returns error code

* DC_ERROR_CUSTOM_INVALID_CAL_TABLE - one or more of the calibration tables are not valid
* DC_ERROR_CUSTOM_INVALID_LEFTRIGHT_RESOLUTION - left and right camera resolution is not supported
* DC_ERROR_CUSTOM_LEFT_INTRINSICS_UNREASONABLE - left camera intrinsics unreasonable
* DC_ERROR_CUSTOM_RIGHT_INTRINSICS_UNREASONABLE - right camera intrinsics unreasonabe

### 4.1.3 ReadCalibrationParameters

Read calibration parameters from device

int ReadCalibrationParameters(int resolutionLeftRight[2], double focalLengthLeft[2], double principalPointLeft[2], double distortionLeft[5], double focalLengthRight[2], double principalPointRight[2], double distortionRight[5], double rotationRight[9], double translationRight[3], bool& hasRGB, int resolutionRGB[2], double focalLengthRGB[2], double principalPointRGB[2], double distortionRGB[5], double rotationRGB[9], double translationRGB[3]);

where

- calibration parameters assumes that left camera is the reference camera and is located at world origin.
- resolutionLeftRight: The resolution of the left and right camera, specified as [width; height]
- focalLengthLeft: The focal length of the left camera, specified as [fx; fy] in pixels
- param principalPointLeft: The principal point of the left camera, specified as [px; py] in pixels
- distortionLeft: The distortion of the left camera, specified as Brown's distortion model [k1; k2; p1; p2; k3]

- focalLengthRight: The focal length of the right camera, specified as [fx; fy] in pixels
- principalPointRight: The principal point of the right camera, specified as [px; py] in pixels
- distortionRight: The distortion of the right camera, specified as Brown's distortion model [k1; k2; p1; p2; k3]
- rotationLeftRight: The rotation from the right camera coordinate system to the left camera coordinate system, specified as a 3x3 row-major rotation matrix
- translationLeftRight: The translation from the right camera coordinate system to the left camera coordinate system, specified as a 3x1 vector in millimeters
- hasRGB: Whether RGB camera calibration parameters are supplied
- resolutionRGB: The resolution of the RGB camera, specified as [width; height]
- focalLengthRGB: The focal length of the RGB camera, specified as [fx; fy] in pixels
- principalPointRGB: The principal point of the RGB camera, specified as [px; py] in pixels
- distortionRGB: The distortion of the RGB camera, specified as Brown's distortion model [k1; k2; p1; p2; k3]
- rotationLeftRGB: The rotation from the RGB camera coordinate system to the left camera coordinate system, specified as a 3x3 row-major rotation matrix
- translationLeftRGB: The translation from the RGB camera coordinate system to the left camera coordinate system, specified as a 3x1 vector in millimeters

returns When DC_SUCCESS then the tables were created successfully, otherwise, returns error code

- DC_ERROR_CUSTOM_INVALID_CAL_TABLE - one or more of the calibration tables are not valid
- DC_ERROR_CUSTOM_INVALID_PARAMS - one or more of the output calibration parameters are not valid

### 4.1.4 ResetDeviceCalibration

Reset active calibration data on device to desired settings

int ResetDeviceCalibration(CalibrationSettings setting = CalibrationSettings::CAL_SETTINGS_GOLD);

where

setting: currently only CalibrationSettings::CAL_SETTINGS_GOLD is valid, depends on the calibrations performed on the device, this setting came from either factory calibration or OEM calibration

return When DC_SUCCESS then calibration is reset to the desired setting

- DC_ERROR_INVALID_CAL_SETTINGS - invalid setting provided
- DC_ERROR_FAIL - failed to restore to the desired setting

### 4.1.5 ReadCalibrationRawData

Read calibration table raw data from device into buffer

int ReadCalibrationRawData(uint8_t* table, CalibrationTableType tableType);

Where

Table is user allocated buffer in type dependent size of DC_CALIB_COEFF_TABLE_SIZE, DC_CALIB_DEPTH_TABLE_SIZE, or DC_CALIB_RGB_TABLE_SIZE

tableType is calibration table type including CAL_TABLE_COEFF, CAL_TABLE_DEPTH, CAL_TABLE_RGB

and return DC_SUCCESS when raw data read successfully, otherwise, in error conditions

- DC_ERROR_NOT_INITIALIZED - library not initialized
- DC_ERROR_INVALID_BUFFER - invalid buffer
- DC_ERROR_INVALID_CAL_TABLE_TYPE - unsupported table type
- DC_ERROR_FAIL - failed to read the raw data from the device

### 4.1.6 WriteCalibrationRawData

Write calibration table raw data to device

int WriteCalibrationRawData(uint8_t* table, CalibrationTableType tableType);

Where

table: user allocated buffer in type dependent size of DC_CALIB_COEFF_TABLE_SIZE, DC_CALIB_DEPTH_TABLE_SIZE, or DC_CALIB_RGB_TABLE_SIZE.. The table content usually is saved from a device or generated from other Intel RealSense tools.

tableType: calibration table type including CAL_TABLE_COEFF, CAL_TABLE_DEPTH, CAL_TABLE_RGB

It returns DC_SUCCESS when raw data write successfully, otherwise, error conditions

- DC_ERROR_NOT_INITIALIZED - library not initialized
- DC_ERROR_INVALID_BUFFER - invalid buffer
- DC_ERROR_INVALID_CAL_TABLE_TYPE - unsupported table type
- DC_ERROR_FAIL - failed to write the raw data to the device

## 4.2 Read/Write through Custom Calibration R/W Interfaces

```
// Get libRealSense device handle for the D400 device
Rs400Device *g_rsDevice = new Rs400Device();
void *g_pDevice = g_rsDevice->GetDeviceHandle();

// Initialize the camera device
g_rsDevice->InitializeCamera();

// The expected calibration parameters for D400 series devices are defined as following:
//   resolutionLeftRight: The resolution of the left and right camera, specified as[width; height]
//   focalLengthLeft : The focal length of the left camera, specified as[fx; fy] in pixels
//   principalPointLeft : The principal point of the left camera, specified as[px; py] in pixels
```

```
//   distortionLeft : The distortion of the left camera, specified as Brown's distortion
model [k1; k2; p1; p2; k3]
//
//   focalLengthRight : The focal length of the right camera, specified as[fx; fy] in pixels
//   principalPointRight : The principal point of the right camera, specified as[px; py] in
pixels
//   distortionRight : The distortion of the right camera, specified as Brown's distortion
model [k1; k2; p1; p2; k3]
//   rotationLeftRight : The rotation from the right camera coordinate system to the left
camera coordinate system, specified as a 3x3 rotation matrix
 //   translationLeftRight : The translation from the right camera coordinate system to
the left camera coordinate system, specified as a 3x1 vector in milimeters
 //
 //   hasRGB : Whether RGB camera calibration parameters are supplied
 //   resolutionRGB : The resolution of the RGB camera, specified as[width; height]
 //   focalLengthRGB : The focal length of the RGB camera, specified as[fx; fy] in pixels
 //   principalPointRGB : The principal point of the RGB camera, specified as[px; py] in
pixels
 //   distortionRGB : The distortion of the RGB camera, specified as Brown's distortion
model [k1; k2; p1; p2; k3]
//   rotationLeftRGB : The rotation from the RGB camera coordinate system to the left
camera coordinate system, specified as a 3x3 rotation matrix
//   translationLeftRGB : The translation from the RGB camera coordinate system to the
left camera coordinate system, specified as a 3x1 vector in milimeters
//
   bool hasRGB;
   int resolutionLeftRight[2], resolutionRGB[2];
   double focalLengthLeft[2], focalLengthRight[2], focalLengthRGB[2];
   double principalPointLeft[2], principalPointRight[2], principalPointRGB[2];
   double distortionLeft[5], distortionRight[5], distortionRGB[5];
   double rotationLeftRight[9], rotationLeftRGB[9];
   double translationLeftRight[3], translationLeftRGB[3];

// Perform calibration with user custom algo here including
//   calibration stream setup and frame capture
//   calibration computation with custom algo
//   obtain optimized calibration parameters
//
// ***** A LOT OF YOUR CODE HERE for custom calibration *****
//
// An example of the parameter from a D400 device
//    resolutionLeftRight: 1920 1080
//
//    FocalLengthLeft :   1365.328979 1372.806641
//    PrincipalPointLeft : 959.393311 536.781494
```

```
//   DistortionLeft :      0.130178 - 0.393367 - 0.000580 0.001172 0.326398
//
//   FocalLengthRight :  1371.838989 1379.178101
//   PrincipalPointRight: 961.465759 540.101624
//   DistortionRight :     0.121403 - 0.385616 - 0.001131 - 0.000172 0.325395
//
//   RotationLeftRight : 0.999980 0.000408 - 0.006359
//                     - 0.000443 0.999985 - 0.005404
//                       0.006357 0.005407 0.999965
//   TranslationLeftRight : -55.109779 - 0.111518 - 0.114459
//   HasRGB : 0
//
// Another example of the parameters from a D435 device
//
//   resolutionLeftRight : 1280 800
//
//   FocalLengthLeft : 639.322205 637.623474
//   PrincipalPointLeft : 645.968262 399.232727
//   DistortionLeft : -0.057135 0.067378 0.000959 - 0.000607 - 0.021941
//
//   FocalLengthRight : 640.831848 639.201416
//   PrincipalPointRight : 641.968384 405.641235
//   DistortionRight : -0.056585 0.065952 0.000865 - 0.000556 - 0.021422
//
//   RotationLeftRight : 0.999997 - 0.001835 - 0.001385
//                       0.001835 0.999998 - 0.000234
//                       0.001385 0.000232 0.999999
//   TranslationLeftRight : -50.030815 - 0.005517 0.060515
//
//   HasRGB : 1
//   resolutionRGB : 1920 1080
//   FocalLengthColor : 1376.079590 1375.954102
//   PrincipalPointColor : 947.814758 543.885315
//   DistortionColor : 0.000000 0.000000 0.000000 0.000000 0.000000
//   RotationLeftColor : 0.999981 - 0.003833 0.004792
//                       0.003843 0.999990 - 0.002125
//                     - 0.004784 0.002143 0.999986
//  TranslationLeftColor : 14.624806 0.352931 0.213506
//
*************************************************************************************************
****


// Initialize Calibration Tool API to custom calibration mode
```

```
DynamicCalibrationAPI::DSDynamicCalibration *m_dcApi = new
DSDynamicCalibration();
int ret = m_dcApi->Initialize(g_pDevice,
DynamicCalibrationAPI::DSDynamicCalibration::CAL_MODE_USER_CUSTOM);

int status = DC_SUCCESS;

//
// Write the optimized calibration parameters to device. This will alter the calibration on
your device, please uncomment the following code only when necessary./
status = m_dcApi->WriteCustomCalibrationParameters(resolutionLeftRight,
focalLengthLeft, principalPointLeft, distortionLeft, focalLengthRight,
principalPointRight, distortionRight, rotationLeftRight, translationLeftRight, hasRGB,
resolutionRGB, focalLengthRGB, principalPointRGB, distortionRGB, rotationLeftRGB,
translationLeftRGB);

// Read back from device to confirm
status = m_dcApi->ReadCalibrationParameters(resolutionLeftRight, focalLengthLeft,
principalPointLeft, distortionLeft, focalLengthRight, principalPointRight, distortionRight,
rotationLeftRight, translationLeftRight, hasRGB, resolutionRGB, focalLengthRGB,
principalPointRGB, distortionRGB, rotationLeftRGB, translationLeftRGB);

// verify parameters
...

// Release device handle
delete g_rsDevice;
return EXIT_SUCCESS;
```

## 4.3 CustomRW for Custom Calibration Parameters Read/Write

A simple example CustomRW is provided to show case support calibration with user custom algo and usage of the calibration parameters read/write interfaces. Source code is part of the installation package under the examples directory.

```
CalibrationToolAPI
        |-------- 2.6.8.0
        |       |-------- bin
        |       |--------examples
        |       |       |
        |       |       |----Common
        |       |       |       |-------- Rs400Device.h
        |       |       |       |-------- Rs400Device.cpp
```

```
|       |       |       |-------- ...
|       |       |
|       |       |----CustomRW
|       |               |-------- Main.cpp
|       |               |-------- ...
|       |               |-------- makefile (Linux)
|       |               |-------- ...
|       |               |-------- CustomRW.sln (Windows)
|       |               |-------- CalibrationParameters.xml
```

To compile the example app

- On Windows, open CustomRW.sln and build the project in **VisualStudio 2015 Update 3**.
- On Linux, use make.

The executables CustomRW will be located under bin directory after the example successful compiled. CustomRW supports two command line options: -r and –w.

The –r option is used to read the calibration parameters from the device and dump to the screen console. –w is used to write the user calibration parameters from a XML file to the device.

For example,

*C:\CalibrationToolAPI\2.6.8.0\bin>Intel.Realsense.CustomRW.exe*
*CustomRW for Intel RealSense D400, Version: 2.6.8.0*

*-help*
*-?*
*display list of command line options*

*-list*
*display list of connected cameras*

*-version*
*-v*
*show dynamic calibration version info*

*-sn <device serial number>*
*-sn <device serial number>*

*if multiple cameras connected to the current system, choose one of the cameras to modify by specifying its serial number*

*-g*
*Reset calibration on device to default gold factory settings*

*-r*
*Read calibration data*

*-w*
*Write calibration data*

*-raw <19, 1f, or 20>*
*-raw <19, 1f, or 20>*
*dump calibration table raw binary content*

*-file <file name>*
*-f <file name>*
*dump calibration data to the specified file*

*-fe*
*fisheye custom data (limited, available ONLY on selected device SKU)*

*Usages:*
*Example #1: reset device calibration to gold settings*
*Intel.Realsense.CustomRW.exe -g*

*Example #2: read device calibration and display to terminal in xml format*
*Intel.Realsense.CustomRW.exe -r*

*Example #3: read device calibration into a text file in xml format*
*Intel.Realsense.CustomRW.exe -r -f mydevice.xml*

*Example #4: write custom calibration from a text file in xml format into the device*
*Intel.Realsense.CustomRW.exe -w -f mycustomdata.xml*

*Example #5: read raw calibration table from device and display to the terminal, supported table id 19, 1F, and 20, for example,*
*Intel.Realsense.CustomRW.exe -r -raw 19*

*Example #6: read raw calibration table from device and save into a binary file, supported table id 19, 1F, and 20, for example,*
*Intel.Realsense.CustomRW.exe -r -raw 19 -f mytable-19.bin*

*Example #7: write raw calibration table from binary file into the device, supported table id 19, 1F, and 20, for example,*
*Intel.Realsense.CustomRW.exe -w -raw 19 -f mytable-19.bin*

*Example #8: read custom fisheye data from device and display to the terminal*
*Intel.Realsense.CustomRW.exe -r -fe*

*Example #9: read custom fisheye data from device and save into a binary file*
*Intel.Realsense.CustomRW.exe -r -fe -f myfe.bin*

*Example #10: write custom fisheye data from binary file into the device*
*Intel.Realsense.CustomRW.exe -w -fe -f myfe.bin*

*Example #11: read custom motion module data from device and save into a binary file*

*bin\CustomRW.exe -r -mm -f myimu.bin*

*Example #12: write custom motion module data from binary file into the device*

*bin\CustomRW.exe -w -mm -f myimu.bin*

*C:\CalibrationToolAPI\2.6.7.0\bin>*

*The* -w option will alter calibration data on your device. If incorrect parameters are written, your device accuracy may be reduced or even malfunction. So please do so only when you really need to change calibration on your device.

An example of the calibration parameter XML file **CalibrationParameters.xml** is supplied in the CustomRW folder. This file is created for a D415 device. The parameters are described in section 2.1 Dynamic Calibration Parameters and also in documentation of the WriteCustomCalibrationParameters and ReadCalibrationParameters APIs in section 9.22 and 9.23. User will need to create the calibration parameter XML file with custom data for their device.

```
<?xml version="1.0"?>
<Config>
  <param name = "ResolutionLeftRight">
    <value>1920</value>
    <value>1080</value>
  </param>
  <param name = "FocalLengthLeft">
    <value>1378.69</value>
    <value>1379.13</value>
  </param>
  <param name = "PrincipalPointLeft">
    <value>965.562</value>
    <value>542.603</value>
  </param>
  <param name = "DistortionLeft">
    <value>0.09689</value>
    <value>-0.0235634</value>
    <value>-5.05513e-05</value>
    <value>0.000105855</value>
    <value>-0.640377</value>
  </param>
  <param name = "FocalLengthRight">
    <value>1389.55</value>
    <value>1390.1</value>
  </param>
  <param name = "PrincipalPointRight">
    <value>957.402</value>
    <value>553.108</value>
  </param>
  <param name = "DistortionRight">
    <value>0.123289</value>
    <value>-0.290911</value>
    <value>0.000149083</value>
    <value>0.000723527</value>
    <value>0.0992178</value>
  </param>
  <param name = "RotationLeftRight">
    <value>0.999983</value>
    <value>0.00150008</value>
    <value>-0.00558928</value>
    <value>-0.00150249</value>
```

```
    <value>0.999999</value>
    <value>-0.000428313</value>
    <value>0.00558863</value>
    <value>0.000436704</value>
    <value>0.999984</value>
</param>
<param name = "TranslationLeftRight">
    <value>-55.3888</value>
    <value>0.0448052</value>
    <value>1.52189</value>
</param>
<param name = "HasRGB">
    <value>1</value>
</param>
<param name = "ResolutionRGB">
    <value>1920</value>
    <value>1080</value>
</param>
<param name = "FocalLengthRGB">
    <value>1376.09</value>
    <value>1376.4</value>
</param>
<param name = "PrincipalPointRGB">
    <value>947.958</value>
    <value>531.068</value>
</param>
<param name = "DistortionRGB">
    <value>0.109862</value>
    <value>-0.122685</value>
    <value>-0.000453445</value>
    <value>-8.61456e-05</value>
    <value>-0.396171</value>
</param>
<param name = "RotationLeftRGB">
    <value>0.999994</value>
    <value>0.00346622</value>
    <value>0.000876088</value>
    <value>-0.00346642</value>
    <value>0.999994</value>
    <value>0.000222113</value>
    <value>-0.000875313</value>
```

```
      <value>-0.000225148</value>
      <value>1</value>
    </param>
    <param name = "TranslationLeftRGB">
      <value>14.6987</value>
      <value>-0.0644999</value>
      <value>0.448085</value>
    </param>
  </Config>
```

### 4.3.1 CustomCalibration Example for Depth/RGB Custom Calibration

A comprehensive example CustomCalibration is provided to show case support calibration with user custom algo and device setup for calibration. Source code is part of the installation package under the examples directory.

```
CalibrationToolAPI
        |-------- 2.6.8.0
        |         |-------- bin
        |         |--------examples
        |         |         |
        |         |         |----Common
        |         |         |       |-------- ...
        |         |         |
        |         |         |----CustomCalibration
        |         |                 |-------- ...
        |         |                 |-------- CustomCalibrationTest.sln (Windows)
```

This example requires installation of OpenCV 3.3.1. It takes some setup to compile the example app.

Please refer to the white paper "**Intel® RealSense™ Product Family D400 Series Custom Calibration**" for details.

# 5 Fisheye Custom Calibration R/W

**Important Notes**:

**Fisheye custom Calibration R/W is only supported on limited devices with a fisheye, i.e., Intel RealSense D430 with Tracking Module**

On devices with Fisheye, the Fisheye itself is not initially calibrated, user will need to calibrate it with their own methods and then write the calibration data to the device and read it back for use later.

There are two ways to read/write into this fisheye custom calibration data storage on the device:

1) Read/write through the Intel.Realsense.CustomRW Tool
2) User develop their own tool through the Fisheye Custom Calibration R/W interfaces in the Dynamic Calibration API

## 5.1 Handling Fisheye Calibration Data with CustomRW Tool

The Intel.Realsense.CustomRW Tool provides users the ability to read/write custom data to the device. For fisheye data, it takes a binary input file and write it to the device. It also can read it back and display on console or save into a binary file. The file size should be 136 bytes in exact, same as the fisheye custom data storage on the device.

### 5.1.1 Dependency

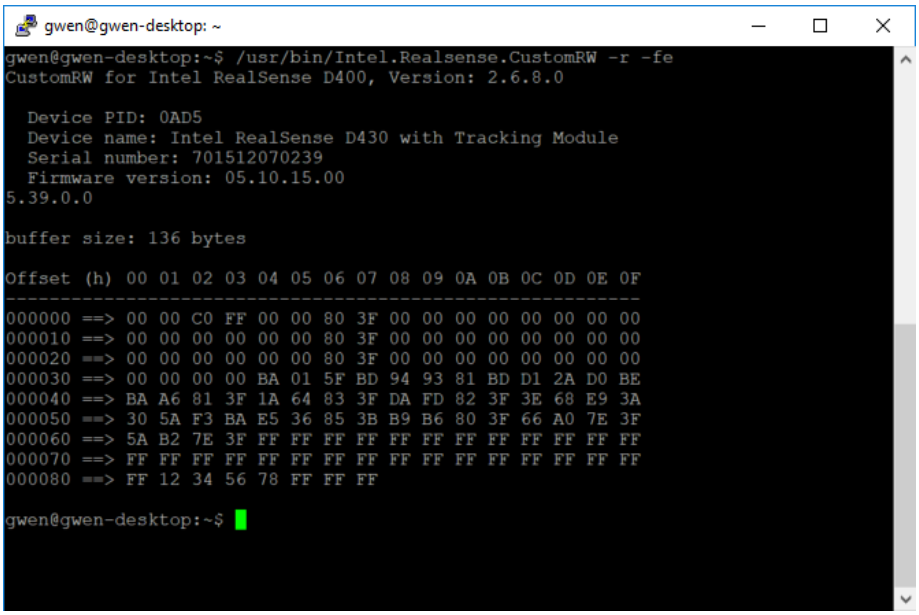
The Intel.Realsense.CustomRW Tool is statically linked. No dependency other than the C++ runtime.

### 5.1.2 Read and Write Custom Fisheye Data

The -fe option supports read/write the custom Fisheye data on devices of limited SKU. This option should only be used on those devices.

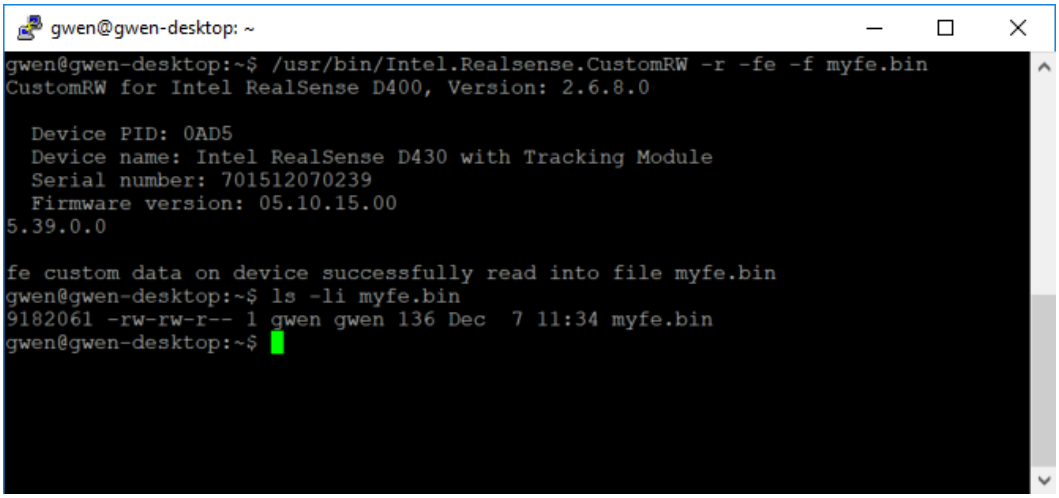
**Example**: read custom Fisheye data from device and display to the terminal
Intel.Realsense.CustomRW.exe -r -fe

**Figure 5-1 Read Fisheye Custom Data on Selected Device and Display on Screen**




**Example**: read custom Fisheye data from device and save into a binary file
Intel.Realsense.CustomRW.exe -r -fe -f myfe.bin

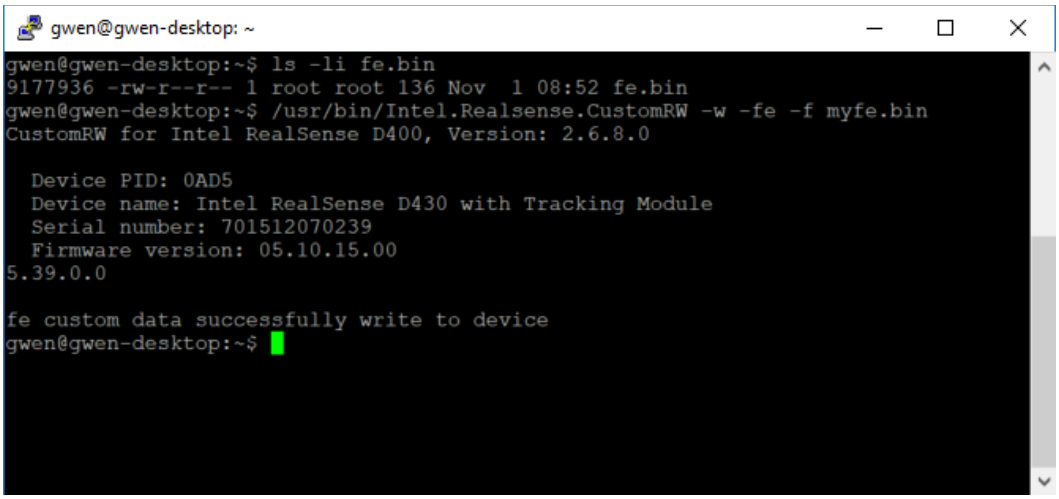
**Figure 5-2 Read Fisheye Custom Data into a Binary File**



**Example**: write a Fisheye custom calibration binary data file into the device When user calibrate the device, the results should be saved in a binary file of 136 bytes in size, then use the tool to write it into the device, or if user saved a backup from the device earlier, the same write option can restore the backup to the device.

Intel.Realsense.CustomRW.exe -w -fe -f myfe.bin

**Figure 5-3 Write Fisheye Custom Data from a Binary File to the Device**




## 5.2 Read and Write through API

The API read the custom data from device and pass it over to the user through a user provided buffer. Similarly, user pass a buffer with the custom data to the API and write it to the device. The size of the buffer should be 136 bytes.

### 5.2.1 Initialize

In custom mode, the Dynamic Calibration API library should be initialized with CalibrationMode = CAL_MODE_USER_CUSTOM.

For example,

Initialize(prs2dev,
DynamicCalibrationAPI::DSDynamicCalibration::CAL_MODE_USER_CUSTOM)

where prs2dev is a pointer to librealsense rs2::device handle.

**Important Notes:**

Before initializing and using the API, the device which rs2::device points to should be running in advanced mode. The example in the next section shows the details.

### 5.2.2 ReadFECustomData

Read fisheye custom data from device into buffer. User is responsible for allocating and deallocating the buffer memory.

int ReadFECustomData(uint8_t* buffer);

where

buffer: user allocated buffer in size of DC_MM_FE_CUSTOM_DATA_SIZE (136 bytes)

and returns DC_SUCCESS when data read successfully, otherwise, errors

- DC_ERROR_NOT_INITIALIZED - library not initialized
- DC_ERROR_INVALID_BUFFER - invalid buffer
- DC_ERROR_FAIL - failed to read data from the device

### 5.2.3 WriteFECustomData

Write FE custom data buffer to device. User is responsible for allocating and deallocating the buffer memory.

int WriteFECustomData(uint8_t* buffer);

where

buffer: user allocated buffer in size of DC_MM_FE_CUSTOM_DATA_SIZE (136 bytes) that contains custom data to be written to device

and returns DC_SUCCESS then data written successfully, otherwise, returns errors

- DC_ERROR_NOT_INITIALIZED - library not initialized
- DC_ERROR_INVALID_BUFFER - invalid buffer
- DC_ERROR_FAIL - failed to write data to device

### 5.2.4 Example

A simple example is provided as part of the API package under the examples/simple-fe folder.

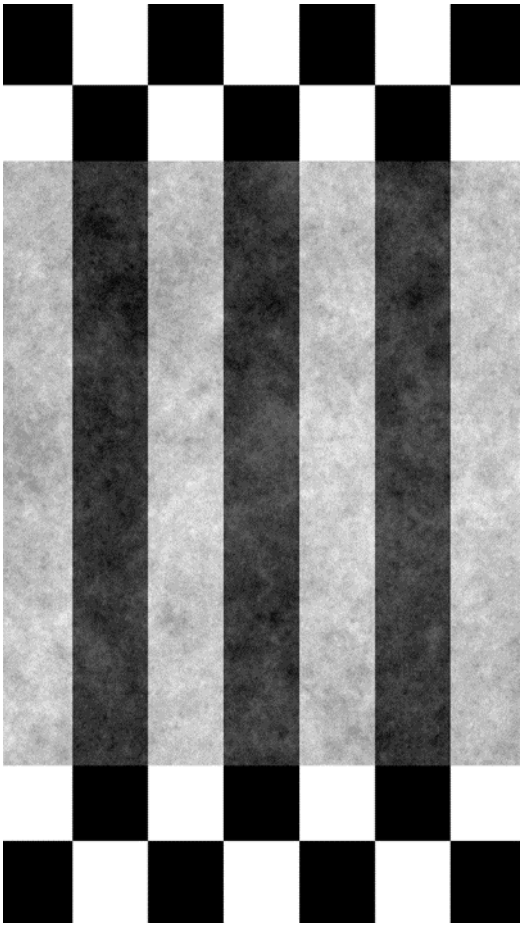
```
  // rs context
  rs2::context *ctx = new context();

  // query devices on host
  auto devices = ctx->query_devices();
  rs2::device mydev = devices[0];

  // to use the custom calibration read/write api, the device should be in advanced
mode
  if (mydev.is<rs400::advanced_mode>())
  {
    rs400::advanced_mode advanced = mydev.as<rs400::advanced_mode>();
    if (!advanced.is_enabled())
    {
      advanced.toggle_advanced_mode(true);
    }
  }

  // initialize custom calibration R/W API
  MMCalibrationRWAPI *m_dcApi = new MMCalibrationRWAPI();

  // mydev should be a pointer to rs2::device from librealsense
  int ret = m_dcApi->Initialize(&mydev);

  uint8_t mmCustom[DC_MM_CUSTOM_DATA_SIZE];

  // the custom data area has DC_MM_CUSTOM_DATA_SIZE (504 bytes) of storage
space. User can
  // define their data own format and write to the device and retrieve it later from
  // the device

  // on a new device, the storage is not initialzed, so should write data before perform
```

```
 // read. reading before writing any data will fail.
// write some data as example
 memset((void*)&mmCustom, 0xFF, sizeof(mmCustom));
 mmCustom[160] = 0xB;
 mmCustom[180] = 0x8;

 ret = m_dcApi->WriteMMCustomData((uint8_t*) &mmCustom);
 ....

 // read the custom data back from the device
 ret = m_dcApi->ReadMMCustomData(mmCustom);

// interpret and use the calibration data in app
```

# 6 Phone Target App Integration

## 6.1 Target

Targeted calibration requires a fixed width (10 mm) bar target, either displayed on phone through a phone app or printed on paper. The target image can be generated through a provided algorithm in Matlab and C functions. When displayed on phone, display configuration should be obtained and so the algorithm can generate precise target image to fit to the phone display.



## 6.2 Target Generation Algorithm and API

A dynamic composition algorithm is provided to generate the target image at runtime. The algorithm is in C/C++ with API for the phone app to call.

```
/*
 * File: createPhonePatternImageAPI.h
 *
 */

#ifndef CREATEPHONEPATTERNIMAGEAPI_H
#define CREATEPHONEPATTERNIMAGEAPI_H

/* Include Files */
#include "rt_nonfinite.h"
#include "rtwtypes.h"
#include "createPhonePatternImage_types.h"
#include "rt_nonfinite.h"
#include "createPhonePatternImage.h"
#include "createPhonePatternImage_terminate.h"
#include "createPhonePatternImage_emxAPI.h"
#include "createPhonePatternImage_initialize.h"

typedef struct _CPPI_SIZE
{
        int cx;
        int cy;
} CPPI_SIZE;

typedef struct _CPPI_SIZE_DOUBLE
{
        double cx;
        double cy;
} CPPI_SIZE_DOUBLE;


/* load and initialize CPPI (create Phone Pattern Image) */
void createPhonePatternImageInit(void);
```

```
/* uninitiate CPPI */
void createPhonePatternImageUnInit(void);

/**
* barWidth: bar width in mm
* phoneRes: phone resolution in pixels
* screenDPI: screen DPI
* hybrid: 1 to blend the texture backgroud
* screenROI: screen ROI in mm
* image: output image in gray-scale (8-bit)
*/
void createPhonePatternImageRun(double barWidth, CPPI_SIZE phoneRes,
        CPPI_SIZE screenDPI, int hybrid, CPPI_SIZE_DOUBLE screenROI,
unsigned char *image);


#endif
```

A sample code along with the API and algorithm code is provided as part of the dynamic calibration release.

## 6.3 Phone App

Intel provides sample phone app on Apple App Store for iPhones and Google Play for Android phones. Please refer to Appendix **Intel® RealSense™ Dynamic Calibration Target Setup** and **Intel® RealSense™ Dynamic Target Tool Phone App Technical Specification** for phone target app technical specification. Source code for the sample phone apps are also provided for integration into OEM apps.

## 6.4 Devices Validated

The sample phone app is validated on a limited number of devices, both IOS and AOS. Please refer to appendix ***Validated Devices for Intel® RealSense™ Dynamic Target Tool Phone App*** for details.

# Appendix

# 7 Dynamic Calibration API Definitions

Calibration APIs provide interfaces for application to access dynamic calibrator for real time re-calibration and also provides APIs to read/write/restore calibration data on the device.

The API is defined in DSDynamicCalibration.h.

For Linux and Windows x64 environment, DSDynamicCalibration Class is defined under namespace DynamicCalibrationAPI.

Calibration Tool API support three calibration modes:

- Intel Targetless Calibration
- Intel Targeted Calibration
- User Custom

Both Targetless and Targeted uses Intel algorithm that embedded in the API. Custom mode enables advanced user who prefer to use their own calibration algorithm and write/read the optimized calibration parameters to the device.

```
/** Calibration Modes */
enum CalibrationMode
{
        /** Intel target-less calibration (target-less depth rectification only)*/
        CAL_MODE_INTEL_TARGETLESS = 0,
        /** Intel targeted calibration (targeted depth rectification and targeted depth
scale)*/
        CAL_MODE_INTEL_TARGETED = 1,
        /** Intel targeted depth scale calibration only */
        CAL_MODE_INTEL_TARGETED_SCALE_ONLY = 3,
        /** Intel targeted RGB calibration */
        CAL_MODE_INTEL_RGB_CALIB = 4,
        /** User custom algorithm */
        CAL_MODE_USER_CUSTOM = 99};

/** Calibration Settings */

enum CalibrationSettings

{
```

```
        /** gold settings */

        CAL_SETTINGS_GOLD = 0

};
```

Grid level enumeration quantifies the amount of the features in each cell of the grid.

```
enum GridLevel
{
   /* Low amount of features in the cell */
   GRID_LEVEL_LOW = 0,
   /* Medium amount of features in the cell */
   GRID_LEVEL_MEDIUM = 1,
   /* High amount of features in the cell */
   GRID_LEVEL_HIGH = 2,
   /* Very High amount of features in the cell */
   GRID_LEVEL_VERY_HIGH = 3,
   /* Not tracked */
  GRID_LEVEL_NOT_TRACKED = 255
};
```

Grid fill enumeration quantifies how the cells are filled with features in the next frame.

```
enum GridFill
{
   /* Normal - features are subject to aging if they are in the buffer for too long */
   GRID_FILL_NORMAL = 0,
   /* Add only - will retain any existing features but keeps adding new features until the
buffer is full */
   GRID_FILL_ADD_ONLY = 1,
   /* Cell full - will retain any existing features and not add any new features and the
feature detection isn't run in the cell */
   GRID_FILL_CELL_FULL = 2
};
```

Grid features enumeration quantifies the amount of the features in the last frame.

```
enum GridFeatures
{
/* Level 0 - no to little amount features */
```

```
GRID_FEATURES_L0 = 0,
   /* Level 1 - moderate amount of features */
   GRID_FEATURES_L1 = 1,
   /* Level 2 - good amount of features */
   GRID_FEATURES_L2 = 2
};
```

## 7.1 Attributes

```
 // target stripe orientation ir aligned with device baseline direction
 bool m_target_aligned;

// force targeted scale calibration and rgb calibration to capture more or less number of target
images than default
int m_numImages;

// force target-less calibration to ignore the blocks on the borders of FOV
bool m_ignore_borders;
```

## 7.2 Initialize

**Initialize** method initializes dynamic calibrator. It needs to be called at the very begin of the process.

For target-less calibration, 1280x720 resolution is supported for all devices, and additional native resolution 1280x800 is supported for modules with wide angle lens, for example, D430 and D420 modules.

For targeted calibration, only 1280x720 resolution is supported across all devices.

The function will throw a runtime error if the resolution is different or the calibration parameters are not valid.

```
int Initialize(void *rs400Dev, CalibrationMode mode, int width, int height, bool
active = false);
```

where
**rs400Dev:** opaque device handle, currently only librealsense rs2device handle is supported on both Windows and Linux

**mode**: calibration mode, CAL_MODE_INTEL_TARGETLESS, CAL_MODE_INTEL_TARGETED, CAL_MODE_INTEL_TARGETED_SCALE_ONLY, CAL_MODE_INTEL_RGB_CALIB or CAL_MODE_USER_CUSTOM
**width**: The width of the images that will be used for dynamic calibration
**height**: The height of the images that will be used for dynamic calibration
**active**: true for laser projector is active, false for not active
**returns error code**:

- DC_SUCCESS - no issue
- DC_ERROR_INVALID_PARAMETER - invalid parameter passed in
- DC_ERROR_RESOLUTION_NOT_SUPPORTED_V2 - if 1280x800 resolution is requested for devices with old version 2 coefficient table, this is not supported
- DC_ERROR_TABLE_NOT_SUPPORTED - calibration coefficient table on device is not supported
- DC_ERROR_TABLE_NOT_VALID_RESOLUTION – calibration table resolution width or height not valid

## 7.3 InitializeRgbCalibrator

Initialize RGB calibrator. It needs to be called after scale (targeted) calibration is completed, only 1280x720 resolution is supported across all devices. The function will throw a runtime error if the resolution is different or the calibration parameters are not valid.
int InitializeRgbCalibrator(void *rs400Dev, int width = 1280, int height = 720)

where
rs400Dev: opaque device handle, currently only librealsense handle is supported on both Windows and Linux
width: The width of the images that will be used for dynamic calibration
height: The height of the images that will be used for dynamic calibration
returns DC_SUCCESS if no issue, error code otherwise:
DC_ERROR_INVALID_PARAMETER - invalid parameter passed in
DC_ERROR_RESOLUTION_NOT_SUPPORTED_V2 - if 1280x800 resolution is requested for devices with old version 2 coefficient table, this is not supported
DC_ERROR_TABLE_NOT_SUPPORTED - calibration coefficient table on device is not supported
DC_ERROR_TABLE_NOT_VALID_RESOLUTION – calibration table resolution width or height not valid

## 7.4 AddImages

**AddImages** method adds a pair of left and right images to dynamic calibrator.

Add a pair of images to be used by dynamic calibration. The images must have the width and height as specified during initialization and must be in 8-bit grayscale format.

int **AddImages**(const uint8_t * leftImage, const uint8_t * rightImage, const uint8_t * depthImage, const uint64_t timeStamp);

where
**leftImage**: Pointer to captured left image buffer in byte format.
**rightImage**: Pointer to captured right image buffer in byte format.
**depthImage**: Pointer to captured depth image buffer in byte format.
**timestamp**: timestamp on the image pair in milliseconds.
**Returns**

Target-less calibration:

- DC_SUCCESS if image added ok, otherwise returns error code.
- DC_ERROR_RECT_INVALID_IMAGES
- DC_ERROR_RECT_INVALID_GRID_FILL
- DC_ERROR_RECT_TOO_SIMILAR
- DC_ERROR_RECT_TOO_MUCH_FEATURES
- DC_ERROR_RECT_NO_FEATURES
- DC_ERROR_RECT_GRID_FULL
- DC_ERROR_UNKNOWN

Targeted Calibration:

DC_SUCCESS
DC_ERROR_RECT_INVALID_IMAGES
DC_ERROR_RECT_TOO_SIMILAR
DC_ERROR_RECT_TARGET_NOT_FOUND_LEFT
DC_ERROR_RECT_TARGET_NOT_FOUND_RIGHT
DC_ERROR_RECT_TARGET_NOT_FOUND_BOTH
DC_ERROR_SCALE_DEPTH_NOT_CONSISTENT
DC_ERROR_TARGET_UNSTABLE
DC_ERROR_TARGET_TOO_CLOSE

DC_ERROR_TARGET_TOO_FAR
DC_ERROR_SCALE_ALREADY_CAPTURED
DC_ERROR_SCALE_DEPTH_TOO_SPARSE
DC_ERROR_SCALE_DEPTH_NOT_A_PLANE
DC_ERROR_SCALE_TARGET_TILT_ANGLE_BIG
DC_ERROR_SCALE_FAILED_COMPUTE
DC_ERROR_PHASE_COMPLETED
DC_ERROR_UNKNOWN

## 7.5 GetLastPhoneROILeftCamera

(for targeted calibration only)
Get last position of target (printed or phone) in left image position is defined by a quadrilateral plane boundary with four corner points

int GetLastPhoneROILeftCamera(float ptr[PHONELOC_DIM])

where
ptr four corner points surrounding the target in the order
top left corner x and y
top right corner x and y
bottom right corner x and y
bottom left corner x and y
return error code indicate target location status

- DC_SUCCESS
- DC_ERROR_NOT_IMPLEMENTED – if called in target-less mode
- DC_ERROR_RECT_TARGET_NOT_FOUND_LEFT

## 7.6 GetLastPhoneROIRightCamera

(for targeted calibration only)

Get last position of target (printed or phone) in right image position is defined by a quadrilateral plane boundary with four corner points

int GetLastPhoneROIRightCamera(float ptr[PHONELOC_DIM])

where
ptr four corner points surrounding the target in the order

top left corner x and y
top right corner x and y
bottom right corner x and y
bottom left corner x and y

return error code indicate target location status

- DC_SUCCESS
- DC_ERROR_NOT_IMPLEMENTED – if called in target-less mode
- DC_ERROR_RECT_TARGET_NOT_FOUND_RIGHT

## 7.7 GetLastTargetDistance

Get target distance - the last distance of the target/phone from camera. The distance is returned in millimeters, but generally can be considered as inaccurate as the updated calibration isn't ready by this time. This function returns value only in the scale phase of the process. The distance of the phone target in mm or -1.0 if the phone wasn't detected yet or not in scale phase of the process.

float **GetLastTargetDistance**()

returns distance of the phone target in mm or -1.0 if the phone wasn't detected yet or not in scale phase of the process.

## 7.8 AccessGridFill

**AccessGridFill** method returns an array of grid quantifies how the cells are filled with features. This method only needs to be called once at the beginning and the pointer to GridFill array will be used to check for continuous update.

void **AccessGridFill**(GridFill *& grid, int & gridWidth, int & gridHeight);

**grid**: Pointer to GridFill array with size of gridWith * gridHeight in row major.
**grdWidth**: Grid width as output.
**grdHeight**: Grid height as output.

## 7.9 GetLastFrameFeaturesGrid

**GetLastFrameFeaturesGrid** method returns the current status of feature detection in the grid array of the last frame. The features are updated whenever new images are added. This method only needs to be called once at the beginning and the pointer to GridFeatures array will be used to check for continuous update.

void **GetLastFrameFeaturesGrid**(const GridFeatures *& grid, int & gridWidth, int & gridHeight);

**grid**: Pointer to GridFeatures array with size of gridWith * gridHeight in row major.
**grdWidth**: Grid width as output.
**grdHeight**: Grid height as output.

## 7.10 GetGridLevels

void **GetGridLevels**(const GridLevel *& grid, int & gridWidth, int & gridHeight);

**GetGridLevels** method return the current status of feature detection in each cell of the grid array. The features are updated whenever new images are added. This method only needs to be called once at the beginning and the pointer to GridLevel array will be used to check for continuous update.
**grid**: Pointer to GridLevel array with size of gridWith * gridHeight in row major.
**grdWidth**: Grid width as output.
**grdHeight**: Grid height as output.

## 7.11 GetGridLevelScore

**GetGridLevelScore** method evaluates the grid level and return a score how well the grid is filled.

float **GetGridLevelScore**();

Return: The score between 0.0f and 1.0f. 0.0f means empty grid. 1.0f means grid level is high for every cell.

## 7.12 IsGridFull

**IsGridFull** method indicates whether the dynamic calibration is completed.

bool **IsGridFull**()

returns

- False: Dynamic calibrator is still working on calibration.
- True: Dynamic calibration is completed.

## 7.13 IsOutOfCalibration

int **IsOutOfCalibration**(bool & outOfCalibration);

Whether the device is out of calibration. The function will fail to evaluate when DSDynamicCalibration::CanComputeCalibration returns false due to insufficient data. You should only use this function when DSDynamicCalibration::IsGridFull returns true when following the official process.

**outOfCalibration**: Returns whether is out of calibration or not

returns error code of the evaluation, DC_SUCCESS if successful, DC_ERROR_PREMATURE if called before scale calibration is completed in targeted calibration.

## 7.14 GetCalibrationError

(**target-less calibration only**)

bool **GetCalibrationError**(float & calibrationError);

Get the calibration error RMS (not normalized). This function is not part of the official process and it is not guaranteed that calibration calculated when DSDynamicCalibration::IsGridFull returns false will meet the specification.

**calibrationError**: Returns the calibration error

return: DC_SUCCESS if the evaluation was successful for target-less rectification. DC_ERROR_NOT_APPLICABLE for targeted calibration.

## 7.15 IsRectificationPhaseComplete

(**targeted calibration only**)

bool **IsRectificationPhaseComplete**();

Check if rectification phase is completed, return true if completed otherwise false.

## 7.16 IsScalePhaseComplete

(**targeted calibration only**)

bool **IsScalePhaseComplete**();

Check if scale calibration phase is completed, return true if completed otherwise false.

## 7.17 NumOfImagesCollected

Get number of image pairs collected in the current calibration phase. return number of image pairs (left/right for target-less calibration and left/right/depth for targeted calibration

int **NumOfImagesCollected**();

## 7.18 SetIntermediateRectificationCalibration

**(targeted calibration only)**

Apply the intermediate calibration results once the rectification phase is completed.

int SetIntermediateRectificationCalibration();

where it returns
DC_SUCCESS on success, error code on failure.
DC_ERROR_NOT_APPLICABLE when called in target-less calibration
DC_ERROR_FAIL if there is problem and failed

## 7.19 GoToScalePhase

(**targeted calibration only**)

Switch to scale calibration phase. This has to be called after the rectification phase is completed before starting scale calibration phase.

bool **GoToScalePhase**();

## 7.20 GetPhase

Get current calibration phase, return DC_TARGETED_RECTIFICATION_PHASE for rectification phase, DC_TARGETED_SCALE_PHASE for scale phase.

DC_PHASE **GetPhase**();

## 7.21 GetTargetedCalibrationCorrection

(**targeted calibration only**)

Returns the updated calibration in rotation modification around the x/y/z axes once the scale phase is complete, where

**rx** Returns rotation modification around the x-axis

**ry** Returns rotation modification around the y-axis

**rz** Returns rotation modification around the z-axis

and true on success, false on failure.

bool **GetTargetedCalibrationCorrection**(double& rx, double& ry, double& rz);

This API requires the scale phase is completed.

## 7.22 GetRGBCalibrationCorrection

(RGB calibration only) Returns the updated calibration in rotation modification around the x/y/z axes once the scale phase is complete. This API requires the scale phase is completed.

bool GetRGBCalibrationCorrection(double& rx, double& ry, double& rz, double& tx, double& ty, double& tz);

where
rx Returns rotation modification around the x-axis
ry Returns rotation modification around the y-axis
rz Returns rotation modification around the z-axis
tx Returns translation modification along the x-axis
ty Returns translation modification along the y-axis
tz Returns translation modification along the z-axis
return true on success, false on failure.

## 7.23 UpdateCalibrationTables

Get the updated calibration tables and write to device. For target-less, you should only use this function when DSDynamicCalibration::IsGridFull returns true. For targeted, only use this function after scale calibration is completed.

int UpdateCalibrationTables();

where

Return

- DC_SUCCESS on success

- DC_ERROR_FAIL on failure

## 7.24 GetVersion

Get version of dynamic calibration as a string

static const char * **GetVersion**()

where
return: Version of Dynamic Calibration API

# 8 Intel® RealSense™ Dynamic Target Tool Phone App Technical Specification

## 8.1 Introduction

Dynamic calibration is used to re-calibrate the Intel® RealSense™ Depth Camera. As part of the process, the algorithm requires a mixed bar and block image as calibration target. An efficient way to show such target is to display the target image on an IOS or Android phone screen through a phone app.

This document provides technical specification for dynamic calibration target phone app.

## 8.2 App Flow and Requirements

### 8.2.1 Overview

The figuration below shows the overall flow how the app will work as part of the dynamic calibration process.

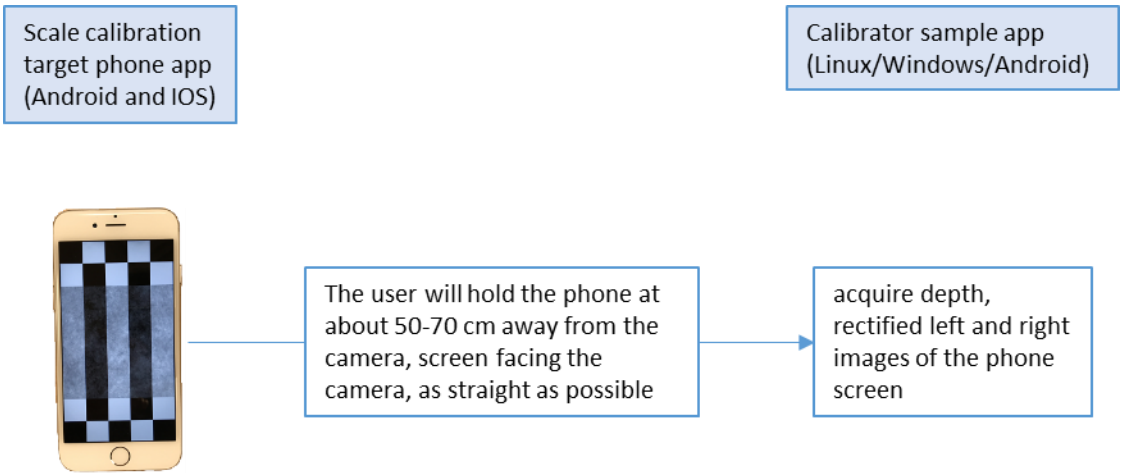



*Figure 8-1 dynamic calibration simplified flow*

1) The app detects phone screen configurations
2) A target image with multiple vertical bars and black and white blocks is dynamically generated on the fly at runtime (as first experiment in prototyping the app, pre-generated images will be pre-supplied by Intel). The image will be exact match to screen resolution and its content rendered in such that the bar widths are exactly 10 mm (except the most left and right bars on the edges) and the vertical distance between the top black blocks and bottom black blocks also meet certain requirement. An algorithm to generate the image is also supplied by Intel for dynamic composition after the prototyping with pre-generated images.
3) The app reads the target image.

The app display the target image on screen. The image resolution is exact match to the screen resolution, so no scaling is required, and the image will be displayed as is so that the physical width of each bar is equal to a pre-designated fixed width, in this case 10 mm. Since the width of the bar is fixed, the number of bars actually displayed is determined by the screen size.

1) The accuracy of its width is critical for the Intel dynamic calibration algorithm to be correctly operational.
2) The user holds the phone at about 50-100 cm away in front of the Intel Realsense Camera as straight as possible.
3) The Intel calibrator on the host acquires the phone target images and perform dynamic calibration on the Intel Realsense Camera.

### 8.2.2 Target Image

An example of the new target image is shown below. The actual image is device dependent (will be dynamically generated at runtime).

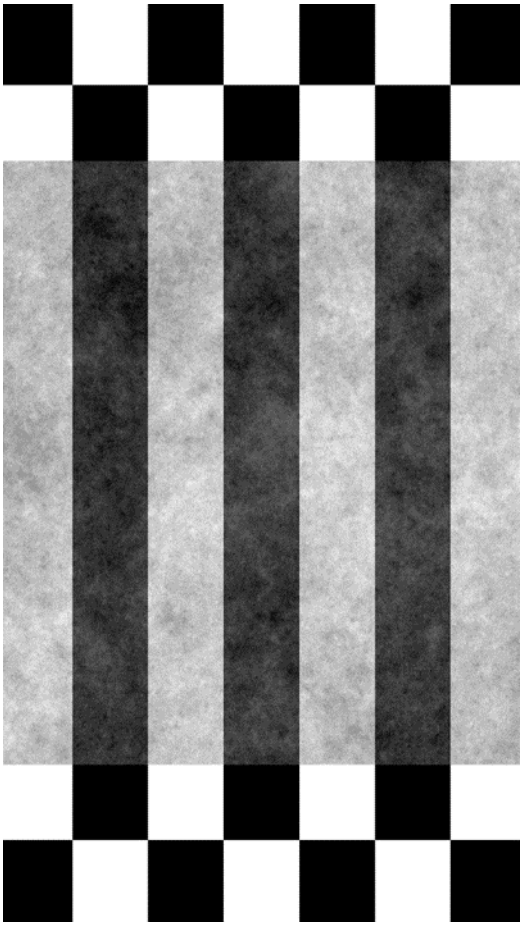


*Figure 8-2 target image*

To prototype the phone app and experiment possible issues on the various devices, a set of pre-generated images are supplied.

| Apple Device Displays | Resolution (pixel) | DPI (PPI) | Screen Physical Dimension (cm) | Devices with this display | Intel Pre-supplied target image | Number of black vertical bars |
|---|---|---|---|---|---|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | Width x Height | | | |
| 4” | 640 x 1136 | 326 | 4.99 x 8.85 | Iphone 5, 5s, 5c, SE | ios-40in.png | 2 |
| 4.7” | 750 x 1334 | 326 | 5.88 x 10.46 | Iphone 6, 6s, 7 | ios-47in.png | 2 |
| 5.5” | 1080 x 1920 | 401 | 6.84 x 12.16 | Iphone 6 plus, 6s plus, 7 plus | ios-55in.png | 3 |

### 8.2.3 Target Composition Algorithm API (Android Only)

On Android devices, displays with different resolutions and DPI are used by the various OEM phone vendors. The pixel density could also be different in horizontal and vertical directions. This makes hard to pre-generate target image and match with device display.

A dynamic composition algorithm is better utilized in this case to generate the target image at runtime. The algorithm is in C/C++ with API for the phone app to call. On Android, this can be called through JNI.

```
/*
 * File: createPhonePatternImageAPI.h
 *
 */

#ifndef CREATEPHONEPATTERNIMAGEAPI_H
#define CREATEPHONEPATTERNIMAGEAPI_H

/* Include Files */
#include "rt_nonfinite.h"
#include "rtwtypes.h"
#include "createPhonePatternImage_types.h"
#include "rt_nonfinite.h"
#include "createPhonePatternImage.h"
#include "createPhonePatternImage_terminate.h"
#include "createPhonePatternImage_emxAPI.h"
#include "createPhonePatternImage_initialize.h"

typedef struct _CPPI_SIZE
{
```

```
	int cx;
	int cy;
} CPPI_SIZE;

typedef struct _CPPI_SIZE_DOUBLE
{
	double cx;
	double cy;
} CPPI_SIZE_DOUBLE;


/* load and initialize CPPI (create Phone Pattern Image) */
void createPhonePatternImageInit(void);


/* uninitiate CPPI */
void createPhonePatternImageUnInit(void);

/**
* barWidth: bar width in mm
* phoneRes: phone resolution in pixels
* screenDPI: screen DPI
* hybrid:
* screenROI: screen ROI in mm
* image: output image in gray-scale (8-bit)
*/
void createPhonePatternImageRun(double barWidth, CPPI_SIZE phoneRes,
	CPPI_SIZE screenDPI, int hybrid, CPPI_SIZE_DOUBLE screenROI, unsigned char *image);


#endif
```

A sample code along with the API and algorithm code will be provided by Intel.

### 8.2.4 Target Image Display

The image is stored or generated at runtime on the phone device as part of the app. The app reads the image and displays it on screen. The image has exact same resolution as the device so no scaling is required and the width for each of the bars in the image measures same as a pre-set fixed width. The width is currently determined to be 10 mm. This pre-determined width might be adjusted later after actual accuracy of the calibration process is measured and tuned. The app needs to take this into account so that it will cause main issue when there is a need to change it in later development process.

Depends on the size of the screen, the image may display more or less number of bars and the image is symmetric.

An example (ios-47.png) with iPhone 6 is shown below.

*Figure 8-3 target displayed on iPhone 6*

### 8.2.5 Target Image Accuracy Check

Target precision is critical to calibration accuracy. After the target image is successfully displayed on phone screen, please verify its physical dimensions of key features with a ruler, as shown below in the sample image.

- Target size - overall image should populate the full screen. Measure target image width (**A**) and height (**B**) and make sure it matches with screen specification

- Bar size - the vertical bars in the middle (exclude the bars on the most left and most right edges, for example, in the sample image below, only count the 3 black and 2 white bars in the middle, in the order, black-white-black-white-black) in equal spacing each 10 mm wide, total number of bars x 10 mm **(C)**, and the vertical bar length is in integral number of 10 mm, for example, depends on screen size, it should be either 80 mm, 90 mm, or 100 mm, etc **(D)**

**Figure 8-4 Target Image Accuracy Check Points**

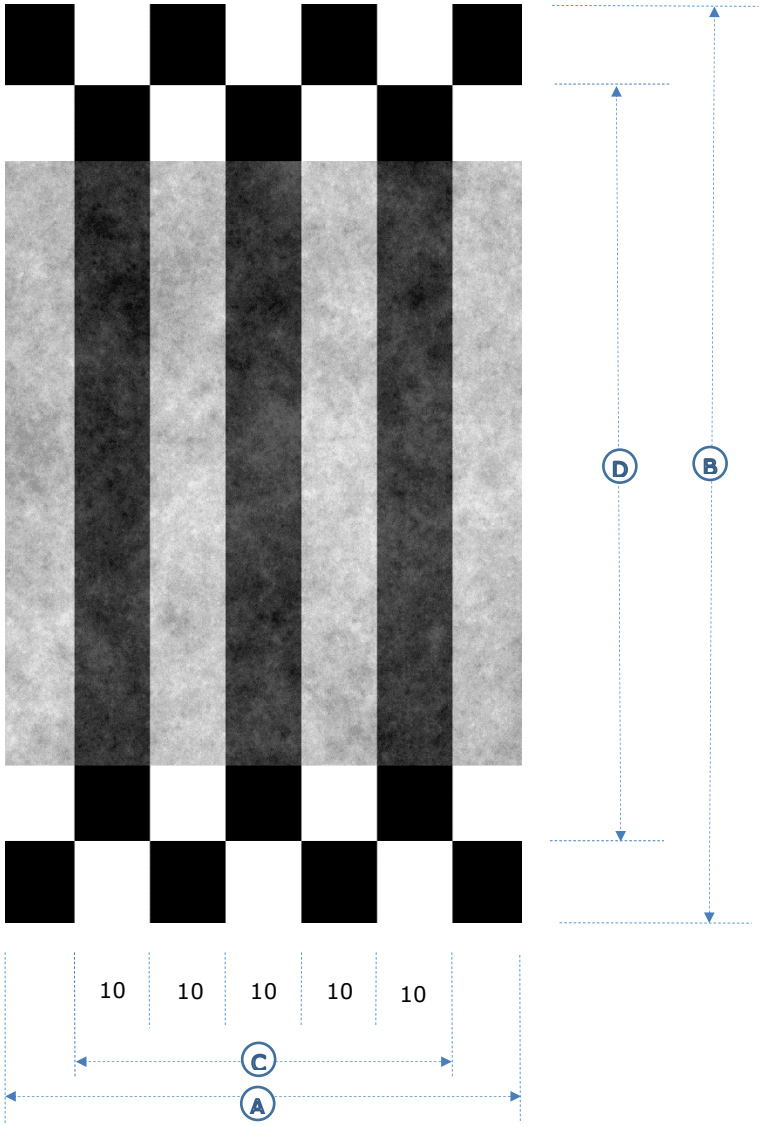


*Figure 8-5 target image*

To prototype the phone app and experiment possible issues on the various devices, a set of pre-generated images are supplied.

### 8.2.6 Display Control

The stripped bar target on the phone needs to present in the entire period of dynamic calibration without interruption. This means the app needs to meet a few requirements:

1) Display on. Make sure the display is not turned off while the app is running.
2) Display auto orientation. The target image should be displayed consistently without interrupt by display auto orientation. This is required on both iPhones.
3) The target images need to be undisturbed, i.e., it should not be resized or overlay with any screen pop ups as result of touch or other activities.

### 8.2.7 Display Zoom on Apple IOS Devices

Some Apple IOS devices, for example, iPhone 6, 6s, 7, 6+, 6s+, and 7+ models, support both a standard display and zoom display modes. The app needs to operate correctly in both modes.

To change display modes, go to

Settings → Display & Brightness → Display Zoom → View → Standard or Zoomed

### 8.2.8 App Name

The app should show the following title in development (to differentiate the previous app). Title may be finalized later. We need to keep both app alive in TestFlight for now, until the new app is ready to replace.

**Intel® RealSense™ Dynamic Target Tool**

### 8.2.9 Device Information Page

The app should provide a simple mechanism to display device and computed information for user to quickly self-check for potential errors.

The information page can be displayed with a swipe from left edge to the right. The information should contain:

- Device number
- Device model
- Display screen information both expected and computed (with same method used in scaling the target image):
- DPI pixels per inches
- Screen resolution in pixels
- Physical dimensions in width and height in cm, with precision to 0.01 cm. Expected values for this device model and actual values computed.
- Physical diagonal size of the screen in inches, expected for this device model and actual computed value.
- Number of black bars expected to display (see table in Target Image section)
- Black bar width in cm – 1 cm
- Version number at bottom, for example, version 2.2

The purpose of this information page is to ensure the user target is displayed as expected and identify any error quickly.

Below is an example of the info page on iPhone 6:

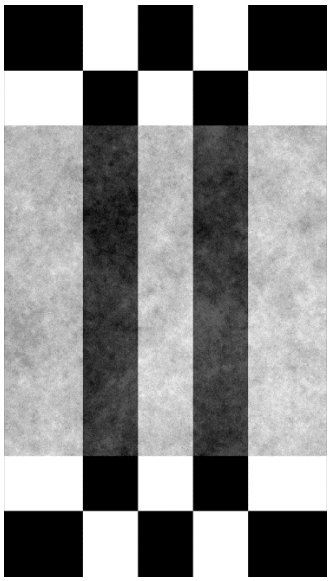


*Figure 8-6 target display screen on iPhone 6*

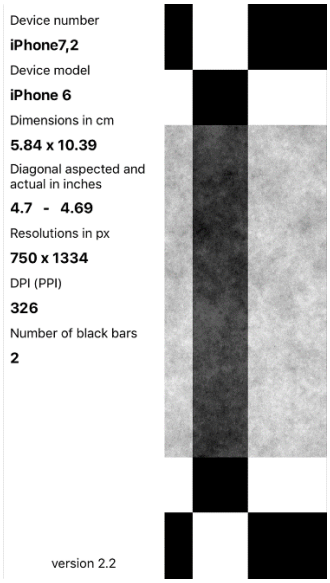



*Figure 8-7 info page display screen on iPhone 6*

## 8.3 Supported Platforms

The app needs to support devices with IOS and Android operating systems including phone and tablets.

### 8.3.1 Device OS

Apple IOS 10 and later

Android

Exact operating systems to be determined

### 8.3.2 Devices

#### 8.3.2.1 Apple devices

Apple iPhone X
Apple iPhone 8 plus
Apple iPhone 8
Apple iPhone 7 plus
Apple iPhone 7
Apple iPhone SE
Apple iPhone 6s plus
Apple iPhone 6s
Apple iPhone 6 plus
Apple iPhone 6
Apple iPhone 5s
Apple iPhone 5

#### 8.3.2.2 Android devices

AOS target app should work on devices with recent Android OS.