Intel® RealSense™ Depth Module D400 Series Dynamic Calibration

Programmer Guide

Revision 2.5.0.0

January 2018

Document Number: 575878

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

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

Contents

Tables

Figures

Revision History

Document Number	Revision Number	Description	Revision Date
575878	2.5.0.0	• Initial version	01/16/2018

§§

1 Introduction

1.1 Purpose and Scope of this Document

Dynamic Calibration API is a software interface to dynamically calibrate the depth sensors of Intel® RealSense™ D400 series 3D Cameras. It is designed to enable an OEM to develop customized calibration applications for their unique usages. This document describes the Application Programming Interfaces (APIs) available for an OEM to customize and build a dynamic calibration solution. Readers should refer to the user guide for installation and application usage instructions (Section 1.5).

1.2 Components

The Intel® RealSense™ Dynamic Calibration API includes:

- Dynamic Calibration API libraries and headers
- Dynamic Calibration API sample application
- Phone target sample phone app

1.3 Hardware Requirements

The Intel® RealSense™ Dynamic Calibration API supports any Intel® RealSense™ Depth Module in the D400 series:

- Intel® RealSense™ Depth Module D400
- Intel® RealSense™ Depth Module D410
- Intel® RealSense™ Depth Module D415
- Intel® RealSense™ Depth Module D420
- Intel® RealSense™ Depth Module D420 with Tracking Module
- Intel® RealSense™ Depth Module D430
- Intel® RealSense™ Depth Module D430 with Tracking Module
- Intel® RealSense™ Depth Camera D415
- Intel® RealSense™ Depth Camera D435

The camera device is connected to the host platform through USB 3.0 cable or USB Type C cable.

Figure 1-1. Intel® RealSense™ Depth Module D400 Series

The Intel® RealSense™ Depth Module D400 series complies with the requirements for a Class 1 laser device as defined by the IEC 60825-1 Edition 2 standard and FDA 21 CFR Subchapter J Parts 1040.10 and 1040.11.

There is no preventative maintenance or user serviceable parts inside the module. If an issue with the module is suspected or the module is damaged, discontinue use of the module and return to Intel. Do not attempt to disassemble or modify the module. Do not use optical instruments with this module. Failure to follow the guidelines can result in exposure to harmful 3B laser radiation.

Figure 1-2. Module Certification

Class 1 Laser Product classified EN/IEC 60825-1 2007
Caution: Class 3B Invisible laser radiation if opened or modified

Table 1-1. Platform Requirements

Feature	Support
• Supported Platforms	• Platform with USB3
• Supported OS	• Ubuntu 16.04, Yocto 2.1, Windows* 10
• Operating Temperature	• 20ºC - 30ºC

1.4 Software Requirements

- Intel® RealSense™ D400 series depth camera firmware version 5.8.15.0 or higher
- Linux*: 64 bit host system supporting Ubuntu* version 16.04
- Yocto 2.1
- Linux* kernel version – 4.4.0 or later
- Windows* 10 64-bit

1.5 Reference Documents

Table 1-2. Reference Documents

Document ID	Title	Link
574999	Intel® RealSense™ Depth Module D400 Series Dynamic Calibration User Guide	https://www.intel.com/content/www/us/en/support/articles/000026723.html

§§

2 Overview

This Chapter covers a brief overview of the dynamic calibration parameters and process.

2.1 Dynamic Calibration Parameters

All dynamic calibrations in this document are optimizing ***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, principal point are not dynamically calibrated.

Most of the dynamic calibrations assume they are re-calibrations after factory calibration, or at least the nominal parameters are known.

2.2 Types of Dynamic Calibration

A few different types of dynamic calibrations are supported for Intel® RealSense™ D400 series 3D camera devices:

1. **Rectification calibration**: aligning the epipolar line to enable the depth pipeline to work correctly and reduce the holes in the depth image
2. **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: target-less and targeted.

Targeted calibration is the recommended approach because it supports both rectification and depth scale calibrations, and will therefore give more accurate results than a simple rectification-only calibration.

2.2.1 Targeted Dynamic Calibration (Depth Scale Calibration)

In targeted mode, Dynamic Calibration API supports depth scale calibration and a target is required. The target is predefined and can be displayed on a smartphone through a phone app. A simplified flow is summarized as following:

1. Take the images from L and R camera, including the depth stream (in real-time)
2. Detect the target on the smartphone in both images
3. Similarly to target-less calibration, user moves the 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.

2.2.2 Target-less Dynamic Calibration (Rectification Calibration)

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 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.

Note: Targeted calibration is likely to give more accurate results than target-less calibration and is therefore the recommended approach.

2.3 Depth Quality Tool

The Intel® RealSense™ SDK includes a Depth Quality Tool that can be used to test the quality of the D400 series depth module if it is suspected that the module has gone out of calibration. Refer to the SDK releases page at https://github.com/IntelRealSense/librealsense/releases to download.

§§

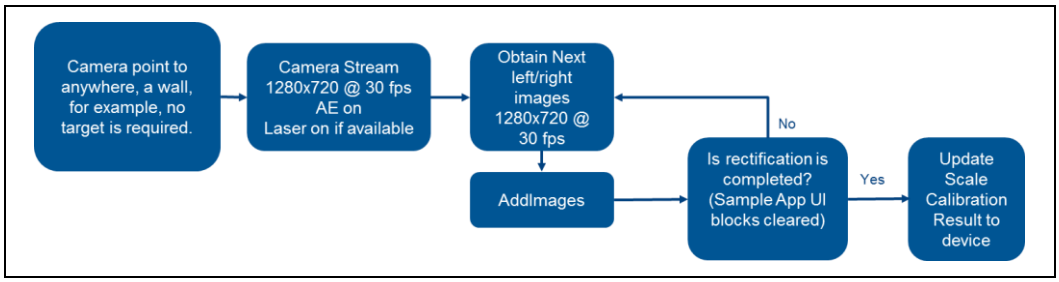
3 Dynamic Calibration API Use Cases

Two basic forms of dynamic calibration is supported: target-less and target based, each with different algorithms and use flows.

3.1 Target-less

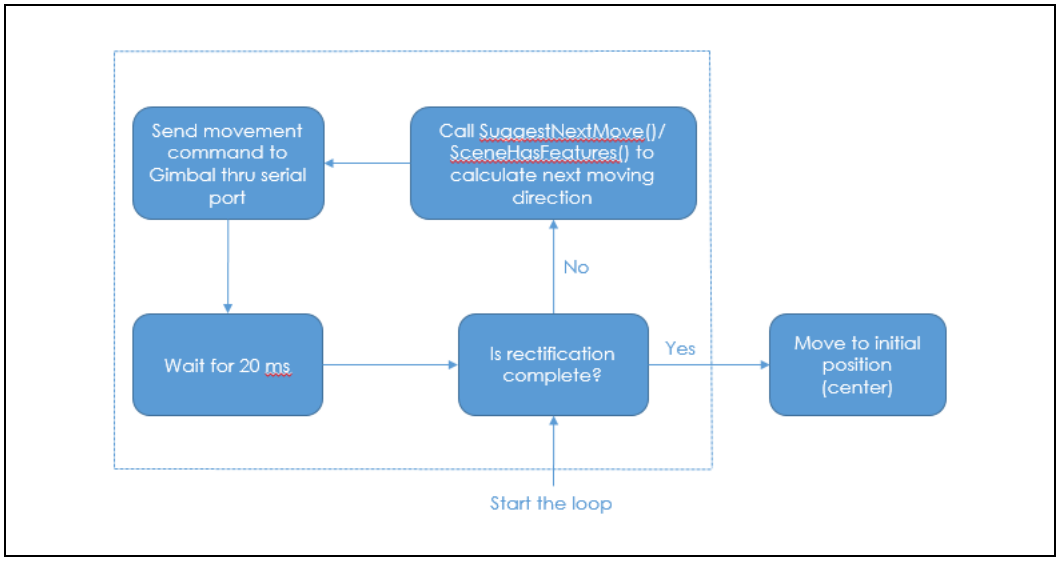
Rectification based dynamic calibration algorithm without target is supported. By default, 1280x720 resolution is supported. On modules with wide angle lenses such as D420 and D430 additional native resolution 1280x800 is also supported.

Figure 3-1. Target-less Calibration Flow



If a Gimbal is used with target-less calibration, it will operate independently of the above flow (Figure 4-1) to move the camera automatically based on scene detection results:

Figure 3-2. Gimbal Flow

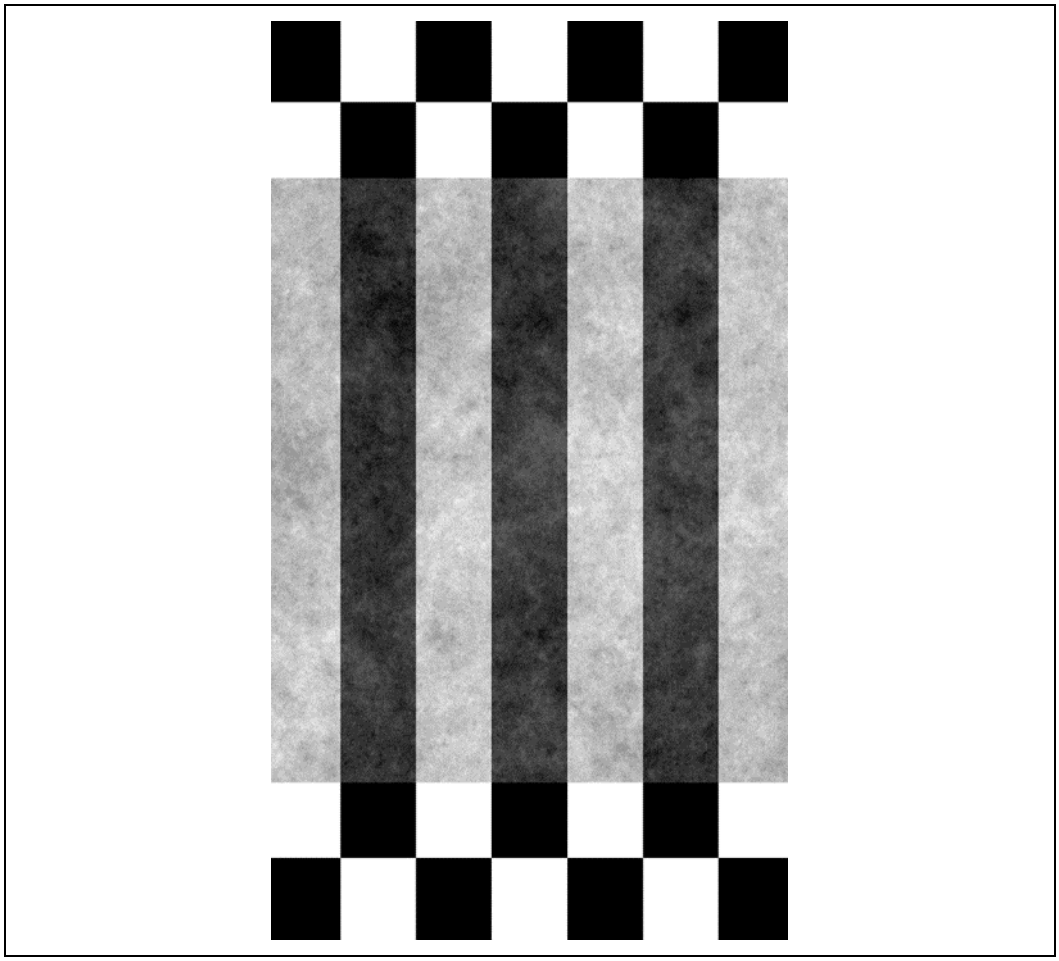


Note: Currently you can only use Gimbal for target-less calibration; targeted calibration is not supported yet.

3.2 Target-based

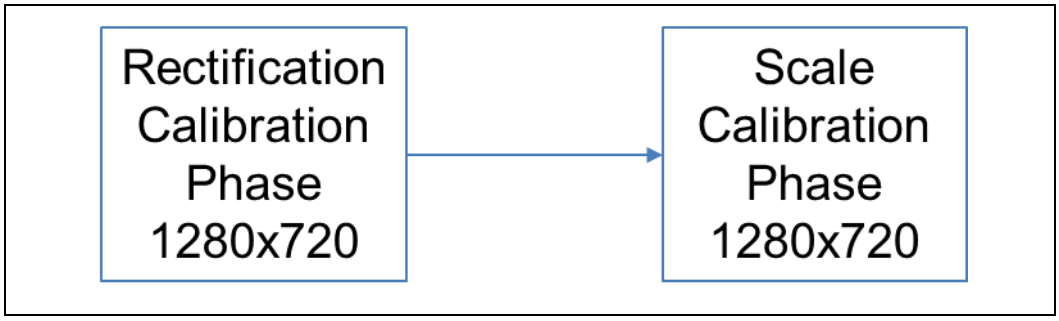
A cell phone with a graphical pattern displayed on the screen is used as a calibration target. An example of the target image is shown below. The actual image is device dependent (will be dynamically generated at runtime).

Figure 3-3. Target Image



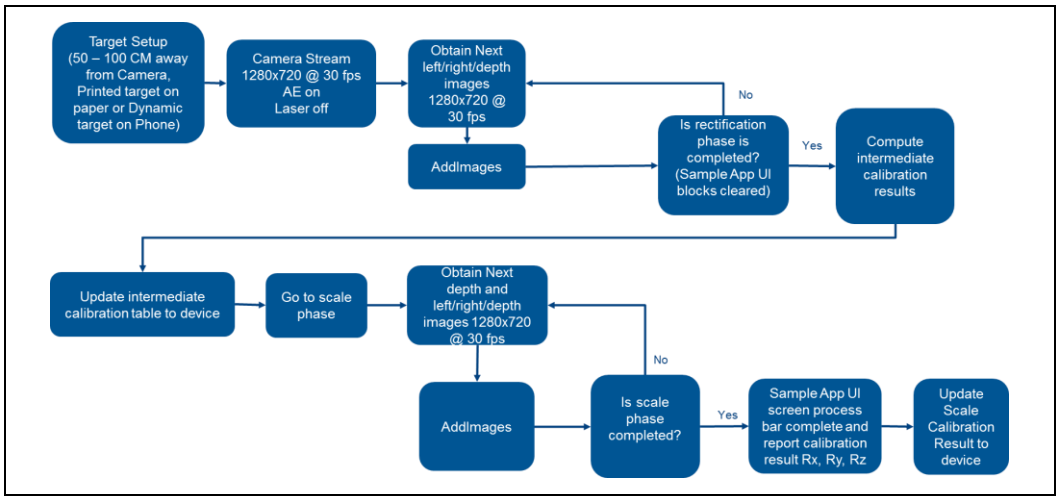
As part of the process, the application requires images of the calibration target and feed into dynamic calibration algorithm. The process is divided into two phases: rectification and scale calibration in sequential order, as shown above. Only 1280x720 resolution frames are supported.

Figure 3-4. Target-based Dynamic Calibration



The figure above shows a high level flow how targeted dynamic calibration is implemented based on a two-phase algorithm.

Figure 3-5. Targeted Dynamic Calibration Integrated Two-Phase Flow



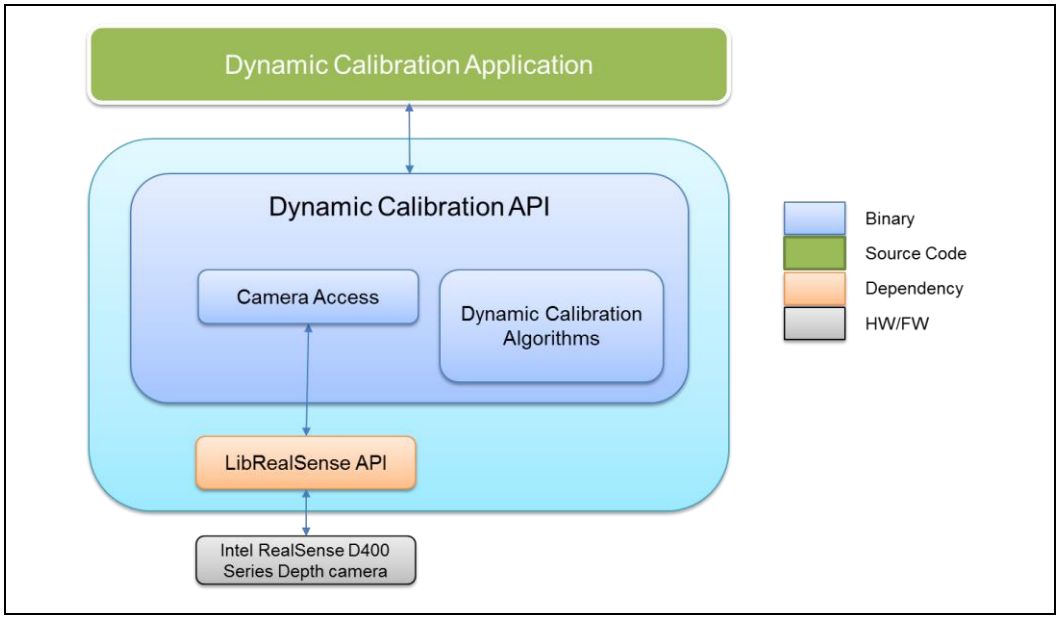
§§

4 Software Stack and Flow

4.1 Software Stack

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

Figure 4-1. Dynamic 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.

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

Camera Access: Camera interface module that provides access to attached cameras for calibration table read/write operations and other device operations using LibRealSense.

LibRealSense API: Open source camera access API for capture and other camera control operations.

Camera Device: Any supported Intel® RealSense™ D400 series Depth Cameras with proper firmware loaded.

4.2 Frame Formats Used in Dynamic Calibration

Format	SKU	Used	Comment
RY8_LY8 [16 bits]	ASR PSR DS5U with OV sensor	Dynamic Calibration Target-less Calibration: 1280x720 @ 30 fps or 1920x1080 @ 25 fps	Dynamic Calibration Target-less [Intel® RealSense™ Camera D400/D410/D415]
	AWG PWG	Dynamic Calibration Target-less Calibration: 1280x720 @ 30 fps or 1280x800 @ 30 fps	Dynamic Calibration Target-less [Intel® RealSense™ Camera D430, D420, D435]
ZR8L8	ASR PSR DS5U with OV sensor	Dynamic Calibration Targeted Calibration: 1280x720 @ 30 fps	Dynamic Calibration Targeted [Intel® RealSense™ Camera D400/D410/D415]
	AWG PWG AWGC	Dynamic Calibration Targeted Calibration: 1280x720 @ 30 fps	Dynamic Calibration Targeted [Intel® RealSense™ Camera D430, D420, D435]

4.3 API Usage for Calibration with Intel Algorithm

Dynamic Calibration API supports two distinctive Intel calibration algorithms: target-less and targeted.

4.3.1 Target-less Calibration

The following calling sequence illustrates a simplified flow for Linux* based target-less calibration. Refer to the API documentation and sample application source code for details.

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

// get camera access API handle g_rHal

// initialize camera

// enable laser projector if available

// setup camera streaming resolution and frame rate for left/right images

// initialize dynamic calibration API for target-less calibration with laser projector on
g_Dyncal.Initialize(g_rHal,
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 dynamic calibration 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
```

4.3.2 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. Refer to the API documentation and sample application source code for details.

```
// declare variable for dynamic calibration 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 dynamic calibration 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 dynamic calibration 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 dynamic calibration 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 the device
g_Dyncal.UpdateCalibrationTables();

// release handles
```

4.3.3 Sample Dynamic Calibrator Application

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

```
DynamicCalibrationAPI
        |-------- 2.5.0.0
        |           |-------- bin
        |           |--------examples
        |           |             |
        |           |             |----Common
        |           |             |       |-------- Rs400Device.h
        |           |             |       |-------- Rs400Device.cpp
        |           |             |       |-------- …
        |           |             |
        |           |             |----DynamicCalibrator
        |           |                     |-------- Main.cpp
        |           |                     |-------- MainView.cpp
        |           |                     |-------- …
        |           |                     |-------- makefile (Linux)
        |           |                     |-------- ...
        |           |                     |--------
        RS400DynamicCalibrationSample.sln (Windows)
```

To Compile the Sample Application

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

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

4.4 API Usage for Calibration with Custom User Algorithm

In most cases, users will use the Dynamic Calibration API with its embedded Intel algorithm to calibrate the D400 series devices. However, some users may prefer to develop their own custom calibration algorithms to fit their application needs.

The Dynamic Calibration API supports such calibration with user custom algorithms. It provides interfaces to write/read the optimized high level calibration parameters to the device and handle all internal data formats and device operations.

```
// 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 millimeters
//
    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 Dynamic Calibration 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, 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;
```

4.4.1 CustomRW Example for Custom Calibration Parameters Read/Write

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

```
DynamicCalibrationAPI
        |-------- 2.5.0.0
        |         |-------- bin
        |         |--------examples
        |         |         |
        |         |         |----Common
        |         |         |        |-------- Rs400Device.h
        |         |         |        |-------- Rs400Device.cpp
        |         |         |        |-------- …
        |         |         |
        |         |         |----CustomRW
        |         |                  |-------- Main.cpp
        |         |                  |-------- …
        |         |                  |-------- makefile (Linux)
        |         |                  |-------- ...
        |         |                  |-------- CustomRW.sln (Windows)
```

To Compile the Example Application

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

The executables CustomRW will be located under bin directory after the example is successful compiled. WriteCustomCalibrationParameters will alter calibration data on your device. If incorrect parameters are written, your device accuracy may be reduced or even malfunction. So, do so only when you really need to change calibration on your device.

§§

5 Dynamic Calibration API

Dynamic Calibration APIs provide interfaces for applications to access dynamic calibrator for real time re-calibration.

The APIs are defined as methods of DSDynamicCalibration Class.

For Linux* and Windows* 32 environment, DSDynamicCalibration Class is defined under namespace DynamicCalibrationAPI.

Dynamic Calibration API supports three calibration modes: target-less, targeted, and custom. Both Target-less and Targeted uses the Intel algorithm that is embedded in the API. Custom mode enables advanced users 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 */
        CAL_MODE_INTEL_TARGETLESS = 0,
        /** Intel targeted */
        CAL_MODE_INTEL_TARGETED = 1,
        /** User custom algorithm */
        CAL_MODE_USER_CUSTOM = 2
};
```

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

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

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

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

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

```
enum GridFeatures
{
    GRID_FEATURES_L0 = 0,          // Level 0 - no to little amount features
    GRID_FEATURES_L1 = 1,          // Level 1 - moderate amount of features
    GRID_FEATURES_L2 = 2           // Level 2 - good amount of features
};
```

5.1 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 handle is supported on both Windows* and Linux*.

mode: Calibration mode, **CAL_MODE_INTEL_TARGETLESS**, **CAL_MODE_INTEL_TARGETED**, 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.

Return Value: **DC_SUCCESS** – No issue.

DC_ERROR_INVALID_PARAMETER – Invalid input parameter.

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.

5.2 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.

Return Value:

(Target-less Calibration)
DC_SUCCESS
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

5.3 GetLastPhoneROILeftCamera()

(Targeted Calibration Only)

Get last position of target (printed or phone) in left image where 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 Value: Return error code indicates target location status.
DC_SUCCESS
DC_ERROR_NOT_IMPLEMENTED – if called in target-less mode
DC_ERROR_RECT_TARGET_NOT_FOUND_LEFT

5.4 GetLastPhoneROIRightCamera()

(Targeted Calibration Only)

Get last position of target (printed or phone) in right image where 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 Value: Return error code indicates target location status.
DC_SUCCESS
DC_ERROR_NOT_IMPLEMENTED – if called in target-less mode
DC_ERROR_RECT_TARGET_NOT_FOUND_RIGHT

5.5 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 is not ready by this time. This function returns value only in the scale phase of the process. Distance of the phone target in mm or -1.0 if the phone was not detected yet or not in scale phase of the process.

float **GetLastTargetDistance**();

Where:

Return Value: Distance of the phone target in mm or -1.0 if the phone was not detected yet or not in scale phase of the process.

5.6 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);

Where:

grid: Pointer to GridFill array with size of gridWith * gridHeight in row major.

gridWidth: Grid width as output.

gridHeight: Grid height as output.

5.7 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);

Where:

grid: Pointer to GridFeatures array with size of gridWith * gridHeight in row major.

gridWidth: Grid width as output.

gridHeight: Grid height as output.

5.8 GetGridLevels()

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.

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

Where:

grid: Pointer to GridLevel array with size of gridWith * gridHeight in row major.

gridWidth: Grid width as output.

gridHeight: Grid height as output.

5.9 GetGridLevelScore()

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

float **GetGridLevelScore**();

Where:

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

5.10 IsGridFull()

IsGridFull() method indicates whether the dynamic calibration is completed.

bool **IsGridFull**();

Where:

Return Value: **False**: Dynamic calibrator is still working on calibration.
True: Dynamic calibration is completed.

5.11 IsOutOfCalibration()

Determine 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.

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

Where:

outOfCalibration: Returns whether the device is out of calibration

Return Value: Returns error code of the evaluation:
DC_SUCCESS if successful,
DC_ERROR_PREMATURE if called before scale calibration is completed in targeted calibration.

5.12 GetCalibrationError()

(Target-less Calibration Only)

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.

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

Where:

calibrationError: Returns the calibration error

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

5.13 IsRectificationPhaseComplete()

(Targeted Calibration Only)

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

bool **IsRectificationPhaseComplete**();

Where:

Return Value: **False**: Rectification phase not complete
True: Rectification phase complete

5.14 IsScalePhaseComplete()

(Targeted Calibration Only)

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

bool **IsScalePhaseComplete**();

Where:

Return Value: **False**: Scale calibration phase not complete
True: Scale calibration phase complete

5.15 NumOfImagesCollected()

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

int **NumOfImagesCollected**();

Where:

Return Value: Number of image pairs

5.16 SetIntermediateRectificationCalibration()

(Targeted Calibration Only)

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

int **SetIntermediateRectificationCalibration**();

Where:

Return Value: DC_SUCCESS on success

DC_ERROR_NOT_APPLICABLE when called in target-less calibration mode.

DC_ERROR_FAIL if a problem causes failure.

5.17 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**();

Where:

Return Value: **False**:
True:

5.18 GetPhase()

Get the current calibration phase.

DC_PHASE **GetPhase**();

Where:

Return Value: **DC_TARGETED_RECTIFICATION_PHASE**
DC_TARGETED_SCALE_PHASE

5.19 GetTargetedCalibrationCorrection()

(Targeted Calibration Only)

Returns the updated calibration in rotation modification around the X, Y and Z axes once the scale phase is complete. This API requires that the scale phase be complete.

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

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.

Return Value: **False**: Failure.
True: Success.

5.20 UpdateCalibrationTables()

Write the updated calibration tables to the 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 Value: **DC_SUCCESS** – On success.
DC_ERROR_FAIL – On failure

5.21 GetVersion()

Get version of dynamic calibration as a string.

static const char * **GetVersion**();

Where:

Return Value: Version of dynamic calibration API.

5.22 WriteCustomCalibrationParameters

Write calibration parameters to the device. Assumes that the left camera is the reference camera and is located at world origin.

```
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:

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 pixelsparam 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.

Return Value: **DC_SUCCESS** – the tables were created successfully. Otherwise returns an 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 unreasonable.

§§

6 Shared API Definitions

6.1 Common Error Codes and Constants

This Chapter contains common error codes and other constants that are defined in the header **DSShared.h**.

```
// Error Codes
#define DC_SUCCESS                                  0   // successful
#define DC_ERROR_SCALE_DEPTH_TOO_SPARSE             4   // depth too sparse on target
                                                        // (maybe reflection)
#define DC_ERROR_SCALE_DEPTH_NOT_CONSISTENT         6   // target depth is not consistent
                                                        // between dense and sparse features
#define DC_ERROR_SCALE_TARGET_TILT_ANGLE_BIG       10   // target tilt angle too big
#define DC_ERROR_SCALE_FAILED_COMPUTE              11   // failed to calculate correction angle
#define DC_ERROR_SCALE_ALREADY_CAPTURED            12   // already captured
#define DC_ERROR_SCALE_DEPTH_NOT_A_PLANE           13   // depth not fit on a plane


#define DC_ERROR_RECT_INVALID_IMAGES             1001   // images are invalid
#define DC_ERROR_RECT_TOO_SIMILAR                1002   // images too similar
#define DC_ERROR_RECT_TARGET_NOT_FOUND_LEFT      1003   // rectification target not found
                                                        // in left image
#define DC_ERROR_RECT_TARGET_NOT_FOUND_RIGHT     1004   // rectification target not found
                                                        // in right image
#define DC_ERROR_RECT_TARGET_NOT_FOUND_BOTH      1005   // target not detected in both
                                                        // left and right images
#define DC_ERROR_RECT_TARGET_NOT_CONSISTENT      1006   // target not consistent between
                                                        // left and right images
#define DC_ERROR_RECT_GRID_FULL                  1007   // grid is full, no more images needed
#define DC_ERROR_RECT_INVALID_GRID_FILL          1008   // invalid grid fill
#define DC_ERROR_RECT_TOO_MUCH_FEATURES          1009   // too much features
#define DC_ERROR_RECT_NO_FEATURES                1010   // no features


#define DC_ERROR_TARGET_UNSTABLE                 1011   // target not stable
#define DC_ERROR_TARGET_TOO_CLOSE                1012   // too close
#define DC_ERROR_TARGET_TOO_FAR                  1013   // too far
#define DC_ERROR_PHASE_COMPLETED                 1014   // phase completed,
                                                        // no more images accepted


#define DC_ERROR_EXCEPTION                       9001   // exception
#define DC_ERROR_DEVICE_INVALID                  9002   // device is null, invalid


#define DC_ERROR_PREMATURE                       9800   // basic requirement not satisfied,
                                                        // too early to call


#define DC_ERROR_RESOLUTION_NOT_SUPPORTED_V2     9900   // version 2.0 coefficient table does not
                                                        // support 1280x800 resolution mode
#define DC_ERROR_TABLE_NOT_SUPPORTED             9901   // calibration coefficient tablet version
                                                        // not supported
#define DC_ERROR_TABLE_NOT_VALID_RESOLUTION      9902   // image width and height in calibration
                                                        // table is not valid. only 1920x1080 and
                                                        // 1280x800 supported depends on
                                                        // device models


#define DC_ERROR_INVALID_PARAMETER               9995   // Invalid argument passed in
#define DC_ERROR_NOT_APPLICABLE                  9996   // Feature or data not applicable to the
                                                        // calibration flow
#define DC_ERROR_NOT_IMPLEMENTED                 9997   // Interface or feature not implemented
                                                        // or not applicable
#define DC_ERROR_FAIL                            9998   // generic error for failure where error
```

```
                                                                    // code is not used
#define DC_ERROR_UNKNOWN                                    9999    // other errors for unknown reason


// Calibration types target-less and targeted
// targeted consists of two phases - rectification and scale
enum DC_PHASE : int
{
      DC_TARGETLESS                         = -1,
      DC_TARGETED_RECTIFICATION_PHASE       = 0,
      DC_TARGETED_SCALE_PHASE               = 1
};


enum ds5_rect_resolutions: unsigned short
{
      res_1920_1080,
      res_1280_720,
      res_640_480,
      res_848_480,
      res_640_360,
      res_424_240,
      res_320_240,
      res_480_270,
      res_1280_800,
      res_960_540,
      reserved_1,
      reserved_2,
      ds5_rect_resoluitons_num
};


struct int2 { int x, y; };


static std::map< ds5_rect_resolutions, int2> resolutions_list = {
      { res_320_240,      { 320, 240 } },
      { res_424_240,      { 424, 240 } },
      { res_480_270,      { 480, 270 } },
      { res_640_360,      { 640, 360 } },
      { res_640_480,      { 640, 480 } },
      { res_848_480,      { 848, 480 } },
      { res_960_540,      { 960, 540 } },
      { res_1280_720,     { 1280, 720 } },
      { res_1280_800,     { 1280, 800 } },
      { res_1920_1080,    { 1920, 1080 } },
};
```

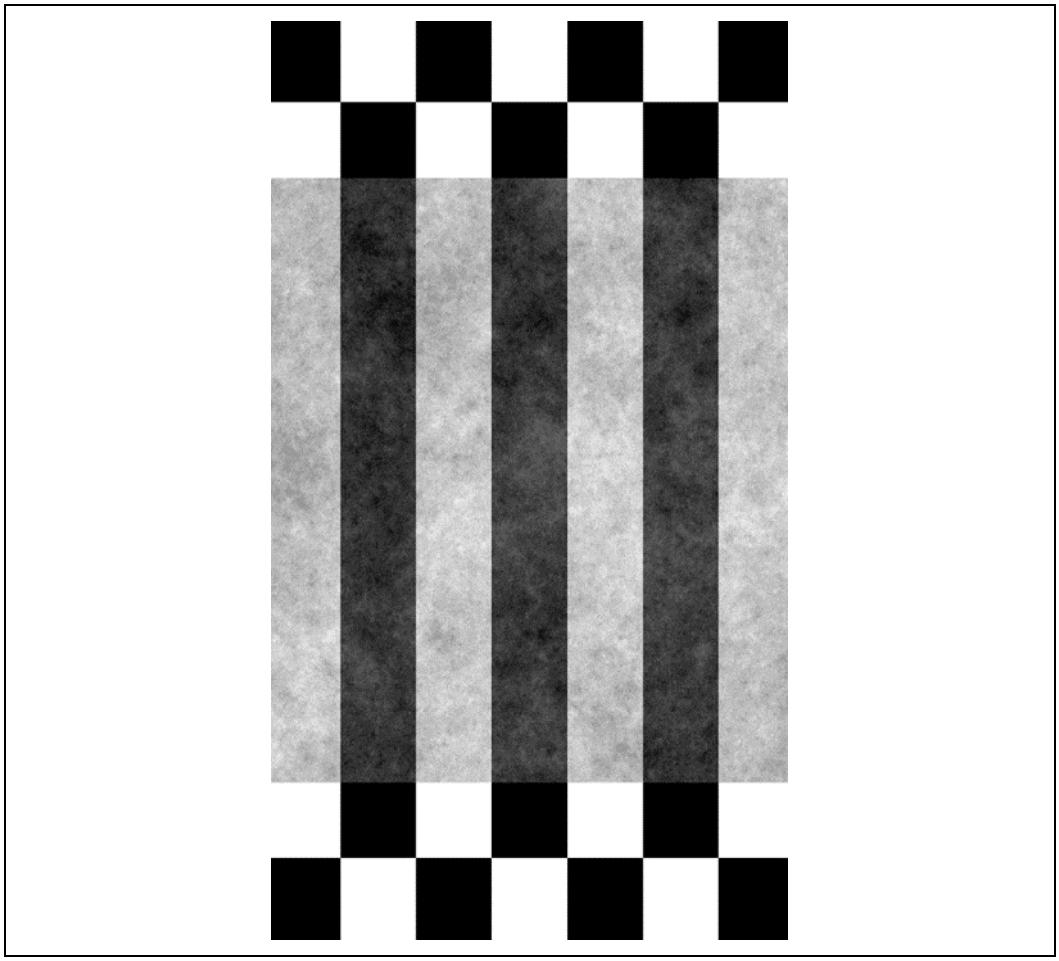
§§

7 Phone Target Application Integration

7.1 Target

Targeted calibration requires a fixed width (10 mm) bar target, either displayed on phone through a phone application 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.

Figure 7-1. Example Phone Calibration Target Image



7.2 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 application 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 background
 * 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.

7.3 Phone Application

Intel provides a sample phone application on the Apple* App Store for iPhones and on Google* Play for Android* phones. Refer to Chapter 8 in this document for the phone target application technical specification. Source code for the sample phone applications is also provided for integration into OEM implementations.

7.4 Devices Validated

The sample phone application is validated on a limited number of devices, both iOS and AOS. Refer to Chapter 9 for details.

§§

8 Intel® RealSense™ Dynamic Target Tool Phone Application Technical Specification

8.1 Introduction

Dynamic calibration is used to re-calibrate the Intel® RealSense™ D400 Series 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 application.

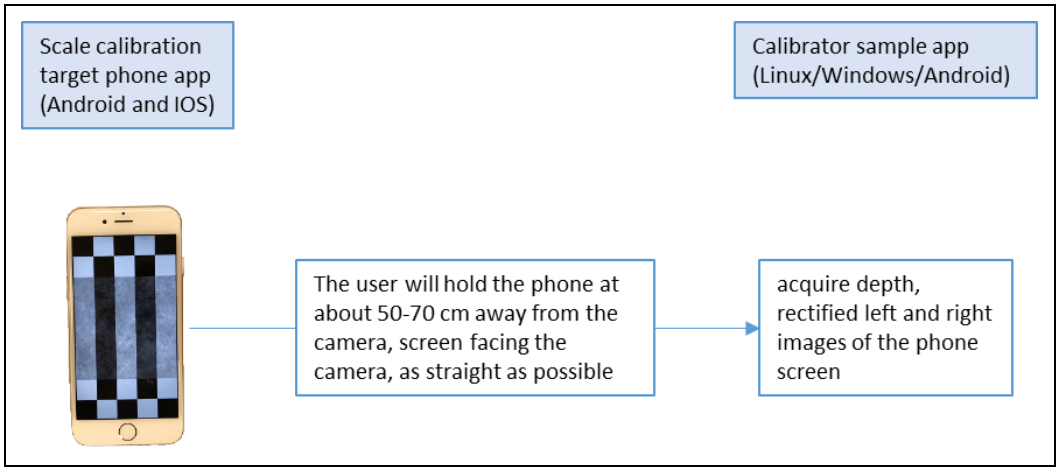
This chapter provides technical specifications for the dynamic calibration target tool application.

8.2 Application Flow and Requirements

8.2.1 Overview

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

Figure 8-1. Dynamic Calibration Simplified Flow



1. The application detects supported 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 a first experiment in prototyping the application, pre-generated images will be pre-supplied by Intel). The image will be an 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 meets certain requirements. An algorithm to generate the image is also supplied by Intel for dynamic composition after the prototyping with pre-generated images.

3. The application reads the target image.

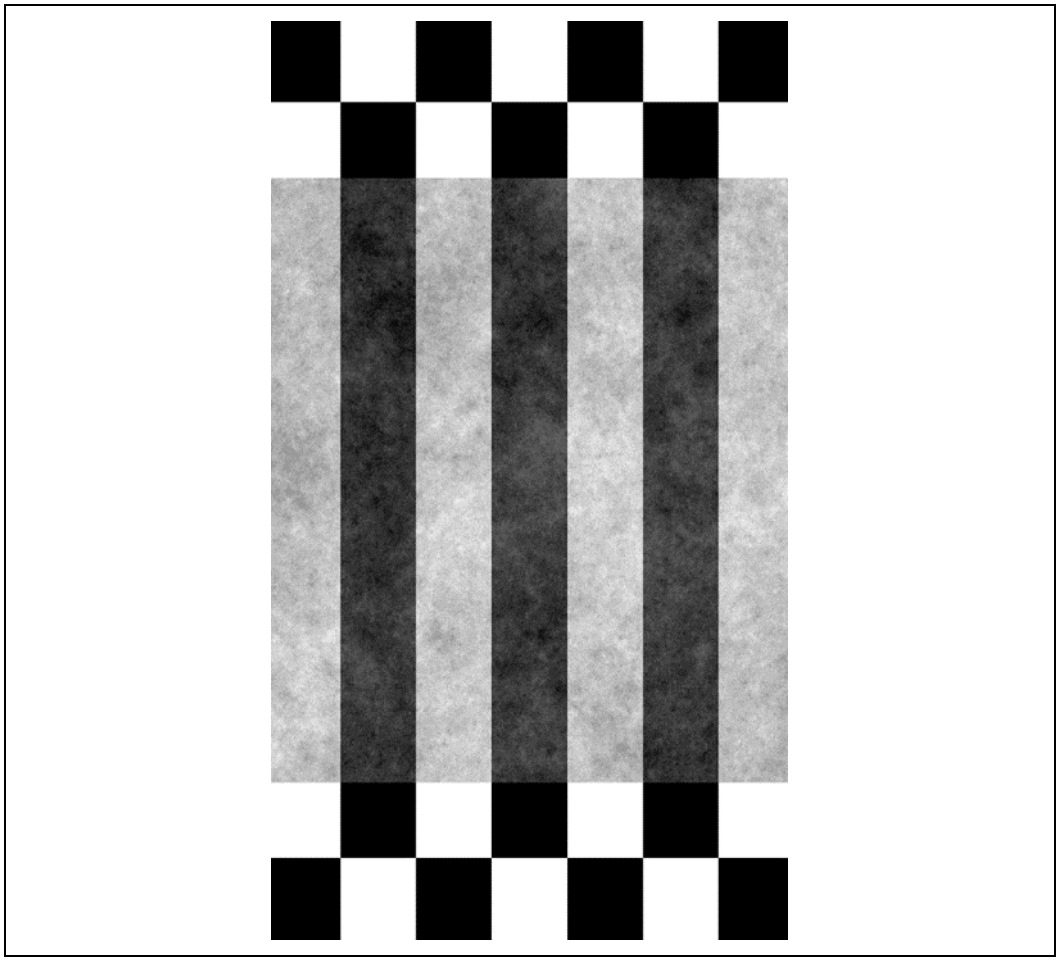
The application displays the target image on the phone screen. The image resolution is exact match to the screen resolution, so no scaling is required, and the image will be displayed as is so that the physical width of each bar is equal to a pre-designated fixed width, in this case 10 mm. Since the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

1. The accuracy of the 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 performs dynamic calibration on the Intel® RealSense™ Camera.

8.2.2 Target Image

An example of the target image is shown below. The actual image is device dependent and is dynamically generated at runtime.

Figure 8-2. Target Image



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

Table 8-1. Supplied Pre-Generated Images

Apple* Device Displays	Resolution (Pixel)	DPI (PPI)	Screen Physical Dimension Width x Height (cm)	Devices With Display	Intel Supplied Target Image	Number of Black Vertical Bars
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

Apple* Device Displays	Resolution (Pixel)	DPI (PPI)	Screen Physical Dimension Width x Height (cm)	Devices With Display	Intel Supplied Target Image	Number of Black Vertical Bars
5.5”	1080 x 1920	401	6.84 x 12.16	iPhone 6 plus, 6s plus, 7 plus	ios-55in.png	3
7.9”	1536 x 2048	326	11.97 x 15.96	iPad mini 2, mini 3, mini 4	ios-79in.png	5
9.7”	1536 x 2048	264	14.78 x 19.70	iPad Air, iPad Air 2, 9.7” iPad Pro iPad 5	ios-97in.png	6
12.9”	2048 x 2732	264	19.70 x 26.29	12.9” iPad Pro	ios-129in.png	9

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 it difficult to pre-generate target images 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 an API for the phone application 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.

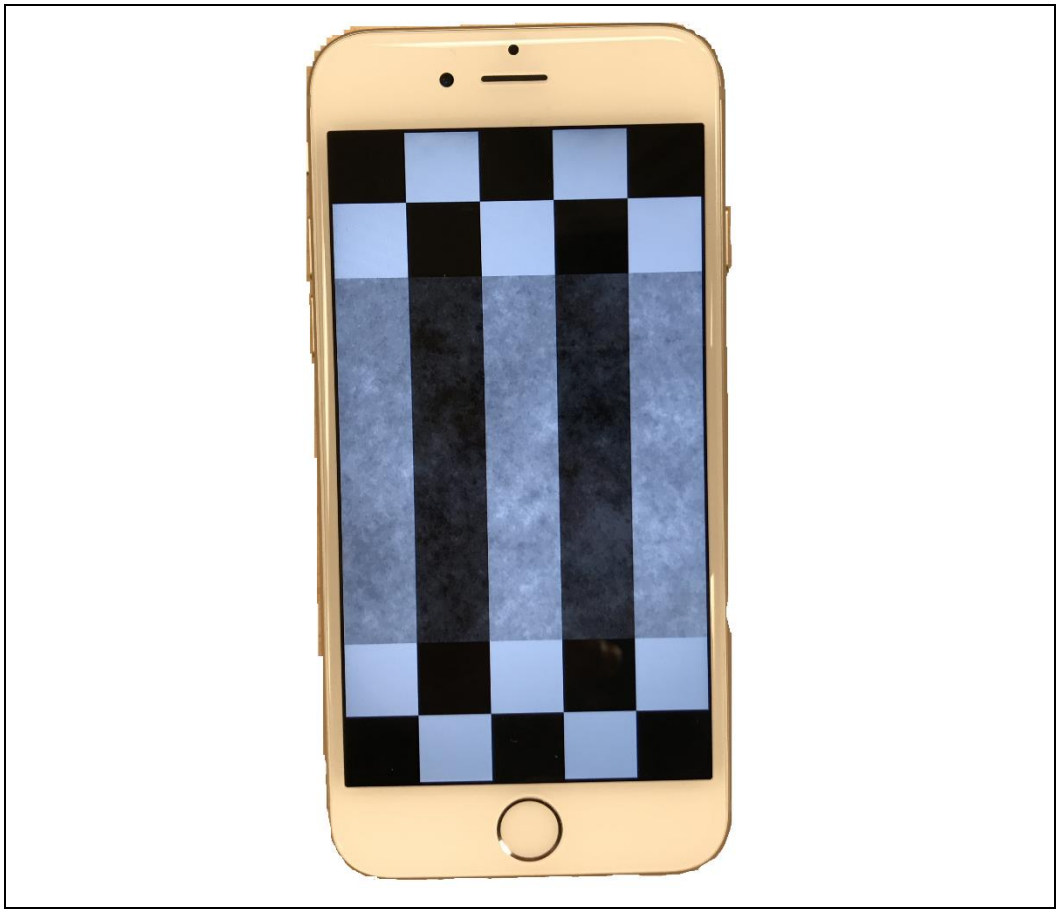
8.2.4 Target Image Display

The image is stored or generated at runtime on the phone device as part of the application. The application 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 preset fixed width. The width is currently determined to be 10 mm. This predetermined width might be adjusted later after actual accuracy of the calibration process is measured and tuned. The application needs to take this into account so that it will not cause issues when there is a need to change it in later development process.

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

An example target image (**`ios-47in.png`**) displayed on an iPhone 6 is shown below.

Figure 8-3. Target Displayed on iPhone 6

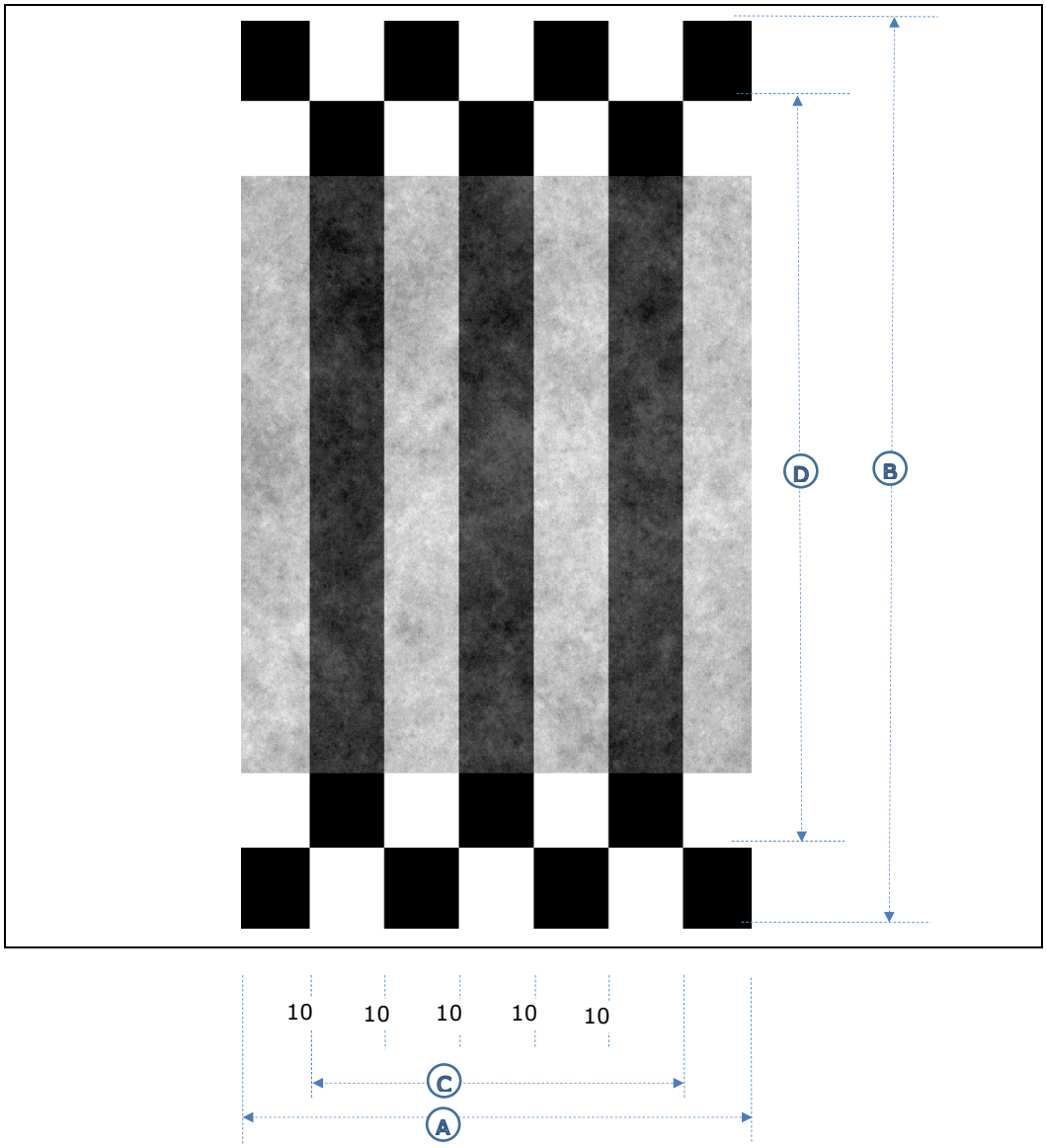


8.2.5 Target Image Accuracy Check

Target precision is critical to calibration accuracy. After the target image is successfully displayed on phone screen, 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



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

8.2.6 Display Control

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

1. **Display on**. Make sure the display is not turned off while the application 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 and iPads.
3. **Image stability**. The target images need to be undisturbed, i.e., they should not be resized or overlaid with any screen pop ups as a 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 plus, 6s plus, and 7 plus models, support both a standard display and zoom display modes. The application needs to operate correctly in both modes.

To change display modes, go to:

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

8.2.8 Application Name

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

Intel® RealSense™ Dynamic Target Tool

8.2.9 Device Information Page

The application should provide a simple mechanism to display device and computed information for users 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 (refer 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 the user can identify any error quickly.

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

Figure 8-5. Target Display Screen on iPhone 6

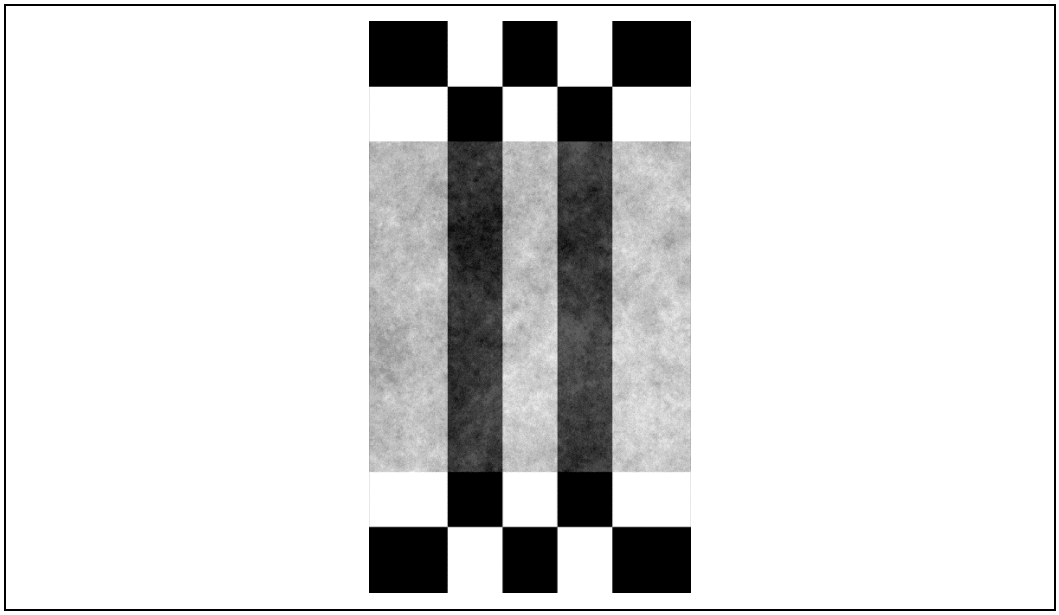
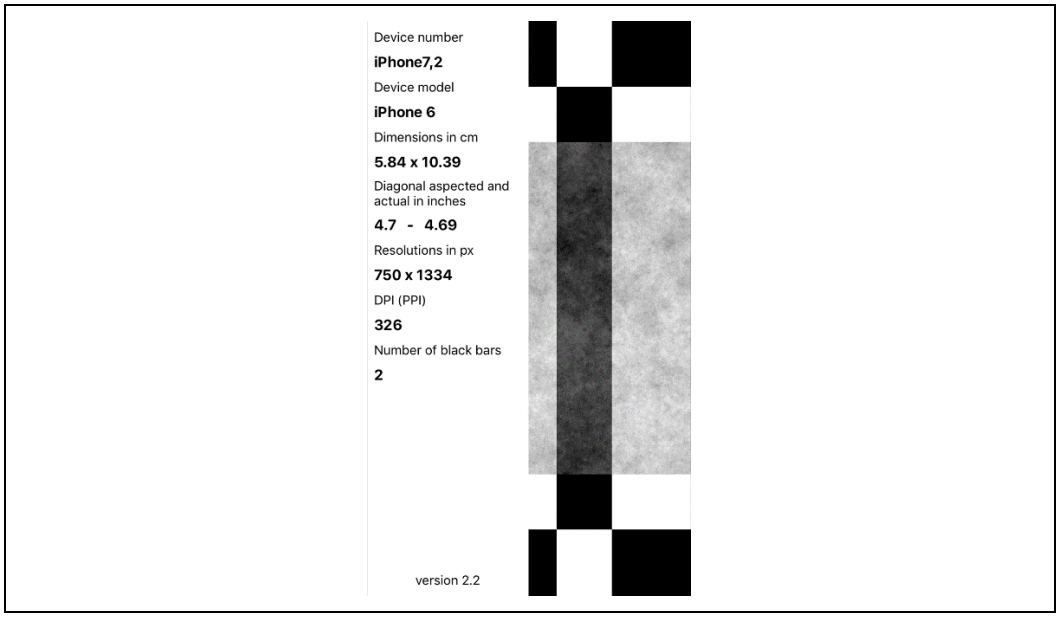


Figure 8-6. Information Page Display Screen on iPhone 6



8.3 Supported Platforms

The application needs to support devices with iOS and Android* operating systems including phones and tablets.

8.3.1 Device OS

- Apple* iOS 10 and later
- Android*
- Exact operating systems to be determined.

8.3.2 Apple* Devices

Table 8-2. Supported Apple* Devices

Apple* iPhone	Apple* iPad
Apple* iPhone 7 plus	Apple* iPad Pro (12.9”)
Apple* iPhone 7	Apple* iPad Pro (9.7”)
Apple* iPhone SE	Apple* iPad 5
Apple* iPhone 6s plus	Apple* iPad Air 2
Apple* iPhone 6s	Apple* iPad Air
Apple* iPhone 6 plus	Apple* iPad 4th Gen
Apple* iPhone 6	Apple* iPad mini 4

Apple* iPhone	Apple* iPad
Apple* iPhone 5s	Apple* iPad mini 3
Apple* iPhone 5	Apple* iPad mini 2

8.3.3 Android* Devices

AOS target application should work on devices with recent Android* OS.

§§

9 Validated Devices for Intel® RealSense™ Dynamic Target Tool Phone Application

The Intel® RealSense™ Dynamic Target Tool phone application is validated on a limited number of mobile devices.

9.1 Validated iOS Devices

On iOS devices, display resolution and pixel density is limited to a few combinations, so although validation is done only on limited number of devices, it covers all display combinations in existing Apple* iOS devices.

Table 9-1. Validated Apple* iOS Devices

Apple* Device Displays	Resolution (Pixel)	DPI (PPI)	Screen Physical Dimension Width x Height (cm)	Devices With Display	Devices Formally Tested by Development and QA	Devices Tested Ad hoc by Others
4”	640 x 1136	326	4.99 x 8.85	iPhone 5, 5s, 5c, SE	iPhone 5s	iPhone 5
4.7”	750 x 1334	326	5.88 x 10.46	iPhone 6, 6s, 7	iPhone 6	iPhone 6s, 7
5.5”	1080 x 1920	401	6.84 x 12.16	iPhone 6 plus, 6s plus, 7 plus	iPhone 6 plus, 7 plus	iPhone 6s plus
7.9”	1536 x 2048	326	11.97 x 15.96	iPad mini 2, mini 3, mini 4	iPad mini 2	
9.7”	1536 x 2048	264	14.78 x 19.70	iPad Air, iPad Air 2, 9.7” iPad Pro	iPad Air 2	iPad Air

9.2 Validated AOS Devices

Display resolution and pixel density varies from device and vendors on Android* devices. The target image is dynamically generated to fit to the display on each device. The image generation algorithm is precise in case the display resolution and pixel density is accurate. However, there are known cases where the OS reports wrong pixel density that would impact target accuracy. It’s important to physically check the target sizes in order to identify such issue early on.

Table 9-2. Validated Android* Devices

Devices Formally Tested by Development and QA	Devices Tested Ad hoc by Others	OS Version
Samsung* Galaxy Tab S2 8”		6.0.1 (Marshmallow)
Samsung* Galaxy S7		6.0.1 (Marshmallow)
Google* Nexus 5		4.4.2 (KitKat)
LG* G4		6.0 (Marshmallow)
Samsung* Galaxy Note 5		6.0 and 7.0
	Google* Nexus 6	
	Samsung* Galaxy S7 Edge	

9.3 Devices with Accuracy Issue in Phone Target

The following table lists devices that are known to generate inaccurate targets. Avoid using these devices:

Table 9-3. Devices with Target Accuracy Issues

Devices with Target Accuracy Issues	OS Version	Notes
Samsung* Galaxy S5	5.1.1	**DS0-5731** – Target on Samsung* Galaxy S5 phone not accurate

§§