Intel® RealSense™ Self-Calibration for D400 Series Depth Cameras

Anders Grunnet-Jepsen, John Sweetser, Tri Khuong, Sergey Dorodnicov, Dave Tong, Ofir Mulla

Rev 2.1

1. INTRODUCTION:

Intel® RealSense™ Depth Cameras D400-series are based on calculating depth from stereo vision. All sensor modules are built from factory to be extremely sturdy, encased in laser-fused steel cages, with the intent of maintaining calibration and performance over their lifetime. However, conditions can occur that lead to degradation over time, such as exposure to extreme temperature cycling, or excessive shock and vibe. Whatever the cause, Intel provides a set of tools to recalibrate cameras back to their pristine factory condition. These tools include "OEM calibration" based on targets, as well as some "Dynamic Calibration" methods that can restore performance in the field.

In this whitepaper we introduce a new set of Intel RealSense™ SDK2.0 (aka LibRealSense) commands that we call "Self-Calibration". They are meant to 1. Restore the depth performance, and 2. Improve the accuracy, for any Intel RealSense™ Depth Camera D400 series that may have degraded over time. These new features work on any Operating System or compute platform, as they simply invoke new Firmware (FW) functions inside the ASIC. As a result, they also have essentially zero load on host CPU and are very fast. Moreover, while we will demonstrate that the methods work best with specific types of targets, we also show that they can work completely target-less on non-flat objects or scenes, with only some minor constraints. Other benefits of these new techniques are that they require no motion or repositioning during calibration, and they can complete in seconds. So how often is it required to run these self-calibration techniques? It may indeed never be required, as RealSense cameras are designed to maintain calibration. However, these tools can also serve as a validation that a device is performing to its limits, in that it is possible to run these calibration routines and compare the performance before and after, without actually permanently updating the new calibration inside the camera. We refer to this feature as the "Health-Check" function that will give a direct metric of the calibration state, that can therefore be monitored over time, without the need for special targets.

2. DEPTH NOISE:

2.1 Defining Depth Precision

We start by introducing a new method for restoring the camera to optimize its *depth performance* in terms of minimizing depth noise. To be specific, this first method focuses on the ability of the cameras to see objects and report back their position with low noise. In other words, we improve the precision, or relative error. In section 2 we will focus on improving the accuracy, or absolute error.

When D4xx cameras degrade in performance, they tend to do so by gradually showing more noise in the depth map. This noise can be best visualized by looking at a textured flat wall where the amount of bumpiness (depth variation) will increase as the units go more out of calibration, until the depth measurements start failing altogether, returning values of 0. At this point the nearly 100% "fill factor" diminishes quickly, and "holes" start to appear in the depth map.

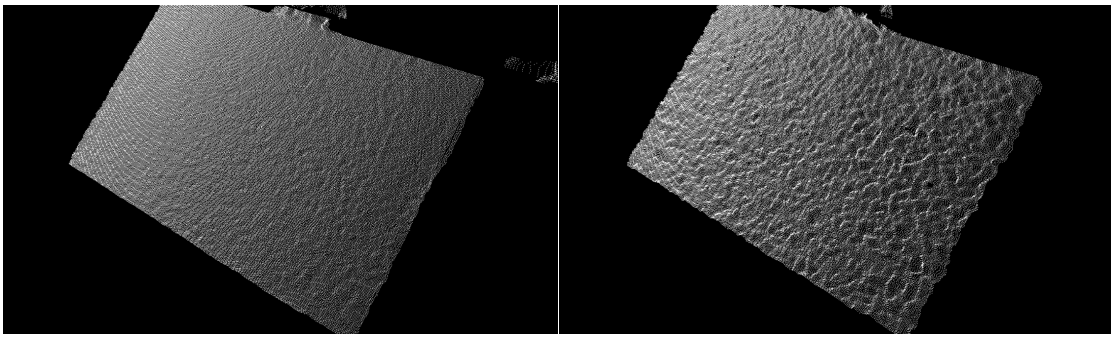


Figure 1. Comparisons of the Point Cloud of a well calibrated camera (LEFT) with a degraded camera (RIGHT) for a flat textured wall. The lower bumpiness on the left is preferred.

To quantify the depth precision (relative error), the cameras can be pointed at a flat target, like a wall, and depth can be measured at every point in a small section of the Field-of-View (FOV), typically a 10%-20% region-of-interest (ROI) near the center. This ROI depth map is then fitted to a plane, and RMS Depth noise is measured as the standard deviation from this plane. While this can be reported in absolute units, like 5mm for example, the best way to compare depth performance across cameras, depth ranges, resolutions, FOV, and projector variations is to use the normalized metric called the Subpixel RMS value:

$$\boldsymbol{Subpixel\ (pixels)} = \frac{\boldsymbol{focal\ length(pixels)x\ Baseline(mm)x\ Depth\ RMS\ error(mm)}}{\boldsymbol{Distance(mm)^2}}$$

$$\boldsymbol{where\ focal\ length(pixels)} = \frac{\mathbf{1}}{\mathbf{2}}\frac{\boldsymbol{Xres(pixels)}}{\boldsymbol{tan}(\frac{\boldsymbol{HFOV}}{\mathbf{2}})}$$

Where the Depth RMS error is the noise of a localized plane fit (generally the “bumps” or in some cases “egg carton effect”), focal length is the depth sensor’s focal length normalized to depth pixels, Baseline is the distance between the left and right imagers, Distance is the range to the wall, HFOV is the horizontal field-of-view of the stereo imager, and Xres is the depth map horizontal resolution at which the measurement is done, for example 1280 or 848. The HFOV, Baseline, and Focal Lengths can be obtained by querying for the camera intrinsics and extrinsics using the Intel RealSense SDK 2.0[1]. The D415 has a nominal HFOV~65deg and baseline of ~55mm, while for the D435 the HFOV~90deg and baseline~50mm.

2.2 Example Target for Calibration & Characterization

When running the on-chip Self-Calibration routine, the ASIC will analyze all the depth in the full field-of-view. However, since it can sometimes be difficult to ensure there is good texture in a full FOV, we have enabled a new resolution mode of 256x144 which outputs the zoomed-in central Region-Of-Interest (ROI) of an image, reduced by 5x in each axis from 1280x720. Figure 2 shows an example of a D435 pointed at a wall with a small target, using the full FOV, and the better suited smaller FOV.

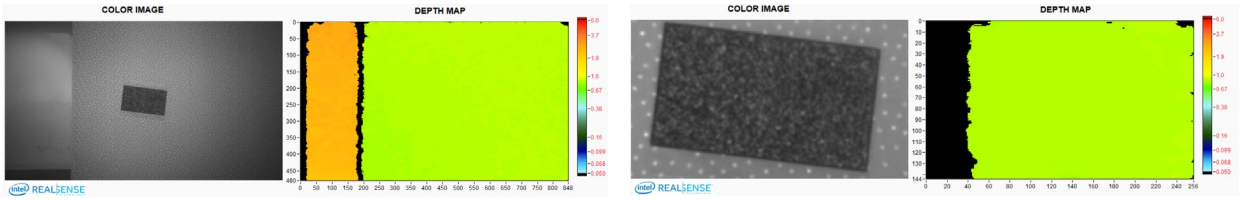


Figure 2. Full Field-of-view of a D435 camera, vs a central region-of-interest zoomed-in view with resolution mode 256x144 (zoomed in 5x from 1280x720). The smaller resolution mode is recommended for calibration.

The smaller ROI is especially well suited to looking at an A4-size target, such as that shown in Figure 3, and reproduced on the last page of this document so it can be printed out.

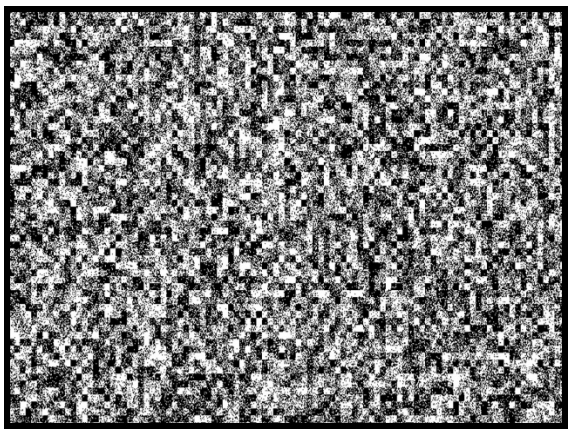


Figure 3. Example A4-sized target with texture which is particularly well suited for both on-chip calibration and depth quality characterization. The exact nature of the texture is not critical, as long as it is semi-random and fairly "noisy" with high spatial frequencies.

2.3 Running the Self-Calibration routine

The process for running the on-chip self-calibration is straight-forward and is separated into three steps. 1. Run the self-calibration routine, 2. Validate the improvement, and 3. Burn the result permanently into FW memory. Before embarking on this it is important to make sure that the Intel RealSense™ Depth Camera D400 has the latest Firmware (FW version: 5.12.02.100) that includes the new self-calibration features, and LibRS version 2.33.

In the following we will describe the measurement set-up and the programming details. We will then show how this can be tested immediately using the latest Intel RealSense Viewer or Intel RealSense Quality Tool, requiring no knowledge of the underlying programming.

We start by describing the recommended set up. While the main rule of thumb is that the camera can be pointed at any scene that would normally generate >50% good depth points, we emphasize here the value of getting up and going under ideal conditions.

As a good reference, you can print the target attached in the appendix A and fasten it flattened to a wall. Note that the flatness has no impact on the self-calibration process itself, but it will have a significant influence on how well it is possible to characterize and validate the results afterwards if that is desired. In general, any natural scene with sufficient texture will work, as described in section 2.4.

Place the camera far enough away that it is beyond the minimum range (aka. The minZ) of the depth camera, but close enough that at least 35% of the target texture is visible in the image. Ideally it should fill

the zoomed-in ROI. With this setup the color and depth map should look like that in Figure 4. It is not critical to ensure that the wall is exactly perpendicular.

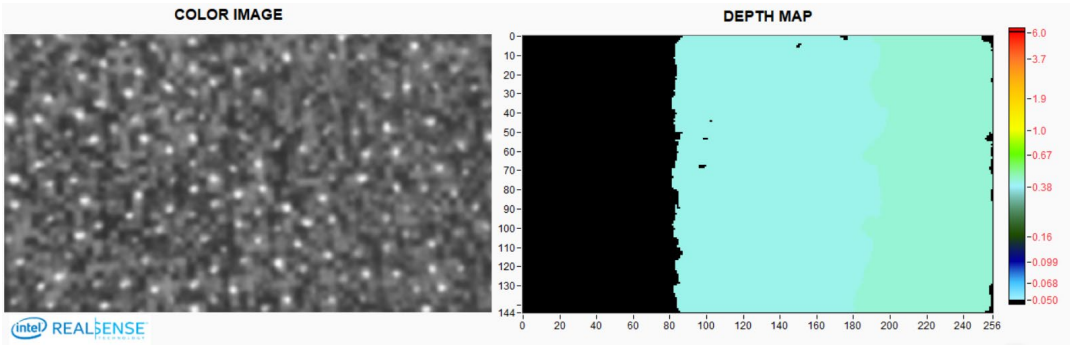


Figure 4. Example zoomed-in 256x144 image of a D435 monochrome image (from left imager) and depth map, as observed by looking at a textured target mounted on a flat wall. The black bar on the left is expected and acceptable. In general, it is important to have >35% of the depth map show depth values (non-zero). In this example the IR projector was left on, but we recommend turning it off when pointing at a well-textured scene.

Figure 5 shows a few different examples of target placements. While all these setups work, the one described in the bottom guarantees best performance and robustness.

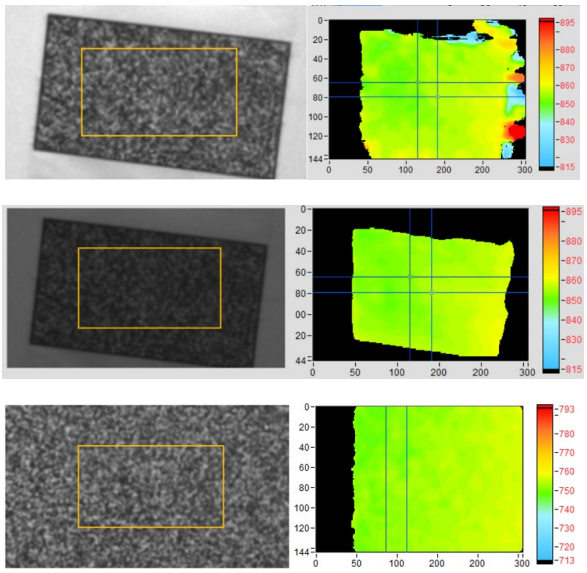


Figure 5. Examples of different set-ups. TOP: Not filling FOV, and not using projector. The depth map shows some noisy depth on the right side. This configuration works but is not recommend. MIDDLE: Not filling FOV, but changing depth setting from default (high density) to High-Accuracy, with manual exposure. The noisy depth is now removed. This is better than before. BOTTOM: Moving closer and filling the FOV completely with the target. This is the preferred configuration. The tilt of the target does not matter.

Now we turn to the actual commands. We have 3 new self-calibration commands in the Intel RealSense SDK 2.0 related to recovering the depth performance. Here is the high-level flow. For a more detailed API description please see the appendix:

```
// 1. Get pointer to current calibration table, before running on-chip calibration.
```

CalTableBefore = ***rs2_get_calibration_table();***

// 2. Runs the on-chip self-calibration routine that returns pointer to new calibration table.

CalTableAfter = ***rs2_run_on_chip_calibration();***

//3. Toggle between calibration tables to assess which is better. This is optional.

rs2_set_calibration_table(**CalTable*****);***

// 4. burns the calibration to FW persistently.

rs2_write_calibration();

The ***rs2_run_on_chip_calibration*** is a blocking call that has speed as an argument:

0 = Very fast (0.66 seconds), for small depth degradation

1= Fast (1.33 seconds), medium depth degradation. This is the recommended setting.

2= Medium (2.84 seconds), medium depth degradation.

3= Slow (2.84 seconds), for significant depth degradation.

As shown, there is the option for performing the self-calibration very fast, taking 0.66 seconds, or for slower speeds which are required when the calibration of the camera has deteriorated greatly. In this case calibration routine needs more time to search the larger parameter space to find the optimum calibration. To correct the largest range of errors, the recommended speed is "Slow" which takes ~2.8 seconds when using the 90fps mode, and otherwise scales in speed with frame rate.

The self-calibration algorithm is currently designed to correct for either "intrinsic" or "extrinsic" distortions, but not both at the same time. The intrinsic distortion comes about through microscopic shifts in the lens positions. Extrinsic distortions are related to microscopic bending and twisting of the stiffener on which the two stereo sensors are mounted. The user can select which mode to run, using the Mode selection in the function call. The "intrinsic mode" is selected by default and is normally the recommended mode. The main observable differences appear in the far edges of the FOV, for very impaired cameras. While both modes correct the central 20% of the FOV, they differ in how they correct the edges of the FOV.

So how is the user supposed to know whether to run intrinsic or extrinsic correction? For the type of normal small degradations, it does not matter. For larger degradations it depends on the type of physical impairment it most likely suffered from. For example, if an operator is mounting a camera module, the cause is most likely extrinsic, e.g., bending of the stiffener. If one of the lenses was touched, it is most likely intrinsic.

Once the calibration has been performed, the ASIC will leave the new calibration active, but will not have burned it automatically to flash memory. This allows the user time to confirm whether the performance is indeed improved. One way to validate the performance is to point to a flat textured surface and calculate the RMS Subpixel value, as described earlier, and compare the before and after values. The ***rs_set_calibration_table*** function allows for switching back and forth if needed to compare calibrations.

We should note that, in principle, this self-calibration function always finds the optimal calibration, and if it does not, then it returns an error. The exception will then be raised via standard ***rs_2_error protocol***.

One of the extremely powerful aspects of the self-calibration algorithm is that it also performs an on-chip "Health-Check", that does not require any specific visual target. Basically, once a self-calibration has been

run, the health-check number will indicate the extent to which calibration deviates from ideal. If the absolute value is below 0.25 then the Camera is essentially optimally calibrated, as shown in Table 1. Any value above this means the camera performance can be improved through re-calibration. Moreover, if the absolute value is larger than 1 then the unit is essentially non-functional without recalibration. The sign of the Health-check is mostly for diagnostic purposes, as it indicates the polarity of the distortion. This health-check number can also be very valuable to some users in allowing for a simple diagnostic that can be monitored over time.

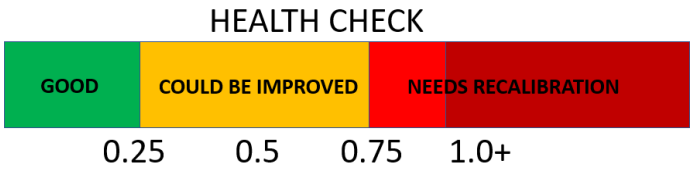


Table 1. The "Health-Check" indicator will return a measure of the need for recalibration. While units are operational for values below 1, it is optimal to have an absolute value below 0.25. This health-check does not need any special target.

In any event, provided the new depth performance is deemed to be better by the user RMS measurement or by the ASIC Health-Check number, it is now advisable to burn the calibration to the flash permanently. This is achieved by calling the ***rs2_write_calibration.*** After burning the new calibration to ASIC, we recommend calling ***rs2_run_on_chip_calibration*** one more time to confirm that health-check number is now low. You can run this at "very fast".

It is also important to note that there is always a way to recover the original factory calibration, if some bad calibration has inadvertently been written to flash. This is done by calling the function called ***rs2_reset_to_factory_calibration***.

We note one more important aspect. We developed a special calibration mode for the case of pointing the D400 Series cameras at a white wall that has no texture while the laser pattern projector is turned on. It turns out that this special case can be troublesome due to the semi-regular laser pattern of the D415 projector. This mode of operation can be selected under the "speed" settings and is called "white wall". It is also highly recommended that the "High Accuracy depth settings" be used during any white-wall calibration run. If not, then the ASIC will most likely return an error. However, to be clear, *we do not* recommend using a white wall for self-calibration for the D415. A textured surface with the projector turned off using a standard speed is the recommended method for on-chip calibration of D415 cameras. On-chip calibration of D435 cameras does not require the "white wall" mode regardless of target type and projector setting; one of the standard speed settings is recommended under most conditions.

2.4 Extending to Beyond Simple Flat Textured Target

We have seen how the new self-calibration technique works under controlled and ideal conditions. Now we look at deviations from this. It turns out that the self-calibration is fairly robust and actually works quite well under a very wide variety of conditions. The primary constraint is that there should be good texture in the scene. Another way to say this is that if a healthy D400 depth camera has a good depth map with a high fill ratio while looking at the scene, then the scene will probably be well suited for self-calibration. The scene does not need to be flat. It can be static or moving. The texture can be applied by a projector, or it can be a natural part of the scene. It works in complete darkness (with a projector), or outside in the brightest sunlight.

Figure 6 shows a non-exhaustive set of examples of scenes that work quite well. Scene "A" is the ideal scene we have described before, of a well textured flat target with the projector turned off. Scene "B" is a flat white wall with a projector turned on. Scene "C" is a textured carpet, which is usually easy to find. Scene "D" is a cluttered desktop. Scene "E" is a flower on a table-top. Scene "F" is some rocks on a patio, outside

in bright sunlight. Scene “G” is a face indoor with projector turn on, and “H” and “I” are faces indoors and outdoors without projector. These are non-exhaustive or specific and are only meant to serve as examples we have tried and confirmed work well.

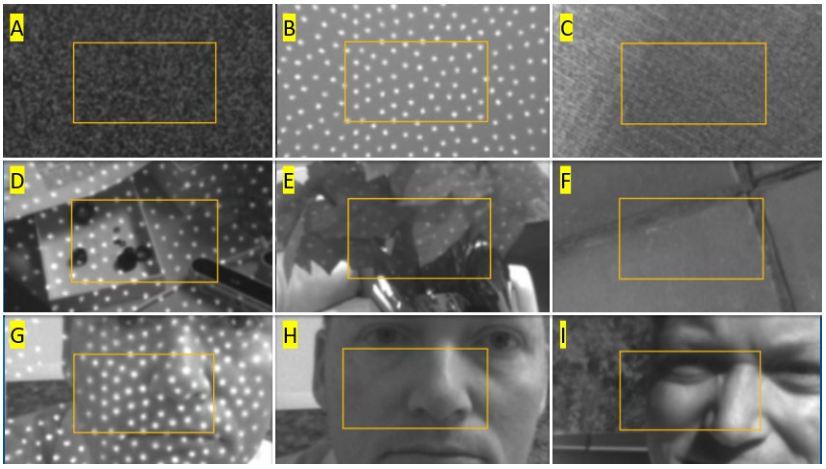


Figure 6. A set of scenes that have successfully been used for self-calibration, ranging from the ideal flat textured target without projector illumination (A), to scenes with projector on (B, D, E, G), to outdoor scenes in bright sunlight (F, I).

3. DEPTH ACCURACY:

We turn now to the new functions to improve the *depth accuracy* of the Intel RealSense cameras D400 Series. The depth accuracy relates to measuring exactly the correct distance. Figure 7 shows two measurements of measured distance vs ground truth distance. This result was collected by translating an Intel RealSense depth camera D435 away from a wall, while carefully measuring the average distance to the wall with the depth camera. Note that these tests were run with the Advanced Depth parameter “A-factor” set to 0.08. Please see separate white paper on how this improves the subpixel linearity. In the left figure there is and offset of about 1.2% in measured range vs true range. Also, the slope is slightly different. In the right image, the two curves overlap, and the error (shown in the lower graph) is seen to vary by about +/-0.2%.

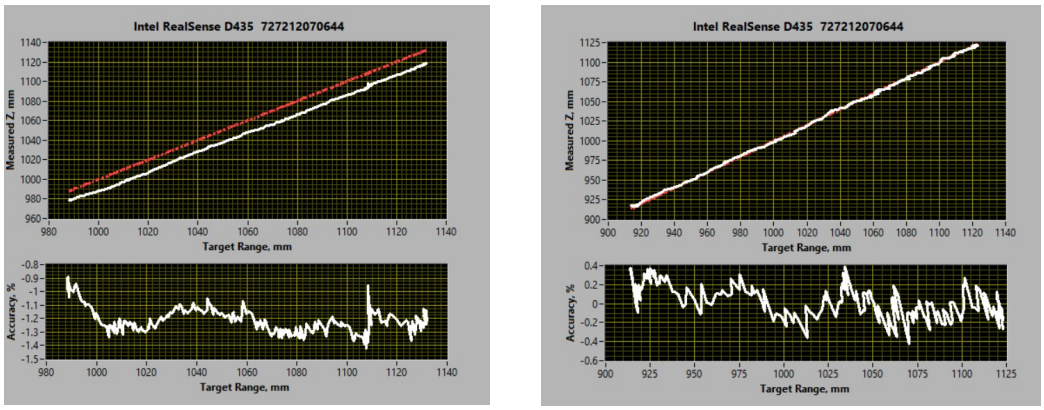


Figure 7. The measured distance vs ground truth (aka Target Range). LEFT shows a calibration with a BIAS of -1.2% near 1m distance. RIGHT shows the corrected calibration where the absolute accuracy is mostly limited by the noise of the measurement.

To correct the absolute distance measurement usually involves correcting both the slope and offset of the depth measurement vs distance. We introduce a new on-chip function we call “Tare” (rhymes with “tear” or “bear”). This is a function commonly used when measuring weight with a scale, as shown in Figure 8.

Figure 8. The "tare" function is commonly used to remove bias and reset to a known value. When weighing objects, this means it is possible to place a bowl on a scale, "tare" to set to zero, and now measure the true value.

The "Tare" command for the Intel RealSense depth cameras D400-series requires that the user tells the ASIC the known ground truth measurement as measured as the perpendicular distance from the left camera lens. This ground truth can be measured ahead of time in several different ways. For example, the user can place the camera into a fixture in which the ground truth distance has been carefully measured. Alternatively, the ground truth can be calculated on-the-fly using simple computer vision techniques, like looking at a target of known dimensions. One such target could be the target in this appendix. Various algorithms can be applied to then optically measure this distance, but that is beyond the scope of this paper.

Before we proceed, we need to mention that one should not underestimate the challenge associated with making a good ground-truth distance measurement. If using an external range finder or "distance meter", make sure to reference the distance to the plane of the left imager, which is 2mm recessed from the front of the D415 or D435 front windows. Make sure to use a laser range finder with the proper resolution, that itself has been properly calibrated and certified. Many off-the-shelf range finders have accuracy specs of 1/16" or +/-1.5mm. Finally, make sure that the target is completely flat and oriented perpendicular to the camera's pointing direction. Also, make sure that the depth map is good (possibly by running On-Chip Self-calibration first and burning results to ASIC). *If the original depth map is not good and flat, then you cannot expect a good tare result*. For best results use a textured patterned wall (or paper target), and not a flat white wall with projector turned on.

Turning now to the software commands, it follows the same logic as for self-calibration. The main function is called ***`rs2_run_tare_calibration().`***

Again, the process is to start by calling ***`rs2_run_tare_calibration`*** with the argument being the known depth. Note that the depth is being calculated on-chip is the average depth to the full field-of-view.

Even though the Tare function is performed at a single distance, the function will generally help improve the accuracy across the whole depth range. This command starts an on-chip calibration routine that will take from 30ms to 1 second to complete. This is also a blocking call.

To Burn the result to Flash, the final command is used: ***`rs2_write_calibration.`***

We highly recommend that before running Tare-calibration, that the camera depth presets be changed to "High Accuracy Depth" mode. This will allow the function to complete successfully with the default chip setting of "Very High" accuracy. After the Tare has been run, the camera depth settings can be reverted back to whatever was used previously.

4. USING INTEL REALSENSE VIEWER:

To make it easier to get familiar with the capabilities of these new calibration features, we have added them to the Intel RealSense Viewer and Depth Quality Tool, as shown below.

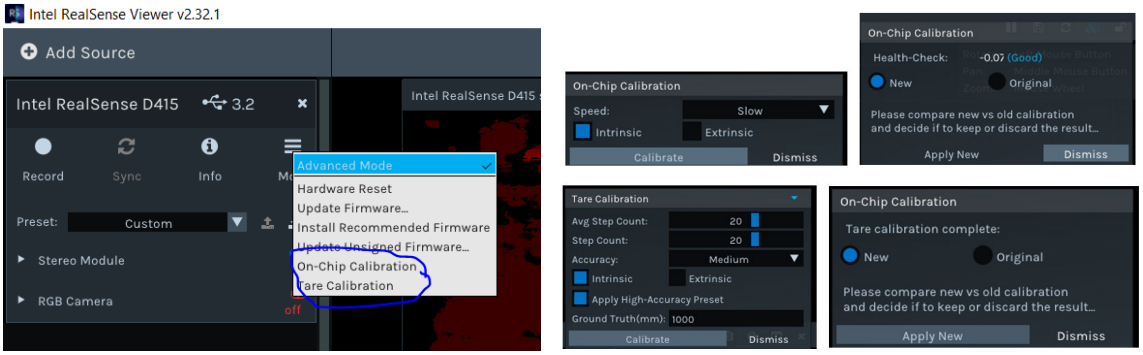


Figure 9. The new on-chip self-calibration functions can be accessed in the Intel RealSense Viewer app. For on-chip calibration, the speed and mode (intrinsic/extrinsic) can be selected. Once the calibration routine has been run, it will provide the result of Health-Check and ability to toggle between old and new calibration table. "Apply New" will burn it to flash on the camera and "Dismiss" will cancel the operation and keep the original calibration. Similarly, Tare calibration can be run in either intrinsic or extrinsic mode. A ground truth value needs to be entered, and Accuracy level and step counts are also selectable, but default settings are recommended for most cases.

To activate the functions, first set up the camera as described in Section 2.3. Once the on-chip calibration has been selected, the Viewer will automatically select the 256x144 resolution during calibration and will return to full resolution afterwards. During the calibration the left Monochrome (or RGB) imager can either be on or off. The Health-Check number is returned as the normalized "Calibration Error", where an absolute value of less than 0.25 is acceptable. The user can subsequently toggle between the original and new calibration before deciding on whether to apply or dismiss. Tare calibration is run in a similar manner, with set-up described in section 3. With Tare calibration selected, intrinsic/extrinsic mode can be selected and a ground truth distance to the flat target must be entered. All other Tare settings may be left with their default values in most cases. Once the Tare process is completed, the user can choose between the new and original calibrations.

5. LIMITATIONS

As stated earlier, the on-chip self-calibration functions are quite robust and work well under a variety of conditions. As a general guideline, conditions under which "good" depth would be expected in the central 20% ROI, should provide good self-calibration results. The Tare function has the added requirement of a flat surface of a known distance over the same 20% ROI. Although depth noise is not required to be small for Tare to work, it is nonetheless recommended to run the on-chip self-calibration before running Tare.

Despite the robustness of self-calibration, we have identified a few specific scenarios that *may* lead to warnings (a.k.a. failures) and these are to be avoided. In almost all cases, the correct error message will prompt the user to adjust the conditions (speed, scene etc.), and the calibration will be prevented from updating. However, though very rare, there are extreme corner cases where we have observed that the function completes successfully but results in a calibration worse than the original. We will describe both scenarios in more detail below:

A. Self-Calibration & Tare Errors prompting a retry:

The most likely scenario for an error message for on-chip self-calibration is when there are not enough valid depth pixels (i.e., fill ratio too low). This can usually be remedied by ensuring that the projector is on and the scene does not include shiny/specular or completely black/absorbing objects. Failures can also occur when the self-calibration algorithm fails to converge which is most likely to occur for severely degraded cameras, or on the D415 pointed at white wall but "White Wall" mode is not used. Figure 10 shows examples of such error messages.

Tare calibration should very rarely fail to converge, but 3 scenarios where this is possible are i) when "high accuracy" depth setting is not used, ii) when the camera calibration is severely degraded such that there is very low fill ratio or very high depth noise, or iii) the Z error is very

large such that the input ground truth is very far from the current reported depth requiring a significant change in calibration parameters. The corresponding error message is shown in Figure 10.

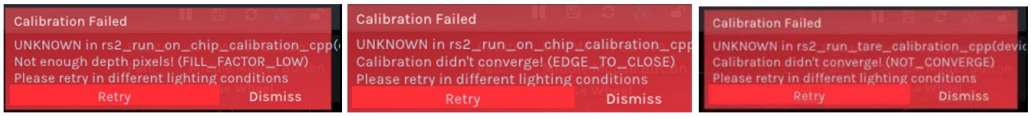


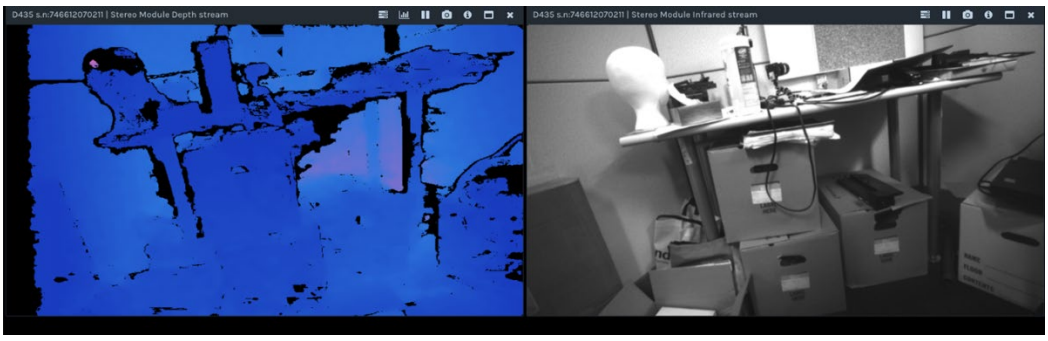
Figure 10. A scenario where on-chip self-calibration may fail due to insufficient texture. The central 20% ROI is mostly invalid leading to the corresponding error message (LEFT). In most cases, turning the projector on will provide the necessary texture when the scene is lacking. Self-calibration or Tare may fail to converge in cases of very large initial error resulting in error messages shown (MIDDLE & RIGHT).

B. Incorrect Result

An incorrect result, where the calibration error determined by the self-calibration or tare function is substantially different from the actual error, requires specific scenarios to occur. We have observed that self-calibration can return a bad health-check value in cases where the *depth fill ratio changes significantly during the scan*. While this is rare, it can occur, for example, in a scene with fast motion of camera, on scenes that have highly varying texture, and with projector turned off. All these conditions need to apply simultaneously, so simply ensuring one of them is good will lead to good results. The failure is illustrated in Figure 11.

Another scenario where self-calibration can possibly return a bad health-check number, is when pointing at a glossy surface with the projector turned on, so this scenario should also be avoided.

The only known scenario where the Tare function can result in a significant Z error vs the entered ground truth is when the function is implemented such that the “High Accuracy Depth” preset is not used as described in section 3. While there are some scenarios that do not require this mode of operation (e.g., a very well textured target), it is generally recommended to ensure accurate results.



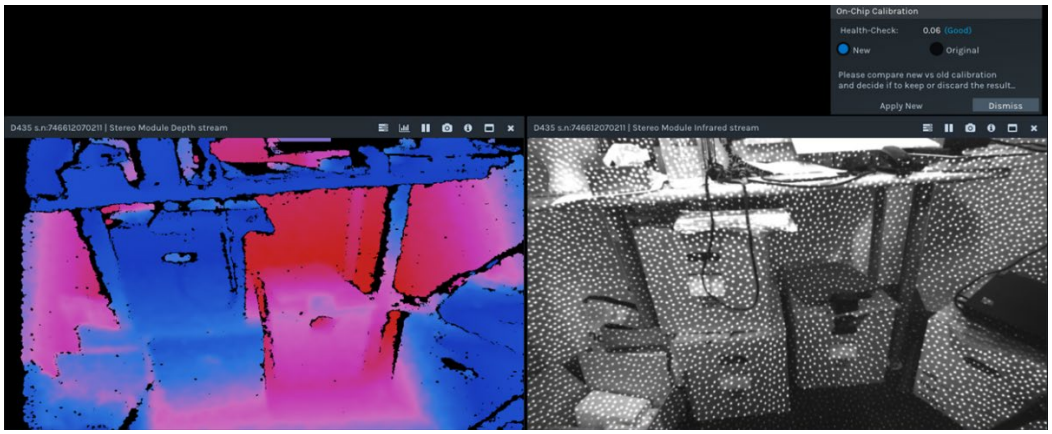


Figure 11. A scenario where on-chip self-calibration may result in a bad calibration. A scene with variable texture, projector off, and where camera or scene is moving while self-calibration is run (top) can lead to an incorrect health-check error and degraded depth. If self-calibration is run in a similar manner but with projector on, providing stable texture, then a more accurate health-check number will be returned, and optimal performance is obtained (bottom).

Please use the latest SW available (Ref#1) and check the Errata for other existing issues (Ref#2).

6. CONCLUSION:

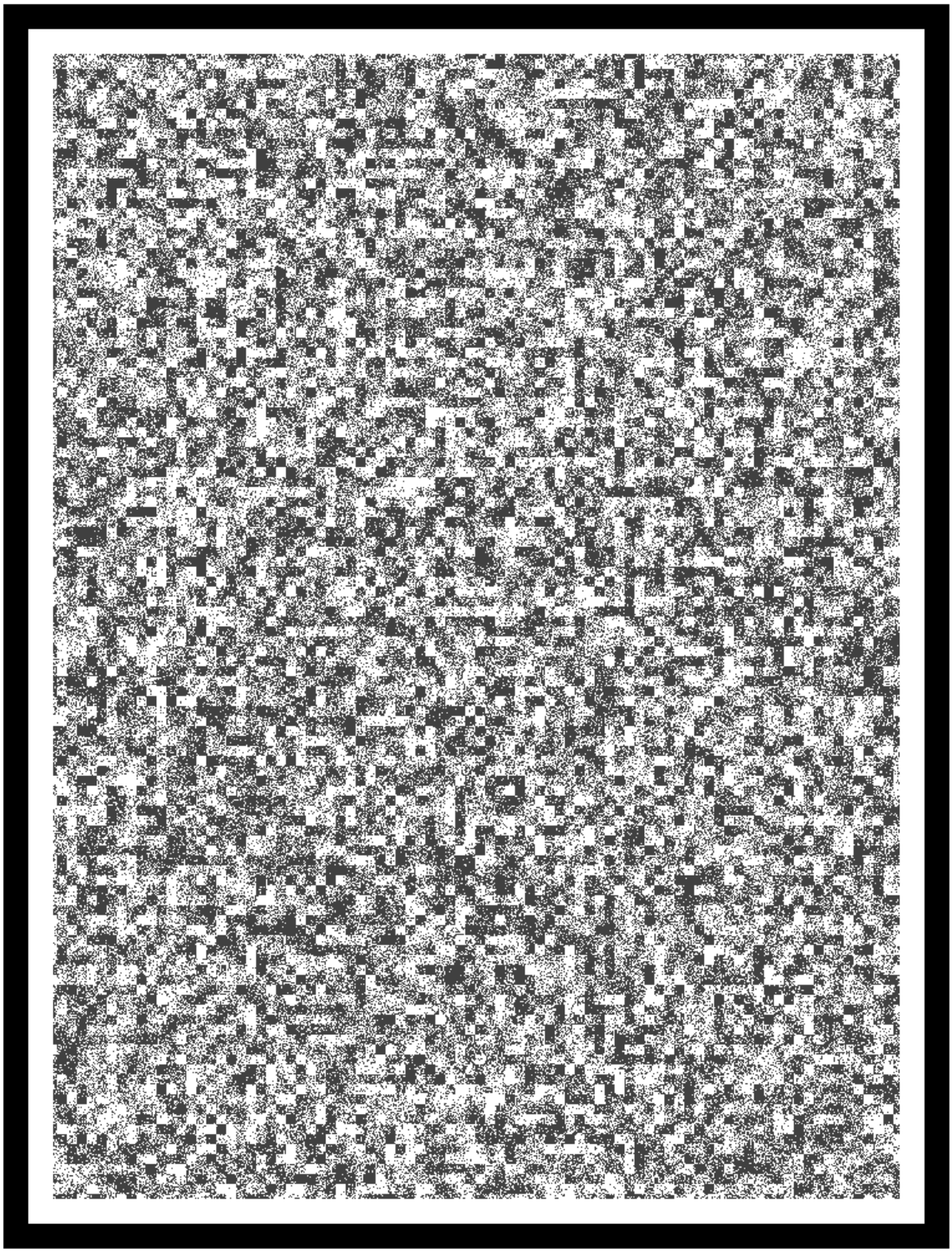
We have presented methods for performing automatic self-calibration of Intel RealSense™ depth cameras D400-series. The new methods are on-chip which means that they are simple, fast, consume little power, and work across all operating systems and platforms. These methods are generally meant to be used to fine tune the performance and are not intended to perform a complete factory calibration from the grounds up. These functions address some of the most common sources of calibration degradation, but may not improve all units, which is why we have separated the functions into two commands: One for performing the calibration and one for burning the calibration to flash. This allows the user to first validate the new calibration before making the decision to keep or discard the new settings. However, the new Health-Check number that is generated is a very accurate and repeatable measure of the calibration quality of the existing calibration and can be relied on without requiring user intervention or flat wall characterization.

The new on-chip calibration techniques are shown to be quite robust. While they work best with a well textured flat target, they have also been shown to work under less controlled environments, even completely target-less. We also emphasize that the on-chip routines inherently use the full field-of-view of the sensor, which is why we *strongly recommend* using the new windowed 256x144 90fps mode which zooms into the central 20% of the image and allows the user to point at specific good regions of a scene.

References:

1. Intel RealSense SDK 2.0: https://github.com/IntelRealSense/librealsense
2. Intel RealSense Camera firmware and corresponding Errata: https://dev.intelrealsense.com/docs/firmware-releases

Appendix A: Example Target



Appendix B: More detailed on-chip calibration C API description.

A. Running on-chip calibration:

```
const rs2_raw_data_buffer* rs2_run_on_chip_calibration(rs2_device* device, const
void* json_content, int content_size, float* health,
rs2_update_progress_callback_ptr callback, void* client_data, int timeout_ms,
rs2_error** error);
```

This starts on-chip calibration. This is a blocking API that will return with success, failure or timeout. When successful, the new calibration table object will be returned. When failing, an exception will be raised via standard rs2_error protocol. Exceptions can be 1. edge to close, 2. not enough fill rate, no converge, 3. device disconnected, 4. protocol error, 5. not supported, or 6. timeout.

The content of the return `rs2_raw_data_buffer` object can be accessed via existing **`rs2_get_raw_data_size`** and **`rs2_get_raw_data`** APIs (for example if the user wants to save calibration results to disk). It should be deleted using **`rs2_delete_raw_data`**. In order to toggle between different calibration, the pointer returned from the **`rs2_get_raw_data`** can to be passed as input argument to the function call **`rs2_set_calibration_table`**. Finally, in order to "burn" the new calibration persistently to memory, the `rs2_write_calibration` function needs to be called.

The returned *health* is a signed value indicating the calibration health.

The *timeout_ms* is 15000 msec by default, and should be set to longer than the expected calibration time as indicated in this document.

`rs2_update_progress_callback_ptr` and `client_data` are optional call-backs and can be set to zero.

The *json_content* and *content_size* are the calibration parameters passed in JSON format. Content size is the count of all characters in the json string (so 49 in example below).

The JSON format is for example:

```
{
 "speed": 3,
 "scan parameter": 0,
 "data sampling": 0
}
```

Passing NULL to json_content or 0 to content_size will run the calibration with default recommended settings.

The speed can be one of the following values: Very fast = 0, Fast = 1, Medium = 2, Slow = 3, White wall = 4, default is Slow.

The *scan_parameter* is 0 for Intrinsic calibration correction (default) or 1 for extrinsic correction. The *data_sampling* default is 0 which uses a polling approach that works on Windows and Linux. For Windows it is possible to select 1 which is interrupt data sampling which is slightly faster.

B. Running tare calibration:

```
const rs2_raw_data_buffer* rs2_run_tare_calibration(rs2_device* dev, float
ground_truth_mm, const void* json_content, int content_size,
rs2_update_progress_callback_ptr callback, void* client_data, int timeout_ms,
rs2_error** error);
```

Similar to on-chip calibration. This will adjust camera calibration to correct the *absolute distance* to the flat target. User needs to enter the known ground truth to a flat target that is the the size of zoomed field of view (256x144).

The *ground_truth_mm* is the ground truth in millimeters in range 2500mm to 2000000mm.

The json content contains the configuration parameters, but we recommend setting json_content to nul, and set content_size to 0, so that default parameters are applied.

The json string is:

```
{
"average_step_count": 20,
"step_count": 20,
"accuracy": 2,
"scan_parameter": 0,
"data_sampling": 0
}
```

The *average_step_count* is the number of frames (from 1-30) that are averaged to improve the noise.

The *step_count* is the max iteration steps (between 5 and 30) that are used in the optimization search. Usually a solution is found within 10 steps.

The *accuracy* is the subpixel accuracy level, and the value can be one of: Very high = 0 (0.025%), High = 1 (0.05%), Medium = 2 (0.1%), Low = 3 (0.2%), Default = Very high (0.025%).

The *scan_parameter* is 0 for Intrinsic calibration correction (default) or 1 for extrinsic correction. The *data_sampling* default is 0 which uses a polling approach that works on Windows and Linux.

The optional *callback* and *client_data* can both be set to zero. The *timeout* default is 5000ms.

C. Reset calibration:

There are three ways to reset calibration.

A. After a self-calibration or Tare, the calibration is only stored temporarily in the camera. Stopping and restarting streaming reverts to the previous calibration state. Also, disconnecting the camera and restarting will also reset the calibration.
B. One can call the **`rs2_get_calibration_table`** before each tare- or self-calibration run. That way it is always possible to revert to the previous calibration using the **`rs2_set_calibration_table`** and **`rs2_write_calibration_table.`**
C. It is possible to restore to factory calibration, which is permanently stored in the camera after factory calibration**.**

```
void rs2_reset_to_factory_calibration(const rs2_device* device, rs2_error** e);
```

Appendix C: On-chip calibration Python API:

A. **Preparing for calibration:**
Starting the pipeline into a mode compatible with on-chip calibration can be done as follows:

```
import pyrealsense2 as rs2
pipe = rs2.pipeline()
cfg = rs2.config()
cfg.enable_stream(rs2.stream.depth, 256, 144, rs2.format.z16, 90)
dev = pipe.start(cfg).get_device()
```

Once started successfully, dev object can be casted to `rs2.auto_calibrated_device` by calling:

```
cal = rs2.auto_calibrated_device(dev)
```

B. **Running On-Chip calibration:**
`rs2.auto_calibrated_device` class allows to invoke on-chip calibration using the following blocking call:

```
def cb(progress):

    print(“.”)

res, health = cal.run_on_chip_calibration(timeout_ms, json, cb)
```

This method has similar signature and behavior to C API **`rs2_run_on_chip_calibration.`**

C. **Running Tare calibration:**
`rs2::auto_calibrated_device` class allows to invoke tare calibration using the following blocking call:

```
res = cal.run_tare_calibration(ground_thruth, timeout_ms, json, cb)
```

This method has similar signature and behavior to C API **`rs2_run_tare_calibration`**.

D. **Setting / Resetting the calibration:**
Assuming calibration completed successfully, new calibration table can be applied to the current streaming session using `cal.set_calibration_table(res)`

Saving new calibration permanently to the device can be done via `cal.write_calibration()` after calling `set_calibration_table`. It is also possible to reset the device to its factory calibration using `cal.reset_to_factory_calibration()`

Please see `depth_auto_calibration_example.py` under `wrappers/python/examples`

Appendix D: On-chip calibration C++ API:

A. **Preparing for calibration:**
Starting the pipeline into a mode compatible with on-chip calibration can be done as follows:

```
rs2::pipeline pipe;
rs2::config cfg;
cfg.enable_stream(RS2_STREAM_DEPTH, 256, 144, RS2_FORMAT_Z16, 90);
rs2::device dev = pipe.start(cfg).get_device();
```

Once started successfully, dev object can be casted to `rs2::auto_calibrated_device` by calling:

```
rs2::auto_calibrated_device cal = dev.as<rs2::auto_calibrated_device>();
```

B. **Running On-Chip calibration:**
`rs2::auto_calibrated_device` class allows to invoke on-chip calibration using the following blocking call:

```
float health;

rs2::calibration_table res = cal.run_on_chip_calibration(json, &health, [&](const
float progress) { /* On Progress */ });
```

This method has similar signature and behavior to C API **`rs2_run_on_chip_calibration`**
However, unlike its C counterpart, it receives JSON parameters via C++ `std::string`, can use C++ 11 anonymous function as the progress callback (as well as regular function pointer) and returns object of type `rs2::calibration_table` that does not require explicit deinitialization. In case of an error, this API will throw an exception of type `rs2::error`.

C. **Running Tare calibration:**
`rs2::auto_calibrated_device` class allows to invoke tare calibration using the following blocking call:

```
rs2::calibration_table res = cal.run_tare_calibration(ground_thruth, json,
[&](const float progress) { /* On Progress */ });
```

This method has similar signature and behavior to C API **`rs2_run_tare_calibration`** with same notes from the previous section apply.

D. **Setting / Resetting the calibration:**
Assuming calibration completed successfully, new calibration table can be applied to the current streaming session using `cal.set_calibration_table(res);`

Saving new calibration permanently to the device can be done via `cal.write_calibration();` after calling `set_calibration_table`. It is also possible to reset the device to its factory calibration using `cal.reset_to_factory_calibration();`

Appendix E: Self-calibration with LabVIEW.

A new “Hello World” example VI was added, that shows how to implement Self-Calibration and Tare in LabView. A few new sub-VIs were added. Note also that the user is guided to be in 256x144 resolution mode.

RS3_Get_Calibration_Table.vi: Call this first to get a pointer to the existing calibration table.
RS3_Run_OnChip_Calibration.vi: Run the self-calibration and return the Health-Check number and a pointer to the new calibration table.
RS3_Get_Raw_Datasize.vi and RS3_Get_Raw_data.vi: Use these to update the calibration table, and allow the user to toggle between the old and new calibration table.
RS3_Run_OnChip_Tare.vi: Run the tare calibration and return pointer to new calibration table.
RS3_Burn_Calibration_table.vi: Write new calibration table permanently to ASIC.
RS3_reset_Calibration_table_to_Factory.vi: Allows to recover to the original factory calibration stored in ASIC.