Time-of-Flight Sensor Calibration

Calibration Software

Rev. 0.60b (Release 6.0)

Description

This document provides overview of calibration software, a simple procedure on how to run those applications as well as a description of the source structure. This release contains the minimal set of sources required to build and run the calibration application. This is delivered in source form in the hope that it can help module integrators to quickly develop their own calibration equipment for time-of-flight (ToF) camera module production.

Contents

1. Terminology

Table 1: Terminology

Arduino Turntable	Externally controlled single degree of freedom robot used to rotate the camera during the calibration process
Confidence	L1 norm (i.e. abs(I) + abs(Q)) of the time-of-flight signal.
Device	Normally referring to the IU316-Minikit module with its SSP500 interface board
SSP500	MIPI to USB interface board
Handoff files	Files that contain all necessary register value settings, for the sensor and other devices on the module, in order to configure the device into a particular ToF mode of operation, e.g. a mode chosen to run the calibration process
IMX316	Name of ToF sensor
IU316	Name of the camera module
JSON	A text based serialization format. http://www.json.org/
MessagePack	A binary serialization format. https://msgpack.org/
Miniconda	Python installation that contains the conda Python package manager. https://conda.io/miniconda.html
Mode	The set of register values that fully define the configuration of the sensor and all other module components.
Phase	Phase of the time-of-flight signal.
PRV number	Production Revision Number. It defines hardware and software version numbers.
Pycharm	Python development IDE
Rawphase	Phase of the raw time-of-flight signal (prior to applying calibration corrections).
UID	A unique 16 bits unsigned integer used as an identifier for a mode.
Backend (configuration)	Backend connection between the sensor and the usb connection, in case of IMX316 backend= SSP500
Cat Tree Calibration	The calibration method supported by this application. In addition to a turntable, this calibration method requires the Cat Tree Hardware to perform the complete calibration
Skv	Internal movie format developed at Sony Depthsensing Solutions (previously softkinetic)

2. Conventions used in this document

This document uses the following typographical conventions:

- `constant width`: for program listings and inline code entities such as python identifiers or json configuration entries, filenames and file extensions
- **`constant width bold`**: for commands or text that should be typed by the user
- *`constant width italic`*: for text that should be replaced by user-supplied values
- *Italic*: for new or special terms, URLs
- ***Bold italic:*** Indicates menu entries or button labels in GUI applications, that user should interact with

3. Overview

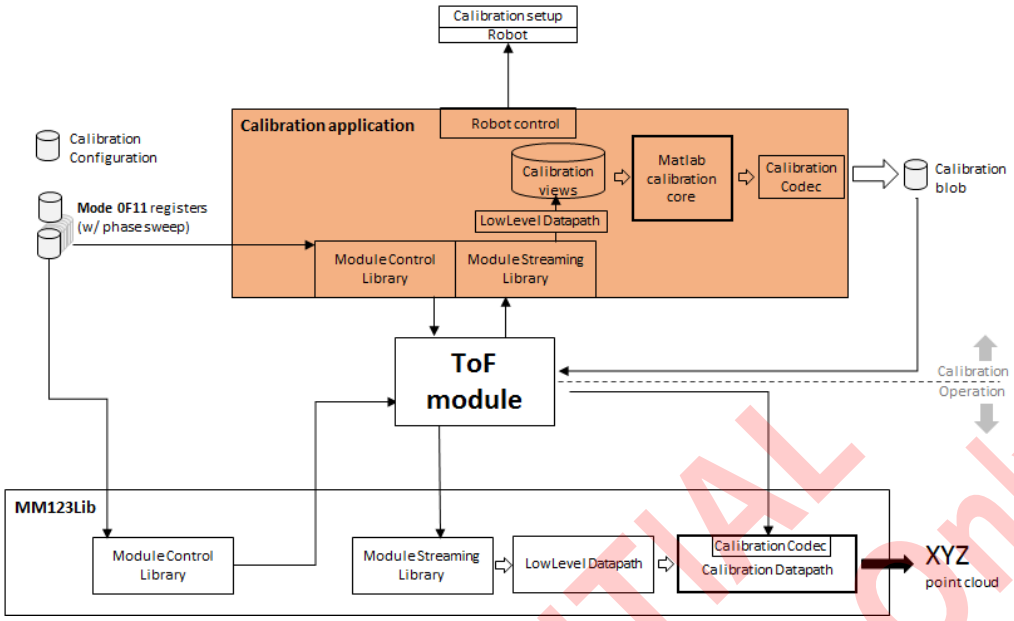


Figure 1: Overview of the deliverable (calibration application and IU316-Minikit driver)

This deliverable contains the calibration application (see Figure 1).

The calibration application is controlling the flow of the calibration process: it acquires camera module data, operates the physical calibration setup (e.g. the turning table), computes calibration parameters, and writes this output into the module's EEPROM, as well as into a selection of external formats, including e.g. a `.json` file, which can be visually inspected by the user. Every part of this application is written in the Python language with some C extensions, except the module which computes the calibration output (called the MATLAB core), which is written in the MATLAB language.

Important: A recent MATLAB version is to be used. See exact version requirements in next section. The calibration application will not work with earlier versions of MATLAB.

The following sections include instructions on how to deploy and start the calibration application and calibrate a IU316 module with it. An outline of both the calibration application will be also provided.

4. Installing Calibration Software Deliverable

In this section we describe how to deploy and run the calibration app. You should first make sure all required third party software is installed (see Prerequisites section), before continuing with the installation and running the calibration application (see Installing and running the calibration application section).

4.1 Prerequisites

The following tools need to be installed to be able to deploy and run the calibration application:

- MathWorks MATLAB, from *https://www.mathworks.com/*. Version 16B is required. Please note that the calibration application will not work with earlier versions of MATLAB e.g. 16A or 15. The calibration requires the following toolboxes:
    - Computer Vision System Toolbox
    - Curve Fitting Toolbox
    - Image Processing Toolbox
    - Statistics Toolbox
- Miniconda, from *http://conda.pydata.org.io/miniconda.html* with Python 2.7 (latest version). Make sure to get the 64-bit version.
  Note: During the installation wizard, be aware of 1. Install for all users and 2. check the option to add to the PATH.
- The Visual Studio 2012 redistributables, from *https://www.microsoft.com/en-us/download/details.aspx?id=30679*.
- The SSP500 interface board Windows drivers. (see *Installing SSP500* section)
- PyCharm IDE, from *https://www.jetbrains.com/pycharm/*. The free Community Edition is enough. Note that this component is not needed, if one wants to just run the calibration application. Nevertheless, it is useful to explore the code.
- Arduino drivers (see ARDUINO IDE + Windows Driver section). We provide two Arduino drivers:
    - One for a single stepper motor to control a *turntable*, to use for the phase stepping calibration method
    - One for the elevation platform of the 2DoF stage, to use for the table sweep calibration method

4.2 Installing and running the calibration application

4.2.1 Installing the calibration application

- Unzip the `IMX316_MI_calib.zip` into a local folder of your choice.
- Open a prompt (terminal window) in this folder.
- Create a conda virtual environment by typing the following command:

```
conda create --name iu316_calibration --file conda_requirements.txt
```

Note: The *conda* command belongs to the conda Python package manager, which allows you to create isolated python environments. The *conda* command appears after you have installed Miniconda (see 4.1 section about prerequisites). More documentation can be found in the conda documentation (https://conda.io/docs/index.html).

4.2.2 Running the full live calibration application

- Open a prompt (terminal window) in this folder.
- Activate the environment:

```
activate iu316_calibration
```

> Note: activate is part of conda Package Manager, and its documentation can be found here: *https://conda.io/docs/glossary.html#activate-deactivate-environment*.

- Set the PYTHONPATH to the current directory:

```
set PYTHONPATH=./
```

> Note: This operation effectively adds the *softkinetic* package folder to the Python path. This package folder contains the calibration application.

- Open the file `test_data/iu316_minikit/run.py` and edit it in the following way:
  - Replace the line where the mode is defined with `“mode = Mode.full_calibration”`.
- Open the file `test_data/iu316_minikit/full_app.json` and edit it the following way:
  - Replace the value of `"handoff_folder"` with the full path of the directory where this file is located.
  - Replace the value of `"rootfolder"` with the full path of the directory where the output of the calibration will be placed.
- To run the calibration application, type the following command and the application should start

```
python test_data/iu316_minikit/run.py
```

4.2.3 Running the offline calibration application

It is possible to perform the calibration application without a camera available, provided we have pre-recorded MAT files from the Cat Tree Setup and the *skv* files containing footage of the checkerboard, with a specific naming convention:

- `cat_tree_data.mat`
- `view_0_phase_000.00.skv`
- `view_1_phase_000.00.skv`
- `view_2_phase_000.00.skv`
- `view_3_phase_000.00.skv`

These are videos of 25 frames each, using each of the 4 views, with the camera rotating 0, 90, 180 and 270 degrees

- Activate the environment:

```
activate iu316_calibration
```

> Note: activate is part of conda Package Manager, and its documentation can be found here: *https://conda.io/docs/glossary.html#activate-deactivate-environment*.

- Set the PYTHONPATH to the current directory:

```
set PYTHONPATH=./
```

> Note: This operation effectively adds the *softkinetic* package folder to the Python path.The python path is the list of directories python looks for when importing a module. This package folder contains the calibration application.

  - Place the MAT file in the folder `test_data/iu316_minikit/Viewer_CEC` and delete the file named “place_here_the_mat.txt”
  - Place said videos in the folder `test_data/iu316_minikit/configurations/Offline_Calibration/ID_3160_EE01_30FPS_100MHz_DT25_570us` and delete the file named “place_here_the_skvs_and_remove_this_file.txt”
  - Run the rawdata_viewer_offline.py and copy the processed cat tree outputs from the results.csv output file
  - Paste the output to the “per_mode” of the selected mode in the Mode JSON file (mode_cattree_offline.json)
- Open the file `test_data/run.py` and edit it in the following way:
  - Replace the line where the mode is defined with “`mode = Mode.offline`”.
- Open the file `test_data/iu316_minikit/skv_app.json` and edit it the following way:
  - Replace the value of `"handoff_folder"` with the full path of the directory where this file is located.

- Replace the value of `"rootfolder"` with the full path of the directory where the output of the calibration will be placed.

- To run the calibration application, type the following command and the application should start

```
python test_data/iu316_minikit/run.py
```

4.3 Running the calibration software

1. **Run** the `run.py` script to start the calibration application in full live calibration mode. The following window should pop up.

Figure 2: Calibration application GUI displays the raw phase and confidence streams when the camera module is connected.

2. **Connect** a module to the host PC if this is not already done. This will start the stream.
3. Once the device is detected the application will automatically configure the device in the mode to be calibrated.
4. Pressing the ***'Run Test'*** button will start the process of calibration.
   Note:
   - Be careful that the turntable will be moving during this process. It will rotate 2 times 90 degrees clockwise from its starting position, and then will return counter-clockwise to its starting position. In case of table sweep calibration, the elevation platform will also move.
   - Both the ***'Run Test'*** and ***'Reset'*** buttons will be greyed out and the ***'Ready'*** label will be replaced with a ***'Testing'*** label.
5. Once the calibration is done, a green ***'Pass'*** or red ***'Fail'*** label will be displayed in GUI. When 'Pass' is displayed, this means that the calibration completed successfully (e.g data acquisition and model fitting). Mind that no performance measurements are made.
   Additional output information may be found in the console.

Figure 3: Calibration application GUI after successful calibration process. The status label is green and reads *Pass*.

4.4 Output of the calibration process

After a successful calibration process the calibration output is:

- written into the module's EEPROM, and
- saved to disk on the host PC in a subfolder of the `"rootfolder"`: as a user readable `.json` file, and/or as a machine readable `.mp` messagepack file.

4.5 Known limitations of the calibration application

- If the app is closed *while* a module is connected, then the module may be left in “streaming state”. To reset it, simply unplug and re-plug the USB cable.
- In case the app is run while the power to the turntable motor is left unconnected, then the app will run without turning the camera for different views, however the app will not give any error message, even though the calibration quality will be sub-standard in this case.

5. Calibration Application Execution Flow

This section will provide details on the execution flow of the calibration application as well as pointing to key locations in the source code so that the reader can find more easily the desired entry points. The process that will be outlined in this section is also described in the IMX316_AppNote_CameraCalibration(E)_28112017.pdf document in section 4.2.

The process of calibrating a module is split in 2 steps. During the first step, data required for calibration is collected while in the second step the data is post-processed and the calibration written to the device. Most of the inputs to the calibration process comes from a user supplied JSON file. More details about the input JSON file can be found in the JSON files specifications section.

Please note that calculation flow (algorithm, any other details) written here is just an example of various possible calibration methodologies. Depending on a sensor to be calibrated, appropriate calibration methodology should be developed, taken and applied.

5.1 Data capture

The calibration application supports the cat tree CEC method. The cat tree CEC method requires data capture at an automated rotating turntable as well as the cat tree setup.
The following sections present the flow for both setups.

5.1.1 Data Capture at the Cat Tree Setup

Data capture flow at the Cat Tree setup is presented in the diagram (Figure 4):

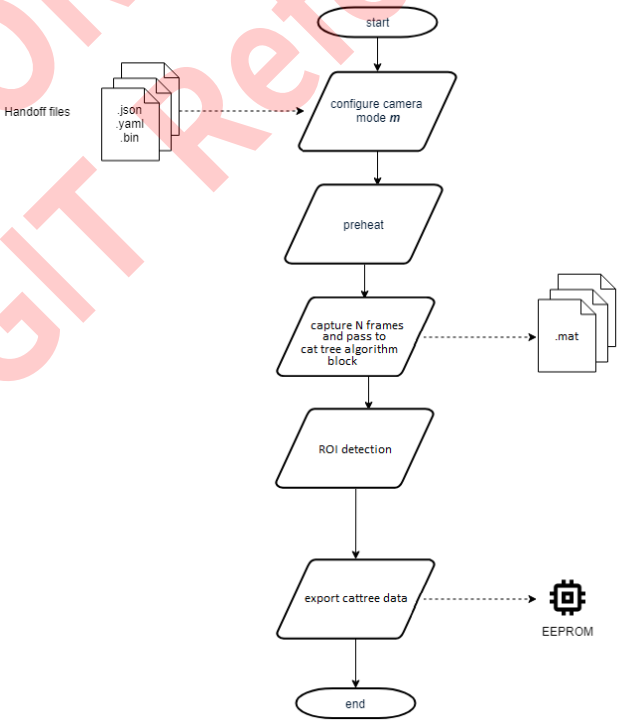


Figure 4: Data Capture @ Cat Tree Setup

1. The module is configured in the mode which to be calibrated
2. The camera module is preheated for a predefined time in order to stabilize the temperature
3. N data frames are captured and passed to the cat tree algorithm block
4. The Cat Tree algorithm block then processes the raw phase and confidence data and stores the required data in the EEPROM using the Cat Tree Codec

5.1.2 Data Capture Flow at the Turntable

Initially, the python instance collects all the data necessary for the MATLAB scripts to generate a suitable calibration for the module. Parameters for the data capture are defined in the input JSON file (see the JSON files specifications section for more details). The key parameters are:

- The mode name to calibrate, referred as *'the mode to be calibrated'* in the rest of this document.
- The PRV number to which this calibration applies, referred as 'the PRV number' in the rest of this document.
- The number of frames to capture for each step (referred as *'N frames'* in the rest of this document).
- The preheat time needed in seconds.
- The number of times the iteration between the cyclic error and gradient error should be performed

Data capture flow at the turntable is presented in the diagram (Figure 5):

1. After the initialization of the camera calibration process, application is waiting for the camera module to be connected. Once the module is connected, it is powered up and the PRV number stored in EEPROM is checked against the PRV number from the configuration file.
2. The module is then configured in one of the modes to be calibrated as defined in calibration process configuration.
3. The turntable motor is set to its home position.
4. Camera module is preheated during the predefined time, to stabilize the temperature conditions.
5. Turntable motor iterates through each of the predefined positions or views out of “num_views”.
6. At each of the views, N depth data frames are captured, and data is passed to MATLAB post-processing object.
7. In the last view position, the cat tree data stored in the EEPROM is read and passed to MATLAB
8. After the data is captured for the last view, data is post-processed, i.e. calibration parameters are computed by MATLAB.
9. The calibration parameters are then exported to an output file and/or serialized to camera EEPROM.

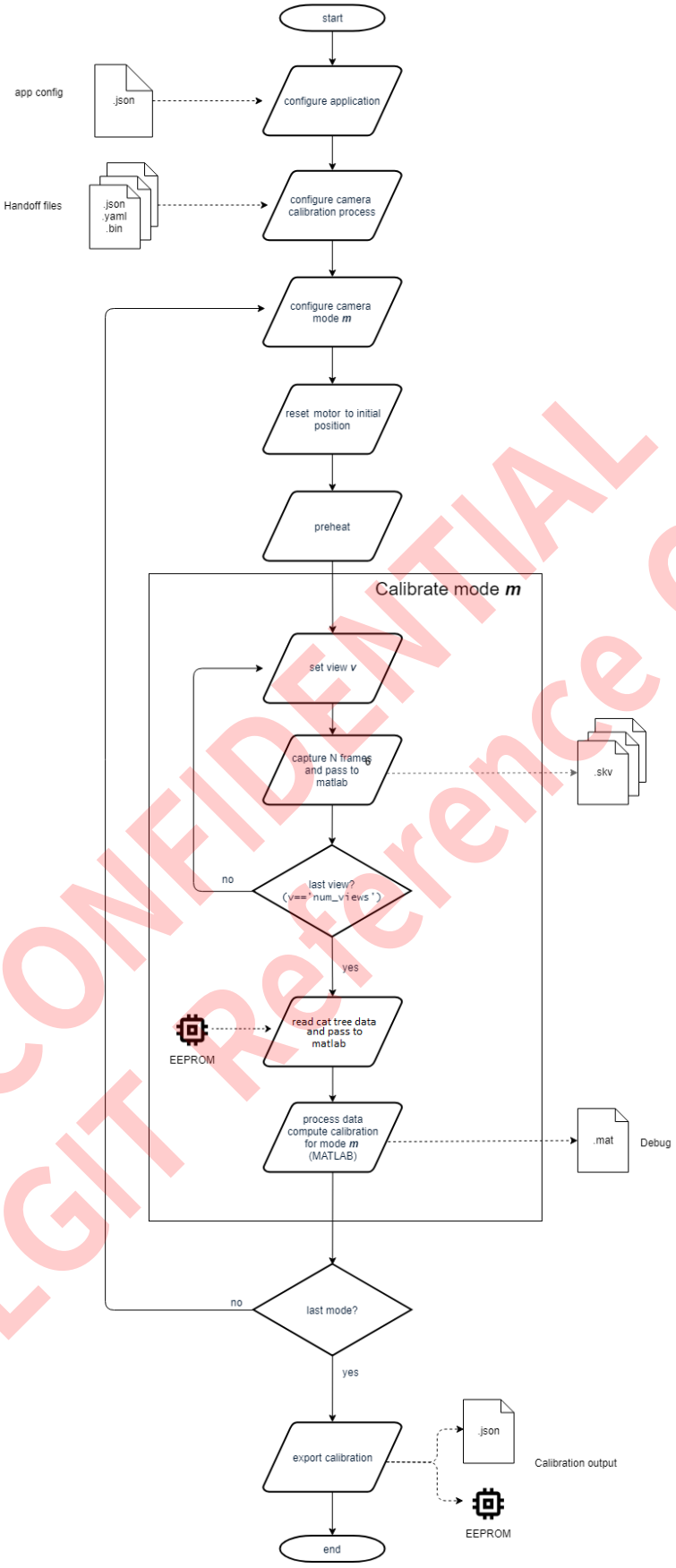


Figure 5: **Cat Tree** Calibration data capture flow at the Turn Table

The data used for calibration is captured for all the modes and all the views in the case of cat tree calibration. Calibration post-processing requires IQ depth data (complex number representing the measured depth phasor, where phase is proportional to measured distance/depth). Next to the depth data, temperature measurement (per component, per frame) from the camera module temperature sensor is recorded and used to correct temperature related phase variations. All this data is passed to and collected by MATLAB post-processing object, along with the information on view and frequency index. This Frequency index always must be zero in this revision of the calibration tool.
Code designated for data capture is further detailed in the next section.

5.1.3 Data capture files, folders and methods of interest

- `softkinetic/mm_calibration/calibrate.py`

  Python module serving as the entry point of the calibration process. This module is mostly responsible to handle incoming devices and launch the calibration process.

  - `calibration_main()`

    Main function of camera calibration process. Instantiate main calibration class `CalibrationApp`.

  - `CalibrationApp.run()`

    Setup MATLAB processing engine. Run calibration process from live camera stream or from recorded data.

  - `CalibrationApp._calibration_worker()`

    Check that the module connected, working and PRV stored in EEPROM is matching the configuration file. Run calibration by calling `CalibrationApp._calibrate()`

  - `CalibrationApp._calibrate()`

    Instantiate `CalibrationKernel` class from `matlab_processors.py` module. Run camera calibration through `CalibrationKernel.calibrate_camera()`. Once calibration is finished, export camera calibration to a file or serialize to camera EEPROM using `CalibrationApp._export()`.

- `softkinetic/mm_calibration/matlab_processors.py`

  Python module holding one main class `CalibrationKernel`, designated to control the data capture for selected mode and interface with MATLAB post-processing code. Interfacing with MATLAB is performed through `CameraCalibrator` class, which serves as a proxy for `CameraCalibrator.m` file.

  - `CalibrationKernel.calibrate_camera()`
    - Iterate through all modes specified for calibration.
    - Set selected mode.
    - Put the physical calibration setup in its initial position.
    - Preheat the camera in selected mode during the time specified by the `"preheat"` setting.
    - Start calibration data capture process for selected mode.
    - After the mode is calibrated, convert MATLAB calibration results from MATLAB data structure to python compatible data structure (call to MATLAB function `calibration2python()`)
  - `CalibrationKernel._calibrate_current_mode()`
    - Iterate through all the views.
    - Position the turntable accordingly.

- In each of the views capture depth data.
- For the last view, iterate through selected number of phase steps and capture the depth data at each phase step. Setting the view and phase step is performed by calling `set_view_and_phase_step()` on the data source. Data source is defined in python module `softkinetic/mm_calibration/source_sink.py`. The data source can either be a camera or a file. Capturing the data and passing it to MATLAB is performed by calling `CalibrationKernel._get_data_and_pass_to_matlab()`.

- `CalibrationKernel._get_data_and_pass_to_matlab()`
  - From the data source (see above) collect the predefined number of frames N.
  - Extract the required data (IQ, temperature, cat tree EEPROM data, phase step index (when applicable), view index and frequency index) and pass it to MATLAB `CameraCalibrator` object using the before mentioned proxy python class.

5.2 Calibration post-processing

5.2.1 Post-processing flow at the Cat Tree Setup

For each mode that needs to be calibrated, the raw phase and confidence data of N frames is processed in a python script as explained below:

1. The post processing at the cat tree setup is called once the N data frames are captured
2. The center of each ROI is detected and the mean rawphase, ROI_center_X and ROI_center_Y are computed for each patch
3. This cat tree data is stored in the EEPROM and transferred to the Turn table setup for further calculations

5.2.2 Calibration post-processing flow at the turn table setup

Once the data has been captured, the post-processing code computes calibration model parameters. The result is a calibration object that is eventually serialized in the EEPROM of the device.

The post-processing requires a number of configuration parameters, which are defined in a JSON file, described in **11.3 Mode JSON file**. Key parameters are:

- the size of the printed checkerboard square (`"square_size_mm"`).
- the calibration method to use (`"cec_mode"`)
- For Cat Tree Calibration
  - The number of iterations between the cyclic error calculations and the gradient error calculations ("number_of_iterations")
- Parameters that set individual error models being calibrated (gradient error, cyclic error, temperature error).
- Other key parameters needed for postprocessing are provided by the python capture code (modulation frequency of mode to be calibrated).

The post-processing is interlaced with the data capture. The interface between the two is step 6 and 7 in the data capture diagram (Figure 5). Iteratively, when N frames are captured for each of the views, Python code calls a MATLAB post-processing function that collects the data and starts part of the processing, while waiting for the data of the next view.

Once the last N frames from the last view and the cat tree data in the EEPROM are collected, all the calibration parameters are computed and passed back to python code.

Each time N frames are captured for a combination of view, frequency, Python code calls a MATLAB post-processing function that collects the data and starts part of the processing, while waiting for the following set of N frames. The outline of the calibration postprocessing is described hereunder and illustrated in the diagram (Figure 6).

1. MATLAB post-processing function is called with a new set of N frames, in combination with specifications regarding view and frequency of the data set.
2. The new data is checked to assess whether it corresponds to a new view (i.e. new rotation angle of the turntable). Specifically, this is performed by checking whether the new data corresponds to the first frequency (in case multi-frequency mode is used).
3. If the new data corresponds to a new view, it will be used for lens calibration (see step 6). Thereto, checkerboard corners are detected in the temporally averaged (over N frames) confidence image. If the data does not correspond to a new view, checkerboard data for this view have already been detected, and this step is skipped.
4. The new data is checked to assess whether it is the last data set expected. Specifically, this is performed by checking whether the new data corresponds to the last frequency (in case of muti-frequency mode) and the last view (out of selected num_views). If true, the cat tree data is read from the EEPROM for that mode and the calibration finalization process starts, otherwise, the process is returned to the start (step 1), waiting for the next view or next frequency.
5. If temperature compensation during calibration is selected, data is corrected for any phase drift during the process of calibration caused by system temperature variation. Mainly this step is important to compensate for phase changes caused by camera module heating up during the calibration process. More information on the temperature compensation of the phase data is provided in the section 1.2.3.2 of IMX316_AppNote_CameraCalibration(E)_28112017.pdf.
6. After temperature compensation step, detected checkerboard corners (from step 3) in all views are used to compute the camera projection model parameters (i.e. lens calibration). As a byproduct of this procedure, the 3D position of the calibration planes w.r.t. camera (extrinsics) are obtained for each of the views. These are used in step 9. More information on camera projection model (lens model) can be found in IMX316_AppNote_CameraCalibration(E)_28112017.pdf, section 1.1.
7. If enabled in calibration setup file, the next step is cat tree cyclic error calibration. The main purpose of this step is to model the phase dependent periodic phase error. To calibrate this phase error, the data measured at the cat tree setup is required. Cyclic Error is calculated by modeling the difference between the measured phase and the ground truth phase of the cyclic error setup.
8. The next step in camera calibration is to compute pixel dependent fixed phase offset or phase gradient error (see step 10). Thereto, we need to correct cyclic error in the measured phase data (this step) and we need to compute the reference phase for each of the views considered (step 9). In this step, we correct the cyclic error in the measured phase data. We use the cyclic error correction model computed in earlier step (see step 7). We correct the phase measurements of all the views and all the captured frequencies.
9. In this step, we compute the ground truth depth for each pixel in each view considered. Thereto, we use the lens calibration parameters and the relative position of each calibration plane (checkerboard plane) w.r.t. the camera

(see step 6). Ground truth depth is translated into ground truth phase or reference phase (i.e. the phase we want to correct our measurements to).

10. Having the measured phase and the reference phase data for all views, we can compute the pixel dependent fixed phase offset or phase gradient error. The phase gradient error is parameterized as a polynomial surface on the 2D image grid, and its parameters are estimated by minimizing a weighted sum of squared differences between reference and measured phase on the one hand and gradient phase error on the other hand. Weights were introduced to filter out bad data. More details on the model are provided in section 1.2.1.2 of IMX316_AppNote_CameraCalibration(E)_28112017.pdf.
11. In the next step we correct the cat tree data using the measured gradient offset model in the above step and loop back to Step 7. The number of repetitions of this iteration process can be controlled through the “number_of_iteration” parameter in the Mode JSON file.
12. Once all the requested models are calibrated, the model is returned to python code instance, which exports the model to the camera EEPROM and/or output file.

Code designated for calibration postprocessing is further detailed in next section.

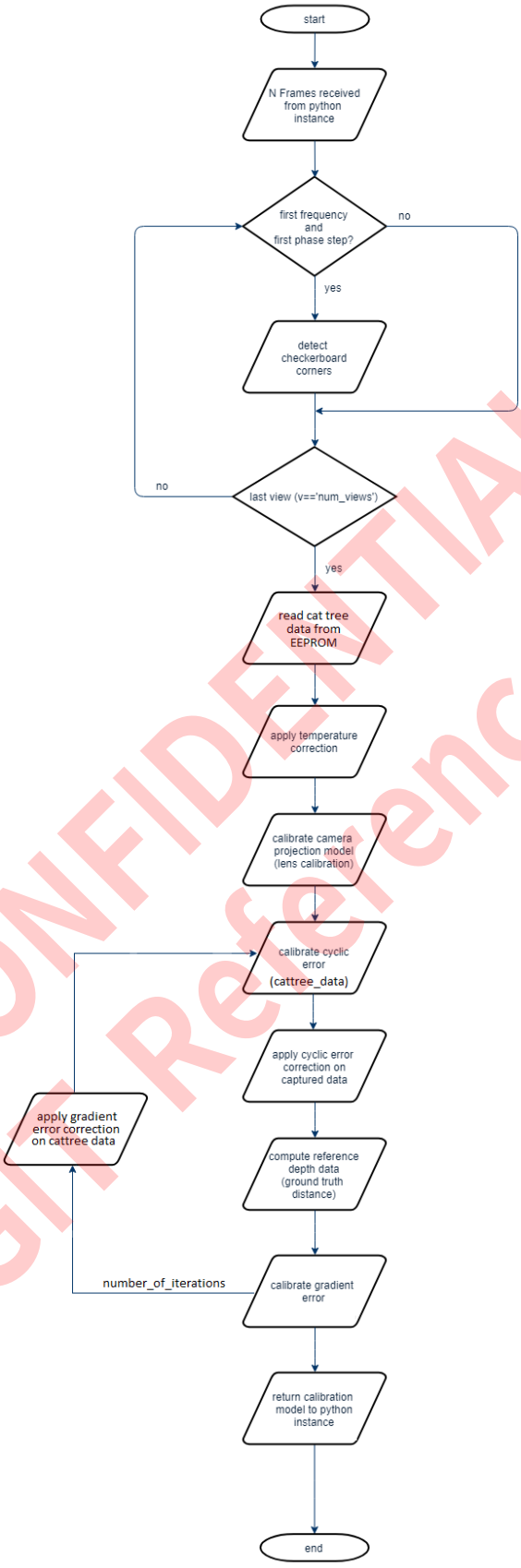


Figure 6: Calibration post-processing flow for **cat tree** calibration

5.2.3 Calibration post-processing files, folders and methods of interest

`softkinetic/mm_calibration/matlab/CameraCalibrator.m`

MATLAB file containing the main calibration class `CameraCalibrator`. `CameraCalibrator` is instantiated in the python code base, specifically as a member of `CalibrationKernel` instance. When the class is instantiated, calibration options are provided to the constructor, defining all the key calibration parameters defined in the calibration configuration JSON file in 11.3 Mode JSON file11.3

- `CameraCalibrator.calibrateMode()`
  - Method to initialize the mode to be calibrated. This function is called before each new mode is being calibrated.
- `CameraCalibrator.addData()`
  - Method that adds the chunk of data to compute calibration from. This method is called by the python code base, each time N frames are captured for selected view or frequency of a given mode.
  - First the data is preprocessed, by computing a temporal average of the captured data, to remove temporal noise. As discussed in the previous section, depending on the data chunk, different steps are then taken:
    - If the data chunk represents a new view, checkerboard corners are detected by calling class method `CameraCalibrator.detectCheckerboard()`
    - If the data chunk represents the last set of N calibration frames, , calibration is finalized by calling class method `CameraCalibrator.finalizeCalibration()`.
- `CameraCalibrator.detectCheckerboard()`
  - Method detects the position of checkerboard corners in the confidence image of the temporally averaged captured data.
  - Before detecting the checkerboard, the confidence image is histogram equalized.
  - The contrast across the image is adaptively enhanced to facilitate the robustness of the checkerboard corner detection algorithm.
  - The checkerboard corners are detected on the enhanced confidence image using MATLAB native function. Besides the checkerboard corners in the image plane, the corresponding world coordinates are defined for each of the corners.
  - The world coordinate system is defined to have x-y plane aligned with the checkerboard plane and z coordinate in the direction of its normal.
- `CameraCalibrator.finalizeCalibration()`
  - This method is the core of camera calibration. It computes all the calibration parameters, once all the calibration data is available (last view, last frequency and – in case of phase sweep - last phase step).
  - If temperature compensation during calibration is enabled, the method will first correct all the phase data to compensate for temperature related phase change. This is performed using the model described in IMX316_AppNote_CameraCalibration(E)_28112017.pdf . As a reference calibration temperature, i.e. the temperature all frames are matched with, it takes the temperature measured at the last frame of the last view.

- The next step is to compute camera projection model (lens model). Details on the model are provided in IMX316_AppNote_CameraCalibration(E)_28112017.pdf .The model parameters are computed from the detected checkerboard corners and their respective world coordinates, collected across all the views. The (non-linear) optimization process computes all the model parameters. Moreover, relative position of the coordinate system of the camera (centered in the optical center with z axis in the optical axis direction) with respect to the world coordinate system defined in the checkerboard plane (see explanation above in `CameraCalibrator.detectCheckerboard())` is computed for each view. These so-called extrinsics are in the next step used to compute the ground truth depth for each pixel in each calibration image.
- In this step the ground truth depth is calculated for each pixel in each calibration image. This ground truth depth is subsequently translated into ground truth phase or reference phase (i.e. the phase we want to measure). The rest of the calibration method intends to compute calibration model parameters that convert measured into reference phase. These reference phases are computed using the output of lens calibration, being the calibrated lens model as well as the position (translation T and rotation R) of the checkerboard panel w.r.t. the camera for each of the calibration images used. Using a function `sk.projections.rt2plane`, we convert R and T of each calibration view into a plane equation describing the checkerboard plane in the camera coordinate system. These plane representations are combined with lens intrinsics into the function `sk.projections.plane2radial` to compute the ground truth depth for each pixel in each calibration image considered. This ground truth depth for a particular pixel in a particular view is to the Euclidean distance between the camera origin on the one hand and the intersection between projection ray (of the pixel considered) and the plane (considered) on the other hand.
- The following step is an iteration over the cyclic error calibration and the gradient offset calibration
- If enabled. `CameraCalibrator.CTCyclicError` method is used for calibrating the cyclic error using the cat tree setup data(see below).
- The last step is computing gradient phase error (i.e. fixed pixel dependent phase error). The gradient error is computed on the cyclic error corrected depth measurements, making it (measured) phase independent. Cyclic error correction is thus applied first, using the model described in the document IMX316_AppNote_CameraCalibration(E)_28112017.pdf and parameters computed in the preceding calibration step. The resulting cyclic error corrected phase data are combined with their corresponding reference phase into `GradientErrorFull` class, that calibrates the gradient error model parameters.

- `CameraCalibrator.CTCyclicError()`
  - This method calibrates the parameters of the cyclic error model described in detail in IMX316_AppNote_CameraCalibration(E)_28112017.pdf. Cyclic phase error model is calibrated using the **cat tree** data from the EEPROM.
  - The Cyclic error for different distances are measured using the patches of the cat tree setup. The Cyclic error model is common to all the pixels of the array.
- `sk.calibration.GradientErrorFull`
  - Methods of `GradientErrorFull` class allow us to compute the gradient error model. There are two main functions, one that pre-processes the input data and one that fits the model on the pre-processed data. For details see IMX316_AppNote_CameraCalibration(E)_28112017.pdf.

- `sk.calibration.GradientErrorFull.fromReferencePhasePreprocess()`
  - Method takes as an input the measured phase (corrected for temperature and cyclic error if enabled), their respective confidence images and the expected reference phase for all the measured calibration panel views. Gradient image of each view is computed as a difference between the reference phase and the measured phase (corrected for temperature and cyclic error if enabled).
  - First each of the views is pre-processed to remove the bad measurements. Spatial noise is computed in 3x3 ROI centered on each pixel. If the spatial measured phase variance exceeds the predefined threshold value, given pixel is discarded.
  - Secondly, each pixel confidence value is compared to maximum specified value. If above, the pixel is discarded, to remove saturated pixels. Pixel confidence is also compared to the lower limit, to remove the unreliable measurements (noise, scattering). The lower limit is defined as an absolute value or as a percentile of all the values within a frame, whichever is higher. In the end, filtered gradient images from all the views are combined by using a weighted averaging, with confidence as a weighting function.
- `sk.calibration.GradientErrorFull.fromReferencePhaseFit()`
  - This function takes the pre-processed phase gradient image and fits a smooth polynomial surface to it.

6. Calibration Application Project Structure

This section explains the content of the calibration application source folder.

At the root of the folder the following files are available:

- `changelog.txt`: A text file summing up the latest changes to the application
- `conda_requirements.txt`: A file allowing to install all the required dependencies of the application through conda
- `open_source_software_licenses.txt`: A file acknowledging all the Open Source Software distributed with the application sources.
- `quickstart.txt`: A quickstart guide.
- `softkinetic/`: A folder that will be described in a next section
- `test_data/`: A folder that will be described in a next section

6.1 Overview of the test_data folder

This folder contains a folder per supported camera (i.e. iu316_minikit). Each of these folders contains the following items:

- `run.py` that will start the calibration application.
- `configurations` folder containing the following items:
  - The handoff files to setup the module in the desired configuration (in yaml format).
    The txt file is used for traceability.
  - The configuration file to shutdown properly the module.
- The `full_app.json` that is the main configuration file of the calibration application. It must be edited before executing the application. This step is described in the Installing Calibration Software Deliverable section. More details about this file can be found in JSON files specifications section.
- The `live_app.json` and `skv_app.json` allow to run the calibration application with different setups than `full_app.json`.
- The `mode.json` contains the configuration for the modes to be calibrated, including the calibration method to use.

6.1.1 Python and MATLAB softkinetic namespace walkthrough

The `softkinetic` folder contains almost all the software packages required to run the calibration application at the exception of the `sgrabber` folder which is located next to the `softkinetic` folder. The `softkinetic` namespace is the root for both the python and MATLAB code.

The `softkinetic` namespace is split in the following namespaces:

- `calibration_codec`: A codec based on msgpack used to serialize and deserialize the calibration data. Note that this namespace doesn't cover how to store or load the serialized data.
- `matlab_packages`: A collection of python and MATLAB scripts to allow the calibration and interaction with MATLAB.
- `mm_calibration`: The calibration application.
- `numpy_aligned_memory:` A helper python module allowing to instantiate numpy arrays with a certain alignment. This is useful whenever one needs to perform SIMD operations on numpy arrays.

- `production_codec`: A codec based on msgpack used to serialize and deserialize the production data such as serial number. Note that this namespace doesn't cover how to store or load the serialized data.
- `cattree_codec`: A codec based on msgpack used to serialize and deserialize the data measured at the cat tree setup. Note that this namespace doesn't cover how to store or load the serialized data.
- `skv`: A wrapper around HDF5 to store movies.
- `skvext`: A wrapper around skv adding semantic in how movies are stored.
- `skwatch`: Collection of helper classes to create GUI using pyqtgraph.
- `streamers`: A python wrapper around a C++ implementation that allows to retrieve raw data from a UVC interface.
- `turntable`: A python package allowing to control the turntable for **phase sweep** calibration.
- `utilities`: A collection of helper functions and classes.

The most important namespaces have been highlighted and will be covered in more details in the following section.

6.1.2 calibration_codec

The `calibration_codec` package mostly exposes a load and dump function to respectively serialize a python object into an array of bytes and deserialize an array of bytes into a python object.

When serializing an object, the module:

- Verifies that the supplied object follows the expected schema (stored in the `schema.json` file)
- Translates all the string keys of the input dictionary to integer tokens
- Pack the resulting dictionary using msgpack.

When deserializing an array of bytes, the module:

- Unpack the byte array using msgpack
- Translates all integer keys to string values.

Note that this implementation of the codec must match the one in the drivers.

6.1.3 Streamers

This package is a python wrapper around the hotplugger.

6.1.4 mm_calibration

This is the main package where most of the code of the calibration application lies. Here is a list of the most relevant modules and packages:

- `camera_implementations`: this package contains IMX316/IU316 specific implementations for being able to operate such module in the calibration application.
- `matlab`: this package contains MATLAB scripts involved in the post processing. Note that additional MATLAB packages are also available in the `softkinetic.matlab_packages` namespace

- `calibrate.py`: this is the entry point of the calibration process. This module is mostly responsible to handle incoming devices and launch the calibration process.
- `matlab_processors.py`: this module implements all the capture and post processing process described in the Calibration Application Execution Flow section of this document.
- `robtos.py`: this module provides a class allowing to control the turntable of the setup

7. Adding a new module in the Calibration Application

This section provides some details on how to integrate a new module in the Calibration Application.

The typical workflow is to duplicate the code of an existing code and then modify the implementation when needed.

7.1 Preparing the code

In this guide, `<Camera-Calibration>` is a placeholder for directory where this application is located on the system.

1. Add the new **module name** in the `CameraModel` class in `<Camera-Calibration>\softkinetic\mm_calibration\calibrate.py` and in the "`camera_model`" list in `<Camera-Calibration>\softkinetic\mm_calibration\app_schema.json`.

This module name must be used to replace the `<module_name>` placeholder in the rest of this guide.

2. Duplicate the `iu316_minikit` folder in `<Camera-Calibration>\softkinetic\mm_calibration\camera_implementations` and rename it to the module name.
3. Define a factory function (like the existing `make_iu316_minikit`) in `<Camera-Calibration>\softkinetic\mm_calibration\calibrate.py` and add a reference to it in the `_camera_library` dictionary in the same file.
4. Duplicate the `iu316_minikit` folder in `<Camera-Calibration>\test_data` and rename it to the module name. Note that the name doesn't matter for the rest of this document.
    a. In the 'configurations' folder, place the handoff files for the modes to be calibrated.
    b. Edit the `handoff_folder` path and the `camera_model` to match the one defined in step 1 in the `<Camera-Calibration>\test_data\<module_name>\full_app.json`, `<Camera-Calibration>\test_data\<module_name>\live_app.json` and `<Camera-Calibration>\test_data\<module_name>\skv_app.json`,
    c. Edit the `<Camera-Calibration>\test_data\<module_name>¥mode.json`, file and adjust the mode names to be calibrated.

7.2 Implementing the new module itself

Most of the implementation of the new module will take place in the `<Camera-Calibration>\softkinetic\mm_calibration\camera_implementations\<module_name>'` folder created in step 2 of the previous sub section.

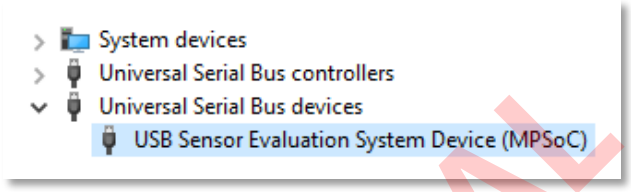
This sub-section will provide some information about each function to be implemented as well as their use in the calibration application.

Function name	Description
Constructor	Instantiates the device. No I/O should be performed on the device at this stage. The `device_info` argument is a named tuple containing at least the 'path' to the device. This function is called whenever a matching device is connected to the host.
Close	Closes all the interfaces so that they can be reused later.
read_calibration_data	Reads the calibration data stored in the EEPROM of the currently connected device and return a deserialized dict of the calibration data. This is used whenever the application is configured to update the content of the EEPROM rather than replacing the content.
store_calibration_data	Writes serialized calibration data in the EEPROM of the currently connected device and returns the serialized calibration data. This function is used when the calibration is complete and the data needs to be stored in the EEPROM of the device.
read_production_data	Reads the production data stored in the EEPROM of the currently connected device and return a deserialized dict of the calibration data. This function is called directly whenever the application is configured to check that the PRV of the connected device matches the one stored in the EEPROM of the device. Note that this is not used at the moment.
power_down	Powers down the module. Note that using the default setup, this function is most of the time not called.
power_cycle	Power cycle the module. This function is called before the module gets configured.
start_streaming	Starts pulling data from the device. Note that the sensor should be in streaming state before this function is called.
stop_streaming	Stop pulling data from the device.
serial_number	Returns the serial number of the module. This is mostly used to name output files.
set_phase_steps	Adjust phase steps to the current desired step. This is used when performing phase stepping.
extract_view_and_phase_step	Extract from the embedded data of the frame the current phase step and view indices and store it in the input buffer. This is to be used in conjunction with the `set_phase_steps` function. In case of **table sweep** calibration, the phase step is always equal to 1, so only the data corresponding to the physical views will be effectively extracted.
replay_handoff_file	Apply the configuration stored in the input filename to the module.
get_traceability_data	Returns any relevant traceability data to be added to the calibration data.

8. Installing SSP500

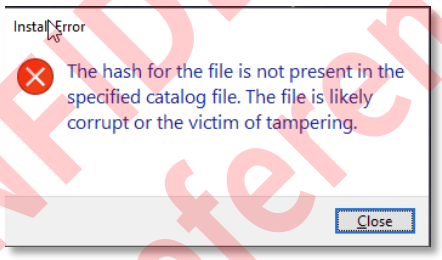
The SSP500 windows drivers must be installed and can be found in the SSP500_drivers.zip archive provided with the Calibration Software package.

1. **Plug** the SSP5000 and power it up
2. The device should appear in the **Windows device manager** after a few seconds under the name *USB Sensor Evaluation System Device (MPSoC)*



3. Right click on the device in the device manager and install the drivers, or right click on `usbclnt_driver_mpsoc.inf` and select ***Install***.

If you see the following error dialog while installing the driver:



This likely means that the `inf` file has been copied on your system with the wrong line ending, and the digital signature of the driver is not matching anymore.

Proceed as follows:

1. Open the `usbclnt_driver_mpsoc.inf` in a text editor such as Notepad++
2. Select ***Edit*** > ***EOL Conversion*** > ***Windows Format***
3. **Save** the file
4. Restart the installation procedure

Also make sure the SSP500 board jumper are set to Down, Up, Down, Up.

9. ARDUINO IDE + Windows Driver

Calibration software can control the turntable on ARDUINO platform. See following instruction to install IDE and drivers, connect the board to PC and write the firmware into the board.

ARDUINO firmware (*stepper_control.ino)* is contained inside *arduino_firmware.zip*

9.1 Install Arduino IDE and windows drivers

- Visit https://www.arduino.cc/en/Main/Software
- Download the package from weblink of “Windows Installer”.
    - e.g. arduino-1.8.4-windows.exe.
- Double click .exe and install all drivers by following the instructions.

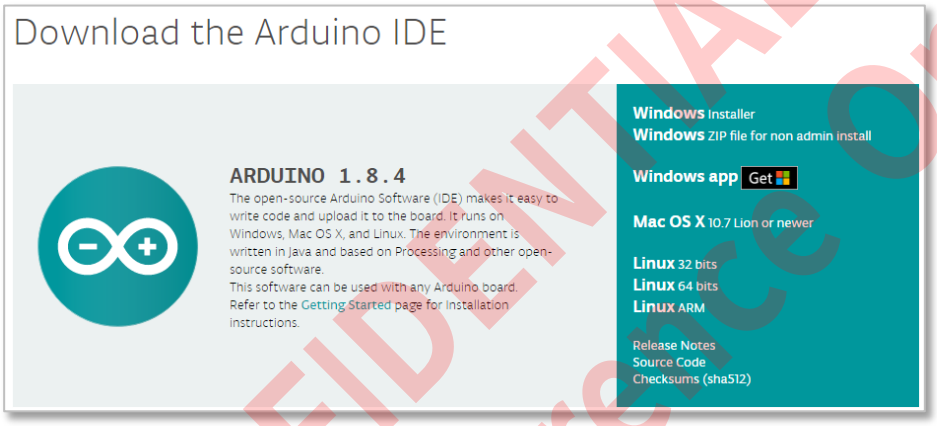


Figure 8: Download the Arduino IDE.

9.2 Connect Arduino board

- **Connect** Arduino board to USB port.
- Open **device manager** and right click on the unknown device.
- Select ***“update driver”*** and update with FTDI driver. New virtual COM port will appear.
- Select the new virtual COM port and update with Arduino driver. “Arduino UNO” will appear.

9.3 Compile Arduino firmware source code

In case the firmware of the turntable is not already installed, follow these steps:

- For the turntable Arduino driver
  - **Connect** the turntable Arduino to the computer
  - Open the delivered "stepper_control.ino" with Arduino IDE application.
  - Run ***"Compile"*** then ***"Upload"***. Firmware is written into the board.

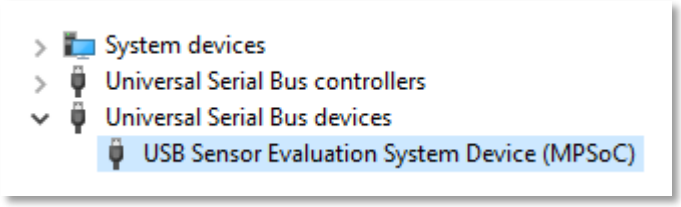
Figure 9: Compile & Upload.

- For the elevation platform control Arduino driver
  - **Connect** the elevation platform Arduino to the computer
  - Open the delivered "dual_motor_stepper_control.ino" with the Arduino IDE application
  - Run ***"Compile"*** then ***"Upload"*****.** Firmware is written into the board

10. Device Interfaces Definition

When connected through a SSP500 to the host platform, the device will appear as a single device.

In the Windows device manager the device should appear as follow:



11. JSON Files Specifications

11.1 Overview

In the following sections we will describe the structure of the most relevant json files required for the functioning of the calibration, with some examples and with an explanation of each of the fields present in these files. Please note that the names below are generic, and we will explicitly name them in each of the sections.

- application.json
- mode.json
- output_calibration.json

11.2 Application JSON file

This JSON file specifies some of the behavior of the application. The supplied application JSON file is checked against the schema stored in `softkinetic/mm_calibration/app_schema.json`. Said schema contains information about the expected file layout, expected datatypes and lists of valid values contained in the `application.json` input file. If there is a mismatch during this check, the process will abort.

Here you can find an example of a valid application json file (also provided with the delivery). The explanations of each field can be found below.

Application.json = full_app.json:

```
{
    "app": {
        "camera_model": "iu316_minikit",
        "prv": "any",
        "rootfolder": "C:\\Logs",
        "export_to": ["json", "rom"],
        "gui_mode": "gui",
        "robot_model": "arduino",
        "handoff_folder": "test_data\\iu316_minikit\\configurations",
        "save_data": true,
        "mipi_mode": true
    }
}
```

Parameter	Accepted values	Note
camera_model	string	Currently only ″iu316_minikit″ is supported
prv	"any" string or int	Allowed values: any integer or ″any″. When set to an integer value, the application will check first whether the PRV stored in the EEPROM of the device matches the one defined in the configuration file. In case there is a mismatch, an error message will be displayed.

rootfolder	string	Root path under which all output files are stored. Outputs includes calibration output in JSON or MessagePack format.
export_to	list of strings	A list of values from `"json"`, `"mp"` and `"rom"`. The first two output the calibration data to a JSON and MessagePack formatted file, respectively in the folder `"rootfolder"`. `"rom"` writes the calibration data to the camera ROM. The ROM uses MessagePack coding as well.
gui_mode	string	Allowed values: `"none"`, `"viewer"`, `"gui"`. With `"none"` and `"viewer"`, calibration runs for a single camera or dataset and then quits the application. The difference is that the former option doesn't produce a GUI, while the latter uses a simple skwatch viewer. The `"gui"` option brings up the full GUI that's used in production and waits for the user to click a button to start calibration.
robot_model	string	Allowed value: `"mock"`, `"arduino"`. • `"mock"` does not move anything. It should be used for testing purposes or offline calibration. It is also given as a sample for the user to write their own robot for the calibration application. • `"arduino"` enables the turntable. Use this robot model for **phase sweep** calibration.
handoff_folder	string	Path from which the mode related files will be loaded from.
save_data	boolean	If true, saves the skv files. Defaults to false
mipi_mode	boolean	True for cameras with a mipi interface

11.3 Mode JSON file

This JSON files specifies the modes to be calibrated as well as some of the calibration parameters to use for the model. The supplied JSON file is checked against the schema stored in `softkinetic/mm_calibration/calibration_schema.json`. Said schema contains information about the expected file layout, expected datatypes and lists of valid values contained in the corresponding `mode.json` input file. If there is a mismatch during this check, the process will abort.

Here you can find an example of a valid mode json file (also provided with the delivery). The explanations of each field can be found below.

mode.json

```
{
```

```
"calibration": {
    "modes": [
        "ID_3161_EF05_30FPS_60MHz_DT50_570us",
        "ID_3161_EF04_30FPS_100MHz_DT33_570us"
    ],
    "per_mode": {
        "ID_3161_EF05_30FPS_60MHz_DT50_570us": {
            "num_frames": 10,
            "temperature_error": {
                "slope": [0, 0]
            },
             "cec_mode": 3,
             "cattree_cec": {
                       "groundtruth_dist":[0.605970296,0.703491294,0.807960395,0.908955445,1.004987562,1.110945543,1.205985075,1.306483831,1.4],
                       “number_of_iterations": 5
                             },
             "num_phase_steps": 1
         },
        "ID_3161_EF04_30FPS_100MHz_DT33_570us": {
            "num_frames": 10,
            "temperature_error": {
                "slope": [0, 0]
            },
             "cec_mode": 3,
             "cattree_cec": {
                       "groundtruth_dist":[0.605970296,0.703491294,0.807960395,0.908955445,1.004987562,1.110945543,1.205985075,1.306483831,1.4],
                       “number_of_iterations": 5
                        },
             "num_phase_steps": 1
        }
    },
    "common": {
        "square_size_mm": 40,
        "preheat_seconds": 120,
        "num_views": 4,
        “cec_mode”: 0,
        "num_phase_steps": 1,
        "gradient_error": {
            "PhaseNoiseThresholdRadians": 999,
            "NormUpperLimit": 10000,
            "NormLowerLimit": 0,
            "NormLowerLimitPercentile": 50,
            "SurfaceType": "poly22"
        },
        "temperature_error": {
            "apply_at_calibration": false,
            "slope": [0, 0]
        },
        "enable_gec": true,
        "frame_type": "standard"
    },
    "depth_intrinsics": "ID_3161_EF04_30FPS_100MHz_DT33_570us",
    "outputmap": {
        "ID_3161_EF05_30FPS_60MHz_DT50_570us": [
            ["ID_3161_EF05_30FPS_60MHz_DT50_570us", 0]
        ],
        "ID_3161_EF04_30FPS_100MHz_DT33_570us": [
            ["ID_3161_EF04_30FPS_100MHz_DT33_570us", 0]
```

```
        ]
      }
    }
}
```

11.3.1 Top-level

Field	Explanation
Modes	A list of mode names to capture data for.
depth_intrinsics	A single mode name. Even though lens intrinsics are computed for each mode, the final calibration output contains only a single lens intrinsics model. This option specifies from which mode the intrinsics are output.
common	A dictionary with options that apply to the calibration of all the modes in the modes list.
per_mode	A dictionary where the keys are mode names, and the values are dictionaries which can contain the same options as the common dictionary. The options for each mode are created by starting from the common dictionary, and then overriding whatever values are defined for that mode in the per_mode dictionary.
outputmap	A dictionary to allow re-use of calibration data between modes. If this field isn't present, we output calibration data for the modes defined in modes directly.

11.3.2 Calibration options in 'common' and 'per_mode'

Below table contains all the parameters that can be set in the `common` and `per_mode` dictionaries. The *Settable* column has the following meaning:

- **common:** this value should only be set in the common dictionary.
- **per mode:** this value can be set in either the common or per_mode dictionary, with the latter overriding the former.

Field	Settable in	Explanation
calibration_method	common	The calibration method to use. Pick from: "phase_sweep" or "table_sweep".
num_views	common	Number of orientations of the camera relative to the checkerboard panel. Angle

		between orientations is 360 degrees/num_views. An error is raised at startup is this setting is used when “calibration_method” is set to “table_sweep”.
square_size_mm	common	The physical length of the side of one square of the checkerboard in millimeters.
preheat_seconds	common per_mode	How long to wait, in seconds, to let the camera preheat between starting a mode and capturing calibration data for it. This preheat step is done for every mode calibrated during a calibration session.
num_frames	common per_mode	How many frames to average for each data capture.
num_phase_steps	common per_mode	Set to 1 for Cat Tree Calibration
enable_gec	common per_mode	Boolean indicating whether or not to perform gradient error calibration.
cec_mode	common per_mode	For Cat Tree Calibration, set cec_mode = 3
cattree_cec	common per_mode	The ground truth distances of the cat tree patches is set using "groundtruth_dist" The number of iterations of the cyclic error correction and gradient error correction is set using “num_of_iterations”
gradient_error	common per_mode	Dictionary with additional options for gradient error calibration.
temperature_error	common per_mode	Dictionary with additional options for temperature error calibration.
frame_type	common per_mode	Must be set to "standard".

11.3.3 Temperature error options

Field	Required	Explanation

slope	yes	A list of integers specifying slope of temperature error correction coefficients for each temperature sensor. Currently, this model is simply written to the output.
apply_at_calibration	yes	Boolean indicating if the temperature correction is applied during camera calibration.

11.3.4 Cat Tree Cyclic error options

Field	Required	Explanation
groundtruth_dist	yes	A list of the ground truth distances of the center of the patches of the cat tree setup
num_of_iterations	yes	The number of iterations of the cat tree cec and the gradient error calibration loop

11.3.5 Gradient error options

Field	Required	Explanation
SurfaceType	no	Degree of surface polynomial to use for the gradient error model. Specified as "polyXY", where X and Y are single digit integers between 0 and 5, indicating the order of the polynomial in X and Y directions, respectively. Since the gradient calibration includes global phase offset calibration, you can use "poly00" for a simple global offset.
PhaseNoiseThresholdRadians	no	The raw phase gradient images are spatially filtered by a 3x3 stddev filter, and any values over this threshold are omitted. This criterion can also be used to filter out discontinuities.
NormUpperLimit	no	All pixels with a confidence higher than this value are omitted.
NormLowerLimit	no	All pixels with a confidence lower than this value are omitted.
NormLowerLimitPercentile	no	All pixels with a confidence in the bottom percentile indicated by this value (between 0 and 100) are omitted. E.g. if the value is 30, the 30% of pixels with the lowest confidence are removed.

12. Production Data Format Specifications

The production data structure is a dictionary holding at least the following entries:

- `serial_number`: the serial number (not including the CRC)
- `prv`: the PRV number.
- `software_id`: the software API id.

More information about the IU316_minikit ROM layout can be found in the IU316-minikit Rom Layout section.

12.1 Binary format specification

All the data is stored in little endian order.

The previously defined production data structure is encoded as:

- A 2 characters magic string stored on 2 bytes: 'RB' (Raw Bytes).
- The size in bytes of the payload (not including the 2 bytes magic number nor this field size) stored on 2 bytes.
- The size of the serial number (excluding any trailing zero) in bytes stored on 1 byte.
- The serial number (not including trailing zero).
- The serial number CRC-8 stored on 1 byte.
- The prv number stored on 4 bytes.
- A placeholder of 4 bytes to replace the deprecated calibration tool version.
- The software_id stored on 4 bytes.

13. Cat Tree Data Format Specifications

The Cat Tree data structure is a dictionary holding the following entries:

- 'num_modes': The number of modes that need to be calibrated
- 'num_points': The number of patches in the cat tree setup
- 'mode_id': The Mode IDs of all the modes that need to be calibrated
- 'roi_x': The X pixel coordinates of the detected patch centers
- 'roi_y': The Y pixel coordinates of the detected patch centers
- 'Rawphase' : The average raw phase measurements of the patch centers

13.1 Binary format specification

In the EEPROM the Cat Tree Data is stored at address 0x40. It is overwritten by the calibration data.

All the data is stored in little endian order.The previously defined production data structure is encoded as:

- NUM_MODES : i – INTEGER
- NUM_POINTS : i – INTEGER
- MODE_ID[i] : i – INTEGER
- ROI_X[I] : I – INTEGER
- ROI_Y[I] : i – INTEGER
- PHASE[i][j] : d - DOUBLE

14. Calibration Data Format Specifications

This section describes how the calibration data are structured and how they are serialized in the EEPROM of this device. This is the format used by the `calibration_codec` used in both the calibration application and in the drivers.

The serialization format for the calibration is as follow:

- 2 bytes magic number: 0x504D
- 4 bytes unsigned int holding the length in bytes of the packed data.
- packed data in msgpack format.

The current specification applies to the input dictionary passed to the codec. All string keys are mapped to a unique integer id when serialized.

Below you can find an example of this file, being the output of the calibration process, with some additional information. The output file format of the calibration application is independent from the chosen calibration method.

Example: calibration_iu316_minikit_20180319_144839.json


```
{
   "calibration_tool_version": [
      0,
      2,
      0
   ],
   "configurations": [
      {
         "cyclic_error": [
            0
         ],
         "gradient_error": [
            0
         ],
         "temperature_error": [
            0
         ],
         "uid": 61188
      },
      {
         "cyclic_error": [
            1
         ],
         "gradient_error": [
            1
         ],
         "temperature_error": [
            1
         ],
         "uid": 61189
      }
   ],
   "cyclic_errors": [
      {
         "algorithm": 2,
         "coefficients": [
```

```
            2,
            0.0008854118428183765,
            -0.0009398299524231279,
            4,
            0.025229661788439967,
            0.05906343335460648
        ],
        "format": 1
    },
    {
        "algorithm": 2,
        "coefficients": [
            2,
            0.00017633765074389816,
            -0.0010483714557991033,
            4,
            0.06462206996922132,
            -0.06439583062977199
        ],
        "format": 1
    }
],
"depth_intrinsics": {
    "cx": 121.65855613916888,
    "cy": 95.81809957439408,
    "fx": 218.64630212346816,
    "fy": 218.48917727630945,
    "k1": 0.18291229271630804,
    "k2": -0.4571222042955127,
    "k3": 0.0,
    "p1": 0.0,
    "p2": 0.0
},
"gradient_errors": [
    {
        "algorithm": 1,
        "coefficients": [
            -0.840867042747537,
            -0.0047480680455253025,
            -0.0022221922536903335,
            -0.004523967714290774,
            0.0005349161797967239,
            0.0022215436835763148
        ]
    },
    {
        "algorithm": 1,
        "coefficients": [
            -0.8821579333304176,
            -0.0021014178786293793,
            0.0012975133400275978,
            0.0033202899669847385,
            -0.00041867796139405405,
            0.00666417611474304
        ]
    }
],
"temperature_errors": [
```

```
        {
            "algorithm": 1,
            "coefficients": [
                0.0,
                0.0
            ],
            "reference_temperatures": [
                12.0,
                61.9474
            ]
        },
        {
            "algorithm": 1,
            "coefficients": [
                0.0,
                0.0
            ],
            "reference_temperatures": [
                12.0,
                63.8457
            ]
        }
      ]
}
```


14.1 Conventions

The data structure should use only native python types (including lists and dictionaries).

Dictionaries should always use strings as keys. Keys must:

- Be lower case and use Snake case.
- Match the MATLAB convention.
- Have a length less than 63 characters.

Lists should use homogeneous types.

> Unless stated otherwise the corrections are added to the measured phase.

14.2 Calibration structure

The calibration structure is defined as a dictionary using string keys in the following list:

- `"depth_intrinsics"`: a single `Intrinsics` object.
- `"configurations"`: a list of `Configuration` objects.
- `"gradient_errors"`: a list of `GradientErrorData` objects.
- `"cyclic_errors"`: a list of `CyclicErrorData` objects.
- `"temperature_errors"`: a list of `TemperatureErrorData` objects.
- `"calibration_tool_version"`: a list of integers containing the major, minor and patch number.

Used models are described in the document IMX316_AppNote_CameraCalibration(E)_28112017.pdf.

14.2.1 Intrinsics

The dictionary can use the following keys:

- `"fx"`: the focal length in the horizontal direction in pixels.
- `"fy"`: the focal length in the vertical direction in pixels.
- `"cx"`: the central point location in the horizontal direction in pixels.
- `"cy"`: the central point location in the vertical direction in pixels.
- `"k1"`, `"k2"`, `"k3"`: the radial distortion coefficients.
- `"p1"`, `"p2"`: the tangential distortion coefficients.

If a key is missing, the value for the said key will be set to 0.0.

14.2.2 Configuration

The dictionary can use the following keys:

- `"uid`: the UID of the configuration, i.e. mode
- `"temperature_error"`: A list of indices of the temperature error data in the `"temperature_errors"` entry.
- `"cyclic_error"`: A list of indices of the cyclic error data in the `"cyclic_errors"` entry.
- `"gradient_error"`: A list of indices of the gradient error data in the `"gradient_errors"` entry.

14.2.3 Temperature Error data

The dictionary can use the following keys:

- `"algorithm"`: the algorithm id.
    - 0: means that temperature error correction is explicitly disabled.
    - 1: $\mathrm{sum}((\mathrm{reference\_temperatures(i)} - \mathrm{temperature(i)}) * \mathrm{c(i)})$
- `"reference_temperatures"`: a list of reference temperatures in celsius degrees.
- `"coefficients"`: a list of coefficients in radians/celsius degree with the same size of the `"reference_temperatures"` entry.

14.2.4 Cyclic Error data

The dictionary can use the following keys:

- `"algorithm"`: the cyclic error algorithm id.
    - 0: explicitly disabled.
    - 1: uses forward model. [deprecated]
    - 2: uses backward model.
- `"coefficients"`: a list of coefficients in radians.
    - when using the forward model: [cos(2x), sin(2x), cos(4x), sin(4x)]
    - the exact format in which the coefficients are stored depends on the format when using the backward model.
- `"format"`: an integer indicating the format in which the coefficients are stored. It defaults to 0.

- when set to 0, indicates that coefficients are stored as: [cos(x), sin(x), cos(2x), sin(2x), cos(3x), sin(3x)…]. Coefficients can be omitted starting from the right.

  Example:

  "coefficients": [a_1,b_1,a_2,b_2,…,a_N,b_N]

  model: $CYC\_ERR(\varphi_t) = a_1 \cdot \cos(\varphi_t) + b_1 \cdot \sin(\varphi_t) + a_2 \cdot \cos(2 \cdot \varphi_t) + b_2 \cdot \sin(2 \cdot \varphi_t) + \cdots + a_N \cdot \cos(N \cdot \varphi_t) + b_N \cdot \sin(N \cdot \varphi_t)$

- when set to 1, indicates that coefficients are stored as: [n, cos(nx), sin(nx),…] where n is the harmonic on which the 2 subsequent coefficients apply.

  Example:

  "coefficients": [2,a_2,b_2,4,a_4,b_4,…,N,a_N,b_N]

  model: $\varphi_{CE}(\varphi_t) = a_2 \cdot \cos(2 \cdot \varphi_t) + b_2 \cdot \sin(2 \cdot \varphi_t) + a_4 \cdot \cos(4 \cdot \varphi_t) + b_4 \cdot \sin(4 \cdot \varphi_t) + \cdots + a_N \cdot \cos(N \cdot \varphi_t) + b_N \cdot \sin(N \cdot \varphi_t)$

14.2.5 Gradient Error data

The dictionary can use the following keys:

- `"algorithm"`: the algorithm id.
  - 0: explicitly disabled.
  - 1: uses poly55 model. p00 + p10x + p01y + p20x^2 + p11xy + p02y^2 + p30x^3 + p21x^2y + p12xy^2 + p03y^3 + p40x^4 + p31x^3y + p22x^2y^2 + p13xy^3 + p04y^4 + p50x^5 + p41x^4y + p32x^3y^2 + p23x^2y^3 + p14xy^4 + p05y^5
- `"coefficients"`: a list of coefficients in radians.
  - when using the poly55 model: [p00, p10, p01, p20, p11, p02, p30, p21, p12, p03, p40, p31, p22, p13, p04, p50, p41, p32, p23, p14, p05]. Coefficients can be omitted starting from the right. Omitted coefficients will have their value set to 0.0.

Note: The coefficients are applied to a normalized data, where the polynomial data is centered at zero mean and scaled to unit standard deviation. For details see IMX316_AppNote_CameraCalibration(E)_28112017.pdf .

15. Configuration File Specifications

One of the files required for the configuration of the camera is a yaml file containing the sequence of actions to be applied for the given calibration mode. The sample below is just an example, different from the one provided with the delivery.

```
sequence:
  - {device: backend, action: set_named_gpio_output_enable, name: gpio_fsc, value: 0}
  - {device: host, action: sleep, value: 0.15}
  - {device: backend, action: upload_firmware, filename: "*.pse"}
  - {device: laser_driver, action: set_register, address: 18, value: 0}
  - {device: backend, action: i2c_write, i2c address: 39, address: 2, value: 20}
```

The serialization format for the configuration file is as follow:

- 2 bytes magic number: 0x504D
- 4 bytes unsigned int holding the length in bytes of the packed data.
- packed data in msgpack format.

15.1 Conventions

The data structure should use only native python types (including lists and dictionaries).

Dictionaries should always use strings as keys. Keys must:

- Be lower case and use Snake case.
- Match the MATLAB convention.
- Have a length less than 63 characters.

Lists should use homogeneous types.

15.2 Built-in devices

Device	serialized token value
fpga	0
sensor	1
laser_driver	2
backend	3
mipi_bridge	4

15.3 Built-in commands

Command	serialized token value
upload_firmware	3
i2c_write	4
sleep	5
set_register	6
set_gpio	7
set_gpio_output_enable	8
custom_function	11

15.4 File meta data

The meta data of the configuration are stored in a dedicated dictionary of the msgpack payload. The following fields are serialized in a typical configuration file.

Field	serialized token value	Unit	Notes
uid	27		The UID of the mode -- see glossary
hardware_width	13		The number of pixels in a row
hardware_height	14		The number of pixels in a column -- includes the meta data line.
master_clock	31	Hz	The master clock frequency -- Needed for dynamic features of the library.
reset_length	3	seconds	The reset length
uframe_dead_length	8	seconds	The microframe dead length
frame_dead_length	9	seconds	The frame dead length
component_length	7	seconds	The component length
readout_length	10	seconds	The readout length
integration_length	6	seconds	The integration length
modulation_frequencies	2	Hz	A list of modulation frequencies - one per

			component
phase_steps	15	degrees	A list of phase steps - one per component
uframe_count	12		The number of microframes per frame
component_count	11		The number of components per microframe

15.5 Command specification

Each command is attached:

- An ‘action’ entry (int). See Built-in commands section.
- Additional command specific arguments.

15.5.1 set_gpio

The additional arguments are:

- A ‘device’ entry (int). See Built-in devices section.
- An ‘address’ entry (int). This is the GPIO address.
- A ‘value’ entry (int). This is the state to put the GPIO in.

15.5.2 i2c_write

The additional arguments are:

- A ‘device’ entry (int). See Built-in devices section.
- A ‘i2c_address’ entry (int). The 7 bits i2c address of the anonymous device.
- An ‘address’ entry (int). The full register address.
- An ‘address_length’ entry (int). The register address length.
- A ‘value’ entry (int). The value to be written.
- A ‘value_length’ (int). The value length.

15.5.3 set_register

The additional arguments are:

- A ‘device’ entry (int). See Built-in devices section.
- An ‘address’ entry (int). The full register address.
- A ‘lsb’ entry (int). The start bit of the register field.
- A ‘msb’ entry (int). The end bit of the register field.
- A ‘value’ entry (int). The value to be written.

15.5.4 sleep

The additional argument is:

- A ‘value’ entry (float). The duration of the sleep in seconds.

15.5.5 custom_function

The additional argument is:

- A ‘name’ entry (string). The name of the function to be called.

16. IU316-minikit Rom Layout

Name	Start Address	Size (bytes)	Comment/Example
Production data - Values are written in Little Endian order			
Magic number	0x0000	2	'RB' in ascii
Length of production data	0x0002	2	Length of the production data not including the magic number and itself
Serial number length	0x0004	1	The length of the serial number record
Serial number	0x0005	19	
crc_8 value	0x0018	1	Normal CRC-8 (x8 + x7 + x6 + x4 + x2 + 1)
prv number	0x0019	4	
Reserved	0x001D	4	Used to hold the calibration tool version - has been relocated in calibration data
software id	0x0021	4	
Calibration data			
Magic number	0x0040	2	'MP' in ascii
Length of calibration data	0x0042	4	Length of the calibration data not including the magic number and itself
Calibration data	0x0046	variable	The calibration data stored in MessagePack format. As of March 2017, a typical MessagePack record is around 500 bytes

The calibration application reads the production data section and writes the calibration data section.

17. Trademarks

- MATLAB is a trademark of MathWorks, Inc.
- Miniconda is a trademark of Anaconda, Inc.
- Python is a trademark of the Python Software Foundation.
- PyCharm is a trademark of JetBrains s.r.o.
- Windows, Visual Studio are trademark of Microsoft Corporation.
- Arduino is a trademark of Arduino AG.

18. Update History

Revision	Date	Description	Author
0.1	2017-09-29	Initial release	SoftKinetic
0.2	2017-10-26	Adding more details about the sources	SoftKinetic
0.3	2017-12-21	Adding more details about the sources	SoftKinetic
0.4	2018-03-20	Supporting the IU316_minikit. Split driver descriptions.	SDS
0.5	-	(Intentionally skipped)	
0.6	2018-06-04	Supporting “table sweep”-based cyclic error correction for better camera module calibration.	SDS