Development of *in vivo* volumetric imaging method for mouse brain utilizing multibeam scanning two-photon microscopy

Ataka, Mitsutoshi

Doctor of Philosophy

Department of Physiological Sciences, School of Life Science, The Graduate University for Advanced Studies, SOKENDAI

Division of Biophotonics, National Institute for Physiological Science, National Institutes of Natural Sciences

Summary

Two-photon microscopy (2PM) is a laser scanning fluorescence microscopy that uses a two-photon excitation process and has been widely used in the neuroscience field as a robust tool for *in vivo* observation of neuronal activities in the mouse brain. For single-beam scanning 2PM, galvanometer-based scanning mirrors are usually equipped and the focal plane is raster-scanned by a single focus. Meanwhile, for multibeam scanning 2PM, a confocal spinning-disk scanner has been implemented for higher temporal resolution (up to 333 frame/second). This scanner splits a single excitation beam into hundreds of foci at the focal plane through the microlens-array. The fluorescence signals at the focal plane are relayed to the image plane and captured by a two-dimensional detector. In addition, a 40–100-fold increase in imaging speed compared with a single-beam scanning 2PM has been demonstrated. Moreover, volumetric imaging approaches based on the 2PM with galvanometer-based scanning mirrors were recently proposed for elucidating neuronal computations *in vivo.* To observe the neuronal activities in the mouse brain, three-dimensional (3D) scanning approaches with both sufficient temporal resolution and penetration depth of the excitation light are required. In this study, a volumetric imaging system using multibeam scanning 2PM with a spinning-disk scanner and high-peak power excitation laser source was proposed. To assess its applicability to *in vivo* volumetric imaging, first, the penetration depth limitation of the multibeam scanning 2PM in living mouse brains was experimentally confirmed. As a result, dendritic fibers were visualized at a depth of over 300 μm. Second, *in vivo* multiplane Ca^{2+} imaging was performed with a piezo *z*-scanner, and Ca^{2+} transients

were recorded at depths of 140 μm (single-plane) and 80–100 μm (three-planes). Next, using an electrically tunable lens (ETL), continuous axial scanning mechanics was introduced to improve the proposed volumetric imaging system; this improved imaging system is called the multibeam continuous axial scanning 2PM (MCAS-2PM) system. Using the MCAS-2PM system, a 1-μm bead phantom was observed and clearly resolved in the 3D volume as a *z*-projection stack with negligible axial spatial gaps. *In vivo* volumetric Ca^{2+} imaging was also performed with a synthetic Ca^{2+} indicator, Cal-590 AM, in the primary visual cortex of a mouse. As a result, spontaneous Ca^{2+} transients were successfully recorded in neurons up to a 155 μm depth from the brain surface with a 200 × 200 × 36 μm^2 field of view. Finally, to improve the brightness of the fluorescence image, a chirped pulse amplification (CPA) system with diffraction gratings and a previously reported Yb-doped fiber amplifier were incorporated. The CPA output had an average power of 12.0 W and a pulse width of ~1 ps. It is expected that, compared with using the original excitation light source, a roughly 3-fold brighter fluorescence image can be obtained.

In the MCAS-2PM system, to increase the penetration depth of Ca^{2+} imaging up to 200–300 μm or the mouse cortical layer 2/3, a higher peak power of excitation laser pulses is still required. The excitation laser source with a low repetition rate might be effective in multibeam scanning 2PM. Nevertheless, there are several techniques to enhance the imaging system and specimen, such as optimizing the detection system, efficiency of illumination, and localization of fluorescence probe.

In conclusion, a multibeam scanning 2PM-based volumetric imaging system for observing living mouse brains is proposed. To realize 3D scanning, lateral scanning with a spinning-disk and continuous axial scanning with an ETL were combined. With the proposed system, *in vivo* volumetric Ca^{2+} imaging was performed in living mouse brains up to a 155 μm depth from the surface. With further improvements, the proposed system can be a practical volumetric imaging system with a simple design and efficient scanning capability. Such features may make volumetric imaging widespread and thereby open new neuroscience pathways for many researchers.

Acronyms

2PM	Two-photon microscopy
DAQ	Data acquisition
ETL	Electrically tunable lens
EYFP	Enhanced yellow fluorescent protein
FOV	Field of view
FTL	Fourier-transform limited pulse
FWHM	Full width at half maximum
GDD	Group delay dispersion
MCAS-2PM	Multibeam continuous axial scanning two-photon microscopy, demonstrated in this study
MIP	Maximum intensity projection
MLA	Microlens-array
NA	Numerical aperture
OME	Open microscopy environment
PSF	Point spread function
ROI	Region of interest
SR101	Sulforhodamine 101
YDFA	Yb-doped fiber amplifier

Index

1. Introduction

1.1. *In vivo* two-photon imaging

Two-photon excitation is a nonlinear optical phenomenon in which a single fluorophore simultaneously absorbs two photons that have a wavelength of nearly twice that of one-photon excitation. The efficiency of two-photon excitation is quadratically proportional to the optical intensity. Therefore, two-photon excitation occurs at the limited spot where the laser light is focused. Denk *et al.* (1990) used a femtosecond Ti:Sa laser light source for two-photon imaging in biology as a pioneering work of two-photon microscopy (2PM). Such an ultrashort pulsed laser light used in 2PM typically has a near-infrared wavelength to excite fluorophores, which emit the visible wavelength light. Notably, in biological specimens containing water molecules and biomolecules, such as hemoglobin, light of near-infrared wavelength is less scattered or absorbed than light of visible wavelength. Thus, 2PM has the advantages of a large penetration depth and less invasiveness for *in vivo* observation.

In vivo 2PM technology has advanced with the novel excitation laser light sources in this decade. Using semiconductor-based high-power picosecond laser sources, hippocampal neurons were observed at a depth of ~1.4 mm in intact mouse brains (Kawakami *et al.*, 2013, 2015). Recently, with the state-of-the-art fluorescence Ca^{2+} probe, *in vivo* two-photon Ca^{2+} imaging has penetrated the mouse hippocampus at a depth of ~1.1 mm, which may reveal the rapid emergence of neuronal ensembles in hippocampal CA1 in response to neuronal activities in the sensory cortices (Inoue *et al.*, 2019). As

they represented, 2PM has been widely used in the neuroscience field as a robust tool for the *in vivo* observation of neuronal activities in mouse brains with single-cell or synaptic resolution.

1.2. Multibeam scanning 2PM with a spinning-disk

Multibeam scanning fluorescence microscopy splits a single excitation beam into multiple beams to simultaneously excite fluorophores at multiple foci. The fluorescence signals at the focal plane are relayed to the image plane and captured by a two-dimensional (2D) detector, such as an eyepiece, a charge-coupled device (CCD) camera, or a complementary metal-oxide-semiconductor (CMOS) imager. In single-beam scanning microscopy equipped with galvanometer-based scanning mirrors, the frame rate primarily depends on the speed of the moving mirrors. Meanwhile, in the multibeam approach, the maximum frame rate increases with the number of foci up to the maximum acquisition rate of the 2D detector. By applying these mechanisms to 2PM, multifocal multiphoton microscopy was proposed in the late 1990s, and a 40–100-fold increase in the efficiency of illumination compared with single-beam 2PM was demonstrated (Bewersdorf *et al.*, 1998; Straub & Hell, 1998).

For multibeam scanning, a spinning-disk scanner is usually employed. In this scanner, microlenses are arranged in a spiral pattern of a constant pitch on the disk, and its rapid rotation enables lateral scanning. The spinning microlens-array (MLA) disk corresponds optically to galvanometer-based scanning mirrors and a scan lens in the single-beam scanning microscope configuration. Several spinning-disk

scanners comprise an MLA disk with the pinhole-array disk located at the focal length of the microlenses. A pinhole-array disk enhances both axial and lateral spatial resolution by blocking the out-of-focus fluorescence, which is scattered or emitted by reabsorption (Fujita *et al.*, 2000).

Recently, the application of multibeam scanning fluorescence microscopy has been expanded to the living mouse brain. A prior study performed *in vivo* wide-field Ca^{2+} imaging in the neocortex in a 1-mm^2 field of view (FOV) at a single-bouton resolution using a confocal spinning-disk scanner and 8K camera (Yoshida *et al.*, 2018). In another study, *in vivo* two-photon imaging was accelerated up to a 1-kHz frame rate by combining an MLA and line scan using a galvanometer-based scanning mirror (Zhang *et al.*, 2019). Confocal spinning-disk scanners have been widely used for one-photon imaging of biological specimens, including mouse brains (Ikegaya *et al.*, 2004; Lam *et al.*, 2014; Yoshida *et al.*, 2018). However, such systems have rarely been applied to two-photon imaging in living mouse brains.

A major issue of the multibeam scanning 2PM which split the excitation laser beam into hundreds of beams is that the lower photon density at the foci decreases the two-photon fluorescence intensity. To address the issue, in this decade, confocal spinning-disk two-photon imaging was optimized by modifying the pinhole size and pitch (Shimozawa *et al.*, 2013) as well as introducing a high-peak power excitation laser light source (Otomo *et al.*, 2015). Based on these advances, we had conceived that *in vivo* observation in mouse cortices could be performed using confocal spinning-disk 2PM.

1.3. Volumetric imaging

Neural circuits are distributed over large volumes in the brain. To elucidate neural computations *in vivo,* three-dimensional (3D) scanning techniques with sufficient temporal resolution and penetration depth of the excitation light are required. The concept of volumetric imaging entails scanning in the axial direction. Various approaches have been proposed for efficient axial scanning in the past two decades, mainly in point scanning microscopy systems equipped with galvanometer-based *xy*-scanning mirrors. Here, some of the volumetric imaging approaches were summarized below: point spread function (PSF) engineering, temporal multiplexing, random-access scanning, and light-field microscopy.

First, the PSF engineering approach, particularly for an axially elongated focus (e.g., a Bessel beam), has been applied to volumetric imaging. A Bessel beam is generated by forming an annular-shaped beam at the pupil plane of the objective lens using a diffractive optical element (Botcherby *et al.*, 2006), a spatial light modulator (Lu *et al.*, 2017), or an axicon lens (Thériault *et al.*, 2013). Bessel beam-based volumetric imaging enabled the acquisition of a *z*-projection image of 3D volume by single lateral scanning (Song *et al.*, 2017; Lu *et al.*, 2020). Moreover, several techniques to resolve the axial position of fluorescence signals have been proposed (Song *et al.*, 2017; Gao *et al.*, 2022; Kozawa *et al.*, 2022).

Second, the temporal multiplexing of an excitation beam increases the pulse number for scanning numerous voxels. By splitting excitation pulses and changing their optical paths, each pulse can be focused on a different position with several nanoseconds delay. Further, the signal-to-noise ratio can

be maximized by 1 pulse/voxel excitation and synchronized detection (Prevedel *et al.*, 2016). Parallel lateral scanning based on highly optimized multiplexed microscopy enabled the recording of Ca^{2+} activities from 12,000 and 1,000,000 neurons at a time (Weisenburger *et al.*, 2019; Demas *et al.*, 2021).

Third, another approach is to optimize the scanning trajectory. Using 2D spiral trajectories and axial scanning with a piezo-*z*-scanner, neuronal and glial Ca^{2+} dynamics were visualized in mouse brains (Göbel *et al.*, 2007). In the random-access scanning approach (Reddy *et al.*, 2008), the scanning area is restricted to the region of interest (ROI) to improve the efficiency of spatial sampling. A recent study implemented holographic patterning (5 × 5 foci) by pulse-to-pulse phase modulation using an excitation laser source with a 40-kHz repetition rate and stably observed orientation tuning of neurons in cortical layers 2/3 and 5 of the mouse primary visual cortex (Akemann *et al.*, 2022).

Fourth, in imaging approaches using a 2D detector, light-field microscopy has been applied to volumetric imaging of the mouse brain. Light-field microscopy employs visible excitation light, and fluorescence signals are detected through the MLA placed before a camera. 3D information on fluorescence emission is collected in a single snapshot and embedded in the 2D image. Therefore, a 3D volume is computationally reconstructed from multiple 2D images containing different 3D information (Levoy *et al.*, 2006). Light-field microscopy combined with a head-mounted miniaturized microscope and a state-of-the-art signal-extraction algorithm visualized neuronal activities in the hippocampus of a freely moving mouse at a depth of 360 μm (Nöbauer *et al.*, 2017; Skocek *et al.*, 2018).

Although the technologies for imaging over a large volume have advanced in recent years, volumetric imaging with a multibeam scanning 2PM approach using a spinning-disk or MLA instead of galvanometer-based scanning mirrors has rarely been reported. Indeed, there is a principal issue of low photon density at hundreds of foci in the multibeam scanning 2PM. However, since volumetric imaging requires the acquisition of numerous voxels, the multibeam approach that of the higher pixel (voxel) rate compared to the single-beam approach is advantageous. For these reasons, this study tackled *in vivo* volumetric imaging in the mouse brain with an improved multibeam scanning 2PM system.

1.4. Excitation beam amplification

In multibeam two-photon excitation with a spinning-disk scanner, an incident beam is split into hundreds of beams, and the peak power of pulses is inversely proportional to the number of splits. A decrease in the peak power of pulses decreases fluorescence yields proportionally. A possible solution to this issue is to amplify the excitation beam to compensate for the decrease in peak power. Chirped pulse amplification (CPA) is a well-established technique for achieving very high-power ultrashort laser pulses (Strickland & Mourou, 1985; Eidam *et al.*, 2010; Ogino *et al.*, 2013). A CPA system comprises three optical circuits: a stretcher, an amplifier, and a compressor. As a summary of the CPA workflow, first, seed pulses are stretched in the time domain to reduce the peak power typically using diffraction gratings. Second, seed pulses that propagate the gain medium are amplified by stimulated

emission. Finally, the amplified pulses are compressed to obtain high-peak-power pulses. Therefore, the peak power during amplification is suppressed to avoid damage to the laser gain medium and nonlinear effects. Such pulse amplification technologies have been implemented in imaging systems, particularly in the multifocal excitation approach (Weisenburger *et al.*, 2019; Zhang *et al.*, 2019; Demas *et al.*, 2021; Akemann *et al.*, 2022).

1.5. Outline of this study

In this study, a concept of volumetric imaging using multibeam scanning 2PM with a spinning-disk scanner and high-peak power excitation laser source is proposed. To assess its applicability to *in vivo* volumetric imaging, first, the penetration depth limitation of the multibeam scanning 2PM in living mouse brain was experimentally confirmed, and *in vivo* multiplane Ca^{2+} imaging was performed with a piezo-*z*-scanner. Next, using an electrically tunable lens (ETL), continuous axial scanning mechanics was introduced into the optical system and a volumetric imaging system, comprising a program to control installed devices, four-dimensional (4D) image reconstruction, calibration procedure for objective lenses, optical layout, and microscope stage, was developed; this system is called the multibeam continuous axial scanning 2PM (MCAS-2PM) system. Using the MCAS-2PM system, volumetric imaging was performed with a 1-μm bead phantom and living mouse brain. In addition, *in vivo* volumetric Ca^{2+} imaging was performed with a synthetic Ca^{2+} indicator in the mouse primary visual cortex, and spontaneous Ca^{2+} activities in neruons were visualized. Finally, to improve the

brightness of the fluorescence image, a CPA system was designed using diffraction gratings and a previously reported Yb-doped fiber amplifier (YDFA; Kanazawa *et al.*, 2014; Kawakami *et al.*, 2015). The CPA output had an average power of 12.0 W and a pulse width of ~1 ps. It is expected that, compared with using the original excitation light source (average power: ~4 W, pulse width: ~350 fs), a roughly 3-fold brighter fluorescence image can be obtained.

2. Materials and Methods

2.1. Volumetric imaging system

2.1.1. Multibeam lateral scanning by a spinning-disk scanner

A confocal spinning-disk scanner (CSUMPΦ100, Yokogawa Electric Corp.) was employed for scanning the focal plane. The rotation speed of the spinning-disk was set up to 10,000 rpm. The excitation beam was expanded to use the entire area of the detection window and split into several hundreds of beamlets by an MLA. Each beam was focused on multiple points through the objective lens (Figure 2.1a). The fluorescence emission was passed through the pinhole-array disk to block the residual out-of-focus fluorescence signals and a long-pass dichroic mirror (DM; 750-nm cutoff wavelength).

2.1.2. Continuous axial scanning by an ETL

Axial scanning of MCAS-2PM was implemented by adjusting the divergence angle of the excitation beam before introducing it into the objective lens (Figure 2.1b). A convergence beam makes the focal spot closer to the objective lens than the nominal working distance. Inversely, a divergence incident beam makes the focal spot far from the objective lens for achieving axial scanning. An ETL enables the axial scanning based on this scheme. Compared to conventional z-scanners such as a piezo objective scanner, an ETL has the such low inertia of the moving part that resulting in a faster response

time (Grewe *et al.*, 2011; Chen *et al.*, 2021). The ETL (EL-16-40-TC-VIS-5D, Optotune Switzerland AG) was placed just before the objective lens to change the divergence of the incident beam. The focal length of the ETL was controlled using a lens driver (Lens Driver 4i, Optotune Switzerland AG) connected to the data acquisition (DAQ) board (USB-6353, National Instruments Corp.).

A periodic waveform to drive the ETL was selected from three waveforms: sine, triangular, and sawtooth (Figure 2.2). First, the sine waveform causes the nonuniform motion of the focal points; therefore, the axial pitches of each captured frame must differ, and many frames are captured near the edges of the axial range. Second, the sine and triangular waveforms cause bidirectional scanning; thus, such drive waveforms update axial positions twice in a short time near the edges, and the update rate for each axial position becomes heterogeneous. Meanwhile, the sawtooth waveform is advantageous for the aforementioned problems (Figure 2.2b, c); therefore, it is adapted as a drive waveform of the ETL (Figure 2.1c). Uniform motion and unidirectionality are essential features of the drive waveform for continuous axial scanning with a fixed exposure time.

2.1.3. Coordinated control of camera and ETL for volumetric imaging

To control the developed imaging system using analog voltage signals with sub-millisecond time resolution, an sCMOS camera (ORCA Fusion BT, Hamamatsu Photonics K.K.) and an ETL were connected to their respective DAQ boards, and 1 output channel per DAQ was used to control each

device independently. The input terminal of the camera received rectangular waveform signals (0 or 3.3 V) as external readout triggers that end the last frame and begin the next exposure. The ETL was driven by a computer-generated periodic waveform through a USB-connected lens driver. The current range of the ETL, corresponding to the desired axial range, was mapped in the range of 0–5 V with 10-bit precision in the lens driver; i.e., the lens driver converted the signal from voltage to current. The analog voltage signals that independently output from the DAQ boards to the camera and ETL were routed to the input terminals and simultaneously recorded with timing information.

2.1.4. Transformation of the ETL focal length into axial position

First, in order to collect z-position of volume images acquired with ETL, z-stack image was captured by using a mechanical focus drive motor (96A404, Ludl Electronic Products, Ltd.) coupled to the focusing knob of the microscope. The z-stack image captured with the focus drive motor is a reference image that contains depth information (z-stack-REF). Meanwhile, the *xy*-image, captured using the ETL, has information on the applied voltage, namely, ETL current (z-stack-ETL; Figure 3.4a). Next, for every frame in z-stack-ETL, the best-matched frame in z-stack-REF, which has the highest cross-correlation, was identified (Figure 3.4b) as follows:

$$\text{Best-matched frames}(i,j)_{0\le j<N} = \text{Max}\{\,\text{Corr}(i,j)\,\}_{0\le i<M}$$

where I and j are the frame numbers of z-stack-REF and z-stack-ETL, respectively. $\text{Corr}(i,j)$ is a

cross-correlation between the *i*-th frame of z-stack-REF and the *j*-th frame of z-stack-ETL. *M* and *N* are the total frame numbers of z-stack-REF and z-stack-ETL, respectively.

Using the depth information derived from each frame in z-stack-ETL and its best-matched frame, the relationship between the ETL current and depth was obtained. Finally, the value of the coefficient of linear regression (μm/mA) was calculated from the relationship (Figure 3.4c). Linear regression was performed by *numpy.polyfit()* function in Python3. In this study, −3.55 μm/mA with 16× 0.8 numerical aperture (NA) objective (N16XLWD-PF, Nikon) was used (data shown in Subsection 3.4) to transform the desired axial range (μm) into the valid ETL current range (mA).

The imaging software that controlled the devices (the camera, ETL, focus drive motor, and spinning-disk scanner) was coded by C# (Microsoft). This allowed for *xy, xyz,* and *xyz-t* recording with a graphical user interface. To configure the ETL and camera parameters, C# source codes provided by Optotune Switzerland AG and Hamamatsu Photonics K.K. were, respectively, used in the software.

2.1.5. 4D image reconstruction and visualization

A virtual 3D (*xyz*) space with uniform voxel grids was prepared on the internal memory, and a 4D (*xyz-t*) image file with OME-TIFF format was created on the high-speed storage of a computer. The driving voltages and corresponding timings recorded by the DAQ boards containing the information on tick counts, ETL currents, and camera triggers (i.e., elapsed time, axial positions, and frame numbers,

respectively) were integrated to determine the axial range of each frame (*xy*-image). Using this information, the area of 3D space corresponding to each frame was updated frame-by-frame (Figure 2.1d) and the entire 3D information at the time was output as a part of the 4D image at a certain interval.

In image reconstruction of MCAS-2PM, following parameters were defined to determine the precision and output file size of the 4D image. First, dt [sec] is an interval time to append the updated 3D images to the 4D image file. Second, dz [μm/voxel] means a depth (or thickness) of a single voxel. Third, *xy*-binning changes the total pixel number of a *xy*-image and determines the width and height of a single voxel (e.g., if *xy*-binning is 2, the original image whose pixel size is 1 μm/pixel becomes the image of 2 μm/pixel and the output file size becomes 1/4). In this study, volume images were generally downscaled from the original images in 4D image reconstruction because of the limitation of the internal memory capacity and the difficulty in the visualization of several tens of gigabytes of images with the available software. The parameters for 4D image reconstruction are also specified in the Results section.

The image reconstruction code was written by Python3 and powered by the tifffile package (Gohlke, 2022) to create OME-TIFF image files. 3D and *z*-projection views were created using image analysis software (NIS-Elements AR, Nikon; ImageJ/Fiji, Schindelin *et al.*, 2012; Schneider *et al.*, 2012).

2.1.6. Parameters for volumetric imaging

Because the focal plane is continuously imaged by a camera and owing to the high simultaneity of lateral scanning by a spinning-disk, the imaging parameters of axial range and pitch can be configured freely. The relational expression of the key parameters for volumetric imaging with MCAS-2PM can be written as follows:

$$f_{xy} = f_z \cdot \frac{z_{range}}{z_{pitch}},$$

where f_{xy} [Hz] denotes the frame rate, f_z [Hz] denotes the volume rate, z_{range} [m] denotes the length of the depth of view, and z_{pitch} [m] denotes the axial range for each frame (*xy*-image). Notably, $f_z \cdot z_{range}$ means that the total travel distance per second along the *z*-axis, and f_{xy} separates the total distance into each frame. Therefore, these parameters determine that z_{pitch}. f_{xy} is a constraint on the other parameters because f_{xy} is limited by the maximum frame rate of a camera or the minimum exposure time needed to detect the fluorescence.

2.1.7. Optical configuration in two microscope systems

In the evaluation of the penetration depth of the multibeam 2PM on multiplane Ca^{2+} imaging experiments, an inverted microscope (Eclipse Ti-E, Nikon) system (Figure 2.3) with 25× 1.0 NA objective lens (4 mm W.D., XLPLN25XSVMP, Olympus; effective objective magnification is 27.78× because the focal length of the tube lens was different for each manufacturer) and a piezo-z-scanner

(P-721 PIFOC, Physik Instrumente GmbH & Co. KG) was used (Subsections 3.1–3.3). A spinning-disk scanner (CSUMP Φ 100, Yokogawa Electric Corp.) was connected to the camera port of the microscope. A Yb-based ultrashort laser light (1,042-nm center wavelength, ~4-W average power, 10-MHz repetition rate, 350-fs pulse width, femtoTrain, Spectra-Physics, Inc.) and a Ti:Sa laser light (tuned to 920-nm wavelength, ~1.4-W average power, 80-MHz repetition rate, ~100-fs pulse width, MaiTai DeepSee, Spectra-Physics) were introduced to the same optical path, and the excitation light sources could be selected by mechanical shutters (SSH-C2B, SIGMAKOKI Co. Ltd.). Laser intensity was adjusted using a Glan-laser polarizer (GL10-A, Thorlabs, Inc.) with a half-wave plate mounted on a motorized rotation stage (PRM1/MZ8, Thorlabs, Inc.). The excitation beam was expanded by plano-convex lenses to be fitted for the imaging FOV. Fluorescence signals of GCaMP7 and SR101 were separated by a DM (cutoff wavelength of 560 nm) and captured by an EM-CCD camera (512 × 512 pixels, 16 × 16 μm^2/pixel, iXon Ultra 897, Andor Technology Ltd.) as two-channel images using half of the sensor area for each image (i.e., 512 × 512 pixels sensor was used as two 512 × 256 pixels sensors; W-VIEW GEMINI, Hamamatsu Photonics K.K.).

In the construction of the MCAS-2PM system for the volumetric imaging experiments, an upright microscope system (BX51, Olympus; Figure 2.4) with 16× 0.8 NA objective lens (3 mm W.D., N16XLWD-PF, Nikon; effective magnification is 14.4×) and an ETL (EL-16-40-TC-VIS-5D, Optotune Switzerland AG) was used for conventional *in vivo* brain imaging of mice. As an upright microscope system, the spinning-disk scanner was installed at ~400 mm height from the optical table

equal to the excitation light port of the microscope. Fluorescence signals were captured as monochromatic images by an sCMOS camera (2,048 × 2,048 pixels, ORCA Flash4.0 V3, Hamamatsu Photonics K.K. or 2,304 × 2,304 pixels, ORCA Fusion BT, Hamamatsu Photonics K.K.).

2.1.8. Preparation of bead samples

A fluorescence bead sample (FluoSpheres carboxylate-modified microspheres, 0.2 μm, yellow-green fluorescent (505/515), Molecular Probes, Inc.) was used to evaluate the PSF of the MCAS-2PM system. Another fluorescence bead sample (FluoSpheres carboxylate-modified, 1.0 μm Nile red (535/575), Molecular Probes, Inc.) was used to examine the relationship between volume rate and axial pitch in volumetric imaging. Each bead sample was dissolved in 1% agarose gel. The solution was briefly mixed using a vortex mixer after melting the agarose and fixed in a Φ35-mm glass bottom dish.

2.1.9. PSF evaluation

The 0.2-μm yellow-green bead phantom was observed using a 60× 1.2 NA objective (UPLSAPO60XW, Olympus) with a 1,042-nm excitation beam, a setup similar to a previous report (Otomo *et al.*, 2015). The lateral and axial pixel sizes were ~43.3 and ~200 nm/pixel, respectively. The lateral pixel size was derived from the 60× objective, 2.5 × intermediate relay lenses, and 6.5 × 6.5-μm^2/pixel sensor of the camera. The intensity profiles of the fluorescence images of beads were fitted by the Gaussian function

and their full width at half maximum (FWHM) values were calculated for the *x*-, *y*-, and *z*-axis using a custom code of ImageJ/Fiji.

2.2. *In vivo* volumetric Ca^{2+} imaging

2.2.1. Animals

Male and female Thy1-EYFP-H (H-line; Feng *et al.*, 2000) and GLT-1-G-CaMP7 (G7NG817; Monai *et al.*, 2016) transgenic mice (7–12-week-old) were used for *in vivo* imaging experiments (Subsections 3.1–3.3). H-line mice that express an enhanced yellow fluorescence protein (EYFP) in the neocortical and hippocampal neurons (Porrero *et al.*, 2010) were used for the deep brain imaging of the fine cortical structures. G7NG817 mice that express a green Ca^{2+} probe G-CaMP7 in astrocytes and neurons were used for *in vivo* Ca^{2+} imaging. Male wild-type C57BL/6J mice (6–9-week-old) were used to perform *in vivo* volumetric Ca^{2+} imaging with a synthetic Ca^{2+} indicator. All mice were housed under a 12 h/12 h light/dark cycle.

2.2.2. Cranial window surgery

The mice were anesthetized with 0.5%–1.5% isoflurane, and their body was warmed by a disposable heat pad during experiments. Local anesthesia with 2% xylocaine was applied to the surgical field, and their skin was incised to expose their skull. A custom-made head chamber (made from a plastic dish or a stainless-steel plate) was attached to the skull centered on the right parietal bone. To decrease the intracranial pressure and loosen the dura mater, 20% mannitol (Terumo Corp.) or glycerol (Taiyo Pharma Co., Ltd.) was administered via intraperitoneal injection (15 μL/g). About 15 min after the

administration, the skull overlying the parietal lobe was partially removed for a ~4.2-mm diameter circle using a dental drill. The exposed brain was sealed by a Φ4.2-mm glass coverslip (0.17-mm thickness; Matsunami Glass Ind., Ltd.) with instant glue (Aron Alpha, Toagosei Co., Ltd.) or an ultraviolet curable resin (Luxa Flow Star, Yoshida Dental Trade Distribution Co., Ltd.).

For specific labeling of astrocytes *in vivo* (Subsections 3.2 and 3.3), a red fluorescent dye, sulforhodamine 101 (SR101), was permeated into the brain (Nimmerjahn *et al.*, 2004). After removing the skull, 50-μM SR101 in saline was exposed to the brain surface where the dura mater remained for 10 min to avoid causing seizure-like neuronal activity (Rasmussen *et al.*, 2016).

For multicell bolus loading of the Ca^{2+} indicator (Subsections 2.2.3 and 3.6), the dura mater was removed for smooth insertion of a glass pipette, and the brain was left unsealed. After injecting the Ca^{2+} indicator into the cortex, the exposed brain area was sealed as described above.

The above experiments were performed following the recommendations in the Guidelines for the Care and Use of Laboratory Animals of the Animal Research Committee of Hokkaido University. The procedures were approved by the Institutional Animal Care and Use Committee of the National University Corporation Hokkaido University and National Institutes of Natural Sciences. The facility used for the care and management of the laboratory animals was approved by the Institutional Animal Care and Use Committee of Hokkaido University and National Institute for Physiological Sciences.

2.2.3. Bolus loading of synthetic Ca^{2+} indicator

The multicell bolus loading approach (Stosiek *et al.*, 2003; Tischbirek *et al.*, 2019) was employed to introduce synthetic Ca^{2+}-sensitive acetoxymethyl (AM) dyes into the cortex. The AM ester groups of Ca^{2+}-sensitive AM dyes are cleaved from the Ca^{2+}-sensitive dye molecules by esterases inside the cell, and the dye becomes fluorescent. In this study, because of its compatibility with a 1,042 nm excitation laser source, a red-shifted Ca^{2+} indicator, Cal-590 AM, that was effectively excited by 1,050 nm (Tischbirek *et al.*, 2015) was used.

Cal-590 AM dye (50 μg, AAT Bioquest, Inc.) was dissolved in DMSO + 20% pluronic F-127 (i.e., 10-mL DMSO + 2-g pluronic F-127). The dye was dissolved to 1.5 mM in a solution containing 150-mM NaCl, 2.5-mM KCl, and 10-mM HEPES at pH 7.4. Dextran conjugated Alexa Fluor 488 (25 μM, 10,000 MW; Thermo Fisher Scientific, Inc.) was also added to the solution for fluorescence guidance during bolus loading. A glass pipette was made from borosilicate glass capillaries (GD-1.5, Narishige) using a micropipette puller (P-1000, Sutter Instrument Co.). The diameter of the pipette tip was adjusted to approximately Φ20–30 μm under the bright field microscope (SMZ25, Nikon). The glass pipette was backfilled with ~10-μL dye solution.

Dye loading was performed via conventional single-beam scanning 2PM (A1R-MP, Nikon) with 16× 0.8 NA objective (N16XLWD-PF, Nikon) and a Ti:Sa laser (MaiTai eHP, Spectra-Physics, Inc.) tuned to a 920-nm wavelength. The glass pipette was angled ~30° and slowly inserted into the neocortex at 150–300-μm depth from the surface using a 3-axis manipulator. To eject the dye from the pipette,

0.01–0.025-MPa air pressure was applied for ~6 min using a custom tool (made from a syringe and F-clamp) with a pressure meter. About 30−45 min after the injection, spontaneous Ca^{2+} activity was briefly confirmed by the same microscope setup using 1,000−1,040-nm excitation wavelength. An *in vivo* Ca^{2+} imaging experiment with MCAS-2PM was started 1 h after the dye loading, and it lasted up to 6 h.

2.2.4. CPA

A custom-made YDFA (Kanazawa *et al.*, 2014; Kawakami *et al.*, 2015) was introduced to the excitation optical system. A pulse stretcher and compressor were designed using two pairs of diffraction gratings (reflective gratings: 25 × 25 mm^2, 1,200 grooves/mm, 900 nm blaze wavelength, 067R, Newport, Corp.; transmission gratings: 31.8 × 12.3 mm^2, 1,000 grooves/mm, 1,040 ± 20 nm designed wavelength, T-1000-1040-31.8 × 12.3-94, II-VI, Inc.). A Yb-doped double-clad fiber (Φ40-μm core diameter, 3-m length, DC-200/40-PZ-Yb, NKT Photonics) was used as a laser gain medium. A continuous wave (CW) laser diode (915-nm wavelength, 55 W, K915FA5RN-55.00W, BWT Beijing, Ltd.) was used for pumping the Yb^{3+} ions. An optical isolator (IO-5-1030-HP, Thorlabs, Inc.) was placed before the YDFA to block the pumping light in the backward direction to protect the laser source and other devices. Figure 3.8 shows a schematic of the CPA system. To construct a stretcher and compressor, the propagation distance of two diffraction gratings to introduce a group delay dispersion (GDD) was derived as follows (single-pass configuration; Backus *et al.*, 1998):

$$GDD = -\frac{\lambda^3 L}{2\pi c^2 d^2}\left[1-\left(\frac{\lambda}{d}-\sin\theta_{in}\right)^2\right]^{-\frac{3}{2}}, \quad (2.1)$$

where λ denotes the wavelength, L denotes the distance between diffraction gratings, d denotes the groove spacing on the diffraction gratings, θ_{in} denotes the incident angle of the beam, and c denotes the speed of light.

To obtain the GDD under the experimental condition, two equations were considered (Equations 2.2 and 2.3). First, the Fourier-transform-limited pulse (FTL) of the seed light (λ_0 =1,042 nm; $\Delta\lambda$ = 4.7 nm) was calculated as follows:

$$\Delta t \geq K\frac{{\lambda_0}^2}{\Delta\lambda \cdot c}, \quad (2.2)$$

where Δt [fs] denotes the pulse width (FWHM), $\Delta\lambda$ [nm] denotes the spectral width, λ_0 [nm] denotes the center wavelength, K is the time–bandwidth product (0.441 for Gaussian pulse shape), c is the speed of light.

Next, using the Δt~340 fs, the GDD necessary to obtain the pulse width τ_o was calculated as follows:

$$GDD = \frac{{\tau_i}^2}{4\ln 2}\left(\frac{{\tau_o}^2}{{\tau_i}^2}-1\right)^{\frac{1}{2}}, \quad (2.3)$$

where τ_i [fs] denotes the pulse width of the input light, and τ_o [fs] denotes the pulse width of the output light.

Using Equations (2.1) and (2.3), the grating distance was determined. The stretcher was designed to stretch the seed light to a pulse width of ~10 ps from ~350 fs (L = 97 cm, GDD ~1,225 × 10^3 fs^2). The compressor was designed to compress the amplified light to a pulse width of ~350 fs from ~10 ps (L = 103 cm, GDD~ −1,284 × 10^3 fs^2).

In this study, the laser intensity was measured using a digital power meter (PM100D, Thorlabs, Inc.) with power sensors (S170C, S470C, S425C-L, Thorlabs, Inc.). The pulse width in the range of a few hundred femtoseconds to several picoseconds was measured by an autocorrelator (Carpe, APE Angewandte Physik & Elektronik GmbH) assuming a Gaussian pulse shape.

2.2.5. Data analysis

In *in vivo* single-plane and multiplane Ca^{2+} imaging, the ROIs were defined to indicate spontaneous Ca^{2+} activities in neurons and astrocytes during the recording (Subsections 3.2 and 3.3). The ROIs were manually selected cell bodies based on the changes in fluorescence intensity of the Ca^{2+} probe, a difference in fluorescence intensities between cytoplasm and nucleus in neurons, and specific labeling of astrocytes. Fluorescence intensities were obtained from the Ca^{2+} probe channel using ImageJ/Fiji for each ROI and indicated as normalized intensities using Python3.

When evaluating the optical properties in MCAS-2PM, PSFs were measured using the 0.2-μm yellow–green bead sample (Subsection 3.4). The fluorescence profile across the pixel of the maximum

intensity was fitted by the Gaussian function, and its FWHM was calculated for the *x*-, *y*-, and *z*-axis.

In *in vivo* volumetric Ca^{2+} imaging with MCAS-2PM, Ca^{2+} activities were processed based on the maximum intensity *z*-projection (Subsection 3.6). The ROIs were manually selected and targeted cell bodies that showed typical Ca^{2+} transients. To illustrate the Ca^{2+} transients, fluorescence changes in the ROIs were calculated as follows:

$$Fluorescence\ change(x, y, t) = \frac{F(x, y, t)}{F_{min}(x, y)},$$

where $F(t)$ denotes the instantaneous fluorescence intensity, and F_{min} denotes the minimum fluorescence intensity along the time-axis for each pixel.

Fluorescence intensities ($F(x, y, t)$) were obtained by ImageJ/Fiji, and the fluorescence changes were indicated as normalized intensities using Python3.

3. Results

3.1. *In vivo* deep brain imaging by multibeam scanning 2PM

To assess the penetration capability of the multibeam scanning 2PM with a high-peak-power laser source with a 1,042-nm wavelength, *in vivo* imaging was performed in an intact brain of an adult Thy1-EYFP-H mouse through a cranial window using a 25× 1.0 NA objective. Each frame of the *z*-stack image had 512 × 512 pixels and was captured with an exposure time of 500 ms. Dendritic fibers were clearly resolved at various depths in the primary visual cortex in a 262 × 262-μm^2 FOV (Figure 3.1). The result shows that the maximum penetration depth is roughly over 300 μm from the brain surface, sufficiently reaching the cortical layer 2/3 region of mice. These results indicate that multibeam scanning 2PM with a high-peak-power laser can also be applied to *in vivo* observation of neuronal activities in mouse neocortical regions up to 300 μm depth or more.

According to a previous report (Kawakami *et al.*, 2015), young adult mice (4-week-old) can be observed at deeper depths than adult mice (over 8-week-old), which is expected in this system as well. However, young adult mice were not used in this study because adult mice have been generally used in behavioral experiments to perform learning tasks repeatedly and/or wait for the expression of fluorescence proteins.

3.2. *In vivo* Ca^{2+} imaging by multibeam scanning 2PM

In vivo Ca^{2+} imaging was performed at a 140-μm depth from the surface in the primary visual cortex of the GLT-1-G-CaMP7 mouse whose astrocytes were labeled by SR101 (Nimmerjahn *et al.*, 2004). To observe a green fluorescent Ca^{2+} probe, G-CaMP7, a Ti:Sa laser tuned to 920 nm was employed as the excitation light source. SR101 was simultaneously excited with G-CaMP7 at the wavelength of 920 nm, and each fluorescence signal was detected in separate channels through image-splitting optics with a DM (560-nm cutoff wavelength). Therefore, the pixel number of the EM-CCD camera was 255 × 121 pixels/channel. A time-lapse image was taken with an exposure time of 500 ms and a frame rate of 1.84 fps. G-CaMP7 and SR101 signals were overlaid as shown in two-color images at several time points (Figure 3.2a). Ca^{2+} transients in the ROIs showed spontaneous activities in both neurons and astrocytes (Figure 3.2b, c). These results indicate that multibeam scanning 2PM can be applied to Ca^{2+} imaging in the living mouse brain.

3.3. *In vivo* multiplane Ca^{2+} imaging by multibeam scanning 2PM with a piezo-*z*-scanner

To expand the imaging field along the depth direction, a piezo-z-scanner was used to change the focal depth in the range of 80–100 μm with a 10-μm *z*-step. Multiplane Ca^{2+} imaging was performed in a 262 × 131-μm^2 FOV at 1.3 stack/sec with an exposure time of 200 ms/frame. Figure 3.3a indicates that G-CaMP7 and SR101 signals were observed across three-planes. Similar to the previous results (Subsection 3.2), the astrocytic Ca^{2+} activities were recorded over different depths (Figure 3.3c). These results demonstrate that the Ca^{2+} activities from the three-planes in the relatively shallow region of the cortex can be obtained using multibeam scanning 2PM with a Ti:Sa laser.

3.4. Evaluation of the optical properties of the MCAS-2PM system

To examine the axial displacement amount for the ETL current, z-stack-ETL was compared with z-stack-REF based on cross-correlation. The schematic of the procedure to obtain the displacement slope is shown in Figure 3.4a. In general, the best-matched frames of the two z-stacks showed higher cross-correlation near the control condition of the ETL current (i.e., ETL current during acquisition of *z*-stack-REF; Figure 3.4b) because of the slightly different FOVs (Grewe *et al.*, 2011) and aberrations caused by different ETL current. Therefore, the depth and ETL current of the highly correlated pairs were used for the linear regression of the relationship between the ETL current and depth (Figure 3.4c). The displacement slope was measured for a 16× 0.8 NA objective lens, which resulted in −3.55 μm/mA. Using this relationship, the ETL drive waveform was transformed to axial positions during volumetric imaging.

To evaluate the spatial resolution of the volumetric imaging system, PSF was measured using the 0.2-μm yellow-green bead phantom and a 60× 1.2 NA objective lens, under the condition of with or without ETL (Figure 3.5a, b). Under the condition with ETL, the ETL current was set to the range of −10 to −13 mA to generate parallel light, which was derived using lens driver controller software (Optotune Switzerland AG) considering the temperature dependency. Under the condition without ETL, no ETL was used. The optical configuration is shown in Figure 2.4. As a result, under the condition with ETL, *x*-, *y*-, and *z*-PSF were 366 ± 7.9, 432 ± 13, and 1580 ± 45 nm (mean ± SEM), respectively. Under the condition without ETL, *x*-, *y*-, and *z*-PSF were 314 ± 13, 368 ± 18, and 1300 ± 25 nm, respectively. In

addition, PSFs under the condition with ETL at 0.8 NA objective lens were estimated using the equations of lateral and axial resolution (Amos *et al.*, 2011) and *x*-, *y*-, and *z*-PSF were 549, 648, and 2650 nm, respectively. The PSFs were sufficiently smaller than neuronal somata, and volumetric imaging could be performed with single-cell resolution.

3.5. Volumetric imaging of bead phantom via MCAS-2PM

As a proof of concept, volumetric imaging was performed in a 1-μm Nile red bead phantom. The 3D and *z*-projection views with volume rates of 1 and 3 Hz are shown in Figures 3.6a–d. The pulse amplifier was inserted between the Yb laser and spinning-disk scanner without a stretcher or compressor, and the average output power was 1.1 W under the objective lens. The parameters for image reconstruction were dt = 120 ms, dz = 2.0 μm/voxel, *xy*-binning = 4, and dx = dy = 0.722 μm/pixel. In this study, many beads were clearly resolved in volume (Figure 3.6c, d). Comparing the two volume rates (1 and 3 Hz), the *z*-pitch was elongated according to the higher volume rate, i.e., a faster change of focal plane occurred under the same exposure time (Figure 3.6e, f). Although the axial intensity distribution in the 3-Hz setup expanded more than that in the 1-Hz setup, the same beads were visualized. These results indicated that the sawtooth-driven continuous axial scanning provided uniformly sampled volumetric images with negligible spatial gaps along the *z*-axis.

3.6. *In vivo* volumetric Ca^{2+} imaging via MCAS-2PM

To investigate the performance of MCAS-2PM in the living mouse brain, spontaneous neuronal activity was observed in the primary visual cortex using a red-shifted Ca^{2+} indicator, Cal-590 AM. The schematic of the dye loading method is shown in Figure 3.7a. Briefly, a pipette containing the mixture of Cal-590 AM and dextran-conjugated Alexa Fluor 488 (10,000 MW) was carefully injected into the brain under fluorescence guidance with Alexa488 using conventional 2PM. At a depth range of 150–300 μm from the surface, air pressure was applied to the pipette to eject the dye. The spontaneous activity was briefly checked in the same microscope setup 30–45 min after the injection, and imaging experiments with MCAS-2PM were started 1 h after the injection.

Volumetric Ca^{2+} imaging was performed at a depth of 155 μm in a 200×200×36-μm^3 FOV with a 3.9-μm *z*-pitch at 1.5 volume/sec. Because the excitation pulses with a 1,042-nm wavelength rarely excite Alexa488, almost all fluorescence signals came from Cal-590 and were detected with a single channel. The parameters for image reconstruction were dt = 150 ms, dz = 1.0 μm/voxel, *xy*-binning = 1, and dx = dy = 0.722 μm/pixel. Figure 3.7b shows the maximum intensity projection (MIP) image along the time-axis, which visualizes the structures of several neuronal somata in volume. The changes in fluorescence intensity in neurons were indicated by *z*-projection views following the specific cell bodies at the different depths (144 and 135 μm) and *xy*-positions (Figure 3.7c). Spontaneous Ca^{2+} transients recorded from neuronal somata corresponding to the ROIs (Figure 3.7d) are shown in Figure 3.7e; the Ca^{2+} activities in neurons were observed. These results demonstrated that MCAS-2PM can

be applied to Ca^{2+} imaging up to a 155-μm depth from the surface of the living mouse brain.

3.7. CPA and evaluation of the amplified beam

To compensate for the decrease in the peak power of excitation pulses in the multibeam approach, a CPA system was designed. Figure 3.8 shows an optical layout for the CPA system. The average power and pulse width of YDFA and the pulse compressor were measured (Figure 3.9).

The stretcher was designed to up-chirp the seed light to 10 ps pulse width using two diffraction gratings. Yb laser pulses (average power: ~4 W, repetition rate: 10 MHz, pulse width: ~350 fs) were introduced to the first reflective diffraction grating (G1) through HWP1 because the diffraction efficiency was polarization-dependent. The first-order diffracted light was steered to the second diffraction grating (G2). Relay lenses with a focal length of 300 mm were placed between G1 and G2 to control the expansion of the diffracted beam propagating along the optical path. The angle of G2 was determined to shorten the optical path of the longer wavelength components. The stretched beam is parallel after G2; however, it has an elliptic beam shape and largely degrades the quality of fiber coupling in the YDFA. Therefore, the elliptic beam was corrected using cylindrical lenses with focal lengths of 400 mm (CL1) and 200 mm (CL2), a pinhole with a diameter of 200 μm, and a plano-convex lens with a focal length of 400 mm.

The YDFA was modified from what used in previous studies (Kanazawa *et al.*, 2014; Kawakami *et al.*, 2015). An optical isolator was placed before the fiber to transmit light in a forward direction only and block light in a backward direction to protect the laser source and other devices. HWP2 was placed to control the polarization direction of seed light incident to the Yb-doped fiber. DM1 is a short-pass DM

placed to safely collect the pumping laser light with a beam blocker. The diameter of the fiber core was Φ 40 μm (0.03 NA), and the incident beam must be focused on the smaller spot with a diameter of less than Φ 40 μm. Therefore, the diameter of the seed light was adjusted to Φ2.8 mm to be focused into the fiber core by a lens with a focal length of 50.2 mm. To pump Yb^{3+} ions, a CW laser light whose center wavelength is 915 nm and maximum output power is 55 W was introduced to the opposite surface of the fiber using a lens with a focal length of 13.9 mm. Because the pumping laser light was mainly passed through the outer layer of the fiber core, the focusing of the pumping light was briefly adjusted compared with the seed light. The seed light with an average power of 76 mW was amplified to a 22.9-W average output power through the fiber amplifier (Figure 3.9a) with a pump power of 53 W. The spectra of the amplified beam broadened and become a trimodal shape with increasing pump power (Figure 3.9b). The amplified beam was reflected by DM2 and propagated into the pulse compressor.

In the pulse compressor, HWP3 was placed to control the polarization direction for maximizing the diffraction efficiency of the transmission gratings (G3 and G4). These gratings were set to shorten the optical path of the shorter wavelength components, conversely to the stretcher. The amplified pulses were passed through G3 and G4 twice and their optical axis was slightly shifted to be extracted by a pickoff mirror. After the compressor, 12.0-W average output power with a ~1-ps pulse width was obtained (Figures 3.9c and d).

The above results indicate that the fluorescence intensity will increase roughly 3-fold higher than that

of the original Yb laser when using the CPA output as a two-photon excitation light source. Meanwhile, the FTL width of the seed light was ~340 fs and the pulse width of the CPA output was ~1 ps. These results may be due to the nonlinear effects that occurred in the Yb-doped fiber during amplification (Figure 3.9b), suggesting that the amount of pulse stretch was insufficient for the experimental configuration (discussed in Subsection 4.6).

4. Discussion

In this study, volumetric imaging in living mouse brains via multibeam scanning 2PM with a confocal spinning-disk scanner was demonstrated. First, the penetration depth of the multibeam scanning 2PM system was evaluated as over 300-μm depth below the brain surface (Subsection 3.1). In the multiplane imaging, Ca^{2+} activities were successfully recorded from cortical neurons and astrocytes up to 140- and 100-μm depths from the surface in the single- and tri-focal planes, respectively (Subsections 3.2 and 3.3). Second, to propose MCAS-2PM system, continuous axial scanning mechanics was implemented using an ETL for remote focusing with a sawtooth drive waveform, and the optical properties of the proposed imaging system were evaluated (Subsection 3.4). Third, using the proposed imaging system, volumetric imaging was performed in a 1-μm bead sample and a living mouse brain (Subsections 3.5 and 3.6). As a result, Ca^{2+} transients were successfully recorded with single-cell resolution up to 155 μm depth from the surface in a 200 × 200 × 36-μm^3 FOV with a volume rate of 1.5 Hz (Subsection 3.6). Finally, a CPA system was designed for further increase in the excitation laser intensity. The maximum output of the CPA system was an average power of 12.0 W and a pulse width of ~1 ps (Subsection 3.7).

4.1. The penetration depth of multibeam scanning 2PM with a spinning-disk scanner

In this study, the penetration depth of multibeam scanning 2PM was examined and dendritic fibers were observed at a depth of over 300 μm from the surface (Figure 3.1). This result is equivalent to or slightly better than that of a previous report using multibeam 2PM (≦300 μm; Zhang *et al.*, 2019). In the *in vivo* Ca^{2+} imaging, Ca^{2+} activities were observed at a depth of 140 μm with single-plane (Figure 3.2), 80–100 μm with tri-plane (Figure 3.3), and 120–155 μm with volume (Figure 3.7). Recent studies that performed multibeam one-photon Ca^{2+} imaging with a confocal spinning-disk scanner reported penetration depths of 160 μm (Iwasaki & Ikegaya, 2018) and 120 μm (Yoshida *et al.*, 2018). These results indicate that although continuous axial scanning was implemented in MCAS-2PM, it is still can be improved. Thus, in the following sections, further optimization of the proposed imaging system with several 2PM and pulsed laser technologies and specific improvements for multibeam approach are discussed.

4.2. Improvements of multibeam scanning 2PM with a spinning-disk scanner

The confocal spinning-disk scanner used in this study has MLA-disk and a pinhole-array disk, which improves the axial resolution by eliminating the out-of-focus fluorescence. However, in the continuous axial scanning approach, the axial resolution depends on the axial pitch (axial range of the *xy*-image)

instead of the PSF because the axial pitch is usually larger than the PSF for imaging with single-cell resolution (Figure 3.6). Therefore, bypassing a pinhole-array disk in the detection path (Fujita *et al.*, 2000) may increase the fluorescence intensity without significant degradation of the axial resolution in the 3D volume image.

In this study, the excitation laser beam had a Gaussian intensity profile, i.e., the laser intensity gradually decreased from the center to the outer in the transverse plane. Unlike single-beam scanning microscopy, this beam profile is projected to the focal plane in the multibeam scanning approach. Therefore, a flat-top (top-hat) beam profile is required for uniform illumination, and π-shaper (AdlOptica; Zhang *et al.*, 2019) and MLAs (Mahecic *et al.*, 2020) have been implemented. These techniques will enable us to avoid excess laser intensity at the center of the focal plane and thus increase the efficiency of illumination in the entire FOV.

Genetically encoded Ca^{2+} indicators have been actively developed in recent years (Chen *et al.*, 2013; Qian *et al.*, 2019; Mohr *et al.*, 2020). In general, to use such Ca^{2+} probes, the adeno-associated viruses, which express the fluorescence Ca^{2+} probes, are injected into the brain and the transfected cells express the Ca^{2+} probes. Recently, novel techniques that restrict the expression of Ca^{2+} probes to the neuronal soma have been reported (Chen *et al.*, 2020; Shemesh *et al.*, 2020). By removing the fluorescence signals emitted from axons and dendrites that overlapped with the soma, a signal-to-background ratio might be enhanced in *in vivo* volumetric imaging.

4.3. Continuous axial scanning for multibeam scanning volumetric imaging

The continuous axial scanning approach arose from a simple idea that reduces the dead time occurring in axial scanning (Subsection 3.3), which is generally adapted for multiplane imaging. Multiplane volumetric imaging is frequently implemented by repeating two steps: 1) depth-to-depth shift of the focal plane and 2) capturing the *xy*-image while the focal plane is stationary. However, such an approach costs the settling time for the damped oscillation of the focal plane in a few tens of milliseconds with a piezo *z*-scanner or ETL; hence, the frame rate is limited (Figure 3.3; Han *et al.*, 2019). In this study, the continuous axial scanning approach enabled the scanning of a 3D volume with no settling time (Figures 3.6 and 3.7). Notably, 3D volume was reconstructed as a *z*-projection stack with negligible axial spatial gaps (Figure 3.6). This feature enables the observation of all neurons in a specific region of the brain and thus will reveal the neuronal representations in a functionally equivalent circuit across animals (Endo *et al.*, 2020).

In the temporal multiplexing approach, many foci and their positions in the axial direction have been fixed for scanning the corresponding volume (e.g., changing the axial pitch, switching from 3D time-lapse to 2D time-lapse imaging; Weisenburger *et al.*, 2019; Demas *et al.*, 2021). Therefore, practically, to observe different FOVs, the optical system must be rearranged. By contrast, the MCAS-2PM system can seamlessly change the imaging parameters from the host computer without the rearrangement of the optical system because of the versatile combination of scanning mechanisms. Overall, MCAS-2PM is a simple, efficient, and practical approach to volumetric imaging.

4.4. Excitation light source for volumetric imaging

Excitation laser light source is a key component in improving the fluorescence intensity in 2PM. The number of photons absorbed per fluorophore per second, *Na*, is proportional to the product of average power and peak power, given as follows (Kawakami *et al.*, 2015):

$$Na \propto P_{ave} \cdot P_{peak} = P_{ave} \cdot \frac{P_{ave}}{f_{rep} \cdot \tau},$$

where P_{ave} denotes the average power [W], P_{peak} denotes the peak power [W], f_{rep} denotes the repetition rate [Hz], and τ denotes the pulse width [m].

The above relationship also shows that a high-peak power of laser pulses is achievable by different techniques: a decrease in the pulse repetition rate or pulse width or an increase in the average power. However, depending on the scanning approach, the technique to improve the excitation light source might differ.

In the volumetric imaging approaches based on lateral scanning with galvanometer-based scanning mirrors, the focal plane is scanned with a single or a few foci of the excitation beam. To improve the fluorescence intensity, these approaches require higher peak power of the excitation laser pulses because their focal spot is axially expanded (Lu *et al.*, 2020) or axially split into multiple foci (Demas *et al.*, 2021). However, such imaging systems still require a sufficiently high pulse repetition rate for scanning the focal plane with a certain resolution (lateral pitch) and volume rate. In general, in the

aforementioned volumetric imaging approaches, a higher peak power of excitation pulses should not only be obtained by decreasing the pulse repetition rate but also by 1) expanding the spectral width and 2) increasing the average power.

In the multibeam approach especially using MLA, because the single excitation beam is split into hundreds of foci at the focal plane, the volume rate should not be limited by a low pulse repetition rate. That is, the peak power of excitation pulses can be increased by decreasing the pulse repetition rate without deteriorating the frame rate, as was realized in previous studies (Otomo *et al.*, 2015; Zhang *et al.*, 2019). In addition, to use the excitation pulse trains with a low repetition rate (<1 MHz), the brightness of a two-photon fluorescence image has been increased through dark-state relaxation (Donnert *et al.*, 2007). Therefore, particularly in the multibeam approach, a sub-MHz repetition rate of the excitation laser light source might be effective in increasing fluorescence intensity in volumetric imaging.

4.5. Limitation of excitation laser intensity in the brain

Although using the high laser intensity of the excitation light source is effective for scanning a large volume in the brain, high-power illumination increase brain temperature. Recently, lasting heat-induced damage has been observed in the neocortex of mice by the illumination of femtosecond pulsed laser light with an average power of 300 mW under the objective lens in a 1 mm^2 FOV for 20 min

using single-beam scanning microscopy (Podgorski & Ranganathan, 2016). Another study reported that photodamage was observed in the cortex using a picosecond pulsed laser light source with an average power of 500 mW under the objective lens in a 0.25 mm^2 FOV (Kawakami *et al.*, 2015). In the limitation of the peak power, the ablative threshold for biological tissues was reported as 1.5–2.2 J/cm^2 for 800–1,450-nm wavelength with a pulse width of 100 fs (Olivié *et al.*, 2008). These reports suggest a limitation of excitation laser intensity in the living mouse brain, and this information might provide a general rule for designing the excitation light sources. Moreover, the strategies to overcome such an issue, not the scanning methods or development of laser light sources, may be available to further advance volumetric imaging technology.

4.6. Characteristics of amplified beam

In this study, the seed light passed through the designed CPA system was amplified to an average output power of 12.0 W and a pulse width of ~1 ps. Comparing the amplified and original beams, fluorescence intensity may increase three-fold. However, the pulse width of the amplified beam was wider than ~340 fs (Figure 3.9d), which is the FTL width of the seed, attributable to the nonlinear effects occurring in the Yb-doped fiber during amplification. As shown in Figure 3.9b, the spectral widths of the amplified beams expanded and no longer had a Gaussian profile as average output power increased. In addition, the pulse width measured after the compressor changed with an increase in average output power (Figure 3.9d). These results indicate that the nonlinear effects become

nonnegligible as the peak power of the amplified beams increases in the gain medium; thus, the amplified pulses could not be compressed to the expected pulse width of ~340 fs.

A previous study that designed a CPA system whose gain medium is similar to that in this study reported that the seed light with an average power of 175 W in a Yb-doped fiber with a pulse width of 120 ps at a 73 MHz repetition rate was amplified and finally obtained an average output power of 131 W and a pulse width of 220 fs (Röser *et al.*, 2005). Therefore, comparing the referenced study and this study, the peak power in the gain medium was estimated to be ~0.02 and ~0.23 MW, respectively. These estimations suggest that the seed light that stretched 12-fold wider (120 ps) might be amplified with negligible nonlinearity. To verify this possibility, a pulse stretcher that gives a larger GDD through the elongated grating distance will be constructed in future experiments.

4.7. Future perspective

In this study, we developed an imaging system with a high-peak power excitation laser light source and a continuous axial scanning technique that is simple and efficient for volumetric imaging. Using this system, *in vivo* volumetric Ca^{2+} imaging was successfully performed in living mouse brains up to a 155 μm depth from the surface in a 200 × 200 × 36 μm^3 FOV. To increase the penetration depth of Ca^{2+} imaging up to 200–300 μm, a region frequently targeted by *in vivo* observation and optogenetic stimulation (Carrillo-Reid *et al.*, 2016, 2019), a higher peak power laser light source is required. The

excitation laser source with a low repetition rate might be effective in multibeam scanning 2PM. Moreover, there are several techniques to enhance the imaging system and sampling, such as optimizing the detection system, efficiency of illumination, and localization of fluorescence probe. With these improvement techniques, MCAS-2PM can be a practical volumetric imaging system with a simple design and efficient scanning capability. Such features may make volumetric imaging widespread and thereby open new neuroscience pathways for many researchers. For example, the cortical representation of perceptual and cognitive states of the behaving animals can be elucidated by visualizing a set of neurons, or ensembles, that are coordinately active within a large 3D volume.

The ability to perform 3D scanning with negligible axial spatial gaps using only a spinning-disk and a liquid lens inserted before the objective lens contributes to the miniaturization of the microscope and control system. Therefore, the proposed system can be installed outside a laboratory without a large optical table, and medical applications, such as a quick 3D visualization of biological sections, are anticipated.

5. Conclusion

In this study, we proposed a multibeam scanning 2PM-based volumetric imaging system for observing the living mouse brain. To realize 3D scanning, lateral scanning by spinning-disk and continuous axial scanning by defocusing with an ETL were combined. Meanwhile, this imaging system has a simple optical layout compared with single-beam scanning 2PM-based volumetric imaging systems. The main reason for the simplicity of the multibeam approach is that the excitation beam is split into multiple foci by the spinning-disk and thus the manipulation of a single pulse is unnecessary. This feature may be paramount for volumetric imaging to be widespread in the future. Next, to demonstrate our MCAS-2PM system, *in vivo* volumetric Ca^{2+} imaging was performed. As a result, Ca^{2+} transients were successfully recorded in neurons up to a 155 μm depth from the surface in a 200 × 200 × 36 μm^3 FOV. This penetration depth can be increased by a higher intensity of excitation laser light because the fluorescence yields are quadratically proportional to the excitation laser intensity. To achieve this with higher peak power, particularly in the multibeam approach, an excitation light source with a low repetition rate may be effective. Thus, a CPA system was designed to increase the excitation laser intensity. The CPA output had an average power of 12.0 W and a pulse width of ~1 ps. The simple microscope configuration with efficient 3D scanning and the use of fiber amplifier technology to ensure the laser intensity for larger penetration depth in living mouse brains may serve as a model for future researchers to design improved volumetric imaging systems.

6. References

Akemann W, Wolf S, Villette V, Mathieu B, Tangara A, Fodor J, Ventalon C, Léger J-F, Dieudonné S & Bourdieu L (2022). Fast optical recording of neuronal activity by three-dimensional custom-access serial holography. *Nat Methods* **19,** 100–110.

Amos B, Mcconnell G & Wilson T (2011). Confocal microscopy. In *Handbook of Comprehensive Biophysics*. Elsevier.

Bewersdorf J, Pick R & Hell SW (1998). Multifocal multiphoton microscopy. *Opt Lett* **23,** 655–657.

Botcherby EJ, Juškaitis R & Wilson T (2006). Scanning two photon fluorescence microscopy with extended depth of field. *Opt Commun* **268,** 253–260.

Carrillo-Reid L, Han S, Yang W, Akrouh A & Yuste R (2019). Controlling Visually Guided Behavior by Holographic Recalling of Cortical Ensembles. *Cell* **178,** 447-457.e5.

Carrillo-Reid L, Yang W, Bando Y, Peterka DS & Yuste R (2016). Imprinting and recalling cortical ensembles. *Science* **353,** 691–694.

Chen L, Ghilardi M, Busfield JJC & Carpi F (2021). Electrically Tunable Lenses: A Review. *Front Robot AI* **8,** 678046.

Chen T-W, Wardill TJ, Sun Y, Pulver SR, Renninger SL, Baohan A, Schreiter ER, Kerr RA, Orger MB, Jayaraman V, Looger LL, Svoboda K & Kim DS (2013). Ultrasensitive fluorescent proteins for imaging neuronal activity. *Nature* **499,** 295–300.

Chen Y, Jang H, Spratt PWE, Kosar S, Taylor DE, Essner RA, Bai L, Leib DE, Kuo T-W, Lin Y-C, Patel M, Subkhangulova A, Kato S, Feinberg EH, Bender KJ, Knight ZA & Garrison JL (2020). Soma-Targeted Imaging of Neural Circuits by Ribosome Tethering. *Neuron* **107,** 454-469.e6.

Demas J, Manley J, Tejera F, Barber K, Kim H, Traub FM, Chen B & Vaziri A (2021). High-speed, cortex-wide volumetric recording of neuroactivity at cellular resolution using light beads microscopy. *Nat Methods* **18,** 1103–1111.

Donnert G, Eggeling C & Hell SW (2007). Major signal increase in fluorescence microscopy through dark-state relaxation. *Nat Methods* **4,** 81–86.

Eidam T, Hanf S, Seise E, Andersen TV, Gabler T, Wirth C, Schreiber T, Limpert J & Tünnermann A (2010). Femtosecond fiber CPA system emitting 830 W average output power. *Opt Lett* **35,** 94–96.

Endo K, Tsuchimoto Y & Kazama H (2020). Synthesis of Conserved Odor Object Representations in a Random, Divergent-Convergent Network. *Neuron* **108,** 367-381.e5.

Feng G, Mellor RH, Bernstein M, Keller-Peck C, Nguyen QT, Wallace M, Nerbonne JM, Lichtman JW & Sanes JR (2000). Imaging Neuronal Subsets in Transgenic Mice Expressing Multiple Spectral Variants of GFP. *Neuron* **28,** 41–51.

Fujita K, Nakamura O, Kaneko T, Oyamada M, Takamatsu T & Kawata S (2000). Confocal multipoint multiphoton excitation microscope with microlens and pinhole arrays. *Opt Commun* **174,** 7–12.

Gao Y, Xia X, Liu L, Wu T, Chen T, Yu J, Xu Z, Wang L, Yan F, Du Z, Chu J, Zhan Y, Peng B, Li H & Zheng W (2022). Axial gradient excitation accelerates volumetric imaging of two-photon microscopy. *Photonics Research* **10,** 687.

Göbel W, Kampa BM & Helmchen F (2007). Imaging cellular network dynamics in three dimensions using fast 3D laser scanning. *Nat Methods* **4,** 73–79.

Gohlke C (2022). *cgohlke/tifffile: v2022.5.4*. Available at: https://zenodo.org/record/6795861.

Grewe BF, Voigt FF, van 't Hoff M & Helmchen F (2011). Fast two-layer two-photon imaging of neuronal cell populations using an electrically tunable lens. *Biomed Opt Express* **2,** 2035–2046.

Han S, Yang W & Yuste R (2019). Two-Color Volumetric Imaging of Neuronal Activity of Cortical Columns. *Cell Rep* **27,** 2229-2240.e4.

Ikegaya Y, Aaron G, Cossart R, Aronov D, Lampl I, Ferster D & Yuste R (2004). Synfire chains and cortical songs: temporal modules of cortical activity. *Science* **304,** 559–564.

Iwasaki S & Ikegaya Y (2018). In vivo one-photon confocal calcium imaging of neuronal activity from the mouse neocortex. *J Integr Neurosci* **17,** 671–678.

Kanazawa S, Kozawa Y & Sato S (2014). High-power and highly efficient amplification of a radially polarized beam using an Yb-doped double-clad fiber. *Opt Lett* **39,** 2857–2859.

Kawakami R, Sawada K, Kusama Y, Fang Y-C, Kanazawa S, Kozawa Y, Sato S, Yokoyama H & Nemoto T (2015). In vivo two-photon imaging of mouse hippocampal neurons in dentate gyrus using a light source based on a high-peak power gain-switched laser diode. *Biomed Opt Express* **6,** 891–901.

Kawakami R, Sawada K, Sato A, Hibi T, Kozawa Y, Sato S, Yokoyama H & Nemoto T (2013). Visualizing hippocampal neurons with in vivo two-photon microscopy using a 1030 nm

picosecond pulse laser. *Sci Rep* **3,** 1–7.

Kozawa Y, Nakamura T, Uesugi Y & Sato S (2022). Wavefront engineered light needle microscopy for axially resolved rapid volumetric imaging. *Biomed Opt Express* **13,** 1702–1717.

Lam P-Y, Fischer RS, Shin WD, Waterman CM & Huttenlocher A (2014). Spinning Disk Confocal Imaging of Neutrophil Migration in Zebrafish. In *Neutrophil Methods and Protocols*, ed. Quinn MT & DeLeo FR, pp. 219–233. Humana Press, Totowa, NJ.

Levoy M, Ng R, Adams A, Footer M & Horowitz M (2006). Light field microscopy. In *ACM SIGGRAPH 2006 Papers*, pp. 924–934. Association for Computing Machinery, New York, NY, USA.

Lu R, Liang Y, Meng G, Zhou P, Svoboda K, Paninski L & Ji N (2020). Rapid mesoscale volumetric imaging of neural activity with synaptic resolution. *Nat Methods* **17,** 291–294.

Lu R, Sun W, Liang Y, Kerlin A, Bierfeld J, Seelig JD, Wilson DE, Scholl B, Mohar B, Tanimoto M, Koyama M, Fitzpatrick D, Orger MB & Ji N (2017). Video-rate volumetric functional imaging of the brain at synaptic resolution. *Nat Neurosci* **20,** 620–628.

Mahecic D, Gambarotto D, Douglass KM, Fortun D, Banterle N, Ibrahim KA, Le Guennec M, Gönczy P, Hamel V, Guichard P & Manley S (2020). Homogeneous multifocal excitation for high-throughput super-resolution imaging. *Nat Methods* **17,** 726–733.

Mohr MA, Bushey D, Aggarwal A, Marvin JS, Kim JJ, Marquez EJ, Liang Y, Patel R, Macklin JJ, Lee C-Y, Tsang A, Tsegaye G, Ahrens AM, Chen JL, Kim DS, Wong AM, Looger LL, Schreiter ER & Podgorski K (2020). jYCaMP: an optimized calcium indicator for two-photon imaging at fiber laser wavelengths. *Nat Methods* **17,** 694–697.

Monai H, Ohkura M, Tanaka M, Oe Y, Konno A, Hirai H, Mikoshiba K, Itohara S, Nakai J, Iwai Y & Hirase H (2016). Calcium imaging reveals glial involvement in transcranial direct current stimulation-induced plasticity in mouse brain. *Nat Commun* **7,** 11100.

Nimmerjahn A, Kirchhoff F, Kerr JND & Helmchen F (2004). Sulforhodamine 101 as a specific marker of astroglia in the neocortex in vivo. *Nat Methods* **1,** 31–37.

Nöbauer T, Skocek O, Pernía-Andrade AJ, Weilguny L, Traub FM, Molodtsov MI & Vaziri A (2017). Video rate volumetric Ca2+ imaging across cortex using seeded iterative demixing (SID) microscopy. *Nat Methods* **14,** 811–818.

Ogino J, Sueda K, Kurita T, Kawashima T & Miyanaga N (2013). High-Gain Regenerative Chirped-Pulse Amplifier Using Photonic Crystal Rod Fiber. *Appl Phys Express* **6,** 122703.

Olivié G, Giguère D, Vidal F, Ozaki T, Kieffer J-C, Nada O & Brunette I (2008). Wavelength dependence of femtosecond laser ablation threshold of corneal stroma. *Opt Express* **16,** 4121–4129.

Otomo K, Hibi T, Murata T, Watanabe H, Kawakami R, Nakayama H, Hasebe M & Nemoto T (2015). Multi-point Scanning Two-photon Excitation Microscopy by Utilizing a High-peak-power 1042-nm Laser. *Anal Sci*.

Podgorski K & Ranganathan G (2016). Brain heating induced by near-infrared lasers during multiphoton microscopy. *J Neurophysiol* **116,** 1012–1023.

Porrero C, Rubio-Garrido P, Avendaño C & Clascá F (2010). Mapping of fluorescent protein-expressing neurons and axon pathways in adult and developing Thy1-eYFP-H transgenic mice. *Brain Res* **1345,** 59–72.

Prevedel R, Verhoef AJ, Pernía-Andrade AJ, Weisenburger S, Huang BS, Nöbauer T, Fernández A, Delcour JE, Golshani P, Baltuska A & Vaziri A (2016). Fast volumetric calcium imaging across multiple cortical layers using sculpted light. *Nat Methods* **13,** 1021–1028.

Qian Y, Piatkevich KD, Mc Larney B, Abdelfattah AS, Mehta S, Murdock MH, Gottschalk S, Molina RS, Zhang W, Chen Y, Wu J, Drobizhev M, Hughes TE, Zhang J, Schreiter ER, Shoham S, Razansky D, Boyden ES & Campbell RE (2019). A genetically encoded near-infrared fluorescent calcium ion indicator. *Nat Methods* **16,** 171–174.

Rasmussen R, Nedergaard M & Petersen NC (2016). Sulforhodamine 101, a widely used astrocyte marker, can induce cortical seizure-like activity at concentrations commonly used. *Sci Rep* **6,** 30433.

Reddy GD, Kelleher K, Fink R & Saggau P (2008). Three-dimensional random access multiphoton microscopy for functional imaging of neuronal activity. *Nat Neurosci* **11,** 713–720.

Röser F, Rothhard J, Ortac B, Liem A, Schmidt O, Schreiber T, Limpert J & Tünnermann A (2005). 131 W 220 fs fiber laser system. *Opt Lett, OL* **30,** 2754–2756.

Schindelin J, Arganda-Carreras I, Frise E, Kaynig V, Longair M, Pietzsch T, Preibisch S, Rueden C, Saalfeld S, Schmid B, Tinevez J-Y, White DJ, Hartenstein V, Eliceiri K, Tomancak P & Cardona A (2012). Fiji: an open-source platform for biological-image analysis. *Nat Methods* **9,** 676–682.

Schneider CA, Rasband WS & Eliceiri KW (2012). NIH Image to ImageJ: 25 years of image analysis. *Nat Methods* **9,** 671–675.

Shemesh OA et al. (2020). Precision Calcium Imaging of Dense Neural Populations via a Cell-Body-Targeted Calcium Indicator. *Neuron* **107,** 470-486.e11.

Shimozawa T, Yamagata K, Kondo T, Hayashi S, Shitamukai A, Konno D, Matsuzaki F, Takayama J, Onami S, Nakayama H, Kosugi Y, Watanabe TM, Fujita K & Mimori-Kiyosue Y (2013). Improving spinning disk confocal microscopy by preventing pinhole cross-talk for intravital imaging. *Proc Natl Acad Sci U S A* **110,** 3399–3404.

Skocek O, Nöbauer T, Weilguny L, Martínez Traub F, Xia CN, Molodtsov MI, Grama A, Yamagata M, Aharoni D, Cox DD, Golshani P & Vaziri A (2018). High-speed volumetric imaging of neuronal activity in freely moving rodents. *Nat Methods* **15,** 429–432.

Song A, Charles AS, Koay SA, Gauthier JL, Thiberge SY, Pillow JW & Tank DW (2017). Volumetric two-photon imaging of neurons using stereoscopy (vTwINS). *Nat Methods* **14,** 420–426.

Stosiek C, Garaschuk O, Holthoff K & Konnerth A (2003). In vivo two-photon calcium imaging of neuronal networks. *Proc Natl Acad Sci U S A* **100,** 7319–7324.

Straub M & Hell SW (1998). Multifocal multiphoton microscopy: A fast and efficient tool for 3-D fluorescence imaging. *Bioimaging*; DOI: 10.1002/1361-6374(199812)6:4<177::AID-BIO3>3.0.CO;2-R.

Strickland D & Mourou G (1985). Compression of amplified chirped optical pulses. *Opt Commun* **56,** 219–221.

Thériault G, De Koninck Y & McCarthy N (2013). Extended depth of field microscopy for rapid volumetric two-photon imaging. *Opt Express* **21,** 10095–10104.

Tischbirek C, Birkner A, Jia H, Sakmann B & Konnerth A (2015). Deep two-photon brain imaging with a red-shifted fluorometric Ca2+ indicator. *Proc Natl Acad Sci U S A* **112,** 11377–11382.

Tischbirek CH, Noda T, Tohmi M, Birkner A, Nelken I & Konnerth A (2019). In Vivo Functional Mapping of a Cortical Column at Single-Neuron Resolution. *Cell Rep* **27,** 1319-1326.e5.

Weisenburger S, Tejera F, Demas J, Chen B, Manley J, Sparks FT, Martínez Traub F, Daigle T, Zeng H, Losonczy A & Vaziri A (2019). Volumetric Ca2+ Imaging in the Mouse Brain Using Hybrid Multiplexed Sculpted Light Microscopy. *Cell* **177,** 1050-1066.e14.

Yoshida E, Terada S-I, Tanaka YH, Kobayashi K, Ohkura M, Nakai J & Matsuzaki M (2018). In vivo wide-field calcium imaging of mouse thalamocortical synapses with an 8 K ultra-high-definition camera. *Sci Rep* **8,** 8324.

Zhang T, Hernandez O, Chrapkiewicz R, Shai A, Wagner MJ, Zhang Y, Wu C-H, Li JZ, Inoue M, Gong Y, Ahanonu B, Zeng H, Bito H & Schnitzer MJ (2019). Kilohertz two-photon brain imaging in awake mice. *Nat Methods* **16,** 1119–1122.

7. Acknowledgments

I would like to thank my supervisor, Prof. Tomomi Nemoto for his precise advice, support, and guidance, Dr. Ryosuke Enoki for valuable suggestions and feedbacks, Dr. Kohei Otomo at Graduate School of Medicine, Juntendo University, for technical advice, suggestions, and discussions. I also thank Prof. Shunichi Sato and Dr. Yuichi Kozawa at Institute of Multidisciplinary Research for Advanced Materials, Tohoku University, for valuable discussions and technical support with YDFA experiments. I also thank Dr. Hajime Hirase at Center for Translational Neuromedicine, University of Copenhagen, for giving us GLT-1-G-CaMP7 transgenic mice. I would gratefully thank to the member of my thesis committee Prof. Yumiko Yoshimura at Division of Visual Information Processing, National Institute for Physiological Sciences, and Prof. Katsumasa Fujita at Department of Applied Physics, Osaka University.

I would like to thank Dr. Hirokazu Ishii, Dr. Motosuke Tsutsumi, Dr. Taiga Takahashi, and Dr. Joe Sakamoto at Biophotonics Research Group, Exploratory Research Center on Life and Living Systems (ExCELLS), National Institutes of Natural Sciences, for valuable discussions and feedbacks. I thank all members of my laboratory, Dr. Chang Ching-Pu, Mrs. Chiemi Hyodoh, Mr. Kaito Nakata, Mrs. Kana Tsuchiya, Dr. Lee Ming-Liang, Mrs. Maki Watanabe, Mrs. Miwa Kawachi, Mr. Sota Hiro, Ms. Yuki Watakabe, and alumni.

8. Figures

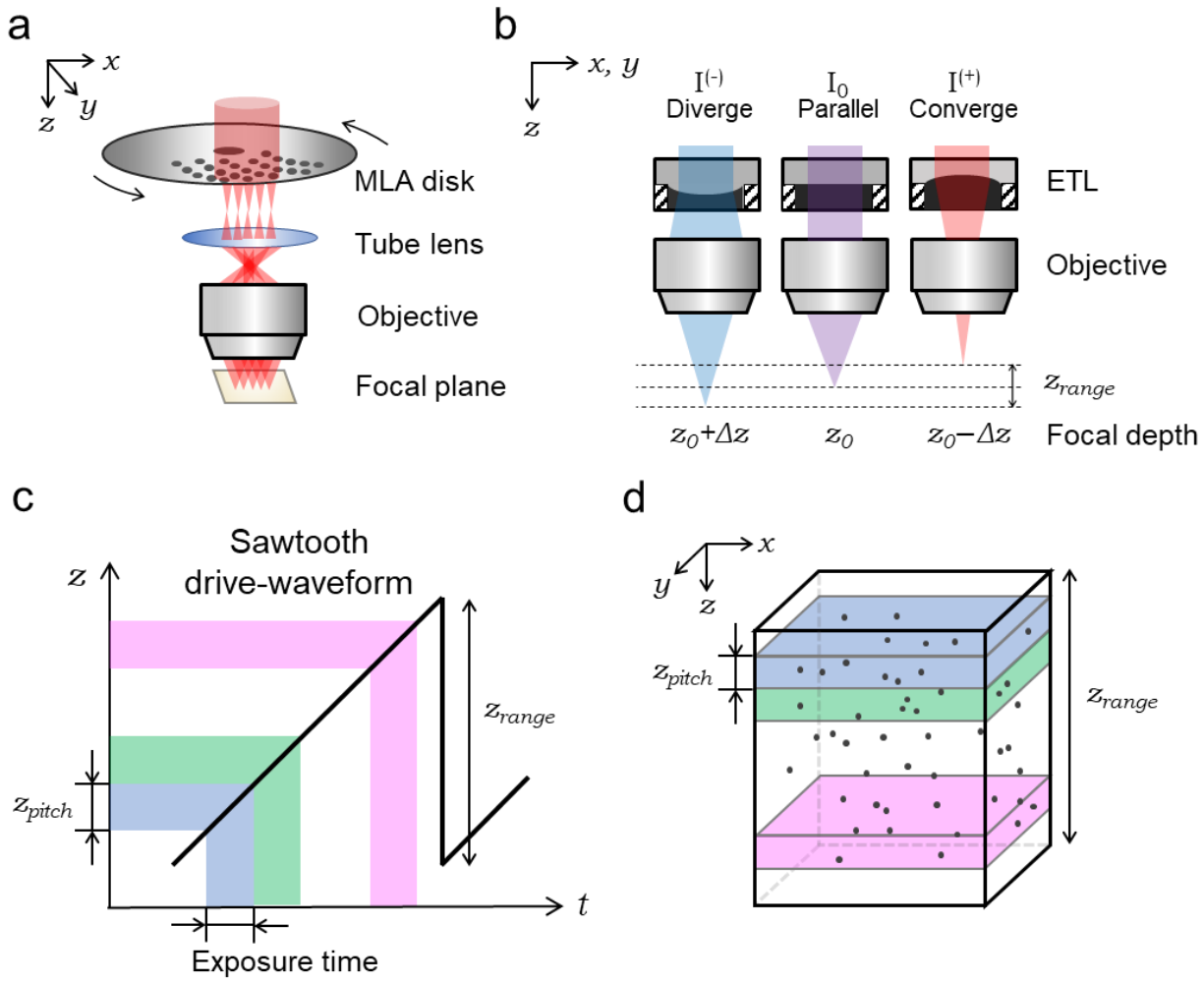


Figure 2.1. Schematic of lateral and axial scanning and image reconstruction.

(a) Lateral scanning by a spinning-disk scanner. An incident excitation beam is split by arrayed microlenses and focused simultaneously on the multiple foci. MLA: Microlens-array.

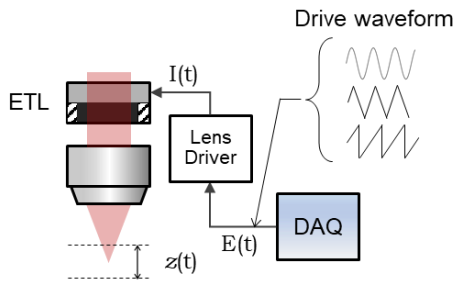
(b) Axial scanning based on a different focal length of incident beam controlled by an ETL

that was placed just before the objective lens. A parallel incident beam produced by the ETL current of I_0 focuses on the depth of z_0. A divergence beam produced by the ETL current of $I^{(-)}$ makes the focal spot far from the objective lens ($z_0 + \Delta z$). Inversely, a convergence beam produced by the ETL current of $I^{(+)}$ makes the focal spot closer to the objective lens ($z_0 - \Delta z$). z_{range} indicates the length of the depth of view.

(c) Exposure time and corresponding *z*-pitch of example frames (colored areas) in a sawtooth drive waveform. z_{pitch} indicates the axial range for each frame.

(d) 3D reconstruction of example frames (colored areas) shown in (c).

a



b

Waveform	Uniform motion	Uni-directional
Sine	False	False
Triangular	True	False
Sawtooth	True	True

c

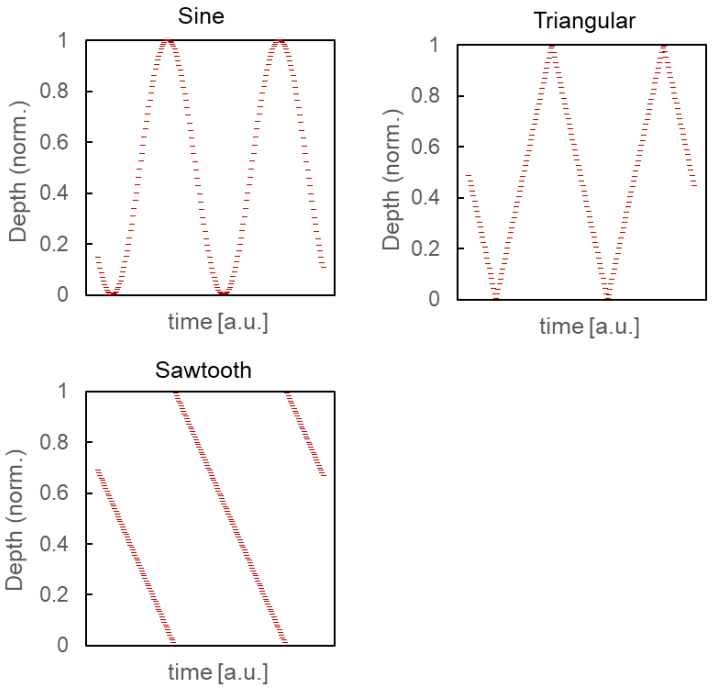


Figure 2.2. Optimal drive waveform for axial scanning.

(a) ETL drive waveforms (sine, triangular, sawtooth) that control the axial positon (z(t)). The DAQ board outputs voltage signals (E(t)) as a periodic drive waveform and are converted to the ETL current (I(t)) by the lens driver.

(b) Key features of a waveform for continuous axial scanning. Uniform motion of the focal plane results in the frames with uniform axial pitch. Unidirectional scanning ensures a constant update rate for each axial positions.

(c) Image acquisition positions (200 frames) with continuous axial scanning by each drive waveform.

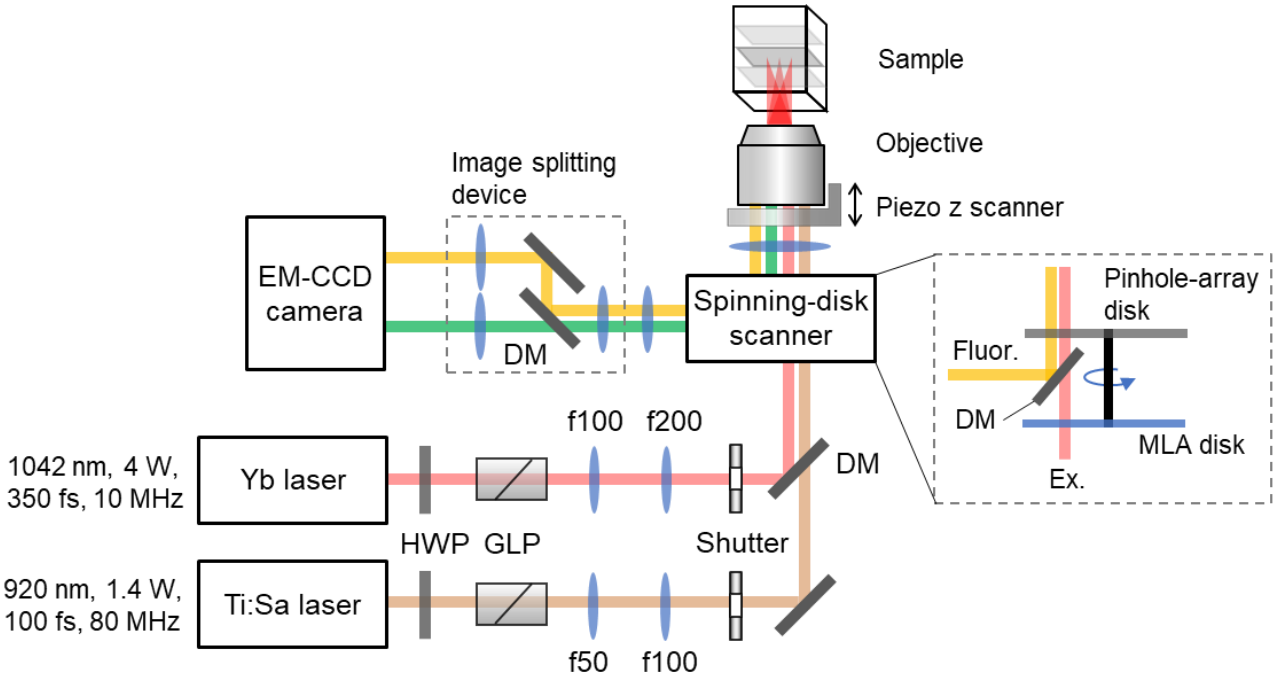


Figure 2.3. Optical layout of an inverted multibeam scanning two-photon microscope system with a confocal spinning-disk scanner.

HWP: Half-wave plate; GLP: Glan-laser polarizer; DM: Dichroic mirror; MLA: Microlens-array; Ex.: Excitation light; Fluor.: Fluorescence.

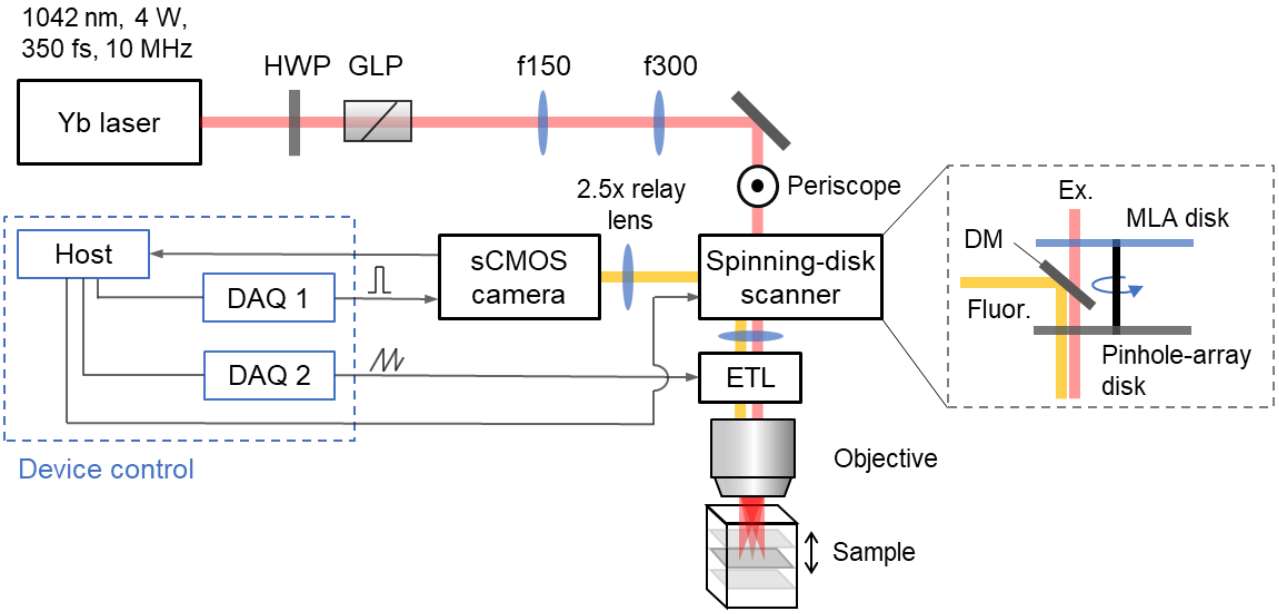


Figure 2.4. Optical and control layout for MCAS-2PM based on an upright microscope system with a confocal spinning-disk scanner.

HWP: Half-wave plate; GLP: Glan-laser polarizer; DM: Dichroic mirror; MLA: Microlens-array; Ex.: Excitation light; Fluor.: Fluorescence.

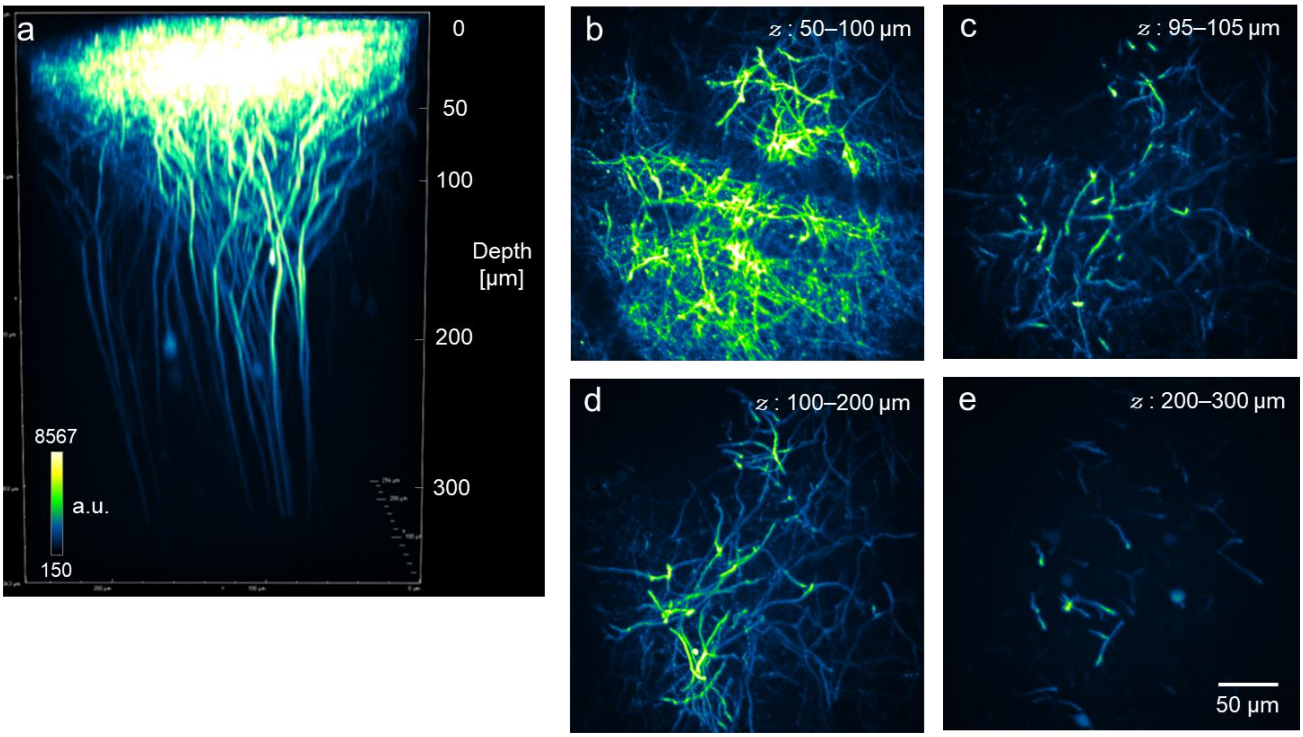


Figure 3.1. *In vivo* deep brain imaging via multibeam scanning 2PM in mouse primary visual cortex.

(a) 3D view of the *z*-stack image in the cortex of Thy1-EYFP-H mouse. Each frame of the z-stack image had 512 × 512 pixels and was captured with an exposure time of 500 ms.

(b–e) MIP images of dendritic fibers observed in the ranges of 50–100-, 95–105-, 100–200-, and 200–300-μm depths, respectively, corresponding to (a).

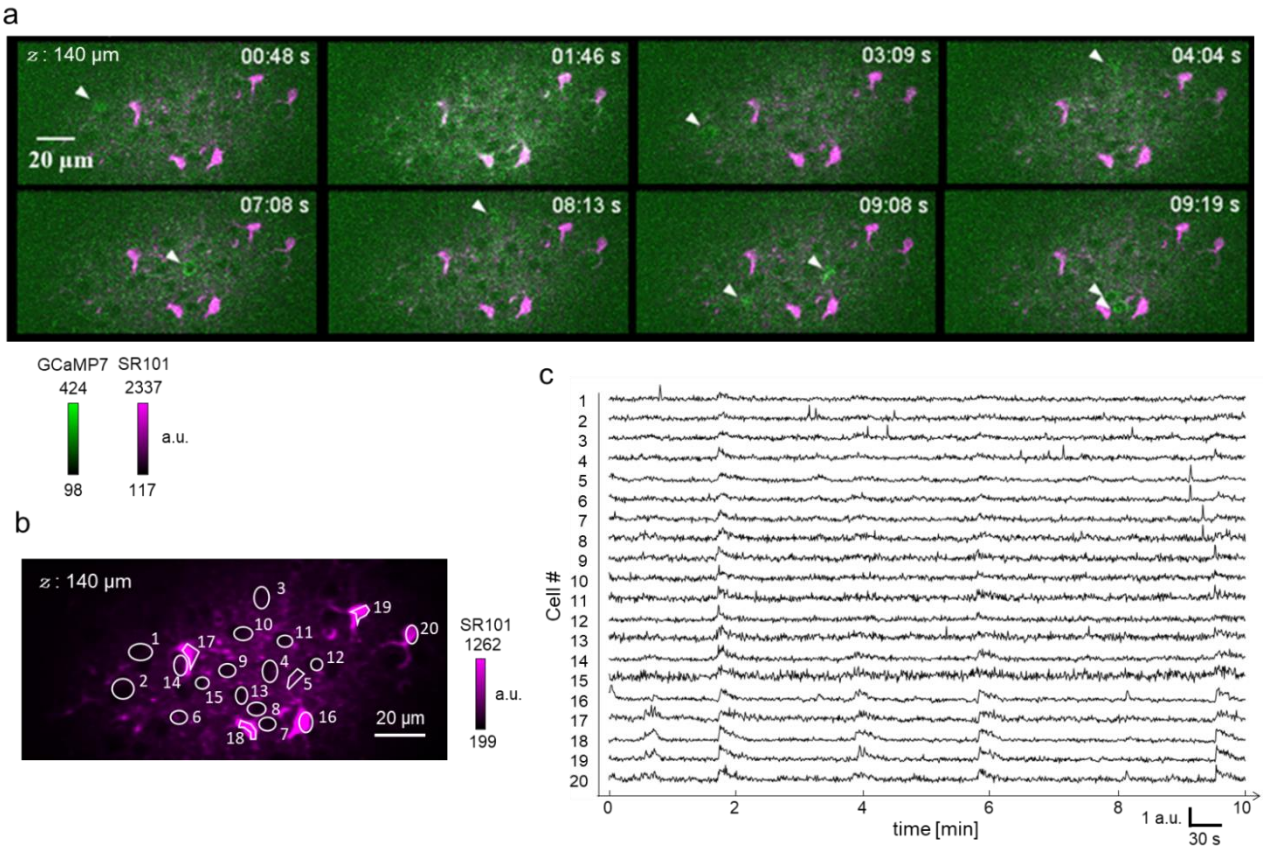


Figure 3.2. *In vivo* Ca^{2+} imaging of spontaneous activity in mouse primary visual cortex via multibeam scanning 2PM.

(a) Two-color *xy*-images at several time points at 140-μm depth in 262 × 131 μm^2 FOV at 1.8 fps. Arrowheads indicate neuronal activities.

(b) Time-averaged fluorescence image of SR101 channel and ROIs including neurons and astrocytes (white circles). The ROIs of #1–15 and #16–20 indicate neurons and astrocytes, respectively.

(c) Ca^{2+} activities extracted from the ROIs shown in (b).

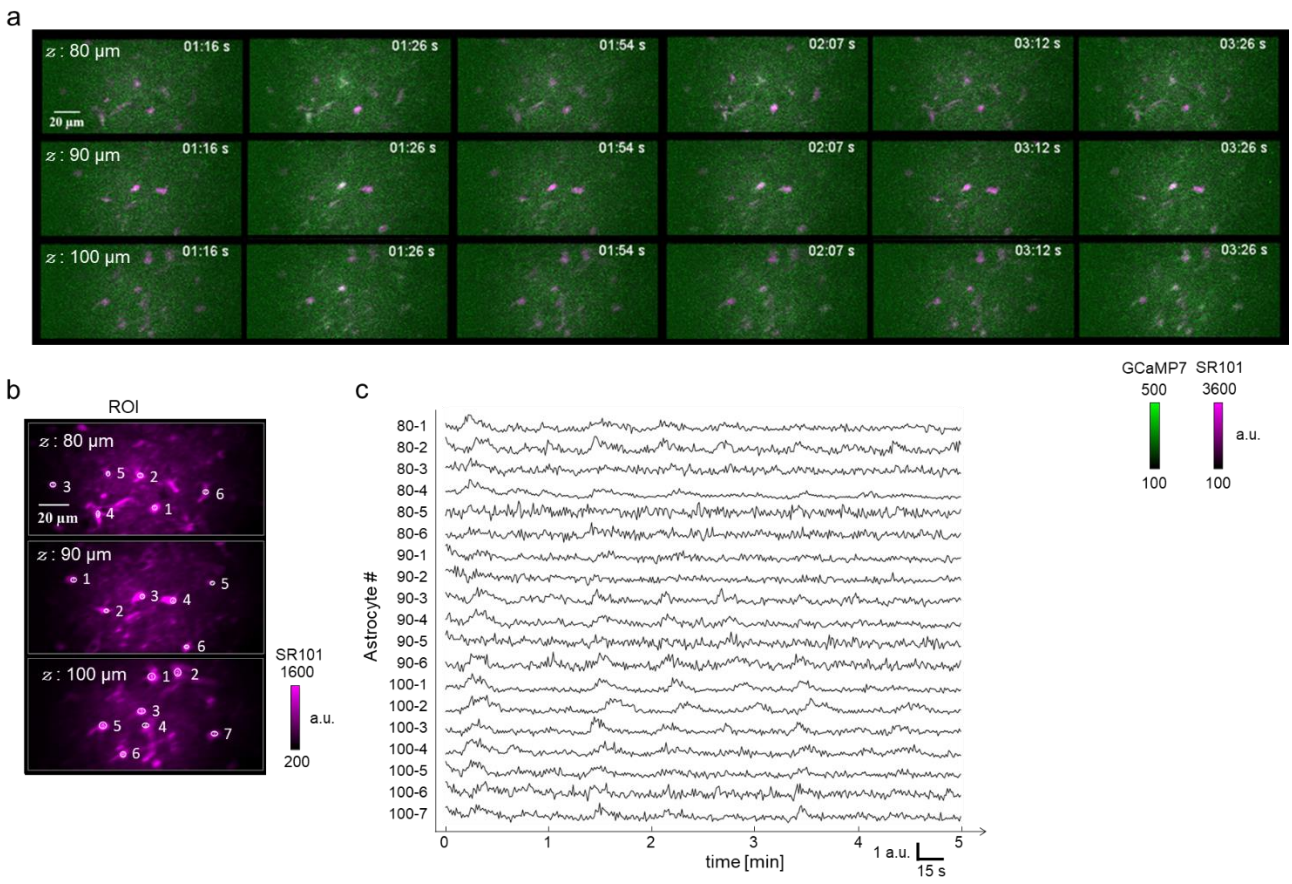


Figure 3.3. Multiplane *in vivo* Ca^{2+} imaging of astrocytic activity in mouse primary visual cortex via multibeam scanning 2PM.

(a) Two-color *xy*-images at different time points for three-planes from 80- to 100-μm depth in a 262 × 131 μm^2 FOV at 1.3 stack/sec.

(b) Time-averaged fluorescence images of SR101 channel and ROIs including astrocytes (white circles) for each focal plane.

(c) Ca^{2+} activities extracted from the ROIs shown in (b).

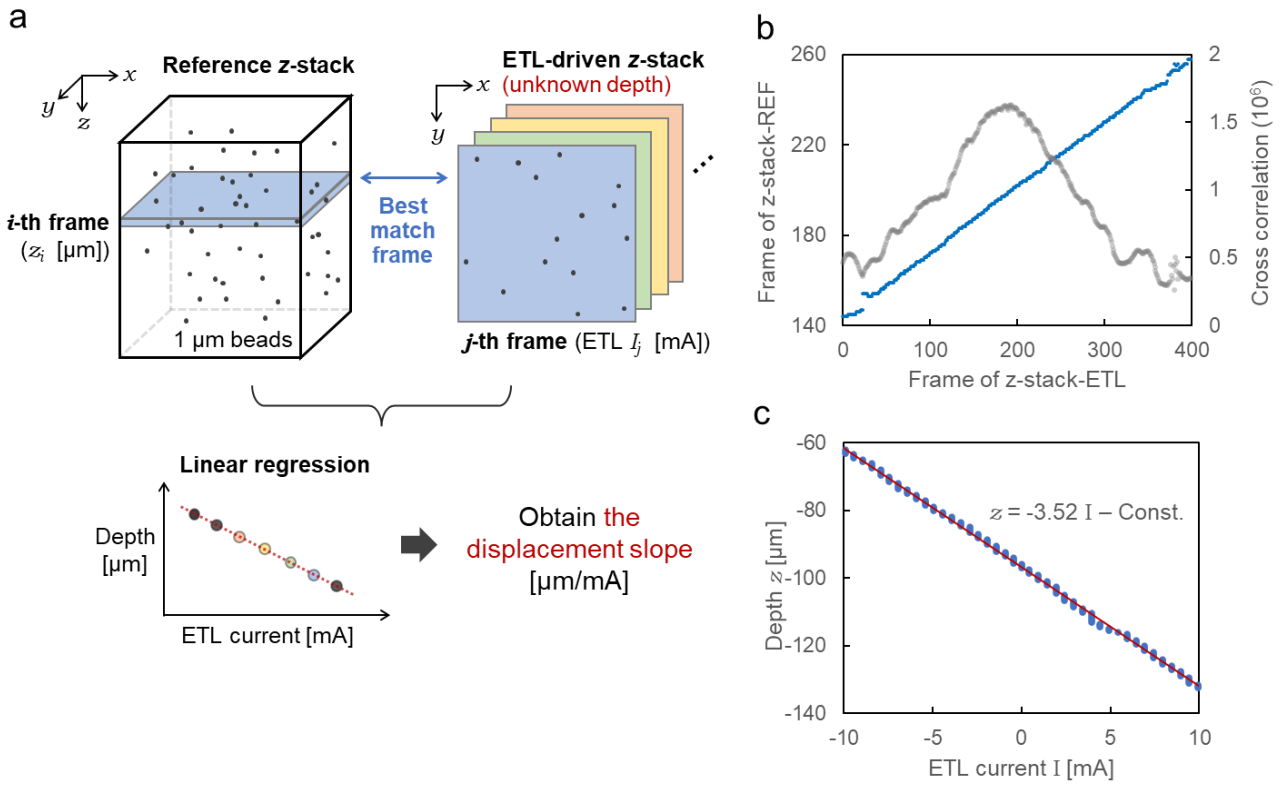


Figure 3.4. Relationship between axial displacement and ETL drive current.

(a) Schematic of the procedure for calculating the displacement slope. The best-match frame in a reference *z*-stack corresponding to each frame in ETL-driven *z*-stack was identified based on cross-correlation (top). The displacement slope was determined as a linear regression coefficient of ETL current with depth derived from the best-match frames (bottom).

(b) The most correlated frames in the two examples of *z*-stacks. The blue dots indicate the best match frame pairs. The gray dots indicate the cross-correlation of the best match frame pairs.

(c) Relationship between ETL current and depth (blue dots) derived from (b) and its linear regression (red line).

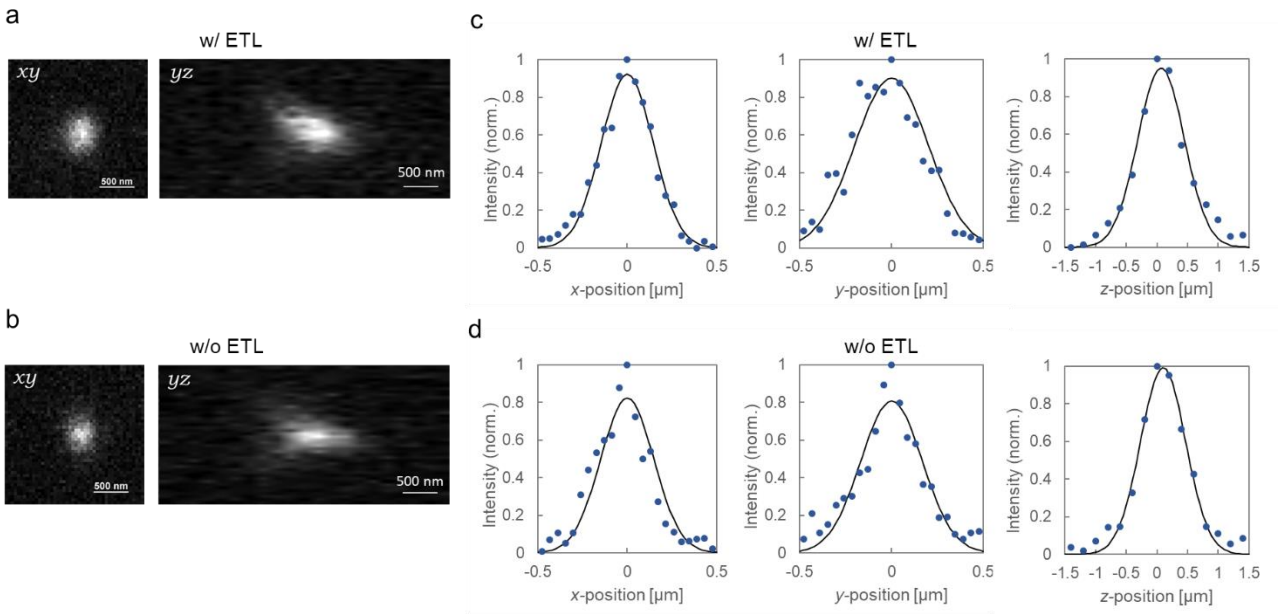


Figure 3.5. Spatial resolution in multibeam scanning 2PM.

(a, b) *xy*- and *yz*-image of a 0.2-μm yellow-green bead observed with an ETL and without ETL, respectively.

(c, d) Normalized fluorescence intensities for the *x*-, *y*-, and *z*-axis in (a) and (b), respectively. The intensity profiles (blue dots) were fitted to a Gaussian function (gray line).

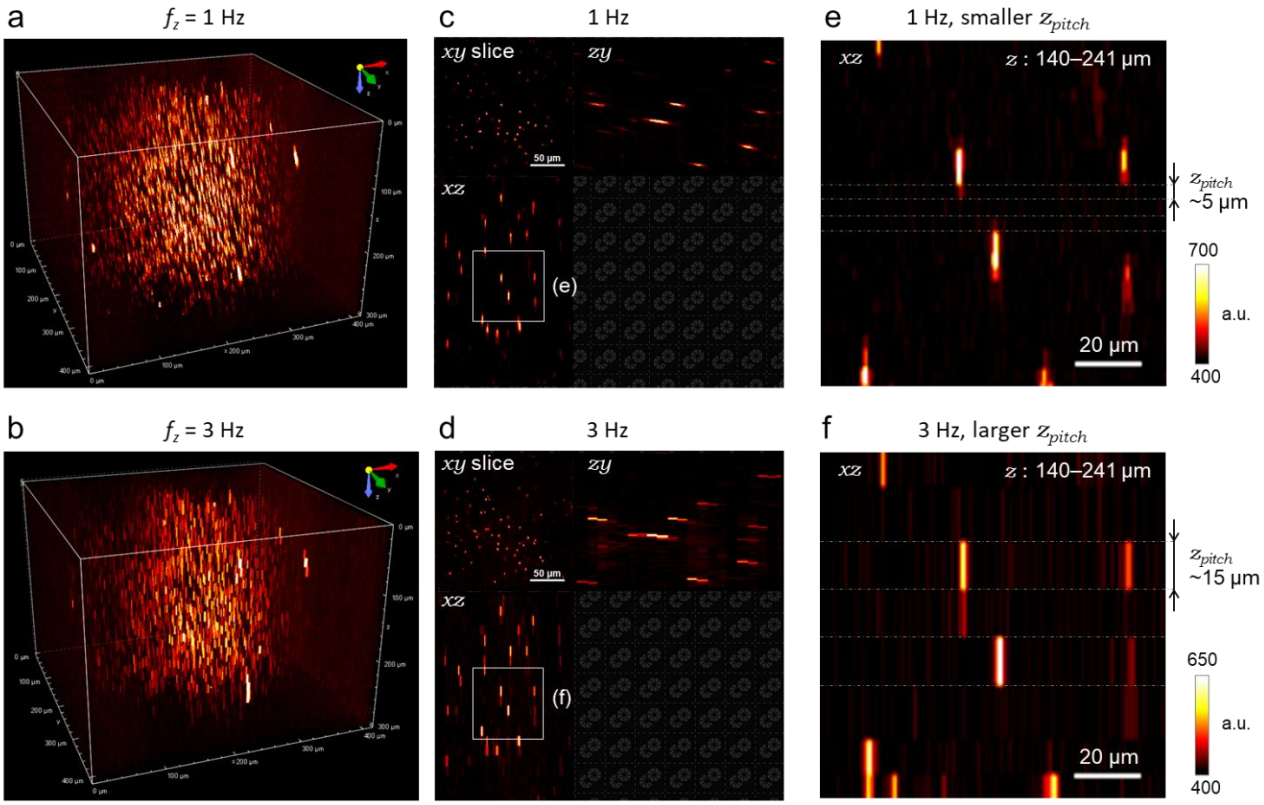


Figure 3.6. Volumetric imaging of 1-μm Nile red beads via MCAS-2PM.

(a, b) 3D view of the 1-μm bead phantom in 415 × 415 × 300 μm^3 FOV with the volume rates of 1 and 3 Hz, respectively.

(c, d) *z*-projection view of the volumes shown in (a) and (b), respectively.

(e, f) *xz*-images cropped from (c) and (d), respectively. The dotted gray lines indicate *z*-pitches, the range of quantized fluorescence intensities in the different volume rates.

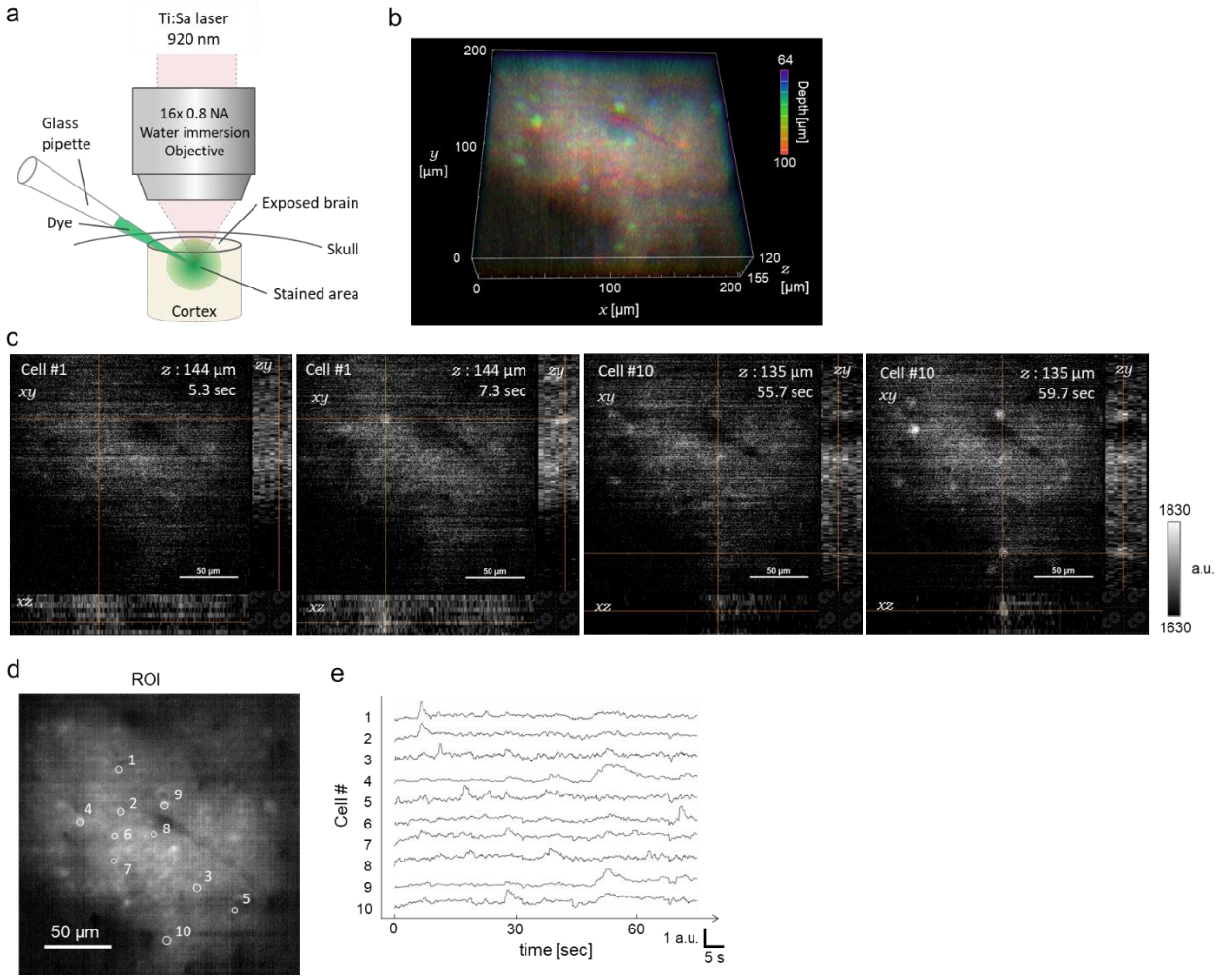


Figure 3.7. *In vivo* volumetric Ca^{2+} imaging of spontaneous activity with Cal-590 AM in mouse primary visual cortex via MCAS-2PM.

(a) Schematic of the dye loading method.

(b) 3D MIP image along the time-axis. The pseudo colors indicate the depth of the fluorescence signals.

(c) *z*-projection images of two cells shown in (d) and (e) at the different depths (144- and 135-μm) and *xy*-positions in a 200 × 200 × 36 $μm^2$ FOV with volume rate of 1.5 Hz.

(d) Time-averaged image of the maximum intensity *z*-projection. The white circles and numbers indicate ROIs.

(e) Spontaneous Ca^{2+} transients recorded from cell bodies corresponding to the ROIs shown in (d).

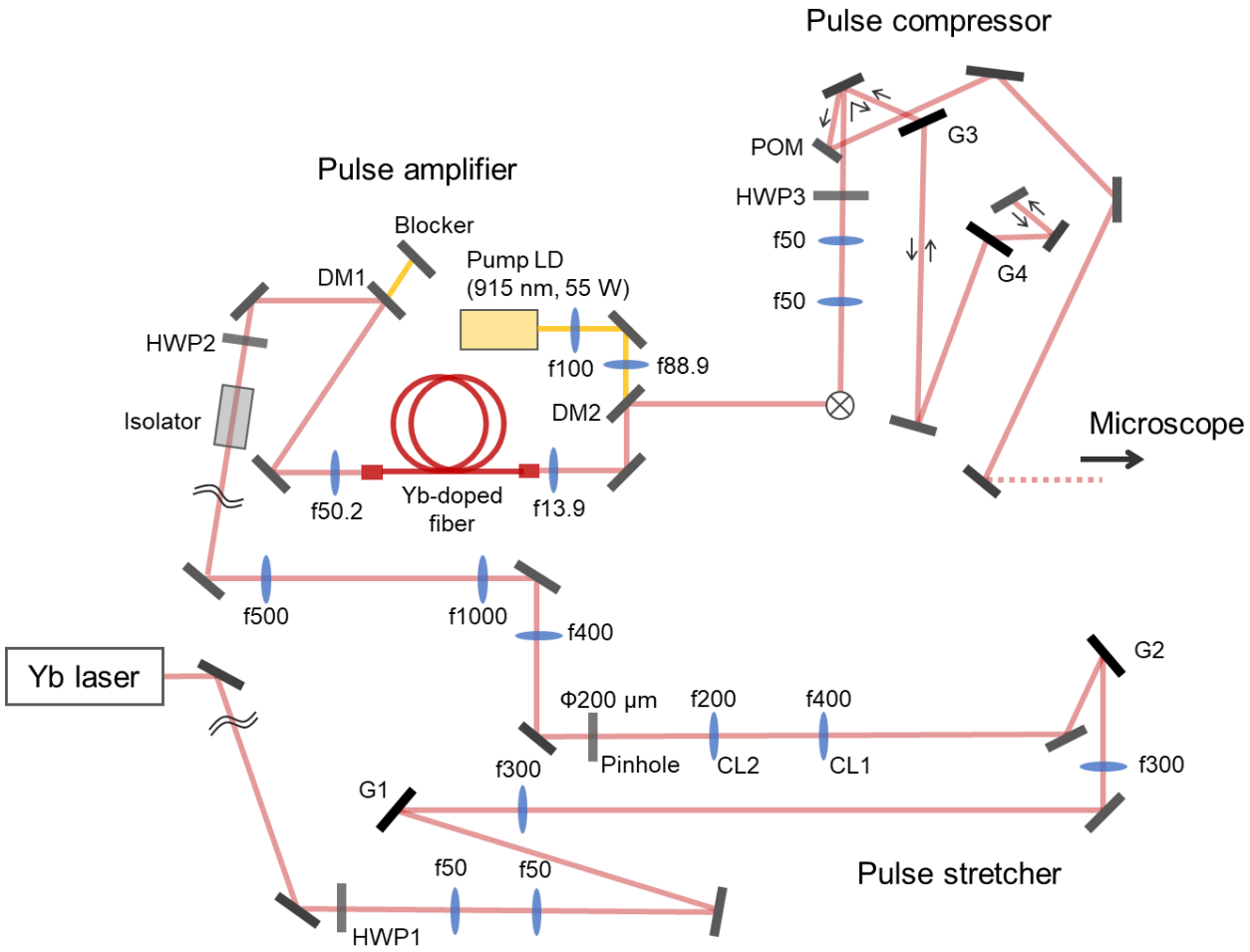


Figure 3.8. Optical layout of a CPA system.

HWP1–3: Half-wave plate; G1–4: Diffraction grating; CL1, 2: Cylindrical lens; DM: Dichroic mirror; POM: Pickoff mirror.

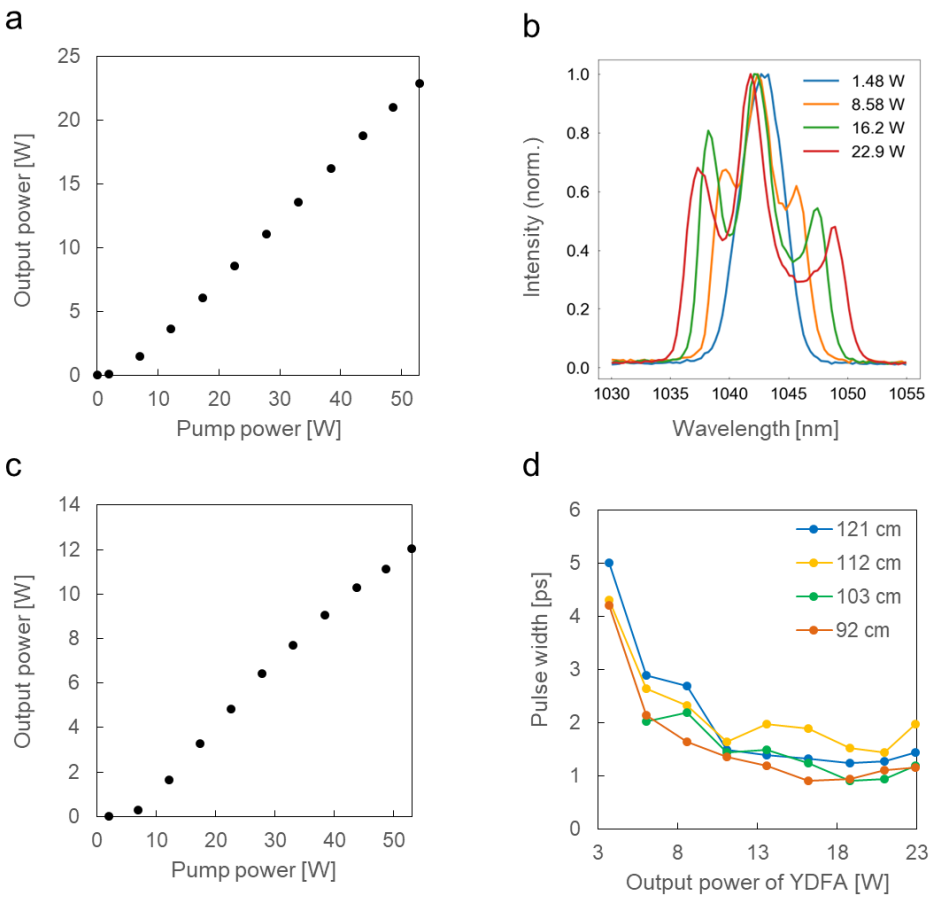


Figure 3.9. Average output power and pulse width of amplified beams.

(a) Average output power of the amplified beam as a function of the pump power.

(b) Emission spectra of the amplified beam at different average output power.

(c) Average output power of the amplified beam measured after the compressor.

(d) Optimization of the grating distance in the compressor. The pulse widths of the amplified beam were measured after the compressor at different grating distance.

Appendix

The programs used in this study were listed as follows: First, the source codes for calculating the relationship between the ETL current and the focal depth were shown (Code A1 and A2; related to Subsection 2.1.4). Next, the program for reconstructing a 4D (xyz-t) image from an xy-t image with driving voltages and corresponding timings recorded by the DAQ boards was shown (Code A3; related to Subsection 2.1.5). Finally, the C# programs that control the devices for realizing volumetric imaging were shown (Code A4–A26; related to Section 2.1). This appendix does not include the source codes that were automatically generated by Windows Form Designer (Microsoft) for creating the graphical user interface and provided by Optotune Switzerland AG and Hamamatsu Photonics K.K. for the configuration of the ETL and camera parameters, respectively.

Index of Appendix

Code A1. Calculation of best-matched frames (Python3)

```
import tifffile
import numpy as np
import cv2
import scipy as sp
from joblib import Parallel, delayed
import csv
import matplotlib.pyplot as mpl
import pandas as pd
import os
from utils import extract_number


def median_3dstack_cv2(stack, ksize):
    # Note that the allowed ksize is only 5 or 3
    res = np.zeros_like(stack)
    for t in range(0, stack.shape[0]):
        cv2.medianBlur(stack[t], ksize, dst=res[t])
    return res


def median_3dstack(stack, ksize):
    # Note that the allowed ksize is only 5 or 3
    res = np.zeros_like(stack)
    for t in range(0, stack.shape[0]):
        res[t] = sp.ndimage.median_filter(stack[t], ksize)
    return res


def get_roi_boundary(stack_ndarr, filepath="filepath", show_result=True, debug=False):
    mip = np.max(stack_ndarr, axis=0)
    mean = np.mean(mip)
    sd = np.std(mip)
    threshold = mean + sd * 2

    mean_px_number = 32
    mean_kernel = np.ones(shape=(mean_px_number, mean_px_number), dtype=np.float32) /
mean_px_number**2
    mip_mean = cv2.filter2D(mip, -1, mean_kernel)
    center_xy = np.unravel_index(np.argmax(mip_mean), mip.shape)

    # initial number and step are changeable
    for r in range(100, min(center_xy[0], center_xy[1]), 5):
        x0 = max(0, center_xy[1]-r)
        x1 = min(mip.shape[1], center_xy[1]+r+1)
        y0 = max(0, center_xy[0]-r)
        y1 = min(mip.shape[0], center_xy[0]+r+1)
        sum = 0
        cnt = 0
        for y, x in zip(range(x0, x1), range(y0, y1)):
            sum += mip[y, x]
            cnt += 1
        result = sum / cnt
        if debug:
            print(f"@Radius={r}, mean value={result:.1f}")
        if result < threshold:
            lt = (x0, y0)
            rb = (x1, y1)
            print(f"ROI: LT={lt}, RB={rb}")
            # showing selected ROI as follows
            peak = np.max(mip)
            for xx in [xxx for xxx in range(x0-2, x1+3) if xxx < x0 or xxx > x1]:
                for yy in range(y0, y1):
                    mip[yy, xx] = peak
            for yy in [yyy for yyy in range(y0-2, y1+3) if yyy < y0 or yyy > y1]:
                for xx in range(x0, x1):
```

```
                    mip[yy, xx] = peak
            break

    mpl.imshow(mip)
    fname = os.path.basename(filepath)
    mpl.title(f"MIP image with ROI (Mean+2σ boundary) of {fname}.¥nLeftTop=({x0}, {y0}),
RightBottom=({x1}, {y1})")
    savepath = filepath + "_roi.png"
    mpl.savefig(savepath)

    if show_result:
        mpl.draw()
        mpl.pause(0.01)

    return lt, rb


def compare_img_along_stack_SAD(stack_a, target_img_b):
    err_min = -1
    for tl in range(0, stack_a.shape[0]):
        subtracted_img = np.abs(target_img_b - stack_a[tl])
        err = np.sum(subtracted_img)
        # seek best-matched frame
        if err < err_min or err_min < 0.0:
            err_min = err
            best_matched_frame = tl

    return best_matched_frame, err_min


def compare_img_along_stack_correlate(stack_a, target_img_b):
    cor_max = -1
    for tl in range(0, stack_a.shape[0]):
        cor = cv2.matchTemplate(stack_a[tl], target_img_b, cv2.TM_CCOEFF_NORMED)
        # seek best-matched frame
        if cor > cor_max or cor_max < 0.0:
            cor_max = cor
            best_matched_frame = tl

    return best_matched_frame, cor_max


def calc_bestmatch_frame(file_base="", file_sample=""):

    if file_base == "" and file_sample == "":
        # base img (ETL)
        file_a = r"C:¥Users¥MCB¥Desktop¥temp¥0655_.tif"
        # sample img (FocusDriveMotor)
        file_b = r"C:¥Users¥MCB¥Desktop¥temp¥0654_.tif"
    else:
        file_a = file_base
        file_b = file_sample

    file_a_basename = os.path.basename(file_a)
    file_b_basename = os.path.basename(file_b)

    #trim Left-Top (LT) and Right-Bottom (RB)
    # (x,y)
    # For 063 and 064
    #lt = (720, 720)
    #rb = (1550, 1550)

    imgs = tifffile.imread(file_a)
    ndarr_a = np.array(imgs)
    # Calculate ROI boundary using MIP. Returns (x, y), NOT (y, x)
    #lt, rb = get_roi_boundary(ndarr_a, file_a)
    # if not want to use ROI
    lt = (0, 0)
    rb = (768, 768)
```

```
    ndarr_a = np.array(imgs[:, lt[1]:rb[1], lt[0]:rb[0]])

    imgs = tifffile.imread(file_b)
    ndarr_b = np.array(imgs[:, lt[1]:rb[1], lt[0]:rb[0]])

    del imgs

    print(f"ndarr_a: {ndarr_a.shape}")
    print(f"ndarr_b: {ndarr_b.shape}")

    # Median Filter (Anti-motion artifacts; StackReg, moco, etc)
    ksize = 3
    stack_a = np.array(median_3dstack_cv2(ndarr_a, ksize), dtype=float)
    stack_b = np.array(median_3dstack_cv2(ndarr_b, ksize), dtype=float)
    # *** Important to change type uint to float. ***

    del ndarr_a, ndarr_b

    # Normalization (z score); Standardize per xy-image
    for t in range(0, stack_a.shape[0]):
        stack_a[t] = sp.stats.zscore(stack_a[t], axis=None)
    for t in range(0, stack_b.shape[0]):
        stack_b[t] = sp.stats.zscore(stack_b[t], axis=None)

    # Calculate best match frame pair
    execute_as_single = True
    method = "cor"
    if execute_as_single:
        best_matched = np.zeros(stack_b.shape[0], dtype=int)
        best_values = np.zeros(stack_b.shape[0], dtype=float)

        if method == "sad":
            for ts in range(0, stack_b.shape[0]):
                err_min = -1
                for tl in range(0, stack_a.shape[0]):
                    subtracted_img = np.abs(stack_b[ts] - stack_a[tl])
                    err = np.sum(subtracted_img)
                    # seek best-matched frame
                    if err < err_min or err_min < 0.0:
                        err_min = err
                        best_matched[ts] = tl
                best_values[ts] = err_min
                print(f"best[{ts}] = {best_matched[ts]}¥t@err={err_min:.2f}")

        elif method == "cor":
            for ts in range(0, stack_b.shape[0]):
                cor_max = -1
                target_img_b = stack_b[ts]
                for tl in range(0, stack_a.shape[0]):
                    result = np.correlate(np.ravel(stack_a[tl]), np.ravel(target_img_b))
                    # seek best-matched frame
                    cor = np.max(result)
                    if cor > cor_max or cor_max < 0.0:
                        cor_max = cor
                        best_matched[ts] = tl
                best_values[ts] = cor_max
                print(f"best[{ts}] = {best_matched[ts]}¥t@corr={cor_max:.2f}")

        elif method == "cor_self":
            for ts in range(0, stack_b.shape[0]):
                cor_max = -1
                target_img_b = stack_b[ts]
                for tl in range(0, stack_a.shape[0]):
                    num = np.sum(stack_a[tl] * target_img_b)
                    den = np.sqrt( np.sum(stack_a[tl] ** 2) * np.sum(target_img_b ** 2) )
                    if den == 0:
                        cor = 0
                    else:
```

```
                    cor = num / den
                # seek best-matched frame
                if cor > cor_max:
                    cor_max = cor
                    best_matched[ts] = tl
            best_values[ts] = cor_max
            print(f"best[{ts}] = {best_matched[ts]}¥t@corr={cor_max:.2f}")

        # output
        file = file_b+".csv"
        num_base_str = extract_number(file_a_basename, 3)
        num_sample_str = extract_number(file_b_basename, 3)
        new_basename = file_b_basename.replace(num_sample_str, f"{num_base_str}base-
{num_sample_str}sample")
        file = file.replace(file_b_basename, new_basename)

        writer = open(file, "w", newline="")
        if "cor" in method:
            header = ["Frame", "Best-matched", "Corr"]
        else:
            header = ["Frame", "Best-matched", "Error"]
        print(header)
        w = csv.writer(writer)
        w.writerow(header)
        for i in range(0, len(best_matched)):
            # print(f"{i}¥t{best_matched[i]}¥t¥t{best_values[i]:.2f}")
            w.writerow([i, best_matched[i], best_values[i]])
        writer.close()

    else:
        out = Parallel(n_jobs=-1, verbose=8)([delayed(compare_img_along_stack_SAD)(stack_a,
stack_b[ts]) for ts in range(0, stack_b.shape[0])])

        # output
        file = file_b+".csv"
        writer = open(file, "w", newline="")
        header = ["Frame", "Best-matched", "Error"]
        print(header)
        w = csv.writer(writer)
        w.writerow(header)
        for i in range(0, len(out)):
            matched = out[i][0]
            err = out[i][1]
            print(f"{i}¥t{matched}¥t¥t{err:.0f}")
            w.writerow([i, matched, err])
        writer.close()

    # Visualize
    fig = mpl.figure()
    vis_bestmatched_corr(file, fig)
    mpl.show()


def vis_bestmatched_corr(file_calculated, mpl_fig):
    df = pd.read_csv(file_calculated, index_col=0)
    x = df.index.values
    y = df["Best-matched"]
    # mpl_fig = mpl.figure()
    ax1 = mpl_fig.add_subplot(111)
    ax1.scatter(x=x, y=y, c="red", label="Best-matched frame")
    ax1.plot(x, y, linestyle='solid', c="red", label="Best-matched frame")
    ax1.set_ylabel("Best-matched frame")
    ax1.set_xlabel("Frame")

    # correlation
    #ax2 = ax1.twinx()
    #ax2.scatter(x=x, y=df["Corr"], c="gray", label="Correlation")
    #ax2.set_ylabel("Correlation")
```

```
def vis_ao_timings(file_rec_sample, mpl_fig):
    df = pd.read_csv(file_rec_sample, index_col=0)
    x = df['FrameNumber']
    y = df['Voltage']  # voltage of AnalogOutput


if __name__ == '__main__':
    import sys, os.path
    args = sys.argv
    print(f"args = {args}")

    # Enter the target file on the command line
    if len(args) == 3:
        if os.path.exists(args[1]) and os.path.exists(args[2]):
            print("Starting ETL Calibration...")
            print(f"Longer stack file = {args[1]}")
            print(f"Shorter stack file = {args[2]}")
            calc_bestmatch_frame(args[1], args[2])
    else:
        calc_bestmatch_frame()
```

Code A2. Calculation of the relationship of ETL current and depth (Python3)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as mpl
import math
import scipy.stats

# if ETL has used as BASE, turn it True
x_has_reference_depth = True


def rsquared(x, y):
    """ Return R^2 where x and y are array-like."""
    slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(x, y)
    return r_value**2


def linear_fit(csvfile="") -> pd.DataFrame:
    ## CONFIG BEGIN ##
    # Enter the calculated best-match frame relationship
    csvfile = r"C:¥Users¥MCB¥Desktop¥temp¥0523base-0520sample_.tif.csv"
     # Set valid range for fitting
    begin, end = 100, 161
    ## END CONFIG ##

    if csvfile=="":
        return
    df = pd.read_csv(csvfile, index_col=0)
    print(df)

    # Visualize
    x = df.index.values
    y = df["Best-matched"]
    fig = mpl.figure()
    ax1 = fig.add_subplot(111)
    ax1.scatter(x=x, y=y, c="C0", label="Best-matched frame")
    ax1.set_ylabel("Best-matched frame")
    ax1.set_xlabel("Frame")

    ax2 = ax1.twinx()
    ax2.scatter(x=x, y=df["Corr"], c="gray", label="Correlation")
    ax2.set_ylabel("Correlation")

    # calculate
    # y = ax + b
    func = np.poly1d(np.polyfit(x[begin:end], y[begin:end], deg=1))
    xf = [i for i in range(begin, end)]
    a, b = np.polyfit(x[begin:end], y[begin:end], deg=1)
    r2 = rsquared(x[begin:end], y[begin:end])
    fitted_eq = f"y={a:.2f}x+({b:.2f})"
    ax1.plot(xf, func(xf), "red", label=f"fitted ({fitted_eq})¥nr^2={r2:.2f}")

    fig.legend(loc=1, bbox_to_anchor=(1,1), bbox_transform=ax1.transAxes)
    #fig.savefig(csvfile+".calib.png")
    #mpl.show()

    # create fitting table
    df_fit = pd.DataFrame({
        "x": xf,
        "y": func(xf)
        })
    print(df_fit)
```

```
    return df_fit


def current_depth_curve(df_fit: pd.DataFrame, x_has_reference_depth: bool):
    ## CONFIG BEGIN ##
    # ETL's all.csv
    current_frame_csvfile = r"C:¥Users¥MCB¥Desktop¥temp¥0651_all.csv"
    # FDM's depth of frame
    depth_frame_csvfile = r"C:¥Users¥MCB¥Desktop¥temp¥0650_depth_of_frame.csv"

    ## Note: best-matched table should contain valid (fitted) frames only.
    bestmatched_csvfile = r""
    ## END CONFIG ##

    if bestmatched_csvfile != "":
         df_fit = pd.read_csv(bestmatched_csvfile)

    key_frame = "FrameNumber"
    key_ticks = "Ticks"
    key_depth = "Depth"
    key_current = "Current"

    df_current = pd.read_csv(current_frame_csvfile, index_col=0)
    df_depth = pd.read_csv(depth_frame_csvfile, index_col=0)

    # Both data are retrieved and made available
    # ETL z-stack, FrameNumber => Current
    x = []
    last_cur = math.nan
    for cur, fnum in zip(df_current[key_current], df_current[key_frame]):
        if not math.isnan(cur):
            last_cur = cur
        if not math.isnan(fnum):
            x.append([int(fnum), last_cur])
    df_current = pd.DataFrame(x, columns=[key_frame, key_current])
    df_current = df_current.set_index(key_frame)
    print(df_current)
    print(df_depth)

    # FocusDriveMotor (Reference), FrameNumber => Depth
    # Not necessary to shape DataFrame of FDM, nothing to do here

    current_depth_curve = []
    for tup in df_fit.itertuples():

        if not x_has_reference_depth:  # means ETL image => x axis
            current = df_current.at[tup.x, key_current]
            try:
                f_ceil = df_depth.at[math.ceil(tup.y), key_depth]
                f_floor = df_depth.at[math.floor(tup.y), key_depth]
                slope = f_ceil - f_floor
                f_delta = slope * (tup.y - math.floor(tup.y))
                depth = f_floor + f_delta
                # depth = df_depth.at[int(tup.y), key_depth]  # test
                current_depth = [current, depth]
                current_depth_curve.append(current_depth)
                #print(f"tup.y={tup.y}, a1={a1}, a2={a2}")
                #print(f"current={current}, depth={depth}")
                #print()
            except Exception as e:
                print(e)
                pass

        else:
            depth = df_depth.at[tup.x, key_depth]
            try:
                f_ceil = df_current.at[math.ceil(tup.y), key_current]
                f_floor = df_current.at[math.floor(tup.y), key_current]
                slope = f_ceil - f_floor
```

```
                f_delta = slope * (tup.y - math.floor(tup.y))
                current = f_floor + f_delta
                #current = df_current.at[int(tup.y), key_current]  # test
                current_depth = [current, depth]
                current_depth_curve.append(current_depth)
                #print(f"tup.y={tup.y}, a1={a1}, a2={a2}")
                #print(f"current={current}, depth={depth}")
                #print()
            except Exception as e:
                print(e)
                pass

    df_curve = pd.DataFrame(current_depth_curve, columns=[key_current, key_depth])
    df_curve = df_curve.set_index(key_current)
    print(df_curve)
    df_curve.to_csv(current_frame_csvfile+"_curveTable.csv")

    x = df_curve.index.values  # current
    y = df_curve[key_depth]  # depth

    # return calibration curve as function. Input it to the main GUI software.

    # linear fitting
    # Always pass through the origin (Current=0, Depth=0)
    cross_xy00 = False
    if cross_xy00:
        x = np.append(x, 0)
        y = np.append(y, 0)
        weight = np.append(np.ones(len(x)-1), 1e9)

        # y = ax + b
        func = np.poly1d(np.polyfit(x, y, deg=1))
        a, b = np.polyfit(x, y, deg=1, w=weight)
        r2 = -1  # not implemented

    else:
        # y = ax + b
        func = np.poly1d(np.polyfit(x, y, deg=1))
        a, b = np.polyfit(x, y, deg=1)
        r2 = rsquared(x, y)

    fitted_eq = f"y={a:.2f}x+({b:.2f})¥nr^2={r2:.2f}"

    # Visualize
    fig = mpl.figure()
    ax1 = fig.add_subplot(111)
    if cross_xy00:
        ax1.set_title("ETL Current-Depth Curve (x,y)=(0,0)")
    else:
        ax1.set_title("ETL Current-Depth Curve")
    ax1.set_xlabel(key_current)
    ax1.set_ylabel(key_depth)

    ax1.scatter(x=x, y=y, c="C0")
    ax1.plot(x, func(x), "red", label=f"fitted ({fitted_eq})")
    fig.legend(loc=1, bbox_to_anchor=(1,1), bbox_transform=ax1.transAxes)
    print(f"depth = {a:.2f} * current + ({b:.2f})")
    mpl.show()


if __name__ == '__main__':
    df_fit = linear_fit()
    current_depth_curve(df_fit, x_has_reference_depth)
```

Code A3. 4D image reconstruction (Python3)

```
import dcimg  # does not work for latest .dcimg file (2021)
import numpy as np
from numpy.core.fromnumeric import shape
from numpy.core.numeric import NaN
import pandas as pd
import os
import math
import cv2
import time
import tifffile
import matplotlib.pyplot as mpl
import bioformats.omexml as ome
import sys

from tifffile.tifffile import TiffFile


### CONFIG BEGIN ###
file_image = r"¥¥server¥Experiments¥0503_.tif"
file_all_csv = r"¥¥server¥Experiments¥0503_all.csv"
etl_waveform = "sawtooth"
img_max_frames = 1000
output_interval_ms = 150

dir_save = r"X:¥work¥recon_0503_dz-1.0_dx-0.722_dt-150__scale-1.00x_1.5hz-sawtooth_test"

z_pitch = 1.0  # [micro meter]; z of single voxel
begin_frame_number_from_zero = True

# Decrease pixel number (< 1.0) or not (=1.0)
# (Referred to in the thesis as binning.)
scaling_x = 1.
scaling_y = scaling_x

# optical setup
obj_mag = 14.4
other_mag = 2.5
cam_pixel_size = 6.5  # [um]
binning = 4

# pixel size in the microscope system
pixel_size_x = (cam_pixel_size * binning) / (obj_mag * other_mag)  # [um]; optical parameter
pixel_size_y = pixel_size_x

# Apply scaled pixel size
pixel_size_x /= scaling_x
pixel_size_y /= scaling_y

# tifffile output preference
use_imagej = False
use_ome = True  # Default: True
use_bigtiff = True
### END CONFIG ###

def load_image(file: str):
    if file.endswith("dcimg"):
        print(f"Loading {file}...")
        dcimgs = dcimg.DCIMGFile(file)
        print(dcimgs.shape)
        img = dcimgs[:, :, :]

    elif file.endswith("tif"):
        print(f"Loading {file}...")
        img = tifffile.imread(file)
```

```
    return img

def depth_to_index(depth_range: tuple, pitch: float, actual_depth: float) -> int:
    # create map
    zz = []
    z_length = abs(depth_range[1] - depth_range[0])
    r = int(z_length / pitch) + 1
    for i in range(r):
        zz.append(depth_range[0] + i*pitch)

    # choose index by using least error
    min_sub = abs(zz[0]-actual_depth)
    min_idx = 0
    for i in range(len(zz)):
        sub = abs(zz[i]-actual_depth)
        if sub < min_sub:
            min_sub = sub
            min_idx = i

    return min_idx

# Return the valid type of value. E.g. Voltage, Current, Depth
def get_valid_inst_column(df:pd.DataFrame):
    candidates = ["Depth", "Current", "Voltage"]
    nan_columns = df.isna().all()  # All raws are NaN => True
    print(nan_columns)

    valid = []
    for col in df.columns:
        if not nan_columns[col] and col in candidates:
            print(f"{col} is ready")
            valid.append(col)
    return valid

# finally returns FrameNumber--Depth data
def analyze_record(data:pd.DataFrame, max_frames=500):
    # create new DataFrame:
    columns = ["FrameNumber", "TickBegin", "TickEnd", "DepthMin", "DepthMax", "VoltageMin",
"VoltageMax"]
    df = pd.DataFrame(columns=columns)

    # order of columns in .csv file
    col_ticks = 0
    col_voltage = 1
    col_camera_state = 3
    col_frame_number = 4
    col_depth = 5
    col_trigger_voltage = 6

    # at beginning
    i = 0
    while i < len(data):
        # FrameNumber == 4
        if data.iat[i, col_camera_state] == "FrameBegin":
            frame = (int)(data.iat[i, col_frame_number])
            if frame >= max_frames:
                break

            # tick begin
            tick_begin = data.iat[i, col_ticks]
            i += 1  # skip current row for CameraState
            while not math.isnan(data.iat[i, col_trigger_voltage]):
                i += 1

            if i >= len(data.index):
                break

            depth_begin = data.iat[i, col_depth]
            depth_min = depth_begin
```

```
            depth_max = depth_begin

            voltage_begin = data.iat[i, col_voltage]
            voltage_min = voltage_begin
            voltage_max = voltage_begin

            while data.iat[i, col_camera_state] != "FrameBegin":
                d = data.iat[i, col_depth]
                v = data.iat[i, col_voltage]
                # depth
                if d < depth_min:
                    depth_min = d
                elif d > depth_max:
                    depth_max = d

                # voltage
                if v < voltage_min:
                    voltage_min = v
                elif v > voltage_max:
                    voltage_max = v
                i += 1

                if i >= len(data.index):
                    break

            # Here, "i" means the CameraState's FrameBegin row, so that "i-1" indicates the row
for last tick and depth
            tick_end = data.iat[i-1, col_ticks]  # actually, if using EdgeTrigger, this is NOT
the end tick
            depth_end = data.iat[i-1, col_depth]
            voltage_end = data.iat[i-1, col_voltage]

            # debug
            # print(f"Ticks of Frame {frame}: begin at {tick_begin}, end at {tick_end}")
            # print(f"Depth of Frame {frame}: begin at {depth_begin:.1f}, end at
{depth_end:.1f}")
            # print(f"Depth Min/Max: {depth_min:.1f} / {depth_max:.1f}")
            # print(f"Voltage Min/Max: {voltage_min:.3f} / {voltage_max:.3f}")

            # columns = ["FrameNumber", "TickBegin", "TickEnd", "DepthMin", "DepthMax",
"VoltageMin", "VoltageMax"]
            row = [frame, tick_begin, tick_end, depth_min, depth_max, voltage_min, voltage_max]
            df.loc[frame] = row

            # for next loop (because i will be incremented), do -1
            i -= 1
        i += 1

    print(df)
    return df

def voltage_to_current(voltage:float, upper_mA:float, lower_mA:float):
    # does not consider 10-bit mapping
    range_mA = abs(upper_mA - lower_mA)
    current = range_mA * (voltage / 5.0) + lower_mA  # ETL lower to upper [mA] <=> DAQ 0 to 5
[V]
    return current

def current_to_depth(current:float, calib_coeff:float, calib_const:float):
    # does not consider 10-bit mapping
    return calib_coeff*current + calib_const

# output xyz image (TIFF file)
# use (Z, Y, X) shape
def output_xyz_files(img:np.array, dtype, savedir, filename, dot_ext, additional_str=None):
    s = filename + dot_ext
    if additional_str is not None:
        s = filename + "_" + additional_str + dot_ext
    fullpath = os.path.join(savedir, s)
```

```
    tifffile.imsave(fullpath, img, shape=img.shape, dtype=dtype)

def output_xyz_as_xy_files(img_3d:np.array, time_ms, dtype, savedir, fname_serial, dot_ext,
additional_str=None):
    # naming rule
    # name_xy_z000001_t000001ms.tif
    for z in range(img_3d.shape[0]):
        filename=f"{fname_serial}_xy_t{int(time_ms):09}ms_z{z:06}"
        if additional_str is None:
            filename = filename + dot_ext
        else:
            filename = filename + "_" + additional_str + dot_ext
        fullpath = os.path.join(savedir, filename)
        tifffile.imsave(fullpath, img_3d[z], dtype=dtype)

# For reducing data amount
def scaling_xy_size_to(img_zyx:np.array, width_scale=0.5, height_scale=0.5):
    new_xy_shape = (int(img_zyx.shape[1]*height_scale), int(img_zyx.shape[2]*width_scale))
    img_scaled = np.zeros(shape=(img_zyx.shape[0], new_xy_shape[0], new_xy_shape[1]),
dtype=img_zyx.dtype)
    for z in range(img_zyx.shape[0]):
        img_scaled[z] = cv2.resize(img_zyx[z], new_xy_shape, interpolation=cv2.INTER_LINEAR)
    return img_scaled

def count_t_max_frames(df_aligned, waveform):
    print("Counting the actual number of time-points...")
    xyz_tif_count = 0
    prev_elapsed_ms = 0
    initial_tick_ms = df_aligned.at[0, "TickBegin"] * 1e-4
    max_frames_org = len(df_aligned.index)
    for frame_org in range(max_frames_org):
        # If Sawtooth, skip frame(s) while "returning" to initial position
        if waveform == "sawtooth":
            v_min = df_aligned.at[frame_org, "VoltageMin"]
            v_max = df_aligned.at[frame_org, "VoltageMax"]
            v_sub = np.abs(v_max - v_min)
            if v_sub > 4.0:  # ETL swing by 0 to 5 V
                continue

        elapsed_ms = df_aligned.at[frame_org, "TickEnd"] * 1e-4 - initial_tick_ms
        if elapsed_ms >= prev_elapsed_ms + output_interval_ms:
            prev_elapsed_ms = elapsed_ms
            xyz_tif_count += 1
            print(f"¥rProcessed frames: {frame_org+1} / {max_frames_org};¥tt-count:
{xyz_tif_count}", end="")

    t_max_frames = xyz_tif_count + 1
    print(f" ... done. t_max_frames = {t_max_frames}")
    return t_max_frames

def main_xyz_as_xy(waveform=None):

    org_df = pd.read_csv(file_all_csv)
    # Voltage, Current, or Depth
    valid_columns = get_valid_inst_column(org_df)
    df_aligned = analyze_record(org_df, max_frames=img_max_frames)

    print(df_aligned.columns)

    z_min = min(df_aligned["DepthMin"])
    z_max = max(df_aligned["DepthMax"])

    z_depth = (int(math.floor(z_min)), int(math.ceil(z_max)))
    z_length = abs(z_depth[1]-z_depth[0])

    print(f"z-length: {z_length} µm")

    # load image file
    img = load_image(file_image)
```

```
    # scaling xy
    img = scaling_xy_size_to(img, width_scale=scaling_x, height_scale=scaling_y)

    shape_zyx = (int(z_length/z_pitch+1), img.shape[1], img.shape[2])
    mesh = np.zeros(shape_zyx, dtype=np.uint16)

    # For transforming z [um] to index
    z_map = [z_depth[0]+i*z_pitch for i in range(shape_zyx[0])]

    # for output of xyz image
    #checkpoint_depth = 0.0 # [um]

    xyz_tif_count = 0
    prev_elapsed_ms = 0
    max_frames_org = len(df_aligned.index)  #img.shape[0]
    initial_tick_ms = df_aligned.at[0, "TickBegin"] * 1e-4
    for frame_org in range(max_frames_org):
        # If Sawtooth, skip frame(s) while "returning" to initial position
        if waveform == "sawtooth":
            v_min = df_aligned.at[frame_org, "VoltageMin"]
            v_max = df_aligned.at[frame_org, "VoltageMax"]
            v_sub = np.abs(v_max - v_min)
            if v_sub > 4.0:  # ETL swing by 0 to 5 V
                continue

        # z-range of this frame, from DAQ recording
        depth_min = df_aligned.at[frame_org, "DepthMin"]
        depth_max = df_aligned.at[frame_org, "DepthMax"]

        # z-range of this frame in voxels (mesh)
        i_z_min = np.argmin(np.abs(z_map - depth_min))
        i_z_max = np.argmin(np.abs(z_map - depth_max))

        # apply same xy-image for z-range
        for i_z in range(i_z_min, i_z_max+1):
            mesh[i_z] = img[frame_org]

        elapsed_ms = df_aligned.at[frame_org, "TickEnd"] * 1e-4 - initial_tick_ms
        #if depth_min < checkpoint_depth and checkpoint_depth <= depth_max:
        if elapsed_ms >= prev_elapsed_ms + output_interval_ms:
            fname = os.path.basename(file_image)
            output_xyz_as_xy_files(mesh, int(elapsed_ms), dtype=np.uint16,
                savedir=dir_save, fname_serial=fname, dot_ext=".tif")
            prev_elapsed_ms = elapsed_ms
            xyz_tif_count += 1
            print(f"Processed frames: {frame_org+1} / {max_frames_org};¥toutput:
{xyz_tif_count}")

def main_xyz_as_4d(waveform=None):

    org_df = pd.read_csv(file_all_csv)
    # Voltage, Current, or Depth
    valid_columns = get_valid_inst_column(org_df)
    df_aligned = analyze_record(org_df, max_frames=img_max_frames)

    max_frames_org = len(df_aligned.index)  #img.shape[0]

    # take smaller number of rows (frames)
    #img_max_frames = min(img_max_frames, len(df_aligned))
    print(df_aligned.columns)

    z_min = min(df_aligned["DepthMin"])
    z_max = max(df_aligned["DepthMax"])

    z_depth = (int(math.floor(z_min)), int(math.ceil(z_max)))
    z_length = abs(z_depth[1]-z_depth[0])
    print(f"z-length: {z_length} µm")

    t_begin = min(df_aligned['TickBegin'])  # 0.1 [micro sec]
```

```
    t_max = max(df_aligned['TickEnd'])  # 0.1 [micro sec]
    t_length = (t_max - t_begin) * 1e-7 # [sec]
    # correct t_max_frames is necessary to avoid to append blank frames
    t_max_frames = count_t_max_frames(df_aligned, waveform)
    print(f"t_length={t_length:.3f} [sec], t_max_frames={t_max_frames} frame")

    # load image file
    img = load_image(file_image)
    # scaling xy
    img = scaling_xy_size_to(img, width_scale=scaling_x, height_scale=scaling_y)

    shape_tzyx = (t_max_frames, int(z_length/z_pitch+1), img.shape[1], img.shape[2])

    # For transforming z [um] to index
    z_map = [z_depth[0]+i*z_pitch for i in range(shape_tzyx[1])]

    # create 4d space on disk
    fname = os.path.basename(file_image)
    fname = fname.replace(".tif", ".ome.tif")  # using ".ome.tif" instead of ".tif" is important
in NIS-E
    filepath = os.path.join(dir_save, fname)

    # Warning about memory usage
    mapsize_GB = shape_tzyx[0]*shape_tzyx[1]*shape_tzyx[2]*shape_tzyx[3]*2 / 1024**3  # mono
color uint16
    print(f"4D map size ~ {mapsize_GB:.1f} GiB.  (t,z,y,x)={shape_tzyx}")
    if mapsize_GB > 40.0:
        print("Map size of the reconst. data will be larger than 40 GiB.")

    ### Create OME Metadata ###
    # initial creation of metadata --- this might be enough
    metadata = {
        'axes': 'TZYX',
        'Pixels': {
                'PhysicalSizeX': pixel_size_x,
                'PhysicalSizeXUnit': 'µm',
                'PhysicalSizeY': pixel_size_y,
                'PhysicalSizeYUnit': 'µm',
                'PhysicalSizeZ': z_pitch,
                'PhysicalSizeZUnit': 'µm'
            },
    }

    mesh4d = tifffile.memmap(
        filepath,
        shape_tzyx,
        dtype='uint16',
        imagej=use_imagej,
        ome = use_ome,
        bigtiff=use_bigtiff,
        metadata=metadata,
    )

    xyz_tif_count = 0
    prev_elapsed_ms = 0
    initial_tick_ms = df_aligned.at[0, "TickBegin"] * 1e-4
    for frame_org in range(max_frames_org):
        # If Sawtooth, skip frame(s) while "returning" to initial position
        if waveform == "sawtooth":
            v_min = df_aligned.at[frame_org, "VoltageMin"]
            v_max = df_aligned.at[frame_org, "VoltageMax"]
            v_sub = np.abs(v_max - v_min)
            if v_sub > 4.0:  # ETL swing by 0 to 5 V
                continue

        # z-range of this frame, from DAQ recording
        depth_min = df_aligned.at[frame_org, "DepthMin"]
        depth_max = df_aligned.at[frame_org, "DepthMax"]
```

```
        # z-range of this frame in voxels (mesh)
        i_z_min = np.argmin(np.abs(z_map - depth_min))
        i_z_max = np.argmin(np.abs(z_map - depth_max))

        # apply same xy-image for z-range
        for i_z in range(i_z_min, i_z_max+1):
            mesh4d[xyz_tif_count, i_z] = img[frame_org]

        elapsed_ms = df_aligned.at[frame_org, "TickEnd"] * 1e-4 - initial_tick_ms
        if elapsed_ms >= prev_elapsed_ms + output_interval_ms:
            prev_elapsed_ms = elapsed_ms
            xyz_tif_count += 1
            print(f"¥rProcessed frames: {frame_org+1} / {max_frames_org};¥tt-count:
{xyz_tif_count}", end="")

            # copy current xyz-image to next time-point
            mesh4d[xyz_tif_count] = mesh4d[xyz_tif_count-1]

            # update mem-mapped file
            if(xyz_tif_count % 305 == 0):
                mesh4d.flush()

    print("")
    mesh4d.flush()
    del mesh4d

if __name__ == "__main__":
    if not os.path.exists(dir_save):
        os.mkdir(dir_save)

    # Turn on MemoryPool in CuPy
    #cp.cuda.set_allocator(cp.cuda.MemoryPool().malloc)

    t_begin = time.perf_counter()

    print(f"Waveform: {etl_waveform}")

    #main_xyz_as_xy(waveform=etl_waveform)
    main_xyz_as_4d(waveform=etl_waveform)

    t_end = time.perf_counter()
    print(f"Elapsed time = {t_end - t_begin:.1f} sec")
```

Code A4. Form/CameraSetting.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

using ETL_system.LiveImaging;
using csAcq4;
using Hamamatsu.DCAM4;
using NationalInstruments.Restricted;

namespace ETL_system
{
    /// <summary>
    /// Form for setting parameters of cameras.
    /// </summary>
    public partial class CameraSetting : Form
    {
        int gain_max = 255;
        int gain_step;
        bool gain_scrolled = false;
        bool gain_text_changed = false;
        CaptureParameters capParams;

        private AnalogControl.AnalogSignal.AnalogOutput aoTriggering;
        private AnalogControl.AnalogSignal.AnalogInput aiTriggering;
        private RecordingLog recLog;

        private bool ProgramChangedExpectedTime = false;
        private bool ProgramChangedMaxFrames = false;

        public RecordingLog RecLog { get => recLog; }
        public RingBuffer<RecordingLog.SingleRow> LiveProjLogBuffer { get; set; }
        public ImageProjection.LiveProjectionBuffer Live2DProjectionBuffer { get; set; }
        public bool flagLiveProjection = false;
        List<dynamic> LevelTrigger_Controls = new List<dynamic>();

        public CaptureParameters CaptureParams { get => capParams; }
        public IntPtr HDCAMPtr { get; set; }

        private OperatorForm _opform;
        private HighResolutionTimer gTimer;  // global timer
        public OperatorForm OPForm
        {
            get => _opform;
            set
            {
                _opform = value;
                gTimer = _opform.GlobalTimer;
                recLog = _opform.RecLog;
            }
        }

        public CameraSetting()
        {
            InitializeComponent();
            capParams = new CaptureParameters();

            // Set initial parameters
            List<ComboBox> comboBox_list = new List<ComboBox>(){
                comboBox_bit_depth, comboBox_bin, comboBox_subarray, comboBox_readout_speed,
```

```
comboBox_global_exposure,
                comboBox_trigger_source, comboBox_trigger_connector, comboBox_trigger_active,
comboBox_active_polarity,
            };
            foreach (var cb in comboBox_list)
                cb.SelectedIndex = 0;

            // Group of Controls for Level Trigger
            LevelTrigger_Controls = new List<dynamic>()
            {
                label1_level_trigger, label1_level_high, label1_level_high_ms,
                label1_level_low, label1_level_low_ms,
                numericUpDown_leveltrigger_high_length,
                numericUpDown_leveltrigger_low_length
            };

            gain_step = gain_max / hScrollBar_gain1.Maximum;

            aoTriggering = new AnalogControl.AnalogSignal.AnalogOutput();
            aiTriggering = new AnalogControl.AnalogSignal.AnalogInput();
        }

        public void StopTriggering()
        {
            aoTriggering.StopAndDispose();
            aiTriggering.StopAndDispose();

            aoTriggering = new AnalogControl.AnalogSignal.AnalogOutput();
            aiTriggering = new AnalogControl.AnalogSignal.AnalogInput();
        }

        /// <summary>
        /// WIP.
        /// </summary>
        /// <param name="mydcam"></param>
        public void ApplyCameraPropeties(IntPtr HDCAMPtr)
        {
            if (HDCAMPtr != null)
            {
                capParams.HDCAMPtr = HDCAMPtr;

                int bitDepth = capParams.GetBitDepth(comboBox_bit_depth.SelectedItem.ToString());
                capParams.SetDCAMProps(DCAMIDPROP.BITSPERCHANNEL, bitDepth);

                // DCAM reference says that Exposure time can be changed according to the change
of Binning.
                // todo: Bug. Binning could not be changed between recording sessions.
                //           Only reconnecting the camera can change binning properly, for now.
                double oldBin = capParams.GetDCAMProps(DCAMIDPROP.BINNING);
                double newBin = capParams.GetBinning(comboBox_bin.SelectedItem.ToString());
                capParams.SetDCAMProps(DCAMIDPROP.BINNING, newBin);
                bool binChanged = (int)oldBin != (int)newBin;

                // Sub-array
                Size subarraySize =
capParams.GetSubarraySize(comboBox_subarray.SelectedItem.ToString());
                //Point subarrayPos =
capParams.GetSubarrayPositionDefault(comboBox_subarray.SelectedItem.ToString());
                Point subarrayPos = new Point((int)numericUpDown_subarray_center_x.Value,
(int)numericUpDown_subarray_center_y.Value);
                capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYHSIZE, subarraySize.Width); //
horizontal size
                capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYVSIZE, subarraySize.Height); //
vertical size
                capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYHPOS, subarrayPos.X);  // horizontal
pos
                capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYVPOS, subarrayPos.Y);  // horizontal
pos
                if (comboBox_subarray.SelectedIndex == 0) // This means that, if default (full
```

```
size), subarray mode is OFF.
                {
                    label_subarray_mode.Text = "OFF";
                    label_subarray_mode.ForeColor = this.ForeColor;
                    capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYMODE, DCAMPROP.MODE.OFF);
                }
                else // if not default (smaller size) , subarray mode is ON.
                {
                    label_subarray_mode.Text = "ON";
                    label_subarray_mode.ForeColor = Color.Blue;
                    capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYMODE, DCAMPROP.MODE.ON);
                }

                capParams.SetDCAMProps(DCAMIDPROP.EXPOSURETIME,
(double)numericUpDown_exposure.Value / 1000.0);  // [sec]
                capParams.SetDCAMProps(DCAMIDPROP.TRIGGER_GLOBALEXPOSURE,
capParams.GetGlobalExposure(comboBox_global_exposure.SelectedItem.ToString()));
                capParams.SetDCAMProps(DCAMIDPROP.READOUTSPEED,
capParams.GetReadoutSpeed(comboBox_readout_speed.SelectedItem.ToString()));

                capParams.SetDCAMProps(DCAMIDPROP.TRIGGERSOURCE,
capParams.GetTriggerSource(comboBox_trigger_source.SelectedItem.ToString()));
                capParams.SetDCAMProps(DCAMIDPROP.TRIGGER_CONNECTOR,
capParams.GetTriggerConnector(comboBox_trigger_connector.SelectedItem.ToString()));
                capParams.SetDCAMProps(DCAMIDPROP.TRIGGERACTIVE,
capParams.GetTriggerActive(comboBox_trigger_active.SelectedItem.ToString()));
                capParams.SetDCAMProps(DCAMIDPROP.TRIGGERPOLARITY,
capParams.GetActivePolarity(comboBox_active_polarity.SelectedItem.ToString()));

                // test
                //capParams.SetDCAMProps(DCAMIDPROP.OUTPUTTRIGGER_KIND,
DCAMPROP.OUTPUTTRIGGER_KIND.EXPOSURE);

#if DEBUG
                // debug
                Console.WriteLine("SubarrayMode:" +
(comboBox_subarray.SelectedIndex!=0).ToString() + " (ComboBox) => "
                    + capParams.SubarrayMode);
                Console.WriteLine("Subarray: " + comboBox_subarray.SelectedItem.ToString() + "
(ComboBox) => "
                    + "Size("+capParams.SubarraySize.Width+"x"+capParams.SubarraySize.Height+"), "
                    + "Pos("+capParams.SubarrayPosition.X+", "+capParams.SubarrayPosition.Y+")");
                Console.WriteLine("ReadoutSpeed: " +
comboBox_readout_speed.SelectedItem.ToString() + " (ComboBox) => "
                    + capParams.ReadoutSpeed + " [1=Slow, 2147483647=Fast]");
                Console.WriteLine("TriggerSource:
"+comboBox_trigger_source.SelectedItem.ToString()+ " (ComboBox) => "
                    + capParams.TriggerSource + " [1=Internal, 3=Software]");
                Console.WriteLine("TriggerConnector: " +
comboBox_trigger_connector.SelectedItem.ToString() + " (ComboBox) => "
                    + capParams.TriggerConnector + " [1=Interface, 2=BNC]");
                Console.WriteLine("TriggerActive: " +
comboBox_trigger_active.SelectedItem.ToString() + " (ComboBox) => "
                    + capParams.TriggerActive + " [1=Edge, 2=Level, 3=SyncReadout]");
                Console.WriteLine("Binning: "+comboBox_bin.SelectedItem.ToString()+" (ComboBox)
=> "
                    + capParams.Binning);
#endif
                // if Binning has changed, it might be necessary to re-allocate buffer for
recording.

                if (OPForm != null)
                    OPForm.AllocationFrameCount = (int)numericUpDown_max_frames.Value;
            }

        }
        private void HScrollBar_gain1_Scroll(object sender, ScrollEventArgs e)
        {
            if (!gain_text_changed)
```

```
            {
                if (hScrollBar_gain1.Value <= 0)
                    textBox_gain1.Text = "1";
                else if (hScrollBar_gain1.Value > gain_max)
                    textBox_gain1.Text = gain_max.ToString();
                else
                    textBox_gain1.Text = (hScrollBar_gain1.Value * gain_step).ToString();
                gain_scrolled = true;
            }
        }

        private void TextBox_gain1_TextChanged(object sender, EventArgs e)
        {
            if (textBox_gain1.Text == "")
                return;

            if (gain_scrolled)
                gain_scrolled = false;
            else
            {
                // TODO: Adjust scroll bar when changed dynamic range
                int new_gain = Convert.ToInt32(textBox_gain1.Text);
                new_gain = Math.Min(new_gain, gain_max);
                new_gain = Math.Max(new_gain, 1);
                int tmp_max = hScrollBar_gain1.Maximum;
                hScrollBar_gain1.Maximum = gain_max;
                hScrollBar_gain1.Value = new_gain;
                hScrollBar_gain1.Maximum = tmp_max;
                textBox_gain1.Text = new_gain.ToString();
            }
        }

        private void CameraSetting_FormClosing(object sender, FormClosingEventArgs e)
        {
            this.Hide();
            e.Cancel = true;
        }

        private void CameraSetting_Load(object sender, EventArgs e)
        {
        }

        public void AddCameraName(string CameraName)
        {
            comboBox_camera.Items.Add(CameraName);
            if (comboBox_camera.Items.Count == 1)
                comboBox_camera.SelectedIndex = 0;
        }
        public void SetHDCAMPtrToCaptureParams(IntPtr ptr)
        {
            capParams.HDCAMPtr = ptr;
        }

        public void SetCameraStatus(string CameraStatus)
        {
            label_cam_status.Text = "Status: " + CameraStatus;
        }

        private void NumericUpDown_exposure_ValueChanged(object sender, EventArgs e)
        {
            capParams.ExposureTime_ms = (int)(numericUpDown_exposure.Value);
            if (capParams.HDCAMPtr != null)
                capParams.SetDCAMProps(DCAMIDPROP.EXPOSURETIME,
(double)numericUpDown_exposure.Value / 1000.0);

            if (OPForm.camStatus != OperatorForm.CamStatus.Acquiring)
            {
                // calc max frame (keep the total time)
                numericUpDown_expected_time_ValueChanged(this, EventArgs.Empty);
```

```
            }
        }

        private void ComboBox_bit_depth_SelectedIndexChanged(object sender, EventArgs e)
        {
            if(comboBox_bit_depth.Text.Length > 0)
                capParams.BitDepth =
capParams.GetBitDepth((string)comboBox_bit_depth.SelectedItem);
        }

        private void ComboBox_bin_SelectedIndexChanged(object sender, EventArgs e)
        {
            if(comboBox_bin.Text.Length > 0)
                capParams.Binning = (int)capParams.GetBinning((string)comboBox_bin.SelectedItem);
        }
        private void comboBox_pixel_number_SelectedIndexChanged(object sender, EventArgs e)
        {
            if(comboBox_subarray.Text.Length > 0)
            {
                Size subSize =
capParams.GetSubarraySize(comboBox_subarray.SelectedItem.ToString());
                capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYHSIZE, subSize.Width);
                capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYVSIZE, subSize.Height);

                Point subPos =
capParams.GetSubarrayPositionDefault(comboBox_subarray.SelectedItem.ToString());
                numericUpDown_subarray_center_x.Value = (decimal)subPos.X;
                numericUpDown_subarray_center_y.Value = (decimal)subPos.Y;
                // Set props in the event "numericUpDown_subarray_center_x_ValueChanged" and
"..._y_ValueChanged"
                //capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYHPOS, subPos.X);
                //capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYVPOS, subPos.Y);
            }

            // if default frame size (full frame), subarray mode turns OFF.
            if (comboBox_subarray.SelectedIndex == 0)
            {
                label_subarray_mode.Text = "OFF";
                label_subarray_mode.ForeColor = this.ForeColor;
                capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYMODE, DCAMPROP.MODE.OFF);

                label_subarray_pos.Enabled = false;
                numericUpDown_subarray_center_x.Enabled = false;
                numericUpDown_subarray_center_y.Enabled = false;
            }
            // if not default frame size (smaller size) , subarray mode turns ON.
            else
            {
                label_subarray_mode.Text = "ON";
                label_subarray_mode.ForeColor = Color.Blue;
                capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYMODE, DCAMPROP.MODE.ON);

                label_subarray_pos.Enabled = true;
                numericUpDown_subarray_center_x.Enabled = true;
                numericUpDown_subarray_center_y.Enabled = true;
            }
        }
        private void numericUpDown_subarray_center_x_ValueChanged(object sender, EventArgs e)
        {
            capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYHPOS,
(double)numericUpDown_subarray_center_x.Value);
        }
        private void numericUpDown_subarray_center_y_ValueChanged(object sender, EventArgs e)
        {
            capParams.SetDCAMProps(DCAMIDPROP.SUBARRAYHPOS,
(double)numericUpDown_subarray_center_y.Value);
        }

        private void comboBox_global_exposure_SelectedIndexChanged(object sender, EventArgs e)
```

```
        {
            if (capParams.HDCAMPtr != null)
            {
                capParams.SetDCAMProps(DCAMIDPROP.TRIGGER_GLOBALEXPOSURE,

capParams.GetGlobalExposure(comboBox_global_exposure.SelectedItem.ToString()));
            }
        }
        private void numericUpDown_max_frames_ValueChanged(object sender, EventArgs e)
        {
            if (capParams.HDCAMPtr != null && OPForm != null)
            {
                OPForm.AllocationFrameCount = (int)numericUpDown_max_frames.Value;

                // calc total time
                if (!ProgramChangedMaxFrames)
                {
                    ProgramChangedExpectedTime = true;
                    numericUpDown_expected_time.Value = (int)((double)numericUpDown_exposure.Value
* 1e-3 * (double)numericUpDown_max_frames.Value);
                }
                ProgramChangedMaxFrames = false;
            }
        }
        private void numericUpDown_expected_time_ValueChanged(object sender, EventArgs e)
        {
            if (!ProgramChangedExpectedTime)
            {
                ProgramChangedMaxFrames = true;
                int frames = (int)Math.Ceiling((double)numericUpDown_expected_time.Value /
((double)numericUpDown_exposure.Value * 1e-3));
                numericUpDown_max_frames.Value = frames;
            }
            ProgramChangedExpectedTime = false;
        }
        private void comboBox_trigger_source_SelectedIndexChanged(object sender, EventArgs e)
        {
            if (capParams.HDCAMPtr != null)
            {
                capParams.SetDCAMProps(DCAMIDPROP.TRIGGERSOURCE,
                    capParams.GetTriggerSource(comboBox_trigger_source.SelectedItem.ToString()));
            }

            groupBox_external_trigger.Enabled = comboBox_trigger_source.Text.Equals("External");
            comboBox_ext_trigger_mode_SelectedIndexChanged(null, EventArgs.Empty);

        }
        private void comboBox_trigger_active_SelectedIndexChanged(object sender, EventArgs e)
        {
            if (capParams.HDCAMPtr != null)
            {
                capParams.SetDCAMProps(DCAMIDPROP.TRIGGERACTIVE,
                    capParams.GetTriggerActive(comboBox_trigger_active.SelectedItem.ToString()));
            }
        }

        private void comboBox_trigger_connector_SelectedIndexChanged(object sender, EventArgs e)
        {
            if (capParams.HDCAMPtr != null)
            {
                capParams.SetDCAMProps(DCAMIDPROP.TRIGGER_CONNECTOR,

capParams.GetTriggerConnector(comboBox_trigger_connector.SelectedItem.ToString()));
            }
        }
        private void comboBox_active_polarity_SelectedIndexChanged(object sender, EventArgs e)
        {
            if (capParams.HDCAMPtr != null)
            {
```

```
                capParams.SetDCAMProps(DCAMIDPROP.TRIGGERPOLARITY,

capParams.GetTriggerConnector(comboBox_active_polarity.SelectedItem.ToString()));
            }
        }
        private void comboBox_readout_speed_SelectedIndexChanged(object sender, EventArgs e)
        {
            if (capParams.HDCAMPtr != null)
            {
                capParams.SetDCAMProps(DCAMIDPROP.READOUTSPEED,
                    capParams.GetReadoutSpeed(comboBox_readout_speed.SelectedItem.ToString()));
            }
        }

        #region Preset Params
        public void Preset_XYZ_Intermittent()
        {
            comboBox_readout_speed.Text = "Slow";

            comboBox_trigger_source.Text = "Software";
            comboBox_trigger_active.Text = "Edge";

            comboBox_trigger_connector.Text = "BNC";
            comboBox_active_polarity.Text = "Positive";
        }
        public void Preset_XYT()
        {
            // TODO: If exposure time is <20 ms, we recommend using a method other than
SoftwareTriggering.
            comboBox_readout_speed.Text = "Fast";

            comboBox_trigger_source.Text = "External";
            comboBox_trigger_active.Text = "SyncReadout";
            comboBox_active_polarity.Text = "Positive";
        }
        public void Preset_XYZT_Continuous()
        {
            comboBox_readout_speed.Text = "Fast";

            comboBox_trigger_source.Text = "External";
            comboBox_trigger_active.Text = "SyncReadout";
            comboBox_trigger_connector.Text = "BNC";
            comboBox_active_polarity.Text = "Positive";
        }
        public void Preset_XYT_Live()
        {
            comboBox_readout_speed.Text = "Fast";

            comboBox_trigger_source.Text = "Internal";
            comboBox_trigger_active.Text = "SyncReadout";
            comboBox_active_polarity.Text = "Positive";
        }
        #endregion

        private void button_apply_test_Click(object sender, EventArgs e)
        {
            ApplyCameraPropeties(this.HDCAMPtr);
        }
        private void button_ext_trigger_off_Click(object sender, EventArgs e)
        {
            StopTriggering();
        }

        private void button_ext_trigger_begin_Click(object sender, EventArgs e)
        {
            string mode = comboBox_trigger_active.SelectedItem.ToString();

            if (aoTriggering.RunningTask != null)
            {
```

```
                aoTriggering.StopAndDispose();

            }
            if(aiTriggering.RunningTask != null)
            {
                if(aiTriggering.RecordingLog.VoltageTiming.Count > 0)
                    aiTriggering.RecordingLog.WriteOutVoltage();
                aiTriggering.StopAndDispose();
            }

            if (mode == "Level")
            {
                double highT = (double)numericUpDown_leveltrigger_high_length.Value / 1000.0;
                double lowT  = (double)numericUpDown_leveltrigger_low_length.Value / 1000.0;

                aoTriggering.CreateAOTaskLevelTrigger(
                    AOChannel: comboBox_ext_trig_aochannel.Text,
                    LowStateLength_sec: lowT, HighStateLength_sec: highT,
                    SamplesPerBuffer: 5000, CyclesPerBuffer: 50,
                    Waveform: AnalogControl.WaveformType.LevelTrigger,
                    WaveAmplitude: 3.3);

                aiTriggering.CreateAITask(
                    AIChannel: comboBox_ext_trig_aichannel.Text,
                    VoltageDataType: RecordingLog.EXTTRIG);

                // for Live 2D Projection
                aiTriggering.flagLive2DProjection = this.flagLiveProjection;
                if (flagLiveProjection)
                    aiTriggering.Live2DProjectionBuffer = this.Live2DProjectionBuffer;

                if (!gTimer.IsRunning)
                    gTimer.Start();

                aiTriggering.StartTask(GlobalTimer: gTimer, recLog);
                aoTriggering.StartTask();
            }

            else if (mode == "SyncReadout")
            {
                double frequency = 1000.0 / (double)numericUpDown_exposure.Value;

                aoTriggering.CreateAOTaskSyncReadoutTrigger(
                    AOChannel: comboBox_ext_trig_aochannel.Text,
                    Frequency_Hz: frequency,
                    SamplesPerBuffer: 1e5, CyclesPerBuffer: 10,
                    Waveform: AnalogControl.WaveformType.SyncReadoutTrigger,
                    WaveAmplitude: 3.3);

                aiTriggering.CreateAITask(
                    AIChannel: comboBox_ext_trig_aichannel.Text,
                    VoltageDataType: RecordingLog.EXTTRIG);

                // for Live 2D Projection
                aiTriggering.flagLive2DProjection = this.flagLiveProjection;
                if (flagLiveProjection)
                    aiTriggering.Live2DProjectionBuffer = this.Live2DProjectionBuffer;

                if (!gTimer.IsRunning)
                    gTimer.Start();

                Console.WriteLine($"Started Ext.Trigger as SyncRead. gTimer elapsed =
{gTimer.ElapsedTicks}");

                aiTriggering.StartTask(GlobalTimer: gTimer, recLog);
                aoTriggering.StartTask();
            }
        }
```

```
        private void label3_Click(object sender, EventArgs e)
        {

        }

        private void label6_Click(object sender, EventArgs e)
        {

        }

        private void label10_Click(object sender, EventArgs e)
        {

        }

        private void comboBox_ext_trigger_mode_SelectedIndexChanged(object sender, EventArgs e)
        {
            bool isLevelTrigger = (comboBox_trigger_active.Text == "Level");
            foreach (var ctrl in LevelTrigger_Controls)
                ((Control)ctrl).Enabled = isLevelTrigger;
        }

        #region Sensor Cooler
        private void __Unused__(object sender, EventArgs e)
        {
            var status = capParams.GetDCAMProps(DCAMIDPROP.SENSORCOOLERSTATUS);          //
0=NONE, 1=OFF, 2=READY, 3=BUSY, 4=ALWAYS, 5=WARNING
            var temp = capParams.GetDCAMProps(DCAMIDPROP.SENSORTEMPERATURE);             //
celcius degree
            var tempTarget = capParams.GetDCAMProps(DCAMIDPROP.SENSORTEMPERATURETARGET); //
celcius degree (?)
            var mode = capParams.GetDCAMProps(DCAMIDPROP.SENSORCOOLER);                  // 1=OFF,
2=ON, 4=MAX
            var fan = capParams.GetDCAMProps(DCAMIDPROP.SENSORCOOLERFAN);                // 0=ON,
1=OFF

            capParams.SetDCAMProps(DCAMIDPROP.SENSORCOOLER, 4.0, false);
            capParams.SetDCAMProps(DCAMIDPROP.SENSORTEMPERATURETARGET, -15.0, false);
            mode = capParams.GetDCAMProps(DCAMIDPROP.SENSORCOOLER);  // 1=OFF, 2=ON, 4=MAX
            tempTarget = capParams.GetDCAMProps(DCAMIDPROP.SENSORTEMPERATURETARGET);  // celcius
degree (?)

        }
        private void InitCoolerSetting()
        {
            double mode = capParams.GetDCAMProps(DCAMIDPROP.SENSORCOOLER);  // 1=OFF, 2=ON, 4=MAX
            switch(mode)
            {
                case 1.0:
                    comboBox_cooler_mode.SelectedItem = "OFF";
                    break;
                case 2.0:
                    comboBox_cooler_mode.SelectedItem = "ON";
                    break;
                case 4.0:
                    comboBox_cooler_mode.SelectedItem = "MAX";
                    break;
                default:
                    comboBox_cooler_mode.SelectedItem = "MAX";
                    break;
            }
        }
        private void comboBox_cooler_mode_SelectedIndexChanged(object sender, EventArgs e)
        {
            DCAMPROP val = capParams.GetCoolerMode(comboBox_cooler_mode.SelectedItem.ToString());
            capParams.SetDCAMProps(DCAMIDPROP.SENSORCOOLER, (double)val, false);
        }
        private void button_reflesh_temp_Click(object sender, EventArgs e)
        {
```

```
            var temp = capParams.GetDCAMProps(DCAMIDPROP.SENSORTEMPERATURE);  // celcius degree
            label_temperature.Text = temp.ToString() + " deg (C)";

            InitCoolerSetting();
        }
        #endregion

        private void CameraRotationAndFlip_ValueChanged(object sender, EventArgs e)
        {
            CameraRotationAndFlip();
        }
        public void CameraRotationAndFlip()
        {
            if (OPForm != null)
            {
                OPForm.ChangeDisplayFlipType((int)(numericUpDown_camera_rotation.Value),
                    checkBox_flip_horizontally.Checked, checkBox_flip_vertically.Checked);
            }
        }

    }
}
```

Code A5. Form/CSUSetting.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO.Ports;
using System.Linq;
using System.Management;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace ETL_system
{
    public partial class CSUSetting : Form
    {
        Dictionary<string, string> ports_dic;
        SerialPortExtended spe;
        string last_str = "";
        const string DefaultPort = "COM3";
        const int MaxRpm = 10000;
        const int MinRpm = 1500;

        // Ref. CSUX_UserManual
        // Communication
        // * Speed: 115,200 bps
        // * Data bit: 8
        // * Parity check: None
        // * Stop bit : 1
        // * Line-feed character: CR (¥r = 0x0d)
        // * Separating character: colon or space
        // ① * Case sensitivity: : Yes

        public CSUSetting()
        {
            InitializeComponent();

            spe = new SerialPortExtended();
        }

        private void CSUSetting_Load(object sender, EventArgs e)
        {
            bool ExistsDefaultPort = false;
            // Get serial ports
            ports_dic = Utils.GetSerialPortList();
            foreach (var name in ports_dic.Keys)
            {
                comboBox_port.Items.Add(name);
                if (ports_dic[name] == DefaultPort)
                {
                    comboBox_port.SelectedIndex = comboBox_port.Items.Count - 1;
                    ExistsDefaultPort = true;
                }
            }

            if (comboBox_port.Items.Count > 0 && !ExistsDefaultPort)
                comboBox_port.SelectedIndex = 0;
            if (!(comboBox_port.SelectedIndex >= 0))
                button_connect.Enabled = false;

            // automatically open serial connection
            if (ExistsDefaultPort)
            {
                button_connect_Click(null, EventArgs.Empty);
                ExistsDefaultPort = false;
```

```
            }

        }

        private void button_connect_Click(object sender, EventArgs e)
        {
            if (comboBox_port.SelectedIndex < 0) return;
            BeginInvoke(new Action(delegate
            {
                string id = ports_dic[(string)comboBox_port.SelectedItem];
                try
                {
                    if (spe != null && spe.IsOpen)
                        spe.Close();
                    else
                    {
                        spe = new SerialPortExtended(id, SerialPortExtended.DeviceTypeEnum.CSUX);
                        spe.OpenPort();

                        spe.DataReceived += (new SerialPortUtilsEx()).DataReceivedHandler_General;
                        spe.GeneralUseEvent += CallBackDataReceivedEvent;
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.ToString());
                }
                button_connect.Text = Utils.ConnectionButtonHelper(spe.IsOpen);
                label_connection = Utils.ConnectionLabelHelper(spe.IsOpen, label_connection);
            }));

        }

        public void SendSetRotationSpeed(int rpm)
        {
            string text = "MS, " + rpm;  // CR <=> ¥r
            byte[] bytes = Encoding.Default.GetBytes(text);
            string hex = BitConverter.ToString(bytes);
            if (spe.IsOpen)
            {
                spe.IsGeneralUse = true;
                // send   "MS, [n]" 0x0d
                spe.Write(text);
                spe.Write(new byte[]{0x0d}, 0, 1);
                textBox_console.AppendText(text + Environment.NewLine);
            }
        }
        public void SendCheckRotationSpeed()
        {
            string text = "MS, ?";
            if (spe.IsOpen)
            {
                spe.IsGeneralUse = true;
                // send   "MS, ?"
                spe.Write(text);
                spe.Write(new byte[] { 0x0d }, 0, 1);
                textBox_console.AppendText(text + Environment.NewLine);
            }
        }


        private void CSUSetting_FormClosing(object sender, FormClosingEventArgs e)
        {
            e.Cancel = true;
            this.Hide();
        }

        private void numericUpDown_rpm_ValueChanged(object sender, EventArgs e)
        {
```

```
            SendSetRotationSpeed((int)numericUpDown_rpm.Value);

        }
        #region CallBack
        public void CallBackDataReceivedEvent(SerialPortExtended.GeneralUseEventArgs e)
        {
            byte[] data = e.Data;
            bool clearBuffer = e.CanClearBuffer;
            try
            {
                if (data.Length > 0)
                {
                    string strHex = BitConverter.ToString(data);
                    //string str = Utils.ConvertHex(strHex, sep: "-").Replace(":A ",
"").Replace("¥n", "").Trim();
                    string str = Utils.ConvertHex(strHex, sep: "-").Replace("¥n", "").Trim();

                    bool doUpdate = !str.Equals(last_str);
                    if (doUpdate)
                    {
                        this.Invoke((MethodInvoker)(() => textBox_console.AppendText(str +
Environment.NewLine)));
                        last_str = str;
                    }
                }
            }
            finally
            {
                if (clearBuffer)
                {
                    spe.DiscardInBuffer();
                }
            }
        }
        #endregion

        private void textBox_console_TextChanged(object sender, EventArgs e)
        {
        }

        private void label2_Click(object sender, EventArgs e)
        {

        }

        private void button_ms_max_Click(object sender, EventArgs e)
        {
            int rpmOld = (int)numericUpDown_rpm.Value;
            numericUpDown_rpm.Value = MaxRpm;
            if (rpmOld == MaxRpm)
                SendSetRotationSpeed(MaxRpm);
        }

        private void button_ms_min_Click(object sender, EventArgs e)
        {
            int rpmOld = (int)numericUpDown_rpm.Value;
            numericUpDown_rpm.Value = MinRpm;
            if (rpmOld == MinRpm)
                SendSetRotationSpeed(MinRpm);
        }

        private void button_ms_check_Click(object sender, EventArgs e)
        {
            SendCheckRotationSpeed();
        }
    }
}
```

Code A6. Form/ETLSetting.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO.Ports;
using System.Management;

using LensDriverController.Logic;
using LensDriverController.Settings;
using AnalogControl;  // Use NIDAQ AO & AI

using NationalInstruments.DAQmx;

namespace ETL_system
{
    /// <summary>
    /// Form to set parameters of ETL.
    /// </summary>
    public partial class ETLSetting : Form
    {
        #region Local Variables
        Dictionary<string, string> ports_dic;
        private LensDriver lensDriver;

        private LensDriver.OperationModes operationMode;

        private bool _sensorControl;
        private int _focalPower;
        private int _current;
        private int _maxFrequency;
        private double _frequency;
        private int _lowerSignalLevel;
        private int _upperSignalLevel;
        private int _desiredPosition;

        private bool _sensorControlState;
        private int _focalPowerState;
        private int _desiredPositionState;
        private int _currentState;
        private int _frequencyState;
        private int _lowerSignalLevelState;
        private int _upperSignalLevelState;

        private bool _showFirmwareUpgradeLabel = false;
        private double zeroDiopterCurrent = -10.66;  // checked on 2022.10.13
        private System.Timers.Timer timerETLConnIndicator;

        private float updatePositionInterval_ms=50;

        private AnalogSignal.AnalogOutput analogOutput;
        private AnalogSignal.AnalogInput analogInput;

        private OperatorForm _opform;
        private HighResolutionTimer gTimer;
        private RecordingLog recLog;

        public ETLCalibration eTLCalibration { get; private set; }
        public RecordingLog RecLog { get => recLog; }
        public RingBuffer<RecordingLog.SingleRow> LiveProjLogBuffer { get; set; }
        public bool flagLiveProjection = false;
```

```
        #endregion

        #region Props
        public LensDriver LensDriver { get=>lensDriver; set { lensDriver = value; } }

        /// <summary>
        /// Updating (ideal) rate of Current or Position which taken from LensDriver and
Lens.
        /// </summary>
        public float UpdatePositionInterval_ms
        {
            get => updatePositionInterval_ms;
            set
            {
                updatePositionInterval_ms = value;
                // TODO: change to use Event
                if (OPForm != null)
                    OPForm.SetIntervalMechLogUpdate(updatePositionInterval_ms);
            }
        }
        public OperatorForm OPForm
        {
            get => _opform;
            set
            {
                _opform = value;
                gTimer = _opform.GlobalTimer;
                recLog = _opform.RecLog;
            }
        }

        public List<double> Z_MovePlanCurrent { get; set; }
        public LiveImaging.ImageProjection.LiveProjectionBuffer Live2DProjectionBuffer
{ get; set; }
        #endregion

        public ETLSetting()
        {
            InitializeComponent();
            analogOutput = new AnalogSignal.AnalogOutput();
            analogInput = new AnalogSignal.AnalogInput();

            eTLCalibration = new ETLCalibration();

            //gTimer = new HighResolutionTimer();
        }


        private void ETLSetting_FormClosing(object sender, FormClosingEventArgs e)
        {
            this.Hide();
            e.Cancel = true;
        }
        private void ETLSetting_Load(object sender, EventArgs e)
        {
            /*** These functions are executed in OperatorForm. ***
            // Get serial ports
            System.Management.ManagementClass mcW32serPort = new
System.Management.ManagementClass("Win32_SerialPort");
            ports_dic = new Dictionary<string, string>();
            foreach (ManagementObject port in mcW32serPort.GetInstances())
            {
                string caption = (string)port.GetPropertyValue("Caption");  // detailed name
                string device_id = (string)port.GetPropertyValue("DeviceID");  // use this
ID to open port
                ports_dic.Add(caption, device_id);
                comboBox_port.Items.Add(caption);
            }
```

```
            if (ports_dic.Count > 0)
            {
                comboBox_port.SelectedIndex = 0;
                foreach (string key in ports_dic.Keys)
                {
                    if (key.ToLower().IndexOf("optotune lens driver") >= 0)
                    {
                        comboBox_port.SelectedItem = key;
                        break;
                    }
                }
            }

            comboBox_waveform.SelectedIndex = 0;
            //*/
            foreach(string s in Enum.GetNames(typeof(LensDriver.OperationModes)))
            {
                if (!s.Equals("Undefined"))
                    comboBox_waveform.Items.Add(s);
            }
            comboBox_waveform.SelectedIndex = 2;

            // AnalogOutput
            foreach(var name in Enum.GetNames(typeof(WaveformType)))
                comboBox_ao_waveform.Items.Add(name);
            comboBox_ao_waveform.SelectedIndex = (int)WaveformType.Sawtooth;  // Default
Sawtooth
            checkBox_analogsignal_CheckedChanged(null, EventArgs.Empty);

            UpdatePositionInterval_ms = (int)numericUpDown_update_position_interval.Value;

            eTLCalibration.Coefficient = (double)numericUpDown_calib_coef.Value;
            eTLCalibration.Constant = (double)numericUpDown_calib_const.Value;
            eTLCalibration.CreateMapVoltageToCurrent();
        }

        #region Lens Driver Function (public)
        public LensDriver InitLensDriver()
        {
            // Instance of Lens Driver singleton
            lensDriver =
LensDriver.getInstance(LensDriverSettings.Default.lensDriverHardwareID);
            lensDriver.Deinitialize();

            //lensDriver.Message += new
Logic.LensDriverMessageEventHandler(LensDriverMessage);
            operationMode = LensDriver.OperationModes.Sinusoidal;

            lensDriver.Initalize(operationMode);

            return lensDriver;
        }
        public void ETLSwitchConnection()
        {
            bool state = lensDriver.lensConnectionEstablished;
            // if the connection established already, disconnect it
            if (state)
                lensDriver.Deinitialize();
            else
                lensDriver.Initalize(operationMode);
        }
        public void ETLSwitchConnection(bool Enabling)
        {
            bool state = lensDriver.lensConnectionEstablished;
            if (Enabling && !state)
                lensDriver.Initalize(operationMode);
            else if(!Enabling && state)
                lensDriver.Deinitialize();
        }
```

```
        public void SetConnectedDeviceName(string name)
        {
            comboBox_port.Items.Add(name);
            comboBox_port.SelectedIndex = 0;
        }
        public void UpdateConnectionState(bool State)
        {
            Utils.ConnectionLabelHelper(State, label_com_test);
        }

        public void setUpperCurrentSwing(double current__, bool setStateVariable = true,
bool encoded = true)
        {
            int current;
            if (encoded)
                current = (int)current__;
            else
                current = lensDriver.firmwareManager.EncodeCurrent(current__);

            //restrain current to limits
            if (current > lensDriver.currentUpperSoftwareLimit && current != 0) current =
lensDriver.currentUpperSoftwareLimit;
            else if (current < lensDriver.currentLowerSoftwareLimit && current != 0) current
= lensDriver.currentLowerSoftwareLimit;

            _upperSignalLevel = current;

            if (setStateVariable) _upperSignalLevelState = _upperSignalLevel;

            lensDriver.SendSetSignalCurrentCommand((Int16)current,
LensDriver.LimitType.Upper);

            if (current < _lowerSignalLevel)
            {
                setLowerCurrentSwing(current, true, encoded: true);
            }
        }

        public void setLowerCurrentSwing(int current__, bool setStateVariable = true, bool
encoded = true)
        {
            int current;
            if (encoded)
                current = (int)current__;
            else
                current = lensDriver.firmwareManager.EncodeCurrent(current__);

            //restrain current to limits
            if (current > lensDriver.currentUpperSoftwareLimit && current != 0) current =
lensDriver.currentUpperSoftwareLimit;
            else if (current < lensDriver.currentLowerSoftwareLimit && current != 0) current
= lensDriver.currentLowerSoftwareLimit;

            _lowerSignalLevel = current;

            if (setStateVariable) _lowerSignalLevelState = _lowerSignalLevel;

            lensDriver.SendSetSignalCurrentCommand((Int16)current,
LensDriver.LimitType.Lower);

            if (current > _upperSignalLevel)
            {
                setUpperCurrentSwing(current, true, encoded: true);
            }
        }

        public void setFrequency(double frequency__, bool setStateVariable = true, bool
encoded = true)
        {
```

```
            int frequency;
            if (encoded)
                frequency = (int)frequency__;
            else
                frequency = lensDriver.firmwareManager.EncodeFrequency(frequency__);

            //restrain current to limits
            if (frequency > _maxFrequency) frequency = _maxFrequency;
            else if (frequency < 250) frequency = 250;

            _frequency = frequency;

            if (setStateVariable) _frequencyState = (int)_frequency;

            lensDriver.SendSetFrequencyCommand(frequency);
        }

        public void setCurrent(double current__, bool setStateVariable = true, bool encoded
= false)
        {
            int current;
            if (encoded)
                current = (int)current__;
            else
                current = lensDriver.firmwareManager.EncodeCurrent(current__);

            //restrain current to limits
            if (current > lensDriver.currentUpperSoftwareLimit && current != 0) current =
lensDriver.currentUpperSoftwareLimit;
            else if (current < lensDriver.currentLowerSoftwareLimit && current != 0) current
= lensDriver.currentLowerSoftwareLimit;

            _current = current;

            if (setStateVariable) _currentState = _current;

            lensDriver.SendSetCurrentCommand((Int16)current);

        }
        #endregion


        #region Parameters
        private void numericUpDown_frequency_ValueChanged(object sender, EventArgs e)
        {
            numericUpDown_period.Value = (decimal)(1.0 /
(double)numericUpDown_frequency.Value * 1000.0);
        }
        private void numericUpDown_period_ValueChanged(object sender, EventArgs e)
        {
            numericUpDown_frequency.Value = (decimal)(1.0 /
(double)numericUpDown_period.Value * 1000.0);
        }
        private void ComboBox_port_SelectedIndexChanged(object sender, EventArgs e)
        {
            /*
            if (label_selected_port.Text != this.SelectedPortID)
            {
                label_selected_port.Text = this.SelectedPortID;
                label_com_test.Text = "unknown";
                label_com_test.ForeColor = Color.Black;
            }
            //*/
        }
        private void comboBox_waveform_SelectedIndexChanged(object sender, EventArgs e)
        {
            LensDriver.OperationModes op;
            if (Enum.TryParse(comboBox_waveform.Text, out op) == false
                || !Enum.IsDefined(typeof(LensDriver.OperationModes), op))
```

```
                comboBox_waveform.SelectedIndex = 0;

            EnableControls();
            switch (op)
            {
                case LensDriver.OperationModes.Current:
                    //numericUpDown_lower_current.Enabled = false;
                    //numericUpDown_upper_current.Enabled = false;
                    numericUpDown_frequency.Enabled = false;
                    numericUpDown_period.Enabled = false;
                    break;
                case LensDriver.OperationModes.Sinusoidal:
                    numericUpDown_current.Enabled = false;
                    break;
                case LensDriver.OperationModes.Rectangular:
                    numericUpDown_current.Enabled = false;
                    break;
                case LensDriver.OperationModes.Triangular:
                    numericUpDown_current.Enabled = false;
                    break;
            }
        }
        private void EnableControls()
        {
            numericUpDown_frequency.Enabled = true;
            numericUpDown_period.Enabled = true;
            numericUpDown_current.Enabled = true;
            numericUpDown_lower_current.Enabled = true;
            numericUpDown_upper_current.Enabled = true;
            numericUpDown_frequency.Enabled = true;
            numericUpDown_period.Enabled = true;

        }
        public void PresetRestrictParams()
        {
            _maxFrequency = lensDriver.firmwareManager.EncodeFrequency(100.0);    // [Hz]
            double i_o = 290.0;
            _upperSignalLevel = lensDriver.firmwareManager.EncodeCurrent(i_o);   // Input:
[mA]; Output: -4095 to 4095 (12 bit level in 293 mA (default max value))
            _lowerSignalLevel = lensDriver.firmwareManager.EncodeCurrent(-i_o);
        }


        #endregion


        #region Logging

        #endregion

        public void ApplyETLSetting()
        {
            // Wave form
            if (Enum.TryParse(comboBox_waveform.Text, out operationMode) == false
                || !Enum.IsDefined(typeof(LensDriver.OperationModes), operationMode))
                operationMode = LensDriver.OperationModes.Current;

            // Frequency
            _frequency = (double)numericUpDown_frequency.Value;

            // Upper & Lower current
            _upperSignalLevel = (int)numericUpDown_upper_current.Value;
            _lowerSignalLevel = (int)numericUpDown_lower_current.Value;

            // Current
            _current = (int)numericUpDown_current.Value;

            // Update interval
            UpdatePositionInterval_ms =
```

```
Convert.ToInt32(numericUpDown_update_position_interval.Value);


            // using NIDAQ Analog Output
            if (checkBox_analogsignal.Checked)
            {
                // run Analog Output & Input
                RunAO();
            }

            // using LensDriver
            else
            {
                lensDriver.SendSetOperationModeCommand(operationMode);

                // Frequency
                setFrequency(_frequency, encoded: false);

                // Upper & Lower current
                setUpperCurrentSwing(_upperSignalLevel, encoded: false);
                setLowerCurrentSwing(_lowerSignalLevel, encoded: false);

                // Current
                setCurrent(_current, encoded: false);

                GetSetLensDriverParams();
            }
        }
        public void button_apply_Click(object sender, EventArgs e)
        {
            ApplyETLSetting();
        }
        /// <summary>
        /// Load the updated lens params to each control (ComboBox, NumericUpDown), after
button_apply_Click.
        /// </summary>
        private void GetSetLensDriverParams()
        {
            // possibly fixed (Aug 2021)
            BeginInvoke(new Action(delegate
            {
                comboBox_waveform.SelectedItem = lensDriver.operationMode.ToString();
            }));
        }

        public void AbortAnalogControl()
        {
            if (analogOutput.RunningTask != null)
                analogOutput.StopAndDispose();
            if (analogInput.RunningTask != null)
            {
                gTimer.Stop();
                Console.WriteLine(analogInput.DataTable);
                Console.WriteLine(analogInput.CallbackCounter.ToString() + " in " +
gTimer.Elapsed_ms + " ms");
                analogInput.StopAndDispose();
#if DEBUG
                recLog.WriteOutVoltage();
#endif
            }
        }
        public void button_abort_Click(object sender, EventArgs e)
        {
            // stop NIDAQ analog signals which controlling ETL
            AbortAnalogControl();

            // stop ETL
            LensDriver.OperationModes prevOpMode = lensDriver.operationMode;
            lensDriver.SendSetOperationModeCommand(LensDriver.OperationModes.Current, false,
```

```
false);
            lensDriver.SendSetCurrentCommand(0, useQueue: false, triggerEvent: false);

            GetSetLensDriverParams();
            BeginInvoke(new Action(delegate
            {
                comboBox_waveform.SelectedItem = prevOpMode.ToString();
            }));
        }

        private void numericUpDown_update_position_interval_ValueChanged(object sender,
EventArgs e)
        {
            UpdatePositionInterval_ms = (int)numericUpDown_update_position_interval.Value;
        }

        private void button_set_plan_Click(object sender, EventArgs e)
        {
            CreateMovePlanCurrent((int)numericUpDown_xyz_partition_number.Value,
checkBox_xyz_lower_to_upper.Checked);
        }
        /// <summary>
        /// Create MovePlan (Current) to operate ETL by sending order to USB driver, not
using NIDAQ AO.
        /// </summary>
        /// <param name="TotalDivided"></param>
        public List<double> CreateMovePlanCurrent(int TotalDivided=100, bool
LowerToUpper=true)
        {
            // Note: When TotalDivided is 100, total steps is 101.
            int totalDivided = TotalDivided;
            double lower = (double)numericUpDown_lower_current.Value;
            double upper = (double)numericUpDown_upper_current.Value;

            List<double> movePlan = new List<double>();
            double deltaI = Math.Abs(upper - lower) / (double)totalDivided;
            for(int i=0; i<=totalDivided; i++)
            {
                if(LowerToUpper)
                    movePlan.Add(lower + deltaI * i);
                else
                    movePlan.Add(upper - deltaI * i);
            }
            Z_MovePlanCurrent = movePlan;
            return movePlan;
        }

        private void numericUpDown_calib_coef_ValueChanged(object sender, EventArgs e)
        {
            eTLCalibration.Coefficient = (double)numericUpDown_calib_coef.Value;
        }

        private void numericUpDown_calib_const_ValueChanged(object sender, EventArgs e)
        {
            eTLCalibration.Constant = (double)numericUpDown_calib_const.Value;
        }

        private void numericUpDown_lower_current_Validating(object sender, CancelEventArgs
e)
        {
            double parsedDouble = (double)numericUpDown_lower_current.Value;
            if (parsedDouble <= 292.77 && parsedDouble >= -292.77)
            {
                if (parsedDouble >
lensDriver.firmwareManager.DecodeCurrent(lensDriver.currentUpperSoftwareLimit))
                {
                    // nothing to do
                    //UpdateGUI();
                }
```

```
                else
                {

lensDriver.SendCalibrationCommand(FirmwareManager.CommandTypes.SetLowerSoftwareLimit,
(Int16)lensDriver.firmwareManager.EncodeCurrent(parsedDouble));
                    //TODO - Workaround - Get new focal power range

//lensDriver.SendSetTemperatureLimitsCommand(Settings.LensDriverSettings.Default.minOperati
onTemperature, Settings.LensDriverSettings.Default.maxOperationTemperature);
                }
            }
            else
            {
                MessageBox.Show("Please enter valid value.");
                //UpdateGUI();
            }
        }

        private void numericUpDown_upper_current_Validating(object sender, CancelEventArgs
e)
        {
            double parsedDouble = (double)numericUpDown_upper_current.Value;
            if (parsedDouble <= 292.77 && parsedDouble >= -292.77)
            {
                if (parsedDouble <
lensDriver.firmwareManager.DecodeCurrent(lensDriver.currentLowerSoftwareLimit))
                {
                    // nothing to do
                    //UpdateGUI();
                }
                else
                {

lensDriver.SendCalibrationCommand(FirmwareManager.CommandTypes.SetUpperSoftwareLimit,
(Int16)lensDriver.firmwareManager.EncodeCurrent(parsedDouble));
                    //TODO - Workaround - Get new focal power range

//lensDriver.SendSetTemperatureLimitsCommand(Settings.LensDriverSettings.Default.minOperati
onTemperature, Settings.LensDriverSettings.Default.maxOperationTemperature);
                }
            }
            else
            {
                MessageBox.Show("Please enter valid value.");
                //UpdateGUI();
            }
        }

        private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
        {

        }

        private void checkBox_analogsignal_CheckedChanged(object sender, EventArgs e)
        {
            if(checkBox_analogsignal.Checked)
            {
                checkBox_ao_sensor_control.Enabled = true;
                comboBox_ao_waveform.Enabled = true;
                label_ao_waveform.Enabled = true;
            }
            else
            {
                checkBox_ao_sensor_control.Enabled = false;
                comboBox_ao_waveform.Enabled = false;
                label_ao_waveform.Enabled = false;
            }
        }
```

```
        private void button_ao_run_Click(object sender, EventArgs e)
        {
            // avoid to run two AO-ETL tasks simultaneously (debugging)
            if (analogOutput.RunningTask != null)
                analogOutput.StopAndDispose();

            RunAO();
        }

        private void RunAO()
        {
            // using NIDAQ Analog Output
            // When uses Analog mode, keeps selected wave function for generating Analog
Output.
            LensDriver.OperationModes waveFunction = operationMode;
            operationMode = LensDriver.OperationModes.Analog;
            lensDriver.SendSetOperationModeCommand(operationMode);

            // Upper & Lower current
            setUpperCurrentSwing(_upperSignalLevel, encoded: false);
            setLowerCurrentSwing(_lowerSignalLevel, encoded: false);

            GetSetLensDriverParams();

            // ---- NIDAQ Analog Output ----
            double spb = 10000;  // Signals per buffer
            double cpb = 10;     // Cycles per buffer
            // Frequency
            _frequency = (double)numericUpDown_frequency.Value;

            // Upper & Lower current
            _upperSignalLevel = (int)numericUpDown_upper_current.Value;
            _lowerSignalLevel = (int)numericUpDown_lower_current.Value;

            // Current
            _current = (int)numericUpDown_current.Value;

            // select waveform
            WaveformType waveform;
            if(!Enum.TryParse<WaveformType>(comboBox_ao_waveform.SelectedItem.ToString(),
out waveform))
            {
                waveform = WaveformType.Sine;
                MessageBox.Show("Error", "Invalid AO waveform.");
            }

            //if (analogOutput.RunningTask != null)
            //    analogOutput.StopAndDispose();

            analogOutput.CreateAOTaskETL(
                comboBox_aochannel.Text,
                _frequency,
                spb,
                cpb,
                waveform,
                5.0);

            // Analog Input
            analogInput.SamplesPerChannel = 100; // [samples]
            analogInput.UpdateRate = 1000; // [Hz]
            // for Live 2D Projection
            analogInput.flagLive2DProjection = this.flagLiveProjection;
            if (flagLiveProjection)
                analogInput.Live2DProjectionBuffer = this.Live2DProjectionBuffer;

            analogInput.CreateAITask();

            // Start AO and AI task
            analogOutput.StartTask();
```

```
            if(!gTimer.IsRunning)
                gTimer.Start();
            Console.WriteLine("AI starting:" + gTimer.ElapsedTicks + " ticks");
            analogInput.StartTask(gTimer, recLog);
            Console.WriteLine("AI started:" + gTimer.ElapsedTicks + " ticks. means " +
gTimer.Elapsed_ms + " ms");
        }

        private void radioButton_obj16x08na_CheckedChanged(object sender, EventArgs e)
        {
            if(radioButton_obj16x08na.Checked)
                numericUpDown_calib_coef.Value = (decimal)-3.55;
            else if(radioButton_obj25x11na.Checked)
                numericUpDown_calib_coef.Value = (decimal)-1.37;
        }

        private void numericUpDown_current_ValueChanged(object sender, EventArgs e)
        {
            if(operationMode == LensDriver.OperationModes.Current)
            {
                //button_apply_Click(null, EventArgs.Empty);
            }
        }
        private void numericUpDown_depth_ValueChanged(object sender, EventArgs e)
        {
            // Apply after calculation of desired current (Mar 2022)

            if(operationMode == LensDriver.OperationModes.Current)
            {

            }

        }
        private void checkBox_use_calibration_info_CheckedChanged(object sender, EventArgs
e)
        {
            // Enable "Depth" (autoset current) mode (Mar 2022)
            // Enable related labels and numericUpDown
            Control[] ctrls = new Control[] {
                label_depth, label_depth2, label_depth_top, label_depth_bottom,
                label_depth_unit1, label_depth_unit2, label_depth_unit3,
                numericUpDown_depth, numericUpDown_depth_top, numericUpDown_depth_bottom};
            foreach (var c in ctrls)
                c.Enabled = checkBox_use_calibration_info.Checked;
        }

        private void numericUpDown_depth_top_ValueChanged(object sender, EventArgs e)
        {
            double coef = (double)numericUpDown_calib_coef.Value;
            double lowerCurrent = coef * (double)numericUpDown_depth_top.Value;
            double upperCurrent = coef * (double)numericUpDown_depth_bottom.Value;

            numericUpDown_depth_top.Maximum = (decimal)coef *
numericUpDown_lower_current.Minimum;
            numericUpDown_depth_top.Minimum = (decimal)coef *
numericUpDown_lower_current.Maximum;
            numericUpDown_depth_bottom.Maximum = (decimal)coef *
numericUpDown_upper_current.Minimum;
            numericUpDown_depth_bottom.Minimum = (decimal)coef *
numericUpDown_upper_current.Maximum;

            if ((decimal)lowerCurrent < numericUpDown_lower_current.Minimum
                || (decimal)lowerCurrent > numericUpDown_lower_current.Maximum)
                return;
            if ((decimal)upperCurrent < numericUpDown_upper_current.Minimum
                || (decimal)upperCurrent > numericUpDown_upper_current.Maximum)
                return;

            numericUpDown_lower_current.Value = (decimal)lowerCurrent;
```

```
            numericUpDown_upper_current.Value = (decimal)upperCurrent;
        }

        private void button_zero_diopter_Click(object sender, EventArgs e)
        {
            comboBox_waveform.SelectedItem = "Current";
            operationMode = LensDriver.OperationModes.Current;
            numericUpDown_current.Value = (decimal)(zeroDiopterCurrent);
            ApplyETLSetting();
        }
    }
}
```

Code A7. Form/MainForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Diagnostics;

using LensDriverController.Forms;  // ETL driver

using Hamamatsu.DCAM4;  // HPK Camera driver
using Hamamatsu.subacq4;

namespace ETL_system
{
    public partial class MainForm : Form
    {
        const string APP_ID = "seq4d_system";
        Stopwatch g_stopwatch;

        private Form[] form_list;

        public CameraSetting camSetForm;
        public ETLSetting etlSetForm;
        public SerialPortUtilsEx g_spuex;  // use this instance globally
        public SerialPortConsole spconsoleForm;

        private Point prev_location;
        private bool flagActivatedOtherForms = false;  // when activated this MainForm,
activate others too

        // Optotune driver
        //private LensDriverFormEx etl_form;
        private Form etlForm;
        private OperatorForm opForm;
        private ImageDisplay imgDisplayForm;// show captured data
        private Visualizer visualizerForm; // Monitoring mechanical instruments

        // Hamamatsu driver
        private csAcq4.FormMain hpkCamForm;

        // FOV position controller
        private PositionController posCon;
        private PositionControllerSetting posConSetForm;

        // CSU-MP (Modified CSU-X1)
        private CSUSetting csuSetForm;

        private ImageProjDisplay imgProjForm;

        private bool isMoving = false;

        // App Status
        private System.Diagnostics.Process process;
        private System.Diagnostics.PerformanceCounter cpuCounter;
        private System.Timers.Timer statusTimer;
        private Dictionary<int, string> cpuUsageThreshold = new Dictionary<int, string>
            {{20, "Low <20" }, {60, "Mid <60"}, {90, "High <90"}, {100, "Highest ~100" } };


        public csAcq4.FormMain HPKCameraDriverForm { get { return hpkCamForm; } }
```

```
        public MainForm()
        {
            InitializeComponent();

            // App status
            process = System.Diagnostics.Process.GetCurrentProcess();
            cpuCounter = new PerformanceCounter("Processor", "% Processor Time", "_Total");
            statusTimer = new System.Timers.Timer();
            statusTimer.Interval = 1000;
            statusTimer.Elapsed += StatusTimer_Elapsed;
            statusTimer.Start();

            this.StartPosition = FormStartPosition.CenterScreen;

            g_spuex = new SerialPortUtilsEx();  // get all ports
            spconsoleForm = new SerialPortConsole(ref g_spuex);
            camSetForm = new CameraSetting(); // Hamamatsu
            etlSetForm = new ETLSetting();
            //etl_form = new LensDriverFormEx(); // LensDriverController, Optotune
            imgDisplayForm = new ImageDisplay();
            visualizerForm = new Visualizer();
            csuSetForm = new CSUSetting();

            // Hamamatsu's camera (from sample code)
            hpkCamForm = new csAcq4.FormMain();

            // z-position controller
            posCon = new PositionController();
            posConSetForm = new PositionControllerSetting();

            // 2D-projection
            imgProjForm = new ImageProjDisplay();

            opForm = new OperatorForm(
                ImgDisplay: imgDisplayForm, CameraSetting: camSetForm, VisForm:
visualizerForm,
                ETLSetting: etlSetForm, PosConSetForm: posConSetForm, imgProjForm
            );

            // Firstly Forms are constructed. Forms won't be Closed but Hided.
            form_list = new Form[] {
                spconsoleForm, opForm, imgDisplayForm, camSetForm, hpkCamForm,
                visualizerForm, posCon, etlSetForm, csuSetForm, posConSetForm,
imgProjForm };

            //
            posConSetForm.OPForm = opForm;
            imgProjForm.OPForm = opForm;
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            Point mainLocation = new Point();
            mainLocation.X = 600;
            mainLocation.Y = 100;
            this.Location = mainLocation;

            //checkBox_topmost_forms.Checked = true;
            checkBox_topmost_main.Checked = true;
        }
        private void MainForm_Shown(object sender, EventArgs e)
        {
            // Init the location of forms
            prev_location = new Point(this.Left, this.Top);

            // For debugging
            BeginInvoke(new Action(delegate
            {
                // *** Bottom of main form ***
```

```
                opForm.Show();
                opForm.Location = new Point(this.Location.X, this.Location.Y + this.Height);

                // *** Left Bottom to Main Form ***
                etlSetForm.Show();
                etlSetForm.Location = new Point(this.Location.X - etlSetForm.Size.Width,
this.Location.Y + this.Height);

                // *** Right Bottom to Main or OP form ***
                camSetForm.Show();
                if(this.Width > opForm.Width)
                    camSetForm.Location = new Point(this.Location.X + this.Size.Width,
this.Location.Y + this.Height);
                else
                    camSetForm.Location = new Point(this.Location.X + opForm.Size.Width,
this.Location.Y + this.Height);

                // Upper of CameraSetting
                csuSetForm.Show();
                csuSetForm.Location = new Point(camSetForm.Location.X, camSetForm.Location.Y
- csuSetForm.Height);

                // *** Right to CSU ***
                posConSetForm.Show();
                posConSetForm.Location = new Point(csuSetForm.Location.X + csuSetForm.Width,
csuSetForm.Location.Y);

                // *** Bottom of PosConSet form ***
                imgDisplayForm.Show();
                imgDisplayForm.Location = new Point(posConSetForm.Location.X,
posConSetForm.Location.Y + posConSetForm.Size.Height);

                // *** Right to PosControlSet ***
                posCon.Show();
                posCon.Location = new Point(posConSetForm.Location.X + posConSetForm.Width,
posConSetForm.Location.Y);

                // *** Right to PosControlSet ***
                imgProjForm.Show();
                imgProjForm.Location = new Point(posConSetForm.Location.X +
posConSetForm.Width, posConSetForm.Location.Y);

            }));
        }

        private void Button1_Click(object sender, EventArgs e)
        {
            byte[] test = { 0x50, 0x77, 0x44, 0x41, 0x07, 0xd0, 0x00, 0x00 };  // crc: 31 FD
            Optotune ot = new Optotune();
            byte[] res = ot.AddAndCheckCRC(test);
            Console.WriteLine(BitConverter.ToString(res));
        }

        private void HekpHToolStripMenuItem_Click(object sender, EventArgs e)
        {

        }

        private void ToolsToolStripMenuItem_Click(object sender, EventArgs e)
        {

        }

        private void SerialPortConsoleToolStripMenuItem_Click(object sender, EventArgs e)
        {
            spconsoleForm.Show();
            spconsoleForm.Activate();
        }
```

```
        private void CheckBox_fix_pos_forms_CheckedChanged(object sender, EventArgs e)
        {
            if (checkBox_fix_pos_forms.Checked)
            {
                prev_location.X = this.Left;
                prev_location.Y = this.Top;
            }
        }

        private void MainForm_LocationChanged(object sender, EventArgs e)
        {
            if (checkBox_fix_pos_forms.Checked)
            {
                Control c = (Control)sender;
                int sub_left = c.Left - prev_location.X;
                int sub_top = c.Top - prev_location.Y;

                foreach(var form in form_list)
                {
                    form.Left += sub_left;
                    form.Top += sub_top;
                }

                prev_location.X = c.Left;
                prev_location.Y = c.Top;
            }
        }


        private void MainForm_FormClosing(object sender, FormClosingEventArgs e)
        {
            // Check ETL/Camera status before closing MainForm. Hopefully safely uninit
that.
            // ETL
            opForm.LensDriver.Deinitialize();
            // Camera
            // TODO:::

            // *** TODO: Some checkpoint here ***

            // Close other forms before main form
            foreach(Form frm in form_list)
            {
                frm.Close();
            }
        }

        private void MainForm_MouseDown(object sender, MouseEventArgs e)
        {

        }


        private void MainForm_Activated(object sender, EventArgs e)
        {
        }

        private void checkBox_topmost_forms_CheckedChanged(object sender, EventArgs e)
        {
            bool chk = checkBox_topmost_forms.Checked;
            foreach (Form frm in form_list)
                frm.TopMost = chk;
            //this.TopMost = checkBox_topmost_main.Checked;
        }
        private void checkBox_topmost_main_CheckedChanged(object sender, EventArgs e)
        {
            this.TopMost = checkBox_topmost_main.Checked;
        }
```

```
        /// <summary>
        /// Change Checked state according to state of Forms
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void viewsStripMenuItem_DropDownOpening(object sender, EventArgs e)
        {
            operatorFormToolStripMenuItem.Checked = opForm.Visible;
            eTLToolStripMenuItem.Checked = etlSetForm.Visible;
            cameraSettingToolStripMenuItem.Checked = camSetForm.Visible;
            confocalScanningUnitToolStripMenuItem1.Checked = csuSetForm.Visible;
            imageDisplayToolStripMenuItem.Checked = imgDisplayForm.Visible;
            cameraToolStripMenuItem.Checked = hpkCamForm.Visible;  // unused
            positionControllerSettingToolStripMenuItem.Checked = posConSetForm.Visible;
        }

        #region ToolStripItems
        private void eTLToolStripMenuItem_Click(object sender, EventArgs e)
        {
            etlSetForm.Show();
            etlSetForm.Activate();
        }

        private void cameraToolStripMenuItem_Click(object sender, EventArgs e)
        {
            hpkCamForm.Show();
            hpkCamForm.Activate();
        }

        private void serialPortConsoleToolStripMenuItem1_Click(object sender, EventArgs e)
        {
            spconsoleForm.Show();
            spconsoleForm.Activate();
        }

        private void confocalScanningUnitToolStripMenuItem1_Click(object sender, EventArgs
e)
        {
            csuSetForm.Show();
            csuSetForm.Activate();
        }
        private void operatorFormToolStripMenuItem_Click(object sender, EventArgs e)
        {
            opForm.Show();
            opForm.Activate();
        }

        private void imageDisplayToolStripMenuItem_Click(object sender, EventArgs e)
        {
            imgDisplayForm.Show();
            imgDisplayForm.Activate();
        }

        private void cameraSettingToolStripMenuItem_Click_1(object sender, EventArgs e)
        {
            camSetForm.Show();
            camSetForm.Activate();
        }
        private void positionControllerSettingToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            posConSetForm.Show();
            posConSetForm.Activate();
        }

        private void quitToolStripMenuItem_Click(object sender, EventArgs e)
        {
            if (MessageBox.Show("Quit the application?", "", MessageBoxButtons.OKCancel) ==
DialogResult.OK)
```

```
                this.Close();
            else
                return;
        }
        #endregion

        private void StatusTimer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
        {
            // Get Memory and CPU usage
            process.Refresh();
            double mem_gb = (double)(process.WorkingSet64 / (Math.Pow(1024, 3)));
            double cpu_pct = cpuCounter.NextValue();

            // show cpu usage as text
            var keys = cpuUsageThreshold.Keys;
            int selectedKey = 100;
            foreach(var key in keys)
            {
                if (cpu_pct <= key)
                {
                    selectedKey = key;
                    break;
                }
            }
            string cpuString = cpuUsageThreshold[selectedKey];

            // color
            Color memColor = Color.Black;
            Color cpuColor = Color.Black;
            if (selectedKey == 100)
                cpuColor = Color.DarkRed;

            // show
            if (InvokeRequired)
            {
                Invoke(new Action(delegate
                {
                    label_memory_usage.Text = $"Memory Usage: {mem_gb:F1} [GiB]";
                    label_cpu_usage.Text = $"CPU Usage: {cpuString} [%]";

                    label_memory_usage.ForeColor = memColor;
                    label_cpu_usage.ForeColor = cpuColor;
                }));
            }
        }
        private void button_gc_Click(object sender, EventArgs e)
        {
            GC.Collect();
        }
    }

}
```

Code A8. Form/OperatorForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Diagnostics;
using System.Threading;
using System.IO;
using System.Drawing.Imaging;
using System.Windows.Media.Media3D;

using LensDriverController;
using LensDriverController.Logic;
using LensDriverController.Forms;
using LensDriverController.Service;
using LensDriverController.Settings;

using csAcq4;
using Hamamatsu.DCAM4;
using Hamamatsu.subacq4;
using NationalInstruments.Restricted;
using System.Runtime.InteropServices;
using BitMiracle.LibTiff.Classic;

namespace ETL_system
{
    public partial class OperatorForm : Form
    {
        #region Local Variables
        HighResolutionTimer gTimer, hTimerCamSoftwareTriggering, hTimerETL, hTimerLogging;
// based on Stopwatch class
        //Stopwatch htimer;
        List<MechLog> mechLogs;         // Mechanical movement of instruments to use for 4D
reconstruction
        ImageDisplay imgDisplayForm;    // A Form to show captured images in live
        CameraSetting camSettingForm;   // Camera parameters
        Visualizer visualizerForm;      // Visualize movement of instruments using
DataGridView table and line chart
        ETLSetting etlSettingForm;      // ETL parameters
        PositionControllerSetting posConSettingForm;  // FocusDriveMotor control
        ImageProjDisplay imgProjForm; // to show the projection (xz, yz) images made from
xy-t raw data in live

        // Optotune ETL  --------------------
        private LensDriver lensDriver;
        private LensDriver.OperationModes operationMode;

        private bool _sensorControl;
        private int _focalPower;
        private int _current;
        private int _maxFrequency;
        private int _frequency;
        private int _lowerSignalLevel;
        private int _upperSignalLevel;
        private int _desiredPosition;

        private bool _sensorControlState;
        private int _focalPowerState;
        private int _desiredPositionState;
        private int _currentState;
        private int _frequencyState;
```

```
        private int _lowerSignalLevelState;
        private int _upperSignalLevelState;

        private bool _showFirmwareUpgradeLabel = false;

        private System.Timers.Timer timerETLConnIndicator;

        // Hamamatsu Camera  -------------------
        private MyDcam mydcam;
        private MyDcamWait mydcamwait;
        private MyDcamRec mydcamrec;
        private DCAMImage dcam_image;
        public DCAMLut m_lut;
        public CamStatus camStatus;

        private int m_indexCamera = 0;      // index of DCAM device.  This is used when
allocating a mydcam instance.
        private int m_nFrameCount = 500;    // frame count of allocation buffer for DCAM
capturing
        //private FormStatus m_formstatus;    // Indicate current Form status. For setting,
Use MyFormStatus() function
        private Thread m_threadCapture;     // System.Threading.  Assigned for monitoring
updated frames during capturing
                                            // ADD: will be used to reflesh PictureBox in Live
mode
        private Bitmap m_bitmap;            // bitmap data for displaying in this Windows
Form

        /// <summary>
        /// The Tick that predicted to end exposure or something. Used by
SoftwareTriggering and FocusDriveMotor.
        /// </summary>
        private long TicksAllowZMove;
        /// <summary>
        /// Additional delay time [ms] for setting TicksAllowZMove. Mainly for capturing z-
stack.
        /// </summary>
        private int SafetyDelay_ms = 50;

        // Record Status of Instruments
        private RecordingLog recLog;

        // Live display form
        private Task taskUpdateLiveDisplay;
        private CancellationTokenSource cancelSourceUpdateLiveDisplay;

        // Live xz-, yz- projection
        private bool flagLiveProjection = false;
        public LiveImaging.ImageProjection.LiveProjectionBuffer liveProjectionBuffer { get;
set; }
        public LiveImaging.ImageProjection.XY_Projection xyProjection { get; set; }
        public int maxRowsLiveRecBuffer = 4096;
        public long tickLastFrame = 0;
        public int lastFrameNumber_Projection = 0;  // store last frame to be updated in
Projection-viewer
        public double lastFrame_MaxReal_Projection = 0;  // store z-position (max) of last
frame to be updated in Projection-viewer
        public bool canUpdateProjectionViewer = true;
        public const int MinimumWaitingCountForLiveProjection = 50;
        private System.Windows.Forms.Timer timerUpdateLiveProjection,
timerUpdateLiveProjectionViewer;

        // Focus changing plan for XYZ imaging
        private HighResolutionTimer hTimerFocusDriveMotor;
        private HighResolutionTimer hTimerETLIntermittent;
        private List<double> Z_MovePlan;
        private int Z_MovePlan_Index = 0;
        private bool flagSoftwareTriggered = false;
```

```
        HighResolutionTimer[] hTimers;

        // setup
        public enum ImagingMode
        {
            XYT,
            XYZ,
            XYZT,
        }

        public enum CamStatus
        {
            Startup,                // After startup or camapi_uninit()
            Initialized,            // After dcamapi_init() or dcamdev_close()
            Opened,                 // After dcamdev_open() or dcamcap_stop() without any
image
            Acquiring,              // After dcamcap_start()
            Acquired                // After dcamcap_stop() with image
        }

        private bool IsLive;
        #endregion

        #region Properties
        // Properties  ----------
        public LensDriver LensDriver { get => lensDriver; }
        public MyDcam DCAM { get => mydcam; set { mydcam = value; } }
        public Visualizer VisForm { get => visualizerForm; }

        /// <summary>
        /// HighRes Timer for recording timing of instruments globally.
        /// </summary>
        public HighResolutionTimer GlobalTimer { get => gTimer; }
        public RecordingLog RecLog { get => recLog; }
        public ETLCalibration eTLCalibration;
        public int AllocationFrameCount { get => m_nFrameCount; set
            {
                m_nFrameCount = value;
                if(camSettingForm != null)
                {
                    camSettingForm.numericUpDown_max_frames.Value = (decimal)m_nFrameCount;
                }
            } }

        public string WorkingDirectory { get; set; }
        public string WorkingFilePath { get; set; }
        public bool FlagLiveProjection { get => flagLiveProjection; set
{ flagLiveProjection = value; } }

        public bool EnableAllFrameOnMemoryTest { get; set; }
        #endregion

        // Constructor
        public OperatorForm(
            ImageDisplay ImgDisplay, CameraSetting CameraSetting, Visualizer VisForm,
            ETLSetting ETLSetting, PositionControllerSetting PosConSetForm,
            ImageProjDisplay ImgProj)
        {
            InitializeComponent();

            // Main timer to be used for sync
            gTimer = new HighResolutionTimer();
            gTimer.UseHighPriorityThread = true;
            recLog = new RecordingLog();  // Record behavior of instruments in session

            // Operate Camera & ETL behavior; e.g. Camera Triggering
            // OnTick events in these hTimers add each time-point (gTimer.Elapsed) to
RecordingLog.
            hTimerCamSoftwareTriggering = new HighResolutionTimer();
```

```
            hTimerCamSoftwareTriggering.UseHighPriorityThread = true;
            hTimerCamSoftwareTriggering.Elapsed += SendSoftwareTrigger_Tick;
            hTimerETL = new HighResolutionTimer();
            hTimerETL.UseHighPriorityThread = true;

            // For logging the movement of instruments (ETL)
            hTimerLogging = new HighResolutionTimer();
            hTimerLogging.UseHighPriorityThread = true;

            // XYZ imaging
            hTimerFocusDriveMotor = new HighResolutionTimer();
            hTimerFocusDriveMotor.UseHighPriorityThread = true;
            hTimerFocusDriveMotor.Elapsed += MoveZPos_FocusDriveMotor_Tick;

            hTimerETLIntermittent = new HighResolutionTimer();
            hTimerETLIntermittent.UseHighPriorityThread = true;
            hTimerETLIntermittent.Elapsed += MoveZPos_ETLCurrent_Tick;

            //htimer = new Stopwatch();
            mechLogs = new List<MechLog>();
            visualizerForm = VisForm;

            if (HighResolutionTimer.IsHighResolution)
                Console.WriteLine("Stopwatch class supports HighResolution: "+
HighResolutionTimer.TickLength*1000.0 + " [us]");
            else
                Console.WriteLine("Stopwatch class does NOT support HighResolution: "+
HighResolutionTimer.TickLength*1000.0 + " [us]");

            // *** Optotune ETL *** -------------------
            // ETLSetting Form
            etlSettingForm = ETLSetting;
            etlSettingForm.OPForm = this;
            lensDriver = etlSettingForm.InitLensDriver();
            eTLCalibration = etlSettingForm.eTLCalibration;

            // First call will try to establish connection
            timerETLConnIndicator = new System.Timers.Timer(interval: 1000); //[ms]
            timerETLConnIndicator.Elapsed += ETLConnIndicator_Tick;

            // *** Hamamatsu Camera *** ---------------------
            SetCamStatus(CamStatus.Startup);
            button_cam_stop.Enabled = false;
            dcam_image = new DCAMImage();

            // PicturePanelForm is constructed in MainForm.cs
            imgDisplayForm = ImgDisplay;
            imgDisplayForm.Location = new Point(this.Location.X, this.Location.Y +
imgDisplayForm.Size.Height);

            // CameraSetting Form
            camSettingForm = CameraSetting;
            camSettingForm.OPForm = this;

            // z-position
            posConSettingForm = PosConSetForm;
            Z_MovePlan = new List<double>();
            Z_MovePlan_Index = 0;

            // Live Projection *** ---------------------
            imgProjForm = ImgProj;
            //imgProjForm.Parent = this;
            timerUpdateLiveProjection = new System.Windows.Forms.Timer();
            timerUpdateLiveProjection.Tick += UpdateLiveProjectionBuffer_Tick;
            liveProjectionBuffer = new LiveImaging.ImageProjection.LiveProjectionBuffer();
            timerUpdateLiveProjectionViewer = new System.Windows.Forms.Timer();
            timerUpdateLiveProjectionViewer.Tick += UpdateLiveProjectionViewer_Tick;

            // To kill all HighResoTimers
```

```
            hTimers = new HighResolutionTimer[]
                {gTimer, hTimerETL, hTimerETLIntermittent, hTimerFocusDriveMotor,
hTimerLogging, hTimerCamSoftwareTriggering };

            WorkingDirectory = "C:¥¥Users¥¥MCB¥¥Documents¥¥";
        }
        private void OperatorForm_Load(object sender, EventArgs e)
        {
            // Establish connections .. ETL
            Task task = Task.Run(() =>
            {
                button_etl_conn_Click(button_etl_conn, EventArgs.Empty);
                etlSettingForm.PresetRestrictParams();
            });
            // Establish connections .. Camera
            Task task2 = Task.Run(() =>
            {
                Button_cam_conn_Click(button_cam_conn, EventArgs.Empty);
            });
            // Establish connections .. FocusDriveMotor (MAC6)
            //Task task3 = Task.Run(() =>
            //{
            //    posConSettingForm.button_connect_Click(null, EventArgs.Empty);
            //});
        }
        private void OperatorForm_Shown(object sender, EventArgs e)
        {
            // camera rotation
            camSettingForm.CameraRotationAndFlip();
        }
        private void OperatorForm_FormClosing(object sender, FormClosingEventArgs e)
        {
            // ETL
            lensDriver.Deinitialize();

            //SaveSession();
            //SaveSettings();
        }

        #region ETL Params Setting

        #endregion

        #region ETL Save and Load Settings (WIP)
        private void UpgradeSettings()
        {
            if (LensDriverSettings.Default.settingsUpgradeRequired)
            {
                LensDriverSettings.Default.Upgrade();
                LensDriverSettings.Default.settingsUpgradeRequired = false;
                LensDriverSettings.Default.Save();
            }

            if (CameraSettings.Default.settingsUpgradeRequired)
            {
                CameraSettings.Default.Upgrade();
                CameraSettings.Default.settingsUpgradeRequired = false;
                CameraSettings.Default.Save();
            }
        }
        private void LoadSettings()
        {
            _maxFrequency = LensDriverSettings.Default.maxFrequency;
            if (operationMode == LensDriver.OperationModes.FocalPower)
            {

lensDriver.UpdateSensorControlConfiguration(LensDriverSettings.Default.focalPowerCalibratio
nList);
            }
```

```
            else if (operationMode == LensDriver.OperationModes.Current)
            {

lensDriver.UpdateSensorControlConfiguration(LensDriverSettings.Default.currentCalibrationLi
st);
            }
            //UpdateGUI(false);
        }
        private void SaveSettings()
        {
            LensDriverSettings.Default.Save();

            //update state variables
            LoadSettings();
        }
        private void LoadSession()
        {
            _sensorControlState = LensDriverSettings.Default.sensorControl;
            _currentState = LensDriverSettings.Default.current;
            _focalPowerState = LensDriverSettings.Default.focalPower;
            _desiredPositionState = LensDriverSettings.Default.desiredPosition;
            _frequencyState = LensDriverSettings.Default.frequency;
            _lowerSignalLevelState = LensDriverSettings.Default.lowerCurrentSwing;
            _upperSignalLevelState = LensDriverSettings.Default.upperCurrentSwing;
        }
        private void SaveSession()
        {
            LensDriverSettings.Default.operationMode = operationMode.ToString();
            if (operationMode == LensDriver.OperationModes.Current)
            {
                LensDriverSettings.Default.sensorControl = _sensorControlState;
                LensDriverSettings.Default.current = _currentState;
            }
            else if (operationMode == LensDriver.OperationModes.FocalPower)
            {
                LensDriverSettings.Default.focalPower = _focalPowerState;
            }
            else if (operationMode == LensDriver.OperationModes.PositionControlled)
            {
                LensDriverSettings.Default.desiredPosition = _desiredPositionState;
            }
            else
            {
                LensDriverSettings.Default.frequency = _frequencyState;
                LensDriverSettings.Default.lowerCurrentSwing = _lowerSignalLevelState;
                LensDriverSettings.Default.upperCurrentSwing = _upperSignalLevelState;
            }

            LensDriverSettings.Default.Save();
            //  SaveCalibrationList(LensDriverSettings.Default.calibrationList);
            //update state variables
            LoadSession();
        }
        private void RestoreSession()
        {
            /*
            setCurrent(_currentState);
            setFocalPower(_focalPowerState);
            SetDesiredHall(_desiredPositionState);
            setUpperCurrentSwing(_upperSignalLevelState);
            setLowerCurrentSwing(_lowerSignalLevelState);
            setFrequency(_frequencyState);
            //*/
        }
        #endregion

        #region ETL Operation
        private void ETLConnIndicator_Tick(object sender, System.Timers.ElapsedEventArgs e)
        {
```

```
            //if (!this.IsHandleCreated) return;

            timerETLConnIndicator.Stop();

            bool state = lensDriver.lensConnectionEstablished;
            BeginInvoke(new Action(delegate
            {
                label_etl_conn = Utils.ConnectionLabelHelper(state, label_etl_conn);
                button_etl_conn.Text = Utils.ConnectionButtonHelper(state);

                label_etl_name.Text = lensDriver.LensSerialNumber;  // Lens Serial Number
has kept when disconnected
                if (label_etl_name.Text == String.Empty)
                    label_etl_name.Text = "lens name";

                etlSettingForm.SetConnectedDeviceName(lensDriver.ConnectedDeviceName);
                etlSettingForm.UpdateConnectionState(state);
            }));
        }
        private void button_etl_conn_Click(object sender, EventArgs e)
        {
            etlSettingForm.ETLSwitchConnection();
            timerETLConnIndicator.Start();
        }
        private void button_etl_op_Click(object sender, EventArgs e)
        {
            operationMode = LensDriver.OperationModes.Rectangular;
            lensDriver.SendSetOperationModeCommand(operationMode);
            etlSettingForm.setFrequency(frequency__: _frequency, encoded: true);
            etlSettingForm.setUpperCurrentSwing(current__: _upperSignalLevel, encoded:
true);
            etlSettingForm.setLowerCurrentSwing(current__: _lowerSignalLevel, encoded:
true);

            Console.WriteLine("freq " + lensDriver.signalFrequency + " = " +
lensDriver.firmwareManager.DecodeFrequency(lensDriver.signalFrequency)+" [Hz]");
            Console.WriteLine("upper " + lensDriver.signalUpperCurrentLevel + " = " +
lensDriver.firmwareManager.DecodeCurrent(lensDriver.signalUpperCurrentLevel)+" [mA]");
            Console.WriteLine("lower " + lensDriver.singalLowerCurrentLevel + " = " +
lensDriver.firmwareManager.DecodeCurrent(lensDriver.singalLowerCurrentLevel) + " [mA]");
        }
        private void button_settozero_Click(object sender, EventArgs e)
        {
            // Stop moving of lens
            // LensDriver.SetZeroCurrent()
            lensDriver.SendSetOperationModeCommand(LensDriver.OperationModes.Current, false,
false);
            lensDriver.SendSetCurrentCommand(0, useQueue:false, triggerEvent:false);
            lensDriver.SendSetFrequencyCommand(1, true);
            //etlSettingForm.setUpperCurrentSwing(current__: 0, encoded: false);
            //etlSettingForm.setLowerCurrentSwing(current__: 0, encoded: false);
        }
        private void button_mechmon_stop_Click(object sender, EventArgs e)
        {
            button_mechmon_start.Enabled = true;
            button_mechmon_stop.Enabled = false;

            MechMonitorStop();
        }
        private void ETLPositiontUpdate()
        {
        }
        private void ETLPositionUpdate_Tick(object sender,
HighResolutionTimerElapsedEventArgs e)
        {
            double current = 0.0;

            ETLPositiontUpdate();
            recLog.AddCurrent(gTimer.ElapsedTicks, current);
```

```
        }
        #endregion

        #region Monitoring Mechanical Instruments (Used for testing)
        private void button_mechmon_start_Click(object sender, EventArgs e)
        {
            // ETL has already begun to swing

            button_mechmon_start.Enabled = false;
            button_mechmon_stop.Enabled = true;

            MechMonitorInit();
            MechMonitorStart();
            // Camera should begin recording after timer
        }

        /// <summary>
        /// Monitoring the circumstance of instruments by StopWatch (or similar thing).
        /// </summary>
        private void MechMonitorInit()
        {
            mechLogs = new List<MechLog>();

            // set reasonable interval
            hTimerLogging.Interval = etlSettingForm.UpdatePositionInterval_ms; // [ms]; 33
ms cycle could not work. 50 ms seems OK, but is necessary to interpolate
            hTimerLogging.Elapsed += MechMonitorUpdate_Tick;
            hTimerLogging.Ready();
        }
        private void MechMonitorStart()
        {
            // DataAcquisition_Tick costs a large time (it take Status, Temperature,
Connection from ETL)
            lensDriver.StopDataAcquisition();

            visualizerForm.InitGraphs();
            visualizerForm.Show();
            hTimerLogging.Start();
        }
        private void MechMonitorStop()
        {
            if (!hTimerLogging.IsRunning) return;

            hTimerLogging.Stop();
            hTimerLogging.Elapsed -= MechMonitorUpdate_Tick;

            lensDriver.StartDataAcquisition();
        }
        private void MechMonitorUpdate_Tick(object sender,
HighResolutionTimerElapsedEventArgs e)
        {
            // TODO: Estimate time-cost of each communication with ETL and construction of
instance
            MechLog log = new MechLog();

            // ETL -- may be necessary to decode these raw values
            lensDriver.SendGetCurrentCommand();  // Get current value from ETL (?)
            log.Current = lensDriver.firmwareManager.DecodeCurrent(lensDriver.current);

            //lensDriver.SendGetActualLensPositionCommand();
            //log.ActualPosition = lensDriver.actualPosition;

            // TODO: Consider delay (e.Delay; see that HighResTimerEventArgs)
            log.Elapsed_ms = hTimerLogging.Elapsed_ms;

            mechLogs.Add(log);

            List<MechLog> cp_mechLogs;
            lock (mechLogs)
```

```
            {
                cp_mechLogs = new List<MechLog>(mechLogs);
            }
            visualizerForm.UpdateDataTable(cp_mechLogs);
        }
        private void Button_visualizer_Click(object sender, EventArgs e)
        {
            visualizerForm.Visible = !visualizerForm.Visible;
        }
        public void SetIntervalMechLogUpdate(float newInterval_ms)
        {
            hTimerLogging.Interval = newInterval_ms;
        }
        #endregion

        #region Camera Operation
        private void Button_cam_conn_Click(object sender, EventArgs e)
        {
            // Each function needs some time to complete

            // Startup camera
            if (camStatus == CamStatus.Startup)
            //if (button_cam_conn.Text == "Connect")  // using literal may not be reliable
            {
                if (camStatus == CamStatus.Startup)
                    DCAMInit();
                if (camStatus == CamStatus.Initialized)
                    DCAMOpen();
                if (camStatus == CamStatus.Opened)
                {
                    Invoke(new Action(delegate
                    {
                        Utils.ConnectionLabelHelper(true, label_cam_conn);
                        label_cam_name.Text = DCAMInfo_GetCamName();
                        button_cam_conn.Text = Utils.ConnectionButtonHelper(true);
                        label_cam_conn = Utils.ConnectionLabelHelper(true, label_cam_conn);

                        camSettingForm.SetHDCAMPtrToCaptureParams(this.DCAM.m_hdcam);
                        // send camera name to CameraSettingForm
                        camSettingForm.AddCameraName(label_cam_name.Text);
                    }));
                }
                else
                    label_cam_conn = Utils.ConnectionLabelHelper(false, label_cam_conn);
            }

            // Shutdown camera
            else if (camStatus == CamStatus.Opened || camStatus == CamStatus.Acquired)
            {
                DCAMClose();
                if (camStatus == CamStatus.Initialized)
                    DCAMUninit();
                if (camStatus == CamStatus.Startup)
                {
                    Invoke(new Action(delegate
                    {
                        Utils.ConnectionLabelHelper(false, label_cam_conn);
                        label_cam_name.Text = "camera name";
                        button_cam_conn.Text = Utils.ConnectionButtonHelper(false);
                        label_cam_conn = Utils.ConnectionLabelHelper(false, label_cam_conn);
                    }));
                }
                else
                {
                    label_cam_conn.Text = "*** wrong state ***";
                }
            }

            // Do nothing in case of other status
```

```
            else
                return;
        }
        private void Button_cam_op_Click(object sender, EventArgs e)
        {
            // Live
            mydcam.m_capmode = DCAMCAP_START.SNAP;
            //dcam.m_capmode = DCAMCAP_START.SEQUENCE;
            if( ! mydcam.cap_start() )
            {
                // when failed cap_start()
            }
        }
        private void Button_cam_live_Click(object sender, EventArgs e)
        {
            button_cam_stop.Enabled = true;
            button_cam_live.Enabled = false;
            button_cam_record.Enabled = false;

            //refer camSettingForm.CaptureParams to begin recording session

            // for stopping Updating Live Display
            cancelSourceUpdateLiveDisplay = new CancellationTokenSource();

            IsLive = true;

            // Live xz-, yz- projection  ----------
            bool flag = checkBox_with_live_projection.Checked;
            this.flagLiveProjection = flag;
            camSettingForm.flagLiveProjection = flag;
            camSettingForm.Live2DProjectionBuffer = this.liveProjectionBuffer;
            etlSettingForm.flagLiveProjection = flag;
            etlSettingForm.Live2DProjectionBuffer = this.liveProjectionBuffer;

            DCAMLive();
        }
        private void Button_cam_stop_Click(object sender, EventArgs e)
        {
            StopRecording();
        }
        private void StopRecording()
        {
            cancelSourceUpdateLiveDisplay.Cancel();

            FinalizeSession();
            DCAMIdle();
            ForceStopHighResolutionTimers(hTimers);

            // stop NIDAQ AO/AI if working
            camSettingForm.StopTriggering();
            etlSettingForm.button_abort_Click(this, EventArgs.Empty);

            // reset gTimer
            gTimer = new HighResolutionTimer();
            gTimer.UseHighPriorityThread = true;

            Invoke(new Action(delegate
            {
                button_cam_stop.Enabled = false;
                button_cam_live.Enabled = true;
                button_cam_record.Enabled = true;
            }));
        }

        private void button_cam_record_Click(object sender, EventArgs e)
        {
            if(InitializeSession() == false)
            {
                // failed to start recording
```

```
                return;
            }

            button_cam_stop.Enabled = false;
            button_cam_live.Enabled = false;
            button_cam_record.Enabled = true;

            // for stopping Updating Live Display
            cancelSourceUpdateLiveDisplay = new CancellationTokenSource();

            IsLive = checkBox_with_live.Checked;

            DCAMRecord();
        }
        private void button_fire_trigger_Click(object sender, EventArgs e)
        {
            DCAMSendSoftwareTrigger();
        }
        private void button_snap_Click(object sender, EventArgs e)
        {
            // Take snapshot from "PictureBox", not real data (Mar 2022)
            // --- 16 BIT GRAYSCALE IS NOT SUPPORTED IN GDI+ (System.Drawing) ---
            if (mydcam == null)
                return;

            int bitPerPixel = 16;

            string dir = WorkingDirectory;
            Directory.SetCurrentDirectory(dir);
            string filename = DateTime.Now.ToString("yyMMdd_HHmmss_")+ "snapshot";
            string ext = "tiff";
            string newFilePath = Utils.CreateNewFilePath(dir, filename + "." + ext, true);

            if (imgDisplayForm.OriginalImage != null)
            {
                var src = dcam_image.bufframe;
                int w = src.width;
                int h = src.height;

                int stride = w * (bitPerPixel / 8);

                byte[] dst_buf = new byte[stride * h];
                Marshal.Copy((IntPtr)(src.buf.ToInt64()), dst_buf, 0, src.rowbytes * h);
                //Marshal.Copy(dst_buf, 0, (IntPtr)(bmpData.Scan0), bmpData.Stride * h);

                //ImageCodecInfo codecInfo = Utils.GetEncoderInfo("image/tiff");
                //EncoderParameters encoderParams =
Utils.GetEncoderParametersToSaveTiff(EnableCompressionLZW: true);

                Tiff output = Tiff.Open(newFilePath, "w");
                output.SetField(TiffTag.IMAGEWIDTH, w);
                output.SetField(TiffTag.IMAGELENGTH, h);
                output.SetField(TiffTag.SAMPLESPERPIXEL, 1);
                output.SetField(TiffTag.BITSPERSAMPLE, bitPerPixel);
                output.SetField(TiffTag.ORIENTATION,
BitMiracle.LibTiff.Classic.Orientation.TOPLEFT);
                output.SetField(TiffTag.XRESOLUTION, 1);
                output.SetField(TiffTag.YRESOLUTION, 1);
                output.SetField(TiffTag.RESOLUTIONUNIT, ResUnit.NONE);
                output.SetField(TiffTag.PLANARCONFIG, PlanarConfig.CONTIG);
                output.SetField(TiffTag.PHOTOMETRIC, Photometric.MINISBLACK);
                output.SetField(TiffTag.COMPRESSION, Compression.LZW);
                output.SetField(TiffTag.FILLORDER, FillOrder.MSB2LSB);

                for (int y = 0; y < h; y++)
                {
                    int dst_offset = stride * y;
                    output.WriteScanline(dst_buf, dst_offset, y, plane: 0);
                }
```

```
                output.WriteDirectory();
            }
        }
        #endregion

        #region DCAM Basic Function (may be moved to another .cs file)
        private void DCAMInit()
        {
            // dcamapi_init() may takes for a few seconds
            Cursor.Current = Cursors.WaitCursor;

            if (!MyDcamApi.init())
            {
                CamShowStatusNG("dcamapi_init()", MyDcamApi.m_lasterr);
                Cursor.Current = Cursors.Default;
                return;                         // Fail: dcamapi_init()
            }

            // Success: dcamapi_init()

            CamShowStatusOK("dcamapi_init()");
            Cursor.Current = Cursors.Default;
            SetCamStatus(CamStatus.Initialized);
        }
        private void DCAMOpen()
        {
            if (mydcam != null)
            {
                CamShowStatus("Internal Error: mydcam is already set");
                //CamFormStatus_Initialized();     // FormStatus should be Initialized.
                return;                         // internal error
            }

            // dcamdev_open() may takes for a few seconds
            Cursor.Current = Cursors.WaitCursor;

            MyDcam aMyDcam = new MyDcam();
            if (!aMyDcam.dev_open(m_indexCamera))
            {
                CamShowStatusNG("dcamdev_open()", aMyDcam.m_lasterr);
                aMyDcam = null;
                Cursor.Current = Cursors.Default;
                return;                         // Fail: dcamdev_open()
            }

            // Success: dcamdev_open()

            mydcam = aMyDcam;                                       // store MyDcam instance

            camSettingForm.HDCAMPtr = mydcam.m_hdcam;  // copy to SettingForm

            CamShowStatusOK("dcamdev_open()");
            Cursor.Current = Cursors.Default;
            SetCamStatus(CamStatus.Opened);             // change dialog FormStatus to
Opened
        }
        private void DCAMClose()
        {
            if (mydcam == null)
            {
                CamShowStatus("Internal Error: mydcam is null");
                //MyFormStatus_Initialized();     // FormStatus should be Initialized.
                return;                         // internal error
            }

            //MyThreadCapture_Abort();           // abort capturing thread if exist

            if (!mydcam.dev_close())
            {
```

```
                CamShowStatusNG("dcamdev_close()", mydcam.m_lasterr);
                return;                         // Fail: dcamdev_close()
            }

            // Success: dcamdev_close()

            mydcam = null;

            CamShowStatusOK("dcamdev_close()");
            SetCamStatus(CamStatus.Initialized);        // change dialog FormStatus to
Initialized
        }
        private void DCAMUninit()
        {
            if (!MyDcamApi.uninit())
            {
                CamShowStatusNG("dcamapi_uninit()", MyDcamApi.m_lasterr);
                return;                         // Fail: dcamapi_uninit()
            }

            // Success: dcamapi_uninit()

            CamShowStatusOK("dcamapi_uninit()");
            SetCamStatus(CamStatus.Startup);            // change dialog FormStatus to
Startup
        }
        private void DCAMLive()
        {
            if (mydcam == null)
            {
                CamShowStatus("Internal Error: mydcam is null");
                //SetCamStatus(CamStatus.Initialized);     // FormStatus should be
Initialized.
                return;                         // internal error
            }

            // Apply capturing parameters
            camSettingForm.ApplyCameraPropeties(mydcam.m_hdcam);
            dcam_image = new DCAMImage();

            if (mydcam != null)
                mydcam.buf_release();

            // Live 2D Projection
            flagLiveProjection = checkBox_with_live_projection.Checked;
            if(flagLiveProjection)
            {
                UpdateETLCalibrationSelf();
                int bpp = camSettingForm.CaptureParams.BitDepth;
                int bin = camSettingForm.CaptureParams.Binning;
                int width = camSettingForm.CaptureParams.SubarraySize.Width / bin;
                int height = camSettingForm.CaptureParams.SubarraySize.Height / bin;

                var volumeSize = GetVolumeSizeForProjection();
                liveProjectionBuffer = new
LiveImaging.ImageProjection.LiveProjectionBuffer(bpp, width, height,
(int)Math.Ceiling(volumeSize.Z));
                liveProjectionBuffer.VoxelSize = imgProjForm.GetVoxelSizeSet();// new
Size3D(x: 1.0, y: 1.0, z: 1.0);

                // Initialize projection
                liveProjectionBuffer.xyProjection = new
LiveImaging.ImageProjection.XY_Projection(bpp, width, height);
                liveProjectionBuffer.AllocationFrameCount = this.AllocationFrameCount;
                timerUpdateLiveProjection.Interval = 40; // temporary
                timerUpdateLiveProjectionViewer.Interval = 76; // temporary
                // (int)camSettingForm.CaptureParams.ExposureTime_ms;
                liveProjectionBuffer.eTLCalibration = this.eTLCalibration;
```

```
                // z-range estimation is need to define the size of volume buffer
                double lowerCurrent = liveProjectionBuffer.eTLCalibration.CurrentMinBound;
                double upperCurrent = liveProjectionBuffer.eTLCalibration.CurrentMaxBound;
                //etlSettingForm.LensDriver.currentLowerSoftwareLimit =
(int)Math.Floor(lowerCurrent);
                //etlSettingForm.LensDriver.currentUpperSoftwareLimit =
(int)Math.Ceiling(upperCurrent);
                double objCoef = eTLCalibration.Coefficient;  // [um/mA]
                int estimatedZRangeReal = (int)Math.Ceiling(Math.Abs((upperCurrent -
lowerCurrent) * objCoef));
                double voxelZ = liveProjectionBuffer.VoxelSize.Z;
                // define the z-axis size of volume buffer using this value
                int estimatedZRangePx = (int)Math.Ceiling(estimatedZRangeReal / voxelZ);

                liveProjectionBuffer.VolumeSizePixel = new Size3D(x: width, y: height, z:
estimatedZRangePx);
                imgProjForm.VolumeSizePixel = liveProjectionBuffer.VolumeSizePixel;

                // Init 3D buffer
                liveProjectionBuffer.InitVolumeBuffer_UInt16(width, height,
estimatedZRangePx);
                liveProjectionBuffer.VoxelSize = imgProjForm.GetVoxelSizeSet();

                // Share Logging Buffer
                etlSettingForm.Live2DProjectionBuffer = liveProjectionBuffer;
                camSettingForm.Live2DProjectionBuffer = liveProjectionBuffer;
                imgProjForm.liveProjBuffer = liveProjectionBuffer;

                // Create Buffers in ProjDisplayForm
                imgProjForm.InitializeBuffers(width, height, estimatedZRangePx);

                // Adjust position of picturebox
                imgProjForm.AdjustControlSizeAndLocation();

                // temporary
                imgProjForm.liveProjBuffer.xyProjection.x0_selected = 100;
                imgProjForm.liveProjBuffer.xyProjection.y0_selected = 100;

                //
                imgProjForm.UpdateParamsView();
            }

            string text = "";

            if (camStatus == CamStatus.Opened || camStatus == CamStatus.Acquired)
            {
                // if FormStatus is Opened, DCAM buffer is not allocated.
                // So call dcambuf_alloc() to prepare capturing.

                text = string.Format("dcambuf_alloc({0})", m_nFrameCount);

                // allocate frame buffer
                if (!mydcam.buf_alloc(m_nFrameCount))
                {
                    // allocation was failed
                    CamShowStatusNG(text, mydcam.m_lasterr);
                    return;                     // Fail: dcambuf_alloc()
                }

                // Success: dcambuf_alloc()
                m_lut = imgDisplayForm.LUT;
                update_lut(false);
            }

            // start acquisition
            mydcam.m_capmode = DCAMCAP_START.SEQUENCE;    // continuous capturing.
continuous acqusition will be done
            if (!mydcam.cap_start())
            {
```

```
                // acquisition was failed. In this sample, frame buffer is also released.
                CamShowStatusNG("dcamcap_start()", mydcam.m_lasterr);

                mydcam.buf_release();           // release unnecessary buffer in DCAM
                SetCamStatus(CamStatus.Opened);         // change dialog FormStatus to
Opened
                return;                         // Fail: dcamcap_start()
            }

            // Success: dcamcap_start()
            // acquisition has started

            // Start the global HiRes timer
            gTimer.Start();

            // Start timer for Triggering if enabled SoftwareTrigger => 3
            //if (camSettingForm.CaptureParams.TriggerSource == 3
&& !hTimerCamSoftwareTriggering.IsRunning)
            //{
            //    hTimerCamSoftwareTriggering.Interval =
(float)(camSettingForm.CaptureParams.ExposureTime_ms);
            //    hTimerCamSoftwareTriggering.Start();
            //}

            if (text.Length > 0)
            {
                text += " && ";
            }
            CamShowStatusOK(text + "dcamcap_start()");

            SetCamStatus(CamStatus.Acquiring);          // change dialog FormStatus to
Acquiring
            //MyThreadCapture_Start();           // start monitoring thread
            imgDisplayForm.StartLiveTimer();

            // Start Live Projection
            if (flagLiveProjection)
            {
                imgProjForm.ApplyMagnificationParams();
                //imgProjForm.StartLiveTimer();
                lastFrameNumber_Projection = 0;
                lastFrame_MaxReal_Projection = 0;
                timerUpdateLiveProjection.Start();          // XY-Proj. Timer start here
                timerUpdateLiveProjectionViewer.Start();          // XY-Proj. Timer start
here
            }

            //CamCapture_Async();  // <-- only worked in xyt Live
            Task.Run(new Action(delegate
            {
                CamCapture_SyncTest();
            }));
        }
        /// <summary>
        /// OK
        /// </summary>
        private void DCAMRecord()
        {
            if (mydcam == null)
            {
                CamShowStatus("Internal Error: mydcam is null");
                //SetCamStatus(CamStatus.Initialized);     // FormStatus should be
Initialized.
                return;                                 // internal error
            }

            // Apply capturing parameters
            camSettingForm.ApplyCameraPropeties(mydcam.m_hdcam);
            dcam_image = new DCAMImage();
```

```
            if (mydcam != null)
                mydcam.buf_release();

            // Recording
            mydcamrec = new MyDcamRec();
            string dir = WorkingDirectory;
            Directory.SetCurrentDirectory(dir);
            string filename = "";
            string ext = "dcimg";
            string newFilePath = Utils.CreateNewFilePath(dir, filename + "." + ext, true);
            WorkingFilePath = newFilePath.Replace("."+ext, "");
            // TODO: Exception

            if (!mydcamrec.dcamrec_open(newFilePath, "", MaxFrameCountPerSession:
m_nFrameCount, IsMetaDataUsed__: true))
            {
                CamShowStatusNG("dcamrec_open()", mydcamrec.m_lasterr);
                return;                     // Fail: dcamrec_open()
            }

            // increment serial number of recording after image file created
            Utils.SaveLastRecordingNumber(dir, newFilePath);

            string text = "";
            if (camStatus == CamStatus.Opened || camStatus == CamStatus.Acquired)
            {
                // if FormStatus is Opened, DCAM buffer is not allocated.
                // So call dcambuf_alloc() to prepare capturing.

                text = string.Format("dcambuf_alloc({0})", m_nFrameCount);

                // allocate frame buffer
                if (!mydcam.buf_alloc(m_nFrameCount))
                {
                    // allocation was failed
                    CamShowStatusNG(text, mydcam.m_lasterr);
                    return;                     // Fail: dcambuf_alloc()
                }

                // Success: dcambuf_alloc()
                m_lut = imgDisplayForm.LUT;
                update_lut(false);
            }

            // start acquisition
            mydcam.cap_record(mydcamrec.recOpen);
            mydcam.m_capmode = DCAMCAP_START.SNAP;
            //mydcam.m_capmode = DCAMCAP_START.SEQUENCE;    // continuous capturing.
continuously acqusition will be done
            if (!mydcam.cap_start())
            {
                // acquisition was failed. In this sample, frame buffer is also released.
                CamShowStatusNG("dcamrec_start()", mydcam.m_lasterr);

                mydcam.buf_release();          // release unnecessary buffer in DCAM
                SetCamStatus(CamStatus.Opened);         // change dialog FormStatus to
Opened
                return;                     // Fail: dcamcap_start()
            }

            // Success: dcamcap_start()
            // acquisition has started

            // Log output
            if (text.Length > 0)
            {
                text += " && ";
            }
```

```
            CamShowStatusOK(text + "dcamrec_start()");

            SetCamStatus(CamStatus.Acquiring);          // change dialog FormStatus to
Acquiring

            // Start the global HiRes timer
            //if (!gTimer.IsRunning)
            if (!gTimer.IsRunning)
                gTimer.Start();

            switch(camSettingForm.CaptureParams.TriggerSource)
            {
                // TriggerSource == Software Trigger
                case 3:
                    if (!hTimerCamSoftwareTriggering.IsRunning)
                    {
                        //debug
                        //hTimerCamTriggering.Interval = 20;
                        //hTimerCamSoftwareTriggering.Interval =
(float)(camSettingForm.CaptureParams.ExposureTime_ms);
                        //hTimerCamSoftwareTriggering.Start();
                    }
                    break;

                // TriggerSource ==  External Trigger
                case 2:
                    // todo
                    break;
            }

            // Cancel token for updating LiveDisplay
            cancelSourceUpdateLiveDisplay = new CancellationTokenSource();
            if(checkBox_with_live.Checked)
                imgDisplayForm.StartLiveTimer();

            CamCapture_Async();
            //CamCapture_SyncTest();
        }
        private void DCAMIdle()
        {
            imgDisplayForm.StopLiveTimer();
            hTimerCamSoftwareTriggering.Stop();  // timer for SoftwareTriggering
            cancelSourceUpdateLiveDisplay.Cancel();

            hTimerETL.Stop();
            hTimerETLIntermittent.Stop();
            hTimerFocusDriveMotor.Stop();
            hTimerLogging.Stop();

            if(flagLiveProjection)
            {
                etlSettingForm.flagLiveProjection = false;
                camSettingForm.flagLiveProjection = false;
                this.flagLiveProjection = false;

                timerUpdateLiveProjection.Stop();
                timerUpdateLiveProjectionViewer.Stop();
                imgProjForm.StopLiveTimer();
                //object lck = new object();
                //lock(lck)
                //{
                //    imgProjForm.ClearBuffer();
                //    liveProjectionBuffer.Clear();
                //}
            }

            string text = "";

            if (mydcam == null)
```

```
            {
                CamShowStatus("Internal Error: mydcam is null");
                SetCamStatus(CamStatus.Initialized);     // FormStatus should be Initialized.
                return;                         // internal error
            }

            if (camStatus != CamStatus.Acquiring)
            {
                CamShowStatus("Internal Error: Idle button is only available when FormStatus
is Acquiring");
                return;                         // internal error
            }

            // stop recording
            if(mydcamrec!=null)
            {
                bool successed = mydcamrec.dcamrec_close();
                if (successed)
                    text = "dcamrec_close()";
                else
                    CamShowStatusNG("dcamrec_close()", mydcamrec.m_lasterr);
            }

            // stop acquisition
            if (!mydcam.cap_stop())
            {
                CamShowStatusNG("dcamcap_stop()", mydcam.m_lasterr);
                return;                         // Fail: dcamcap_stop()
            }

            if(text.Length > 0)
                text += " && ";

            // Success: dcamcap_stop()
            CamShowStatusOK(text+"dcamcap_stop()");
            //MyFormStatus_Acquired();           // change dialog FormStatus to Acquired
            SetCamStatus(CamStatus.Acquired);

            CamAbort();

            imgDisplayForm.NextImage = null;
        }


        // From FormInfo.cs
        private string DCAMInfo_GetCamName()
        {
            if (mydcam == null || camStatus != CamStatus.Opened)
                return "null";

            // get camera name to confirm the connection
            string series_name = mydcam.dev_getstring(DCAMIDSTR.CAMERA_SERIESNAME);     //
e.g. ORCA-Flash4.0 V3
            string model_name = mydcam.dev_getstring(DCAMIDSTR.MODEL);                  //
e.g. C13440-20C

            return series_name + "("+model_name+")";
        }

        /// <summary>
        /// Main function to pick the captured frames
        /// </summary>
        private async void CamCapture_Async()
        {
            bool bContinue = true;
            mydcamwait = new MyDcamWait(ref mydcam);  // using(mydcamwait = ...)
            {
                // TEST
                mydcamwait.m_timeout = 10 * 1000; // [ms]
```

```
                DCAMWAIT eventmask = DCAMWAIT.CAPEVENT.FRAMEREADY |
DCAMWAIT.CAPEVENT.STOPPED;
                DCAMWAIT eventhappened = DCAMWAIT.NONE;
                Task<(bool Successed, DCAMWAIT EventHappened)> camWaitTask =
mydcamwait.Start_Async(eventmask);

                int iFrameCountPrev = 0;  // TODO: Consider about this works "async". Frame
count will not be sequential
                while (bContinue && mydcam != null)
                {
                    //test
                    //Console.WriteLine($"{dcam_image.bufframe.iFrame} Good.");

                    //Console.WriteLine("DCAMWAIT task Successed:",
camWaitTask.Result.Successed.ToString());
                    if (camWaitTask.Result.Successed)
                    {
                        if (camWaitTask.Result.EventHappened & DCAMWAIT.CAPEVENT.FRAMEREADY)
                        {
                            // iNewestFrame => FrameNumber in the Buffer
                            // iFrameCount => Total FrameNumber.
                            // e.g. If FrameBuffer=200 and iFrameCount=300 then
iNewestFrame=100
                            int iNewestFrame = 0;
                            int iFrameCount = 0;

                            // TODO: Error occured after press Stop Button because mydcam was
null
                            if (mydcam.cap_transferinfo(ref iNewestFrame, ref iFrameCount))
                            {
                                if (iFrameCount > iFrameCountPrev)
                                {
                                    //MyUpdateImage(iNewestFrame);
                                    CameraUpdatePicture_Async(iNewestFrame);
                                    iFrameCountPrev = iFrameCount;
#if DEBUG
                                    //Console.WriteLine("iNewestFrame=" +
iNewestFrame.ToString() + "¥tiFrameCount = " + iFrameCount.ToString());
#endif
                                }
                            }
                        }

                        if (camWaitTask.Result.EventHappened & DCAMWAIT.CAPEVENT.STOPPED)
                            bContinue = false;
                    }
                    else
                    {
                        if (mydcamwait.m_lasterr == DCAMERR.TIMEOUT)
                        { }                             // nothing to do
                        else if (mydcamwait.m_lasterr == DCAMERR.ABORT)
                            bContinue = false;
                    }
                    await Task.Delay(1); // [ms]
                }
            }
        }
        private void CamCapture_SyncTest()
        {
            bool bContinue = true;
            using (mydcamwait = new MyDcamWait(ref mydcam))
            {
                mydcamwait.m_timeout = 10 * 1000; // [ms]
                DCAMWAIT eventmask = DCAMWAIT.CAPEVENT.FRAMEREADY |
DCAMWAIT.CAPEVENT.STOPPED;
                DCAMWAIT eventhappened = DCAMWAIT.NONE;
                //Task<(bool Successed, DCAMWAIT EventHappened)> camWaitTask =
mydcamwait.Start_Async(eventmask);
```

```
                bool successed = mydcamwait.start(eventmask, ref eventhappened);

                int iFrameCountPrev = 0;  // TODO: Consider this function is "async". Frame
count will not be sequential
                while (bContinue && mydcam != null)
                {
                    if (successed)
                    {
                        if (eventhappened & DCAMWAIT.CAPEVENT.FRAMEREADY)
                        {
                            // iNewestFrame => FrameNumber in the Buffer
                            // iFrameCount => Total FrameNumber.
                            // e.g. If FrameBuffer=200 and iFrameCount=300 then
iNewestFrame=100
                            int iNewestFrame = 0;
                            int iFrameCount = 0;

                            // TODO: Error occured after press Stop Button because mydcam was
null
                            if (mydcam.cap_transferinfo(ref iNewestFrame, ref iFrameCount))
                            {
                                if (iFrameCount > iFrameCountPrev)
                                {
                                    //MyUpdateImage(iNewestFrame);
                                    CameraUpdatePicture_Async(iNewestFrame);
                                    iFrameCountPrev = iFrameCount;
#if DEBUG
                                    //Console.WriteLine("iNewestFrame=" +
iNewestFrame.ToString() + "¥tiFrameCount = " + iFrameCount.ToString());
#endif
                                }
                            }
                        }

                        if (eventhappened & DCAMWAIT.CAPEVENT.STOPPED)
                            bContinue = false;
                    }
                    else
                    {
                        if (mydcamwait.m_lasterr == DCAMERR.TIMEOUT)
                        { }                         // nothing to do
                        else if (mydcamwait.m_lasterr == DCAMERR.ABORT)
                            bContinue = false;
                    }
                }
            }
        }

        private void CamAbort()
        {
            if (mydcamwait != null)
                mydcamwait.abort();
        }

        private void CameraUpdatePicture_Async(int iFrame)
        {
            // temporary luts
            //m_lut = picturePanel.LUT;
            //m_lut.inmax = 230;// HScrollLutMax.Value;
            //m_lut.inmin = 70;// HScrollLutMin.Value;

            //Task task = Task.Run(() =>
            taskUpdateLiveDisplay = Task.Run(() =>
            {
                // Can be aborted updating display using Stop Button
                if (cancelSourceUpdateLiveDisplay.Token.IsCancellationRequested)
                {
                    Console.WriteLine("Live display stopped. (CancelToken)");
                    imgDisplayForm.pictureBox.Image = null;
```

```
                    imgDisplayForm.ImageResolution = new Size(0, 0);
                    imgDisplayForm.NextImage = null;
                    imgDisplayForm.OriginalImage = null;
                    imgDisplayForm.OriginalImageBuffer = null;
                    return;
                }

                m_lut = imgDisplayForm.LUT;

                // lock selected frame by iFrame
                dcam_image.set_iFrame(iFrame);
                if (!mydcam.buf_lockframe(ref dcam_image.bufframe))
                {
                    // Fail: dcambuf_lockframe()
                    dcam_image.clear();
                }

                // Record Depth of Frames
                if (radioButton_capmode_xyz.Checked && radioButton_axial_mac6.Checked)
                {
                    // log[iFrame] = Depth
                    recLog.AddDepthOfFrame_Override(iFrame, posConSettingForm.Depth_micron);
                    recLog.AddDepthRawOfFrame(iFrame, posConSettingForm.Depth_Raw);
                }


                // Refresh PictureBox.Image of PicturePanel.cs (another Form) and/or this
Form
                if (dcam_image.isValid())
                {
                    Rectangle rc = new Rectangle(0, 0, dcam_image.width, dcam_image.height);
                    Bitmap m_bitmap_org = new Bitmap(dcam_image.width, dcam_image.height);
                    Bitmap m_bitmap = new Bitmap(dcam_image.width, dcam_image.height);

                    // get original, unscaled image
                    SUBACQERR err = subacq.copydib(ref m_bitmap_org, dcam_image.bufframe,
ref rc, m_lut.cameramax, 0);
                    if (err == SUBACQERR.SUCCESS)
                    {
                        var src = dcam_image.bufframe;
                        int w = src.width;
                        int h = src.height;

                        int stride = w * (m_lut.camerabpp / 8);

                        byte[] dst_buf = new byte[stride * h];
                        Marshal.Copy((IntPtr)(src.buf.ToInt64()), dst_buf, 0, src.rowbytes *
h);
                        // OriginalImageBuffer is used to show actual intensity on Live
display
                        imgDisplayForm.OriginalImageBuffer = dst_buf;

                        // OriginalImage (32bit-ARGB) is only for displaying, not to see the
actual intensity
                        imgDisplayForm.OriginalImage = (Image)m_bitmap_org;

                        // Live Projection: Copy current raw frame with frame number  -------
-------------
                        if (flagLiveProjection)
                        {
                            // RawFrameBuffer is Dictionary<iFrame, Image>
                            liveProjectionBuffer.RawFrameBuffer[iFrame] = dst_buf;
                        }

                        // Auto LUTs function
                        if (imgDisplayForm.checkBox_auto_luts.Checked)
                        {
                            (int min, int max) resultLuts =
Utils.GetMinMaxIntensityOfImage(src);
```

```
                        //m_lut.inmax = resultLuts.max;
                        //m_lut.inmin = resultLuts.min;
                        imgDisplayForm.SetLUTs(resultLuts.min, resultLuts.max);
                    }
                }

                // get scaled image
                err = subacq.copydib(ref m_bitmap, dcam_image.bufframe, ref rc,
m_lut.inmax, m_lut.inmin);

                if (err == SUBACQERR.SUCCESS)
                {
                    // Live Display
                    imgDisplayForm.NextImage = (Image)m_bitmap;
                    imgDisplayForm.UpdateFrameNumber(iFrame+1, m_nFrameCount);
                    imgDisplayForm.FlagImageUpdated = true;

                    // Live Projection
                    if(flagLiveProjection)
                    {
                    }
                }
                else
                {
                    imgDisplayForm.pictureBox.Image = null;
                    CamShowStatus(String.Format("NG: SUBACQERR: 0x{0:X8}", (int)err));
                }
            }
            else
            {
                Console.WriteLine("Live display stopped. (dcam_image is NOT valid)");
                imgDisplayForm.pictureBox.Image = null;
                imgDisplayForm.ImageResolution = new Size(0,0);
                imgDisplayForm.NextImage = null;
                imgDisplayForm.OriginalImage = null;
            }

            // finish recording
            if(iFrame >= m_nFrameCount -1)
            {
                //Invoke(new Action(delegate
                //{
                //Button_cam_stop_Click(button_cam_stop, EventArgs.Empty);
                //}));
                if (!IsLive)
                {
                    StopRecording();
                }
            }
        }, cancellationToken: cancelSourceUpdateLiveDisplay.Token);
    }
    private delegate void MyDelegate_UpdateImage(int iFrame);

    private void CameraUpdatePicture(int iFrame)
    {
        if (InvokeRequired)
        {
            // worker thread calls this function
            Invoke(new MyDelegate_UpdateImage(CameraUpdatePicture), iFrame);
            return;
        }

        if (iFrame > 0)
        {
            // lock selected frame by iFrame
            dcam_image.set_iFrame(iFrame);
            if (!mydcam.buf_lockframe(ref dcam_image.bufframe))
            {
                // Fail: dcambuf_lockframe()
```

```
                    dcam_image.clear();
                }
            }

            // Refresh PictureBox.Image of PicturePanel.cs Form
            if (dcam_image.isValid())
            {
                // temporary luts
                m_lut.inmax = 300;// HScrollLutMax.Value;
                m_lut.inmin = 0;// HScrollLutMin.Value;

                Rectangle rc = new Rectangle(0, 0, dcam_image.width, dcam_image.height);
                Bitmap m_bitmap = new Bitmap(dcam_image.width, dcam_image.height);
                SUBACQERR err = subacq.copydib(ref m_bitmap, dcam_image.bufframe, ref rc,
m_lut.inmax, m_lut.inmin);

                if (err == SUBACQERR.SUCCESS)
                {
                    imgDisplayForm.NextImage = (Image)m_bitmap;
                    //picturePanel.pictureBox.Image = m_bitmap;
                    //picturePanel.UpdateImage((Image)m_bitmap);
                    //pictureBox_cam.Image = picturePanel.pictureBox.Image;
                }
                else
                {
                    imgDisplayForm.pictureBox.Image = null;
                    //pictureBox_cam.Image = picturePanel.pictureBox.Image;
                    CamShowStatus(String.Format("NG: SUBACQERR: 0x{0:X8}", (int)err));
                }
            }
            else
            {
                imgDisplayForm.pictureBox.Image = null;
                //pictureBox_cam.Image = picturePanel.pictureBox.Image;
            }
        }
        // update LUT condition
        private void update_lut(bool bUpdatePicture)
        {
            //m_lut.inmax = 300;
            //m_lut.inmin = 80;
            //*
            if (mydcam != null)
            {
                MyDcamProp prop = new MyDcamProp(mydcam, DCAMIDPROP.BITSPERCHANNEL);

                double v = 0;
                prop.getvalue(ref v);
                m_lut.camerabpp = (int)v;
                m_lut.cameramax = (1 << m_lut.camerabpp) - 1;

                //m_lut.inmax = HScrollLutMax.Value;
                //m_lut.inmin = HScrollLutMin.Value;

                imgDisplayForm.numericUpDown_max.Maximum = m_lut.cameramax;
                imgDisplayForm.numericUpDown_min.Maximum = m_lut.cameramax;

                if (m_lut.inmax > m_lut.cameramax)
                {
                    m_lut.inmax = m_lut.cameramax;
                    imgDisplayForm.numericUpDown_max.Value = m_lut.inmax;
                    bUpdatePicture = true;
                }
                if (m_lut.inmin > m_lut.cameramax)
                {
                    m_lut.inmin = m_lut.cameramax;
                    imgDisplayForm.numericUpDown_min.Value = m_lut.inmin;
                    bUpdatePicture = true;
                }
```

```
                if (bUpdatePicture)
                    CameraUpdatePicture(-1);
            }
        //*/
        }
        /// <summary>
        /// Create and update projection image using Timer
        /// </summary>
        private void timerLive2DProjection()
        {


        }
        #endregion

        #region Camera Utils
        private void SetCamStatus(CamStatus status)
        {
            Boolean isStartup = (status == CamStatus.Startup);
            Boolean isInitialized = (status == CamStatus.Initialized);
            Boolean isOpened = (status == CamStatus.Opened);
            Boolean isAcquring = (status == CamStatus.Acquiring);
            Boolean isAcquired = (status == CamStatus.Acquired);

            /*
            PushInit.Enabled = isStartup;
            PushOpen.Enabled = isInitialized;
            PushInfo.Enabled = (isOpened || isAcquired || isAcquring);
            PushSnap.Enabled = (isOpened || isAcquired);
            PushLive.Enabled = (isOpened || isAcquired);
            PushFireTrigger.Enabled = isAcquring;
            PushIdle.Enabled = isAcquring;
            PushBufRelease.Enabled = isAcquired;
            PushClose.Enabled = (isOpened || isAcquired);
            PushUninit.Enabled = isInitialized;

            PushProperties.Enabled = (isOpened || isAcquired);
            //*/
            if (InvokeRequired)
            {
                Invoke(new Action(delegate
                {
                    button_cam_conn.Enabled = !isAcquring;
                    button_cam_live.Enabled = (isOpened || isAcquired);
                    button_cam_record.Enabled = (isOpened || isAcquired);
                    button_cam_stop.Enabled = isAcquring;
                }));
            }
            else
            {
                button_cam_conn.Enabled = !isAcquring;
                button_cam_live.Enabled = (isOpened || isAcquired);
                button_cam_record.Enabled = (isOpened || isAcquired);
                button_cam_stop.Enabled = isAcquring;
            }

            if (isInitialized || isOpened)
            {
                // acquisition is not starting
                //MyThreadCapture_Abort();
            }

            if (camStatus == status)
                return;
            else
                camStatus = status;

            // change label in CameraSetting.cs Form
            BeginInvoke(new Action(delegate
```

```
            {
                if(!camSettingForm.IsDisposed)
                    camSettingForm.SetCameraStatus(camStatus.ToString());  // use enum Key
string
            }));
        }

        private void radioButton_axial_scan_CheckedChanged(object sender, EventArgs e)
        {
            if(radioButton_axial_mac6.Checked || radioButton_axial_etl_intermittent.Checked)
            {
                radioButton_capmode_xyt.Checked = false;
                radioButton_capmode_xyz.Enabled = true;
                radioButton_capmode_xyzt.Enabled = false;

                // Using Level Trigger
                camSettingForm.Preset_XYZ_Intermittent();
            }
            else if(radioButton_axial_etl_continuous.Checked)
            {
                radioButton_capmode_xyt.Checked = false;
                radioButton_capmode_xyz.Enabled = true;
                radioButton_capmode_xyzt.Enabled = true;

                camSettingForm.Preset_XYZT_Continuous();
            }
            else if(radioButton_axial_none.Checked)
            {
                radioButton_capmode_xyt.Checked = true;
                radioButton_capmode_xyz.Enabled = false;
                radioButton_capmode_xyzt.Enabled = false;

                camSettingForm.Preset_XYT_Live();
            }
        }

        /// <summary>
        /// Display camera status
        /// </summary>
        /// <param name="text"></param>
        private void CamShowStatus(string text)
        {
            if(InvokeRequired)
            {
                Invoke(new Action(delegate
                {
                    textBox_cam_status.Text = text;
                }));
            }
            else
                textBox_cam_status.Text = text;
        }
        private void CamShowStatusOK(string text) { CamShowStatus("OK: " + text); }


        private void CamShowStatusNG(string text, DCAMERR err)
        {
            CamShowStatus(String.Format("NG: 0x{0:X8}:{1}", (int)err, text));
        }
        #endregion

        #region Live Projection
        /// <summary>
        /// Update ImageProjDisplay Form using timer
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void UpdateLiveProjectionBuffer_Tick(object sender, EventArgs e)
        {
```

```
            if (timerUpdateLiveProjection.Interval != imgProjForm.UpdateIntervalShouldBe_ms)
            {
                timerUpdateLiveProjection.Interval = imgProjForm.UpdateIntervalShouldBe_ms;
                timerUpdateLiveProjectionViewer.Interval =
(int)(imgProjForm.UpdateIntervalShouldBe_ms * 1.9);  // randomly set
            }

            if (liveProjectionBuffer.Count == 0)
                return;

            // Get the latest frames' z-pitch (min/max depth of frame)
            // code
            List<LiveImaging.ImageProjection.ZPitchRow> zPitches =
liveProjectionBuffer.GetZPitchFromInstRecording();
            if (zPitches.Count == 0)
                return;

            foreach (var zp in zPitches)
            {
                // Get the corresponding number of frames (raw byte[] xy-image) from DCAM
buffer
                // or Simply count-up from zero, one-by-one?
                lastFrameNumber_Projection = (int)zp.FrameNumber;  // something

                // To solve the delays of updating of projection-view,
                // Skip update if the waiting-frames >= frame-per-volume, means it is one or
more laps behind
                int newestFrameNumber = dcam_image.bufframe.iFrame;
                double fxy = 1000 / camSettingForm.CaptureParams.ExposureTime_ms;  // [fps]
                double fz = (double)etlSettingForm.numericUpDown_frequency.Value;  //
[Hz/volume]
                double framePerVolume = fxy / fz;  // [fpv]
                int delayedConditionFrameCount = 2 * (int)framePerVolume;
                if (delayedConditionFrameCount < MinimumWaitingCountForLiveProjection)
                    delayedConditionFrameCount = MinimumWaitingCountForLiveProjection;

                int waitingFrameCount = newestFrameNumber - lastFrameNumber_Projection;
                if (newestFrameNumber < lastFrameNumber_Projection)
                {
                    int maxFrames = (int)(camSettingForm.numericUpDown_max_frames.Value);
                    waitingFrameCount = (maxFrames - lastFrameNumber_Projection) +
newestFrameNumber;
                }
                if (waitingFrameCount >= delayedConditionFrameCount)
                {
                    canUpdateProjectionViewer = false;
                    continue;
                }
                else
                    canUpdateProjectionViewer = true;

                if (zp.MinReal >= 0)
                {
                    // Update volumeBuffer
                    imgProjForm.liveProjBuffer.UpdateVolumeBufferAt((int)zp.FrameNumber, new
double[] { zp.MinReal, zp.MaxReal });

                    // Update display

imgProjForm.UpdateImageBuffer(liveProjectionBuffer.RawFrameBuffer[(int)zp.FrameNumber],
zp);
                }
            }
            lastFrame_MaxReal_Projection = zPitches[zPitches.Count - 1].MaxReal;
            //canUpdateProjectionViewer = true;
        }
        /// <summary>
        /// [WIP] Update projection viewer by loading the images from volume buffer. This
function does not change the volume buffer.
```

```
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void UpdateLiveProjectionViewer_Tick(object sender, EventArgs e)
        {
            if (!canUpdateProjectionViewer
|| !liveProjectionBuffer.RawFrameBuffer.ContainsKey(lastFrameNumber_Projection))
                return;

            var lastFrame = liveProjectionBuffer.RawFrameBuffer[lastFrameNumber_Projection];
            imgProjForm.UpdateViewer(lastFrame, lastFrame_MaxReal_Projection,
                (int)imgProjForm.VolumeSizePixel.X, (int)imgProjForm.VolumeSizePixel.Y);

            if (InvokeRequired)
                BeginInvoke(new Action(delegate
                {
                    imgProjForm.label_frameNumber.Text = $"{lastFrameNumber_Projection} vs
{dcam_image.bufframe.iFrame}";
                }));
            else
                imgProjForm.label_frameNumber.Text = $"{lastFrameNumber_Projection} vs
{dcam_image.bufframe.iFrame}";
        }
        #endregion

        #region Misc.
        public void ChangeDisplayFlipType(int rotationAngle, bool flipX, bool flipY)
        {
            // create info text
            string s = "";
            if (flipX) s += "X";
            if (flipY) s += "Y";
            if (!flipX && !flipY) s = "None";
            imgDisplayForm.RotationLabel = "Rotation " + rotationAngle + " deg & Flip " + s;

            RotateFlipType type = Utils.GetRotateFlipType(rotationAngle, flipX, flipY);
            imgDisplayForm.RotationFlipType = type;

            // for Projection form
            if(imgProjForm != null)
            {
                imgProjForm.TransformDirectionsRelatedToStageHelper(type);
            }
        }
        public Size3D GetVolumeSizeForProjection()
        {
            Size sizeXY = camSettingForm.CaptureParams.SubarraySize;
            double subCurrent = (double)(etlSettingForm.numericUpDown_upper_current.Value -
etlSettingForm.numericUpDown_lower_current.Value);
            double depth = Math.Abs(etlSettingForm.eTLCalibration.Coefficient * subCurrent);
            Size3D result = new Size3D(sizeXY.Width, sizeXY.Height, depth);
            return result;
        }
        #endregion

        #region Initialize & Finalize Recording session
        /// <summary>
        /// Before recording session
        /// </summary>
        public bool InitializeSession()
        {
            recLog.Clear();

            // Initialize Map for transforming ETL Voltage to Current
            eTLCalibration.CurrentMinBound =
(double)etlSettingForm.numericUpDown_lower_current.Value;
            eTLCalibration.CurrentMaxBound =
(double)etlSettingForm.numericUpDown_upper_current.Value;
            eTLCalibration.CreateMapVoltageToCurrent();
```

```
            // Apply Camera Props
            camSettingForm.ApplyCameraPropeties(HDCAMPtr: mydcam.m_hdcam);

            // set new file name (this initialization runs before DCAMRecord())
            string newFilePath = Utils.CreateNewFilePath(WorkingDirectory, "", true);
            newFilePath = Path.ChangeExtension(newFilePath, null);

            // Output variables of setup
            recLog.WriteOutMetadata(newFilePath + "metadata.txt",
                this, etlSettingForm, camSettingForm, posConSettingForm);

            // 2D time-lapse, XYT
            if (radioButton_capmode_xyt.Checked)
            {
            }

            // Conventional 3D, XYZ
            else if (radioButton_capmode_xyz.Checked)
            {
                if (radioButton_axial_mac6.Checked)
                {
                    // check connection
                    if (!posConSettingForm.MAC6_Instance.IsOpen)
                    {
                        MessageBox.Show("FocusDriveMotor (MAC6000) has NOT connected.",
"Error");
                        return false;
                    }

                    // FocusDriveMotor: interval of updating z-position
                    posConSettingForm.ChangeTimerInterval(Interval_ms: 25);
                }
                else if (radioButton_axial_etl_intermittent.Checked)
                {
                    // ETL Control (DAQ AI - [Ticks, Voltage])

                    // Translate Voltage to Current and/or Focal Power of ETL

                }
            }

            // 4D Volumetric, XYZT
            else if (radioButton_capmode_xyzt.Checked)
            {
                if (radioButton_axial_etl_continuous.Checked)
                {
                }
            }

            return true;
        }

        /// <summary>
        /// After recording session, gather & output data from instruments.
        /// </summary>
        public void FinalizeSession()
        {
            //-------------------------------------------------------------------
            // Gather all Rec-Logs to merge & output.

            if (radioButton_axial_etl_continuous.Checked)
            {
                // Reduce Excess zero-volt external trigger data
                // necessary to invoke when taking z-stack
                Invoke(new Action(delegate
                {
                    if (camSettingForm.comboBox_trigger_source.Text == "External"
                    && camSettingForm.checkBox_external_trigger_by_nidaq.Checked
```

```
                    && camSettingForm.comboBox_trigger_active.Text == "SyncReadout")
                    {
                        recLog.ReduceExcessTriggerRecord(timeResolution_ms: 5);  //
experimental, but effective
                        recLog.SetCameraStatesFromExternalTriggerRecord();
                    }
                }));
            }

            // Transform ETL Voltage to Current, and/or Current to Depth
            if (etlSettingForm.checkBox_use_calibration_info.Checked)
            {
                UpdateETLCalibrationSelf();

                if (recLog.VoltageTiming.Count > 0)
                {
                    double current;
                    foreach (var keyTick in recLog.VoltageTiming.Keys)
                    {
                        current =
eTLCalibration.GetCurrentFromVoltage(recLog.VoltageTiming[keyTick]);
                        recLog.AddCurrent(keyTick, current);
                    }
                }
                if(recLog.CurrentTiming.Count > 0)
                {
                    double depth;
                    foreach (var keyTick in recLog.CurrentTiming.Keys)
                    {
                        depth =
eTLCalibration.GetDepthFromCurrent(recLog.CurrentTiming[keyTick]);
                        recLog.AddDepthTiming(keyTick, depth);
                    }
                }
            }

            // output all data
            recLog.WriteOutAllData(WorkingFilePath + "all.csv");
            if (recLog.DepthOfFrame.Count > 0)
                recLog.WriteOutDepthOfFrame(WorkingFilePath + "depth_of_frame.csv");

            // 2D time-lapse, XYT
            if (radioButton_capmode_xyt.Checked)
            {
            }

            // Conventional 3D, XYZ
            else if (radioButton_capmode_xyz.Checked)
            {
                if (radioButton_axial_mac6.Checked)
                {
                    // FocusDriveMotor log => Depth of each frame

                    // Reset the state of PositionControlloerSettingForm (MAC6)
                    if (posConSettingForm.IsOperating)
                    {
                        posConSettingForm.IsOperating = false;
                        if (posConSettingForm.IsShowingRealDistance)
                        {
                            if (posConSettingForm.InvokeRequired)
                                Invoke(new Action(delegate
                                {
                                    posConSettingForm.numericUpDown_z_pos.Value =
(decimal)posConSettingForm.MAC6_Instance.Z_RealPositionMicron;
                                }));
                            else
                                posConSettingForm.numericUpDown_z_pos.Value =
(decimal)posConSettingForm.MAC6_Instance.Z_RealPositionMicron;
                        }
```

```
                    else
                    {
                        if (posConSettingForm.InvokeRequired)
                            Invoke(new Action(delegate
                            {
                                posConSettingForm.numericUpDown_z_pos.Value =
(decimal)posConSettingForm.MAC6_Instance.Z_ReadValue;
                            }));
                        else
                            posConSettingForm.numericUpDown_z_pos.Value =
(decimal)posConSettingForm.MAC6_Instance.Z_ReadValue;
                    }
                }
                posConSettingForm.MAC6_Instance.SendHalt();  // test
            }
            else if(radioButton_axial_etl_intermittent.Checked)
            {
                // ETL Control (USB driver - [Ticks, Current])
            }
        }

        // 4D Volumetric, XYZT
        else if (radioButton_capmode_xyzt.Checked)
        {
            if (radioButton_axial_etl_continuous.Checked)
            {
            }
        }
    }

    private void UpdateETLCalibrationSelf()
    {
        eTLCalibration.Coefficient =
(double)etlSettingForm.numericUpDown_calib_coef.Value;
        eTLCalibration.Constant =
(double)etlSettingForm.numericUpDown_calib_const.Value;
        eTLCalibration.CurrentMinBound =
(double)etlSettingForm.numericUpDown_lower_current.Value;
        eTLCalibration.CurrentMaxBound =
(double)etlSettingForm.numericUpDown_upper_current.Value;
        eTLCalibration.CreateMapVoltageToCurrent();
    }
    #endregion

    #region Software Triggering
    private void DCAMSendSoftwareTrigger()
    {
        Console.WriteLine($"===== fire    {gTimer.ElapsedTicks * 1e-4:F1} ms");
        if (!mydcam.cap_firetrigger())
        {
            if (mydcam != null)
                Console.WriteLine("Failed FireTrigger (mydcam is null).");
            else
                Console.WriteLine("Failed FireTrigger.");
        }
        Console.WriteLine($"===== fireddd {gTimer.ElapsedTicks * 1e-4:F1} ms");

        double current = lensDriver.firmwareManager.DecodeCurrent(lensDriver.current);
        //Console.WriteLine("GetCurrent @SendTrigger = " + current + " mA (Before
SendGetCurrentCommand())");
        //lensDriver.SendGetCurrentCommand();
        //current = lensDriver.firmwareManager.DecodeCurrent(lensDriver.current);
        Console.WriteLine("GetCurrent @SendTrigger = " + current + " mA");
        flagSoftwareTriggered = true;
    }
    private void SendSoftwareTrigger_Tick(object sender,
HighResolutionTimerElapsedEventArgs e)
    {
        // XYZ capturing by FocusDriveMotor
```

```
            // when Motor is moving, skip SoftwareTrigger
            if (radioButton_axial_mac6.Checked && radioButton_capmode_xyz.Checked)
            {
                if (posConSettingForm.MAC6_Instance.IsMoving)
                {
                    return;
                }
                else
                {
                    recLog.AddDepthOfFrame(recLog.DepthOfFrame.Count,
posConSettingForm.Depth_micron);
                    recLog.AddDepthRawOfFrame(recLog.DepthRawOfFrame.Count,
posConSettingForm.Depth_Raw);
                    //Console.WriteLine($"Ended Moving {gTimer.Elapsed_ms / 1000:F1} sec");
                }
            }
            // XYZ with ETL
            else if (radioButton_axial_etl_intermittent.Checked &&
radioButton_capmode_xyz.Checked)
            {
                // ETL movement is faster than FocusDriveMotor.
                // But introducing something to check like "IsMoving" will make the system
more reliable. (TODO)

                //if (ETLwasStopped)
                {
                    // Get ETL current when SoftwareTriggering
                    double currentDecoded =

etlSettingForm.LensDriver.firmwareManager.DecodeCurrent(etlSettingForm.LensDriver.current);
                    recLog.AddCurrent(gTimer.ElapsedTicks, currentDecoded);  // Not cofirmed
this "current" has really taken from ETL
#if DEBUG
                    Console.WriteLine(string.Format("ETL Current = {0} mA",
currentDecoded));
#endif
                }
            }

            // XYZT 4D capturing (ETL)
            else if (radioButton_axial_etl_continuous.Checked &&
radioButton_capmode_xyzt.Checked)
            {
                // We calculate physical z-depth using voltage and/or current AFTER END OF
RECORDING SESSION, not here
                // because voltage data are handled as chunk in NIDAQ.
            }

            var nextState = RecordingLog.TimingEnum.FrameBegin;
            double prep = gTimer.ElapsedTicks * 1e-4;
            Console.WriteLine($"===== Prepared Trigger: {prep:F1} ms
[SendSoftwareTrigger_Tick]");

            DCAMSendSoftwareTrigger();
            long elapsed = gTimer.ElapsedTicks;
            Console.WriteLine($"===== SentTrigger: {elapsed * 1e-7 * 1e3:F1} ms
[SendSoftwareTrigger_Tick]");
            recLog.AddCamTiming(elapsed, nextState);

            // 1 [sec] <=> 1e7 [ticks]. This will be used with FocusDriveMotor
            if(radioButton_capmode_xyz.Checked)
                TicksAllowZMove = elapsed +
(int)((camSettingForm.CaptureParams.ExposureTime_ms + SafetyDelay_ms) * 1e-4);

            // debug
            int cnt = recLog.CamCapturingTiming.Count;
            if (cnt % 100 == 0)
            {
                Console.WriteLine($"---------- CamTriggeredTiming[{cnt - 1}] @ {elapsed *
```

```
1e-7:F1} sec [SendSoftwareTrigger_Tick]");

                //testing
                long a = recLog.CamCapturingTiming.ElementAt(cnt - 1).Key;
                long b = recLog.CamCapturingTiming.ElementAt(0).Key;
                double totalElapsedSec = (recLog.CamCapturingTiming.ElementAt(cnt-1).Key -
recLog.CamCapturingTiming.ElementAt(0).Key) * 1e-7;
                Console.WriteLine($"total elapsed sec: {totalElapsedSec:F1} ;
{totalElapsedSec/cnt * 1e3:F2} [ms/frame]");

                if (cnt >= m_nFrameCount - 1)
                {
                    string file = Path.Combine(Directory.GetCurrentDirectory(),
WorkingFilePath + "cam_timing.csv");
                    recLog.WriteOutCamTiming(file);
                }
            }
        }
        #endregion

        private void radioButton_capmode_xyt_CheckedChanged(object sender, EventArgs e)
        {
        }

        private void checkBox_with_live_projection_CheckedChanged(object sender, EventArgs
e)
        {
            button_cam_record.Enabled = !checkBox_with_live_projection.Checked;
        }

        private void button_begin_trigger_Click(object sender, EventArgs e)
        {
            if (radioButton_capmode_xyz.Checked)
            {
                if (radioButton_axial_mac6.Checked)
                {
                    Z_MovePlan = posConSettingForm.CreateMovePlan();
                    Z_MovePlan_Index = 0;

                    posConSettingForm.IsOperating = true;
                    // change timer setting

                    hTimerCamSoftwareTriggering.Interval = 200;
                    hTimerFocusDriveMotor.Interval = 100;

                    gTimer.Start();
                    hTimerCamSoftwareTriggering.Start();
                    hTimerFocusDriveMotor.Start();
                }
                else if(radioButton_axial_etl_intermittent.Checked)
                {
                    etlSettingForm.checkBox_analogsignal.Checked = false;
                    etlSettingForm.comboBox_waveform.SelectedIndex = 0;  // 0 means "Current"

//etlSettingForm.LensDriver.SendSetOperationModeCommand(LensDriver.OperationModes.Current);
                    etlSettingForm.button_apply_Click(null, EventArgs.Empty);

                    Z_MovePlan = etlSettingForm.CreateMovePlanCurrent(
                        TotalDivided:
(int)etlSettingForm.numericUpDown_xyz_partition_number.Value,
                        LowerToUpper: etlSettingForm.checkBox_xyz_lower_to_upper.Checked);
                    Z_MovePlan_Index = 0;

                    hTimerCamSoftwareTriggering.Interval = 1000;
                    hTimerETLIntermittent.Interval = 200;

                    etlSettingForm.LensDriver.SendGetCurrentCommand();

                    gTimer.Start();
```

```
                    hTimerCamSoftwareTriggering.Start();
                    hTimerETLIntermittent.Start();
                }
            }

            else if(radioButton_capmode_xyzt.Checked)
            {
                if (radioButton_axial_etl_continuous.Checked)
                {
                    etlSettingForm.checkBox_analogsignal.Checked = true;
                    //etlSettingForm.comboBox_waveform.SelectedItem = "Analog";
                    etlSettingForm.button_apply_Click(null, EventArgs.Empty);

                    // Use AnalogOutput Voltage for Camera Triggering
                    if (camSettingForm.checkBox_external_trigger_by_nidaq.Checked)
                    {
                        // TODO: code
                    }
                    // Use Timer control
                    else
                    {
                        hTimerCamSoftwareTriggering.Interval =
(int)camSettingForm.CaptureParams.ExposureTime_ms;
                        hTimerCamSoftwareTriggering.Start();
                    }

                    gTimer.Start();
                }
            }
        }

        private void MoveZPos_FocusDriveMotor_Tick(object sender,
HighResolutionTimerElapsedEventArgs e)
        {
            //Console.WriteLine($"¥t¥t¥tFDM Tick: {gTimer.Elapsed_ms / 1000:F1}");

            // Before the end of exposure, keep z-position
            if (camStatus != CamStatus.Acquiring || gTimer.ElapsedTicks <= TicksAllowZMove)
                return;

            // Move z-position to next step
            if (Z_MovePlan_Index < Z_MovePlan.Count &&
posConSettingForm.MAC6_Instance.IsMoving==false)
            {
                double pos = Z_MovePlan[Z_MovePlan_Index];
                posConSettingForm.MAC6_Instance.SendMovePosition_ZReal(pos, true);
                toolStripStatusLabel1.Text = string.Format("Capturing at Z={0:f1} µm,
Index={1}/{2}", pos.ToString(), Z_MovePlan_Index, Z_MovePlan.Count-1);
                Z_MovePlan_Index++;

                TicksAllowZMove = gTimer.ElapsedTicks + (int)(5 * 1e7);  // Temporarily add
5 sec for waiting.
            }
            else if (Z_MovePlan_Index > 0 && Z_MovePlan_Index < Z_MovePlan.Count
                && posConSettingForm.MAC6_Instance.IsMoving && gTimer.ElapsedTicks >
TicksAllowZMove)
            {
                // avoid stuck

posConSettingForm.MAC6_Instance.SendMovePosition_ZVal(posConSettingForm.MAC6_Instance.Z_Tar
getValue);
                Console.WriteLine("### ZMove TimeOut [Stuck]");

                TicksAllowZMove = gTimer.ElapsedTicks + (int)(5 * 1e7);  // Temporarily add
5 sec for waiting.
            }
            else if (Z_MovePlan_Index >= Z_MovePlan.Count)
            {
                // finish recording
```

```
                hTimerCamSoftwareTriggering.Stop();
                hTimerFocusDriveMotor.Stop();
                //Invoke(new Action(delegate
                //{
                    //Button_cam_stop_Click(button_cam_stop, EventArgs.Empty);
                //}));
                StopRecording();
                toolStripStatusLabel1.Text = string.Format("Finished. Z={0:f1} µm,
Index={1}/{2}", Z_MovePlan[Z_MovePlan.Count - 1].ToString(), Z_MovePlan_Index,
Z_MovePlan.Count - 1);
            }
        }

        private void MoveZPos_ETLCurrent_Tick(object sender,
HighResolutionTimerElapsedEventArgs e)
        {
            // Before the end of exposure, keep z-position
            if (camStatus != CamStatus.Acquiring || gTimer.ElapsedTicks <= TicksAllowZMove)
                return;

            // Move z-position to next step
            if (Z_MovePlan_Index < Z_MovePlan.Count ) // todo: get ETL current using USB
driver command.. && etlSettingForm.IsMoving == false)
            {
                double current = Z_MovePlan[Z_MovePlan_Index];
                if (flagSoftwareTriggered && gTimer.ElapsedTicks > TicksAllowZMove)
                {
                    //Invoke(new Action(() =>
                    //{
                    //    etlSettingForm.numericUpDown_current.Value = (decimal)current;
                    //    etlSettingForm.button_apply_Click(null, EventArgs.Empty);
                    //}));
                    etlSettingForm.setCurrent(current, setStateVariable: true, encoded:
false);
                    string logStr = string.Format("Capturing at Current={0:f2} mA,
Index={1}/{2}",
                        current.ToString(), Z_MovePlan_Index, Z_MovePlan.Count - 1);
                    toolStripStatusLabel1.Text = logStr;
                    Console.WriteLine(logStr);
                    Z_MovePlan_Index++;

                    //etlSettingForm.LensDriver.SendGetCurrentCommand();

                    TicksAllowZMove = gTimer.ElapsedTicks + (int)(5 * 1e7);  // Temporarily
add 5 sec for waiting.
                    flagSoftwareTriggered = false;
                }
            }

            // finish recording
            else
            {
                // finish recording
                hTimerCamSoftwareTriggering.Stop();
                hTimerETLIntermittent.Stop();
                //Invoke(new Action(delegate
                //{
                //Button_cam_stop_Click(button_cam_stop, EventArgs.Empty);
                //}));
                StopRecording();
                toolStripStatusLabel1.Text = string.Format("Finished. at Current={0:f2} mA,
Index={1}/{2}", Z_MovePlan[Z_MovePlan.Count - 1].ToString(), Z_MovePlan_Index,
Z_MovePlan.Count - 1);
            }
        }

        public void ForceStopHighResolutionTimers(HighResolutionTimer[] timers)
        {
            foreach(var ht in timers)
```

```
            {
                if (ht == null || !ht.IsRunning)
                    continue;

                ht.Stop(true);
            }

        }
    }
}
```

Code A9. Form/PositionControllerSetting.cs

```
using Hamamatsu.DCAM4;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO.Ports;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace ETL_system
{
    /// <summary>
    /// Setting Form of xyz position controller such as LEP MAC6000.
    /// </summary>
    public partial class PositionControllerSetting : Form
    {
        Dictionary<string, string> ports_dic;
        SerialPortExtended spe;
        MAC6 mac6;

        System.Windows.Forms.Timer timerGet, timerMove;
        int sendMoveCount = 0;

        private string last_str="";
        private bool zPosLocker = false;

        public MAC6 MAC6_Instance { get => mac6; }
        public double Depth_micron { get => mac6.Z_RealPositionMicron; }
        public double Depth_Raw { get => mac6.Z_ReadValue; }
        /// <summary>
        ///
        /// </summary>
        public List<double> Z_MovePlan { get; set; }

        public bool IsCalibrating { get; set; }
        /// <summary>
        /// "IsOperating" flag will turn ON when using SetZero command -- timer updating of
position should be stopped.
        /// </summary>
        public bool IsOperating { get; set; }
        public bool IsShowingRealDistance { get; set; }
        public OperatorForm OPForm { get; set; }

        public PositionControllerSetting()
        {
            InitializeComponent();

            IsCalibrating = false;
            IsOperating = false;
            IsShowingRealDistance = true;

            // reflesh position
            timerGet = new Timer();
            timerGet.Tick += TimerUpdateZPosition_Tick;
            timerGet.Interval = 100;

            timerMove = new Timer();
            timerMove.Tick += TimerMoveZ_Tick;
            timerMove.Interval = 120;
        }

        private void PositionControllerSetting_Load(object sender, EventArgs e)
```

```
        {
            // Get serial ports
            ports_dic = Utils.GetSerialPortList();
            foreach (var name in ports_dic.Keys)
                comboBox_port.Items.Add(name);

            if (comboBox_port.Items.Count > 0)
            {
                for (int i = 0; i < comboBox_port.Items.Count; i++)
                {
                    if (comboBox_port.Items[i].ToString().IndexOf("COM4") != -1)
                    {
                        comboBox_port.SelectedIndex = i;
                        break;
                    }
                }
            }
            else
                button_connect.Enabled = false;

            // debug
            button_connect_Click(null, EventArgs.Empty);
            //radioButton_z_speed_fastest.Checked = true;
        }
        public void button_connect_Click(object sender, EventArgs e)
        {
            if (comboBox_port.SelectedIndex < 0) return;
            BeginInvoke(new Action(delegate
            {
                string id = ports_dic[(string)comboBox_port.SelectedItem];
                try
                {
                    if (spe != null && spe.IsOpen)
                        spe.Close();
                    else
                    {
                        spe = new SerialPortExtended(id,
SerialPortExtended.DeviceTypeEnum.MAC6);
                        mac6 = new MAC6(spe);
                        spe.OpenPort();

                        spe.DataReceived += (new
SerialPortUtilsEx()).DataReceivedHandler_General;
                        spe.GeneralUseEvent += mac6.CallBackDataReceivedEvent;
                        //if (spe != null && spe.IsOpen)
                        //    UpdateDataReceivedEvents(spe,
SerialPortExtended.DeviceTypeEnum.MAC6);
                    }
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.ToString());
                }
                bool isOpen = spe.IsOpen;
                button_connect.Text = Utils.ConnectionButtonHelper(isOpen);
                label_connection = Utils.ConnectionLabelHelper(isOpen, label_connection);
                mac6.IsOpen = isOpen;

                numericUpDown_ReadValuePerMicron_ValueChanged(null, EventArgs.Empty);
                mac6.SendSetSpeed_Z((int)numericUpDown_z_speed.Value);
            }));
        }

        private void PositionControllerSetting_FormClosing(object sender,
FormClosingEventArgs e)
        {
            e.Cancel = true;
            this.Hide();
        }
```

```
        private void TimerUpdateZPosition_Tick(object sender, EventArgs e)
        {
            if (mac6.IsOpen)
            {
                spe.IsGeneralUse = true;
                mac6.SendGetPosition_Z();
            }

            // Update label
            if (this.InvokeRequired)
            {
                Invoke(new Action(delegate
                {
                    label_readvalue.Text = string.Format("Raw = {0}", mac6.Z_ReadValue);
                    label_z_real.Text = string.Format("z = {0:f1} µm",
mac6.Z_RealPositionMicron);
                }));
            }
            else
            {
                label_readvalue.Text = string.Format("Raw = {0}", mac6.Z_ReadValue);
                label_z_real.Text = string.Format("z = {0:f1} µm",
mac6.Z_RealPositionMicron);
            }

            // Update StatusStrip
            if (IsOperating)
                toolStripStatusLabel1.Text = "Operating";
            else
                toolStripStatusLabel1.Text = "Watching";
        }
        private void TimerMoveZ_Tick(object sender, EventArgs e)
        {
            if (!IsOperating)
                mac6.RefleshTargetPos();

            else if (IsOperating && mac6.IsMoving)
                mac6.CheckZReached();  // handle mac6.IsMoving flag
        }

        private void button1_Click(object sender, EventArgs e)
        {
            StartTimers();
        }
        private void numericUpDown_z_pos_ValueChanged(object sender, EventArgs e)
        {
            if (!mac6.IsOpen || IsOperating || zPosLocker)
                return;

            double val = (double)numericUpDown_z_pos.Value;
            if (radioButton_z_pos_unit_real.Checked)
                val *= (double)numericUpDown_ReadValuePerMicronPos.Value;

            mac6.PushOrder((int)val);
        }

        public void StartTimers()
        {
            timerGet.Start();
            timerMove.Start();
        }

        /// <summary>
        /// change interval to faster in recording session
        /// </summary>
        /// <param name="Interval_ms">New interval for updating z-position</param>
        public void ChangeTimerInterval(int Interval_ms)
        {
```

```
            timerGet.Interval = Interval_ms;
        }
        private void numericUpDown_ReadValuePerMicron_ValueChanged(object sender, EventArgs
e)
        {
            mac6.Z_ReadValuePerMicronPos =
(double)numericUpDown_ReadValuePerMicronPos.Value;
            mac6.Z_ReadValuePerMicronNeg =
(double)numericUpDown_ReadValuePerMicronNeg.Value;
        }
        #region FocusDriveMotor Calibration

        private void button_fdm_set_zero_Click(object sender, EventArgs e)
        {
            //if (!IsCalibrating) return;
            IsOperating = true;
            numericUpDown_z_pos.Value = 0;
            mac6.SendSetZero();
            IsOperating = false;
        }

        private void button_fdm_set_a_micron_Click(object sender, EventArgs e)
        {
            if (!IsCalibrating) return;
            // todo
        }

        private void button_halt_Click(object sender, EventArgs e)
        {
            // not work yet
            // todo
            mac6.SendHalt();
        }

        public void RelativeMoveZMicron_ToDeep()
        {

        }

        private void numericUpDown_zstack_begin_ValueChanged(object sender, EventArgs e)
        {
            // Condition: Begin <= End
            if (numericUpDown_zstack_top.Value > numericUpDown_zstack_bottom.Value)
            {
                if(sender == numericUpDown_zstack_top)
                    numericUpDown_zstack_top.Value = numericUpDown_zstack_bottom.Value;
                else if (sender == numericUpDown_zstack_bottom)
                    numericUpDown_zstack_bottom.Value = numericUpDown_zstack_top.Value;
            }

            int range = (int)numericUpDown_zstack_bottom.Value -
(int)numericUpDown_zstack_top.Value;
            double frames = Math.Ceiling(range / (double)numericUpDown_zstack_zstep.Value +
1);

            label1_total_frames.Text = string.Format("Number of frames in session: {0:f0}",
frames);
        }

        private void radioButton_z_pos_unit_CheckedChanged(object sender, EventArgs e)
        {
            // This event is called when switched state
            if (zPosLocker)
                return;

            IsShowingRealDistance = radioButton_z_pos_unit_real.Checked;

            // change to Real distance
            if(sender==radioButton_z_pos_unit_real && radioButton_z_pos_unit_real.Checked)
```

```
            {
                label1_z_pos_unit.Text = "[µm]";
                numericUpDown_z_pos.DecimalPlaces = 1;
                numericUpDown_z_pos.Increment = 1;
                numericUpDown_z_pos.Maximum = 100000;
                numericUpDown_z_pos.Minimum = -100000;
                zPosLocker = true;
                double newValue = (double)numericUpDown_z_pos.Value /
(double)numericUpDown_ReadValuePerMicronPos.Value;
                numericUpDown_z_pos.Value = (decimal)newValue;
            }

            // change to Raw value
            else if(sender == radioButton_z_pos_unit_raw &&
radioButton_z_pos_unit_raw.Checked)
            {
                label1_z_pos_unit.Text = "[a.u.]";
                numericUpDown_z_pos.DecimalPlaces = 0;
                numericUpDown_z_pos.Increment = 10000;
                numericUpDown_z_pos.Maximum = 1000000000;
                numericUpDown_z_pos.Minimum = -1000000000;
                zPosLocker = true;
                double newValue = (double)numericUpDown_z_pos.Value *
(double)numericUpDown_ReadValuePerMicronPos.Value;
                numericUpDown_z_pos.Value = (decimal)newValue;
            }

            zPosLocker = false;
        }

        public List<double> CreateMovePlan()
        {
            double zstep = (double)numericUpDown_zstack_zstep.Value;
            double top = (double)numericUpDown_zstack_top.Value;
            double bottom = (double)numericUpDown_zstack_bottom.Value;

            List<double> movePlan = new List<double>();
            bool topToBottom = radioButton_zstack_top_to_bottom.Checked;
            movePlan = mac6.CreateZMovePlan(top, bottom, zstep, topToBottom);
            Z_MovePlan = movePlan;

            return movePlan;
        }

        private void radioButton_z_speed_CheckedChanged(object sender, EventArgs e)
        {
            if(radioButton_z_speed_slow.Checked)
            {
                numericUpDown_z_speed.Value = 500;
            }
            else if(radioButton_z_speed_faster.Checked)
            {
                numericUpDown_z_speed.Value = 1000;
            }
            else if(radioButton_z_speed_fastest.Checked)
            {
                numericUpDown_z_speed.Value = 2000;
            }
        }

        private void numericUpDown_z_speed_ValueChanged(object sender, EventArgs e)
        {
            mac6.SendSetSpeed_Z((int)numericUpDown_z_speed.Value);
        }

        private void button_stack_helper_Click(object sender, EventArgs e)
        {
            // temporary
            MAC6_Instance.SendMovePosition_ZVal(MAC6_Instance.Z_TargetValue);
```

```
        }

        private void button1_Click_1(object sender, EventArgs e)
        {
            CreateMovePlan();
            label1_total_frames.Text = string.Format("Number of frames in session: {0:f0}",
Z_MovePlan.Count);

            // change Allocation frame number in CameraSetting Form
            OPForm.AllocationFrameCount = Z_MovePlan.Count;
        }
        #endregion
    }
}
```

Code A10. LiveImaging/ImageDisplay.cs

```
using Hamamatsu.DCAM4;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace ETL_system
{
    public partial class ImageDisplay : Form
    {
        private BufferedGraphicsContext bufferContext;
        private BufferedGraphics buffer;
        // People say that Forms.Timer is better than System.Timers.Timer if use in Form.
Probably true.
        private System.Windows.Forms.Timer timer;
        private System.Windows.Forms.Timer timer_RefleshRate;

        private int counter = 0;    // measure refresh rate
        private float targetFPS;

        private float prevImgScaling = 0;
        private DCAMLut lut;
        private RotateFlipType rotationFlipType;
        private string rotationFlipLable;

        private const double ScrollBarMarginRatio = 0.20;

        public Graphics Buffer { get => buffer.Graphics; }
        /// <summary>
        /// For showing live images in PictureBox. Intensity has scaled by LUTs setup.
        /// </summary>
        public Image NextImage { get; set; }
        /// <summary>
        /// This Image has an original, unscaled intensities (8 bit RGB).
        /// </summary>
        public Image OriginalImage { get; set; }
        /// <summary>
        /// Simple byte array of original image.
        /// </summary>
        public byte[] OriginalImageBuffer { get; set; }

        /// <summary>
        /// If true, displayed image is updated. Set by OPForm, "CameraUpdatePicture_Async"
        /// </summary>
        public bool FlagImageUpdated { get; set; }
        /// <summary>
        /// Size of Image that showing in PictureBox.
        /// </summary>
        public Size ImageResolution { get; set; }
        /// <summary>
        /// Indicates size scaling ratio, PictureBox vs OriginalImage.
        /// </summary>
        public float ImageScaling { get; private set; }
        public int BitDepth { get; set; }
        public int MaxIntensity { get; set; }
        public DCAMLut LUT { get => lut; set { lut = value; } }
        /// <summary>
        /// Rotation angle (clock wise; [deg]) and flip/mirror type of displayed image
        /// </summary>
```

```
        public RotateFlipType RotationFlipType
        {
            get => rotationFlipType;
            set
            {
                rotationFlipType = value;
            }
        }
        public string RotationLabel
        {
            get => rotationFlipLable;
            set
            {
                rotationFlipLable = value;
                this.Invoke((MethodInvoker)(() => label_rotation.Text = value));
            }
        }

        /// <summary>
        /// For updating NextImage by multi-thread
        /// </summary>
        public object LockObj { get; }

        public ImageDisplay()
        {
            InitializeComponent();
            pictureBox.CreateGraphics().InterpolationMode =
System.Drawing.Drawing2D.InterpolationMode.HighQualityBicubic;
            LockObj = new object();

            bufferContext = BufferedGraphicsManager.Current;
            buffer = bufferContext.Allocate(pictureBox.CreateGraphics(),
pictureBox.DisplayRectangle);

            // temporary
            timer = new System.Windows.Forms.Timer();
            timer.Interval = 33;
            timer.Tick += new EventHandler(timer_Tick);

            // measure reflesh rate
            timer_RefleshRate = new System.Windows.Forms.Timer();
            timer_RefleshRate.Interval = 1000;
            timer_RefleshRate.Tick += new EventHandler(timer_RefleshRate_Tick);

            // TODO: Take info from camera or SettingForm
            BitDepth = 16;
            MaxIntensity = 65535;

            rotationFlipType = RotateFlipType.RotateNoneFlipNone;

            // temporary
            numericUpDown_scrollbar_max_limit_ValueChanged(null, EventArgs.Empty);
            lut = new DCAMLut();
            lut.camerabpp = BitDepth;
            lut.cameramax = MaxIntensity;
            lut.inmax = 230;  // set max first bacause (min, max) begins from (0, 0)
            lut.inmin = 70;
        }

        public void StartLiveTimer(int fps = 30)
        {
            // init labels and so on
            label_frame_number.Text = "Frame Number";

            // clear old images (?)
            NextImage = null;
            OriginalImage = null;

            counter = 0;
```

```
            timer.Interval = (int)(1000.0 / fps);
            targetFPS = fps;
            timer.Start();
            timer_RefleshRate.Start();
        }
        public void StopLiveTimer()
        {
            timer.Stop();
            timer_RefleshRate.Stop();
        }
        private void timer_Tick(object sender, EventArgs e)
        {
            if (FlagImageUpdated)
            {
                // temporarily use timer for live imaging
                pictureBox.Image = NextImage;
                if(NextImage != null)
                    pictureBox.Image.RotateFlip(RotationFlipType);
                FlagImageUpdated = false;
            }
            counter++;

            UpdateImagePropertyLabels_Async();
        }
        private void timer_RefleshRate_Tick(object sender, EventArgs e)
        {

            float rate = (float)(counter * (1000.0 / timer_RefleshRate.Interval));
            counter = 0;

            // check current FPS and adjust timer
            if (Math.Abs(rate - targetFPS) > 2)
            {
                int interval = timer.Interval;
                if (rate > targetFPS)
                    interval += 2;
                else if (rate < targetFPS && timer.Interval > 1)
                    interval -= 2;

                timer.Interval = 1;
                timer.Stop();
                timer.Interval = interval;
                timer.Start();
            }

            BeginInvoke(new Action(delegate
            {
                label_reflesh_rate.Text = string.Format("Reflesh Rate: {0:#.#} fps
(interval={1})", rate, timer.Interval);
            }));
        }

        private void Paint(object sender, PaintEventArgs e)
        {
            if (buffer != null)
            {
                //buffer.Graphics.Clear(pictureBox.BackColor);
                //buffer.Graphics.DrawImageUnscaled(, new Point(0, 0));
                buffer.Render();
            }
        }

        private void PicturePanel_Load(object sender, EventArgs e)
        {
            numericUpDown_max.Maximum = lut.cameramax;
            numericUpDown_min.Maximum = lut.cameramax;
            hScrollBar_max.Maximum = lut.cameramax;
            hScrollBar_min.Maximum = lut.cameramax;
```

```
            SetLUTs(lut.inmin, lut.inmax);
        }

        public void SetLUTs(int min, int max)
        {
            BeginInvoke(new Action(delegate
            {
                lut.inmin = min;
                lut.inmax = max;
                numericUpDown_min.Value = min;
                numericUpDown_max.Value = max;
                //hScrollBar_min.Value = min;
                //hScrollBar_max.Value = max;
            }));
        }

        public void UpdateImage(Image img)
        {
            buffer.Graphics.Clear(pictureBox.BackColor);
            buffer.Graphics.InterpolationMode =
System.Drawing.Drawing2D.InterpolationMode.HighQualityBicubic;
            buffer.Graphics.DrawImageUnscaled(img, new Point(0, 0));
            buffer.Render();
        }

        public void UpdateImagePropertyLabels_Async()
        {
            if (NextImage == null) return;
            // ImageScaling is updated in TransformPosition function.
            //ImageScaling = pictureBox.Size.Width / (float)NextImage.Width;
            //if (ImageScaling == prevImgScaling) return;
            ImageResolution = NextImage.Size;

            Task task = Task.Run(() =>
            {
                prevImgScaling = ImageScaling;
                BeginInvoke(new Action(delegate
                {
                    label_img_resolution.Text = string.Format(
                    "{0}x{1}", ImageResolution.Width, ImageResolution.Height);
                    label_scaling.Text = string.Format("Display Scaling: {0:#.#} %",
ImageScaling * 100);
                }));
            });
        }

        private void PicturePanel_ResizeEnd(object sender, EventArgs e)
        {

        }

        private void checkBox_auto_luts_CheckedChanged(object sender, EventArgs e)
        {
            //if (pictureBox.Image == null) return;
            //int min, max;
            //(min, max) = Utils.GetMinMaxIntensityOfImage((Bitmap)NextImage, MaxIntensity);

            //lut.inmax = max;
            //lut.inmin = min;
            //SetLUTs(min, max);

            // TODO: apply this lut to the image. (may be thrown to OpForm)
        }

        private void UpdateIntensityLabel(Point Pos, bool PosOnImage = false)
        {
            if (OriginalImage == null || OriginalImageBuffer == null)
                return;
```

```
            Point p = Pos;
            if (!PosOnImage)
                p = TransformPosition(Pos);

            if (p.X >= 0 && p.Y >= 0)
            {
                var src = OriginalImageBuffer;
                int w = this.pictureBox.Image.Width;
                int h = this.pictureBox.Image.Height;

                int bytePerPixels = LUT.camerabpp / 8;
                int stride = w * bytePerPixels;
                int currentPosIndex = stride * p.Y + bytePerPixels * p.X;

                UInt16 intensity = 0;
                if(LUT.camerabpp > 8)
                    // Single data from 2-byte
                    intensity = BitConverter.ToUInt16(src, currentPosIndex);
                else
                    // Single data from 1-byte (does not exist ToUInt8() function)
                    intensity = Convert.ToUInt16(src[currentPosIndex]);

                //var color = ((Bitmap)OriginalImage).GetPixel(p.X, p.Y);
                //int intensity = (int)(color.GetBrightness() * (float)MaxIntensity);    //
TODO: maybe not correct
                //int intensity = (int)(color.GetBrightness() * this.LUT.inmax);    // TODO:
maybe not correct
                label_xy_intensity.Text = string.Format("({0}, {1}) = {2}", p.X, p.Y,
intensity);
            }
        }
        public void UpdateFrameNumber(int iFrame, int maxFrame)
        {
            if (InvokeRequired)
            {
                Invoke(new Action(() =>
                {
                    label_frame_number.Text = string.Format("Frames: {0} / {1}", iFrame,
maxFrame);
                }));
            }
            else
                label_frame_number.Text = string.Format("Frames: {0} / {1}", iFrame,
maxFrame);
        }

        /// <summary>
        /// Transform position on pictureBox to position on Image. This function consider
PictureBox.SizeMode == Zoom.
        /// </summary>
        /// <returns>Position on Image</returns>
        private Point TransformPosition(Point PosOnControl)
        {
            Point newPos = new Point();
            if (pictureBox.SizeMode == PictureBoxSizeMode.Zoom)
            {
                // Get coordinate that mouse positioned, for 'Zoom' mode PictureBox
                double ratioPicBox = pictureBox.Width / (double)pictureBox.Height;
                double ratioImage = pictureBox.Image.Width /
(double)pictureBox.Image.Height;

                double scaling, paddingOneSide;
                if (ratioPicBox > ratioImage)
                {
                    // fitted vertically
                    scaling = pictureBox.Height / (double)pictureBox.Image.Height;
                    // check width of PicBox
                    paddingOneSide = (pictureBox.Width - (double)pictureBox.Image.Width *
```

```
scaling) / 2.0;

                    newPos.X = (int)((PosOnControl.X - paddingOneSide)
                        * (pictureBox.Image.Width / (double)(pictureBox.Size.Width -
paddingOneSide * 2.0)));
                    newPos.Y = (int)(PosOnControl.Y * (pictureBox.Image.Height /
(double)pictureBox.Size.Height));
                }
                else
                {
                    // fitted horizontally
                    scaling = pictureBox.Width / (double)pictureBox.Image.Width;
                    // check height of PicBox
                    paddingOneSide = (pictureBox.Height - (double)pictureBox.Image.Height *
scaling) / 2.0;

                    newPos.X = (int)(PosOnControl.X * (pictureBox.Image.Width /
(double)pictureBox.Size.Width));
                    newPos.Y = (int)((PosOnControl.Y - paddingOneSide)
                        * (pictureBox.Image.Height / (double)(pictureBox.Size.Height -
paddingOneSide * 2.0)));
                }
                ImageScaling = (float)scaling;
            }
            else
            {
                // for 'StretchImage' and 'AutoSize'
                newPos.X = (int)(PosOnControl.X * (pictureBox.Image.Width /
(double)pictureBox.Size.Width));
                newPos.Y = (int)(PosOnControl.Y * (pictureBox.Image.Height /
(double)pictureBox.Size.Height));
            }

            if (newPos.X >= pictureBox.Image.Width) newPos.X = pictureBox.Image.Width - 1;
            if (newPos.Y >= pictureBox.Image.Height) newPos.Y = pictureBox.Image.Height - 1;

            return newPos;
        }

        /// <summary>
        /// Show intensity of pixel that mouse positioned
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void pictureBox_MouseMove(object sender, MouseEventArgs e)
        {
            if (pictureBox.Image == null) return;
            UpdateIntensityLabel(new Point(e.X, e.Y), false);
        }

        private void pictureBox_Paint(object sender, PaintEventArgs e)
        {
            if (pictureBox.Image == null) return;
            Point p = pictureBox.PointToClient(Cursor.Position);
            UpdateIntensityLabel(new Point(p.X, p.Y), false);
        }

        private void PicturePanel_FormClosing(object sender, FormClosingEventArgs e)
        {
            e.Cancel = true;
            this.Hide();
        }

        #region LUTs Setup UI
        private void hScrollBar_max_ValueChanged(object sender, EventArgs e)
        {
            int val = Math.Min(hScrollBar_max.Value,
(int)((double)numericUpDown_max.Value*(1+ScrollBarMarginRatio)));
            numericUpDown_max.Value = hScrollBar_max.Value;
```

```
            lut.inmax = (int)(numericUpDown_max.Value);
        }
        private void hScrollBar_min_ValueChanged(object sender, EventArgs e)
        {
            numericUpDown_min.Value = hScrollBar_min.Value;
            lut.inmin = (int)(numericUpDown_min.Value);
        }
        private void numericUpDown_max_ValueChanged(object sender, EventArgs e)
        {
            if (numericUpDown_max.Value > hScrollBar_max.Maximum)
                hScrollBar_max.Maximum = (int)(numericUpDown_max.Value + 1);

            if (numericUpDown_max.Value < numericUpDown_min.Value)
                numericUpDown_max.Value = numericUpDown_min.Value;

            //hScrollBar_max.Maximum = (int)numericUpDown_max.Value;
            //hScrollBar_max.Value = (int)numericUpDown_max.Value;
            lut.inmax = (int)(numericUpDown_max.Value);

            return;

            // better UI for dynamic range (2022 Jan)
            int newlimit = (int)((double)numericUpDown_max.Value *
(1+ScrollBarMarginRatio));
            int lowerlimit = (int)numericUpDown_min.Value;
            newlimit = Math.Min(65535, newlimit);
            newlimit = Math.Max(newlimit, lowerlimit);
            hScrollBar_max.Maximum = newlimit;
            hScrollBar_min.Maximum = newlimit;
            numericUpDown_scrollbar_max_limit.Value = newlimit;
        }
        private void numericUpDown_min_ValueChanged(object sender, EventArgs e)
        {
            if (numericUpDown_min.Value > hScrollBar_min.Minimum)
                hScrollBar_min.Minimum = (int)(numericUpDown_min.Value - 1);

            if (numericUpDown_max.Value < numericUpDown_min.Value)
                numericUpDown_min.Value = numericUpDown_max.Value;

            //hScrollBar_min.Minimum = (int)numericUpDown_min.Value;
            //hScrollBar_min.Value = (int)numericUpDown_min.Value;
            lut.inmin = (int)(numericUpDown_min.Value);

            return;

            // better UI for dynamic range (2022 Jan)
            int newlimit = (int)((double)numericUpDown_min.Value * (1-
ScrollBarMarginRatio));
            int higherlimit = (int)numericUpDown_max.Value;
            newlimit = Math.Max(0, newlimit);
            newlimit = Math.Min(newlimit, higherlimit);
            hScrollBar_max.Minimum = newlimit;
            hScrollBar_min.Minimum = newlimit;
            //numericUpDown_scrollbar_max_limit.Value = newlimit;
        }
        private void numericUpDown_scrollbar_max_limit_ValueChanged(object sender,
EventArgs e)
        {
            hScrollBar_max.Maximum = (int)(numericUpDown_scrollbar_max_limit.Value);
            hScrollBar_min.Maximum = (int)(numericUpDown_scrollbar_max_limit.Value);
        }
        #endregion
    }
}
```

Code A11. LiveImaging/ImageProjDisplay.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Windows.Media.Media3D;
using System.Runtime.InteropServices;
using ETL_system.LiveImaging;

namespace ETL_system
{
    public partial class ImageProjDisplay : Form
    {
        public ImageProjDisplay()
        {
            InitializeComponent();
        }
        #region Props
        /// <summary>
        /// voxel resolution [um]
        /// </summary>
        public double PixelSizeXY { get; set; }
        /// <summary>
        /// voxel resolution [um]
        /// </summary>
        public double PixelSizeZ { get; set; }

        public Image ImageXY { get { return pictureBox_xy.Image; } set {
                if (InvokeRequired)
                    BeginInvoke(new Action(delegate
                    {
                        pictureBox_xy.Image = value;
                    }));
                else
                    pictureBox_xy.Image = value;
            } }
        public Image ImageXZ { get { return pictureBox_xz.Image; } set {
                if (InvokeRequired)
                    BeginInvoke(new Action(delegate
                    {
                        pictureBox_xz.Image = value;
                    }));
                else
                    pictureBox_xz.Image = value;
            }
        }
        public Image ImageYZ { get { return pictureBox_yz.Image; } set {
                if (InvokeRequired)
                    BeginInvoke(new Action(delegate
                    {
                        pictureBox_yz.Image = value;
                    }));
                else
                    pictureBox_yz.Image = value;
            }
        }
        public Image ImageZIndicator { get { return pictureBox_zindicator.Image; } set {
                if (InvokeRequired)
                    BeginInvoke(new Action(delegate
                    {
```

```
                    pictureBox_zindicator.Image = value;
                }));
            else
                pictureBox_zindicator.Image = value;
        }
    }

    public Size3D VolumeSizePixel { get => volumeSizePixel; set {
            volumeSizePixel = value;
            VolumeSizeReal = new Size3D(value.X * PixelSizeXY, value.Y * PixelSizeXY,
value.Z * PixelSizeZ);
        } }
    public Size3D VolumeSizeReal { get; set; }

    public OperatorForm OPForm { get; set; }

    #endregion

    public ImageProjection.LiveProjectionBuffer liveProjBuffer;
    public ImageProjection.XY_Projection xy_Projection;
    private Timer timerLiveUpdate = new Timer();

    private Size3D volumeSizePixel;
    private int LUTsMaxIntensity = 1000;
    public int UpdateIntervalShouldBe_ms = 100; // [ms]

    // Test
    TestProjection proj;
    ushort[][][] ExampleVolume;
    Timer timerTestUpdate = new Timer();
    double updateDisplayRate = 9;  // [Hz]
    double updateAxialScanRate = 0.5; // [Hz]; for TEST Projection
    int ZPositionPx = 0;
    int ZPositionPxPrev = 0;
    int ZStridePxPerUpdate = 0;

    Bitmap PlaneBmpBuffer;
    byte[] VolumeByteBuffer;
    ushort[][] xzBuffer;
    ushort[][] yzBuffer;                // for calculation of yz-image if needed
    //ushort[][] yzBuffer_transpose;       // for YZ display
    ushort[][] CurrentZIndicatorBuffer;  // Pixel value must be Zero or Max
    int CurrentZIndicatorWidthPx = 1;
    int BitPerPixel = 16;


    private void ImageProjDisplay_Load(object sender, EventArgs e)
    {
        // for debug  ----------------------------------------
        PixelSizeXY = 1.5;
        PixelSizeZ = 1.5;
        int xmax = 256, ymax = 256, zmax = 64;
        VolumeSizePixel = new Size3D(xmax, ymax, zmax);
        VolumeByteBuffer = new byte[xmax * ymax * 4 * zmax];  // 32 bpp
        // for debug  ----------------------------------------

        //liveProjBuffer = new ImageProjection.LiveProjectionBuffer();
        //liveProjBuffer.xyProjection = new ImageProjection.XY_Projection(BitPerPixel,
xmax, ymax);
        //xy_Projection = liveProjBuffer.xyProjection;

        timerLiveUpdate.Tick += TimerLiveUpdate_Tick;
        timerLiveUpdate.Interval = (int)(1000.0 / updateDisplayRate);

        // for TEST  ----------------------------------------
        //proj = new TestProjection();
        //hTimerTestUpdate.Elapsed += testUpdateTimer_Tick;
        //hTimerTestUpdate.Interval = (int)(1000.0 / updateDisplayRate);
```

```
            //ZStridePxPerUpdate = (int)(updateAxialScanRate * VolumeSizePixel.Z /
updateDisplayRate);

            // Reticle for slice image
            pictureBox_xy_transparent.Parent = pictureBox_xy;
            pictureBox_xy_transparent.BackColor = Color.Transparent;
            pictureBox_xy_transparent.Location = pictureBox_xy.Location;
            pictureBox_xy_transparent.Size = pictureBox_xy.Size;

            numericUpDown_xy_voxel_size_ValueChanged(null, EventArgs.Empty);
        }
        public void InitializeBuffers(int width, int height, int depth)
        {
            int xmax = width;
            int ymax = height;
            int zmax = depth;

            VolumeSizePixel = new Size3D(xmax, ymax, zmax);
            VolumeByteBuffer = new byte[xmax * ymax * 4 * zmax];  // 32 bpp

            // Calc real size [um]
            int realX = (int)(PixelSizeXY * VolumeSizePixel.X);
            int realY = (int)(PixelSizeXY * VolumeSizePixel.Y);
            int realZ = (int)(PixelSizeZ * VolumeSizePixel.Z);
            VolumeSizeReal = new Size3D(realX, realY, realZ);

            xzBuffer = new ushort[zmax][];  // [z, x]
            yzBuffer = new ushort[zmax][];  // [z, y]
            //yzBuffer_transpose = new ushort[ymax][];  // [y, z] for display
            CurrentZIndicatorBuffer = new ushort[zmax][];
            for (int z = 0; z < zmax; z++)
            {
                xzBuffer[z] = new ushort[xmax];
                yzBuffer[z] = new ushort[ymax];
                CurrentZIndicatorBuffer[z] = new ushort[CurrentZIndicatorWidthPx];
            }
            //for (int y = 0; y < ymax; y++)
            //    yzBuffer_transpose[y] = new ushort[zmax];

            xy_Projection = new ImageProjection.XY_Projection(BitPerPixel, xmax, ymax);

            this.LUTsMaxIntensity = (int)numericUpDown_luts_max.Value;
        }
        /// <summary>
        /// Fit the size and location of controls such as picturebox.
        /// </summary>
        public void AdjustControlSizeAndLocation(double Magnify = 1.0, double MagnifyAxial
= 1.0)
        {
            if(Magnify == 1.0 && MagnifyAxial == 1.0)
            {
                pictureBox_xy.SizeMode = PictureBoxSizeMode.AutoSize;
                pictureBox_xz.SizeMode = PictureBoxSizeMode.AutoSize;
                pictureBox_yz.SizeMode = PictureBoxSizeMode.AutoSize;
                pictureBox_zindicator.SizeMode = PictureBoxSizeMode.StretchImage;
            }
            else if(MagnifyAxial != 1.0 )
            {
                pictureBox_xy.SizeMode = PictureBoxSizeMode.Zoom;
                pictureBox_xz.SizeMode = PictureBoxSizeMode.StretchImage;
                pictureBox_yz.SizeMode = PictureBoxSizeMode.StretchImage;
                pictureBox_zindicator.SizeMode = PictureBoxSizeMode.StretchImage;
            }
            else
            {
                pictureBox_xy.SizeMode = PictureBoxSizeMode.Zoom;
                pictureBox_xz.SizeMode = PictureBoxSizeMode.Zoom;
                pictureBox_yz.SizeMode = PictureBoxSizeMode.Zoom;
                pictureBox_zindicator.SizeMode = PictureBoxSizeMode.StretchImage;
```

```
            }

            // Image size
            int x = (int)(volumeSizePixel.X * Magnify);
            int y = (int)(volumeSizePixel.Y * Magnify);
            int z = (int)(volumeSizePixel.Z * Magnify * MagnifyAxial);

            // At first, Form Size should be changed because Controls might have Anchor
property
            int formSizeX = x + z + 30 + groupBox_info.Size.Width + 50;
            int formSizeY = y + z + 50 + 50;
            this.Size = new Size(formSizeX, formSizeY);

            // picturebox
            pictureBox_xy.Size = new Size(x, y);
            pictureBox_xz.Size = new Size(x, z);
            pictureBox_yz.Size = new Size(z, y);
            pictureBox_zindicator.Size = new Size((int)(16*Magnify), z);

            pictureBox_xz.Location = new Point(pictureBox_xy.Location.X, y + 10);
            pictureBox_yz.Location = new Point(x + 10, pictureBox_xy.Location.Y);
            pictureBox_zindicator.Location = new Point(pictureBox_yz.Location.X,
pictureBox_xz.Location.Y);

            // info box. Right to YZ panel
            groupBox_info.Location = new Point(
                pictureBox_yz.Location.X + pictureBox_yz.Size.Width + 10,
                pictureBox_yz.Location.Y);

            // Image direction related to Stage
            panel_direction_to_stage.Location = new Point(
                groupBox_info.Location.X,
                groupBox_info.Location.Y + groupBox_info.Height + 15);

        }
        public void StartLiveTimer()
        {
            //hTimerLiveUpdate.Start();  // Not implemented

        }
        public void StopLiveTimer()
        {
            timerLiveUpdate.Stop();
            timerTestUpdate.Stop();
        }
        public void ClearBuffer()
        {
            liveProjBuffer.RawFrameBuffer.Clear();
            VolumeByteBuffer = new byte[] { };
        }

        public unsafe void UpdateImageBuffer(byte[] PlaneBuffer, ImageProjection.ZPitchRow
zPitch)
        {
            // data from example volume
            xy_Projection.SetImageFromRaw(PlaneBuffer);

            Stopwatch sw = new Stopwatch();
            sw.Start();
            xy_Projection.SetImageFromRaw(PlaneBuffer);
            sw.Stop();
            Console.WriteLine($"Test {sw.ElapsedMilliseconds} ms");

            (ushort[] xz, ushort[] yz) result;
            // Check the Projection Method
            switch (liveProjBuffer.projMethod)
            {
                case ImageProjection.ProjectionMethods.MaxIntensity:
                    result = xy_Projection.GetMaxIntensityProjection();
```

```
                    break;
                case ImageProjection.ProjectionMethods.AtThePoint:
                    result = xy_Projection.GetSliceProjectionAtPoint();
                    break;
                default:
                    result = xy_Projection.GetMaxIntensityProjection();
                    break;
            }

            // Get z-pitch as Pixel (referring VolumeBuffer of LiveProjBuffer class)
            int[] zPitchPx = new int[] {
            liveProjBuffer.TransformZRealToPixel(zPitch.MinReal),
            liveProjBuffer.TransformZRealToPixel(zPitch.MaxReal)};

            // Update corresponding z with same xy-image
            for (int z = zPitchPx[0]; z < zPitchPx[1]; z++)
            {
                // Update 2D buffers
                // Marshal.Copy(result.xz, 0, xzBuffer, result.xz.Length);

                xzBuffer[z] = result.xz;  // result.xz is 1-dim data actually
                yzBuffer[z] = result.yz;  // same
                                          // For displaying YZ image, use transposed yzBuffer
                                          //for (int y = 0; y < yzBuffer_transpose.Length;
y++)
                                          //    yzBuffer_transpose[y][z] = result.yz[y];

                // z-position indicator (Zero or Max)
                ushort scanState = 0;
                if (CurrentZIndicatorBuffer[z][0] == 0)
                    scanState = 65535;  // 16 bit
                for (int x = 0; x < CurrentZIndicatorWidthPx; x++)
                    CurrentZIndicatorBuffer[z][x] = scanState;
            }
        }

        public void UpdateViewer(byte[] ImageXY_byte, double DepthReal, int xmax, int ymax)
        {
            // Z-position
            if (InvokeRequired)
                BeginInvoke(new Action(delegate
                {
                    label_z_position.Text = ((int)DepthReal).ToString();
                    PlaneBmpBuffer =
ImageProjection.XY_Projection.ConvertPlaneToBitmap_Unsafe(ImageXY_byte, xmax, ymax,
LUTsMaxIntensity);
                    ImageXY = PlaneBmpBuffer;
                }));
            else
            {
                label_z_position.Text = ((int)DepthReal).ToString();
                PlaneBmpBuffer =
ImageProjection.XY_Projection.ConvertPlaneToBitmap_Unsafe(ImageXY_byte, xmax, ymax,
LUTsMaxIntensity);
                ImageXY = PlaneBmpBuffer;
            }

            string test = "";
            if (false && PlaneBmpBuffer != null)
            {
                for (int y = 0; y < ymax; y++)
                {
                    test += "¥n";
                    for (int x = 0; x < xmax; x++)
                        test += $"{(int)PlaneBmpBuffer.GetPixel(x, y).R}, ";
                }
                Console.WriteLine(test);
            }
```

```
            //Stopwatch sw = new Stopwatch();
            //sw.Start();
            // update viewer
            ImageXZ = ImageProjection.XY_Projection.ConvertPlaneToBitmap_Unsafe(xzBuffer,
LUTsMaxIntensity);
            //ImageYZ =
ImageProjection.XY_Projection.ConvertPlaneToBitmap_Unsafe(yzBuffer_transpose,
LUTsMaxIntensity);
            var yz_transposed =
ImageProjection.XY_Projection.ConvertPlaneToBitmap_Unsafe(yzBuffer, LUTsMaxIntensity);
            yz_transposed.RotateFlip(RotateFlipType.Rotate90FlipX);
            ImageYZ = yz_transposed;
            ImageZIndicator =
ImageProjection.XY_Projection.ConvertPlaneToBitmap_Unsafe(CurrentZIndicatorBuffer);
            //sw.Stop();
            //Console.WriteLine($"projection calc time: {sw.ElapsedTicks} [ms]");
        }
        private void TimerLiveUpdate_Tick(object sender, EventArgs e)
        {
        }

        #region Test Volume Update
        private void testUpdateTimer_Tick(object sender,
HighResolutionTimerElapsedEventArgs e)
        {
            // Update present z-position (px); Valid only in the test, whole volume already
exist
            ZPositionPxPrev = ZPositionPx;
            ZPositionPx += ZStridePxPerUpdate;
            if (ZPositionPx >= VolumeSizePixel.Z)
                ZPositionPx = ZPositionPx % (int)VolumeSizePixel.Z;

            // Indicate Z-position
            if(InvokeRequired)
                BeginInvoke(new Action(delegate
                {
                    label_z_position.Text = ZPositionPx.ToString()+" ";
                    PlaneBmpBuffer =
TestProjection.ConvertPlaneToBitmap(ExampleVolume[ZPositionPx]);
                    pictureBox_xy.Image = PlaneBmpBuffer;
                }));

            // "ZPositionPxPrev" is the last index processed. So +1 here
            int unprocessedZBegin = (ZPositionPxPrev + 1) % (int)(VolumeSizePixel.Z);
            bool rangeIsSeparated = false;
            if (unprocessedZBegin >= ZPositionPx)  // if crossing MaxLength
                rangeIsSeparated = true;

            int zPx = unprocessedZBegin;
            while (true)
            {
                if (!rangeIsSeparated && zPx > ZPositionPx)
                    break;
                // Firstly, process "ZBegin to MaxLength".
                // Next (rangeIsSeparated <= false), process "0 to ZPosPx".
                if (rangeIsSeparated && zPx >= (int)VolumeSizePixel.Z)
                {
                    zPx = 0;
                    rangeIsSeparated = false;
                }

                // data from example volume
                xy_Projection.SetImageFromUInt16Array(ExampleVolume[zPx]);

                (ushort[] xz, ushort[] yz) result;
                // Check the Projection Method
                switch (liveProjBuffer.projMethod)
                {
                    case ImageProjection.ProjectionMethods.MaxIntensity:
```

```
                    result = xy_Projection.GetMaxIntensityProjection();
                    break;
                case ImageProjection.ProjectionMethods.AtThePoint:
                    result = xy_Projection.GetSliceProjectionAtPoint();
                    break;
                default:
                    result = xy_Projection.GetMaxIntensityProjection();
                    break;
            }

            // Update 2D buffers
            xzBuffer[zPx] = result.xz;  // result.xz is 1-dim data actually
            yzBuffer[zPx] = result.yz;  // same
            // For displaying YZ image, use transposed yzBuffer
            //for (int y = 0; y < yzBuffer_transpose.Length; y++)
            //    yzBuffer_transpose[y][zPx] = result.yz[y];

            // z-position indicator (Zero or Max)
            ushort scanState = 0;
            if (CurrentZIndicatorBuffer[zPx][0] == 0)
                scanState = 65535;  // 16 bit
            for (int x = 0; x < CurrentZIndicatorWidthPx; x++)
                CurrentZIndicatorBuffer[zPx][x] = scanState;

            // increment
            zPx++;
        }

        // update viewer
        ImageXZ = TestProjection.ConvertPlaneToBitmap(xzBuffer);
        var yz_temp = TestProjection.ConvertPlaneToBitmap(yzBuffer);
        yz_temp.RotateFlip(RotateFlipType.Rotate90FlipX);
        ImageYZ = yz_temp;
        ImageZIndicator = TestProjection.ConvertPlaneToBitmap(CurrentZIndicatorBuffer);
    }

    private void button1_Click(object sender, EventArgs e)
    {
        timerTestUpdate.Stop();

        ExampleVolume = proj.CreateRandom3DImage(
            (int)VolumeSizePixel.X, (int)VolumeSizePixel.Y, (int)VolumeSizePixel.Z);
        ZPositionPx = 0;

        timerTestUpdate.Start();
    }
    #endregion

    /// <summary>
    /// Indicate current z-position.
    /// </summary>
    private void UpdateZIndicator()
    {

    }
    private void radioButton_slice_CheckedChanged(object sender, EventArgs e)
    {
        if (radioButton_mip.Checked)
            liveProjBuffer.projMethod = ImageProjection.ProjectionMethods.MaxIntensity;
        else if (radioButton_slice.Checked)
            liveProjBuffer.projMethod = ImageProjection.ProjectionMethods.AtThePoint;
        else
            liveProjBuffer.projMethod = ImageProjection.ProjectionMethods.MaxIntensity;
    }
    public void UpdateParamsView()
    {
    }

    public void ShowVoxelBufferParamsLabel()
```

```
        {
            Size3D settingSize3D = OPForm.GetVolumeSizeForProjection();
            Size3D size = new Size3D();
            if(liveProjBuffer != null)
                size = liveProjBuffer.VoxelSize;
            else
            {
                size.X = (double)numericUpDown_xy_voxel_size.Value;
                size.Y = (double)numericUpDown_xy_voxel_size.Value;
                size.Z = (double)numericUpDown_z_voxel_size.Value;
            }

            Size3D voxelNumbers = new Size3D(
                (double)(settingSize3D.X / size.X),
                (double)(settingSize3D.Y / size.Y),
                (double)(settingSize3D.Z / size.Z));
            double totalVoxels = (voxelNumbers.X * voxelNumbers.Y * voxelNumbers.Z) * 1e-9;

            if (InvokeRequired)
                BeginInvoke(new Action(delegate
                {
                    numericUpDown_xy_voxel_size.Value = (decimal)size.X;
                    numericUpDown_z_voxel_size.Value = (decimal)size.Z;
                    label_total_voxels.Text = $"{totalVoxels:F3} M [voxels]";
                }));
            else
            {
                numericUpDown_xy_voxel_size.Value = (decimal)size.X;
                numericUpDown_z_voxel_size.Value = (decimal)size.Z;
                label_total_voxels.Text = $"{totalVoxels:F3} M [voxels]";
            }
        }

        private void numericUpDown_luts_max_ValueChanged(object sender, EventArgs e)
        {
            this.LUTsMaxIntensity = (int)numericUpDown_luts_max.Value;
        }

        private void numericUpDown_update_interval_ValueChanged(object sender, EventArgs e)
        {
            UpdateIntervalShouldBe_ms = (int)numericUpDown_update_interval.Value;
        }

        private void numericUpDown_slice_x_ValueChanged(object sender, EventArgs e)
        {
            int x0 = (int)numericUpDown_slice_x.Value;
            int y0 = (int)numericUpDown_slice_y.Value;

            if(ImageXY.Width > x0)
                xy_Projection.x0_selected = x0;
            if(ImageXY.Height > y0)
                xy_Projection.y0_selected = y0;

            Bitmap canvas = new Bitmap(pictureBox_xy.Width, pictureBox_xy.Height);
            Graphics g = Graphics.FromImage(canvas);
            g.DrawLine(Pens.Red, x0, 0, x0, pictureBox_xy.Height);
            g.DrawLine(Pens.Red, 0, y0, pictureBox_xy.Width, y0);
            g.Dispose();
            pictureBox_xy_transparent.Image = canvas;
        }

        private void pictureBox_xy_SizeChanged(object sender, EventArgs e)
        {
            pictureBox_xy_transparent.Location = pictureBox_xy.Location;
            pictureBox_xy_transparent.Size = pictureBox_xy.Size;
        }

        #region Helper for Directions related to Stage
        public void ApplyMagnificationParams()
```

```
        {
            comboBox_magnification_SelectedIndexChanged(null, EventArgs.Empty);
        }
        private void comboBox_magnification_SelectedIndexChanged(object sender, EventArgs
e)
        {
            double mag, magAxial;
            double.TryParse(comboBox_magnification.Text, out mag);
            double.TryParse(comboBox_axial_zoom.Text, out magAxial);
            if (mag == 0) mag = 1;
            if (magAxial == 0) magAxial = 1;
            AdjustControlSizeAndLocation(mag, magAxial);
        }
        private void SetDirectionLabels(string LUp, string RUp, string RDown, string LDown)
        {
            if (InvokeRequired)
            {
                BeginInvoke(new Action(delegate
                {
                    label_pos_lup.Text = LUp;
                    label_pos_ldown.Text = LDown;
                    label_pos_rup.Text = RUp;
                    label_pos_rdown.Text = RDown;
                }));
            }
            else
            {
                label_pos_lup.Text = LUp;
                label_pos_ldown.Text = LDown;
                label_pos_rup.Text = RUp;
                label_pos_rdown.Text = RDown;
            }
        }
        /// <summary>
        /// Indicate the Rotation/Flip status set in "CameraSetting Form" to this form
        /// </summary>
        /// <param name="type"></param>
        public void TransformDirectionsRelatedToStageHelper(RotateFlipType type)
        {
            const string LU = "L.Up";
            const string LD = "L.Down";
            const string RU = "R.Up";
            const string RD = "R.Down";

            switch (type)
            {
                case RotateFlipType.RotateNoneFlipNone:
                    SetDirectionLabels(LU, RU, RD, LD);
                    break;
                case RotateFlipType.Rotate90FlipNone:
                    SetDirectionLabels(LD, LU, RU, RD);
                    break;
                case RotateFlipType.Rotate180FlipNone:
                    SetDirectionLabels(RD, LD, LU, RU);
                    break;
                case RotateFlipType.Rotate270FlipNone:
                    SetDirectionLabels(RU, RD, LD, LU);
                    break;

                case RotateFlipType.RotateNoneFlipX:
                    SetDirectionLabels(RU, LU, LD, RD);
                    break;
                case RotateFlipType.Rotate90FlipX:
                    SetDirectionLabels(LU, LD, RD, RU);
                    break;

                case RotateFlipType.RotateNoneFlipY:
                    SetDirectionLabels(LD, RD, RU, LU);
                    break;
```

```
                case RotateFlipType.Rotate90FlipY:
                    SetDirectionLabels(RD, RU, LU, LD);
                    break;
            }
        }
        #endregion

        private void numericUpDown_xy_voxel_size_ValueChanged(object sender, EventArgs e)
        {
            Size3D size = new Size3D(
                (double)numericUpDown_xy_voxel_size.Value,
                (double)numericUpDown_xy_voxel_size.Value,
                (double)numericUpDown_z_voxel_size.Value);
            if(liveProjBuffer != null)
                liveProjBuffer.VoxelSize = size;
            ShowVoxelBufferParamsLabel();
        }
        public Size3D GetVoxelSizeSet()
        {
            Size3D size = new Size3D(
                (double)numericUpDown_xy_voxel_size.Value,
                (double)numericUpDown_xy_voxel_size.Value,
                (double)numericUpDown_z_voxel_size.Value);
            return size;
        }
    }
}
```

Code A12. LiveImaging/ImageProjection.cs

```
using Hamamatsu.DCAM4;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace ETL_system
{
    public partial class ImageDisplay : Form
    {
        private BufferedGraphicsContext bufferContext;
        private BufferedGraphics buffer;
        // People say that Forms.Timer is better than System.Timers.Timer if use in Form.
Probably true.
        private System.Windows.Forms.Timer timer;
        private System.Windows.Forms.Timer timer_RefleshRate;

        private int counter = 0;    // measure refresh rate
        private float targetFPS;

        private float prevImgScaling = 0;
        private DCAMLut lut;
        private RotateFlipType rotationFlipType;
        private string rotationFlipLable;

        private const double ScrollBarMarginRatio = 0.20;

        public Graphics Buffer { get => buffer.Graphics; }
        /// <summary>
        /// For showing live images in PictureBox. Intensity has scaled by LUTs setup.
        /// </summary>
        public Image NextImage { get; set; }
        /// <summary>
        /// This Image has an original, unscaled intensities (8 bit RGB).
        /// </summary>
        public Image OriginalImage { get; set; }
        /// <summary>
        /// Simple byte array of original image.
        /// </summary>
        public byte[] OriginalImageBuffer { get; set; }

        /// <summary>
        /// If true, displayed image is updated. Set by OPForm, "CameraUpdatePicture_Async"
        /// </summary>
        public bool FlagImageUpdated { get; set; }
        /// <summary>
        /// Size of Image that showing in PictureBox.
        /// </summary>
        public Size ImageResolution { get; set; }
        /// <summary>
        /// Indicates size scaling ratio, PictureBox vs OriginalImage.
        /// </summary>
        public float ImageScaling { get; private set; }
        public int BitDepth { get; set; }
        public int MaxIntensity { get; set; }
        public DCAMLut LUT { get => lut; set { lut = value; } }
        /// <summary>
        /// Rotation angle (clock wise; [deg]) and flip/mirror type of displayed image
        /// </summary>
```

```
        public RotateFlipType RotationFlipType
        {
            get => rotationFlipType;
            set
            {
                rotationFlipType = value;
            }
        }
        public string RotationLabel
        {
            get => rotationFlipLable;
            set
            {
                rotationFlipLable = value;
                this.Invoke((MethodInvoker)(() => label_rotation.Text = value));
            }
        }

        /// <summary>
        /// For updating NextImage by multi-thread
        /// </summary>
        public object LockObj { get; }

        public ImageDisplay()
        {
            InitializeComponent();
            pictureBox.CreateGraphics().InterpolationMode =
System.Drawing.Drawing2D.InterpolationMode.HighQualityBicubic;
            LockObj = new object();

            bufferContext = BufferedGraphicsManager.Current;
            buffer = bufferContext.Allocate(pictureBox.CreateGraphics(),
pictureBox.DisplayRectangle);

            // temporary
            timer = new System.Windows.Forms.Timer();
            timer.Interval = 33;
            timer.Tick += new EventHandler(timer_Tick);

            // measure reflesh rate
            timer_RefleshRate = new System.Windows.Forms.Timer();
            timer_RefleshRate.Interval = 1000;
            timer_RefleshRate.Tick += new EventHandler(timer_RefleshRate_Tick);

            // TODO: Take info from camera or SettingForm
            BitDepth = 16;
            MaxIntensity = 65535;

            rotationFlipType = RotateFlipType.RotateNoneFlipNone;

            // temporary
            numericUpDown_scrollbar_max_limit_ValueChanged(null, EventArgs.Empty);
            lut = new DCAMLut();
            lut.camerabpp = BitDepth;
            lut.cameramax = MaxIntensity;
            lut.inmax = 230;  // set max first bacause (min, max) begins from (0, 0)
            lut.inmin = 70;
        }

        public void StartLiveTimer(int fps = 30)
        {
            // init labels and so on
            label_frame_number.Text = "Frame Number";

            // clear old images (?)
            NextImage = null;
            OriginalImage = null;

            counter = 0;
```

```
            timer.Interval = (int)(1000.0 / fps);
            targetFPS = fps;
            timer.Start();
            timer_RefleshRate.Start();
        }
        public void StopLiveTimer()
        {
            timer.Stop();
            timer_RefleshRate.Stop();
        }
        private void timer_Tick(object sender, EventArgs e)
        {
            if (FlagImageUpdated)
            {
                // temporarily use timer for live imaging
                pictureBox.Image = NextImage;
                if(NextImage != null)
                    pictureBox.Image.RotateFlip(RotationFlipType);
                FlagImageUpdated = false;
            }
            counter++;

            UpdateImagePropertyLabels_Async();
        }
        private void timer_RefleshRate_Tick(object sender, EventArgs e)
        {

            float rate = (float)(counter * (1000.0 / timer_RefleshRate.Interval));
            counter = 0;

            // check current FPS and adjust timer
            if (Math.Abs(rate - targetFPS) > 2)
            {
                int interval = timer.Interval;
                if (rate > targetFPS)
                    interval += 2;
                else if (rate < targetFPS && timer.Interval > 1)
                    interval -= 2;

                timer.Interval = 1;
                timer.Stop();
                timer.Interval = interval;
                timer.Start();
            }

            BeginInvoke(new Action(delegate
            {
                label_reflesh_rate.Text = string.Format("Reflesh Rate: {0:#.#} fps
(interval={1})", rate, timer.Interval);
            }));
        }

        private void Paint(object sender, PaintEventArgs e)
        {
            if (buffer != null)
            {
                //buffer.Graphics.Clear(pictureBox.BackColor);
                //buffer.Graphics.DrawImageUnscaled(, new Point(0, 0));
                buffer.Render();
            }
        }

        private void PicturePanel_Load(object sender, EventArgs e)
        {
            numericUpDown_max.Maximum = lut.cameramax;
            numericUpDown_min.Maximum = lut.cameramax;
            hScrollBar_max.Maximum = lut.cameramax;
            hScrollBar_min.Maximum = lut.cameramax;
```

```
            SetLUTs(lut.inmin, lut.inmax);
        }

        public void SetLUTs(int min, int max)
        {
            BeginInvoke(new Action(delegate
            {
                lut.inmin = min;
                lut.inmax = max;
                numericUpDown_min.Value = min;
                numericUpDown_max.Value = max;
                //hScrollBar_min.Value = min;
                //hScrollBar_max.Value = max;
            }));
        }

        public void UpdateImage(Image img)
        {
            buffer.Graphics.Clear(pictureBox.BackColor);
            buffer.Graphics.InterpolationMode =
System.Drawing.Drawing2D.InterpolationMode.HighQualityBicubic;
            buffer.Graphics.DrawImageUnscaled(img, new Point(0, 0));
            buffer.Render();
        }

        public void UpdateImagePropertyLabels_Async()
        {
            if (NextImage == null) return;
            // ImageScaling is updated in TransformPosition function.
            //ImageScaling = pictureBox.Size.Width / (float)NextImage.Width;
            //if (ImageScaling == prevImgScaling) return;
            ImageResolution = NextImage.Size;

            Task task = Task.Run(() =>
            {
                prevImgScaling = ImageScaling;
                BeginInvoke(new Action(delegate
                {
                    label_img_resolution.Text = string.Format(
                    "{0}x{1}", ImageResolution.Width, ImageResolution.Height);
                    label_scaling.Text = string.Format("Display Scaling: {0:#.#} %",
ImageScaling * 100);
                }));
            });
        }

        private void PicturePanel_ResizeEnd(object sender, EventArgs e)
        {

        }

        private void checkBox_auto_luts_CheckedChanged(object sender, EventArgs e)
        {
            //if (pictureBox.Image == null) return;
            //int min, max;
            //(min, max) = Utils.GetMinMaxIntensityOfImage((Bitmap)NextImage, MaxIntensity);

            //lut.inmax = max;
            //lut.inmin = min;
            //SetLUTs(min, max);

            // TODO: apply this lut to the image. (may be thrown to OpForm)
        }

        private void UpdateIntensityLabel(Point Pos, bool PosOnImage = false)
        {
            if (OriginalImage == null || OriginalImageBuffer == null)
                return;
```

```
            Point p = Pos;
            if (!PosOnImage)
                p = TransformPosition(Pos);

            if (p.X >= 0 && p.Y >= 0)
            {
                var src = OriginalImageBuffer;
                int w = this.pictureBox.Image.Width;
                int h = this.pictureBox.Image.Height;

                int bytePerPixels = LUT.camerabpp / 8;
                int stride = w * bytePerPixels;
                int currentPosIndex = stride * p.Y + bytePerPixels * p.X;

                UInt16 intensity = 0;
                if(LUT.camerabpp > 8)
                    // Single data from 2-byte
                    intensity = BitConverter.ToUInt16(src, currentPosIndex);
                else
                    // Single data from 1-byte (does not exist ToUInt8() function)
                    intensity = Convert.ToUInt16(src[currentPosIndex]);

                //var color = ((Bitmap)OriginalImage).GetPixel(p.X, p.Y);
                //int intensity = (int)(color.GetBrightness() * (float)MaxIntensity);    //
TODO: maybe not correct
                //int intensity = (int)(color.GetBrightness() * this.LUT.inmax);    // TODO:
maybe not correct
                label_xy_intensity.Text = string.Format("({0}, {1}) = {2}", p.X, p.Y,
intensity);
            }
        }
        public void UpdateFrameNumber(int iFrame, int maxFrame)
        {
            if (InvokeRequired)
            {
                Invoke(new Action(() =>
                {
                    label_frame_number.Text = string.Format("Frames: {0} / {1}", iFrame,
maxFrame);
                }));
            }
            else
                label_frame_number.Text = string.Format("Frames: {0} / {1}", iFrame,
maxFrame);
        }

        /// <summary>
        /// Transform position on pictureBox to position on Image. This function consider
PictureBox.SizeMode == Zoom.
        /// </summary>
        /// <returns>Position on Image</returns>
        private Point TransformPosition(Point PosOnControl)
        {
            Point newPos = new Point();
            if (pictureBox.SizeMode == PictureBoxSizeMode.Zoom)
            {
                // Get coordinate that mouse positioned, for 'Zoom' mode PictureBox
                double ratioPicBox = pictureBox.Width / (double)pictureBox.Height;
                double ratioImage = pictureBox.Image.Width /
(double)pictureBox.Image.Height;

                double scaling, paddingOneSide;
                if (ratioPicBox > ratioImage)
                {
                    // fitted vertically
                    scaling = pictureBox.Height / (double)pictureBox.Image.Height;
                    // check width of PicBox
                    paddingOneSide = (pictureBox.Width - (double)pictureBox.Image.Width *
```

```
scaling) / 2.0;

                    newPos.X = (int)((PosOnControl.X - paddingOneSide)
                        * (pictureBox.Image.Width / (double)(pictureBox.Size.Width -
paddingOneSide * 2.0)));
                    newPos.Y = (int)(PosOnControl.Y * (pictureBox.Image.Height /
(double)pictureBox.Size.Height));
                }
                else
                {
                    // fitted horizontally
                    scaling = pictureBox.Width / (double)pictureBox.Image.Width;
                    // check height of PicBox
                    paddingOneSide = (pictureBox.Height - (double)pictureBox.Image.Height *
scaling) / 2.0;

                    newPos.X = (int)(PosOnControl.X * (pictureBox.Image.Width /
(double)pictureBox.Size.Width));
                    newPos.Y = (int)((PosOnControl.Y - paddingOneSide)
                        * (pictureBox.Image.Height / (double)(pictureBox.Size.Height -
paddingOneSide * 2.0)));
                }
                ImageScaling = (float)scaling;
            }
            else
            {
                // for 'StretchImage' and 'AutoSize'
                newPos.X = (int)(PosOnControl.X * (pictureBox.Image.Width /
(double)pictureBox.Size.Width));
                newPos.Y = (int)(PosOnControl.Y * (pictureBox.Image.Height /
(double)pictureBox.Size.Height));
            }

            if (newPos.X >= pictureBox.Image.Width) newPos.X = pictureBox.Image.Width - 1;
            if (newPos.Y >= pictureBox.Image.Height) newPos.Y = pictureBox.Image.Height - 1;

            return newPos;
        }

        /// <summary>
        /// Show intensity of pixel that mouse positioned
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void pictureBox_MouseMove(object sender, MouseEventArgs e)
        {
            if (pictureBox.Image == null) return;
            UpdateIntensityLabel(new Point(e.X, e.Y), false);
        }

        private void pictureBox_Paint(object sender, PaintEventArgs e)
        {
            if (pictureBox.Image == null) return;
            Point p = pictureBox.PointToClient(Cursor.Position);
            UpdateIntensityLabel(new Point(p.X, p.Y), false);
        }

        private void PicturePanel_FormClosing(object sender, FormClosingEventArgs e)
        {
            e.Cancel = true;
            this.Hide();
        }

        #region LUTs Setup UI
        private void hScrollBar_max_ValueChanged(object sender, EventArgs e)
        {
            int val = Math.Min(hScrollBar_max.Value,
(int)((double)numericUpDown_max.Value*(1+ScrollBarMarginRatio)));
            numericUpDown_max.Value = hScrollBar_max.Value;
```

```
            lut.inmax = (int)(numericUpDown_max.Value);
        }
        private void hScrollBar_min_ValueChanged(object sender, EventArgs e)
        {
            numericUpDown_min.Value = hScrollBar_min.Value;
            lut.inmin = (int)(numericUpDown_min.Value);
        }
        private void numericUpDown_max_ValueChanged(object sender, EventArgs e)
        {
            if (numericUpDown_max.Value > hScrollBar_max.Maximum)
                hScrollBar_max.Maximum = (int)(numericUpDown_max.Value + 1);

            if (numericUpDown_max.Value < numericUpDown_min.Value)
                numericUpDown_max.Value = numericUpDown_min.Value;

            //hScrollBar_max.Maximum = (int)numericUpDown_max.Value;
            //hScrollBar_max.Value = (int)numericUpDown_max.Value;
            lut.inmax = (int)(numericUpDown_max.Value);

            return;

            // better UI for dynamic range (2022 Jan)
            int newlimit = (int)((double)numericUpDown_max.Value *
(1+ScrollBarMarginRatio));
            int lowerlimit = (int)numericUpDown_min.Value;
            newlimit = Math.Min(65535, newlimit);
            newlimit = Math.Max(newlimit, lowerlimit);
            hScrollBar_max.Maximum = newlimit;
            hScrollBar_min.Maximum = newlimit;
            numericUpDown_scrollbar_max_limit.Value = newlimit;
        }
        private void numericUpDown_min_ValueChanged(object sender, EventArgs e)
        {
            if (numericUpDown_min.Value > hScrollBar_min.Minimum)
                hScrollBar_min.Minimum = (int)(numericUpDown_min.Value - 1);

            if (numericUpDown_max.Value < numericUpDown_min.Value)
                numericUpDown_min.Value = numericUpDown_max.Value;

            //hScrollBar_min.Minimum = (int)numericUpDown_min.Value;
            //hScrollBar_min.Value = (int)numericUpDown_min.Value;
            lut.inmin = (int)(numericUpDown_min.Value);

            return;

            // better UI for dynamic range (2022 Jan)
            int newlimit = (int)((double)numericUpDown_min.Value * (1-
ScrollBarMarginRatio));
            int higherlimit = (int)numericUpDown_max.Value;
            newlimit = Math.Max(0, newlimit);
            newlimit = Math.Min(newlimit, higherlimit);
            hScrollBar_max.Minimum = newlimit;
            hScrollBar_min.Minimum = newlimit;
            //numericUpDown_scrollbar_max_limit.Value = newlimit;
        }
        private void numericUpDown_scrollbar_max_limit_ValueChanged(object sender,
EventArgs e)
        {
            hScrollBar_max.Maximum = (int)(numericUpDown_scrollbar_max_limit.Value);
            hScrollBar_min.Maximum = (int)(numericUpDown_scrollbar_max_limit.Value);
        }
        #endregion
    }
}
```

Code A13. SerialPort/SerialPortConsole.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Timers;

namespace ETL_system
{
    /// <summary>
    /// Interactive console form. Can send string or bytes.
    /// </summary>
    public partial class SerialPortConsole : Form
    {
        SerialPortUtilsEx spuex;
        string selected_port_id;
        public SerialPortConsole(ref SerialPortUtilsEx SPUEx)
        {
            InitializeComponent();
            spuex = SPUEx;

            // Moved to VisibleChanged event
            //spuex.InteractiveEvent += new
SerialPortUtilsEx.InteractiveEventHandler(CallBackDataReceivedEvent);
        }

        public string SelectedPortID { get => selected_port_id; }
        public SerialPortExtended SelectedPortInstance { get { return
spuex.GetSerialPortInstanceFromCOMID(SelectedPortID); } }
        public SerialPortUtilsEx SerialPortUtilsExInstance { get { return spuex; } set
{spuex=value; } }

        private void SerialPortConsole_Load(object sender, EventArgs e)
        {
            spuex.GetAndSetPorts(comboBox_port);
            foreach (string s in Enum.GetNames(typeof(SerialPortExtended.DeviceTypeEnum)))
                comboBox_device_type.Items.Add(s);
            if (comboBox_device_type.Items.Count > 0)
                comboBox_device_type.SelectedIndex = 0;
        }

        private void Button_send_Click(object sender, EventArgs e)
        {
            if(SelectedPortInstance != null)
            {

SelectedPortInstance.ApplyDeviceType((string)comboBox_device_type.SelectedItem);
                SelectedPortInstance.IsInteractiveConsole = true;
                SelectedPortInstance.InteractiveConsoleEvent -= CallBackDataReceivedEvent;
                SelectedPortInstance.InteractiveConsoleEvent += CallBackDataReceivedEvent;

                // Send string
                if ((Button)sender == button_send_string)
                {
                    if (SelectedPortInstance.OpenPort())
                    {
                        SelectedPortInstance.Write(textBox_command_string.Text);
                        if (checkBox_append_eof.Checked)
                            SelectedPortInstance.Write(new byte[]
{ (byte)SelectedPortInstance.WritingEOF }, 0, 1);
```

```
                    textBox_display.Text += "[S]" + textBox_command_string.Text + "[/S]
(str)" + Environment.NewLine;
                }
                else
                    textBox_display.Text += "Failed to send message (string)." +
Environment.NewLine;
            }

            // Send 2 byte hex
            else if ((Button)sender == button_send_bytes)
            {
                byte[] bytes = new byte[textBox_command_bytes.Text.Length / 2];
                for (int i = 0; i < bytes.Length; i++)
                {
                    //Console.Write(textBox_command_bytes.Text.Substring(i * 2, 2)+" ");
                    byte b = Convert.ToByte(textBox_command_bytes.Text.Substring(i * 2,
2), 16);
                    bytes[i] = b;
                }

                if (SelectedPortInstance.OpenPort())
                {
                    SelectedPortInstance.Write(bytes, 0, bytes.Length);
                    if (checkBox_append_eof.Checked)
                        SelectedPortInstance.Write(new byte[]
{ (byte)SelectedPortInstance.WritingEOF }, 0, 1 );

                    string str = "";
                    for (int i = 0; i < bytes.Length; i++)
                        str += bytes[i].ToString("X2");
                    textBox_display.Text += string.Format("[S]{0}[/S] (hex)" +
Environment.NewLine, str);
                }
                else
                    textBox_display.Text += "Failed to send message (byte)." +
Environment.NewLine;
            }

            // cursor position
            TextBoxMoveCursorToEnd(textBox_display);
        }
        else
        {
            Console.WriteLine("Not connected.");
        }
    }

    private void SerialPortConsole_FormClosing(object sender, FormClosingEventArgs e)
    {
        if (SelectedPortInstance != null)
        {
            SelectedPortInstance.IsInteractiveConsole = false;
            SelectedPortInstance.Close();  // To use the port from other Form
        }
        e.Cancel = true;
        this.Hide();
    }

    private void
CallBackDataReceivedEvent(SerialPortExtended.InteractiveConsoleEventArgs e)
    {
        byte[] data = e.Data;
        bool clearBuffer = e.CanClearBuffer;
        try
        {
            if (data.Length > 0)
            {
                string str = BitConverter.ToString(data);
                Console.Write(str);
```

```
                    Invoke(new Action(delegate
                    {
                        if(checkBox_hex.Checked)
                            textBox_display.Text += "[R]" + str + "[/R] (hex)" +
Environment.NewLine;
                        if (checkBox_ascii.Checked)
                            textBox_display.Text += "[R]" + Utils.ConvertHex(str, sep: "-
").Replace("¥n", "¥¥n") + "[/R] (str)" + Environment.NewLine;
                        TextBoxMoveCursorToEnd(textBox_display);
                    }));
                }
            }
            finally
            {
                if (clearBuffer)
                {
                    spuex.ClearExistingData();
                }
                SelectedPortInstance.InteractiveConsoleEvent -= CallBackDataReceivedEvent;
            }
        }

        private void TextBox_display_TextChanged(object sender, EventArgs e)
        {
        }

        private void ComboBox_ports_SelectedIndexChanged(object sender, EventArgs e)
        {
            selected_port_id = spuex.PortsDic[comboBox_port.Text];
            label_selected_port.Text = selected_port_id;
        }

        private void button_close_ports_Click(object sender, EventArgs e)
        {
            foreach(var spe in spuex.SerialPortList)
            {
                if (spe.IsOpen)
                {
                    spe.ReadExisting();
                    spe.Close();
                }
            }
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (SelectedPortInstance.IsOpen)
            {
                string data = SelectedPortInstance.ReadExisting();
                Console.WriteLine(data);
            }
        }

        private void TextBoxMoveCursorToEnd(TextBox tb)
        {
            if(tb.InvokeRequired)
            {
                Invoke(new Action(delegate
                {
                    TextBoxMoveCursorToEnd(tb);
                }));
            }
            else
            {
                tb.SelectionStart = tb.Text.Length;
                tb.Focus();
                tb.ScrollToCaret();
            }
        }
```

```
        private void SerialPortConsole_VisibleChanged(object sender, EventArgs e)
        {
            if(this.Visible)
                SelectedPortInstance.InteractiveConsoleEvent += CallBackDataReceivedEvent;
            else
                SelectedPortInstance.InteractiveConsoleEvent -= CallBackDataReceivedEvent;
        }
    }
}
```

Code A14. SerialPort/SerialPortExtended.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO.Ports;
using System.Timers;
using System.Collections;

namespace ETL_system
{
    /// <summary>
    /// Added some initial parameters to the SerialPort class for use of devices like ETL.
    /// </summary>
    public class SerialPortExtended : System.IO.Ports.SerialPort
    {
        protected byte[] received_data_buffer = Array.Empty<byte>();
        private EOFEnum reading_eof = EOFEnum.LF;
        private EOFEnum writing_eof = EOFEnum.LF;

        public enum DeviceTypeEnum
        {
            // Add specific device name here, and specific condition into Init().
            Any, ETL, CSUX, MAC6
        }
        DeviceTypeEnum device_type = DeviceTypeEnum.Any;

        public enum TestTypeEnum
        {
            None, Handshake, SampleCurrentCRC, SampleCurrentCalcCRC
        }
        TestTypeEnum test_type = TestTypeEnum.None;
        public enum EOFEnum : byte
        {
            CR = 0x0d,  // ¥r
            LF = 0x0a,  // ¥n
        }

        Timer timer_timeout;  // Not so accurate
        private int wait_time = 500;  // [ms]

        public string HANDSHAKE_SEND;
        public string HANDSHAKE_ANSWER;
        public string ANSWER_INVALID;

        public SerialPortExtended()
        {
        }
        public SerialPortExtended(string COM_ID)
        {
            Init(COM_ID, DeviceTypeEnum.Any);
        }
        public SerialPortExtended(string COM_ID, DeviceTypeEnum DeviceType)
        {
            Init(COM_ID, DeviceType);
        }
        private void Init(string COM_ID, DeviceTypeEnum DeviceType)
        {
            PortName = COM_ID;
            device_type = DeviceType;

            ApplyDeviceType(device_type);
        }
        public bool OpenPort()
        {
```

```
            try
            {
                for (int i = 0; i < 3; i++)
                {
                    if (IsOpen)
                        return true;
                    else
                    {
                        Close();
                        Open();
                    }
                }
            }
            catch(UnauthorizedAccessException ex)
            {
                Console.WriteLine(ex.Message);
                Close();
                return false;
            }
            return IsOpen;
        }

        #region Properties
        /// <summary>
        /// Device specific parameters will be used
        /// </summary>
        public DeviceTypeEnum DeviceType
        {
            get { return device_type; }
            set
            {
                device_type = value;
                ApplyDeviceType(value);
            }
        }
        public TestTypeEnum TestType { get { return test_type; } set { test_type =
value; } }
        /// <summary>
        /// NOTE: Can cast by byte.
        /// </summary>
        public EOFEnum ReadingEOF { get => reading_eof; set { reading_eof = value; } }
        /// <summary>
        /// NOTE: Can cast by byte.
        /// </summary>
        public EOFEnum WritingEOF { get => writing_eof; set { writing_eof = value; } }
        /// <summary>
        /// For SerialPort Console Form
        /// </summary>
        public bool IsInteractiveConsole { get; set; }
        /// <summary>
        /// For transfering the data read by DataReceived event to Form
        /// </summary>
        public bool IsGeneralUse { get;  set; }
        /// <summary>
        /// For Handshake Testing
        /// </summary>
        public bool IsTesting { get; set; }
        public bool IsLogging { get; set; }
        #endregion

        #region Comm Protocol and Event Setup
        /// <summary>
        /// Setup parameters of this port. After that, execute
UpdateDataReceivedEventHandler() in this function.
        /// </summary>
        /// <param name="DeviceType_str"></param>
        public void ApplyDeviceType(string DeviceType_str)
        {
            DeviceTypeEnum dev;
```

```
            if (Enum.TryParse(DeviceType_str, out dev) == false
                || !Enum.IsDefined(typeof(DeviceTypeEnum), dev))
                ApplyDeviceType(DeviceTypeEnum.Any);
            else
                ApplyDeviceType(dev);
        }
        /// <summary>
        /// Setup parameters of this port. After that, execute
UpdateDataReceivedEventHandler() in this function.
        /// </summary>
        /// <param name="DeviceType_str"></param>
        public void ApplyDeviceType(DeviceTypeEnum NewDeviceType)
        {
            device_type = NewDeviceType;
            switch (NewDeviceType)
            {
                // Set specific parameters.
                case DeviceTypeEnum.ETL:
                    // From official manual (Manual: Lens Driver 4)
                    BaudRate = 115200;
                    Parity = Parity.None;
                    StopBits = StopBits.One;
                    DataBits = 8;

                    // Not confirmed EOF byte. See Optotune's LensDriver.cs file

                    HANDSHAKE_SEND = "Start";
                    HANDSHAKE_ANSWER = "Ready¥r¥n";
                    ANSWER_INVALID = "N¥r¥n";

                    IsLogging = true;
                    IsInteractiveConsole = false;
                    break;

                case DeviceTypeEnum.CSUX:
                    // From CSUX_UserManual
                    BaudRate = 115200;  // [bps]
                    DataBits = 8;
                    Parity = Parity.None;
                    StopBits = StopBits.One;
                    NewLine = "¥r";  // not confirmed how NewLine work; use 0x0d to end a
single command

                    ReadingEOF = EOFEnum.CR;
                    WritingEOF = EOFEnum.CR;
                    break;

                case DeviceTypeEnum.MAC6:
                    // From
                    BaudRate = 9600;  // [bps]; max 115200
                    DataBits = 8;
                    Parity = Parity.None;
                    StopBits = StopBits.One;
                    NewLine = "¥n";  // not confirmed how NewLine work; use 0x0d to end a
single command

                    ReadingEOF = EOFEnum.LF;
                    WritingEOF = EOFEnum.CR;
                    break;
            }

            UpdateDataReceivedEventHandler();
        }
        public void UpdateDataReceivedEventHandler()
        {
            UpdateDataReceivedEventHandler(DeviceType);
        }
        public void UpdateDataReceivedEventHandler(DeviceTypeEnum NewDeviceType)
        {
```

```
            device_type = NewDeviceType;

            // At first, remove all candidate handlers
            DataReceived -= DataReceivedHandler_Any;
            DataReceived -= DataReceivedHandler_ETL;
            DataReceived -= DataReceivedHandler_General;

            // Register
            switch (NewDeviceType)
            {
                case DeviceTypeEnum.ETL:
                    DataReceived += new
SerialDataReceivedEventHandler(DataReceivedHandler_ETL);
                    break;
                case DeviceTypeEnum.CSUX:
                    DataReceived += DataReceivedHandler_General;
                    break;
                case DeviceTypeEnum.MAC6:
                    DataReceived += DataReceivedHandler_General;
                    break;
                default:
                    DataReceived += DataReceivedHandler_Any;
                    break;
            }
        }

        #endregion

        #region DataReceived EventHandlers
        /// <summary>
        /// Main EventHandler for using GUI Console.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void DataReceivedHandler_ETL(object sender, SerialDataReceivedEventArgs e)
        {
            SerialPortExtended spe = (SerialPortExtended)sender;
            if (!spe.IsOpen) return;

            if (spe.IsInteractiveConsole)
            {
                int len = spe.BytesToRead;
                byte[] buf = new byte[len];
                spe.Read(buf, 0, len);
                UpdateInteractiveProgress(buf, true);

                int offset = received_data_buffer.Length;
                Array.Resize(ref received_data_buffer, received_data_buffer.Length + len);
                Buffer.BlockCopy(buf, 0, received_data_buffer, offset, len);

                spe.IsInteractiveConsole = false;
            }

            if (spe.IsTesting)
            {
                string data = spe.ReadExisting();
                Console.Write(data);
                switch (spe.TestType)
                {
                    case SerialPortExtended.TestTypeEnum.Handshake:
                        if (data == spe.HANDSHAKE_ANSWER)
                            UpdateTestProgress(true);
                        else
                            UpdateTestProgress(false);
                        break;
                }
                spe.IsTesting = false;
            }
        }
```

```
        public void DataReceivedHandler_General(object sender, SerialDataReceivedEventArgs
e)
        {
            SerialPortExtended spe = (SerialPortExtended)sender;
            if (!spe.IsOpen) return;

            if (spe.IsInteractiveConsole || spe.IsGeneralUse)
            {
                ArrayList entireBuffer = new ArrayList();
                int len = spe.BytesToRead;
                while (len > 0)
                {
                    byte[] sub = new byte[1];
                    spe.Read(sub, 0, 1);
                    entireBuffer.Add(sub[0]);

                    if ((byte)entireBuffer[entireBuffer.Count - 1] == (byte)spe.ReadingEOF)
// check for ¥n =0x0a, ¥r=0x0d
                    {
                        // Export our arraylist into a byte array.
                        byte[] currentByteBuffer =
(byte[])entireBuffer.ToArray(typeof(byte));
                        int offset = received_data_buffer.Length;
                        Array.Resize(ref received_data_buffer, offset + len);
                        if (len > currentByteBuffer.Length)
                            len = currentByteBuffer.Length;  // testing..
                        Buffer.BlockCopy(currentByteBuffer, 0, received_data_buffer, offset,
len);

                        if (spe.IsInteractiveConsole)
                            spe.UpdateInteractiveProgress(currentByteBuffer, true);
                        else if (spe.IsGeneralUse)
                            spe.UpdateGeneralUseProgress(currentByteBuffer, true);

                        entireBuffer.Clear();
                        break;
                    }
                    if (entireBuffer.Count > 100)
                    {
                        break; //break while if more than 10 bytes were received
                    }
                }
                // not confirmed how flags work
                spe.IsInteractiveConsole = false;
                spe.IsGeneralUse = false;
            }
        }
        private void DataReceivedHandler_Any(object sender, SerialDataReceivedEventArgs e)
        {
            SerialPortExtended spe = (SerialPortExtended)sender;
            if (spe.IsOpen)
            {
                int len = spe.BytesToRead;
                byte[] buf = new byte[len];
                spe.Read(buf, 0, len);
                if (spe.IsInteractiveConsole)
                    UpdateInteractiveProgress(buf);
                else if (spe.IsGeneralUse)
                    UpdateGeneralUseProgress(buf);
            }
        }
        #endregion


        #region Events, Transfer received data to other Form
        //*** Interactive Console ***
        public event InteractiveConsoleEventHandler InteractiveConsoleEvent;
        public void UpdateInteractiveProgress(byte[] Data, bool CanClearBuffer = false)
```

```
        {
            InteractiveConsoleEvent?.Invoke(new InteractiveConsoleEventArgs(Data,
CanClearBuffer));
        }
        public delegate void InteractiveConsoleEventHandler(InteractiveConsoleEventArgs e);
        public class InteractiveConsoleEventArgs : EventArgs
        {
            byte[] data;
            bool clear_buf;
            public InteractiveConsoleEventArgs(byte[] Data, bool CanClearBuffer = false)
            {
                data = Data;
                clear_buf = CanClearBuffer;
            }
            public byte[] Data { get => data; }
            public bool CanClearBuffer { get => clear_buf; }
        }

        //*** General Use ***
        public event GeneralUseEventHandler GeneralUseEvent;
        public void UpdateGeneralUseProgress(byte[] Data, bool CanClearBuffer = false)
        {
            GeneralUseEvent(new GeneralUseEventArgs(Data, CanClearBuffer));
        }
        public delegate void GeneralUseEventHandler(GeneralUseEventArgs e);
        public class GeneralUseEventArgs : EventArgs
        {
            byte[] data;
            bool clear_buf;
            public GeneralUseEventArgs(byte[] Data, bool CanClearBuffer = false)
            {
                data = Data;
                clear_buf = CanClearBuffer;
            }
            public byte[] Data { get => data; }
            public bool CanClearBuffer { get => clear_buf; }
        }

        //*** Communication Test ***
        public event TestEventHandler TestEvent;
        public void UpdateTestProgress(bool Successed)
        {
            TestEvent(new TestEventArgs(Successed));
        }
        public delegate void TestEventHandler(TestEventArgs e);
        public class TestEventArgs : EventArgs
        {
            bool successed;
            public TestEventArgs(bool Successed)
            {
                successed = Successed;
            }
            public bool Successed { get { return successed; } }
        }
        #endregion
    }
}
```

Code A15. SerialPort/SerialPortUtils.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Management;
using System.Windows.Forms;
using System.IO.Ports;
using System.Timers;

namespace ETL_system
{
    /// <summary>
    /// Add to the SerialPort class: Events, Initialization, and Methods for pooling
received data.
    /// This class uses SerialPortExtended class.
    /// </summary>
    public class SerialPortUtils
    {
        protected Dictionary<string, string> ports_dic;
        protected List<SerialPortExtended> serialPortList;
        protected byte[] received_data_buffer = Array.Empty<byte>();

        System.Timers.Timer timer_timeout;  // No problem if not so accurate
        private int wait_time = 500;  // [ms]

        #region Props
        public Dictionary<string, string> PortsDic { get => ports_dic; }
        public List<SerialPortExtended> SerialPortList { get => serialPortList; }
        public SerialPortExtended TestingSerialPort { get; set; }
        /// <summary>
        /// Buffered received data (byte[]) from SerialPort.
        /// </summary>
        public byte[] ExistingData
        {
            get
            {
                if (received_data_buffer.Length == 0)
                    return Array.Empty<byte>();

                //string str = BitConverter.ToString(received_data_buffer);
                //str = str.Replace("-", string.Empty);
                return received_data_buffer;
            }
        }
        #endregion

        public SerialPortUtils()
        {
            serialPortList = new List<SerialPortExtended>();

            InitSerialPorts();

            // For communication test
            timer_timeout = new System.Timers.Timer();
            timer_timeout.Interval = wait_time;
            timer_timeout.Elapsed += Timer_timeout_Elapsed;
        }
        public virtual void InitSerialPorts()
        {
            // Get serial ports
            ports_dic = Utils.GetSerialPortList();
            foreach (string name in ports_dic.Keys)
            {
                serialPortList.Add(new SerialPortExtended(ports_dic[name]));
```

```
            }
        }
        public virtual void GetAndSetPorts(ComboBox comboBox_port)
        {
            comboBox_port.Items.Clear();

            foreach (var caption in ports_dic.Keys)
                comboBox_port.Items.Add(caption);

            if (ports_dic.Count > 0)
                comboBox_port.SelectedIndex = 0;
        }

        public SerialPortExtended GetSerialPortInstanceFromCOMID(string com_id)
        {
            foreach (var serial_port in serialPortList)
            {
                if (serial_port.PortName == com_id)
                    return serial_port;
            }
            Console.WriteLine(com_id + " does not exist.");
            return null;
        }

        public bool OpenPort(string com_id)
        {
            foreach(var port in serialPortList)
            {
                if (port.PortName == com_id)
                {
                    if (port.IsOpen)
                        return true;
                    else
                    {
                        port.Open();
                        return true;
                    }
                }
            }
            return false;
        }

        public void ClearExistingData()
        {
            // Clear buffer
            received_data_buffer = Array.Empty<byte>();
        }

        #region Comm Test (ETL)
        //***  Test button (Handshake) ***//
        public void TestCommunication(SerialPortExtended spe)
        {
            TestingSerialPort = spe;
            spe.IsTesting = true;
            timer_timeout.Interval = wait_time;  // [ms]

            spe.TestType = SerialPortExtended.TestTypeEnum.Handshake;

            //if (spe.IsOpen)
            if (spe.OpenPort())
            {
                switch (spe.TestType)
                {
                    case SerialPortExtended.TestTypeEnum.Handshake:
                        spe.Write(spe.HANDSHAKE_SEND);
                        break;
                    case SerialPortExtended.TestTypeEnum.SampleCurrentCalcCRC:
                        // TODO
                        break;
```

```
                    case SerialPortExtended.TestTypeEnum.SampleCurrentCRC:
                        // TODO
                        byte[] command = { 0x04, 0xb2, 0x26, 0x93 };

                        spe.Write("Aw");
                        spe.Write(command, 0, command.Length);
                        //serialPortETL.Write(optotune.SampleCurrentSetCommand, 0,
optotune.SampleCurrentSetCommand.Length);
                        break;
                }
                timer_timeout.Start();
            }
            else
            {
                Console.WriteLine("Could not open " + spe.PortName + ".");
            }
        }
        private void Timer_timeout_Elapsed(object sender, ElapsedEventArgs e)
        {
            timer_timeout.Stop();
            if (TestingSerialPort.IsTesting)
            {
                UpdateTestProgress(false);
                Console.WriteLine("Test was failed.");
            }
            TestingSerialPort.IsTesting = false;
            TestingSerialPort = null;
        }
        #endregion


        //***  EVENT  ***//
        #region Events, Transfer received data to other Form
        //*** Interactive Console ***
        public event InteractiveConsoleEventHandler InteractiveConsoleEvent;
        public void UpdateInteractiveProgress(byte[] Data, bool CanClearBuffer = false)
        {
            InteractiveConsoleEvent(new InteractiveConsoleEventArgs(Data, CanClearBuffer));
        }
        public delegate void InteractiveConsoleEventHandler(InteractiveConsoleEventArgs e);
        public class InteractiveConsoleEventArgs : EventArgs
        {
            byte[] data;
            bool clear_buf;
            public InteractiveConsoleEventArgs(byte[] Data, bool CanClearBuffer=false)
            {
                data = Data;
                clear_buf = CanClearBuffer;
            }
            public byte[] Data { get => data; }
            public bool CanClearBuffer { get => clear_buf; }
        }

        //*** General Use ***
        public event GeneralUseEventHandler GeneralUseEvent;
        public void UpdateGeneralUseProgress(byte[] Data, bool CanClearBuffer = false)
        {
            GeneralUseEvent(new GeneralUseEventArgs(Data, CanClearBuffer));
        }
        public delegate void GeneralUseEventHandler(GeneralUseEventArgs e);
        public class GeneralUseEventArgs : EventArgs
        {
            byte[] data;
            bool clear_buf;
            public GeneralUseEventArgs(byte[] Data, bool CanClearBuffer = false)
            {
                data = Data;
                clear_buf = CanClearBuffer;
            }
```

```
            public byte[] Data { get => data; }
            public bool CanClearBuffer { get => clear_buf; }
        }

        //*** Communication Test ***
        public event TestEventHandler TestEvent;
        public void UpdateTestProgress(bool Successed)
        {
            TestEvent(new TestEventArgs(Successed));
        }
        public delegate void TestEventHandler(TestEventArgs e);
        public class TestEventArgs : EventArgs
        {
            bool successed;
            public TestEventArgs(bool Successed)
            {
                successed = Successed;
            }
            public bool Successed { get { return successed; } }
        }
        #endregion


    }
}
```

Code A16. SerialPort/SerialPortUtilsEx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Management;
using System.IO.Ports;
using System.Timers;
using System.Windows.Forms;
using System.Collections;

namespace ETL_system
{
    /// <summary>
    /// Control ETL through the Optotune's USB Lens Driver 4i by using
Sytem.IO.Ports.SerialPort.
    /// </summary>
    public class SerialPortUtilsEx : SerialPortUtils
    {
        const string KEY_ETL_PORT = "optotune lens driver";

        public SerialPortUtilsEx()
        {
            InitSerialPorts();
        }
        public override void InitSerialPorts()
        {
            // Get serial ports
            ports_dic = Utils.GetSerialPortList();
            foreach (string name in ports_dic.Keys)
            {
                // For Optotune's ETL
                if (name.ToLower().IndexOf(KEY_ETL_PORT) >= 0)
                    serialPortList.Add(new SerialPortExtended(ports_dic[name],
SerialPortExtended.DeviceTypeEnum.ETL));
                else
                    serialPortList.Add(new SerialPortExtended(ports_dic[name]));
            }
        }
        public void UpdateDataReceivedEvents(SerialPortExtended SPE)
        {
            UpdateDataReceivedEvents(SPE, SPE.DeviceType);
        }
        public void UpdateDataReceivedEvents(SerialPortExtended SPE,
SerialPortExtended.DeviceTypeEnum DeviceType)
        {
            // TODO: dirty. rewrite.
            foreach (SerialPortExtended spe in this.SerialPortList)
            {
                if (SPE == null || spe.PortName != SPE.PortName) continue;
                spe.DeviceType = DeviceType;

                // remove all candidate handlers
                spe.DataReceived -= DataReceivedHandler_Any;
                spe.DataReceived -= DataReceivedHandler_ETL;
                spe.DataReceived -= DataReceivedHandler_General;

                // register
                switch (DeviceType)
                {
                    case SerialPortExtended.DeviceTypeEnum.ETL:
                        spe.DataReceived += new
SerialDataReceivedEventHandler(DataReceivedHandler_ETL);
                        break;
                    case SerialPortExtended.DeviceTypeEnum.CSUX:
```

```
                    spe.DataReceived += DataReceivedHandler_General;
                    break;
                case SerialPortExtended.DeviceTypeEnum.MAC6:
                    spe.DataReceived += DataReceivedHandler_General;
                    break;
                default:
                    spe.DataReceived += DataReceivedHandler_Any;
                    break;
            }
        }
    }

    public override void GetAndSetPorts(ComboBox comboBox_port)
    {
        comboBox_port.Items.Clear();

        foreach (var caption in ports_dic.Keys)
            comboBox_port.Items.Add(caption);

        if (ports_dic.Count > 0)
        {
            comboBox_port.SelectedIndex = 0;

            // For Optotune's ETL
            foreach (string key in ports_dic.Keys)
            {
                if (key.ToLower().IndexOf(KEY_ETL_PORT) >= 0)
                {
                    comboBox_port.SelectedItem = key;
                    break;
                }
            }
        }
    }

    #region DataReceived EventHandlers
    /// <summary>
    /// Main EventHandler for using GUI Console.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void DataReceivedHandler_ETL(object sender, SerialDataReceivedEventArgs e)
    {
        SerialPortExtended spe = (SerialPortExtended)sender;

        if (spe.IsInteractiveConsole && spe.IsOpen)
        {
            int len = spe.BytesToRead;
            byte[] buf = new byte[len];
            spe.Read(buf, 0, len);
            UpdateInteractiveProgress(buf, true);

            int offset = received_data_buffer.Length;
            Array.Resize(ref received_data_buffer, received_data_buffer.Length + len);
            Buffer.BlockCopy(buf, 0, received_data_buffer, offset, len);
        }

        if (spe.IsTesting && spe.IsOpen)
        {
            string data = spe.ReadExisting();
            Console.Write(data);
            switch (spe.TestType)
            {
                case SerialPortExtended.TestTypeEnum.Handshake:
                    if (data == spe.HANDSHAKE_ANSWER)
                        UpdateTestProgress(true);
                    else
                        UpdateTestProgress(false);
                    break;
```

```
                }
                spe.IsTesting = false;
            }
        }

        public void DataReceivedHandler_General(object sender, SerialDataReceivedEventArgs
e)
        {
            SerialPortExtended spe = (SerialPortExtended)sender;
            if (!spe.IsOpen) return;

            if (spe.IsInteractiveConsole || spe.IsGeneralUse)
            {
                ArrayList entireBuffer = new ArrayList();
                int len = spe.BytesToRead;
                while (len > 0)
                {
                    byte[] sub = new byte[1];
                    spe.Read(sub, 0, 1);
                    entireBuffer.Add(sub[0]);

                    if ((byte)entireBuffer[entireBuffer.Count - 1] == 0x0a) // check for ¥n
                    {
                        // Export our arraylist into a byte array.
                        byte[] currentByteBuffer =
(byte[])entireBuffer.ToArray(typeof(byte));
                        int offset = received_data_buffer.Length;
                        Array.Resize(ref received_data_buffer, offset + len);
                        if(len > currentByteBuffer.Length)
                            len = currentByteBuffer.Length;  // testing..
                        Buffer.BlockCopy(currentByteBuffer, 0, received_data_buffer, offset,
len);

                        if(spe.IsInteractiveConsole)
                            spe.UpdateInteractiveProgress(currentByteBuffer, true);
                        else if(spe.IsGeneralUse)
                            spe.UpdateGeneralUseProgress(currentByteBuffer, true);

                        entireBuffer.Clear();
                        break;
                    }
                    if (entireBuffer.Count > 100)
                    {
                        break; //break while if more than 10 bytes were received
                    }
                }
                // not confirmed how flags work
                spe.IsInteractiveConsole = false;
                spe.IsGeneralUse = false;
            }
        }
        private void DataReceivedHandler_Any(object sender, SerialDataReceivedEventArgs e)
        {
            SerialPortExtended spe = (SerialPortExtended)sender;
            if (spe.IsOpen)
            {
                int len = spe.BytesToRead;
                byte[] buf = new byte[len];
                spe.Read(buf, 0, len);
                if (spe.IsInteractiveConsole)
                    UpdateInteractiveProgress(buf);
                else if (spe.IsGeneralUse)
                    UpdateGeneralUseProgress(buf);
            }
        }
        #endregion

        /// <summary>
        /// Send commands to drive ETL. Use Optotune class and extended SerialPort class.
```

```
        /// </summary>
        private class ETLCommands
        {
            // CRC calculation and so on
            Optotune optotune;

            public ETLCommands()
            {
                optotune = new Optotune();
            }

            /// <summary>
            /// (Don't use this one. Use Optotune's method.) Focal Power Set Commnad. This
method sends byte array to ETL driver.
            /// </summary>
            /// <param name="spe"></param>
            /// <param name="fp_diopter"></param>
            /// <param name="CRC_Mode"></param>
            public void FocalPowerSetCommand(SerialPortExtended spe, double fp_diopter, bool
CRC_Mode = false)
            {
                byte[] command_bytes = new byte[10];
                // prefix: P
                // write identifier: w
                // coding: D
                // channel: A
                byte[] pre = { 0x50, 0x77, 0x44, 0x41 };  // PwDA (ASCII)

                // Firmware Type-F (EL-16-40): xi = fp*200    <-- This program is
                // Firmware Type-A (EL-10-30): xi = (fp+5)*200
                int xi = (int)(fp_diopter * 200);
                byte[] val = optotune.CreateDividedBytes(xi);

                byte[] dummy = { 0x00, 0x00 };

                byte[] crc = { 0x00, 0x00 };  // TODO: calc CRC

                // Merge arrays
                Buffer.BlockCopy(pre, 0, command_bytes, 0, pre.Length);
                Buffer.BlockCopy(val, 0, command_bytes, pre.Length, val.Length);
                Buffer.BlockCopy(dummy, 0, command_bytes, pre.Length + val.Length,
dummy.Length);
                Buffer.BlockCopy(crc, 0, command_bytes, pre.Length + val.Length +
dummy.Length, crc.Length);

                // Send command
                spe.Write(command_bytes, 0, command_bytes.Length);
            }

            /// <summary>
            /// (Don't use this one. Use Optotune's method.) Current Set Commnad. This
method sends byte array to ETL driver.
            /// </summary>
            /// <param name="spe"></param>
            /// <param name="current_mA"></param>
            /// <param name="CRC_Mode"></param>
            public void CurrentSetCommand(SerialPortExtended spe, double current_mA, bool
CRC_Mode = false)
            {
                byte[] command_bytes = new byte[8];
                // channel: A
                // write identifier: w
                byte[] pre = { 0x41, 0x77 };  // Aw (ASCII)

                // Firmware Type-F (EL-16-40): xi = fp*200    <-- This program is
                // Firmware Type-A (EL-10-30): xi = (fp+5)*200
                double i_o = current_mA;
                double i_c = 293;  // TODO: check
                int xi = (int)(i_o / i_c * 4096);  // TODO: check accuracy
```

```
                byte[] val = optotune.CreateDividedBytes(xi);

                byte[] crc = { 0x00, 0x00 };  // TODO: calc CRC

                // Merge arrays
                Buffer.BlockCopy(pre, 0, command_bytes, 0, pre.Length);
                Buffer.BlockCopy(val, 0, command_bytes, pre.Length, val.Length);
                Buffer.BlockCopy(crc, 0, command_bytes, pre.Length + val.Length,
crc.Length);

                // Send command
                spe.Write(command_bytes, 0, command_bytes.Length);
            }
        }
    }

}
```

Code A17. AnalogControl/AnalogSignal.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using ETL_system;
using NationalInstruments;
using NationalInstruments.DAQmx;
using ETL_system.LiveImaging;

namespace AnalogControl
{
    public class AnalogSignal
    {
        public AnalogSignal()
        {
        }

        public class AnalogOutput
        {
            public NationalInstruments.DAQmx.Task RunningTask { get; set; }
            public NationalInstruments.DAQmx.Task AOTask { get => myTask; set { myTask =
value; } }
            private NationalInstruments.DAQmx.Task myTask;

            public void CreateAOTaskSyncReadoutTrigger(
                string AOChannel = "dev2/ao3", double Frequency_Hz = 100,
                double SamplesPerBuffer = 5000, double CyclesPerBuffer = 50,
                WaveformType Waveform = WaveformType.SyncReadoutTrigger, double
WaveAmplitude = 3.3)
            {
                FunctionGenerator fGen;
                try
                {
                    // Create the task and channel
                    myTask = new NationalInstruments.DAQmx.Task();

                    myTask.AOChannels.CreateVoltageChannel(
                        AOChannel,
                        "aoChannelExternalTrigger",
                        -10.0,  // Minimum Voltage
                        10.0,  // Maximum Voltage
                        AOVoltageUnits.Volts);

                    //DigitalSingleChannelReader dreader = new
DigitalSingleChannelReader(myTask.Stream);

                    // Verify the task before doing the waveform calculations
                    myTask.Control(TaskAction.Verify);

                    // Calculate some waveform parameters and generate data
                    // clock rate for level trigger
                    double freq = Frequency_Hz;
                    double clockRate = Math.Ceiling(SamplesPerBuffer / (CyclesPerBuffer /
freq));

                    fGen = new FunctionGenerator(
                        myTask.Timing, freq, SamplesPerBuffer, CyclesPerBuffer,
WaveformType.SyncReadoutTrigger,
                        WaveAmplitude);

                    // Debug
                    if (fGen.Data != null)
```

```
                    FunctionGenerator.WriteOutWaveform(fGen.Data);

                // Configure the sample clock with the calculated rate
                myTask.Timing.ConfigureSampleClock(
                    "", // onboard clock
                    fGen.ResultingSampleClockRate,
                    SampleClockActiveEdge.Rising,
                    //SampleQuantityMode.FiniteSamples,
                    SampleQuantityMode.ContinuousSamples,
                    fGen.Data.Length);

                AnalogSingleChannelWriter writer_ao = new
AnalogSingleChannelWriter(myTask.Stream);
                writer_ao.WriteMultiSample(false, fGen.Data);
            }
            catch (Exception x)
            {
                MessageBox.Show(x.Message);
                //logger.AddText(x.Message.Trim());
                //if (task != null)
                //    task.Dispose();
            }
        }

        public void CreateAOTaskLevelTrigger(
            string AOChannel = "dev2/ao3", double LowStateLength_sec = 0.5, double
HighStateLength_sec = 0.1,
            double SamplesPerBuffer = 5000, double CyclesPerBuffer = 50,
            WaveformType Waveform = WaveformType.LevelTrigger, double WaveAmplitude =
5.0)
        {
            FunctionGenerator fGen;
            try
            {
                // Create the task and channel
                myTask = new NationalInstruments.DAQmx.Task();

                myTask.AOChannels.CreateVoltageChannel(
                    AOChannel,
                    "aoChannelExternalTrigger",
                    -10.0,  // Minimum Voltage
                    10.0,  // Maximum Voltage
                    AOVoltageUnits.Volts);

                //DigitalSingleChannelReader dreader = new
DigitalSingleChannelReader(myTask.Stream);

                // Verify the task before doing the waveform calculations
                myTask.Control(TaskAction.Verify);

                // Calculate some waveform parameters and generate data
                // clock rate for level trigger
                double freq = 1.0 / (LowStateLength_sec + HighStateLength_sec);
                double dutyCycle = HighStateLength_sec / (LowStateLength_sec +
HighStateLength_sec);
                double clockRate = Math.Ceiling(SamplesPerBuffer / (CyclesPerBuffer /
freq));

                fGen = new FunctionGenerator(
                    myTask.Timing, freq, SamplesPerBuffer, CyclesPerBuffer,
WaveformType.LevelTrigger,
                    WaveAmplitude, dutyCycle);

                // Debug
                if (fGen.Data != null)
                    FunctionGenerator.WriteOutWaveform(fGen.Data);

                // Configure the sample clock with the calculated rate
                myTask.Timing.ConfigureSampleClock(
```

```
                    "", // onboard clock
                    fGen.ResultingSampleClockRate,
                    SampleClockActiveEdge.Rising,
                    //SampleQuantityMode.FiniteSamples,
                    SampleQuantityMode.ContinuousSamples,
                    fGen.Data.Length);

                AnalogSingleChannelWriter writer_ao = new
AnalogSingleChannelWriter(myTask.Stream);
                writer_ao.WriteMultiSample(false, fGen.Data);
            }
            catch (Exception x)
            {
                MessageBox.Show(x.Message);
                //logger.AddText(x.Message.Trim());
                //if (task != null)
                //    task.Dispose();
            }
        }

        public void CreateAOTaskETL(
            string AOChannel = "dev1/ao0", double Frequency_Hz = 10, double
SamplesPerBuffer = 5000, double CyclesPerBuffer = 50,
            WaveformType Waveform = WaveformType.Sine, double WaveAmplitude = 5.0)
        {
            FunctionGenerator fGen;
            try
            {
                // Create the task and channel
                myTask = new NationalInstruments.DAQmx.Task();

                myTask.AOChannels.CreateVoltageChannel(
                    AOChannel,
                    "aoChannelETLControl",
                    -10.0,  // Minimum Voltage
                    10.0,  // Maximum Voltage
                    AOVoltageUnits.Volts);

                //DigitalSingleChannelReader dreader = new
DigitalSingleChannelReader(myTask.Stream);

                // Verify the task before doing the waveform calculations
                myTask.Control(TaskAction.Verify);

                // Calculate some waveform parameters and generate data
                fGen = new FunctionGenerator(
                    myTask.Timing,
                    Frequency_Hz,
                    SamplesPerBuffer,
                    CyclesPerBuffer,
                    Waveform,
                    WaveAmplitude);

                // Debug
                FunctionGenerator.WriteOutWaveform(fGen.Data);

                // Configure the sample clock with the calculated rate
                myTask.Timing.ConfigureSampleClock(
                    "", // onboard clock
                    fGen.ResultingSampleClockRate,
                    SampleClockActiveEdge.Rising,
                    //SampleQuantityMode.FiniteSamples,
                    SampleQuantityMode.ContinuousSamples,
                    fGen.Data.Length);

                AnalogSingleChannelWriter writer_ao = new
AnalogSingleChannelWriter(myTask.Stream);
                writer_ao.WriteMultiSample(false, fGen.Data);
            }
```

```
                catch (Exception x)
                {
                    MessageBox.Show(x.Message);
                }
            }

            public void CreateAOTaskETL_Intermittent(
                string AOChannel = "dev1/ao0", double Frequency_Hz = 1, double
SamplesPerBuffer = 5000, double CyclesPerBuffer = 50,
                WaveformType Waveform = WaveformType.Constant, double WaveAmplitude = 5.0)
            {
                FunctionGenerator fGen;
                try
                {
                    // Create the task and channel
                    myTask = new NationalInstruments.DAQmx.Task();

                    myTask.AOChannels.CreateVoltageChannel(
                        AOChannel,
                        "aoChannelETLControl",
                        -10.0,  // Minimum Voltage
                        10.0,  // Maximum Voltage
                        AOVoltageUnits.Volts);

                    // Verify the task before doing the waveform calculations
                    myTask.Control(TaskAction.Verify);

                    // Calculate some waveform parameters and generate data
                    fGen = new FunctionGenerator(
                        myTask.Timing,
                        Frequency_Hz,
                        SamplesPerBuffer,
                        CyclesPerBuffer,
                        Waveform,
                        WaveAmplitude);

                    // Debug
                    //FunctionGenerator.WriteOutWaveform(fGen.Data);

                    // Configure the sample clock with the calculated rate
                    myTask.Timing.ConfigureSampleClock(
                        "", // onboard clock
                        fGen.ResultingSampleClockRate,
                        SampleClockActiveEdge.Rising,
                        //SampleQuantityMode.FiniteSamples,
                        SampleQuantityMode.ContinuousSamples,
                        fGen.Data.Length);

                    AnalogSingleChannelWriter writer_ao = new
AnalogSingleChannelWriter(myTask.Stream);
                    writer_ao.WriteMultiSample(false, fGen.Data);
                }
                catch (Exception x)
                {
                    MessageBox.Show(x.Message);
                }
            }
            public void StartTask()
            {
                if(myTask!=null)
                    myTask.Start();
                RunningTask = myTask;
            }
            public void StopTask()
            {
                if (myTask != null)
                    myTask.Stop();
                RunningTask = null;
            }
```

```
            public void StopAndDispose()
            {
                if (myTask != null)
                {
                    myTask.Stop();
                    myTask.Dispose();
                    myTask = null;
                }
                RunningTask = null;
            }
        }

        public class AnalogInput
        {
            // Refered ContAcqVoltageSamples_IntClk in .NET4.5 Examples
            private AnalogMultiChannelReader analogInReader;
            private NationalInstruments.DAQmx.Task myTask;
            private AsyncCallback analogCallback;

            private AnalogWaveform<double>[] data;
            private DataColumn[] dataColumn = null;
            private DataTable dataTable = null;

            private string dataType = RecordingLog.VOLTAGE;  // use const string in
RecordingLog

            // params
            public double MinVoltage { get; set; }
            public double MaxVoltage { get; set; }
            public double UpdateRate { get; set; }      // [Hz]  Default 10000; Max 0.1
million (Sample code)
            public int SamplesPerChannel { get; set; }  // Default 1000; Max 0.1 million
(Sample code)
            /// <summary>
            /// Global Tick will be stored when AI recording started.
            /// </summary>
            public long GlobalTicksAtBeginning { get; private set; }
            public long GlobalTicksAtEnding { get; private set; }
            public long DAQTickAtBeginning { get; private set; }
            /// <summary>
            /// Use const string which defined in RecordingLog.cs
            /// </summary>
            public string DataType { get => dataType; set { dataType = value; } }

            public NationalInstruments.DAQmx.Task RunningTask { get; set; }
            public NationalInstruments.DAQmx.Task AITask { get => myTask; set { myTask =
value; } }
            public DataGrid acquisitionDataGrid { get; set; }
            public DataTable DataTable { get => dataTable; set { dataTable = value; } }
            private RecordingLog recLog;

            public RecordingLog RecordingLog { get => recLog; }

            // Live projection
            public bool flagLive2DProjection { get; set; }
            public ImageProjection.LiveProjectionBuffer Live2DProjectionBuffer { get; set; }


            //debug
            public HighResolutionTimerAlt hTimer;
            public int CallbackCounter = 0;

            public AnalogInput()
            {
                MinVoltage = -1.0;
                MaxVoltage = 6.0;
                UpdateRate = 1e4;      // [Hz]  Default 10000; Max 0.1 million (Sample code)
                SamplesPerChannel = 1000;  // Default 1000; Max 0.1 million (Sample code)
                // Callback Interval = SapmlesPerChannel / UpdateRate [sec]
```

```
            dataTable = new DataTable();
        }

        /// <summary>
        /// Created and started Task instance become accessable through RunningTask
property.
        /// </summary>
        /// <param name="AIChannel"></param>
        public void CreateAITask(string AIChannel = "dev1/ai0", string
VoltageDataType=RecordingLog.VOLTAGE)
        {
            if (RunningTask == null)
            {
                try
                {
                    // Create a new task
                    myTask = new NationalInstruments.DAQmx.Task();

                    // Create a virtual channel
                    myTask.AIChannels.CreateVoltageChannel(AIChannel, "",
                        (AITerminalConfiguration)(-1), MinVoltage, MaxVoltage,
AIVoltageUnits.Volts);

                    // Configure the timing parameters
                    myTask.Timing.ConfigureSampleClock("", UpdateRate,
                        SampleClockActiveEdge.Rising,
SampleQuantityMode.ContinuousSamples, 1000);

                    // Verify the Task
                    myTask.Control(TaskAction.Verify);

                    // Prepare the table for Data
                    InitializeDataTable(myTask.AIChannels, ref dataTable);
                    if (acquisitionDataGrid != null)
                        acquisitionDataGrid.DataSource = dataTable;

                    // set data type (simple Voltage or Voltage of External Trigger)
                    dataType = VoltageDataType;

                    //RunningTask = myTask;
                    analogInReader = new AnalogMultiChannelReader(myTask.Stream);
                    analogCallback = new AsyncCallback(AnalogInCallback);

                    // Use SynchronizeCallbacks to specify that the object
                    // marshals callbacks across threads appropriately.
                    analogInReader.SynchronizeCallbacks = true;

                    // Moved to StartTask()
                    //analogInReader.BeginReadWaveform(SamplesPerChannel,
                    //    analogCallback, myTask);
                    hTimer = new HighResolutionTimerAlt();
                }
                catch (DaqException exception)
                {
                    // Display Errors
                    MessageBox.Show(exception.Message);
                    RunningTask = null;
                    myTask.Dispose();
                    myTask = null;
                }
            }
        }

        /// <summary>
        /// Start Analog Input task.
        /// </summary>
        /// <param name="GlobalTimer">Used for taking Ticks at beginning.</param>
        /// <param name="RecLog"></param>
```

```
            public void StartTask(HighResolutionTimer GlobalTimer, RecordingLog RecLog)
            {
                RunningTask = myTask;
                //myTask.Start();
                analogInReader.BeginReadWaveform(SamplesPerChannel,
                    analogCallback, myTask);

                // Use this Tick as an offset of the DAQ Timings
                GlobalTicksAtBeginning = GlobalTimer.ElapsedTicks;
                Console.WriteLine($"Analog In gTimer.ElapsedTicks at beginning:
{GlobalTicksAtBeginning}");
                DAQTickAtBeginning = (long)0;
                recLog = RecLog;

                hTimer.Start();

                CallbackCounter = 0;
            }

            /// <summary>
            /// Dispose myTask and set runningTask to null.
            /// </summary>
            public void StopAndDispose()
            {
                object lck = new object();
                lock (lck)
                {
                    if (myTask != null)
                        myTask.Stop();
                    if (myTask != null)
                    {
                        myTask.Dispose();
                        myTask = null;
                    }
                    RunningTask = null;
                }
            }

            /*
            public void ForceRead()
            {
                try
                {
                    if (RunningTask != null && RunningTask == ar.AsyncState)
                    {
                        // Read the available data from the channels
                        data = analogInReader.EndReadWaveform(ar);
                        analogInReader.end

                        // Plot your data here
                        dataToDataTable(data, ref dataTable);
                    }
                }
                catch (DaqException exception)
                {
                    // Display Errors
                    MessageBox.Show(exception.Message);
                    StopAndDispose();
                }
            }
            //*/

            private void AnalogInCallback(IAsyncResult ar)
            {
                try
                {
                    if (RunningTask != null && RunningTask == ar.AsyncState)
                    {
                        // Read the available data from the channels
```

```
                    data = analogInReader.EndReadWaveform(ar);
                    //Console.WriteLine("¥tEndRead¥t" + hTimer.Elapsed_ms);

                    // Plot your data here
                    if (flagLive2DProjection)
                        dataToDataTable_ForLiveProjection(data, ref dataTable,
liveProjBuffer: Live2DProjectionBuffer);
                    else
                        dataToDataTable(data, ref dataTable);

                    //Console.WriteLine("¥tStartingRead¥t" + hTimer.Elapsed_ms);
                    analogInReader.BeginMemoryOptimizedReadWaveform(SamplesPerChannel,
analogCallback, myTask, data);
                    //Console.WriteLine("¥tStartedRead¥t" + hTimer.Elapsed_ms);

                    // debug
                    if (CallbackCounter < 3 || (CallbackCounter - 1) % 50 == 0)
                    {
                        Console.WriteLine("Analog In callback count: "+(CallbackCounter +
1).ToString() + " - Rows: " +dataTable.Rows.Count +" - Elapsed ms: "+ hTimer.Elapsed_ms);
                    }
                    CallbackCounter++;
                }
            }
            catch (DaqException exception)
            {
                // Display Errors
                MessageBox.Show(exception.Message);
                StopAndDispose();
            }
        }

        private void dataToDataTable(AnalogWaveform<double>[] sourceArray, ref DataTable
dataTable)
        {
            // Iterate over channels
            int currentLineIndex = 0;
            foreach (AnalogWaveform<double> waveform in sourceArray)
            {
                for (int sample = 0; sample < waveform.Samples.Count; ++sample)
                {
                    var sampleData = waveform.Samples[sample];
                    //dataTable.Rows[sample][currentLineIndex] = sampleData.Value;
                    long tick = GlobalTicksAtBeginning;
                    if ( (dataType == RecordingLog.VOLTAGE && recLog.VoltageTiming.Count
== 0)
                      || (dataType == RecordingLog.EXTTRIG &&
recLog.ExternalTriggerVoltageTiming.Count == 0)
                      || DAQTickAtBeginning == 0)
                        DAQTickAtBeginning = sampleData.TimeStamp.Ticks;
                    else
                        tick += sampleData.TimeStamp.Ticks - DAQTickAtBeginning;

                    // switch saving method depends on the type of voltage data
                    if(dataType == RecordingLog.VOLTAGE)
                        recLog.AddVoltage(tick, sampleData.Value);
                    else if(dataType == RecordingLog.EXTTRIG)
                        recLog.AddExternalTriggerVoltage(tick, sampleData.Value);
                    else
                        recLog.AddVoltage(tick, sampleData.Value);

                    // debug June 2021
                    // Tick will be broken (too high) when beginning the AO-task
                    //     if already running another instance of same AO-task
                    if (tick > 63760488068)
                        Console.WriteLine("Tick error: " + tick);
                }
                currentLineIndex++;
                // 1 Tick = 0.1 micro sec. Therefore use "long" type for storing Tick
```

```
counts
                    //Console.WriteLine("First Ticks: " +
waveform.Samples[0].TimeStamp.Ticks);
                }
            }
            private void dataToDataTable_ForLiveProjection(AnalogWaveform<double>[]
sourceArray, ref DataTable dataTable,
                ImageProjection.LiveProjectionBuffer liveProjBuffer)
            {
                // Iterate over channels
                int currentLineIndex = 0;
                foreach (AnalogWaveform<double> waveform in sourceArray)
                {
                    for (int sample = 0; sample < waveform.Samples.Count; ++sample)
                    {
                        var sampleData = waveform.Samples[sample];
                        //dataTable.Rows[sample][currentLineIndex] = sampleData.Value;
                        long tick = GlobalTicksAtBeginning;
                        if ((dataType == RecordingLog.VOLTAGE && recLog.VoltageTiming.Count
== 0)
                          || (dataType == RecordingLog.EXTTRIG &&
recLog.ExternalTriggerVoltageTiming.Count == 0)
                          || DAQTickAtBeginning == 0)
                            DAQTickAtBeginning = sampleData.TimeStamp.Ticks;
                        else
                            tick += sampleData.TimeStamp.Ticks - DAQTickAtBeginning;

                        // for live reconst.
                        RecordingLog.SingleRow row = new RecordingLog.SingleRow();
                        row.ElapsedTicks = tick;

                        ImageProjection.DataRow drow = new ImageProjection.DataRow(
                            tick: tick, type: dataType, value: sampleData.Value);
                        liveProjBuffer.Enqueue(drow);

                        // switch saving method depends on the type of voltage data
                        if (dataType == RecordingLog.VOLTAGE)
                        {
                            recLog.AddVoltage(tick, sampleData.Value);
                            row.Data[dataType] = sampleData.Value;
                            drow.Value = sampleData.Value;
                        }
                        else if (dataType == RecordingLog.EXTTRIG)
                        {
                            recLog.AddExternalTriggerVoltage(tick, sampleData.Value);
                            row.Data[dataType] = sampleData.Value;
                        }
                        else
                        {
                            recLog.AddVoltage(tick, sampleData.Value);
                            row.Data[RecordingLog.VOLTAGE] = sampleData.Value;
                        }

                        // debug June 2021
                        // Tick will be broken (too high) when beginning the AO-task
                        //      if already running another instance of same AO-task
                        if (tick > 63760488068)
                            Console.WriteLine("Tick error: " + tick);
                    }
                    currentLineIndex++;
                    // 1 Tick = 0.1 micro sec. Therefore use "long" type for storing Tick
counts
                    //Console.WriteLine("First Ticks: " +
waveform.Samples[0].TimeStamp.Ticks);
                }
            }

            public void InitializeDataTable(AIChannelCollection channelCollection, ref
DataTable data)
```

```
            {
                int numOfChannels = channelCollection.Count;
                data.Rows.Clear();
                data.Columns.Clear();
                dataColumn = new DataColumn[numOfChannels];
                int numOfRows = 10;

                for (int currentChannelIndex = 0; currentChannelIndex < numOfChannels;
currentChannelIndex++)
                {
                    dataColumn[currentChannelIndex] = new DataColumn();
                    dataColumn[currentChannelIndex].DataType = typeof(double);
                    dataColumn[currentChannelIndex].ColumnName =
channelCollection[currentChannelIndex].PhysicalName;
                }

                data.Columns.AddRange(dataColumn);

                for (int currentDataIndex = 0; currentDataIndex < numOfRows;
currentDataIndex++)
                {
                    object[] rowArr = new object[numOfChannels];
                    data.Rows.Add(rowArr);
                }
            }

        }
    }
}
```

Code A18. AnalogControl/FunctionGenerator.cs

```
using System;
using NationalInstruments.DAQmx;
using System.Diagnostics;
using System.Collections.Generic;
using ETL_system;

namespace AnalogControl
{
    public enum WaveformType {
        Constant           = 0,
        Sine               = 1,
        Triangular         = 2,
        Rectangular        = 3,  // not implemented
        LevelTrigger       = 4,
        SyncReadoutTrigger  = 5,
        Sawtooth           = 6,
        TangentBase        = 7,  // not implemented
    }

         public class FunctionGenerator
         {
        public FunctionGenerator() { }
        public FunctionGenerator(
            NationalInstruments.DAQmx.Timing timingSubobject,
            string desiredFrequency,
            string samplesPerBuffer,
            string cyclesPerBuffer,
            WaveformType type,
            string amplitude,
            string dutyCycle = "-1")
        {
            switch(type)
            {
                case WaveformType.Constant:
                    break;
                case WaveformType.Sine:
                    break;
                case WaveformType.Triangular:
                    break;
                case WaveformType.Rectangular:
                    break;
                case WaveformType.LevelTrigger:
                    break;
                case WaveformType.SyncReadoutTrigger:
                    break;
                case WaveformType.Sawtooth:
                    break;
                case WaveformType.TangentBase:
                    break;
                default:
                    Debug.Assert(false,"Invalid Waveform Type");
                    break;
            }

            Init(
                timingSubobject,
                Double.Parse(desiredFrequency),
                Double.Parse(samplesPerBuffer),
                Double.Parse(cyclesPerBuffer),
                type,
                Double.Parse(amplitude),
                Double.Parse(dutyCycle));
        }

        public FunctionGenerator(
```

```
            NationalInstruments.DAQmx.Timing timingSubobject,
            double desiredFrequency,
            double samplesPerBuffer,
            double cyclesPerBuffer,
            WaveformType type,
            double amplitude,
            double dutyCycle = -1)
        {
            Init(
                timingSubobject,
                desiredFrequency,
                samplesPerBuffer,
                cyclesPerBuffer,
                type,
                amplitude,
                dutyCycle);
        }

        private void Init(
            NationalInstruments.DAQmx.Timing timingSubobject,
            double desiredFrequency,
            double samplesPerBuffer,
            double cyclesPerBuffer,
            WaveformType type,
            double amplitude,
            double dutyCycle = -1)
        {
            if (desiredFrequency <= 0)
                throw new ArgumentOutOfRangeException("desiredFrequency", desiredFrequency,
"This parameter must be a positive number");
            if (samplesPerBuffer <= 0)
                throw new ArgumentOutOfRangeException("samplesPerBuffer", samplesPerBuffer,
"This parameter must be a positive number");
            if (cyclesPerBuffer <= 0)
                throw new ArgumentOutOfRangeException("cyclesPerBuffer", cyclesPerBuffer,
"This parameter must be a positive number");

            // First configure the Task timing parameters
            if (timingSubobject.SampleTimingType == SampleTimingType.OnDemand)
                timingSubobject.SampleTimingType = SampleTimingType.SampleClock;

            _samplesPerCycle = samplesPerBuffer / cyclesPerBuffer;
            _desiredSampleClockRate = desiredFrequency * _samplesPerCycle;

            // Determine the actual sample clock rate
            timingSubobject.SampleClockRate = _desiredSampleClockRate;
            _resultingSampleClockRate = timingSubobject.SampleClockRate;

            _resultingFrequency = _resultingSampleClockRate / _samplesPerCycle;

            switch (type)
            {
                case WaveformType.Constant:
                    _data = GenerateConstant(_resultingFrequency, amplitude,
_resultingSampleClockRate, samplesPerBuffer);
                    break;
                case WaveformType.Sine:
                    _data = GenerateSineWave(_resultingFrequency, amplitude,
_resultingSampleClockRate, samplesPerBuffer);
                    break;
                case WaveformType.Triangular:
                    _data = GenerateTriangularWave(_resultingFrequency, amplitude,
_resultingSampleClockRate, samplesPerBuffer);
                    break;
                case WaveformType.Rectangular:
                    // not implemented
                    break;
                case WaveformType.LevelTrigger:
                    double period_T = 1.0 / _resultingFrequency;
```

```
                    double highLength = period_T * dutyCycle;
                    double lowLength = period_T - highLength;
                    _data = GenerateLevelTrigger(highLength, lowLength, amplitude,
_resultingSampleClockRate, samplesPerBuffer);
                    break;
                case WaveformType.SyncReadoutTrigger:
                    _data = GenerateSyncReadoutTrigger(_resultingFrequency, amplitude,
_resultingSampleClockRate, samplesPerBuffer,
                        TriggerHighDuration: 1e-4);
                    break;
                case WaveformType.Sawtooth:
                    _data = GenerateSawtoothUpWave(_resultingFrequency, amplitude,
_resultingSampleClockRate, samplesPerBuffer);
                    break;
                case WaveformType.TangentBase:
                    // not implemented
                    break;
                default:
                    // Invalid type value
                    Debug.Assert(false);
                    break;
            }
        }

        public double[] Data { get => _data; }
        public double ResultingSampleClockRate { get => _resultingSampleClockRate; }

        public static double[] GenerateConstant(
            double frequency,
            double amplitude,
            double sampleClockRate,
            double samplesPerBuffer)
        {
            //double deltaT = 1 / sampleClockRate; // sec/sample
            int intSamplesPerBuffer = (int)samplesPerBuffer;

            double[] rVal = new double[intSamplesPerBuffer];

            for (int i = 0; i < intSamplesPerBuffer; i++)
                rVal[i] = amplitude;

            return rVal;
        }

        /// <summary>
        /// Generate Sine wave. Range: 0 - 5 V
        /// </summary>
        /// <param name="frequency"></param>
        /// <param name="amplitude_pp"></param>
        /// <param name="sampleClockRate"></param>
        /// <param name="samplesPerBuffer"></param>
        /// <returns></returns>
        public static double[] GenerateSineWave(
            double frequency,
            double amplitude_pp,
            double sampleClockRate,
            double samplesPerBuffer)
        {
            double deltaT = 1/sampleClockRate; // sec/sample
            int intSamplesPerBuffer = (int)samplesPerBuffer;
            double amp = amplitude_pp / 2;

            double[] rVal = new double[intSamplesPerBuffer];

            for (int i=0;i<intSamplesPerBuffer;i++)
                rVal[i] = amp * Math.Sin( (2.0 * Math.PI) * frequency * (i*deltaT) ) + amp;

            return rVal;
        }
```

```
        /// <summary>
        /// Generate Triangular wave. Range: 0 - 5 V
        /// </summary>
        /// <param name="frequency"></param>
        /// <param name="amplitude_pp"></param>
        /// <param name="sampleClockRate"></param>
        /// <param name="samplesPerBuffer"></param>
        /// <returns></returns>
        public static double[] GenerateTriangularWave(
            double frequency,
            double amplitude_pp,
            double sampleClockRate,
            double samplesPerBuffer)
        {
            double deltaT = 1 / sampleClockRate; // sec/sample
            double T = 1 / frequency;           // period [sec]
            int intSamplesPerBuffer = (int)samplesPerBuffer;

            double[] rVal = new double[intSamplesPerBuffer];

            for (int i = 0; i < intSamplesPerBuffer; i++)
            {
                double t = i * deltaT;
                // means "t mod T" and t becomes the remainder.
                while (t - T > 0)
                    t -= T;

                if (t < T / 2)  // 0 <= t < 2/T;  f(0)=-1;  f(T/4)=0
                    rVal[i] = amplitude_pp * (2 / T) * t;
                else            // 2/T <= t < T;  f(T/2)=1;  f(3T/4)=0;
                    rVal[i] = amplitude_pp * ((-2 / T) * (t - T / 2) + 1);
            }

            return rVal;
        }

        /// <summary>
        /// Generate Triangular wave. Range: 0 - 5 V
        /// </summary>
        /// <param name="frequency"></param>
        /// <param name="amplitude_pp"></param>
        /// <param name="sampleClockRate"></param>
        /// <param name="samplesPerBuffer"></param>
        /// <returns></returns>
        public static double[] GenerateSawtoothUpWave(
            double frequency,
            double amplitude_pp,
            double sampleClockRate,
            double samplesPerBuffer)
        {
            double deltaT = 1 / sampleClockRate; // sec/sample
            double T = 1 / frequency;           // period [sec]
            int intSamplesPerBuffer = (int)samplesPerBuffer;

            double[] rVal = new double[intSamplesPerBuffer];

            for (int i = 0; i < intSamplesPerBuffer; i++)
            {
                double t = i * deltaT;
                // means "t mod T" and t becomes the remainder.
                while (t - T > 0)
                    t -= T;

                // f(x) = (A/T) * (x mod T)
                rVal[i] = (amplitude_pp / T) * (t % T);
            }

            return rVal;
```

```
        }

        /// <summary>
        /// For camera triggering
        /// </summary>
        /// <param name="high_len_sec"></param>
        /// <param name="low_len_sec"></param>
        /// <param name="amplitude_pp"></param>
        /// <param name="sampleClockRate"></param>
        /// <param name="samplesPerBuffer"></param>
        /// <returns></returns>
        public static double[] GenerateLevelTrigger(
            double high_len_sec,
            double low_len_sec,
            double amplitude_pp,
            double sampleClockRate,
            double samplesPerBuffer)
        {
            double deltaT = 1 / sampleClockRate; // sec/sample
            double T = high_len_sec + low_len_sec;  // period [sec]
            int intSamplesPerBuffer = (int)samplesPerBuffer;

            double[] rVal = new double[intSamplesPerBuffer];

            for (int i = 0; i < intSamplesPerBuffer; i++)
            {
                double t = i * deltaT;
                // means "t mod T" and t becomes the remainder.
                while (t - T > 0)
                    t -= T;

                if (t < low_len_sec)  // 0 <= t < LowT
                    rVal[i] = 0;
                else                  // LowT < t <= LowT + HighT
                    rVal[i] = amplitude_pp;
            }

            return rVal;
        }

        public static double[] GenerateSyncReadoutTrigger(
            double frequency,
            double amplitude_pp,
            double sampleClockRate,
            double samplesPerBuffer,
            double TriggerHighDuration = 1e-4)
        {
            double deltaT = 1.0 / sampleClockRate; // sec/sample
            double T = 1.0 / frequency;  // period [sec]
            int intSamplesPerBuffer = (int)samplesPerBuffer;

            double[] rVal = new double[intSamplesPerBuffer];

            // to consider the case which a pulse continues shorter time than dt
            double triggerHighDuration = TriggerHighDuration;//
Math.Max(TriggerHighDuration, deltaT);
            double triggerLowDuration = T - triggerHighDuration;

            double totalTime = deltaT * samplesPerBuffer;  // [sec]
            int pulseCount = (int)Math.Floor(totalTime * frequency);
            double[] pulseTimings = new double[pulseCount];
            for (int i = 0; i < pulseCount; i++)
                pulseTimings[i] = i * T;

            int cntTrigger = 0;
            for (int i = 0; i < intSamplesPerBuffer; i++)
            {
                double t = i * deltaT;
```

```
                if(t >= pulseTimings[cntTrigger])
                {
                    int k = i;
                    double tk = k * deltaT;
                    do
                    {
                        tk = k * deltaT;
                        rVal[k] = amplitude_pp;
                        k++;
                        //System.Console.WriteLine("Triggered @ " + t + " sec");
                    } while (tk < pulseTimings[cntTrigger] + triggerHighDuration && k + 1 <
intSamplesPerBuffer);
                    cntTrigger++;
                }
                if (cntTrigger >= pulseTimings.Length)
                    break;
            }
            //System.Console.WriteLine("Trigger count:" + cntTrigger);

            return rVal;
        }

        private double[] _data;
        private double _resultingSampleClockRate;
        private double _resultingFrequency;
        private double _desiredSampleClockRate;
        private double _samplesPerCycle;

        public static void WriteOutWaveform(double[] val, string file= "waveform.csv")
        {
            System.IO.StreamWriter sw = new System.IO.StreamWriter(file);
            foreach (double v in val)
                sw.WriteLine(v);
            sw.Close();
        }
    }
}
```

Code A19. CaptureParameters.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Hamamatsu.DCAM4;
using csAcq4;
using System.Windows.Forms;
using System.Drawing;

namespace ETL_system
{
    /// <summary>
    /// Parameters for driving camera.
    /// </summary>
    public class CaptureParameters
    {
        public IntPtr HDCAMPtr { get; set; }

        public int BitDepth { get; set; }
        public int Binning { get; set; }
        public int SubarrayMode { get; set; }
        public Size SubarraySize { get; set; }
        public Point SubarrayPosition { get; set; }
        public int PixelNumberHeight { get; set; }
        public int ReadoutSpeed { get; set; }
        public double ExposureTime_ms { get; set; }
        public int Gain { get; set; }
        public int TriggerSource { get; set; }
        public int TriggerConnector { get; set; }
        public int TriggerActive { get; set; }
        public int GlobalShutter { get; set; }

        public CaptureParameters()
        {
        }
        public CaptureParameters(IntPtr __HDCAMPtr)
        {
            HDCAMPtr = __HDCAMPtr;
        }

        public int GetBitDepth(string BitDepth_str)
        {
            return Convert.ToInt32(BitDepth_str.Replace("bit", ""));
        }
        public double GetBinning(string Binning_str)
        {
            if (Binning_str.IndexOf("x") == -1)
                return 1;
            else
            {
                double x = Convert.ToDouble(Binning_str.Substring(Binning_str.IndexOf("x") +
1));
                return x;
            }
        }
        /// <summary>
        /// Setup for ORCA Fusion BT (updated from ORCA-Flash4.0 V3). TODO: enable
different setup e.g. 2048x1024, 1024x128
        /// </summary>
        /// <param name="Subarray_str"></param>
        /// <returns></returns>
        public Size GetSubarraySize(string Subarray_str)
        {
```

```
            string upper = Subarray_str.ToUpper();
            List<int> nums = new List<int>();
            foreach(string num_str in upper.Split('X'))
            {
                int.TryParse(num_str, out int num);
                nums.Add(num);
            }
            return new Size(nums[0], nums[1]);

            /*
            switch (upper)
            {
                case "2304X2304":
                    return new Size(2304, 2304);
                case "2304X2048":
                    return new Size(2304, 2048);
                case "2048X2048":
                    return new Size(2048, 2048);
                case "1280X1280":
                    return new Size(1280, 1280);
                case "1024X1024":
                    return new Size(1024, 1024);
                case "512X512":
                    return new Size(512, 512);
                default:
                    return new Size(2048, 2048);
            }
            //*/
        }
        public Point GetSubarrayPositionDefault(string Subarray_str)
        {
            Size size = GetSubarraySize(Subarray_str);
            // ORCA Fusion BT
            int maxSize = 2304;
            int posX = (maxSize - size.Width) / 2;
            int posY = (maxSize - size.Height) / 2;
            Point pos = new Point(posX, posY);
            return pos;

            /*
            string upper = Subarray_str.ToUpper();
            switch (upper)
            {
                //*  ORCA Flash 4.0 V3
                case "2048X2048":
                    return new Point(0, 0);
                case "1024X1024":
                    return new Point(512, 512);
                case "512X512":
                    return new Point(768, 768);
                //*
                // ORCA Fusion BT //
                case "2304X2304":
                    return new Point(0, 0);
                case "2304X2048":
                    return new Point(0, 128);
                case "2048X2048":
                    return new Point(128, 128);
                case "1280X1280":
                    return new Point(512, 512);
                case "1024X1024":
                    return new Point(640, 640);
                case "512X512":
                    return new Point(896, 896);
                default:
                    return new Point(0, 0);
            }
            //*/
        }
```

```
        public double GetExposureTime(string ExposureTime_str)
        {
            return Convert.ToDouble(ExposureTime_str);
        }
        public DCAMPROP GetReadoutSpeed(string ReadoutSpeed_str)
        {
            string upper = ReadoutSpeed_str.ToUpper();
            switch (upper)
            {
                case "SLOW":
                    return DCAMPROP.READOUTSPEED.SLOWEST;
                case "FAST":
                    return DCAMPROP.READOUTSPEED.FASTEST;
                case "STANDARD":
                    return DCAMPROP.READOUTSPEED.STANDARD;
                default:
                    return DCAMPROP.READOUTSPEED.FASTEST;
            }
        }
        public DCAMPROP GetTriggerSource(string TriggerSrc_str)
        {
            string upper = TriggerSrc_str.ToUpper();
            switch (upper)
            {
                case "INTERNAL":
                    return DCAMPROP.TRIGGERSOURCE.INTERNAL;
                case "EXTERNAL":
                    return DCAMPROP.TRIGGERSOURCE.EXTERNAL;
                case "SOFTWARE":
                    return DCAMPROP.TRIGGERSOURCE.SOFTWARE;
                case "MASTERPULSE":
                    return DCAMPROP.TRIGGERSOURCE.MASTERPULSE;
                default:
                    return DCAMPROP.TRIGGERSOURCE.INTERNAL;
            }
        }
        public DCAMPROP GetTriggerConnector(string TriggerConnector_str)
        {
            string upper = TriggerConnector_str.ToUpper();
            switch (upper)
            {
                case "BNC":
                    return DCAMPROP.TRIGGER_CONNECTOR.BNC;
                case "INTERFACE":
                    return DCAMPROP.TRIGGER_CONNECTOR.INTERFACE;
                case "MULTI":
                    return DCAMPROP.TRIGGER_CONNECTOR.MULTI;
                default:
                    return DCAMPROP.TRIGGER_CONNECTOR.BNC;
            }
        }
        public DCAMPROP GetTriggerActive(string TriggerActive_str)
        {
            string upper = TriggerActive_str.ToUpper();
            switch (upper)
            {
                case "EDGE":
                    return DCAMPROP.TRIGGERACTIVE.EDGE;
                case "LEVEL":
                    return DCAMPROP.TRIGGERACTIVE.LEVEL;
                case "SYNCREADOUT":
                    return DCAMPROP.TRIGGERACTIVE.SYNCREADOUT;
                default:
                    return DCAMPROP.TRIGGERACTIVE.EDGE;
            }
        }
        public DCAMPROP GetActivePolarity(string str)
        {
            string upper = str.ToUpper();
```

```
            switch (upper)
            {
                case "NEGATIVE":
                    return DCAMPROP.TRIGGERPOLARITY.NEGATIVE;
                case "POSITIVE":
                    return DCAMPROP.TRIGGERPOLARITY.POSITIVE;
                default:
                    return DCAMPROP.TRIGGERPOLARITY.NEGATIVE;
            }
        }
        public DCAMPROP GetGlobalExposure(string GlobalExposure_str)
        {
            string upper = GlobalExposure_str.ToUpper();
            switch (upper)
            {
                case "GLOBAL (AREA)":
                    return DCAMPROP.TRIGGER_GLOBALEXPOSURE.GLOBALRESET; // Expose whole
pixels at the same time
                case "EMULATED GLOBAL":
                    return DCAMPROP.TRIGGER_GLOBALEXPOSURE.EMULATE; // similar to ALWAYS.
This uses ROLLING
                case "ROLLING (LINE)":
                    return DCAMPROP.TRIGGER_GLOBALEXPOSURE.DELAYED; // Expose line by line.
This causes delay
                default:
                    return DCAMPROP.TRIGGER_GLOBALEXPOSURE.ALWAYS;
            }
        }
        public DCAMPROP GetCoolerMode(string CoolerMode_str)
        {
            string upper = CoolerMode_str.ToUpper();
            switch (upper)
            {
                case "OFF":
                    return DCAMPROP.SENSORCOOLER.OFF;
                case "ON":
                    return DCAMPROP.SENSORCOOLER.ON;
                case "MAX":
                    return DCAMPROP.SENSORCOOLER.MAX;
                default:
                    return DCAMPROP.SENSORCOOLER.MAX;
            }
        }


        public void SetDCAMProps(Dictionary<DCAMIDPROP, double> dic, ref MyDcam mydcam)
        {
            foreach (DCAMIDPROP key in dic.Keys)
                dcamapidll.dcamprop_setvalue(mydcam.m_hdcam, key, dic[key]);
        }
        /// <summary>
        ///
        /// </summary>
        /// <param name="idProp"></param>
        /// <param name="fVal"></param>
        /// <param name="Verify">Verify the value and Save it as Property.</param>
        public void SetDCAMProps(DCAMIDPROP idProp, double fVal, bool Verify = true)
        {
            if (Verify && HDCAMPtr != null)
            {
                // Verify the value and Save it as property for further use.
                double ret = fVal;
                dcamapidll.dcamprop_setgetvalue(HDCAMPtr, idProp, ref ret, 0);  // 3rd arg
will be set. And get the result.

                if (ret != fVal)
                    Console.WriteLine("The DCAM-Prop value did not match. ¥n(set:" + fVal +
" != get:" + ret + ")", "Warning");
```

```
                if (idProp == DCAMIDPROP.TRIGGERSOURCE)
                    TriggerSource = (int)ret;
                else if (idProp == DCAMIDPROP.TRIGGER_CONNECTOR)
                    TriggerConnector = (int)ret;
                else if (idProp == DCAMIDPROP.TRIGGERACTIVE)
                    TriggerActive = (int)ret;
                else if (idProp == DCAMIDPROP.BITSPERCHANNEL)
                    BitDepth = (int)ret;
                else if (idProp == DCAMIDPROP.BINNING)
                    Binning = (int)ret;

                else if (idProp == DCAMIDPROP.SUBARRAYMODE)
                    SubarrayMode = (int)ret;
                else if (idProp == DCAMIDPROP.SUBARRAYHSIZE)
                    SubarraySize = new Size((int)ret, SubarraySize.Height);
                else if (idProp == DCAMIDPROP.SUBARRAYVSIZE)
                    SubarraySize = new Size(SubarraySize.Width, (int)ret);
                else if (idProp == DCAMIDPROP.SUBARRAYHPOS)
                    SubarrayPosition = new Point((int)ret, SubarrayPosition.Y);
                else if (idProp == DCAMIDPROP.SUBARRAYVPOS)
                    SubarrayPosition = new Point(SubarrayPosition.X, (int)ret);

                else if (idProp == DCAMIDPROP.EXPOSURETIME)
                    ExposureTime_ms = (double)ret * 1e3;  // ret [sec]
                else if (idProp == DCAMIDPROP.READOUTSPEED)
                    ReadoutSpeed = (int)ret;
            }
            else
            {
                // without Verifying
                dcamapidll.dcamprop_setvalue(HDCAMPtr, idProp, fVal);
            }

        }

        public double GetDCAMProps(DCAMIDPROP idProp)
        {
            double result = 0;
            dcamapidll.dcamprop_getvalue(HDCAMPtr, idProp, ref result);
            return result;
        }

    }
}
```

Code A20. DCAMImage.cs

```
using Hamamatsu.DCAM4;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ETL_system
{
    public class DCAMImage
    {
        public DCAMBUF_FRAME bufframe;
        public DCAMImage()
        {
            bufframe = new DCAMBUF_FRAME(0);
        }
        public int width { get { return bufframe.width; } }
        public int height { get { return bufframe.height; } }
        public DCAM_PIXELTYPE pixeltype { get { return bufframe.type; } }
        public bool isValid()
        {
            if (width <= 0 || height <= 0 || pixeltype == DCAM_PIXELTYPE.NONE)
            {
                return false;
            }
            else
            {
                return true;
            }
        }
        public void clear()
        {
            bufframe.width = 0;
            bufframe.height = 0;
            bufframe.type = DCAM_PIXELTYPE.NONE;
        }
        public void set_iFrame(int index)
        {
            bufframe.iFrame = index;
        }
    }

    public struct DCAMLut
    {
        public int camerabpp;         // camera bit per pixel.  This sample code only
support MONO.
        public int cameramax;

        public int inmax;
        public int inmin;
    };
}
```

Code A21. ETLCalibration.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using System.Windows.Forms;
using Windows.Media.Devices;
using LensDriverController.Forms;
using System.Diagnostics.Eventing.Reader;
using System.Collections.ObjectModel;

namespace ETL_system
{
    public class ETLCalibration
    {
        #region Props
        public string TablePath { get; set; }
        public double Coefficient { get; set; }
        public double Constant { get; set; }
        public double CurrentMinBound { get; set; }
        public double CurrentMaxBound { get; set; }
        public double QuantizationLevels { get; set; }
        public const double VoltageMinBound = 0.0;
        public const double VoltageMaxBound = 5.0;
        public double VoltageStep { get; set; }
        public double CurrentStep { get; set; }
        public Dictionary<double, double> VoltageToCurrentMap { get; set; }
        public Dictionary<double, double> CurrentToDepthMap { get; set; }
        #endregion

        public ETLCalibration()
        {
            QuantizationLevels = 1024;  // 10-bit
            VoltageStep = 5.0 / QuantizationLevels;  // 0 to 5 V

            // example
            CurrentMinBound = -100;
            CurrentMaxBound = 50;
        }

        public double GetDepthFromCurrent(double Current)
        {
            double depth = Coefficient * Current + Constant;
            return depth;
        }
        public double GetDepthFromCurrent(double Current, double _Coefficient, double
_Constant)
        {
            double depth = _Coefficient * Current + _Constant;
            return depth;
        }

        public double GetCurrentFromVoltage(double Voltage)
        {
            double current = double.NaN;
            var keys = VoltageToCurrentMap.Keys;
            double lastVolt = VoltageMinBound;
            foreach (double volt in VoltageToCurrentMap.Keys)
            {
                if (volt == VoltageMinBound)
                    continue;

                if(lastVolt <= Voltage && Voltage <= volt &&
VoltageToCurrentMap.ContainsKey(volt))
```

```
                {
                    // Simple interpolation (TODO: use full range)
                    double slope = (VoltageToCurrentMap[volt] -
VoltageToCurrentMap[lastVolt]) / (volt - lastVolt);
                    double x_volt = Voltage - lastVolt;
                    current = slope * x_volt + VoltageToCurrentMap[lastVolt];
                    break;
                }
                lastVolt = volt;
            }
            return current;
        }

        /// <summary>
        /// Voltage (0 to 5V) to current (lower to upper bounds), 10-bit mapping in ETL
        /// </summary>
        public void CreateMapVoltageToCurrent()
        {
            VoltageStep = VoltageMaxBound / QuantizationLevels;  // 0 to 5 V
            CurrentStep = (CurrentMaxBound - CurrentMinBound) / QuantizationLevels;

            VoltageToCurrentMap = new Dictionary<double, double>();
            for (int i = 0; i < QuantizationLevels; i++)
            {
                double key = VoltageStep * i;
                double val = CurrentMinBound + CurrentStep * i;
                VoltageToCurrentMap.Add(key, val);
            }
        }
        /// <summary>
        /// Offset to set zero minimum depth. Higher focal power (higher current) means
shorter axial position.
        /// </summary>
        /// <returns></returns>
        public double GetOffsetDepth()
        {
            // higher focal power means shorter axial position, so that use MAX Bound here
            double offset = -1 * GetDepthFromCurrent(CurrentMaxBound);
            return offset;
        }

    }
}
```

Code A22. HighResolutionTimer.cs

```
using System;
using System.Diagnostics;
using System.Globalization;
using System.IO;
using System.Text;
using System.Threading;

namespace ETL_system
{
    /// Downloaded from the page:
    /// https://gist.github.com/DraTeots/436019368d32007284f8a12f1ba0f545
    /// Author: Dmitry Romanov
    /// <summary>
    /// High precision non overlapping timer
    /// https://stackoverflow.com/a/41697139/548894
    /// </summary>
    /// <remarks>
    /// This implementation guaranteed that Elapsed events
    /// are not overlapped with different threads.
    /// Which is important, because a state of the event handler attached to  Elapsed,
    /// may be left unprotected of multi threaded access
    /// </remarks>
    public class HighResolutionTimer
    {
        Stopwatch stopwatch;
        /// <summary>
        /// Tick time length in [ms]
        /// </summary>
        public static readonly double TickLength = 1000f / Stopwatch.Frequency;

        public double Elapsed_ms { get { return ElapsedTicks * TickLength; } }

        /// <summary>
        /// Tick frequency
        /// </summary>
        public static readonly double Frequency = Stopwatch.Frequency;

        /// <summary>
        /// True if the system/operating system supports HighResolution timer
        /// </summary>
        public static bool IsHighResolution = Stopwatch.IsHighResolution;

        /// <summary>
        /// Invoked when the timer is elapsed
        /// </summary>
        public event EventHandler<HighResolutionTimerElapsedEventArgs> Elapsed;

        /// <summary>
        /// The interval of timer ticks [ms]
        /// </summary>
        private volatile float _interval;

        /// <summary>
        /// The timer is running
        /// </summary>
        private volatile bool _isRunning;

        /// <summary>
        ///  Execution thread
        /// </summary>
        private Thread _thread;

        private bool _isReady;

        /// <summary>
```

```
        /// Creates a timer with 1 [ms] interval
        /// </summary>
        public HighResolutionTimer() : this(1f)
        {
        }

        /// <summary>
        /// Creates timer with interval in [ms]
        /// </summary>
        /// <param name="interval">Interval time in [ms]</param>
        public HighResolutionTimer(float interval)
        {
            Interval = interval;
        }

        public long ElapsedTicks { get => stopwatch.ElapsedTicks; }

        /// <summary>
        /// The interval of a timer in [ms]
        /// </summary>
        public float Interval
        {
            get => _interval;
            set
            {
                if (value < 0f || Single.IsNaN(value))
                {
                    throw new ArgumentOutOfRangeException(nameof(value));
                }
                _interval = value;
            }
        }

        /// <summary>
        /// True when timer is running
        /// </summary>
        public bool IsRunning => _isRunning;

        /// <summary>
        /// If true, sets the execution thread to ThreadPriority.Highest
        /// (works after the next Start())
        /// </summary>
        /// <remarks>
        /// It might help in some cases and get things worse in others.
        /// It suggested that you do some studies if you apply
        /// </remarks>
        public bool UseHighPriorityThread { get; set; } = false;

        /// <summary>
        /// To make Start() lighter and faster, prepare new thread separately.
        /// </summary>
        public void Ready()
        {
            stopwatch = new Stopwatch();
            _thread = new Thread(ExecuteTimer)
            {
                IsBackground = true,
            };
            if (UseHighPriorityThread)
            {
                _thread.Priority = ThreadPriority.Highest;
            }
            _isReady = true;
        }

        /// <summary>
        /// Starts the timer
        /// </summary>
        public void Start()
```

```
        {
            if (_isRunning) return;
            if(!_isReady)
                Ready();

            _isReady = false;
            _isRunning = true;
            _thread.Start();
        }

        /// <summary>
        /// Stops the timer
        /// </summary>
        /// <remarks>
        /// This function is waiting an executing thread (which do  to stop and join.
        /// </remarks>
        public void Stop(bool joinThread = true)
        {
            if (_isRunning)
            {
                _isRunning = false;

                // Even if _thread.Join may take time it is guaranteed that
                // Elapsed event is never called overlapped with different threads
                if (joinThread && Thread.CurrentThread != _thread && _thread != null)
                {
                    _thread.Join();
                }
            }
        }
        private void ExecuteTimer()
        {
            float nextTrigger = 0f;
            double elapsed, diff, delay;

            // edit (temporary)
            //Stopwatch stopwatch = new Stopwatch();
            stopwatch.Start();

            while (_isRunning)
            {
                nextTrigger += _interval;

                while (true)
                {
                    elapsed = stopwatch.ElapsedTicks * TickLength;
//ElapsedHiRes(stopwatch);
                    diff = nextTrigger - elapsed;
                    if (diff <= 0f)
                        break;

                    else if (diff < 1f)
                        Thread.SpinWait(10);
                    else if (diff < 5f)
                        Thread.SpinWait(50);  // changed 100 to 50
                    else if (diff < 15f)
                        Thread.Sleep(1);
                    else
                        Thread.Sleep(10);

                    if (!_isRunning)
                        return;
                }


                delay = elapsed - nextTrigger;
                Elapsed?.Invoke(this, new HighResolutionTimerElapsedEventArgs(delay));

                if (!_isRunning)
```

```
                    return;

                // restarting the timer in every hour to prevent precision problems
                if (stopwatch.Elapsed.TotalHours >= 1d)
                {
                    stopwatch.Restart();
                    nextTrigger = 0f;
                }
            }

            stopwatch.Stop();
        }

        private static double ElapsedHiRes(Stopwatch stopwatch)
        {
            return stopwatch.ElapsedTicks * TickLength;
        }
    }


    public class HighResolutionTimerElapsedEventArgs : EventArgs
    {
        /// <summary>/// Real timer delay in [ms]/// </summary>
        public double Delay { get; }

        internal HighResolutionTimerElapsedEventArgs(double delay)
        {
            Delay = delay;
        }
    }
}
```

Code A23. HighResolutionTimerAlternative.cs

```
using System;
using System.Diagnostics;
using System.Globalization;
using System.IO;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace ETL_system
{
    /// Downloaded from the page:
    /// https://gist.github.com/DraTeots/436019368d32007284f8a12f1ba0f545
    /// Author: Dmitry Romanov
    /// <summary>
    /// High precision non overlapping timer (changed from "Thread" to "Task")
    /// https://stackoverflow.com/a/41697139/548894
    /// </summary>
    /// <remarks>
    /// This implementation guaranteed that Elapsed events
    /// are not overlapped with different threads.
    /// Which is important, because a state of the event handler attached to  Elapsed,
    /// may be left unprotected of multi threaded access
    /// </remarks>
    public class HighResolutionTimerAlt
    {
        Stopwatch stopwatch;
        /// <summary>
        /// Tick time length in [ms]
        /// </summary>
        public static readonly double TickLength = 1000f / Stopwatch.Frequency;

        public double Elapsed_ms { get { return ElapsedTicks * TickLength; } }

        /// <summary>
        /// Tick frequency
        /// </summary>
        public static readonly double Frequency = Stopwatch.Frequency;

        /// <summary>
        /// True if the system/operating system supports HighResolution timer
        /// </summary>
        public static bool IsHighResolution = Stopwatch.IsHighResolution;

        /// <summary>
        /// Invoked when the timer is elapsed
        /// </summary>
        public event EventHandler<HighResolutionTimerAltElapsedEventArgs> Elapsed;

        /// <summary>
        /// The interval of timer ticks [ms]
        /// </summary>
        private volatile float _interval;

        /// <summary>
        ///  Execution thread
        /// </summary>
        //private Thread _thread;
        private CancellationTokenSource _cancellationTokenSource;
        private Task _task;

        private bool _isReady;

        public bool IsRunning { get { return (_task != null); } }

        /// <summary>
```

```
        /// Creates a timer with 1 [ms] interval
        /// </summary>
        public HighResolutionTimerAlt() : this(1f)
        {
        }

        /// <summary>
        /// Creates timer with interval in [ms]
        /// </summary>
        /// <param name="interval">Interval time in [ms]</param>
        public HighResolutionTimerAlt(float interval)
        {
            Interval = interval;
            _cancellationTokenSource = new CancellationTokenSource();
        }

        public long ElapsedTicks { get => stopwatch.ElapsedTicks; }

        /// <summary>
        /// The interval of a timer in [ms]
        /// </summary>
        public float Interval
        {
            get => _interval;
            set
            {
                if (value < 0f || Single.IsNaN(value))
                {
                    throw new ArgumentOutOfRangeException(nameof(value));
                }
                _interval = value;
            }
        }

        /// <summary>
        /// If true, sets the execution thread to ThreadPriority.Highest
        /// (works after the next Start())
        /// </summary>
        /// <remarks>
        /// It might help in some cases and get things worse in others.
        /// It suggested that you do some studies if you apply
        /// </remarks>
        public bool UseHighPriorityThread { get; set; } = false;

        /// <summary>
        /// To make Start() lighter and faster, prepare new thread separately.
        /// </summary>
        public void Ready()
        {
            stopwatch = new Stopwatch();
            _isReady = true;
        }

        /// <summary>
        /// Starts the timer
        /// </summary>
        public void Start()
        {
            if (_task != null)
                return;
            if(!_isReady)
                Ready();

            _isReady = false;
            _cancellationTokenSource = new CancellationTokenSource();
            _task = Task.Factory.StartNew(() => {
                try
                {
                    ExecuteTimer(_cancellationTokenSource.Token);
```

```
            }
            catch (OperationCanceledException e)
            {
                // Do nothing
            }
        });
    }

    /// <summary>
    /// Stops the timer
    /// </summary>
    /// <remarks>
    /// This function is waiting an executing thread (which do  to stop and join.
    /// </remarks>
    public void Stop()
    {
        _cancellationTokenSource.Cancel();
        _task = null;

        // Even if _thread.Join may take time it is guaranteed that
        // Elapsed event is never called overlapped with different threads
        /*if (joinThread && Thread.CurrentThread != _thread && _thread != null)
        {
            _thread.Join();
        }*/
    }
    private void ExecuteTimer(CancellationToken cancelToken)
    {
        float nextTrigger = 0f;
        double elapsed, diff, delay;

        // edit (temporary)
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();

        while (!cancelToken.IsCancellationRequested)
        {
            nextTrigger += _interval;

            while (true)
            {
                elapsed = ElapsedHiRes(stopwatch);
                diff = nextTrigger - elapsed;
                if (diff <= 0f)
                    break;

                if (diff < 1f)
                    Thread.SpinWait(10);
                else if (diff < 5f)
                    Thread.SpinWait(100);
                else if (diff < 15f)
                    Thread.Sleep(1);
                else
                    Thread.Sleep(10);
            }

            delay = elapsed - nextTrigger;
            Elapsed?.Invoke(this, new HighResolutionTimerAltElapsedEventArgs(delay));

            if (cancelToken.IsCancellationRequested)
            {
                cancelToken.ThrowIfCancellationRequested();
                return;
            }

            // restarting the timer in every hour to prevent precision problems
            if (stopwatch.Elapsed.TotalHours >= 1d)
            {
                stopwatch.Restart();
```

```
                    nextTrigger = 0f;
                }
            }

            stopwatch.Stop();
            cancelToken.ThrowIfCancellationRequested();
        }

        private static double ElapsedHiRes(Stopwatch stopwatch)
        {
            return stopwatch.ElapsedTicks * TickLength;
        }
    }


    public class HighResolutionTimerAltElapsedEventArgs : EventArgs
    {
        /// <summary>/// Real timer delay in [ms]/// </summary>
        public double Delay { get; }

        internal HighResolutionTimerAltElapsedEventArgs(double delay)
        {
            Delay = delay;
        }
    }
}
```

Code A24. MAC6.cs

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ETL_system
{
    public class MAC6
    {
        private SerialPortExtended spe;
        private double z_RealPosition;

        const string WHERE = "WHERE B";
        const string MOVE = "MOVE B=";
        const string SETZERO = "HERE B=0";
        const string HALT = "HALT";
        const string SPEED = "SPEED B=";
        byte[] END_BYTE = { 0x0d };
        private string last_str = "";

        public SerialPortExtended SerialPortEx { get=>spe; set { spe = value; } }
        public bool IsOpen { get; set; }
        private ConcurrentStack<int> moveOrderStack;
        /// <summary>
        /// Target position (Read Value)
        /// </summary>
        public int Z_TargetValue { get; private set; }
        public int Z_TargetValue_Last { get; private set; }
        public int Z_ReadValue { get; set; }
        public int Z_ReadValue_LastReported { get; set; }
        public double Z_RealPositionMicron
        {
            get {
                z_RealPosition = (double)Z_ReadValue / Z_ReadValuePerMicronPos;
                return z_RealPosition;
            } set { z_RealPosition = value; } }
        /// <summary>
        /// Parameter for translating the FocusDriveMotor's read-value to the actual euclid
distance.
        /// TODO: Consider "Positive direction" or "Negative direction"
        /// </summary>
        public double Z_ReadValuePerMicronPos { get; set; }
        public double Z_ReadValuePerMicronNeg { get; set; }

        /// <summary>
        /// For XYZ imaging (conventional step by step imaging).
        /// </summary>
        public List<double> Z_MovePlan { get; set; }
        public List<double> Z_MovePlanInReadValue { get; set; }
        /// <summary>
        /// Present Index (not real number). Check also IsMoving state
        /// </summary>
        public int Z_MovePlan_Index { get; private set; }
        public bool IsMoving { get; private set; }

        public MAC6(SerialPortExtended SPE)
        {
            spe = SPE;
            IsOpen = false;
            Z_RealPositionMicron = 0;
            IsMoving = false;
            moveOrderStack = new ConcurrentStack<int>();
            //spe.GeneralUseEvent += new
```

```
SerialPortExtended.GeneralUseEventHandler(CallBackDataReceivedEvent);
        }
        public void RefleshTargetPos()
        {
            if (moveOrderStack.Count > 0)
            {
                lock (moveOrderStack)
                {
                    if (moveOrderStack.TryPop(out int result))
                    {
                        Z_TargetValue = result;
                        moveOrderStack.Clear();
                    }
                }
                if(Z_TargetValue != Z_TargetValue_Last)
                    SendMovePosition_ZVal(Z_TargetValue, true);
            }
            else
            {
                if (Z_ReadValue != Z_TargetValue && Z_TargetValue_Last != Z_TargetValue)
                {
                    SendMovePosition_ZVal(Z_TargetValue, true);
                }
            }
        }

        public void SendMovePosition_ZReal(double Pos_ZReal, bool UseReturnVal = false,
bool RelativePos = false)
        {
            double pos_val = Pos_ZReal * Z_ReadValuePerMicronPos;
            if (RelativePos)
                pos_val += Z_ReadValue;

            SendMovePosition_ZVal((int)pos_val, UseReturnVal);
        }
        public void SendMovePosition_ZVal(int Pos_Val, bool UseReturnVal=false)
        {
            if (Pos_Val == Z_ReadValue)
                return;

            string cmd = MOVE+Pos_Val.ToString();

            Z_TargetValue_Last = Z_TargetValue;

            Z_TargetValue = Pos_Val;
            IsMoving = true;

            spe.IsGeneralUse = true;
            spe.Write(cmd);
            spe.Write(END_BYTE, 0, 1);

            if(UseReturnVal)
                SendGetPosition_Z();
        }
        public void SendGetPosition_Z()
        {
            spe.IsGeneralUse = true;
            spe.Write(WHERE);
            spe.Write(END_BYTE, 0, 1);
        }
        public void SendSetZero()
        {
            Z_TargetValue_Last = Z_TargetValue;
            Z_TargetValue = 0;
            IsMoving = true;

            spe.IsGeneralUse = true;
            spe.Write(SETZERO);
            spe.Write(END_BYTE, 0, 1);
```

```
        }
        public void SendHalt()
        {
            spe.IsGeneralUse = true;
            spe.Write(HALT);
            spe.Write(END_BYTE, 0, 1);

            //test
            Z_TargetValue = Z_TargetValue_Last;
            moveOrderStack.Clear();
            this.IsMoving = false;
        }
        /// <summary>
        /// Default speed is 18000. Max = 500000.
        /// </summary>
        /// <param name="speed">Max: 500000</param>
        public void SendSetSpeed_Z(int speed)
        {
            string cmd = SPEED + speed.ToString();

            spe.IsGeneralUse = true;
            spe.Write(cmd);
            spe.Write(END_BYTE, 0, 1);
        }

        public List<double> CreateZMovePlan(double top, double bottom, double step_micron,
bool top_to_bottom=true)
        {
            // Create MovePlan in micro meter
            List<double> plan = new List<double>();
            double add = 0;
            for (int i = 0; top+i*step_micron <= bottom; i++)
            {
                add = step_micron * (double)i;
                plan.Add(top + add);
            }
            if (!top_to_bottom)
                plan.Reverse();

            // Create MovePlan in ReadValue
            List<double> planInReadValue = new List<double>();
            foreach(double depth in plan)
                 planInReadValue.Add(depth* Z_ReadValuePerMicronPos);

            return plan;
        }
        public void ZMoveNext()
        {
        }

        public void CheckZReached()
        {
            if (Z_TargetValue == Z_ReadValue)
                IsMoving = false;
        }

        #region CallBack
        public void CallBackDataReceivedEvent(SerialPortExtended.GeneralUseEventArgs e)
        {
            byte[] data = e.Data;
            bool clearBuffer = e.CanClearBuffer;
            try
            {
                if (data.Length > 0)
                {
                    string strHex = BitConverter.ToString(data);
                    string str = Utils.ConvertHex(strHex, sep: "-").Replace(":A ",
"").Replace("¥n", "").Trim();
                    if (int.TryParse(str, out int result))
```

```
                    {
                        bool doUpdate = !str.Equals(last_str);
                        if (doUpdate)
                        {
                            // Set data
                            Z_ReadValue = result;
                            // todo: use also Negative Direction
                            if (Z_ReadValuePerMicronPos > 0)
                                Z_RealPositionMicron = (double)Z_ReadValue /
Z_ReadValuePerMicronPos;

                            // whether reach to the target position
                            if (Z_ReadValue == Z_TargetValue)
                            {
                                IsMoving = false;
#if DEBUG
                                Console.WriteLine($"Reached to {z_RealPosition:F1} um
({Z_ReadValue} [ReadValue])");
#endif
                            }
                            else
                            {
#if DEBUG
                                int displacement = Math.Abs(Z_ReadValue -
Z_ReadValue_LastReported);
                                if (displacement > Math.Abs(Z_ReadValuePerMicronPos) ||
displacement > Math.Abs(Z_ReadValuePerMicronNeg))
                                {
                                    //Console.WriteLine($"[displacement={displacement}] Moving
at {z_RealPosition:F1} um ({str.Replace("¥n", "")})¥tTarget = {Z_TargetValue}");
                                    Z_ReadValue_LastReported = Z_ReadValue;
                                }
#endif
                            }
                            last_str = str;
                        }
                    }
                }
            }
            finally
            {
                if (clearBuffer)
                {
                    spe.DiscardInBuffer(); // If don't do this, response slows down
                    //SelectedPortInstance.Close();
                }
            }
        }
        #endregion

        public void PushOrder(int movePosRawValue)
        {
            moveOrderStack.Push(movePosRawValue);
        }
    }
}
```

Code A25. RecordingLog.cs

```
using NationalInstruments.Restricted;
using System;
using System.Collections.Specialized;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ETL_system
{
    /// <summary>
    /// Logging behavior of instruments in recording session.
    /// </summary>
    public class RecordingLog
    {
        // Column names used in csv file
        public const string TICKS = "Ticks";
        public const string VOLTAGE = "Voltage";
        public const string CURRENT = "Current";
        public const string CAMERATIMING = "CameraState";
        public const string FRAME = "FrameNumber";
        public const string DEPTH = "Depth";
        public const string DEPTHRAW = "DepthRawValue";
        public const string EXTTRIG = "ExternalTriggerVoltage";
        private static readonly string[] KEYS = { TICKS, VOLTAGE, CURRENT, CAMERATIMING,
FRAME, DEPTH, DEPTHRAW, EXTTRIG };

        private Dictionary<long, double> currentTiming;
        private Dictionary<long, double> voltageTiming;
        private Dictionary<long, TimingEnum> camCapturingTiming;
        private Dictionary<int, double> depthOfFrame;
        private Dictionary<int, double> depthRawOfFrame;
        private Dictionary<long, double> depthTiming;
        private Dictionary<long, double> extTrigVoltageTiming;

        public Dictionary<long, double> CurrentTiming { get => currentTiming; }
        public Dictionary<long, double> VoltageTiming { get => voltageTiming; }
        public Dictionary<long, double> ExternalTriggerVoltageTiming { get =>
extTrigVoltageTiming; }
        public Dictionary<long, TimingEnum> CamCapturingTiming { get =>
camCapturingTiming; }
        /// <summary>
        /// Different type from <Ticks, Value> dataset.
        /// </summary>
        public Dictionary<int, double> DepthOfFrame { get => depthOfFrame; }
        public Dictionary<int, double> DepthRawOfFrame { get => depthRawOfFrame; }
        public Dictionary<long, double> DepthTiming { get => depthTiming; }

        public ETLCalibration eTLCalibration { get; set; }

        public enum TimingEnum
        {
            NA = 0,
            FrameBegin = 1,
            FrameEnd = 2,
            ExposureBegin = 3,
            ExposureEnd = 4,
        }

        #region Timing Data Integration
        // not clear whether this is really necessary. but maybe useful for 4d
reconstruction
        private Dictionary<long, SingleRow> mergedResultTiming;
```

```
        public class SingleRow
        {
            /// <summary>
            /// "dynamic" means arbitrary type. e.g. double (ETL), TimingEnum (Camera), ..
            /// </summary>
            public Dictionary<string, dynamic> Data;
            public long ElapsedTicks;  // same as key

            public SingleRow()
            {
                Data = new Dictionary<string, dynamic>();
                ElapsedTicks = -1;
            }

            /// <summary>
            /// Deep copy.
            /// </summary>
            /// <returns></returns>
            public SingleRow Copy()
            {
                SingleRow cp = new SingleRow();
                cp.Data = new Dictionary<string, dynamic>(this.Data);
                cp.ElapsedTicks = this.ElapsedTicks;
                return cp;
            }
        }

        /// <summary>
        /// Merges lists of Voltage, Current, CamCapturingTiming.
        /// </summary>
        /// <returns></returns>
        public Dictionary<long, SingleRow> GetMergedResultTiming()
        {
            mergedResultTiming = new Dictionary<long, SingleRow>();

            // Extract whole Keys (namely Ticks) and Merge them
            List<long> mergedKeys = new List<long>();
            mergedKeys.AddRange(voltageTiming.Keys);
            mergedKeys.AddRange(currentTiming.Keys);
            mergedKeys.AddRange(camCapturingTiming.Keys);
            mergedKeys.AddRange(depthTiming.Keys);
            mergedKeys.AddRange(extTrigVoltageTiming.Keys);
            mergedKeys = mergedKeys.Distinct().ToList();  // make keys unique
            mergedKeys.Sort();

            SingleRow rec = new SingleRow();
            foreach (long key in mergedKeys)
            {
                if (!mergedResultTiming.ContainsKey(key))
                    rec = new SingleRow();

                // create single row data
                if (voltageTiming.ContainsKey(key))
                    rec.Data[VOLTAGE] = voltageTiming[key];
                if (currentTiming.ContainsKey(key))
                    rec.Data[CURRENT] = currentTiming[key];
                if (camCapturingTiming.ContainsKey(key))
                    rec.Data[CAMERATIMING] = camCapturingTiming[key];
                if (depthTiming.ContainsKey(key))
                    rec.Data[DEPTH] = depthTiming[key];
                if (extTrigVoltageTiming.ContainsKey(key))
                    rec.Data[EXTTRIG] = extTrigVoltageTiming[key];

                if (rec.Data.Keys.Count > 0)
                {
                    rec.ElapsedTicks = key;
                    mergedResultTiming.Add(key, rec.Copy());
                }
            }
```

```
            // TODO OrderBy ??

            return mergedResultTiming;
        }
        /// <summary>
        /// Voltage of External Triggers are recorded at high time resolution,
        ///     but the huge amounts of zero-volt data are not essential.
        /// This method reduces only those excess zero-volts and doesn't change triggered
timings.
        /// </summary>
        public void ReduceExcessTriggerRecord(double timeResolution_ms = 5)
        {
            Dictionary<long, double> adjustedVoltage = new Dictionary<long, double>();

            // 1 tick = 0.1 micro sec = 1e-4 ms;    1 ms = 1e4 ticks
            long interval = (long)(timeResolution_ms * 1e4);

            long lastTickAdded = 0;
            foreach(long tick in extTrigVoltageTiming.Keys)
            {
                if (tick >= lastTickAdded + interval
                 || extTrigVoltageTiming[tick] > 0.5
                 || lastTickAdded == 0)
                {
                    adjustedVoltage.Add(tick, extTrigVoltageTiming[tick]);
                    lastTickAdded = tick;
                }
            }
            int reduced = extTrigVoltageTiming.Keys.Count - adjustedVoltage.Keys.Count;
            Console.WriteLine($"Removed {reduced} zero-volt samples. (Voltage of External
Trigger)");
            extTrigVoltageTiming = adjustedVoltage;
        }
        public void SetCameraStatesFromExternalTriggerRecord(double TriggerThresholdVolt =
3.0, double TriggerLength_ms = 0.1, double SafetyDelay_ms = 0.1)
        {
            long lastTick = 0;
            foreach (long tick in extTrigVoltageTiming.Keys)
            {
                if (extTrigVoltageTiming[tick] > TriggerThresholdVolt)  // TODO: not sure
the threshold of this trigger
                {
                    if (tick - lastTick > (TriggerLength_ms + SafetyDelay_ms) * 1e4
                     || lastTick == 0)
                    {
                        if (camCapturingTiming.ContainsKey(tick)) continue;  // temporary
                        camCapturingTiming.Add(tick, TimingEnum.FrameBegin);
                        lastTick = tick;
                    }
                }
            }
        }
        /// <summary>
        /// Reduce data amount (mainly voltage/current) before first frame
        /// </summary>
        public long GetStartTickBeforeFirstFrame(double TimeFromOnset_ms = 1000)
        {
            // 1 tick = 0.1 micro sec = 1e-4 ms;    1 ms = 1e4 ticks
            long timeFromOnset_tick = (long)(TimeFromOnset_ms * 1e4);

            long tickFirstFrame = -1;
            if (camCapturingTiming.Count > 0)
            {
                foreach(long tick in camCapturingTiming.Keys)
                {
                    if (camCapturingTiming[tick] == TimingEnum.FrameBegin
                        && (tick < tickFirstFrame || tickFirstFrame == -1))
                        tickFirstFrame = tick;
                }
```

```
            }
            return Math.Max(0, tickFirstFrame - timeFromOnset_tick);
        }

        /// <summary>
        /// Used for live reconstruction (2D projection).
        /// </summary>
        public void GetLogsNewerThanGivenTick(long tick, string KEY)
        {
            if(KEY == VOLTAGE)
            {

            }
            else if(KEY == EXTTRIG)
            {
            }


        }
        #endregion

        /// <summary>
        /// Constructor
        /// </summary>
        public RecordingLog()
        {
            currentTiming = new Dictionary<long, double>();
            voltageTiming = new Dictionary<long, double>();
            camCapturingTiming = new Dictionary<long, TimingEnum>();
            depthTiming = new Dictionary<long, double>();
            depthOfFrame = new Dictionary<int, double>();
            depthRawOfFrame = new Dictionary<int, double>();
            extTrigVoltageTiming = new Dictionary<long, double>();
        }
        public void Clear()
        {
            currentTiming.Clear();
            voltageTiming.Clear();
            camCapturingTiming.Clear();
            depthTiming.Clear();
            depthOfFrame.Clear();
            depthRawOfFrame.Clear();
            extTrigVoltageTiming.Clear();
        }

        #region Add data
        public void AddCurrent(long tick, double current)
        {
            if (!currentTiming.ContainsKey(tick))
                currentTiming.Add(tick, current);
        }
        public void AddVoltage(long tick, double voltage)
        {
            if (!voltageTiming.ContainsKey(tick))
                voltageTiming.Add(tick, voltage);
        }
        public void AddExternalTriggerVoltage(long tick, double voltage)
        {
            if (!extTrigVoltageTiming.ContainsKey(tick))
                extTrigVoltageTiming.Add(tick, voltage);
        }
        public void AddCamTiming(long tick, TimingEnum state)
        {
            if (!camCapturingTiming.ContainsKey(tick))
                camCapturingTiming.Add(tick, state);
        }
        public void AddDepthTiming(long tick, double depthMicron)
        {
            if(!depthTiming.ContainsKey(tick))
```

```
                depthTiming.Add(tick, depthMicron);
        }
        public void AddDepthOfFrame(int frame, double depthMicron)
        {
            if(!depthOfFrame.ContainsKey(frame))
                depthOfFrame.Add(frame, depthMicron);
        }
        public void AddDepthOfFrame(int frame, double depthMicron, double depthRawValue)
        {
            if (!depthOfFrame.ContainsKey(frame))
                depthOfFrame.Add(frame, depthMicron);
            if (!depthRawOfFrame.ContainsKey(frame))
                depthRawOfFrame.Add(frame, depthRawValue);
        }
        public void AddDepthRawOfFrame(int frame, double depthRawValue)
        {
            if (!depthRawOfFrame.ContainsKey(frame))
                depthRawOfFrame.Add(frame, depthRawValue);
        }
        public void AddDepthOfFrame_Override(int frame, double depthMicron)
        {
            // test override (for xyz w/ FocusDriveMotor. 4 Jun 2021)
            //if (depthOfFrame.ContainsKey(frame))
            //    depthOfFrame.Remove(frame);
            if (!depthOfFrame.ContainsKey(frame))
                depthOfFrame.Add(frame, depthMicron);
        }
        #endregion

        #region WriteOut
        public void WriteOutAllData(string file = "./all.csv")
        {
            // create header (Ticks, Voltage, Current, CameraState, FrameNumber)
            string header = string.Format("{0},{1},{2},{3},{4},{5},{6}",
                TICKS, VOLTAGE, CURRENT, CAMERATIMING, FRAME, DEPTH, EXTTRIG);

            var mergedResult = GetMergedResultTiming();

            // ignore records which is too early before capturing
            long startTick = GetStartTickBeforeFirstFrame(TimeFromOnset_ms: 1000);

            StreamWriter sw;
            using (sw = new StreamWriter(file, false))
            {
                sw.WriteLine(header);
                int frameNumber = -1;  // * The FrameNumber begins at 0, see following code
                foreach (var keyTick in mergedResult.Keys)
                {
                    if (keyTick < startTick)
                        continue;

                    SingleRow row = mergedResult[keyTick];
                    string str = "";
                    double voltage = double.NaN, current = double.NaN, depth = double.NaN;
                    double extTrigVoltage = double.NaN;
                    string camState = "NaN";

                    foreach(string keyColumn in KEYS)
                    {
                        if (row.Data.ContainsKey(keyColumn))
                        {
                            switch (keyColumn)
                            {
                                case VOLTAGE:
                                    voltage = (double)row.Data[keyColumn];
                                    break;
                                case CURRENT:
                                    current = (double)row.Data[keyColumn];
                                    break;
```

```
                                case CAMERATIMING:
                                    camState = Enum.GetName(typeof(TimingEnum),
row.Data[keyColumn]);
                                    if (row.Data[keyColumn] == TimingEnum.FrameBegin)
                                        frameNumber++;
                                    break;
                                case DEPTH:
                                    depth = (double)row.Data[keyColumn];
                                    break;
                                case EXTTRIG:
                                    extTrigVoltage = (double)row.Data[keyColumn];
                                    break;
                            }
                        }
                    }

                    // Add FrameNumber only for CameraTiming (e.g. FrameBegin)
                    // considering timings of XYZ capturing
                    if (!row.Data.ContainsKey(CAMERATIMING))
                        str = string.Format("{0},{1},{2},{3},{4},{5},{6}",
                            keyTick.ToString(), voltage.ToString(), current.ToString(),
camState,
                            double.NaN.ToString(), depth.ToString(),
extTrigVoltage.ToString());
                    else
                        str = string.Format("{0},{1},{2},{3},{4},{5},{6}",
                            keyTick.ToString(), voltage.ToString(), current.ToString(),
camState,
                            frameNumber.ToString(), depth.ToString(),
extTrigVoltage.ToString());

                    sw.WriteLine(str);
                }
            }
        }

        /// <summary>
        /// Output important data for recording with free format. Temporary
        /// </summary>
        /// <param name="file"></param>
        public void WriteOutMetadata(string file = "./metadata.txt",
            OperatorForm operatorForm=null, ETLSetting eTLSetting = null, CameraSetting
cameraSetting = null,
            PositionControllerSetting positionControllerSetting = null)
        {
            string ret = Environment.NewLine;

            // Operator form. Capturing
            string strOp = "";
            // capture mode
            if (operatorForm.radioButton_capmode_xyt.Checked)
                strOp += $"OpCaptureMode=XYT" + ret;
            else if (operatorForm.radioButton_capmode_xyz.Checked)
                strOp += $"OpCaptureMode=XYZ" + ret;
            else if (operatorForm.radioButton_capmode_xyzt.Checked)
                strOp += $"OpCaptureMode=XYZT" + ret;
            // axial scanning
            if (operatorForm.radioButton_axial_none.Checked)
                strOp += $"OpAxialScan=None" + ret;
            else if (operatorForm.radioButton_axial_mac6.Checked)
                strOp += $"OpAxialScan=FocusDriveMotor" + ret;
            else if (operatorForm.radioButton_axial_etl_continuous.Checked)
                strOp += $"OpAxialScan=ETL_Continuous" + ret;
            else if (operatorForm.radioButton_axial_etl_intermittent.Checked)
                strOp += $"OpAxialScan=ETL_Intermittent" + ret;

            // ETL setting form
            string strETL =
                $"ETLSwingFrequency={eTLSetting.numericUpDown_frequency.Value:F2}" + ret +
```

```
                $"ETLCurrentLower={eTLSetting.numericUpDown_lower_current.Value:F0}" + ret +
                $"ETLCurrentUpper={eTLSetting.numericUpDown_upper_current.Value:F0}" + ret +
                $"ETLEnabledAnalogControl={eTLSetting.checkBox_analogsignal.Checked}" + ret
+
                $"ETLAOWaveform={eTLSetting.comboBox_ao_waveform.Text}" + ret +
                $"ETLUsedCalibrationInfo={eTLSetting.checkBox_use_calibration_info.Checked}"
+ ret +
                $"ETLCalibrationCoefficient={eTLSetting.numericUpDown_calib_coef.Value:F2}"
+ ret +
                $"ETLCalibrationConstant={eTLSetting.numericUpDown_calib_const.Value:F2}" +
ret;

            // Camera form
            string strCam =
                $"CameraName={cameraSetting.comboBox_camera.Text}" + ret +
                $"CamBitDepth={cameraSetting.comboBox_bit_depth.Text}" + ret +
                $"CamPixels={cameraSetting.comboBox_subarray.Text}" + ret +
                $"CamBinning={cameraSetting.comboBox_bin.Text}" + ret +
                $"CamExposureTime_ms={cameraSetting.numericUpDown_exposure.Value}" + ret +
                $"CamExposureShutterMode={cameraSetting.comboBox_global_exposure.Text}" +
ret +
                $"CamReadoutSpeed={cameraSetting.comboBox_readout_speed.Text}" + ret +
                $"CamAllocatedFrames={cameraSetting.numericUpDown_max_frames.Value}" + ret +
                // Trigger
                $"CamTriggerSource={cameraSetting.comboBox_trigger_source.Text}" + ret +
                $"CamTriggerMode={cameraSetting.comboBox_trigger_active.Text}" + ret +
                $"CamTriggerConnector={cameraSetting.comboBox_trigger_connector.Text}" + ret
+
                $"CamTriggerPolarity={cameraSetting.comboBox_active_polarity.Text}" + ret +
                // External Trigger

$"CamEnabledExtTrigger={cameraSetting.checkBox_external_trigger_by_nidaq.Checked}" + ret +
                $"CamExtTriggerAOChannel={cameraSetting.comboBox_ext_trig_aochannel.Text}" +
ret +
                $"CamExtTriggerAIChannel={cameraSetting.comboBox_ext_trig_aichannel.Text}" +
ret +

$"CamLevelTriggerHighDuration={cameraSetting.numericUpDown_leveltrigger_high_length.Value}"
+ ret +

$"CamLevelTriggerLowDuration={cameraSetting.numericUpDown_leveltrigger_low_length.Value}" +
ret;

            // Position Controller form
            string strPosCon =

$"PosConZMultiplierPosDirection={positionControllerSetting.numericUpDown_ReadValuePerMicron
Pos.Value:F2}" + ret +

$"PosConZMultiplierNegDirection={positionControllerSetting.numericUpDown_ReadValuePerMicron
Neg.Value:F2}" + ret +

$"PosConZChangingSpeed={positionControllerSetting.numericUpDown_z_speed.Value}" + ret +

$"PosConZStackTopShallow={positionControllerSetting.numericUpDown_zstack_top.Value}" + ret
+

$"PosConZStackBottomDeep={positionControllerSetting.numericUpDown_zstack_bottom.Value}" +
ret +

$"PosConZStackZStep={positionControllerSetting.numericUpDown_zstack_zstep.Value:F1}" + ret;
            if (positionControllerSetting.radioButton_zstack_bottom_to_top.Checked)
                strPosCon += $"PosConZStackCaptureDirection=BottomToTop" + ret;
            else if (positionControllerSetting.radioButton_zstack_top_to_bottom.Checked)
                strPosCon += $"PosConZStackCaptureDirection=TopToBottom" + ret;

            string[] strs = { strOp, strCam, strETL, strPosCon };

            StreamWriter sw;
```

```
            using (sw = new StreamWriter(file, false))
            {
                foreach (string s in strs)
                    sw.Write(s);
            }
        }

        /// <summary>
        /// Create csv file.
        /// </summary>
        public void WriteOutVoltage(string file= "./voltage_out.csv")
        {
            string header = string.Format("{0},{1}", TICKS, VOLTAGE);

            StreamWriter sw;
            using(sw = new StreamWriter(file, false))
            {
                // header
                sw.WriteLine(header);
                foreach (long key in voltageTiming.Keys)
                {
                    string line = string.Format("{0},{1}", key.ToString(),
voltageTiming[key].ToString());
                    sw.WriteLine(line);
                }
            }
        }
        public void WriteOutCurrent(string file = "./current_out.csv")
        {
            string header = string.Format("{0},{1}", TICKS, CURRENT);

            StreamWriter sw;
            using (sw = new StreamWriter(file, false))
            {
                // header
                sw.WriteLine(header);
                foreach (long key in currentTiming.Keys)
                {
                    string line = string.Format("{0},{1}", key.ToString(),
currentTiming[key].ToString());
                    sw.WriteLine(line);
                }
            }
        }
        public void WriteOutExternalTriggerVoltage(string file =
"./ext_trig_voltage_out.csv")
        {
            string header = string.Format("{0},{1}", TICKS, EXTTRIG);

            StreamWriter sw;
            using (sw = new StreamWriter(file, false))
            {
                // header
                sw.WriteLine(header);
                foreach (long key in extTrigVoltageTiming.Keys)
                {
                    string line = string.Format("{0},{1}", key.ToString(),
extTrigVoltageTiming[key].ToString());
                    sw.WriteLine(line);
                }
            }
        }

        public void WriteOutCamTiming(string file = "./cam_timing_out.csv")
        {
            string header = string.Format("{0},{1}", TICKS, CAMERATIMING);

            StreamWriter sw;
            using (sw = new StreamWriter(file, false))
```

```
            {
                // header
                sw.WriteLine(header);
                foreach (long key in camCapturingTiming.Keys)
                {
                    string line = string.Format("{0},{1}", key.ToString(),
camCapturingTiming[key].ToString());
                    sw.WriteLine(line);
                }
            }
        }
        public void WriteOutDepthOfFrame(string file = "./depth_of_frame_out.csv")
        {
            // header
            string header = string.Format("{0},{1}", FRAME, DEPTH);
            if (depthRawOfFrame.Count > 0)
                header += string.Format(",{0}", DEPTHRAW);

            StreamWriter sw;
            using (sw = new StreamWriter(file, false))
            {
                sw.WriteLine(header);
                int frameNumber = 0;
                foreach (int key in depthOfFrame.Keys)
                {
                    frameNumber++;
                    //string line = string.Format("{0},{1}", key.ToString(),
depthOfFrame[key].ToString());
                    string line = string.Format("{0},{1}", frameNumber.ToString(),
depthOfFrame[key].ToString());
                    if (depthRawOfFrame.ContainsKey(key))
                        line += string.Format(",{0}", depthRawOfFrame[key].ToString());
                    sw.WriteLine(line);
                }
            }
        }
        #endregion
    }
}
```

Code A26. Utils.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO.Ports;
using System.IO;
using System.Runtime.InteropServices;

using csAcq4;
using Hamamatsu.DCAM4;
using System.Management;
using System.Diagnostics;

namespace ETL_system
{
    public static class Utils
    {
        /// <summary>
        /// saving last serial number of recording
        /// </summary>
        public const string serialFile = "expserialnumber.ini";

        public static Label ConnectionLabelHelper(bool connected, Label label)
        {
            Label new_font = label;
            if (connected)
            {
                new_font.Text = "Connected";
                new_font.ForeColor = Color.Blue;
            }
            else
            {
                new_font.Text = "Disconnected";
                new_font.ForeColor = Color.DarkRed;
            }
            return new_font;
        }

        public static string ConnectionButtonHelper(bool connected)
        {
            // the Button control is called by reference, to keep its events
            string next_action = "Connect";
            if (connected)
                next_action = "Disconnect";

            return next_action;
        }

        public static (int, int) GetMinMaxIntensityOfImage_Slow(Bitmap Img, int
MaxIntensity)
        {
            // Simplest way
            int max=0, min=MaxIntensity;
            for (int x = 0; x < Img.Width; x++)
            {
                for (int y = 0; y < Img.Height; y++)
                {
                    int intensity = (int)(Img.GetPixel(x, y).GetBrightness() *
MaxIntensity);
                    if (intensity > max)
```

```
                    max = intensity;
                if (intensity < min)
                    min = intensity;
            }
        }
        return (min, max);
    }
    public static (int, int) GetMinMaxIntensityOfImage(DCAMBUF_FRAME dcamBuf)
    {
        // 23 to 25 ms for frame 2304x2304 px
        // < 1 ms for 1024x1024 px
        //Stopwatch sw = new Stopwatch();
        //sw.Start();
        int min = 65535, max = 0;
        unsafe
        {
            // 16 bit = 2 byte
            int bytePerPixel = 2;
            IntPtr ptr = (IntPtr)(dcamBuf.buf.ToInt64());
            for (int i = 0; i < dcamBuf.rowbytes * dcamBuf.height; i += bytePerPixel)
            {
                ushort* intensity = (ushort*)(ptr + i);
                if (*intensity > max)
                    max = *intensity;
                if (*intensity < min)
                    min = *intensity;
            }
        }
        //sw.Stop();
        //Console.WriteLine($"{min}, {max} ... {sw.ElapsedMilliseconds} ms");
        return (min, max);
    }

    public static string ConvertHex(string hexString__, string sep="-")
    {
        try
        {
            string ascii = string.Empty;

            string hexString = hexString__.Replace(sep, "");

            for (int i = 0; i < hexString.Length; i += 2)
            {
                string hs = string.Empty;

                hs = hexString.Substring(i, 2);
                uint decval = System.Convert.ToUInt32(hs, 16);
                char character = System.Convert.ToChar(decval);
                ascii += character;
            }

            return ascii;
        }
        catch (Exception ex) { Console.WriteLine(ex.Message); }

        return string.Empty;
    }

    public static Dictionary<string, string> GetSerialPortList()
    {
        // Get serial ports
        //System.Management.ManagementClass mcW32serPort = new
System.Management.ManagementClass("Win32_SerialPort");
        var ports_dic = new Dictionary<string, string>();
        string[] nameList = Utils.GetDeviceNames();
        string[] deviceIDs = SerialPort.GetPortNames();

        //foreach (ManagementObject port in mcW32serPort.GetInstances())  // This method
could not work on some devices
```

```
            foreach (string name in nameList)
            {
                foreach (string id in deviceIDs)
                {
                    if (name.IndexOf("(" + id + ")") >= 0)
                    {
                        ports_dic.Add(name, id);
                        //comboBox_port.Items.Add(name);
                    }
                }
            }
            return ports_dic;
        }

        /// <summary>
        /// Get COM Port IDs and Names. Ref:
http://truthfullscore.hatenablog.com/entry/2014/01/10/180608
        /// </summary>
        /// <returns></returns>
        public static string[] GetDeviceNames()
        {
            List<string> deviceNameList = new List<string>();
            System.Text.RegularExpressions.Regex check = new
System.Text.RegularExpressions.Regex("(COM[1-9][0-9]?[0-9]?)");

            ManagementClass mcPnPEntity = new ManagementClass("Win32_PnPEntity");
            ManagementObjectCollection manageObjCol = mcPnPEntity.GetInstances();

            foreach (ManagementObject manageObj in manageObjCol)
            {
                string name = manageObj.GetPropertyValue("Name") as string;

                if (name != null && check.IsMatch(name))
                {
                    deviceNameList.Add(name);
                }
            }

            return (deviceNameList.ToArray());
        }

        /// <summary>
        /// In order to save captured images without overriding, generate unique file
name/path.
        /// </summary>
        /// <returns></returns>
        public static string CreateNewFilePath(string SaveDir, string FileName, bool
ReturnFullPath=true)
        {
            if (!Directory.Exists(SaveDir))
            {
                MessageBox.Show("Invalid directory.");
                return string.Empty;
            }

            string newFileName = FileName;

            // remember last number
            string serialFilePath = Path.Combine(SaveDir, serialFile);
            if (File.Exists(serialFilePath))
            {
                using (StreamReader sr = new StreamReader(serialFilePath))
                {
                    if(int.TryParse(sr.ReadLine().Trim(), out int count))
                    {
                        do
                        {
                            count++;
                            string countStr = string.Format("{0:D4}", count);
```

```
                        newFileName = countStr + "_" + FileName;
                    } while (File.Exists(newFileName));
                }
            }
        }

        if (newFileName == FileName)
        {
            for (int i = 0; i < 100000; i++)  // temporary 99999 max
            {
                bool flagBreak = true;
                string countStr = string.Format("{0:D4}", i);
                newFileName = countStr + "_" + FileName;
                foreach (string f in Directory.GetFiles(SaveDir,
                    searchPattern: countStr, searchOption:
SearchOption.TopDirectoryOnly))
                {
                    if (f.IndexOf(countStr) == 0)
                    {
                        flagBreak = false;
                        break;
                    }
                }
                if (flagBreak)
                {
                    if (!File.Exists(newFileName))
                        break;
                }
            }
        }

        if(ReturnFullPath)
            return Path.Combine(SaveDir, newFileName);
        else
            return newFileName;
    }

    public static bool SaveLastRecordingNumber(string SaveDir, string LastFileName)
    {
        int count = 0;
        // using filename
        int numName = 0, numFile = 0;
        for(int i=0; i<LastFileName.Length; i++)
        {
            if (int.TryParse(LastFileName.Substring(0, i + 1), out numName))
                count = numName;
            else
                break;
        }

        // using ini file
        string serialFilePath = Path.Combine(SaveDir, serialFile);
        if (File.Exists(serialFilePath))
        {
            using (StreamReader sr = new StreamReader(serialFilePath))
            {
                if (int.TryParse(sr.ReadLine().Trim(), out numFile))
                    numFile++;
            }
        }

        // larger number is saved
        if (numFile > numName)
            count = numFile;
        else
            count = numName;

        using (StreamWriter sw = new StreamWriter(serialFilePath))
        {
```

```
                sw.Write(count.ToString());
            }
            return true;
        }

        public static RotateFlipType GetRotateFlipType(int angle, bool flipX, bool flipY)
        {
            RotateFlipType r = RotateFlipType.RotateNoneFlipNone;
            if(flipX && flipY)
            {
                switch(angle)
                {
                    case 0:
                        r = RotateFlipType.RotateNoneFlipXY;
                        break;
                    case 90:
                        r = RotateFlipType.Rotate90FlipXY;
                        break;
                    case 180:
                        r = RotateFlipType.Rotate180FlipXY;
                        break;
                    case 270:
                        r = RotateFlipType.Rotate270FlipXY;
                        break;
                    case 360:
                        r = RotateFlipType.RotateNoneFlipXY;
                        break;
                }
            }
            else if (flipX && !flipY)
            {
                switch (angle)
                {
                    case 0:
                        r = RotateFlipType.RotateNoneFlipX;
                        break;
                    case 90:
                        r = RotateFlipType.Rotate90FlipX;
                        break;
                    case 180:
                        r = RotateFlipType.Rotate180FlipX;
                        break;
                    case 270:
                        r = RotateFlipType.Rotate270FlipX;
                        break;
                    case 360:
                        r = RotateFlipType.RotateNoneFlipX;
                        break;
                }
            }
            else if (!flipX && flipY)
            {
                switch (angle)
                {
                    case 0:
                        r = RotateFlipType.RotateNoneFlipY;
                        break;
                    case 90:
                        r = RotateFlipType.Rotate90FlipY;
                        break;
                    case 180:
                        r = RotateFlipType.Rotate180FlipY;
                        break;
                    case 270:
                        r = RotateFlipType.Rotate270FlipY;
                        break;
                    case 360:
                        r = RotateFlipType.RotateNoneFlipY;
                        break;
```

```
            }
        }
        else
        {
            switch (angle)
            {
                case 0:
                    r = RotateFlipType.RotateNoneFlipNone;
                    break;
                case 90:
                    r = RotateFlipType.Rotate90FlipNone;
                    break;
                case 180:
                    r = RotateFlipType.Rotate180FlipNone;
                    break;
                case 270:
                    r = RotateFlipType.Rotate270FlipNone;
                    break;
                case 360:
                    r = RotateFlipType.RotateNoneFlipNone;
                    break;
            }
        }

        return r;
    }

    /// <summary>
    /// Pending. DOES NOT WORK
    /// </summary>
    /// <param name="src_ARGB32"></param>
    /// <returns></returns>
    public static Bitmap ConvertBitmapFormatToGrey16(Bitmap src_ARGB32)
    {
        Bitmap bmpGrey16 = new Bitmap(src_ARGB32.Width, src_ARGB32.Height,
System.Drawing.Imaging.PixelFormat.Format16bppGrayScale );
        Rectangle rect = new Rectangle(0, 0, src_ARGB32.Width, src_ARGB32.Height);
        BitmapData dst = bmpGrey16.LockBits(rect,
System.Drawing.Imaging.ImageLockMode.ReadWrite,
System.Drawing.Imaging.PixelFormat.Format16bppGrayScale);
        if(src_ARGB32.PixelFormat == System.Drawing.Imaging.PixelFormat.Format32bppArgb)
        {
            src_ARGB32.LockBits(rect, System.Drawing.Imaging.ImageLockMode.ReadOnly,
System.Drawing.Imaging.PixelFormat.Format32bppArgb);
            Color c;
            for (int y = 0; y < src_ARGB32.Height; y++)
            {
                for (int x = 0; x < src_ARGB32.Width; x++)
                {
                    c = src_ARGB32.GetPixel(x, y);
                    int avg = (c.R + c.G + c.B) / 3;
                    //newBmp.SetPixel(x, y, Color.FromArgb(c.A, avg, avg, avg));

                }
            }
        }
        //bmpGrey16.UnlockBits()
        return bmpGrey16;
    }

    /// <summary>
    /// Copied from https://docs.microsoft.com/ja-
jp/dotnet/api/system.drawing.imaging.encoder.compression?view=dotnet-plat-ext-6.0
    /// </summary>
    public static EncoderParameters GetEncoderParametersToSaveTiff(bool
EnableCompressionLZW = true)
    {
        System.Drawing.Imaging.Encoder myEncoder;
        EncoderParameter myEncoderParameter;
```

```
            EncoderParameters myEncoderParameters;

            // Create an Encoder object based on the GUID
            // for the Compression parameter category.
            myEncoder = System.Drawing.Imaging.Encoder.Compression;

            // Create an EncoderParameters object.
            // An EncoderParameters object has an array of EncoderParameter
            // objects. In this case, there is only one
            // EncoderParameter object in the array.
            myEncoderParameters = new EncoderParameters(1);

            // Save the bitmap as a TIFF file with LZW compression.
            if(EnableCompressionLZW)
                myEncoderParameter = new EncoderParameter(
                    myEncoder,
                    (long)EncoderValue.CompressionLZW);
            else
                myEncoderParameter = new EncoderParameter(
                    myEncoder,
                    (long)EncoderValue.CompressionNone);

            myEncoderParameters.Param[0] = myEncoderParameter;

            return myEncoderParameters;
        }
        /// <summary>
        /// Copied from https://docs.microsoft.com/ja-
jp/dotnet/api/system.drawing.image.save?view=dotnet-plat-ext-6.0
        /// </summary>
        /// <param name="mimeType"></param>
        /// <returns></returns>
        public static ImageCodecInfo GetEncoderInfo(string mimeType)
        {
            int j;
            ImageCodecInfo[] encoders;
            encoders = ImageCodecInfo.GetImageEncoders();
            for (j = 0; j < encoders.Length; ++j)
            {
                if (encoders[j].MimeType == mimeType)
                    return encoders[j];
            }
            return null;
        }
    }
}
```