libuca Documentation

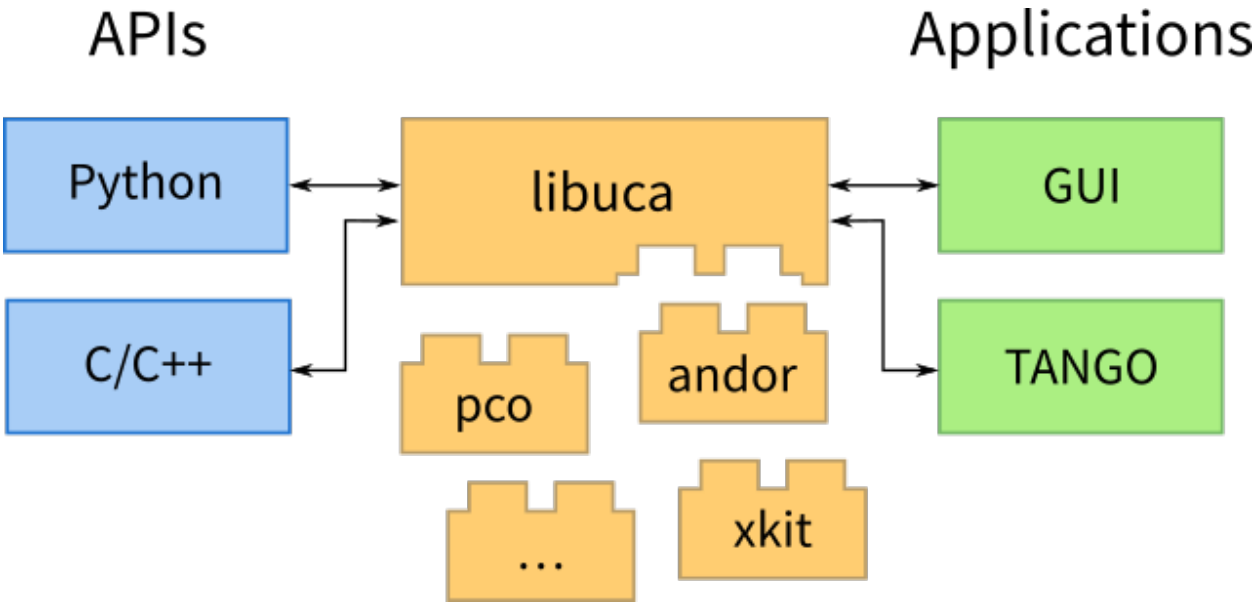
Release 2.3.0

Matthias Vogelgesang

Apr 20, 2018

Contents

libuca is a light-weight camera abstraction library written in C and GObject, focused on scientific cameras used at the ANKA synchrotron.



CHAPTER 1

Contents

1.1 Quickstart

1.1.1 Installation

Before installing *libuca* itself, you should install any drivers and SDKs needed to access the cameras you want to access through *libuca*. Now you have two options: install pre-built packages or build from source.

Installing packages

Packages for the core library and all plugins are currently provided for openSUSE and can be obtained from the openSUSE Build Service at https://build.opensuse.org/package/show/home:ufo-kit/libuca.

Building on Linux

In order to build *libuca* from source, you need

- CMake,
- a C compiler (currently tested with gcc and clang),
- GLib and GObject development libraries and
- any required camera SDKs.

For the base system, install

```
[Debian] sudo apt-get install libglib2.0 cmake gcc
[openSUSE] sudo zypper in glib2-devel cmake gcc
```

In case you want to use the graphical user interface you also need the Gtk+ development libraries:

```
[Debian] sudo apt-get install libgtk+2.0-dev
[openSUSE] sudo zypper in gtk2-devel
```

To generate bindings for third-party languages, you have to install

```
[Debian] sudo apt-get install gobject-introspection
[openSUSE] sudo zypper in gobject-introspection-devel
```

Fetching the sources

Clone the repository

```
git clone https://github.com/ufo-kit/libuca
```

or download the latest release at https://github.com/ufo-kit/libuca/releases and unzip the `.zip` file:

```
unzip libuca-x.y.z.zip
```

or untar the `.tar.gz` file:

```
tar -zxvf libuca-x.y.z.tar.gz
```

and create a new, empty build directory inside:

```
cd libuca/
mkdir build
```

Configuring and building

Now you need to create the Makefile with CMake. Go into the build directory and point CMake to the `libuca` top-level directory:

```
cd build/
cmake ..
```

As long as the last line reads "Build files have been written to", the configuration stage is successful. In this case you can build `libuca` with

```
make
```

and install with

```
sudo make install
```

If an *essential* dependency could not be found, the configuration stage will stop and build files will not be written. If a *non-essential* dependency (such as a certain camera SDK) is not found, the configuration stage will continue but that particular camera support not built.

If you want to customize the build process you can pass several variables to CMake:

```
cmake .. -DPREFIX=/usr -DLIBDIR=/usr/lib64
```

The former tells CMake to install into `/usr` instead of `/usr/local` and the latter that we want to install the libraries and plugins into the `lib64` subdir instead of the default `lib` subdir as it is common on SUSE systems.

Building on Windows

Using MSYS2, the build procedure is similar to Linux but differs in some points. First, download msys2-<arch>-<release-date>.exe from msys2.org (preferably the x86_64 variant) and install it to `C:\msys64` or any other location.

Run the MSYS2 MinGW shell from the start menu and update the core if this is the first time using:

```
pacman -Syu
```

Close the terminal and open a new shell again. Install all required dependencies with:

```
pacman -S gcc make cmake pkg-config git glib2-devel gettext-devel
```

Clone libuca and any plugins you want to use on Windows:

```
git clone https://github.com/ufo-kit/libuca
```

and create an empty `build` directory in libuca's root folder. Change directory to that folder, configure libuca using CMake and build and install it:

```
cd libuca
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=/usr ..
make && make install
```

Before proceeding with the plugins you *must* soft link the library to fit the naming scheme:

```
ln -s /usr/bin/libuca.so /usr/lib/libuca.dll.a
```

To build plugins nothing special is required. Clone the repository, create an empty build directory, configure and build:

```
git clone https://github.com/ufo-kit/uca-net
cd uca-net
mkdir build && cd build
cmake ..
make && make install
```

1.1.2 Usage

The API for accessing cameras is straightforward. First you need to include the necessary header files:

```
#include <glib-object.h>
#include <uca/uca-plugin-manager.h>
#include <uca/uca-camera.h>
```

Then you need to setup the type system:

```
int
main (int argc, char *argv[])
{
    UcaPluginManager *manager;
    UcaCamera *camera;
    GError *error = NULL; /* this _must_ be set to NULL */

#if !(GLIB_CHECK_VERSION (2, 36, 0))
    g_type_init();
#endif
```

Now you can instantiate new camera *objects*. Each camera is identified by a human-readable string, in this case we want to access any pco camera that is supported by libpco. To instantiate a camera we have to create a plugin manager first:

```
manager = uca_plugin_manager_new ();
camera = uca_plugin_manager_get_camera (manager, "pco", &error, NULL);
```

Errors are indicated with a returned value `NULL` and `error` set to a value other than `NULL`:

```
if (camera == NULL) {
    g_error ("Initialization: %s", error->message);
    return 1;
}
```

You should always remove the reference from the camera object when not using it in order to free all associated resources:

```
    g_object_unref (camera);
    return 0;
}
```

Compile this program with

```
cc `pkg-config --cflags --libs libuca glib-2.0` foo.c -o foo
```

Now, run `foo` and verify that no errors occur.

Grabbing frames

To synchronously grab frames, first start the camera:

```
uca_camera_start_recording (camera, &error);
g_assert_no_error (error);
```

Now, you have to allocate a suitably sized buffer and pass it to `uca_camera_grab`:

```
gpointer buffer = g_malloc0 (640 * 480 * 2);

uca_camera_grab (camera, buffer, &error);
```

You have to make sure that the buffer is large enough by querying the size of the region of interest and the number of bits that are transferred.

Getting and setting camera parameters

Because camera parameters vary tremendously between different vendors and products, they are realized with so-called GObject *properties*, a mechanism that maps string keys to typed and access restricted values. To get a value, you use the `g_object_get` function and provide memory where the result is stored:

```
guint roi_width;
gdouble exposure_time;

g_object_get (G_OBJECT(camera),
              "roi-width", &roi_width,
              "exposure-time", &exposure_time,
```

```
                /* The NULL marks the end! */
                NULL
                );

g_print ("Width of the region of interest: %d\n", roi_width);
g_print ("Exposure time: %3.5fs\n", exposure_time);
```

In a similar way, properties are set with `g_object_set`:

```
guint roi_width = 512;
gdouble exposure_time = 0.001;

g_object_set (G_OBJECT (camera),
              "roi-width", roi_width,
              "exposure-time", exposure_time,
              NULL);
```

Each property can be associated with a physical unit. To query for the unit call `uca_camera_get_unit` and pass a property name. The function will then return a value from the `UcaUnit` enum.

1.2 Supported cameras

The following cameras are supported:

- pco.edge, pco.dimax, pco.4000 (all CameraLink) via libpco. You need to have the SiliconSoftware frame grabber SDK with the `menable` kernel module installed.
- PhotonFocus
- Pylon
- UFO Camera developed at KIT/IPE.

A remote access is available for `libuca` cameras.

1.2.1 Property documentation

mock

string name Name of the camera

Default:

unsigned int sensor-width Width of the sensor in pixels

Default: 512
Range: [1, 4294967295]

unsigned int sensor-height Height of the sensor in pixels

Default: 512
Range: [1, 4294967295]

double sensor-pixel-width Width of sensor pixel in meters

Default: 1e-05
Range: [2.22507385851e-308, 1.79769313486e+308]

double sensor-pixel-height Height of sensor pixel in meters

Default: 1e-05
Range: [2.22507385851e-308, 1.79769313486e+308]

unsigned int sensor-bitdepth Number of bits per pixel

Default: 8
Range: [1, 32]

unsigned int sensor-horizontal-binning Number of sensor ADCs that are combined to one pixel in horizontal direction

Default: 1
Range: [1, 4294967295]

unsigned int sensor-vertical-binning Number of sensor ADCs that are combined to one pixel in vertical direction

Default: 1
Range: [1, 4294967295]

None trigger-source Trigger source

Default: <enum UCA_CAMERA_TRIGGER_SOURCE_AUTO of type UcaCameraTriggerSource>

None trigger-type Trigger type

Default: <enum UCA_CAMERA_TRIGGER_TYPE_EDGE of type UcaCameraTriggerType>

double exposure-time Exposure time in seconds

Default: 1.0
Range: [0.0, 1.79769313486e+308]

double frames-per-second Frames per second

Default: 1.0
Range: [2.22507385851e-308, 1.79769313486e+308]

unsigned int roi-x0 Horizontal coordinate

Default: 0
Range: [0, 4294967295]

unsigned int roi-y0 Vertical coordinate

Default: 0
Range: [0, 4294967295]

unsigned int roi-width Width of the region of interest

Default: 1
Range: [1, 4294967295]

unsigned int roi-height Height of the region of interest

Default: 1
Range: [1, 4294967295]

unsigned int roi-width-multiplier Minimum possible step size of horizontal ROI

Default: 1
Range: [1, 4294967295]

unsigned int roi-height-multiplier Minimum possible step size of vertical ROI

Default: 1
Range: [1, 4294967295]

bool has-streaming Is the camera able to stream the data

Default: True

bool has-camram-recording Is the camera able to record the data in-camera

Default: False

unsigned int recorded-frames Number of frames recorded into internal camera memory

Default: 0
Range: [0, 4294967295]

bool transfer-asynchronously Specify whether data should be transfered asynchronously using a specified callback

Default: False

bool is-recording Is the camera currently recording

Default: False

bool is-readout Is camera in readout mode

Default: False

bool buffered TRUE if libuca should buffer frames

Default: False

unsigned int num-buffers Number of frame buffers in the ring buffer

Default: 4
Range: [0, 4294967295]

bool fill-data Fill data with gradient and random image

Default: True

pco

string name Name of the camera

Default:

unsigned int sensor-width Width of the sensor in pixels

Default: 1
Range: [1, 4294967295]

unsigned int sensor-height Height of the sensor in pixels

Default: 1
Range: [1, 4294967295]

double sensor-pixel-width Width of sensor pixel in meters

Default: 1e-05
Range: [2.22507385851e-308, 1.79769313486e+308]

double sensor-pixel-height Height of sensor pixel in meters

Default: 1e-05
Range: [2.22507385851e-308, 1.79769313486e+308]

unsigned int sensor-bitdepth Number of bits per pixel

Default: 1
Range: [1, 32]

unsigned int sensor-horizontal-binning Number of sensor ADCs that are combined to one pixel in horizontal direction

Default: 1
Range: [1, 4294967295]

None sensor-horizontal-binnings Array of possible binnings in horizontal direction

Default: None

unsigned int sensor-vertical-binning Number of sensor ADCs that are combined to one pixel in vertical direction

Default: 1
Range: [1, 4294967295]

None sensor-vertical-binnings Array of possible binnings in vertical direction

Default: None

None trigger-source Trigger source

Default: <enum UCA_CAMERA_TRIGGER_SOURCE_AUTO of type UcaCameraTriggerSource>

None trigger-type Trigger type

Default: <enum UCA_CAMERA_TRIGGER_TYPE_EDGE of type UcaCameraTriggerType>

double exposure-time Exposure time in seconds

Default: 1.0
Range: [0.0, 1.79769313486e+308]

double frames-per-second Frames per second

Default: 1.0
Range: [2.22507385851e-308, 1.79769313486e+308]

unsigned int roi-x0 Horizontal coordinate

Default: 0
Range: [0, 4294967295]

unsigned int roi-y0 Vertical coordinate

Default: 0

Range: [0, 4294967295]

unsigned int roi-width Width of the region of interest

Default: 1
Range: [1, 4294967295]

unsigned int roi-height Height of the region of interest

Default: 1
Range: [1, 4294967295]

unsigned int roi-width-multiplier Minimum possible step size of horizontal ROI

Default: 1
Range: [1, 4294967295]

unsigned int roi-height-multiplier Minimum possible step size of vertical ROI

Default: 1
Range: [1, 4294967295]

bool has-streaming Is the camera able to stream the data

Default: True

bool has-camram-recording Is the camera able to record the data in-camera

Default: False

unsigned int recorded-frames Number of frames recorded into internal camera memory

Default: 0
Range: [0, 4294967295]

bool transfer-asynchronously Specify whether data should be transfered asynchronously using a specified callback

Default: False

bool is-recording Is the camera currently recording

Default: False

bool is-readout Is camera in readout mode

Default: False

bool buffered TRUE if libuca should buffer frames

Default: False

unsigned int num-buffers Number of frame buffers in the ring buffer

Default: 4
Range: [0, 4294967295]

bool sensor-extended Use extended sensor format

Default: False

unsigned int sensor-width-extended Width of the extended sensor in pixels

Default: 1
Range: [1, 4294967295]

unsigned int sensor-height-extended Height of the extended sensor in pixels

Default: 1
Range: [1, 4294967295]

double sensor-temperature Temperature of the sensor in degree Celsius

Default: 0.0
Range: [-1.79769313486e+308, 1.79769313486e+308]

None sensor-pixelrates Array of possible sensor pixel rates

Default: None

unsigned int sensor-pixelrate Pixel rate

Default: 1
Range: [1, 4294967295]

unsigned int sensor-adcs Number of ADCs to use

Default: 1
Range: [1, 2]

unsigned int sensor-max-adcs Maximum number of ADCs that can be set with "sensor-adcs"

Default: 1
Range: [1, 4294967295]

bool has-double-image-mode Is double image mode supported by this model

Default: False

bool double-image-mode Use double image mode

Default: False

bool offset-mode Use offset mode

Default: False

None record-mode Record mode

Default: <enum UCA_PCO_CAMERA_RECORD_MODE_SEQUENCE of type UcaPcoCameraRecordMode>

None storage-mode Storage mode

Default: <enum UCA_PCO_CAMERA_STORAGE_MODE_FIFO_BUFFER of type UcaPcoCameraStorageMode>

None acquire-mode Acquire mode

Default: <enum UCA_PCO_CAMERA_ACQUIRE_MODE_AUTO of type UcaPcoCameraAcquireMode>

bool fast-scan Use fast scan mode with less dynamic range

Default: False

int cooling-point Cooling point of the camera in degree celsius

Default: 5
Range: [0, 10]

int cooling-point-min Minimum cooling point in degree celsius

Default: 0
Range: [-2147483648, 2147483647]

int cooling-point-max Maximum cooling point in degree celsius

Default: 0
Range: [-2147483648, 2147483647]

int cooling-point-default Default cooling point in degree celsius

Default: 0
Range: [-2147483648, 2147483647]

bool noise-filter Noise filter

Default: False

None timestamp-mode Timestamp mode

Default: <enum UCA_PCO_CAMERA_TIMESTAMP_NONE of type UcaPcoCameraTimestamp>

string version Camera version given as 'serial number, hardware major.minor, firmware major.minor'

Default: 0, 0.0, 0.0

bool global-shutter Global shutter enabled

Default: False

file

string name Name of the camera

Default:

unsigned int sensor-width Width of the sensor in pixels

Default: 512
Range: [1, 4294967295]

unsigned int sensor-height Height of the sensor in pixels

Default: 512
Range: [1, 4294967295]

double sensor-pixel-width Width of sensor pixel in meters

Default: 1e-05
Range: [2.22507385851e-308, 1.79769313486e+308]

double sensor-pixel-height Height of sensor pixel in meters

Default: 1e-05
Range: [2.22507385851e-308, 1.79769313486e+308]

unsigned int sensor-bitdepth Number of bits per pixel

Default: 8
Range: [1, 32]

unsigned int sensor-horizontal-binning Number of sensor ADCs that are combined to one pixel in horizontal direction

Default: 1
Range: [1, 4294967295]

unsigned int sensor-vertical-binning Number of sensor ADCs that are combined to one pixel in vertical direction

Default: 1
Range: [1, 4294967295]

None trigger-source Trigger source

Default: <enum UCA_CAMERA_TRIGGER_SOURCE_AUTO of type UcaCameraTriggerSource>

None trigger-type Trigger type

Default: <enum UCA_CAMERA_TRIGGER_TYPE_EDGE of type UcaCameraTriggerType>

double exposure-time Exposure time in seconds

Default: 1.0
Range: [0.0, 1.79769313486e+308]

double frames-per-second Frames per second

Default: 1.0
Range: [2.22507385851e-308, 1.79769313486e+308]

unsigned int roi-x0 Horizontal coordinate

Default: 0
Range: [0, 4294967295]

unsigned int roi-y0 Vertical coordinate

Default: 0
Range: [0, 4294967295]

unsigned int roi-width Width of the region of interest

Default: 1
Range: [1, 4294967295]

unsigned int roi-height Height of the region of interest

Default: 1
Range: [1, 4294967295]

unsigned int roi-width-multiplier Minimum possible step size of horizontal ROI

Default: 1
Range: [1, 4294967295]

unsigned int roi-height-multiplier Minimum possible step size of vertical ROI

Default: 1
Range: [1, 4294967295]

bool has-streaming Is the camera able to stream the data

Default: True

bool has-camram-recording Is the camera able to record the data in-camera

Default: False

unsigned int recorded-frames Number of frames recorded into internal camera memory

Default: 0
Range: [0, 4294967295]

bool transfer-asynchronously Specify whether data should be transfered asynchronously using a specified callback

Default: False

bool is-recording Is the camera currently recording

Default: False

bool is-readout Is camera in readout mode

Default: False

bool buffered TRUE if libuca should buffer frames

Default: False

unsigned int num-buffers Number of frame buffers in the ring buffer

Default: 4
Range: [0, 4294967295]

string path Path to directory containing TIFF files

Default: .

1.3 Application Programming Interface

In the introduction we had a quick glance over the basic API used to communicate with a camera. Now we will go into more detail and explain required background to understand the execution model.

1.3.1 Instantiating cameras

We have already seen how to instantiate a camera object from a name. If you have more than one camera connected to a machine, you will most likely want the user decide which to use. To do so, you can enumerate all camera strings with `uca_plugin_manager_get_available_cameras`:

```
GList *types;

types = uca_plugin_manager_get_available_cameras (manager);

for (GList *it = g_list_first (types); it != NULL; it = g_list_next (it))
    g_print ("%s\n", (gchar *) it->data);

/* free the strings and the list */
```

```
g_list_foreach (types, (GFunc) g_free, NULL);
g_list_free (types);
```

1.3.2 Errors

All public API functions take a location of a pointer to a `GError` structure as a last argument. You can pass in a `NULL` value, in which case you cannot be notified about exceptional behavior. On the other hand, if you pass in a pointer to a `GError`, it must be initialized with `NULL` so that you do not accidentally overwrite and miss an error occurred earlier.

Read more about `GErrors` in the official GLib documentation.

1.3.3 Recording

Recording frames is independent of actually grabbing them and is started with `uca_camera_start_recording`. You should always stop the recording with `ufo_camera_stop_recording` when you finished. When the recording has started, you can grab frames synchronously as described earlier. In this mode, a block to `uca_camera_grab` blocks until a frame is read from the camera. Grabbing might block indefinitely, when the camera is not functioning correctly or it is not triggered automatically.

1.3.4 Triggering

`libuca` supports three trigger sources through the "trigger-source" property:

1. `UCA_CAMERA_TRIGGER_SOURCE_AUTO`: Exposure is triggered by the camera itself.
2. `UCA_CAMERA_TRIGGER_SOURCE_SOFTWARE`: Exposure is triggered via software.
3. `UCA_CAMERA_TRIGGER_SOURCE_EXTERNAL`: Exposure is triggered by an external hardware mechanism.

With `UCA_CAMERA_TRIGGER_SOURCE_SOFTWARE` you have to trigger with `uca_camera_trigger`:

```
/* thread A */
g_object_set (G_OBJECT (camera),
              "trigger-source", UCA_CAMERA_TRIGGER_SOURCE_SOFTWARE,
              NULL);

uca_camera_start_recording (camera, NULL);
uca_camera_grab (camera, buffer, NULL);
uca_camera_stop_recording (camera, NULL);

/* thread B */
uca_camera_trigger (camera, NULL);
```

Moreover, the "trigger-type" property specifies if the exposure should be triggered at the rising edge or during the level signal.

1.3.5 Grabbing frames asynchronously

In some applications, it might make sense to setup asynchronous frame acquisition, for which you will not be blocked by a call to `libuca`:

```
static void
callback (gpointer buffer, gpointer user_data)
{
    /*
     * Do something useful with the buffer and the string we have got.
     */
}

static void
setup_async (UcaCamera *camera)
{
    gchar *s = g_strdup ("lorem ipsum");

    g_object_set (G_OBJECT (camera),
                  "transfer-asynchronously", TRUE,
                  NULL);

    uca_camera_set_grab_func (camera, callback, s);
    uca_camera_start_recording (camera, NULL);

    /*
     * We will return here and `callback` will be called for each newo
     * new frame.
     */
}
```

1.3.6 Bindings

Since version 1.1, libuca generates GObject introspection meta data if `g-ir-scanner` and `g-ir-compiler` can be found. When the XML description `Uca-x.y.gir` and the typelib `Uca-x.y.typelib` are installed, GI-aware languages can access libuca and create and modify cameras, for example in Python:

```
from gi.repository import Uca

pm = Uca.PluginManager()

# List all cameras
print(pm.get_available_cameras())

# Load a camera
cam = pm.get_camerav('pco', [])

# You can read and write properties in two ways
cam.set_properties(exposure_time=0.05)
cam.props.roi_width = 1024
```

Note, that the naming of classes and properties depends on the GI implementation of the target language. For example with Python, the namespace prefix `uca_` becomes the module name `Uca` and dashes separating property names become underscores.

Integration with Numpy is relatively straightforward. The most important thing is to get the data pointer from a Numpy array to pass it to `uca_camera_grab`:

```
import numpy as np

def create_array_from(camera):
```

```
    """Create a suitably sized Numpy array and return it together with the
    arrays data pointer"""
    bits = camera.props.sensor_bitdepth
    dtype = np.uint16 if bits > 8 else np.uint8
    a = np.zeros((cam.props.roi_height, cam.props.roi_width), dtype=dtype)
    return a, a.__array_interface__['data'][0]

# Suppose 'camera' is a already available, you would get the camera data like
# this:
a, buf = create_array_from(camera)
camera.start_recording()
camera.grab(buf)

# Now data is in 'a' and we can use Numpy functions on it
print(np.mean(a))

camera.stop_recording()
```

1.3.7 Integrating new cameras

A new camera is integrated by sub-classing `UcaCamera` and implement all virtual methods. The simplest way is to take the `mock` camera and rename all occurences. Note, that if you class is going to be called `FooBar`, the upper case variant is `FOO_BAR` and the lower case variant is `foo_bar`.

In order to fully implement a camera, you need to override at least the following virtual methods:

- `start_recording`: Take suitable actions so that a subsequent call to `grab` delivers an image or blocks until one is exposed.
- `stop_recording`: Stop recording so that subsequent calls to `grab` fail.
- `grab`: Return an image from the camera or block until one is ready.

1.3.8 Asynchronous operation

When the camera supports asynchronous acquisition and announces it with a true boolean value for `"transfer-asynchronously"`, a mechanism must be setup up during `start_recording` so that for each new frame the grab func callback is called.

1.3.9 Cameras with internal memory

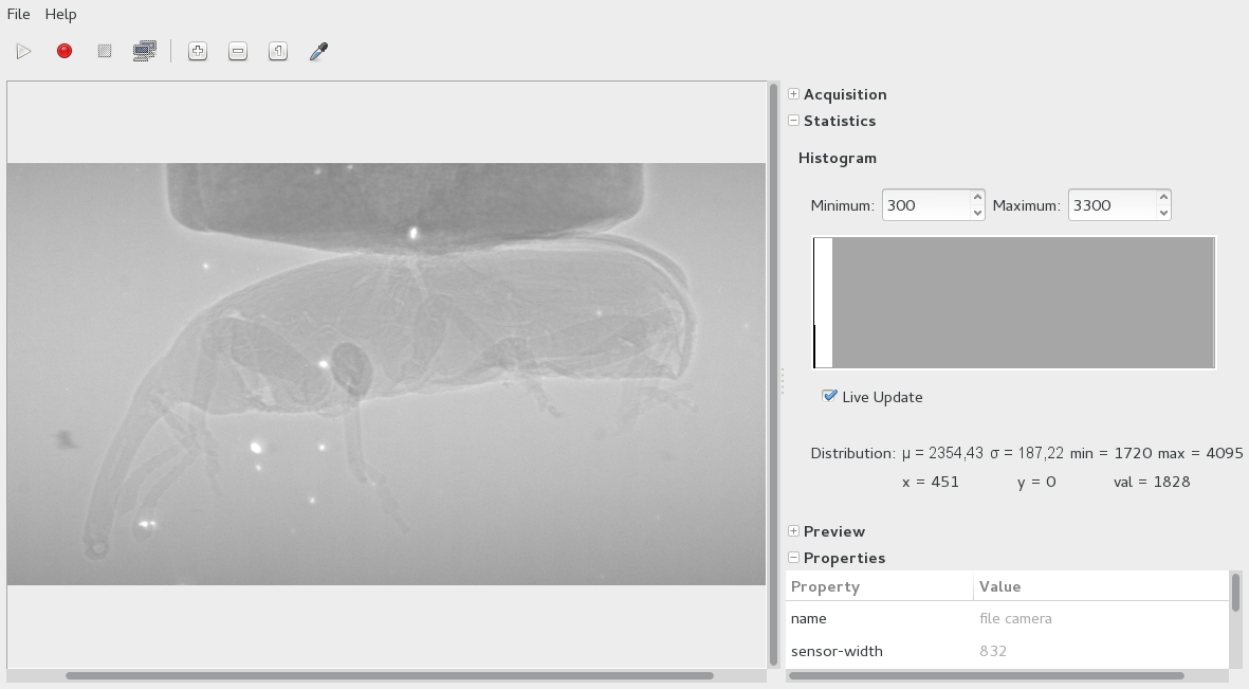
Cameras such as the pco.dimax record into their own on-board memory rather than streaming directly to the host PC. In this case, both `start_recording` and `stop_recording` initiate and end acquisition to the on-board memory. To initiate a data transfer, the host calls `start_readout` which must be suitably implemented. The actual data transfer happens either with `grab` or asynchronously.

1.4 Tools

Several tools are available to ensure `libuca` works as expected. All of them are installed with `make install`.

1.4.1 uca-camera-control – simple graphical user interface

Records and shows frames. Moreover, you can change the camera properties in a side pane:



You can see all available options of `uca-camera-control` with:

```
$ uca-camera-control --help-all
```

1.4.2 uca-grab – grabbing frames

Grab frames with

```
$ uca-grab --num-frames=10 camera-model
```

store them on disk as `frames.tif` if `libtiff` is installed, otherwise as `frame-00000.raw`, `frame-000001.raw`. The raw format is a memory dump of the frames, so you might want to use ImageJ to view them. You can also specify the output filename or filename prefix with the `-o/--output` option:

```
$ uca-grab -n 10 --output=foobar.tif camera-model
```

Instead of reading exactly *n* frames, you can also specify a duration in fractions of seconds:

```
$ uca-grab --duration=0.25 camera-model
```

You can see all available options of `uca-grab` with:

```
$ uca-grab --help-all
```

1.4.3 uca-benchmark – check bandwidth

Measure the memory bandwidth by taking subsequent frames and averaging the grabbing time:

```
$ uca-benchmark option camera-model
```

You can specify the number of frames per run with the `-n/--num-frames` option, the number of runs with the `-r/--num-runs` option and test asynchronous mode with the `async` option:

```
$ uca-benchmark -n 100 -r 3 --async mock

# Type    Trigger Source   FPS        Bandwidth   Frames acquired/total
  sync    auto             17.57 Hz   4.39 MB/s   300/300 acquired (0.00% dropped)
  async   auto             19.98 Hz   4.99 MB/s   300/300 acquired (0.00% dropped)

# --- General information ---
# Camera: mock
# Sensor size: 4096x4096
# ROI size: 512x512
# Exposure time: 0.050000s
```

You can see all available options of `uca-benchmark` with:

```
$ uca-benchmark --help-all
```

1.4.4 uca-info – get properties information

Get information about camera properties with:

```
$ uca-info camera-model
```

For example:

```
$ uca-info mock
# RO | name                      | "mock camera"
# RO | sensor-width              | 4096
# RO | sensor-height             | 4096
# RO | sensor-pixel-width        | 0.000010
# RO | sensor-pixel-height       | 0.000010
# RO | sensor-bitdepth           | 8
...
```

1.4.5 uca-gen-doc – generate properties documentation

Generate HTML source code of property documentation of a camera with:

```
$ uca-gen-doc camera-model
```

1.5 Concert

Concert is a light-weight control system interface, which can also control `libuca` cameras.

1.5.1 Installation

In the official documentation you can read how to install `Concert`.

1.5.2 Usage

`Concert` can be used from a session and within an integrated `IPython` shell or as a library.

In order to create a concert session you should first initialize the session and then start the editor:

```
$ concert init session
$ concert edit session
```

You can simply add your camera, for example the `mock` camera with:

```
from concert.devices.cameras.uca import Camera

camera = Camera("mock")
```

and start the session with:

```
$ concert start session
```

The function `ddoc()` will give you an overview of all defined devices in the session:

```
session > ddoc()
# ------------------------------------------------------------------------------------
↪------------
#  Name     Description              Parameters
# ------------------------------------------------------------------------------------
↪------------
#  camera   libuca-based camera.     Name            Unit     Description
#                                    buffered        None     info    TRUE if libuca␣
↪should buffer
#           All properties that are                           frames
#           exported by the                                   locked  no
#           underlying camera are    exposure_time   second   info    Exposure time␣
↪in seconds
#           also visible.                                     locked  no
#                                                             lower   -inf second
#                                                             upper   inf second
...
```

Getting and setting camera parameters

You can get an overview of the camera parameters by calling the `dstate()` function:

```
session > dstate()
# ------------------------------------------------------
#  Name     Parameters
# ------------------------------------------------------
#  camera    buffered                 False
#            exposure_time            0.05 second
#            fill_data                True
#            frame_rate               20.0 1 / second
#            has_camram_recording     False
#            has_streaming            True
...
```

set the value of a parameter with:

```
session > camera.exposure_time = 0.01 * q.s
```

and check the set value with:

```
session > camera.exposure_time
# <Quantity(0.01, 'second')>
```

or you can use the `get()` and `set()` methods:

```
session > exposure_time = camera["exposure_time"]
session > exposure_time.set(0.1 * q.s)
session > exposure_time.get().result()
# <Quantity(0.1, 'second')>
```

In order to set the trigger source property you can use `trigger_sources.AUTO`, `trigger_sources.SOFTWARE` or `trigger_sources.EXTERNAL`:

```
session > camera.trigger_source = camera.trigger_sources.AUTO
```

Grabbing frames

To grab a frame, first start the camera, use the `grab()` function and stop the camera afterwards:

```
session > camera.start_recording()
session > frame = camera.grab()
session > camera.stop_recording()
```

You get the frame as an array:

```
session > frame
# array([[  0,   0,   0, ...,   0,   0,   0],
#        [  0,   0,   0, ...,   0,   0,   0],
#        [  0,   0, 255, ...,   0,   0,   0],
#        ...,
#        [  0,   0,   0, ...,   0,   0,   0],
#        [  0,   0,   0, ...,   0,   0,   0],
#        [  0,   0,   0, ...,   0,   0,   0]], dtype=uint8)
```

Saving state and locking parameters

You can store the current state of your camera with:

```
session > camera.stash()
# <Future at 0x2b8ab10 state=running>
```

And go back to it again with:

```
session > camera.restore()
# <Future at 0x299f550 state=running>
```

In case you want to prevent a parameter or all the parameters from being written you can use the `lock()` method:

```
session > camera["exposure_time"].lock()
session > camera["exposure_time"].set(1 * q.s)
```

```
# <Future at 0x2bb3d90 state=finished raised LockError>

# lock all parameters of the camera device
session > camera.lock()
```

and to unlock them again, just use the `unlock()` method:

```
session > camera.unlock()
```

Concert as a library - more examples

You can also use `Concert` as a library.

For example test the bit depth consistency with:

```
import numpy as np
from concert.quantities import q
from concert.devices.cameras.uca import Camera


def acquire_frame(camera):
    camera.start_recording()
    frame = camera.grab()
    camera.stop_recording()
    return frame

def test_bit_depth_consistency(camera):
    camera.exposure_time = 1 * q.s
    frame = acquire_frame(camera)

    bits = camera.sensor_bitdepth
    success = np.mean(frame) < 2**bits.magnitude
    print "success" if success else "higher values than possible"

camera = Camera("mock")
test_bit_depth_consistency(camera)
```

or the exposure time consistency with:

```
def test_exposure_time_consistency(camera):
    camera.exposure_time = 1 * q.ms
    first = acquire_frame(camera)

    camera.exposure_time = 100 * q.ms
    second = acquire_frame(camera)

    success = np.mean(first) < np.mean(second)
    print "success" if success else "mean image value is lower than expected"
```

Official Documentation

If you have more questions or just want to know more about `Concert`, please take a look at the very detailed official documentation.

1.6 Remote access

A Remote access is available for `libuca` cameras:

1.6.1 TCP-based network bridge camera

uca-net is a transparent TCP-based network bridge camera for remote access of `libuca` cameras.

Installation

The only dependency is libuca itself and any camera you wish to access.

Clone the repository:

```
$ git clone https://github.com/ufo-kit/uca-net.git
```

and create a new build directory inside:

```
$ cd uca-net/
$ mkdir build
```

The installation process is the same as by `libuca`:

```
$ cd build/
$ cmake ..
$ make
$ sudo make install
```

Usage

You can start a server on a remote machine with:

```
$ ucad camera-model
```

and connect to it from any other machine, for example:

```
$ UCA_NET_HOST=foo.bar.com:4567 uca-grab -n 10 net  # grab ten frames
$ uca-camera-control -c net     # control graphically
```

1.6.2 GObject Tango device

UcaDevice is a generic Tango Device that wraps `libuca` in order to make libuca controlled cameras available as Tango devices.

Note: The documentation of UcaDevice can be outdated.

Architecture

UcaDevice is derived from GObjectDevice and adds libuca specific features like start/stop recording etc. The most important feature is *acquisition control*. UcaDevice is responsible for controlling acquisition of images from libuca. The last aquired image can be accessed by reading attribute `SingleImage`. UcaDevice is most useful together with ImageDevice. If used together, UcaDevice sends each aquired image to ImageDevice, which in turn does configured post-processing like flipping, rotating or writing the image to disk.

Attributes

Besides the dynamic attributes translated from libuca properties UcaDevice has the following attributes:

- `NumberOfImages (Tango::DevLong)`: how many images should be acquired? A value of -1 means unlimited *(read/write)*
- `ImageCounter (Tango::DevULong)`: current number of acquired images *(read-only)*
- `CameraName (Tango::DevString)`: name of libuca object type *(read-only)*
- `SingleImage (Tango::DevUChar)`: holds the last acquired image

Acquisition Control

In UcaDevice acquisition control is mostly implemented by two `yat4tango::DeviceTasks`: *AcquisitionTask* and *GrabTask*. *GrabTask*'s only responsibility is to call `grab` on `libuca` synchronously and post the data on to AcquisitionTask.

AcquisitionTask is responsible for starting and stopping GrabTask and for passing image data on to `ImageDevice` (if exisiting) and to `UcaDevice` for storage in attribute `SingleImage`. It counts how many images have been acquired and checks this number against the configured `NumberOfImages`. If the desired number is reached, it stops GrabTask, calls `stop_recording` on `libuca` and sets the Tango state to STANDBY.

Plugins

As different cameras have different needs, plugins are used for special implementations. Plugins also makes UcaDevice and Tango Servers based on it more flexible and independent of libuca implementation.

- PCO: The Pco plugin implements additional checks when writing ROI values.
- Pylon: The Pylon plugin sets default values for `roi-width` and `roi-height` from libuca properties `roi-width-default` and `roi-height-default`.

Installation

Detailed installation depends on the manifestation of UcaDevice. All manifestations depend on the following libraries:

- YAT
- YAT4Tango
- Tango
- GObjectDevice
- ImageDevice

Build

```
export PKG_CONFIG_PATH=/usr/lib/pkgconfig
export PYLON_ROOT=/usr/pylon
export LD_LIBRARY_PATH=$PYLON_ROOT/lib64:$PYLON_ROOT/genicam/bin/Linux64_x64
git clone git@iss-repo:UcaDevice.git
cd UcaDevice
mkdir build
cd build
cmake ..
make
```

Setup in Tango Database / Jive

Before `ds_UcaDevice` can be started, it has to be registered manually in the Tango database. With `Jive` the following steps are necessary:

1. Register Server Menu *Tools* → Server Wizard Server name → ds_UcaDevice Instance name → my_server *(name can be chosen freely)* Next Cancel
2. Register Classes and Instances In tab *Server*: context menu on ds_UcaDevice → my_server → Add Class Class: UcaDevice Devices: `iss/name1/name2` Register server same for class ImageDevice
3. Start server

   ```
   export TANGO_HOST=anka-tango:100xx
   export UCA_DEVICE_PLUGINS_DIR=/usr/lib(64)
   ds_UcaDevice pco my_server
   ```

4. Setup properties for UcaDevice context menu on device → Device wizard Property StorageDevice: *address of previously registered ImageDevice instance*
5. Setup properties for ImageDevice context menu on device → Device wizard PixelSize: how many bytes per pixel for the images of this camera? GrabbingDevice: *address of previously registered UcaDevice instance*
6. Finish restart ds_UcaDevice

FAQ

UcaDevice refuses to start up. . . ? Most likely there is no instance registered for class UcaDevice. Register an instance for this class and it should work.

Why does UcaDevice depend on ImageDevice? UcaDevice pushes each new Frame to ImageDevice. Polling is not only less efficient but also prone to errors, e.g. missed/double frames and so on. Perhaps we could use the Tango-Event-System here!

Open Questions, Missing Features etc.

- *Why do we need to specify* `Storage` *for UcaDevice and* `GrabbingDevice` *for ImageDevice?*

  ImageDevice needs the Tango-Address of UcaDevice to mirror all Attributes and Commands and to forward them to it. UcaDevice needs the Tango-Address of ImageDevice to push a new frame on reception. A more convenient solution could be using conventions for the device names, e.g. of the form `$prefix/$instance_name/uca` and `$prefix/$instance_name/image`. That way we could get rid of the

two Device-Properties and an easier installation without the process of registering the classes and instances in `Jive`.

- *Why does UcaDevice dynamically link to GObjectDevice?*

  There is no good reason for it. Packaging and installing would be easier if we linked statically to `GObjectDevice` because we would hide this dependency. Having a separate `GObjectDevice` is generally a nice feature to make `GObjects` available in Tango. However, there is currently no GObjectDevice in use other than in the context of UcaDevice.

- *Why must the plugin name be given as a command line parameter instead of a Device-Property?*

  There is no good reason for it. UcaDevice would be easier to use, if the plugin was configured in the Tango database as a Device-Property for the respective server instance.

1.6.3 Python Tango server

`libuca/tango` is a Python-based Tango server.

Installation

In order to install `libuca/tango` you need

- PyTango and
- tifffile

Go to the `libuca` directory and install the server script with:

```
$ cd tango
$ sudo python setup.py install
```

and create a new TANGO server `Uca/xyz` with a class named `Camera`.

Usage

Before starting the server, you have to create a new device property `camera` which specifies which camera to use. If not set, the `mock` camera will be used by default.

Start the device server with:

```
$ Uca device-property
```

You should be able to manipulate camera attributes like `exposure_time` and to store frames using a `Start`, `Store`, `Stop` cycle:

```
import PyTango

camera = PyTango.DeviceProxy("foo/Camera/mock")
camera.exposure_time = 0.1
camera.Start()
camera.Store('foo.tif')
camera.Stop()
```

The HZG Tango server can also be used with `libuca` cameras.