# “itom”
# Plugin Programming

Marc Gronle
Institut für Technische Optik
Universität Stuttgart
Germany

What are plugins?

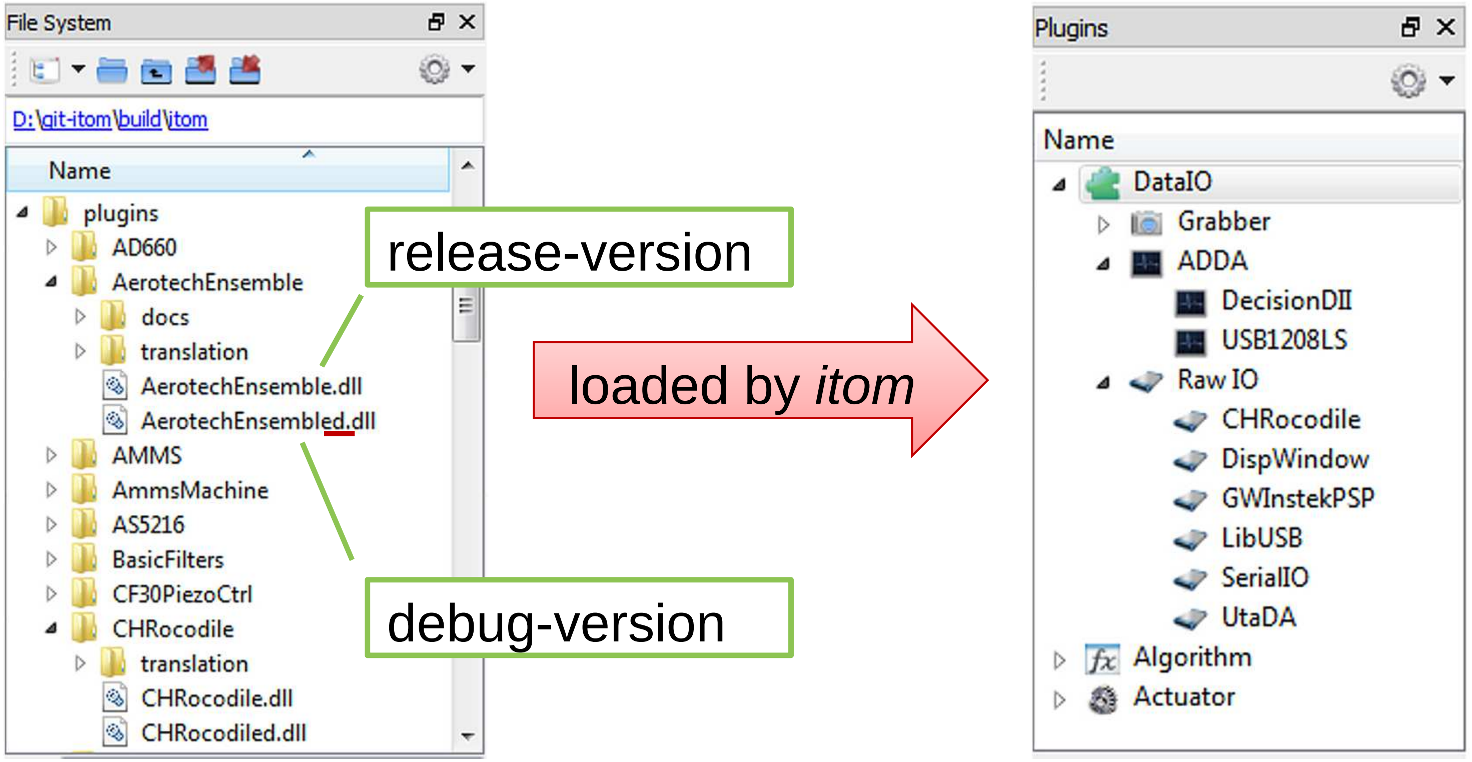
Plugin architecture

Important classes and structures

Working principle of plugins

# Plugin System

- Plugins extend the basic functionalities of itom
- Every plugin is a library (*.dll, *.so) in the subfolder *plugins* of the itom path (*build* directory)



File System

D:\git-itom\build\itom

Name

- plugins
  - AD660
  - AerotechEnsemble
    - docs
    - translation
    - AerotechEnsemble.dll
    - AerotechEnsembled.dll
  - AMMS
  - AmmsMachine
  - AS5216
  - BasicFilters
  - CF30PiezoCtrl
  - CHRocodile
    - translation
    - CHRocodile.dll
    - CHRocodiled.dll

release-version

debug-version

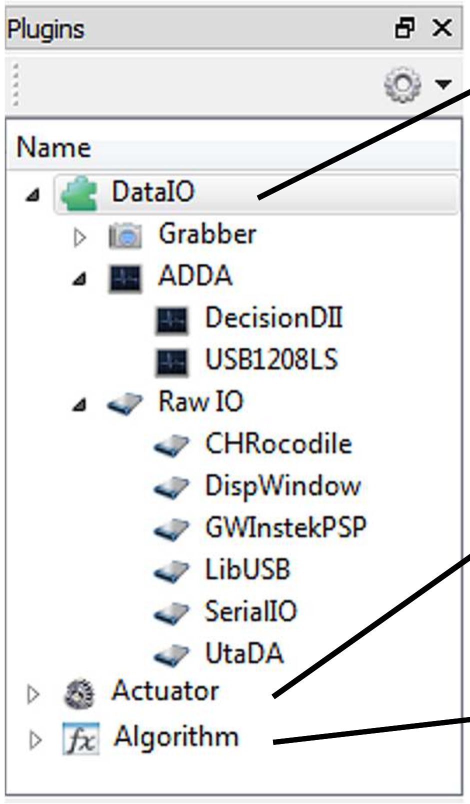
loaded by *itom*

Plugins

Name

- DataIO
  - Grabber
  - ADDA
    - DecisionDII
    - USB1208LS
  - Raw IO
    - CHRocodile
    - DispWindow
    - GWInstekPSP
    - LibUSB
    - SerialIO
    - UtaDA
- Algorithm
- Actuator

# Different types of plugins



Plugins

Name

- DataIO
  - Grabber
  - ADDA
    - DecisionDII
    - USB1208LS
  - Raw IO
    - CHRocodile
    - DispWindow
    - GWInstekPSP
    - LibUSB
    - SerialIO
    - UtaDA
- Actuator
- Algorithm

Hardware **dataIO**

**Grabber**
Cameras

**ADDA**
A/D Converters

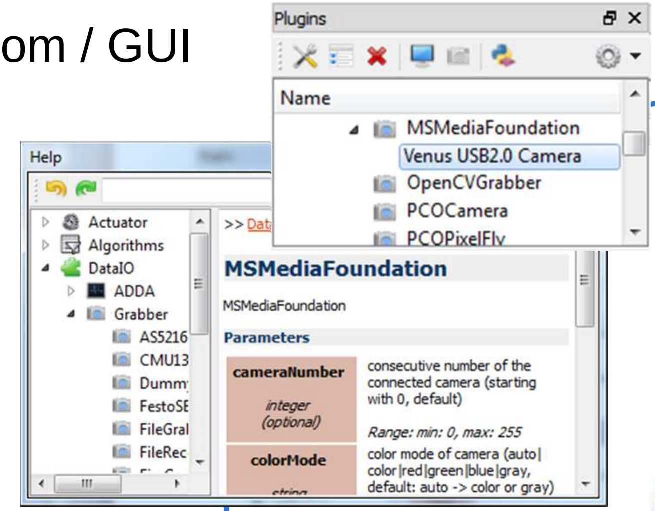
**Raw IO**
further I/O devices

Hardware **actuator**
piezo actuators, stages, …

Software **algorithm**
algorithms, filters
complex GUIs and widgets

# Communication to hardware plugins

itom / GUI



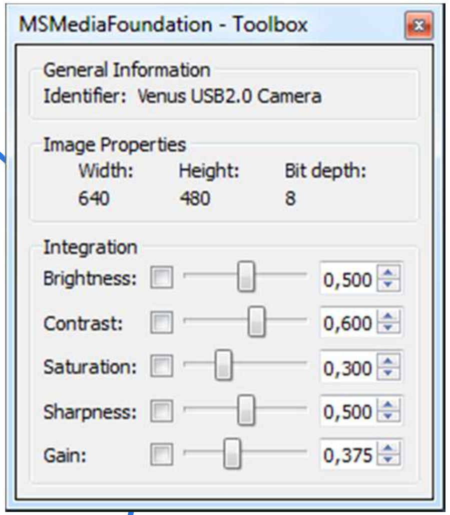
**MSMediaFoundation**

MSMediaFoundation

**Parameters**

| | |
|---|---|
| **cameraNumber**<br>*integer (optional)* | consecutive number of the connected camera (starting with 0, default)<br>*Range: min: 0, max: 255* |
| **colorMode**<br>*string* | color mode of camera (auto\|color\|red\|green\|blue\|gray, default: auto -> color or gray) |

plugin's dock widget (optional)



MSMediaFoundation - Toolbox

General Information

Identifier: Venus USB2.0 Camera

Image Properties

| Width: | Height: | Bit depth: |
|---|---|---|
| 640 | 480 | 8 |

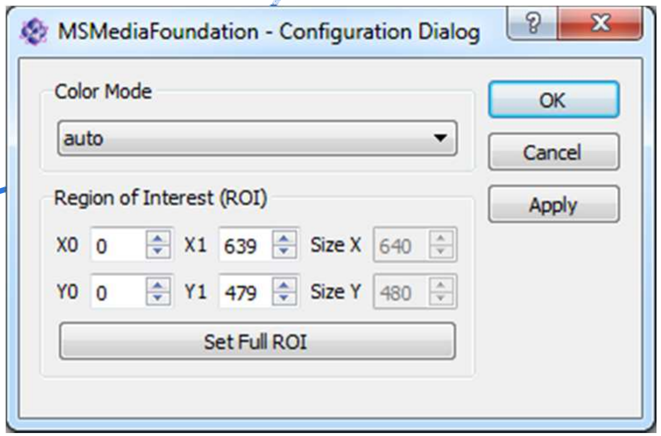
Integration

| | | |
|---|---|---|
| Brightness: | □ | 0,500 |
| Contrast: | □ | 0,600 |
| Saturation: | □ | 0,300 |
| Sharpness: | □ | 0,500 |
| Gain: | □ | 0,375 |

itom / Python

```
#open device
cam = dataIO("MSMediaFoundation")
cam.startDevice()
d = dataObject()
#acquire images
for i in range(0,5):
    cam.acquire()
    cam.getVal(d)
    plot(d)
cam.stopDevice()
```

plugin's configuration dialog (optional)



MSMediaFoundation - Configuration Dialog

Color Mode

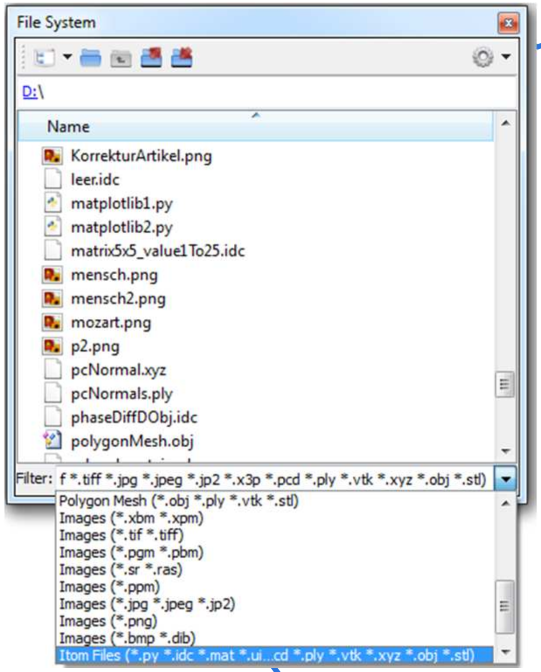
auto

Region of Interest (ROI)

| | | | | | |
|---|---|---|---|---|---|
| X0 | 0 | X1 | 639 | Size X | 640 |
| Y0 | 0 | Y1 | 479 | Size Y | 480 |

# Communication to algorithm plugins

itom / GUI



File System

D:\

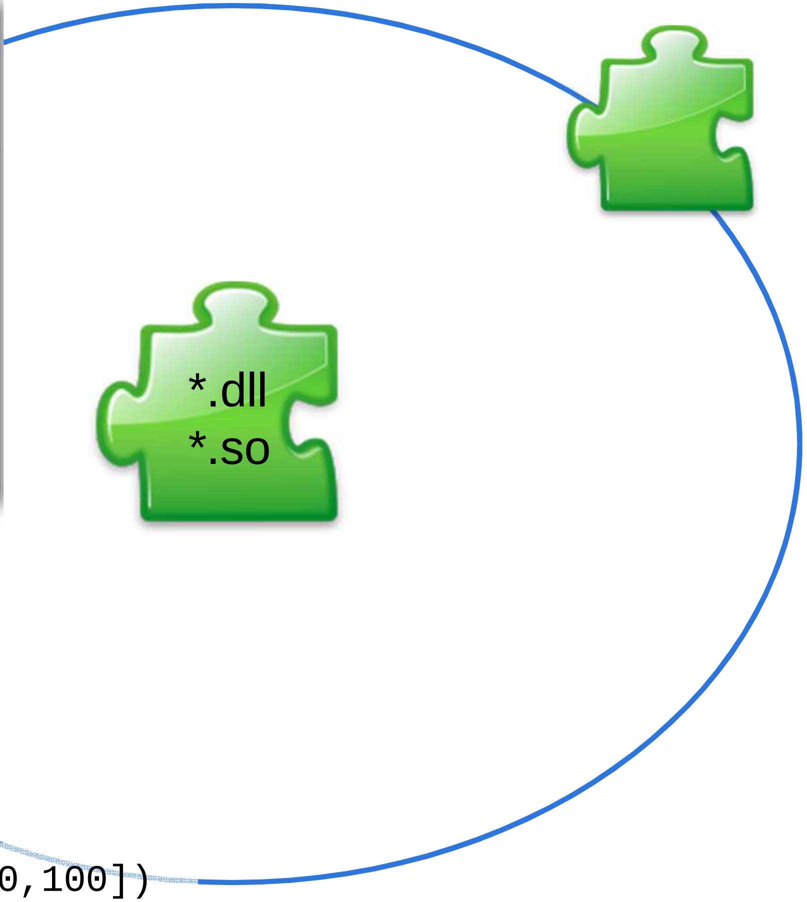
Name

- KorrekturArtikel.png
- leer.idc
- matplotlib1.py
- matplotlib2.py
- matrix5x5_value1To25.idc
- mensch.png
- mensch2.png
- mozart.png
- p2.png
- pcNormal.xyz
- pcNormals.ply
- phaseDiffDObj.idc
- polygonMesh.obj

Filter: f *.tiff *.jpg *.jpeg *.jp2 *.x3p *.pcd *.ply *.vtk *.xyz *.obj *.stl)

- Polygon Mesh (*.obj *.ply *.vtk *.stl)
- Images (*.xbm *.xpm)
- Images (*.tif *.tiff)
- Images (*.pgm *.pbm)
- Images (*.sr *.ras)
- Images (*.ppm)
- Images (*.jpg *.jpeg *.jp2)
- Images (*.png)
- Images (*.bmp *.dib)
- Itom Files (*.py *.idc *.mat *.ui...cd *.ply *.vtk *.xyz *.obj *.stl)

from other plugins via API



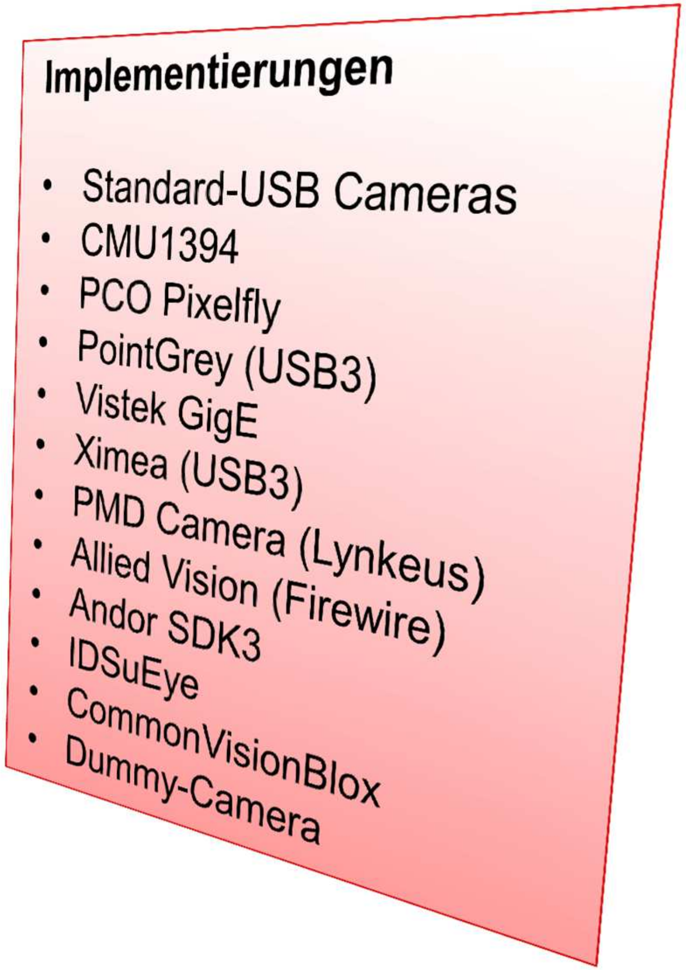
itom / Python

```
d = dataObject.randN([100,100])
r = filter("minValue",d)
```

# Interface ‚dataIO + Grabber‘

## Primary functionality

- getParam(..) → read a parameter
- setParam(..) → set a parameter
- startDevice() → start camera
- stopDevice() → stop camera
- acquire() → take a picture
- getVal(..) / copyVal(..) → load image from camera into itom/Python
- …



**Implementierungen**

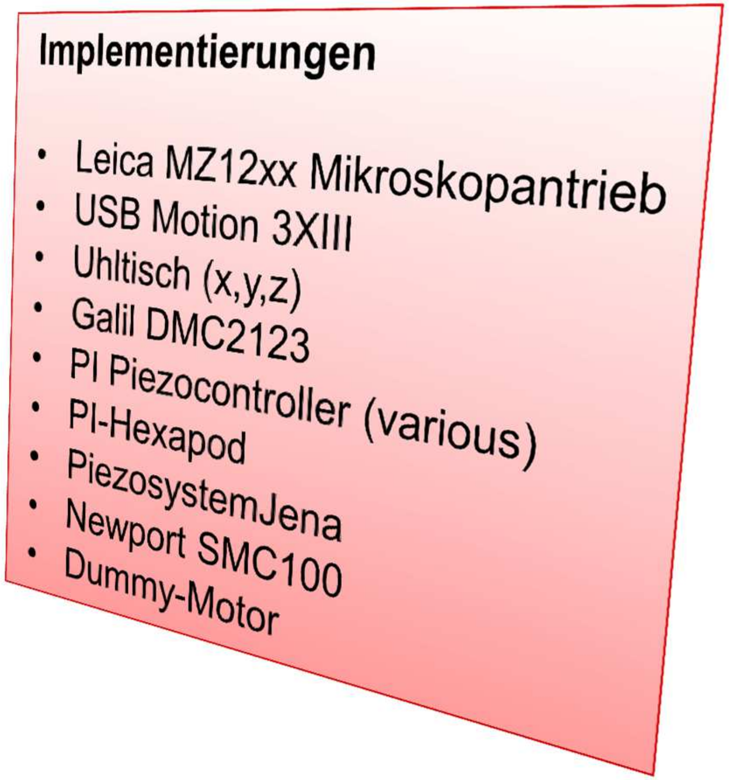
- Standard-USB Cameras
- CMU1394
- PCO Pixelfly
- PointGrey (USB3)
- Vistek GigE
- Ximea (USB3)
- PMD Camera (Lynkeus)
- Allied Vision (Firewire)
- Andor SDK3
- IDSuEye
- CommonVisionBlox
- Dummy-Camera

Live images from the camera can be displayed in separate windows or integrated into custom GUIs

# Interface ‚actuator‘

**Primary Functionality**

- getParam(..) → read Parameter
- setParam(..) → set Parameter
- getStatus(..) → get status per axis
- getPos(..) → read current position
- setPosAbs/Rel() → move to position
- …



**Implementierungen**

- Leica MZ12xx Mikroskopantrieb
- USB Motion 3XIII
- Uhltisch (x,y,z)
- Galil DMC2123
- PI Piezocontroller (various)
- PI-Hexapod
- PiezosystemJena
- Newport SMC100
- Dummy-Motor

Signals about position and status of the actuator can be linked to and processed by the GUI.

# Interface ‚algo‘

‚Algo‘ Plugins define

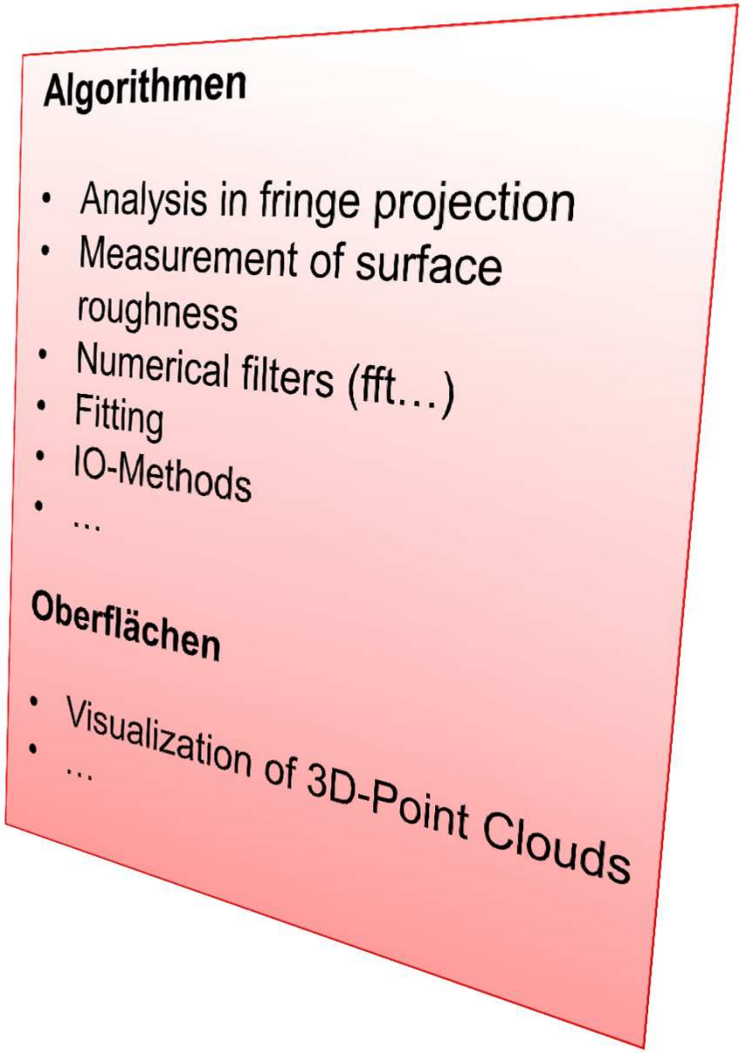
- Numerical algorithms
- GUI elements

Call:

- From a Python script
- By other Plugins

Each method is defined by :

- Mandatory parameters (Type, description…)
- Optional parameters
- Return values



**Algorithmen**

- Analysis in fringe projection
- Measurement of surface roughness
- Numerical filters (fft…)
- Fitting
- IO-Methods
- …

**Oberflächen**

- Visualization of 3D-Point Clouds
- …

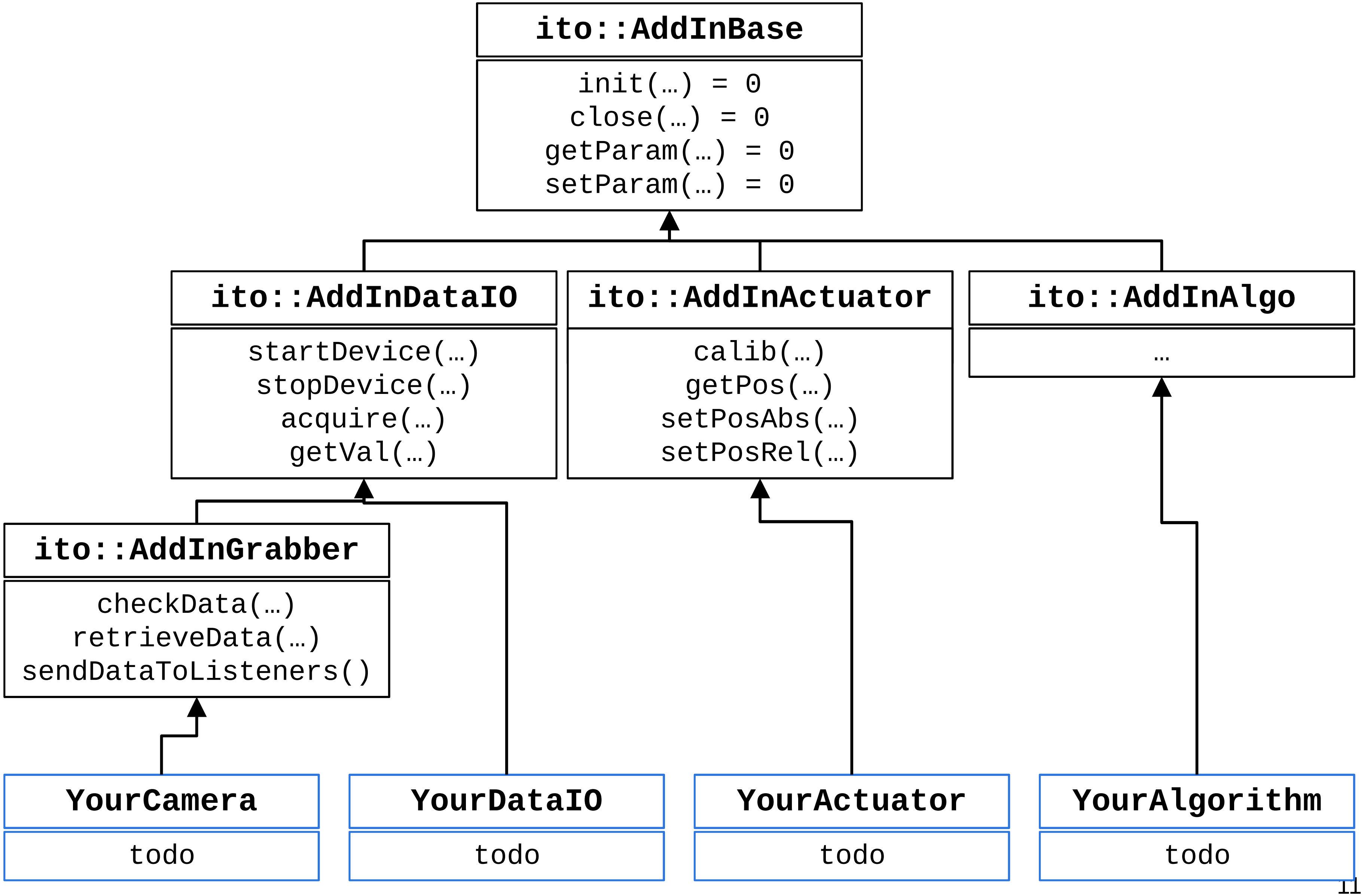
What are plugins?

Plugin architecture

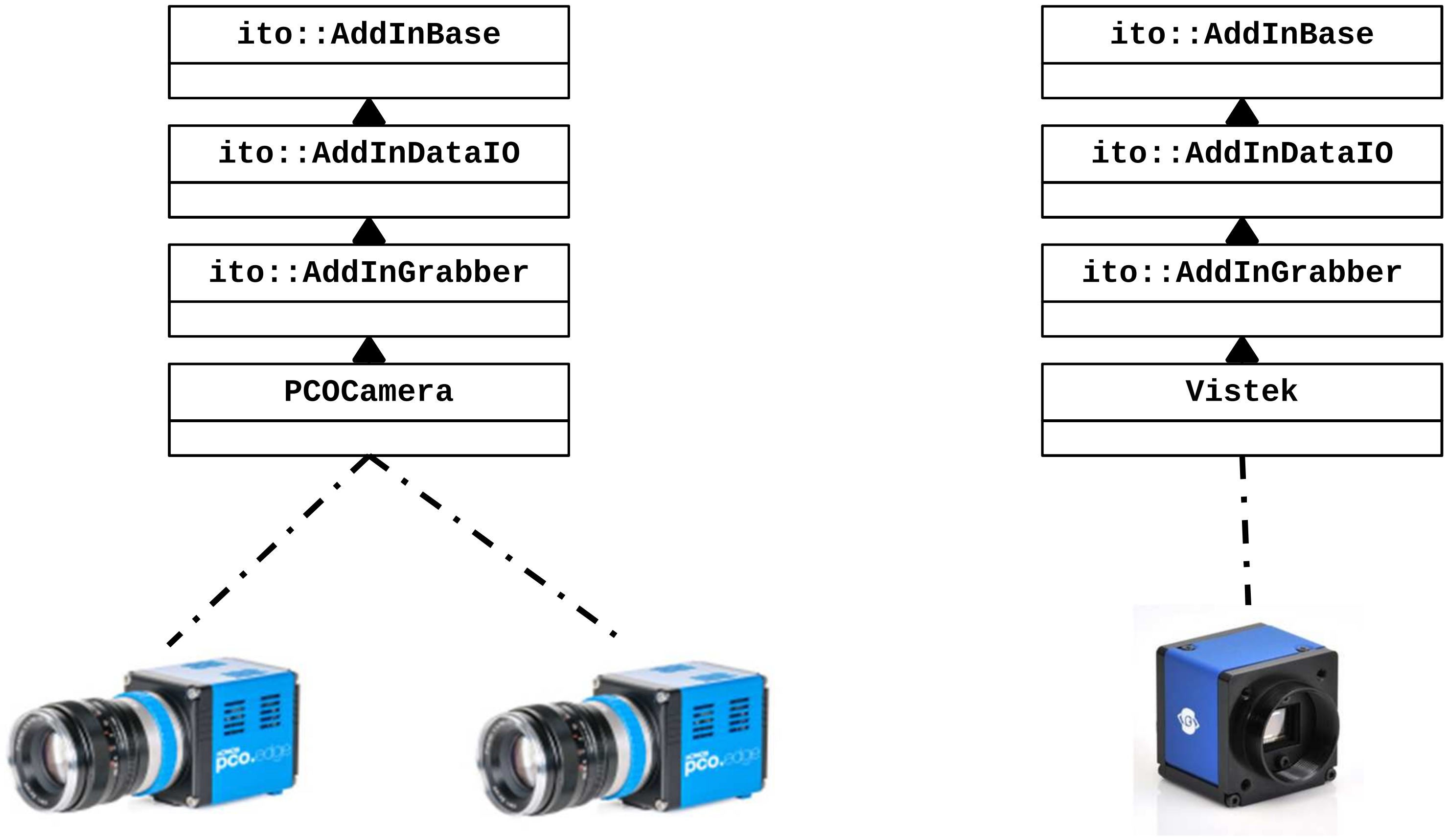
Important classes and structures

Working principle of plugins

# Plugin Architecture – Plugin Class



# Example: Cameras



ito::AddInBase

ito::AddInDataIO

ito::AddInGrabber

PCOCamera

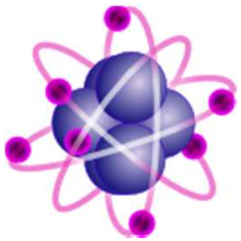
ito::AddInBase

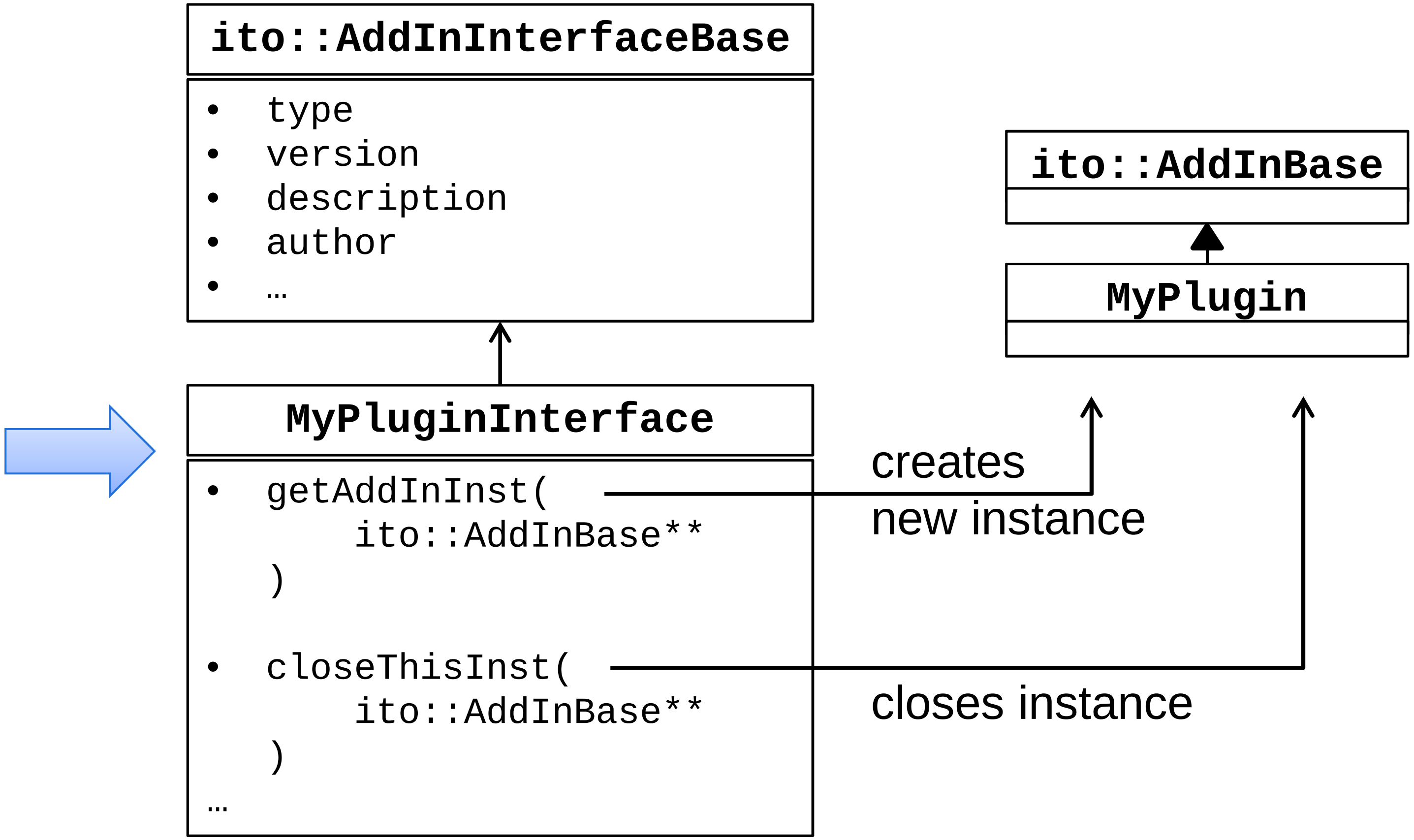
ito::AddInDataIO

ito::AddInGrabber

Vistek

# Plugin Architecture – Plugin Interface Class





**ito::AddInInterfaceBase**

- type
- version
- description
- author
- …

**MyPluginInterface**

- getAddInInst(
      ito::AddInBase**
  )
- closeThisInst(
      ito::AddInBase**
  )

…

**ito::AddInBase**

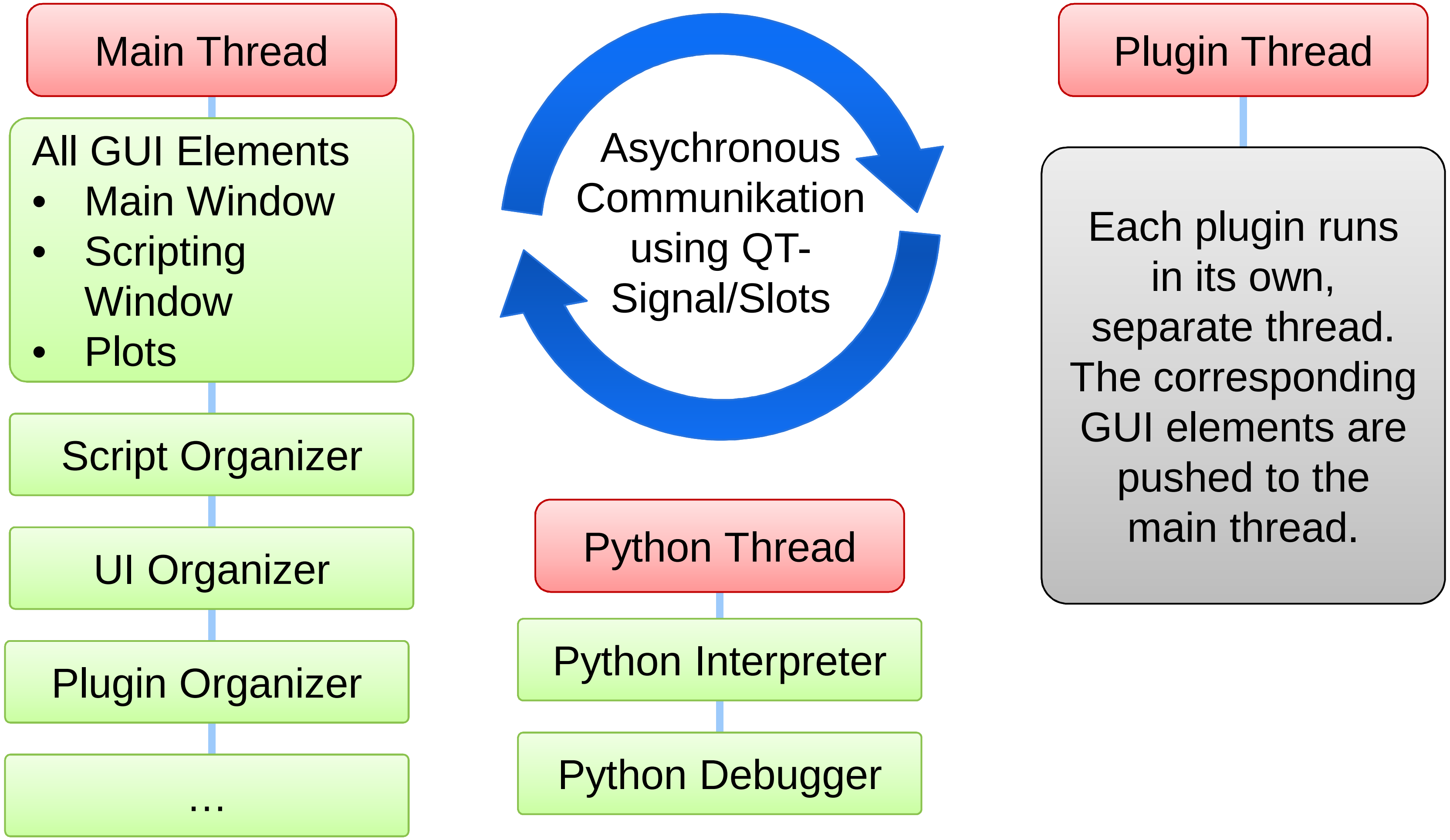
**MyPlugin**

creates
new instance

closes instance

- itom loads all *MyPluginInterfaces* at startup (singleton)
- request of new hardware instance is executed via corresponding *MyPluginInterface* class.

# Multithreading



Main Thread

All GUI Elements

- Main Window
- Scripting Window
- Plots

Script Organizer

UI Organizer

Plugin Organizer

…

Asychronous Communikation using QT-Signal/Slots

Python Thread

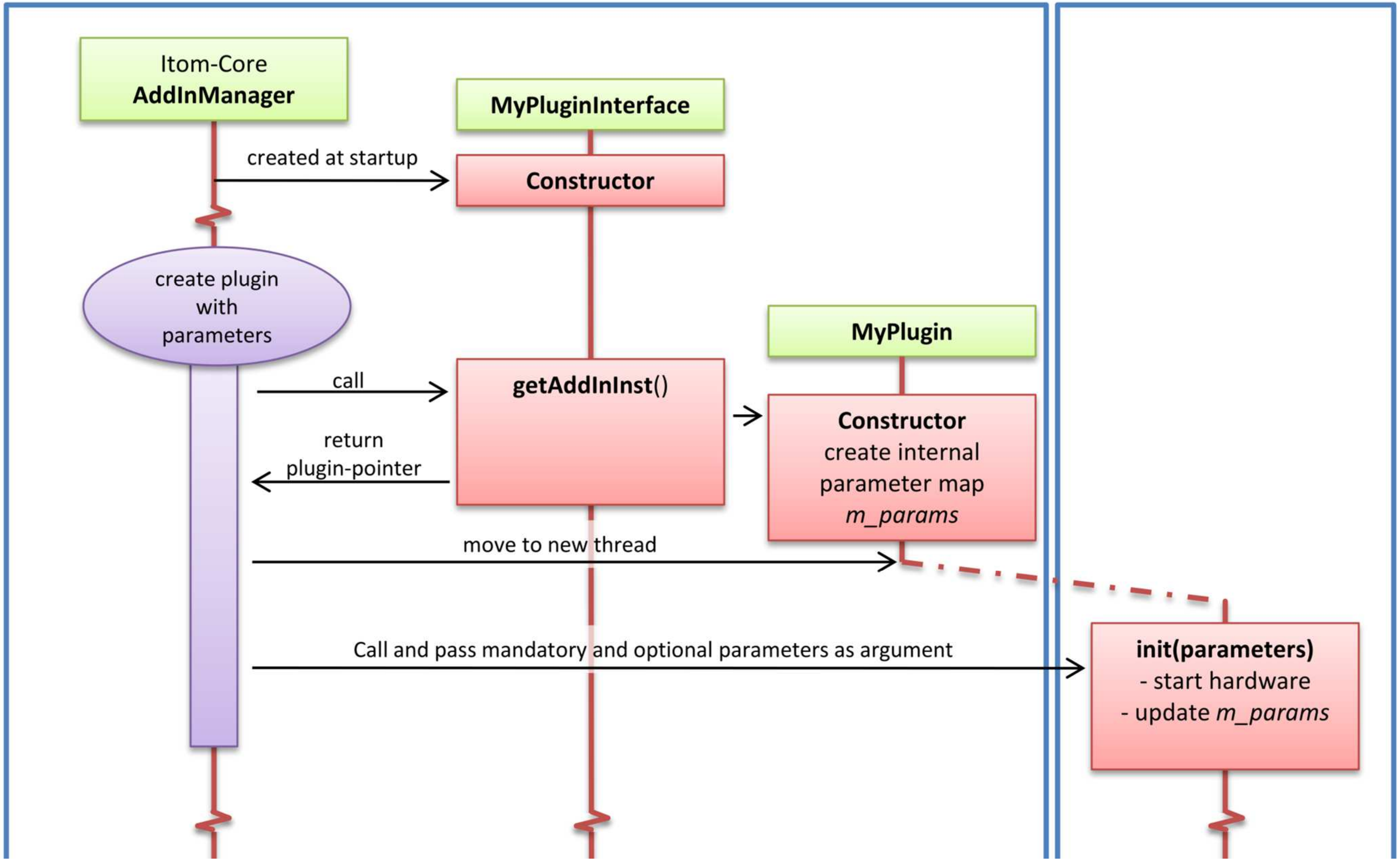
Python Interpreter

Python Debugger

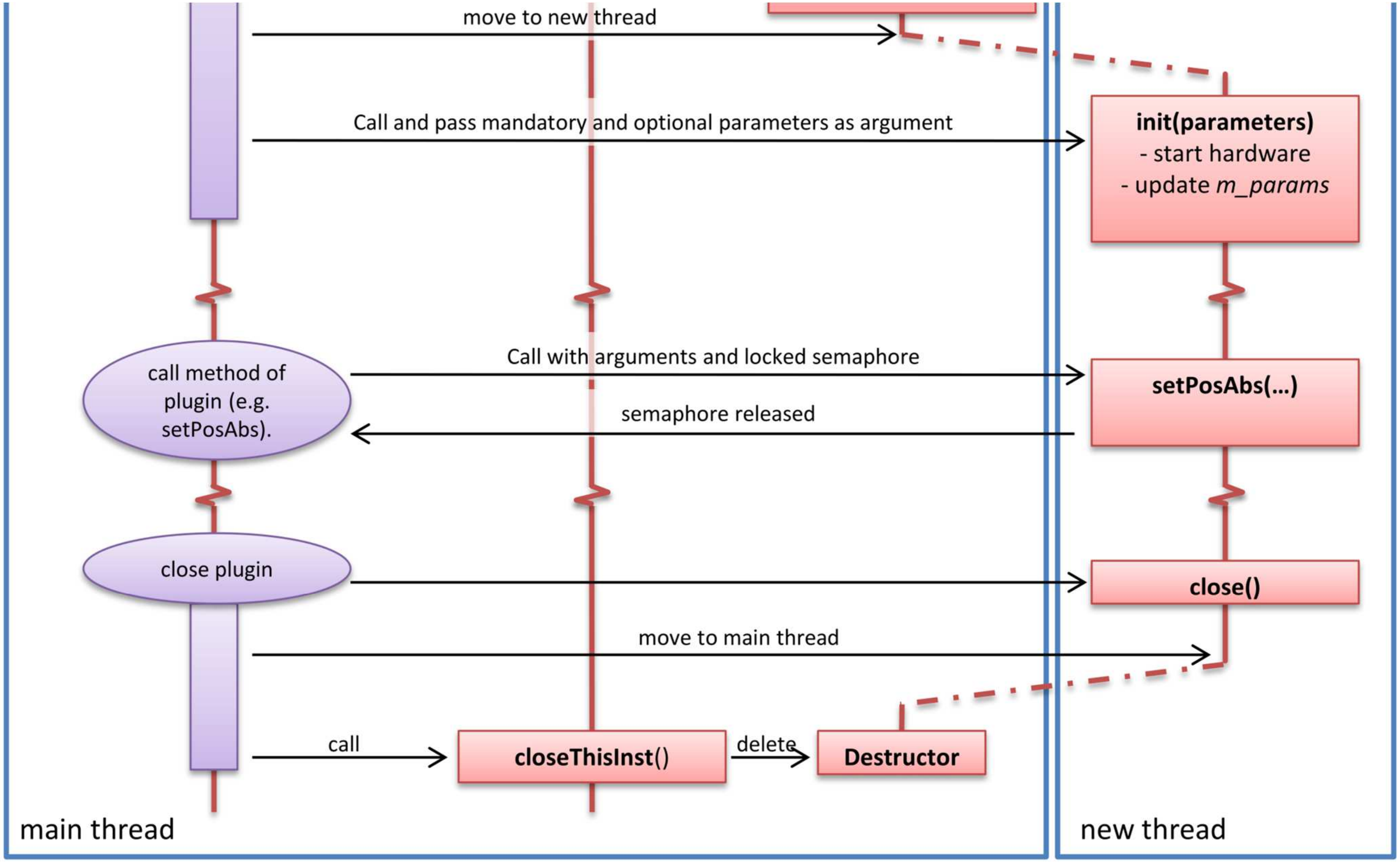
Plugin Thread

Each plugin runs in its own, separate thread. The corresponding GUI elements are pushed to the main thread.

# Life-Cycle of Plugin Instance (I)



# Life-Cycle of Plugin Instance (I)



move to new thread

Call and pass mandatory and optional parameters as argument

**init(parameters)**
- start hardware
- update *m_params*

call method of plugin (e.g. setPosAbs).

Call with arguments and locked semaphore

**setPosAbs(…)**

semaphore released

close plugin

**close()**

move to main thread

call

**closeThisInst()**

delete

**Destructor**

main thread

new thread

What are plugins?

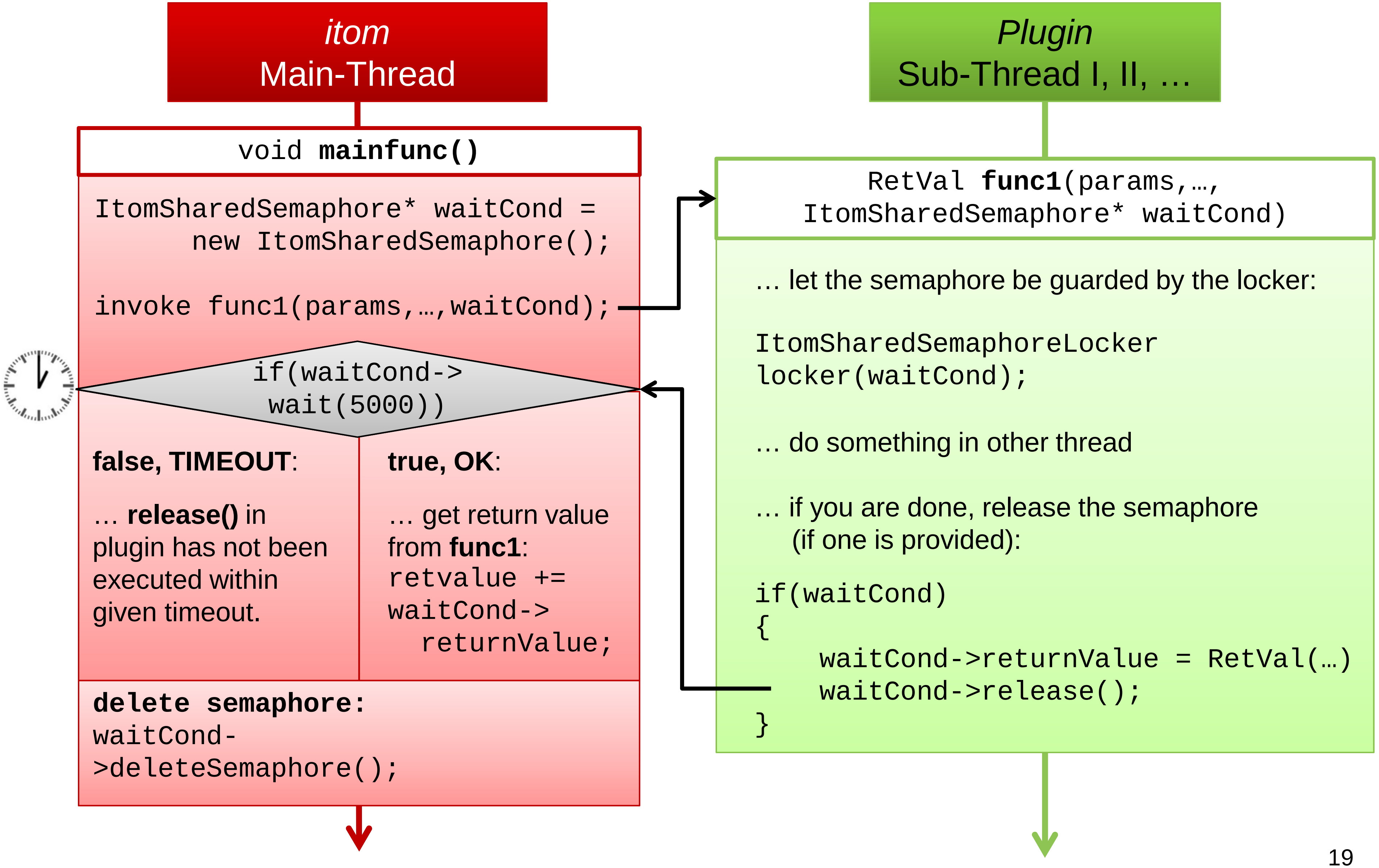
Plugin architecture

Important classes and structures

Working principle of plugins

# class ItomSharedSemaphore

# Thread-Communication using semaphores



*itom*
Main-Thread

```
void mainfunc()
```

```
ItomSharedSemaphore* waitCond =
      new ItomSharedSemaphore();

invoke func1(params,…,waitCond);
```

```
if(waitCond->
  wait(5000))
```

**false, TIMEOUT**:

… **release()** in plugin has not been executed within given timeout.

**true, OK**:

… get return value from **func1**:

```
retvalue +=
waitCond->
   returnValue;
```

```
delete semaphore:
waitCond-
>deleteSemaphore();
```

*Plugin*
Sub-Thread I, II, …

```
RetVal func1(params,…,
ItomSharedSemaphore* waitCond)
```

… let the semaphore be guarded by the locker:

```
ItomSharedSemaphoreLocker
locker(waitCond);
```

… do something in other thread

… if you are done, release the semaphore
(if one is provided):

```
if(waitCond)
{
    waitCond->returnValue = RetVal(…)
    waitCond->release();
}
```

# Thread-safe data transmission

**Problem:**

- Obtain value(s) from another function

**Simple Solution:**

```
void func1()
{
    double val = func2();
}
```

```
double func2()
{
    return 2.0;
}
```

**More flexible solution for multiple values:**

```
void func1()
{
    double v1, v2;
    int ret = func2(v1, v2);
}
```

```
int func2(double &a1, double &a2)
{
    a1 = 2.0;
    a2 = 3.0;
    return 0; //success
}
```

# Thread-safe data transmission

## What happens?

```
void func1()
{
    double v1, v2;
    int ret = func2(v1, v2);
}

int func2(double &a1, double &a2)
{
    a1 = 2.0;
    a2 = 3.0;
    return 0; //success
}
```

Two double variables are reserved in memory

By reference: The address (*0xabff)* of the variables are passed.

***func2*** assigns *2.0* to ***a1***. This is the same chunk of memory than ***v1*** of ***func1***.

Memory:

| | |
|---|---|
| | … |
| 0xabfe → | `value of v1` |
| 0xabff → | `value of v2` |

# Thread-safe data transmission

## Equal operations:

```
void func1()
{
    double v1, v2;
    int ret = func2(v1, v2);
}

int func2(double &a1, double &a2)
{
    a1 = 2.0;
    a2 = 3.0;
    return 0; //success
}
```

```
void func1()
{
    double v1, v2;
    int ret = func2(&v1, &v2);
}

int func2(double *a1, double *a2)
{
    *a1 = 2.0;
    *a2 = 3.0;
    return 0; //success
}
```

---

```
int *intPtr;
print(intPtr) -> 0xffee83ef //addr
*intPtr = 2; //dereferencing
print(*intPtr) -> 2
```

```
int val = 2;
print(val) -> 2
int *intPtr = &val; //referencing
*(&val) = 2; //equal than val = 2
```

# Thread-safe data transmission

## Asynchronous function calls:

```
void func1()
{
    double v1;
    invoke func2(&v1);
}
```

```
void func2(double *a1)
{
    *a1 = 2.0;
}
```

## Scenario 1:

memory for `v1` is allocated in `func1`

`func1` invokes `func2` (in different thread)

`func2` finishes with success

`func1` has a modified variable `v1`

memory of `v1` is deleted if `func1` ends

✓

## Scenario 2:

memory for `v1` is allocated in `func1`

`func1` invokes `func2` (in different thread)

`func2` is executed with **delay**

`func1` receives timeout

memory of `v1` is deleted if `func1` ends

`func2` still wants to access `v1`  (CRASH)

✗

# Thread-safe data transmission

**Solution (thread-safe):**

- SharedPointer
- C++11: std::shared_ptr
- Boost: boost::shared_ptr
- Qt: QSharedPointer

```
{
    QSharedPointer<int> a(new int); //new creation, underlying memory
                                    //allocated (ref = 1)
    {

        QSharedPointer<int> b = a; //assignment, ref incremented (2)

    } //b deleted, ref of memory is decremented (1)

} //a deleted, ref of memory is decremented (0) -> memory is deleted
```

# Thread-safe data transmission

```
void func1()
{
    QSharedPointer<double> v1(new double);
    func2(v1)
}



void func2(QSharedPointer<double> a)
{
    *a = 2.0;
}
```

template parameter indicates type of value

allocated memory is passed to QSharedPointer

`v1` is passed to `func2` in terms of variable a → copy constructor, increment reference

syntax: consider a to be a pointer.

a runs out scope and is deleted, the reference of the underlying memory is decremented.

# Stack vs. Heap

## Heap

```
void func1()
{
    double *v1 = new double;
    func2(v1)
    delete v1;
    v1 = NULL;
}
```

64bit of memory is reserved (allocated) on the heap

Memory needs to be freed.

## Stack

```
void func1()
{
    double v1;
    func2(&v1)
}

void func2(double *a)
{
    *a = 2.0;
}
```

64bit of memory is reserved (allocated) on the stack

v1 runs out of scope, its memory is automatically freed.

# class
# ito::RetVal

# RetVal as general status / return value

**Problem:**

- Status messages like success or any error needs to be returned.
- Often done by simple `int` return values (e.g. 0 → success, -x → error no x)

- We want to have a unified status message system with the following features:
  - different status levels (ok, warning, error)
  - error codes
  - error message text transmissions

**Solution:**

Class **ito::RetVal** in common/retVal.h

# RetVal as general status / return value

```
ito::RetVal ret1; //status: ok, no message
ito::RetVal ret2(ito::retError, 1002, "my message") //status: error
ito::RetVal ret3(ito::retWarning, 1003, "warn") // status: warning

//appending errors
ret1 += ret2; //append ret2 to ret1, ret1 contains now error
ret1 += ret3; //add ret3, status is still error!!!

if (ret1.containsError())
{
    int code =ret1.errorCode();
    std::cout << ret1.errorMessage() << std::endl;
}
```

see example *education/retVal*

# class ito::ParamBase, ito::Param

# Generic parameter passing

**Desired:**

- Pass parameters with different types, but unknown types at compile time, to other functions.
- Add a description and further meta information to these parameters.

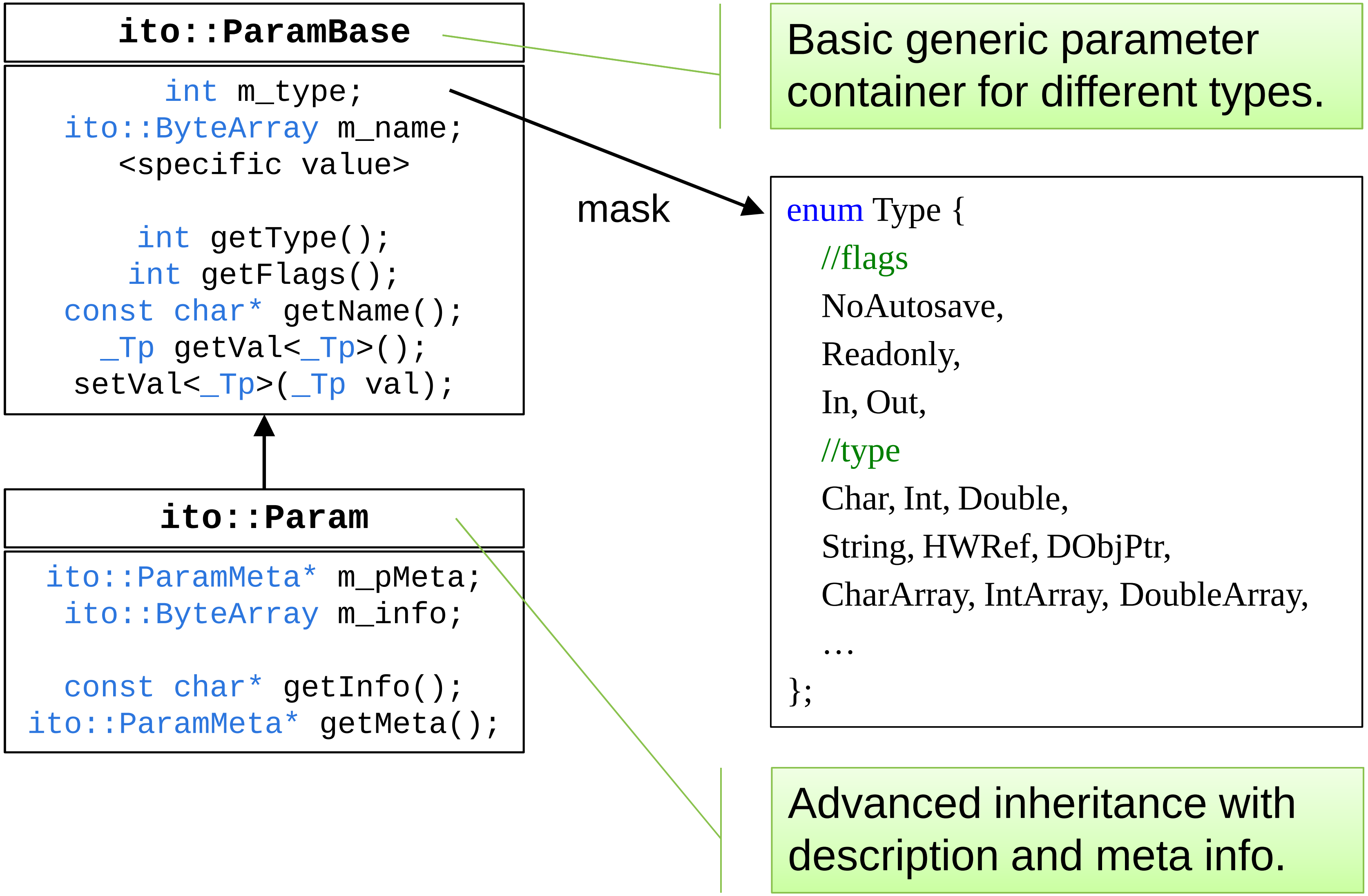
**Problem:**

- C++ is a type-based language, types of variables need to be known at compile-time.

**Solution:**

- Generic parameter class
- Qt: QVariant
- itom: ito::Param, ito::ParamBase (*in common/param.h*)

# Generic parameter passing



```
ito::ParamBase
```

```
int m_type;
ito::ByteArray m_name;
<specific value>

int getType();
int getFlags();
const char* getName();
_Tp getVal<_Tp>();
setVal<_Tp>(_Tp val);
```

```
ito::Param
```

```
ito::ParamMeta* m_pMeta;
ito::ByteArray m_info;

const char* getInfo();
ito::ParamMeta* getMeta();
```

```
enum Type {
    //flags
    NoAutosave,
    Readonly,
    In, Out,
    //type
    Char, Int, Double,
    String, HWRef, DObjPtr,
    CharArray, IntArray, DoubleArray,
    …
};
```

Basic generic parameter container for different types.

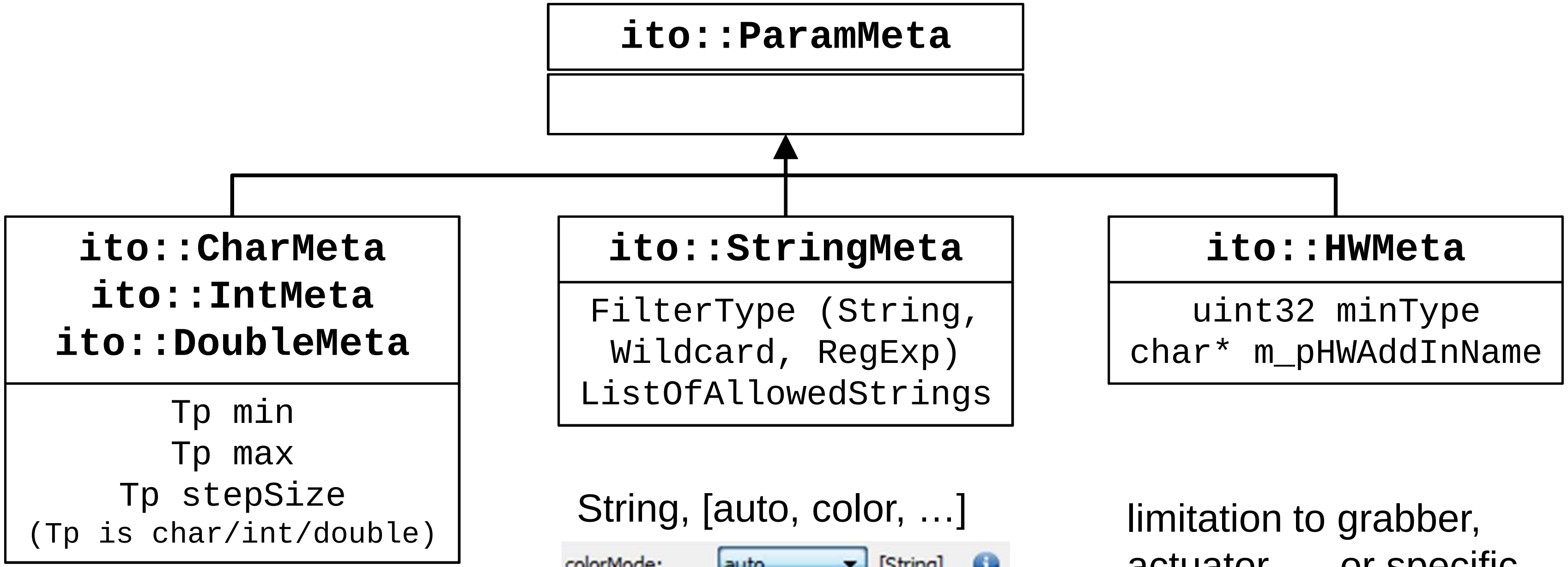
Advanced inheritance with description and meta info.

# ito::ParamBase

```
//integer parameter
ito::ParamBase p1("param1", ito::ParamBase::Int, 2);
p1.setVal<int>(3);
int value = p1.getVal<int>();

//double parameter
ito::ParamBase p2("param2", ito::ParamBase::Double, 2.0);
p2.setVal<double>(3.0);
double value = p2.getVal<double>();

//string parameter
ito::ParamBase p3("param3", ito::ParamBase::String, "hello");
p3.setVal<char*>("test");
char* value = p3.getVal<char*>();

//dataObject parameter
ito::DataObject dObj(3,5,2,ito::tFloat32);
ito::ParamBase p4("param4", ito::ParamBase::DObjPtr, NULL);
p4.setVal<ito::DataObject*>(&dObj);
ito::DataObject *value = p4.getVal<ito::DataObject*>();
```

Ptr-based types: Be careful, no shared pointers are used, therefore do not delete the original object before the last use of the value.

# Meta Information (I)



ito::ParamMeta

ito::CharMeta
ito::IntMeta
ito::DoubleMeta

Tp min
Tp max
Tp stepSize
(Tp is char/int/double)

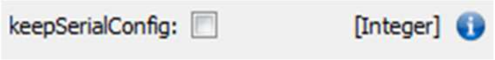
ito::StringMeta

FilterType (String, Wildcard, RegExp)
ListOfAllowedStrings

ito::HWMeta

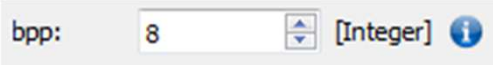
uint32 minType
char* m_pHWAddInName

int, 1, [0,1]



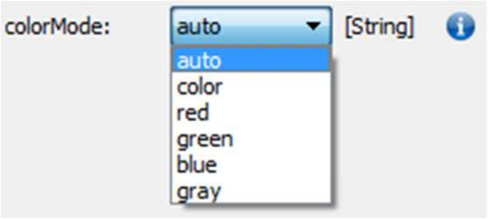
keepSerialConfig: □ [Integer]

int, 8, [0,32]



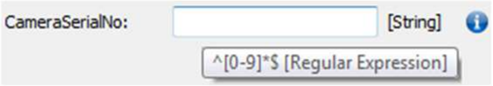
bpp: 8 [Integer]

String, [auto, color, …]



colorMode: auto [String]

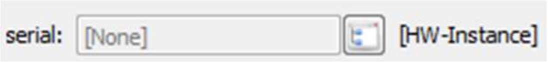
RegExp, only numbers



CameraSerialNo: [String]

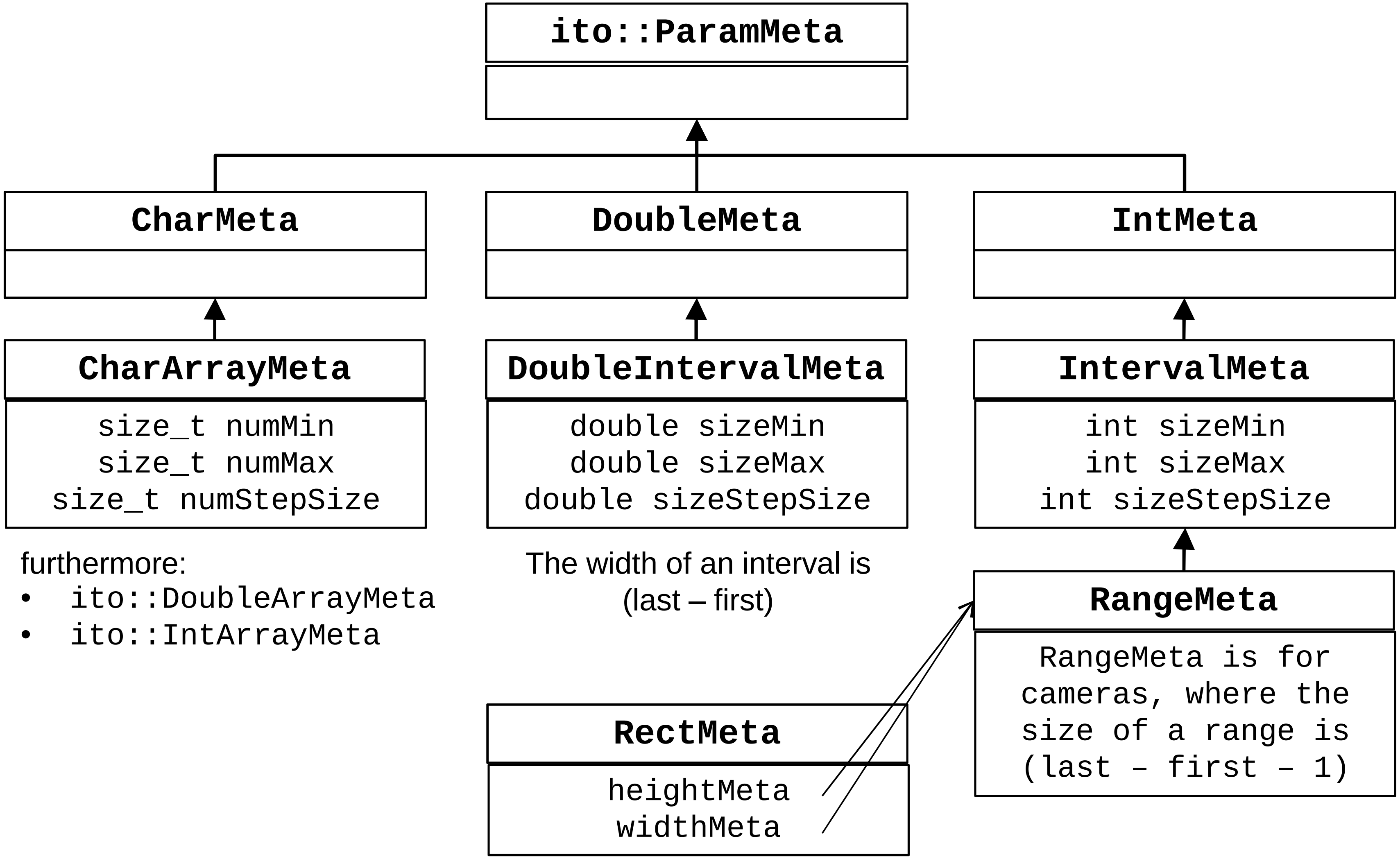
^[0-9]*$ [Regular Expression]

limitation to grabber, actuator, … or specific plugin name possible



serial: [None] [HW-Instance]

# Meta Information (II)



ito::ParamMeta

CharMeta

DoubleMeta

IntMeta

CharArrayMeta

size_t numMin
size_t numMax
size_t numStepSize

DoubleIntervalMeta

double sizeMin
double sizeMax
double sizeStepSize

IntervalMeta

int sizeMin
int sizeMax
int sizeStepSize

RangeMeta

RangeMeta is for cameras, where the size of a range is (last – first – 1)

RectMeta

heightMeta
widthMeta

furthermore:

- ito::DoubleArrayMeta
- ito::IntArrayMeta

The width of an interval is (last – first)

# ito::Param

```
//integer value between 0 and 10, default: 5
ito::Param param("intNumber", ito::ParamBase::Int, 0, 10, 5, "description");

// or
ito::Param param("intNumber", ParamBase::Int, 5, new IntMeta(0,10),
"description");

// or (integer-variable without meta information)
ito::Param param("intNumber", ParamBase::Int, 5, NULL, "description");
param.setMeta(new IntMeta(0,10), true); //take ownership of IntMeta-instance

// accessing the min-max-value is obtained by getting the IntMeta-struct
IntMeta *meta = dynamic_cast<IntMeta*>(param.getMeta());
int min = meta->getMin()     //returns 0
int max = meta->getMax()     //returns 10
```

# ito::Param

- *ito::Param* inherits from *ito::ParamBase*
- *ito::Param* has all than *ito::ParamBase* has including a pointer to an additional *ito::ParamMeta* instance and a description string.
- A new value set to the parameter using *setVal* is **never** checked with respect to the given meta information!
- This check can be done using the api function

```
ito::Param tmpl("tmpl", ito::ParamBase::Int, 2, \
      new ito::IntMeta(0, 5));
ito::ParamBase param("test", ito::ParamBase::Int, 7);

ito::RetVal ret = apiValidateParam(tmpl, param, true, false);
//return retError since value of 'test' is out of bounds.
```

# Excursion: itom API functions

- itom provides some functions that can be used by all plugins
- defined in itom API, accessible via

```
common/apiFunctionsInc.h
common/apiFunctionsGraphInc.h
```

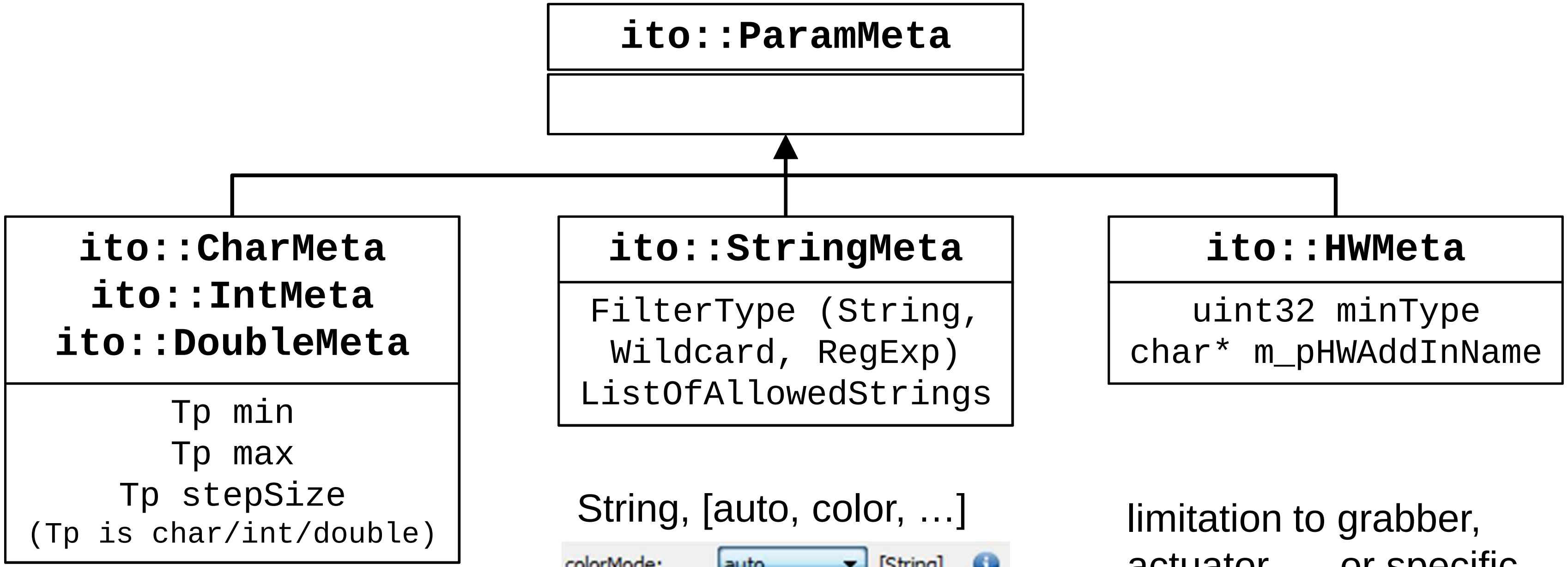
- In your main source file (cpp!!!) of the plugin, write at the beginning (before any other include statement):

```
#define ITOM_IMPORT_API
#define ITOM_IMPORT_PLOTAPI
```

- Then you can use functions like:

```
apiParseParamName(val->getName(), key, hasIndex, idx, suffix);
apiGetParamFromMapByKey(m_params, key, it, true);
apiValidateParam(*it, *val, false, true);
```

# Meta Information



ito::ParamMeta

ito::CharMeta
ito::IntMeta
ito::DoubleMeta

Tp min
Tp max
Tp stepSize
(Tp is char/int/double)

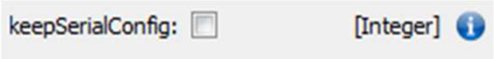
ito::StringMeta

FilterType (String, Wildcard, RegExp)
ListOfAllowedStrings

ito::HWMeta

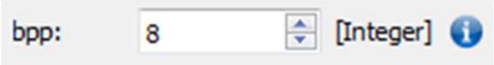
uint32 minType
char* m_pHWAddInName

int, 1, [0,1]



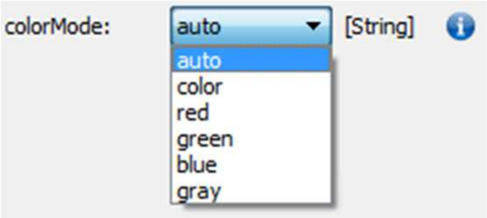
keepSerialConfig: □ [Integer]

int, 8, [0,32]



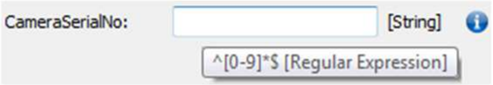
bpp: 8 [Integer]

String, [auto, color, …]



colorMode: auto [String]

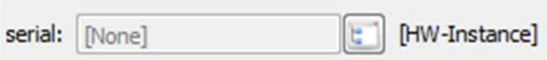
RegExp, only numbers



CameraSerialNo: [String]

^[0-9]*$ [Regular Expression]

limitation to grabber, actuator, … or specific plugin name possible



serial: [None] [HW-Instance]

# class
# ito::DataObject

# Data Object

**Goal:**

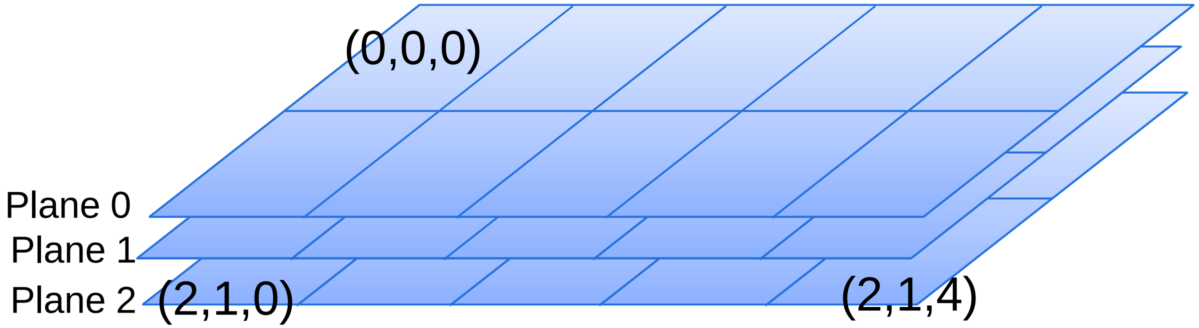
- Different basic types of data (including complex)
- Processing of very large, multi-dimensional data sets (series of images)
- Compatible with Matlab, Numpy

**Implementation:**

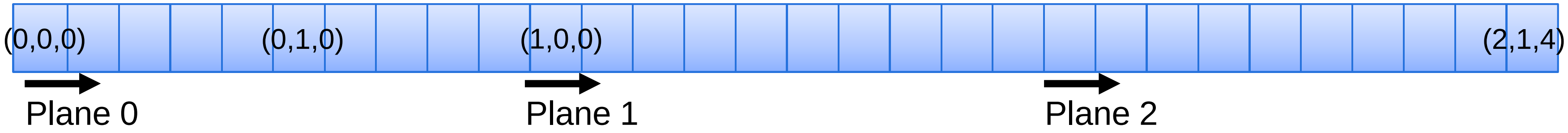
- *DataObject* very similar to OpenCV data structures
- Basic data types supported: *int8, uint8, int16, uint16, int32, uint32, float, double, complex(float), complex(double)*
- These data types were chosen as they are in the overlap of  Numpy and OpenCV
- *DataObject* supports tags

# Data Storage

**Assume:** Series of 2D-images *(3 x 2 x 5)*



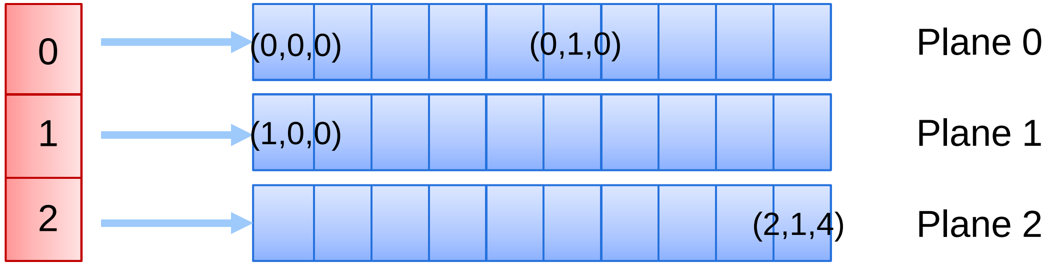
**C / Matlab:** continuous chunk of memory



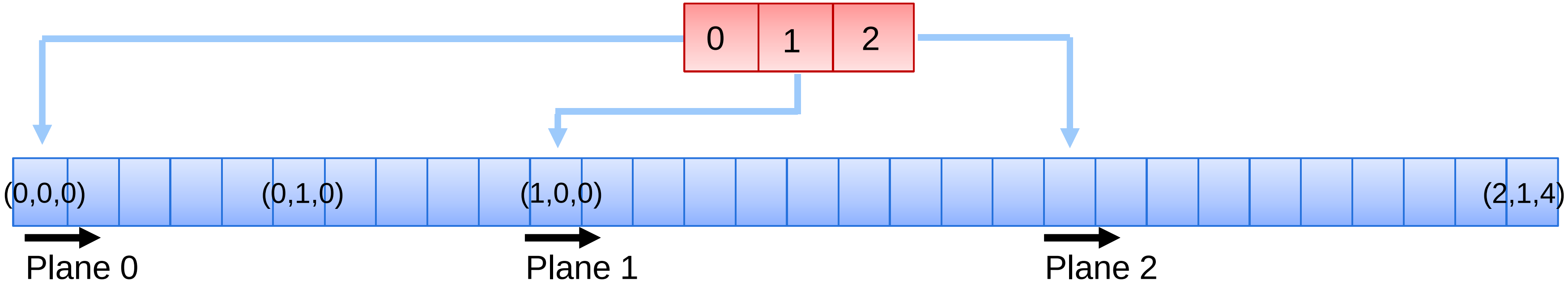
# DataObject

## DataObject:

N-2 dimensional vector of 2D matrices (implemented as continuous memory)



## continuous DataObject:



# *DataObject:* Constructors and Functions

- dataObject()
  - dims, dtype, data,…
- dataObject(*array*)
- eye()
- ones()
- zeros()
- rand()
- randN()

- adjustROI()
- locateROI()
- copy(*region_only=0*)
- set*[Metadata]*()
- …

What are plugins?

Plugin architecture

Important classes and structures

Working principle of plugins

# Algorithms

# Algorithm plugins

- algorithm plugin can contain multiple algorithms
- Their name must be unique within itom, else they are rejected.
- Every algorithm consists of two static methods plus one doc-string:

```
static const char* algo1doc;

static ito::RetVal algo1Params( QVector<ito::Param> *paramsMand, \
                                QVector<ito::Param> *paramsOpt, \
                                QVector<ito::Param> *paramsOut);

static ito::RetVal algo1(       QVector<ito::ParamBase> *paramsMand, \
                                QVector<ito::ParamBase> *paramsOpt, \
                                QVector<ito::ParamBase> *paramsOut);
```

- The real algorithm is defined in *algo1*

# Algorithm plugins (II)

```
static ito::RetVal algo1Params( QVector<ito::Param> *paramsMand, \
                                QVector<ito::Param> *paramsOpt, \
                                QVector<ito::Param> *paramsOut);
```

- defines vectors of multiple mandatory parameters, optional parameters and output parameters
- every parameter has a name, type, default value (important for optional ones only) and a description.

# Algorithm plugins (III)

```
[ou1,out2,...] = itom.filter("algo1", mand1, mand2, ..., opt1, ...)
```

- resolve method *algo1* and *algo1Params* from string „algo1“ (see later)
- call *algo1Params* and get vectors of mandatory, optional and output parameters

```
static ito::RetVal algo1Params( QVector<ito::Param> *paramsMand, \
                                QVector<ito::Param> *paramsOpt, \
                                QVector<ito::Param> *paramsOut);
```

- update vectors with user input in python
- call *algo1* and execute algorithm. *algo1* can modify certain mandatory and optional parameters as well as the output vector (see later).

```
static ito::RetVal algo1(       QVector<ito::ParamBase> *paramsMand, \
                                QVector<ito::ParamBase> *paramsOpt, \
                                QVector<ito::ParamBase> *paramsOut);
```

- itom checks errors and returns the set of output parameters as return tuple in python.

# Algorithm plugins (IV)

**Modifiers** `ito::ParamBase::In, ito::ParamBase::Out`

**In:**
Parameter is only read but not changed within the plugin. Applicable to all types of mandatory and optional parameters.

```
ito::Param("mandParam1", ito::ParamBase::String | ito::ParamBase::In, \
        "default", "description") );
```

**In | Out** (both flags set!):
Parameter is read and modified by plugin. Applicable to all pointer-based types of mandatory and optional parameters (e.g. dataObject).

```
ito::Param("optParam1", ito::ParamBase::DObjPtr | ito::ParamBase::In | \
         ito::ParamBase::Out, NULL, "description") );
```

**Out** (both flags set!):
Parameter is only set within the plugin. All output parameters must have this option, not applicable to dataObjects, pointClouds, polygonMeshes!

```
ito::Param("outParam1", ito::ParamBase::Double | ito::ParamBase::Out, \
        0.0, ito::DoubleMeta::all(), "description") );
```

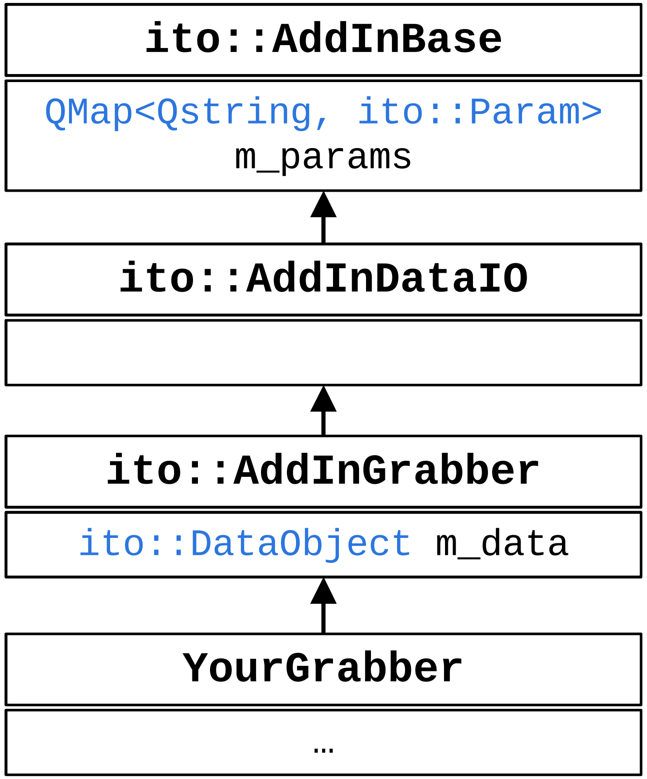
# Algorithm plugins (V)

## Register new algorithm

```
ito::RetVal AlgoPlugin::init(
        QVector<ito::ParamBase> * /*paramsMand*/,
        QVector<ito::ParamBase> * /*paramsOpt*/,
        ItomSharedSemaphore * /*waitCond*/)
{
    ito::RetVal retval = ito::retOk;
    FilterDef *filter = NULL;

    //register each algorithm with the following code snippet
    filter = new FilterDef(AlgoPlugin::algo1, \
        AlgoPlugin::algo1Params, tr(algo1doc));
    m_filterList.insert("algo1", filter);

    setInitialized(true);
    return retval;
}
```

# Grabber / Camera

# Structure



Parameters:

**Required**

- name (string) → name of plugin (read-only)
- bpp (int) → bit-depth 8, 10, 12 …
- sizex (int) → current width of image (read-only)
- sizey (int) → current height of image (read-only)

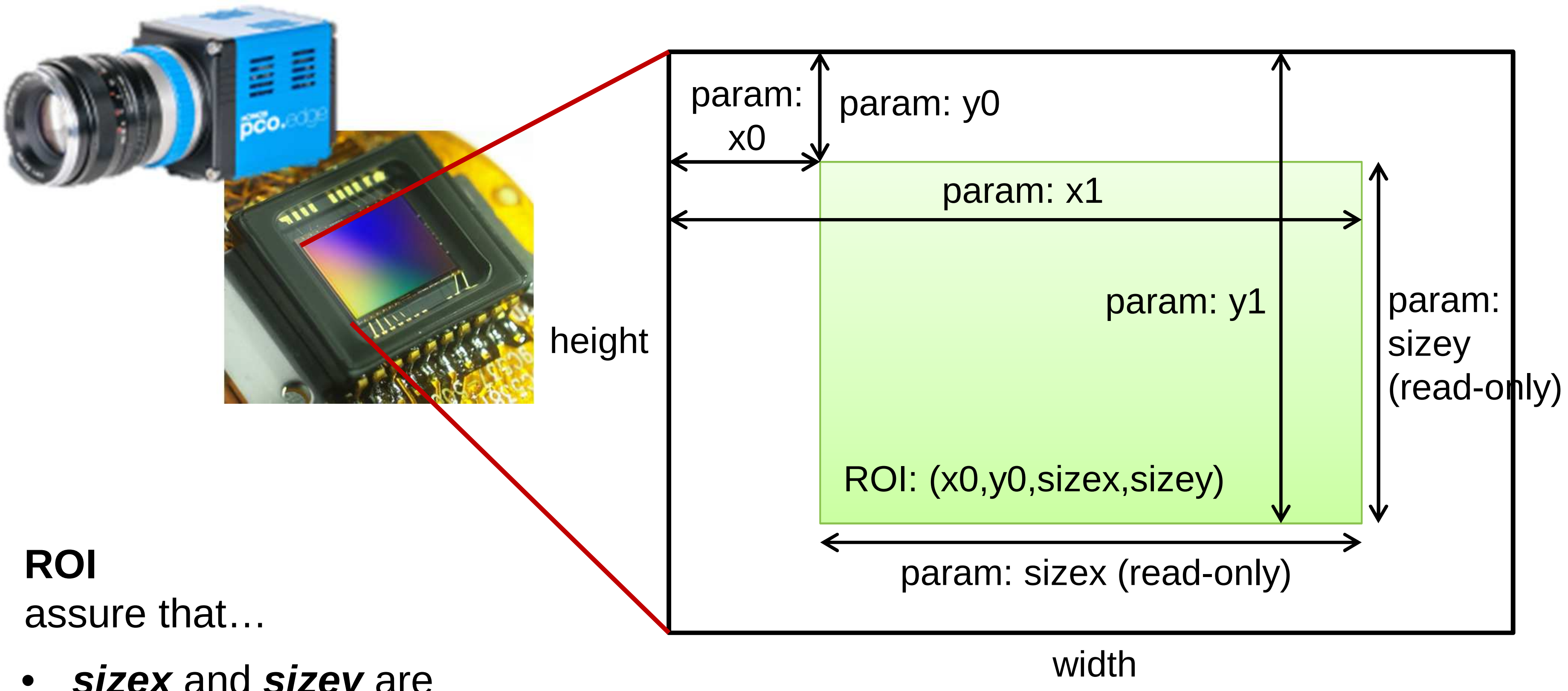
**Optional**

- x0, x1 (int) → left and right index of ROI (adjusts sizex) deprecated: use roi
- y0, y1 (int) → top and bottom index of ROI (adjusts sizey) deprecated: use roi
- roi (int-array) → (x0,y0,width,height) of ROI
- integration_time, frame_time, gain, offset…

If a parameter changed, inform the GUI by

```
emit parametersChanged(m_params);
```

# Image Size / ROI



**ROI**
assure that…

- ***sizex*** and ***sizey*** are always dependent on $x0, x1, y0, y1$
- $0 \leq x0 < x1$
- $0 \leq y0 < y1$
- $1 \leq x1 < width$
- $1 \leq y1 < height$

**ito::DataObject m_data**

- m_data has the size of the ROI and a data type that fits to the currently bpp.
- Reallocate m_data once ***bpp, x0, x1, y0*** or ***y1*** changed → done in *checkData()*

# Camera connection

**init(mandParams, optParams)**

- connect to the camera
- update parameters (m_param) of the plugin with respect to current camera parameters.
- setIdentifier(specificCamName)

**startDevice(…)**

- make the camera ready for acquisition in a triggered mode
- e.g. allocate necessary camera buffers

⚠ startDevice can be called multiple times (e.g. by live windows). Therefore count the calls and only start the camera during the first call:

```
- void incGrabberStarted()
- void decGrabberStarted()
- int grabberStartedCount()
```

# Camera disconnection

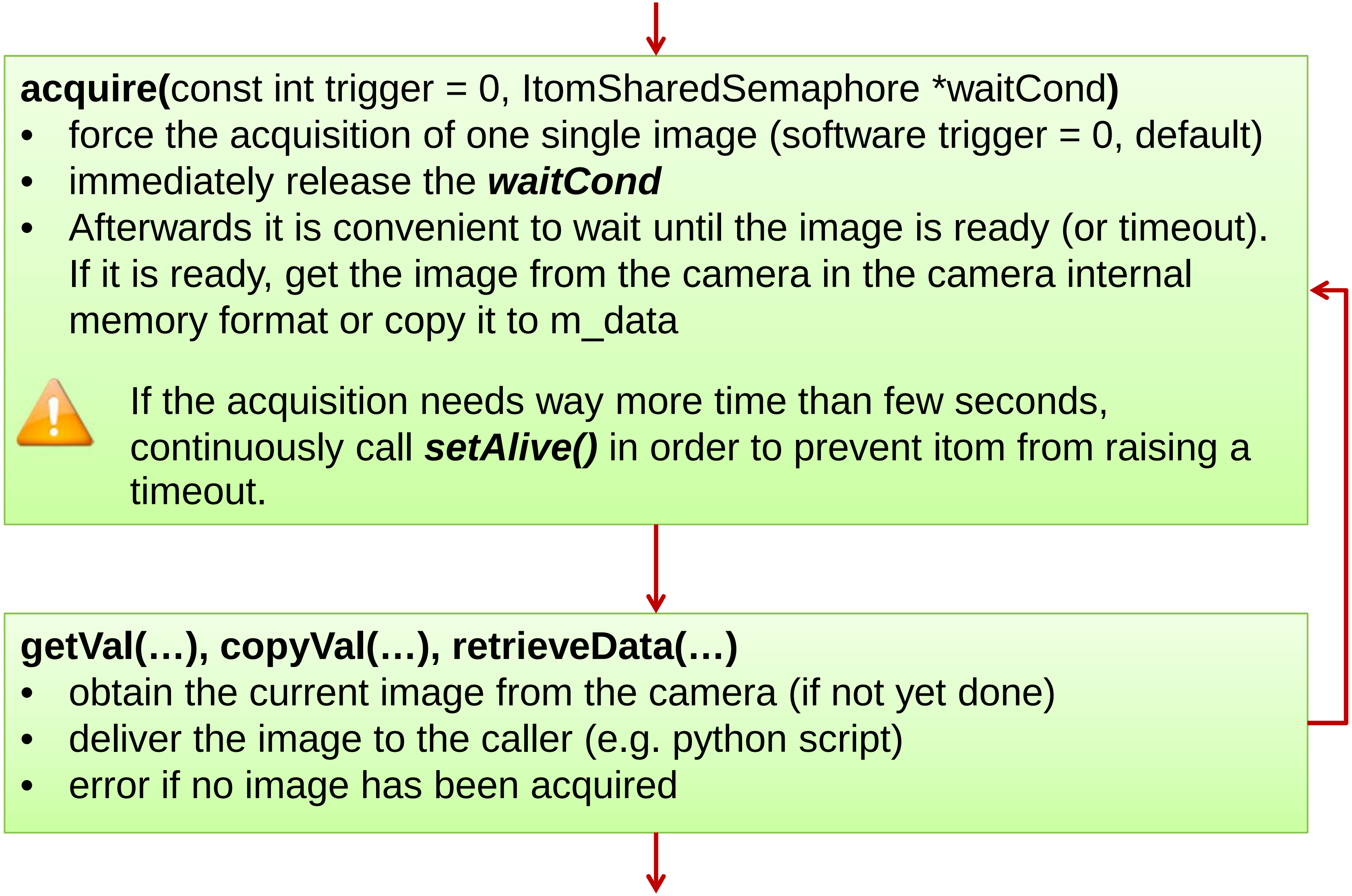
**stopDevice(…)**

- decrement the counter (`void decGrabberStarted()`)
- If last: delete buffers, stop camera acquisition

**close(…)**

- stopDevice(…) if not yet done
- disconnect from camera

# Acquisition (I)



**acquire(**const int trigger = 0, ItomSharedSemaphore *waitCond**)**

- force the acquisition of one single image (software trigger = 0, default)
- immediately release the ***waitCond***
- Afterwards it is convenient to wait until the image is ready (or timeout). If it is ready, get the image from the camera in the camera internal memory format or copy it to m_data

⚠ If the acquisition needs way more time than few seconds, continuously call ***setAlive()*** in order to prevent itom from raising a timeout.

**getVal(…), copyVal(…), retrieveData(…)**

- obtain the current image from the camera (if not yet done)
- deliver the image to the caller (e.g. python script)
- error if no image has been acquired

# Acquisition (II)

**getVal(**void *vpdObj, ItomSharedSemaphore *waitCond**)**

- save camera image in *m_data*
- deliver reference to *m_data* in given *vpdObj* (ito::DataObject*)
- inform connected live windows about new data in *m_data*

```
ito::DataObject *dObj = \
        reinterpret_cast<ito::DataObject *>(vpdObj);
//data from camera -> m_data
retValue += retrieveData(/*no args*/);    TODO
//live images
sendDataToListeners(0);
//deliver reference
(*dObj) = m_data;
```

+ fast delivery to user due to reference

- image is not persistent, the next acquisition changes the delivered data (but safe)

# Acquisition (III)

**copyVal(**void *vpdObj, ItomSharedSemaphore *waitCond**)**

- save camera image in *m_data* ONLY IF live window connected
- save camera image in externally given data object (ALWAYS)
- inform connected live windows about new data in *m_data*

```
ito::DataObject *dObj =
reinterpret_cast<ito::DataObject *>(vpdObj);
retValue += retrieveData(dObj); //pass external
object
sendDataToListeners(0);
```

**TODO**

**external Object:**

- empty → will be reallocated to right size and type. OK.
- 2D, right type, right size → image is copied into the given memory. OK.
- 3D, right type, ROI has the right „2D"-size → image is copied into ROI. OK.
- else → ERROR.

+ external object can be a 3D stack → image is stored in one plane<br>
+ image is persistent due to deep copy<br>
- slightly slower (marginal)

# Acquisition (IV)

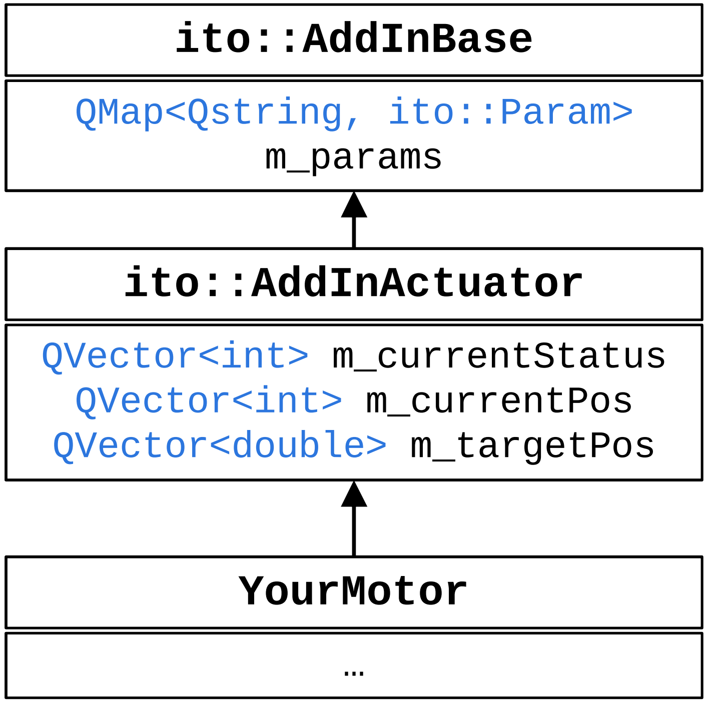
**retrieveData(**ito::DataObject *externalDataObject**)**

- if *externalDataObject* → check it and copy recent image data into (ROI of) this external data object
- else (NULL): → check *m_data* and copy recent image data into this.
- if connected live windows → always additionally copy recent image into m_data (independent on *externalDataObject*)

```
bool hasListeners = (m_autoGrabbingListeners.size() > 0);
bool copyExternal = (externalDataObject != NULL);

if (externalDataObject && hasListeners) { checkData(NULL); //update m_data }
else { checkData(externalDataObject); }

if (imageType == ito::tUInt8) {
    if (copyExternal)
        externalDataObject->copyFromData2D<ito::uint8>( \
            (ito::uint8*) imgBuffer, nrCols, nrRows);
    if (!copyExternal || hasListeners)
        m_data.copyFromData2D<ito::uint8>( \
            (ito::uint8*) m_pImaqBuffer, imgBuffer, nrCols, nrRows);
}
else {...}
}
```

# Actuator

# Structure



ito::AddInBase

QMap<Qstring, ito::Param> m_params

ito::AddInActuator

QVector<int> m_currentStatus
QVector<int> m_currentPos
QVector<double> m_targetPos

YourMotor

…

Parameters:

**Required**

- name (string) → name of plugin (read-only)
- numAxis (int) → number of axes
- async (int) → 1: asynchronous mode, 0: synchronous mode (default)

**Optional**

- speed (double , doubleArray)
  → axis-specific speed (in mm/s or °/s)
- accel (double , doubleArray)
  → axis-specific acceleration (in mm/s$^2$ or °/s$^2$)
- decel(double , doubleArray)
  → axis-specific deceleration (in mm/s$^2$ or °/s$^2$)

If a parameter changed, inform the GUI by

```
emit parametersChanged(m_params);
```

# Status, Current Position, Targets

Status: `QVector<int> m_currentStatus`

- size = number of axes
- each value is a bitmask representing the axis specific status
- once changed, inform the GUI using `sendStatusUpdate(true)`

Enum ito::tActuatorStatus (in addInInterface.h)

status flags

```
actuatorAvailable
actuatorEnabled
```

switches flags

```
actuatorEndSwitch
actuatorLeftEndSwitch
actuatorRightEndSwitch
actuatorRefSwitch
actuatorLeftRefSwitch
actuatorRightRefSwitch
```

moving flags

```
actuatorUnknown
actuatorInterrupted
actuatorMoving
actuatorAtTarget
actuatorTimeout
```

Helper functions to manipulate the bitmasks:

- `setStatus(int &status, const int newFlags, const int keepMask = 0)`
- `replaceStatus(int &status, const int existingFlag, const int replaceFlag)`

# Status, Current Position, Targets

Status: `QVector<int> m_currentStatus`

- size = number of axes
- each value is a bitmask representing the axis specific status
- once changed, inform the GUI using `sendStatusUpdate(true)`

Current positions: `QVector<double> m_currentPos`

- size = number of axes
- each value is the current position of a specific axis (in mm or degree)
- once changed, inform the GUI using `sendStatusUpdate(false)`

Target positions: `QVector<double> m_targetPos`

- size = number of axes
- each value is the target position of a specific axis (in mm or degree)
- once changed, inform the GUI using `sendTargetUpdate()`