SQUIDSTAT GUI

PROGRAMMERS GUIDE

2017

TABLE OF CONTENTS

1 Introduction

This document contains a whole description of the SquidStat GUI software. Developers can use this guide to extend the software functionallity.

2 General description of the structure

The basis of the application is the "MainWindow" class (see fig. 1). It is the Controller of the application. Its duty is to create and coordinate all other classes.

First, the "MainWindow" creates the "MainWindowUI". The "MainWindowUI" is the View of the application. It handles all user activity and operates all UI components.

Next, the "MainWindow" loads prebuild experiment plugins (see p. 12), builder element plugins (see p. 13) and custom experiments (see p. 8).

Next, the "MainWindow" creates the "InstrumentEnumerator" (see p. 5). It continuously checks the set of connected instruments and changes in this set. All changes are reported to the "MainWindow".

Finally, the "MainWindow" creates connections (in terms of Qt) to handle corresponding happenings.

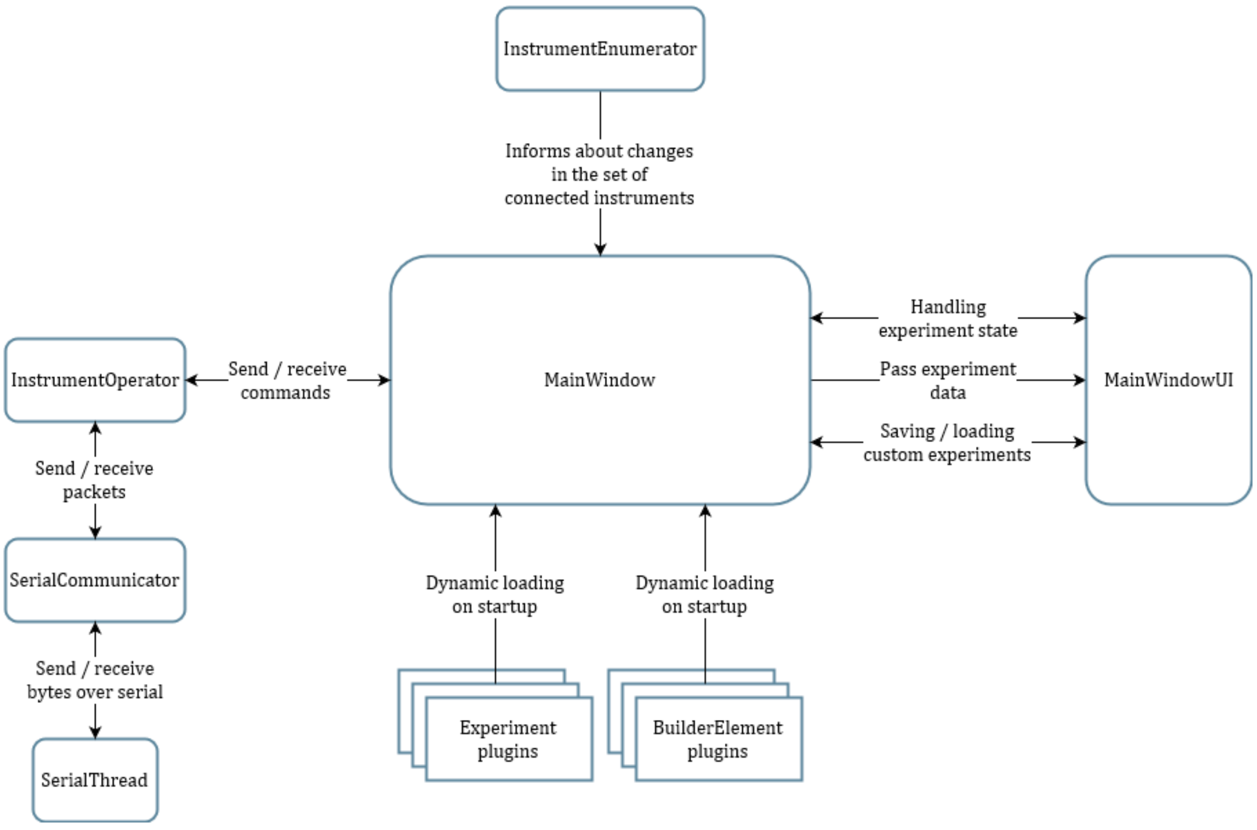

InstrumentEnumerator
Informs about changes in the set of connected instruments
MainWindow
Handling experiment state
Pass experiment data
Saving / loading custom experiments
MainWindowUI
InstrumentOperator
Send / receive commands
Send / receive packets
SerialCommunicator
Send / receive bytes over serial
SerialThread
Dynamic loading on startup
Dynamic loading on startup
Experiment plugins
BuilderElement plugins

Figure 1 – General scheme of the application

Generally, the startup looks as the following:

1. Load fonts from the resources.
2. Create all UI elements.
3. Load prebuild experiment plugins.
4. Load builder element plugins.
5. Load custom experiments.
6. Create the "InstrumentEnumerator".
7. Apply stylesheets (loaded from the resources) to the whole application.

For every connected instrument the "MainWindow" has the instance of the "InstrumentOperator" (see p. 4) that provides API for sending and receiving all supported commands and responses.

The application is mostly single-threaded. Separated threads are used for raw data exchange over serial and for the instrument enumeration. So, the total amount of threads in application is equal to $2 + [amount\ of\ connected\ instruments]$.

3 Serial communicator

For the PC every instrument is the COM-port. Every instrument supports the specific protocol[1] over serial interface.

To handle the communication over the serial the application use two classes: the "SerialThread" and the "SerialCommunicator".

The "SerialThread" is inherited from the "QThread". It owns the corresponding "QSerial" and perform read/write operations over the serial. All Qt-connections outside "SerialThread" have to be of the "Qt::QueuedConnection" type[2]. So, the "SerialThread" reads all data from the serial even on high data rates and queue the data for the further handling to the main thread.

The "SerialCommunicator" is inherited from the "QObject" and owns the "SerialThread". The "SerialCommunicator" handles the instrument protocol over the raw serial data. It allows to send commands and emits signal "SerialCommunicator::ResponseReceived" on every valid packet in the data flow that was recognized.

[1] This document does not contain the description of the instrument protocol
[2] Connection type is the fifth parameter of the "QObject::connect" method

4 Instrument operator

The "InstrumentOperator" class is the abstraction from the instrument protocol for the application. It has API that duplicates every command and every response to guarantee the correctness of the input parameter set and parameters type.

The "InstrumentOperator" owns the "SerialCommunicator".

5 Instrument enumerator

The "InstrumentEnumerator" is the thread that continuously checks the set of the connected instruments and emits signal on every change (arrival or removal). It is inherited from the "QThread".

Every second (hardcoded value) the "InstrumentEnumerator" performs the following:

1. Get the list of COM-ports (available ports).
2. Check if "already connected ports" are still among "available ports".
3. Every "already connected port" that is not in the "available ports" list move to the "instruments to delete" list.
4. Emit the "InstrumentEnumerator::RemoveDisconnectedInstruments" signal for every port from the "instruments to delete" list.
5. For every port from the "available ports" list and not in the "already connected ports" list try to request the handshake via instrument protocol. If success – add this port into the "instruments to add" list.
6. Emit the "InstrumentEnumerator::AddNewInstruments" signal with the "instruments to add" list as a parameter.

Ports are considered to be the same if they have the same name and serial number.

6 What is an experiment

Experiment is a kind of subprogram that tells the instrument what to do. Generally, this "subprogram" is a vector of the "ExperimentNode_t" structures. The "ExperimentNode_t" is a primitive action that the instrument can perform. However, the single "ExperimentNode_t" can describe rather complicated operations[3].

Generally, lifecycle of the experiment is the following:

1. Collect settings from the user.
2. Generate the vector of the "ExperimentNode_t" structures.
3. Transfer the vector to the instrument.
4. Display the data that arrived from the instrument until the experiment is not ended.

Experiments are of the following types:

– regular experiments (see p. 7) – these experiments are prebuilt by the developer, stored in dynamic linked libraries;

– custom experiments (see p. 8) – these experiments are built by the user, stored in text files in format of JSON;

– manual experiments (see p. 10) – direct commands to the instrument sent by the user.

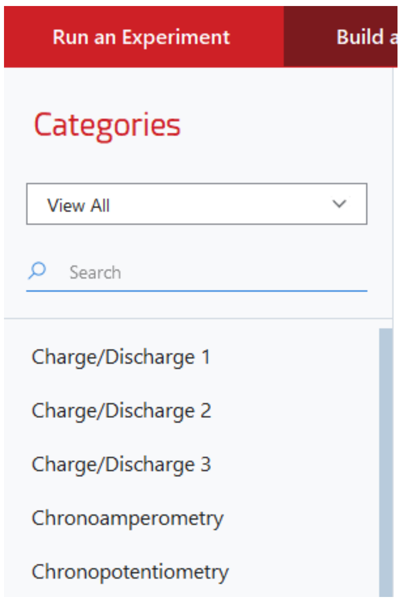
In terms of application the experiment is a class that is inherited from the "AbstractExperiment" interface. See p. 12 for the guide for creating new prebuilt experiments.

[3] This document does not contain the detailed description of the "ExperimentNode_t" structure and its capabilities.

7 Workflow of a regular experiment

First, all plugins that contain prebuilt experiments are loaded on startup in the "MainWindow" class.

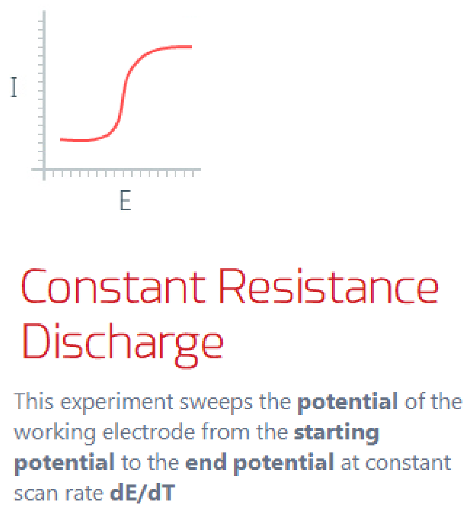
Next, information about loaded experiments is sent to the "MainWindowUI". The list of short names (calling "AbstractExperiment::GetShortName" and "AbstractExperiment::GetCategory") of all loader experiments is shown on the left side of the "Run an Experiment" tab (see fig. 2).



Charge/Discharge 1
Charge/Discharge 2
Charge/Discharge 3
Chronoamperometry
Chronopotentiometry

Figure 2 – List of loaded experiments

When user selects the experiment in the displayed list, "MainWindowUI" shows detailed description of the experiment (calling "AbstractExperiment::GetFullName", "AbstractExperiment::GetDescription", "AbstractExperiment::GetImage") on the central part of the "Run an Experiment" tab (see fig. 3).

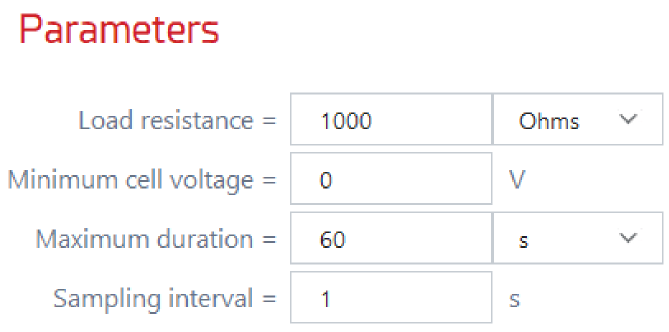


Constant Resistance Discharge

This experiment sweeps the **potential** of the working electrode from the **starting potential** to the **end potential** at constant scan rate **dE/dT**

Figure 3 – Detailed experiment description

Next, the UI for collecting settings from the user is generated (calling the "AbstractExperiment::CreateUserInput") and placed on the right side of the "Run an Experiment" tab (see fig. 4).


Parameters

Load resistance =	1000	Ohms
Minimum cell voltage =	0	V
Maximum duration =	60	s
Sampling interval =	1	s

Figure 4 – UI for parameters setting of a specific experiment

When user clicks the "Start Experiment" button for the selected experiment the vector of "ExperimentNode_t" structures is generated (calling the "AbstractExperiment::GetNodesData").

Next, for each experiment type that is returned from the "AbstractExperiment::GetTypes" application requests the save folder and (on success) saves into the corresponding files file headers via "AbstractExperiment:: SaveDcDataHeader" or "AbstractExperiment::SaveAcDataHeader".

Next, the "InstrumentOperator::StartExperiment" method executed. It sends the vector of "ExperimentNode_t" structures to the instrument and a command to start an experiment.

After experiment started the data responses are sent from the instrument. For each data point the "AbstractExperiment::PushNewDcData" or "AbstractExperiment::PushNewAcData" called.

Finally, the "AbstractExperiment::SaveDcData" or "AbstractExperiment:: SaveAcData" called for those data point that should be saved into the file.

8 What is a custom experiment

Unlike prebuild experiments that are defined by the developer custom experiments can be defined by the user. Custom experiment consists of primitive building blocks – "builder elements". Structure of each builder element is quite similar to the structure of prebuilt experiment. Builder element is a dynamically linked library that is loaded on a startup.

Each builder element has its own settings. The main task of the builder element is to generate a vector of "ExperimentNode_t" structures based on settings.

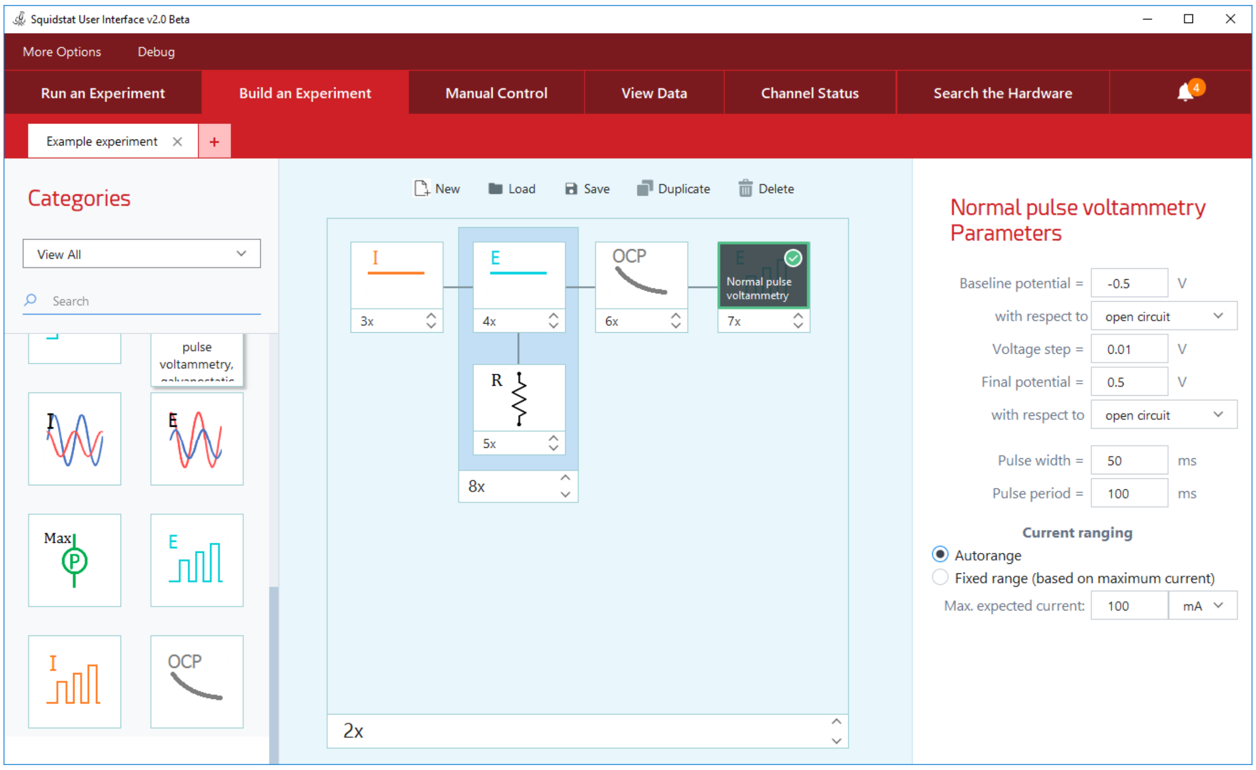
To create a custom experiment user must combine builder elements in a structure (via GUI, see p. 15), set parameters values for each element and save the experiment. On saving application will generate a text file in the user home directory in JSON format (see details below) with the description of the experiment structure and element parameters.

The lifecycle of the custom experiment is the same as of the regular one (see p. 6). Custom experiment implemented as a "CustomExperimentRunner" class that is inherited from the "AbstractExperiment" and works based on JSON files.

8.1 Custom experiment structure

Custom experiment is a nested collection (see fig. 5) of nodes with maximum depth of 3. Internally there are two types of nodes: "set" and "element". Top-level node always has type "set".

Every node has required parameter "repeats" that has default value of "1".



Baseline potential = -0.5 V
with respect to open circuit
Voltage step = 0.01 V
Final potential = 0.5 V
with respect to open circuit
Pulse width = 50 ms
Pulse period = 100 ms
Current ranging
[✓ Autorange]
Fixed range (based on maximum current)
Max. expected current: 100 mA

Figure 5 – View of an experiment builder screen

Type "set" means that current node contains an array of nodes. Type "element" means that current node represents a builder elements and contains its settings with values defined by the user.

8.2 Custom experiment file format

Every custom experiment is a text file that must:

– have the "***.json**" extension;
– be placed at the "***%UserDir%_SquidStat\custom\***" directory;
– be written using ***JSON***[4] syntax.

The table 1 contains the fields description of the top-level JSON-object.

Table 1 – Top-level fields

Field name	Type	Description
name	String	The ***experiment name*** that will be displayed at the list on the left side of the "Run an Experiment" tab and at the detailed description region (the center of the "Run an Experiment" tab).
uuid	String	The ***unique identifier*** of the current experiment.
elements	Object	Top-level node of the experiment. Must be of the type "set".

The table 2 contains the fields description of the node JSON-object.

Table 2 – Node object fields

Field name	Type	Description
repeats	Double	Represents the ***count of repeats*** of the object that will be sent to the hardware. Will be interpreted as integer.
type	String	***Type*** of the object. Allowed values: ***"set"*** and ***"element"***. If type is "set" – the "elements" filed expected. Otherwise – the "plugin-name" and the "user-input" fields expected.
elements	Array	Will be processed ***ONLY*** if type is "set". Contains the array of the node objects.
plugin-name	Object	Will be processed ***ONLY*** if type is "elements". Contains the name of the builder element that is returned from the "AbstractBuilderElement:: GetFullName".
user-input	Object	Will be processed ***ONLY*** if type is "elements".

[4] JavaScript Object Notation (JSON) syntax description: https://goo.gl/s36B16

Field name	Type	Description
		Contains the list of parameters and their values inputted by user. Particular set of parameters is defined by the each builder element and generated by the "AbstractBuilderElement::CreateUserInput".

Below there is an example of the experiment text file content.
It describes the experiment that is shown on the figure 5.

```
{
    "name": "Example experiment",
    "uuid": "{f490e739-065e-4fbe-8974-be59ad62f789}",
    "elements": {
        "repeats": 2,
        "type": "set",
        "elements": [
            {
                "plugin-name": "Constant Current",
                "repeats": 3,
                "type": "element",
                "user-input": {
                    "Maximum-voltage": 5,
                    "Minimum-voltage": 0,
                    "capacity": 10,
                    "constant-current": 10,
                    "constant-current-units": "mA",
                    "duration": 60,
                    "duration-units": "s",
                    "sampling-interval": 0.5,
                    "sampling-interval-units": "s"
                }
            },
            {
                "repeats": 8,
                "type": "set",
                "elements": [
                    {
                        "plugin-name": "Constant Potential",
                        "repeats": 4,
                        "type": "element",
                        "user-input": {
                            "constant-potential": 0.5,
                            "duration": 60,
                            "duration-units": "s",
                            "potential-vs-ocp": "reference",
                            "sampling-interval": 0.10000000000000001
                        }
                    },
                    {
                        "plugin-name": "Constant resistance discharge",
                        "repeats": 5,
                        "type": "element",
                        "user-input": {
                            "maximum-time": 60,
                            "minimum-voltage": 0,
                            "sampling-interval": 1,
                            "target-resistance": 1000,
                            "target-resistance-units": "Ohms",
                            "time-units": "s"
                        }
                    }
                ]
            },
            {
                "plugin-name": "Open Circuit",
                "repeats": 6,
                "type": "element",
                "user-input": {
                    "dvdt-minimum": 0,
                    "experiment-duration": 60,
                    "experiment-duration-units": "s",
                    "maximum-voltage": 12,
                    "minimum-voltage": -12,
                    "sampling-interval": 1,
                    "sampling-interval-units": "s"
                }
            },
            {
                "plugin-name": "Normal pulse voltammetry",
                "repeats": 7,
                "type": "element",
                "user-input": {
                    "Autorange-mode": "Autorange",
                    "final-voltage": 0.5,
                    "final-voltage-vs-ocp": "open circuit",
                    "max-current": 100,
                    "max-current-units": "mA",
                    "pulse-period": 100,
                    "pulse-width": 50,
                    "start-voltage": -0.5,
                    "start-voltage-vs-ocp": "open circuit",
                    "voltage-step": 0.01
                }
            }
        ]
    }
}
```

9 Workflow of a custom experiment

The list of custom experiments is loaded on startup and updates on saving or deleting custom experiment on the "Build an Experiment" tab. All custom experiments are placed to the bottom of the list of prebuilt experiments on the "Run an Experiment" tab.

As long as any custom experiment is inherited from the "AbstractExperiment" interface the workflow is quite the same as described in the paragraph 7.

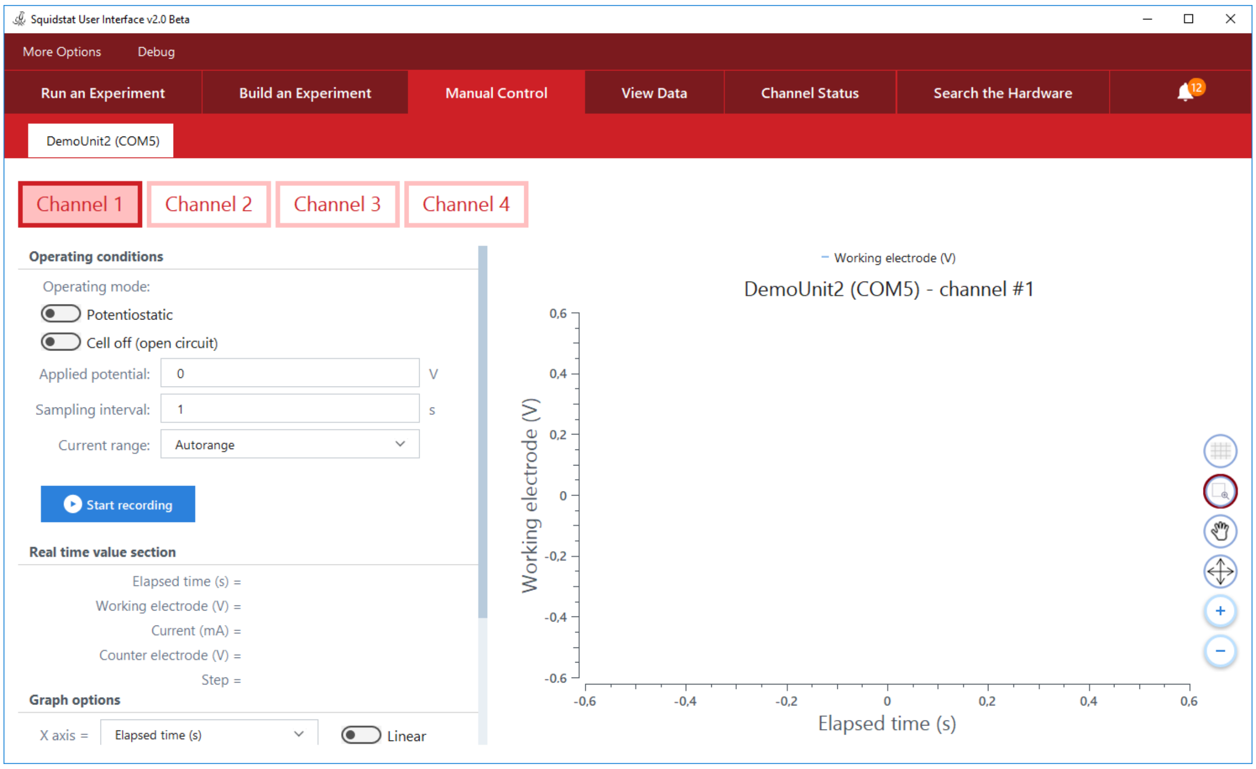
10 What is a manual experiment

There is an ability for the user to run instrument in a "Manual Control" mode. On the one hand, it is quite similar to other experiments but there is a main difference: to run the manual experiment there is no need to generate the vector of the "ExperimentNode_t" structures and send it to the instrument. Also, to start such experiment there is a separate command: "SET_MANUAL_MODE".

On the other hand, the way of delivery of the experiment data is the same as in the other types of experiment.

To operate with the manual experiments the is a separate tab "Manual Control" (see fig. 6). Every sub-tab represents every connected instrument and every button on sub-tab represents a channel of the corresponding instrument.

There is a "Operating condition" section that allows to send "MANUAL_SAMPLING_PARAMS_SET", "MANUAL_SAMPLING_PARAMS_SET", "MANUAL_POT_SETPOINT_SET", "MANUAL_OCP_SET" and "MANUAL_CURRENT_RANGING_MODE_SET" directly to the instrument (instead of vector of the "ExperimentNode_t" structures).



Applied potential: 0 V

Sampling interval: 1 s

Current range: Autorange

X axis = Elapsed time (s)

Figure 6 – Manual Control tab view

Manual experiment is implemented as the "ManualExperimentRunner" class that is inherited from the "AbstractExperiment". Manual experiments are always DC experiments. So, meaningful methods are the following: "ManualExperimentRunner:: SaveDcDataHeader", "ManualExperimentRunner::SaveDcData" and "ManualExperimentRunner::PushNewDcData".

11 Workflow of a manual experiment

When user clicks the "Start recording" button the "InstrumentOperator::StartManualExperiment" method executed.

After experiment started the data responses are sent from the instrument. For each data point the "AbstractExperiment::PushNewDcData" called.

Finally, the "AbstractExperiment::SaveDcData" called for those data point that should be saved into the file.

When user changes values of the inputs at the "Operating condition" section the "MainWindow::SetManualSamplingParams", "MainWindow:: SetManualGalvanoSetpoint", "MainWindow::SetManualPotentioSetpoint", "MainWindow::SetManualOcp" and "MainWindow::SetCurrentRangingMode" are correspondently executed.

12 Experiment plugin creation

Every prebuilt experiment on the "Run an Experiment" tab is loaded dynamically based on the specific dynamically loaded library (DLL).

Every DLL must be a Qt Plugin[5] and provide the "***ExperimentFactoryInterface***" interface.

12.1 ExperimentFactoryInterface

The main interface exported from the plugin.
Developer needs to implement the pure virtual method:

```
virtual AbstractExperiment* CreateExperiment(const QVariant&) = 0;
```

Implementation must create an instance of an "***AbstractExperiment***".
Instance must be created on the heap.
Singletons are NOT allowed.
Also, developer doesn't need to manage created object any more.
Example:

```
class ExampleExperiment: public AbstractExperiment {
    …
};

class Factory: public QObject, public ExperimentFactoryInterface {
    …
};

AbstractExperiment* Factory::CreateExperiment(const QVariant&) {
    return new ExampleExperiment;
}
```

There is an ability to pass a parameter to the Factory (type of the QVariant). This parameter is optional and typically developer does not need to pass it.

12.2 AbstractExperiment

The interface for the Experiments objects.
Developer needs to implement the following pure virtual methods:

```
virtual QString GetShortName() const = 0;
virtual QString GetFullName() const = 0;
virtual QString GetDescription() const = 0;
virtual QStringList GetCategory() const = 0;
virtual QPixmap GetImage() const = 0;
virtual QWidget* CreateUserInput() const = 0;
virtual QByteArray GetNodesData(QWidget*) const = 0;
```

[5] How to Create Qt Plugins: http://doc.qt.io/qt-5/plugins-howto.html

Table 3 – Description of the AbstractExperiment interface

Method	**Return type**	**Description**
GetShortName	QString	Returns the ***experiment name*** that will be displayed at the list on the left side of the "Run an Experiment" tab (see fig. 7).
GetFullName	QString	Returns the ***experiment name*** that will be displayed at the detailed description region (the center of the "Run an Experiment" tab, see fig. 8).
GetDescription	QString	Returns the ***experiment description*** that will be displayed at the detailed description region (the center of the "Run an Experiment" tab, see fig. 8). ***Rich text*** formatting allowed.
GetCategory	QStringList	Returns the ***category*** of the experiment. Unique categories will be displayed above the list on the left side of the "Run an Experiment" tab (see fig. 7).
GetImage	QPixmap	Returns the ***image*** that will be displayed at the detailed description region (the center of the "Run an Experiment" tab, see fig. 8).
CreateUserInput	QWidget*	Returns the ***widget***, that contains all user inputs.
GetNodesData	QByteArray	Returns the ***data*** that will be sent to the instrument (array of the ExperimentNode_t).

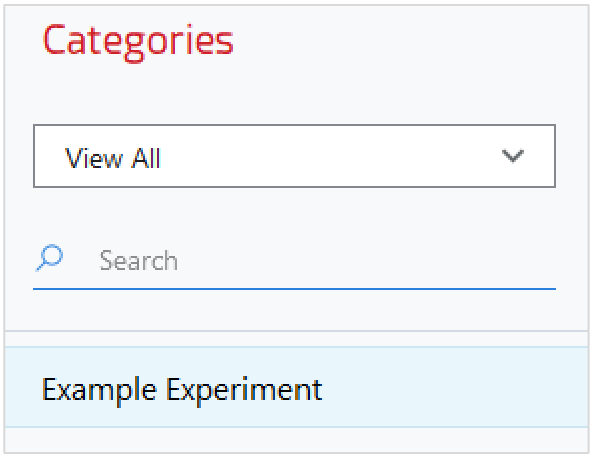
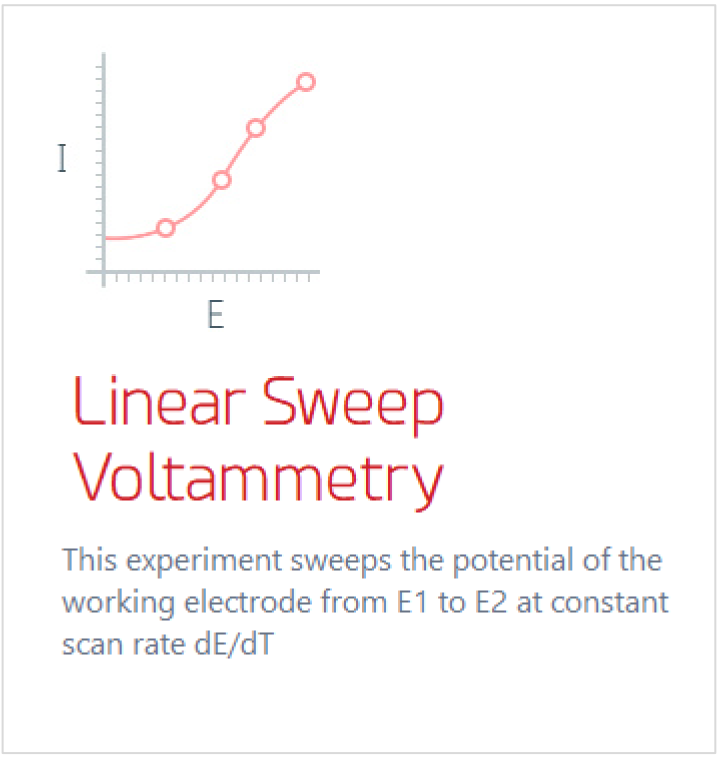


Figure 7 – List of the prebuilt experiments



Linear Sweep Voltammetry

This experiment sweeps the potential of the working electrode from E1 to E2 at constant scan rate dE/dT

Figure 8 – Detailed experiment description

12.3 GetShortName method

Example:

```
QString ExampleExperiment::GetShortName() const {
    return "Example Experiment";
}
```

12.4 GetFullName method

Example:

```
QString ExampleExperiment::GetFullName() const {
    return "Linear Sweep Voltammetry";
}
```

12.5 GetDescription method

Example:

```
QString ExampleExperiment::GetDescription() const {
    return "This experiment sweeps the <b>potential</b>";
}
```

12.6 GetCategory method

Example:

```
QString ExampleExperiment::GetCategory() const {
    return "Example Category";
}
```

12.7 GetImage method

Example:

```
QPixmap ExampleExperiment::GetImage() const {
    return QPixmap(":/GUI/Resources/experiment.png");
}
```

Image path may be specified either as local relative path or Qt resource path (as in the example above).

12.8 CreateUserInput method

To facilitate the creation of the user inputs there are some implemented macros. To use them developer needs to include the following:

```
#include <ExperimentUIHelper.h>
```

There are two required macros that must be used:

```
QWidget* ExampleExperiment::CreateUserInput() const {
    USER_INPUT_START("top-widget-unique-id");
    …
    USER_INPUT_END();
}
```

The input parameter of the "***USER_INPUT_START***" macro is a string value. It is needed for checking if the correct widget passed to the "***GetNodesData***" method.

Other macros allow to place following widgets:

– text label (right and left aligned);
– text input;
– drop-down;
– radio button.

All widgets are placed at the grid layout so all of them have input parameters "row" and "column". All inputs have additional text id parameter to find specific widget when reading data.

To place text labels developer needs to use the following macros:

```
_INSERT_RIGHT_ALIGN_COMMENT("Label text", row, column);
_INSERT_LEFT_ALIGN_COMMENT("Label text", row, column);
```

To place the text input developer needs to use the following macro (first parameter is a default value):

```
_INSERT_TEXT_INPUT("0", "start-voltage-id", row, column);
```

To place the drop-down developer needs to use the following macros:

```
_START_DROP_DOWN("drop-down-id", row, column);
    _ADD_DROP_DOWN_ITEM("Item 1");
    _ADD_DROP_DOWN_ITEM("Item 2");
    _ADD_DROP_DOWN_ITEM("Item 3");
_END_DROP_DOWN();
```

There are two ways for placing radio button:

- each button is placed at the separate cell;
- all buttons of the same group are placed at the one cell (horizontally).

To place radio buttons at the separate cells developer needs to use the following macros:

```
_START_RADIO_BUTTON_GROUP("radio-button-group-id");
    _INSERT_RADIO_BUTTON("Radio 1", row, column);
    _INSERT_RADIO_BUTTON("Radio 2", row, column);
_END_RADIO_BUTTON_GROUP();
```

To place the group of radio buttons at the single cell developer needs to use the following macros:

```
_START_RADIO_BUTTON_GROUP_HORIZONTAL_LAYOUT("radio-button-group-id", row, col);
    _INSERT_RADIO_BUTTON_LAYOUT("Radio 1");
    _INSERT_RADIO_BUTTON_LAYOUT("Radio 2");
_END_RADIO_BUTTON_GROUP_LAYOUT();
```

Also, there is an ability to set stretches for specific row or column. To do this developer needs to use the following macros:

```
_SET_ROW_STRETCH(row, 1);
_SET_COL_STRETCH(column, 1);
```

12.9 GetNodesData method

To facilitate the reading from the user inputs and combining data there are some implemented macros. To use them developer needs to include the following:

```
#include <ExperimentUIHelper.h>
```

There are two required macros that must be used:

```
QByteArray ExampleExperiment::GetNodesData(QWidget *wdg) const {
    NODES_DATA_START(wdg, "top-widget-unique-id");
    …
    NODES_DATA_END();
}
```

The input parameters of the "***NODES_DATA_START***" macro are the pointer of the widget that passed to the method and a string value. String is needed for checking if the correct widget passed to the method.

There is the following object declared in the macro:

```
ExperimentNode_t exp;
```

So, to add an "***ExperimentNode_t***" to the data that will be send to the instrument developer needs to fill corresponding parameters of the "***exp***" object and call the following macro:

```
PUSH_NEW_NODE_DATA();
```

To read the data that was inputted to the text edit developer needs to call the following macro:

```
qint32 var;
GET_TEXT_INPUT_VALUE(var, "text-input-id");
```

To read the text of the selected radio button developer needs to call the following macro:

```
QString var;
GET_SELECTED_RADIO(var, "radio-button-id");
```

To read the selected text of the drop-down developer needs to call the following macro:

```
QString var;
GET_SELECTED_DROP_DOWN(var, "drop-down-id");
```

13 Builder element plugin creation

Every builder element on the "Build an Experiment" tab is loaded dynamically based on the specific DLL.

Every DLL must be a Qt Plugin and provide the "***BuilderElementFactoryInterface***" interface.

13.1 BuilderElementFactoryInterface

The main interface exported from the plugin.
Developer needs to implement the pure virtual method:

```
virtual AbstractBuilderElement* CreateElement(const QVariant&) = 0;
```

Implementation must create an instance of an "***AbstractBuilderElement***".
Instance must be created on the heap.
Singletons are NOT allowed.
Also, developer doesn't need to manage created object any more.
Example:

```
class ExampleElement: public AbstractBuilderElement {
    …
};

class Factory: public QObject, public BuilderElementFactoryInterface {
    …
};

AbstractBuilderElement* Factory::CreateElement(const QVariant&) {
    return new ExampleElement;
}
```

There is an ability to pass a parameter to the Factory (type of the QVariant). This parameter is optional and typically developer does not need to pass it.

13.2 AbstractBuilderElement

The interface for the Builder elements objects.
Developer needs to implement the following pure virtual methods:

```
virtual QString GetFullName() const = 0;
virtual QStringList GetCategory() const = 0;
virtual QPixmap GetImage() const = 0;
virtual ExperimentType GetType() const = 0;
virtual QWidget* CreateUserInput(UserInput&) const = 0;
virtual NodesData GetNodesData(const UserInput&,
                               const CalibrationData&,
                               const HardwareVersion&) const = 0;
```

Table 4 – Description of the AbstractBuilderElement interface

Method	**Return type**	**Description**
GetFullName	QString	Returns the ***name*** that will be displayed on mouse hovering and selection (see fig. 9 and 10).
GetCategory	QStringList	Returns the ***category*** of the element. Unique categories will be displayed above the list on the left side of the "Build an Experiment" tab (see fig. 11).
GetImage	QPixmap	Returns the ***image*** that will be displayed at the element widget (see fig. 9 and 10).
GetType	ExperimentType	Returns the ***type*** of the nodes, that will be returned from the GetNodesData.
CreateUserInput	QWidget*	Returns the ***widget***, that contains all user inputs.
GetNodesData	NodesData	Returns the ***data*** that will be sent to the instrument (array of the ExperimentNode_t).

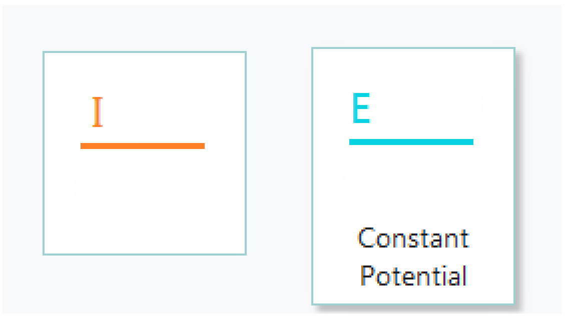


Figure 9 – Builder elements at the collection list (right one is mouse hovered)

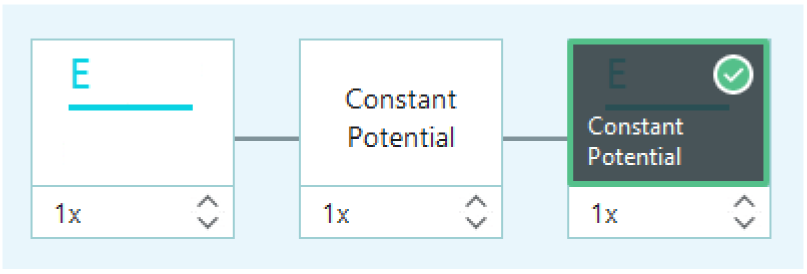


Figure 10 – Builder elements at the experiment creating area (central is mouse hovered, right one is selected)

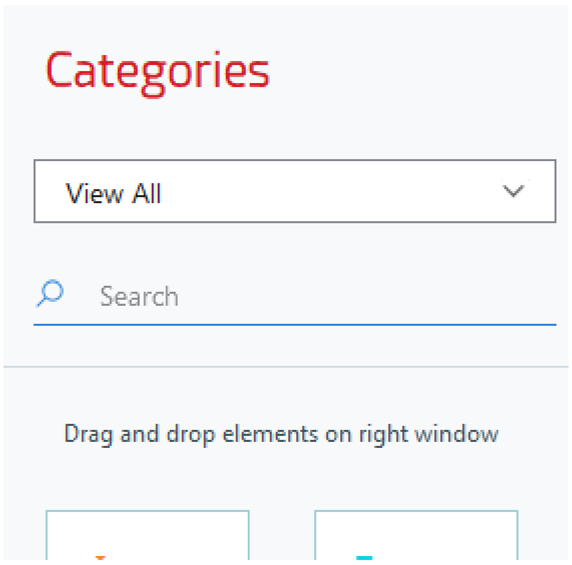


Figure 11 – Builder elements categories

13.3 GetFullName method

Example:

```
QString ExampleElement::GetFullName() const {
    return "Example Element";
}
```

13.4 GetCategory method

Example:

```
QStringList ExampleElement::GetCategory() const {
    return QStringList() <<
        "Example Category" <<
        "Example Category 2";
}
```

13.5 GetImage method

Example:

```
QPixmap ExampleElement::GetImage() const {
    return QPixmap(":/GUI/ExampleElement");
}
```

Image path may be specified either as local relative path or Qt resource path (as in the example above).

13.6 GetType method

Example:

```
ExperimentType ExampleElement::GetType() const {
    return ET_DC;
}
```

13.7 CreateUserInput method

To facilitate the creation of the user inputs there are some implemented macros. They are the same as for the "AbstractExperiment::CreateUserInput" (see p. 12.8).

13.8 GetNodesData method

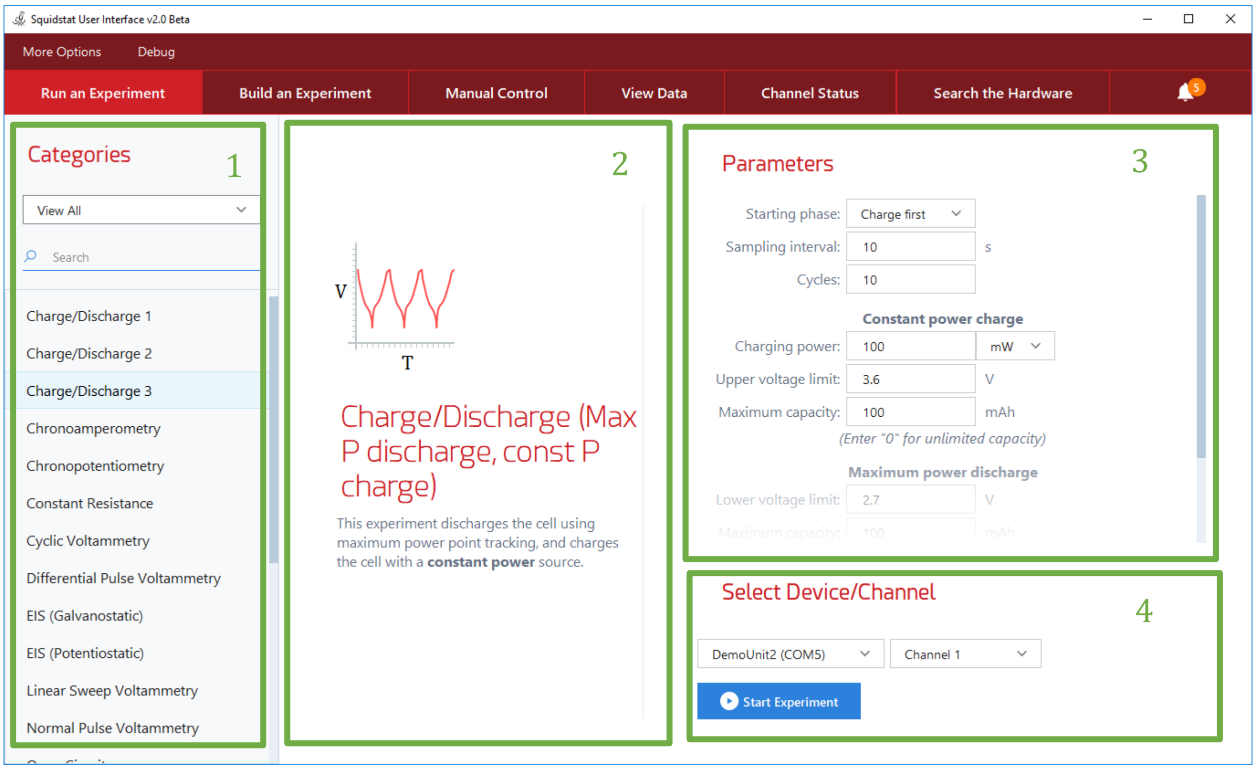
To facilitate the reading from the user inputs and combining data there are some implemented macros. They are the same as for the "AbstractExperiment::GetNodesData" (see p. 12.9).

14 How does the "Run an Experiment" tab work

The "Run an Experiment" tab is created through the executing the "MainWindowUI::GetRunExperimentTab" method.

There are four major parts of the tab (see fig. 12):

1. Experiment list.
2. Experiment description.
3. Experiment parameters.
4. Instrument selection.



Charge/Discharge (Max P discharge, const P charge)

This experiment discharges the cell using maximum power point tracking, and charges the cell with a **constant power** source.

Parameters

Starting phase: Charge first
Sampling interval: 10 s
Cycles: 10

Constant power charge
Charging power: 100 mW
Upper voltage limit: 3.6 V
Maximum capacity: 100 mAh
(Enter "0" for unlimited capacity)

Maximum power discharge
Lower voltage limit: 2.7 V
Maximum capacity: 100 mAh

DemoUnit2 (COM5)
Channel 1

Figure 12 – View of the "Run an Experiment" tab

14.1 Experiment list

Experiment list is formed by the handlers for "MainWindow:: PrebuiltExperimentsFound", "MainWindow::AddNewCustomExperiments" and "MainWindow::RemoveCustomExperiment" signals.

As the data for "Qt::DisplayRole" the output of the "AbstractExperiment:: GetShortName" is taken. As the data for "Qt::UserRole" the pointer of the corresponding "AbstractExperiment" is taken.

The list itself is filtered by the proxy model[6] "ExperimentFilterModel" inherited from the "QSortFilterProxyModel". As the data for the filtering used the contents of the "Search" line edit and the "Category" combo box.

[6] Model/View Programming – http://doc.qt.io/qt-5/model-view-programming.html

14.2 Experiment description

Experiment description is filled by the handler for "QItemSelectionModel:: currentChanged" signal of the selection model of the experiment list.

The image, the name and the description of the experiment are taken from the outputs of "AbstractExperiment::GetImage", "AbstractExperiment::GetFullName" and "AbstractExperiment::GetDescription" correspondently. The pointer of the "AbstractExperiment" is taken from the "Qt::UserRole" data of the selected item.

14.3 Experiment parameters

Experiment parameters region consists of two widgets: the output of the "AbstractExperiment::CreateUserInput" and the overlay widget that is transparent for mouse events.

The purpose of the overlay widget is to create blurring at the bottom of the parameters scroll area.

14.4 Instrument selection

The list of available instruments is edited by the handlers of the "MainWindow::AddNewInstruments" and "MainWindow:: RemoveDisconnectedInstruments" signals. As the data for the "Qt::UserRole" of the list is the amount of the channels taken.

On the "QComboBox::currentTextChanged" signal of the instrument list the contents of the channel list is updated based on the "Qt::UserRole" data value of the selected instrument.

Also, on instrument or channel selection changing ("QComboBox:: currentTextChanged" signals) the "MainWindow::SelectHardware" and "MainWindow::UpdateCurrentExperimentState" are called.

The "MainWindow::SelectHardware" is used internally by the "MainWindow" to handle last selected by the user instrument and channel pair[7].

The "MainWindow::UpdateCurrentExperimentState" method is used to emit corresponding to the current hardware state signals. They are the following: "MainWindow::CurrentHardwareBusy", "MainWindow::CurrentExperimentPaused", "MainWindow::CurrentExperimentResumed", "MainWindow:: CurrentExperimentIsNotManual" and "MainWindow:: CurrentExperimentCompleted".

Pressing "Start Experiment" forces creation of the new sub-tab on the "View Data" tab (see p. 17) and leads to the running selected experiment (via the "MainWindow:: StartExperiment").

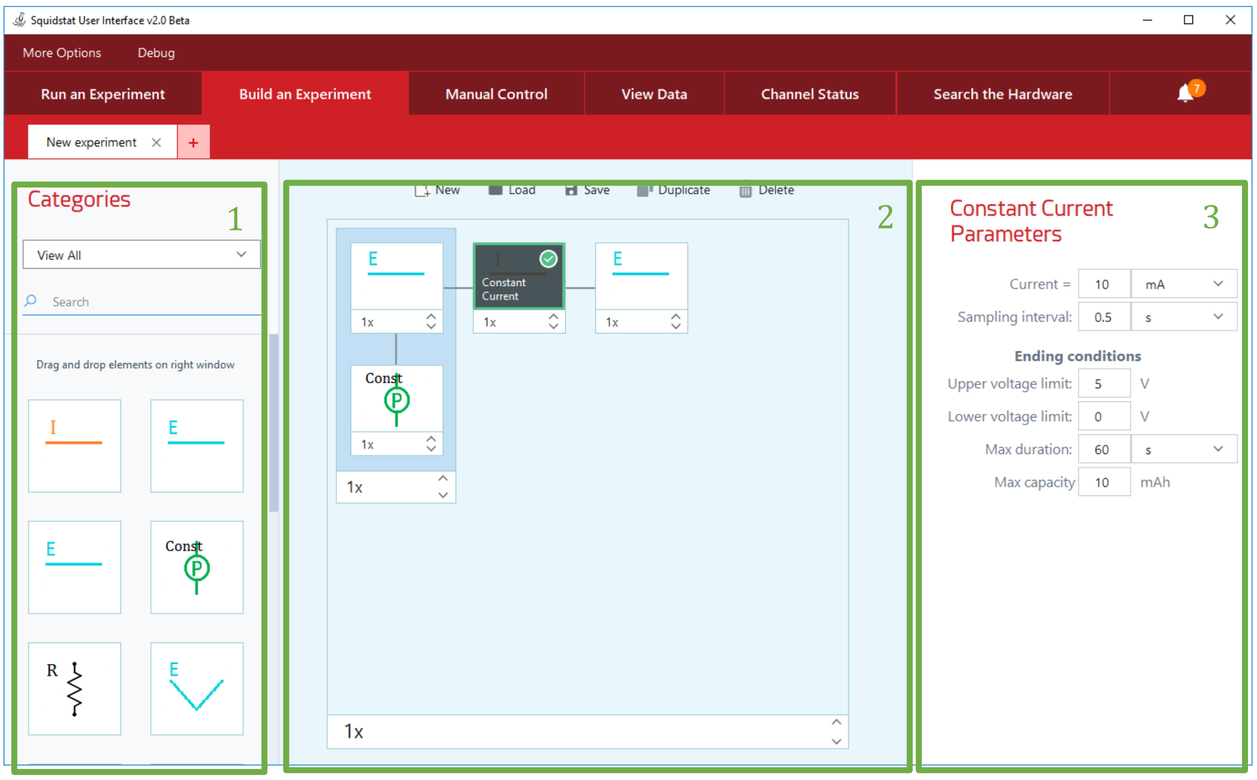
[7] Important – As far as instrument and channel may be selected from the different places (right now from the "Run an Experiment" and "Manual Control" tabs) "MainWindow::SelectHardware" and "MainWindow::UpdateCurrentExperimentState" methods are called on every "Run an Experiment" tab selection and on every experiment selection to update internal state to the visible one.

15 How does the "Build an Experiment" tab work

The "Build an Experiment" tab is created through the executing the "MainWindowUI::GetBuildExperimentTab" method.

There are three major parts of the tab (see fig. 13):

1. Builder element list.
2. Builder area.
3. Builder element parameters.



Current = 10 mA
Sampling interval: 0.5 s

Ending conditions

Upper voltage limit: 5 V
Lower voltage limit: 0 V
Max duration: 60 s
Max capacity 10 mAh

Figure 13 – View of the "Build an Experiment" tab

Sub tab bar at the top switches widgets inside "QStackedLayout" on the region № 2 (builder area). Areas № 1 and № 2 are always the same.

15.1 Builder element list

Builder elements list area is created through the executing the "MainWindowUI::CreateElementsListWidget" method.

Element list is formed by the handler for "MainWindow:: BuilderElementsFound" signal. Each element in the list is a "QLabel" placed on a "QGridLayout". As an image for "QLabel" the "AbstractBuilderElement::GetImage" is taken.

When data values at "Category" combo box or "Search" line edit changed the "QGridLayout" is cleared and filled by elements that are fit entered data.

When mouse hovered the "QLabel" the overlay widget is created (see fig. 14). It consists of two "QLabel" with image (from the "AbstractBuilderElement:: GetImage") and text (from "AbstractBuilderElement::GetFullName").

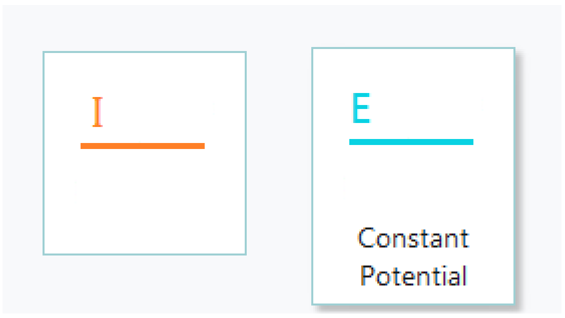


Figure 14 – Regular (left) and mouse hovered (right) builder elements

15.2 Builder area

Builder area is a "QFrame" with a "QGridLayout" on it. There are two types of widgets that can be placed on it: element (created by the "BuilderWidget:: CreateBuildExpElementWidget") and element container (created by the "BuilderWidget::CreateBuildExpContainerWidget").

When element or container is selected the half-transparent overlay widget is created (see fig. 15).

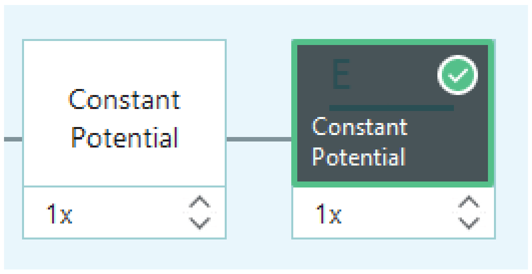


Figure 15 – Mouse hovered (left) and selected (right) builder element

To handle drag and drop elements on the area the following mechanism is implemented. On the parent widget the is a hidden image that always has the same size as the widget (see fig. 16). There are colored areas for each available for dropping region. When "QDrag" is executed only non-black areas are available for dropping.

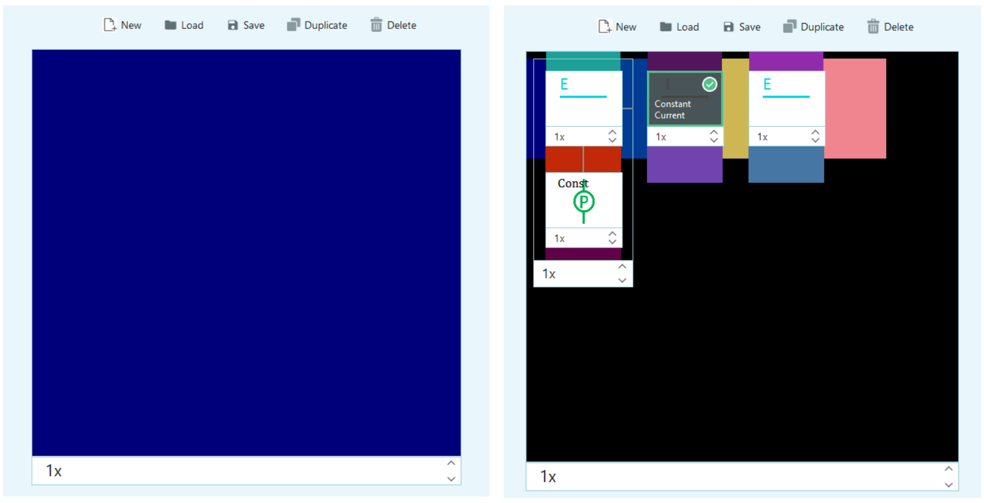


Figure 16 – Internal background map for drag and drop

15.3 Builder element parameters

The builder element parameters area is quite similar to the experiment parameters area on the "Run an Experiment" tab (see p. 14.3). It also has an overlay widget for blurring when scrolling available.

16 How does the "Manual Control" tab work

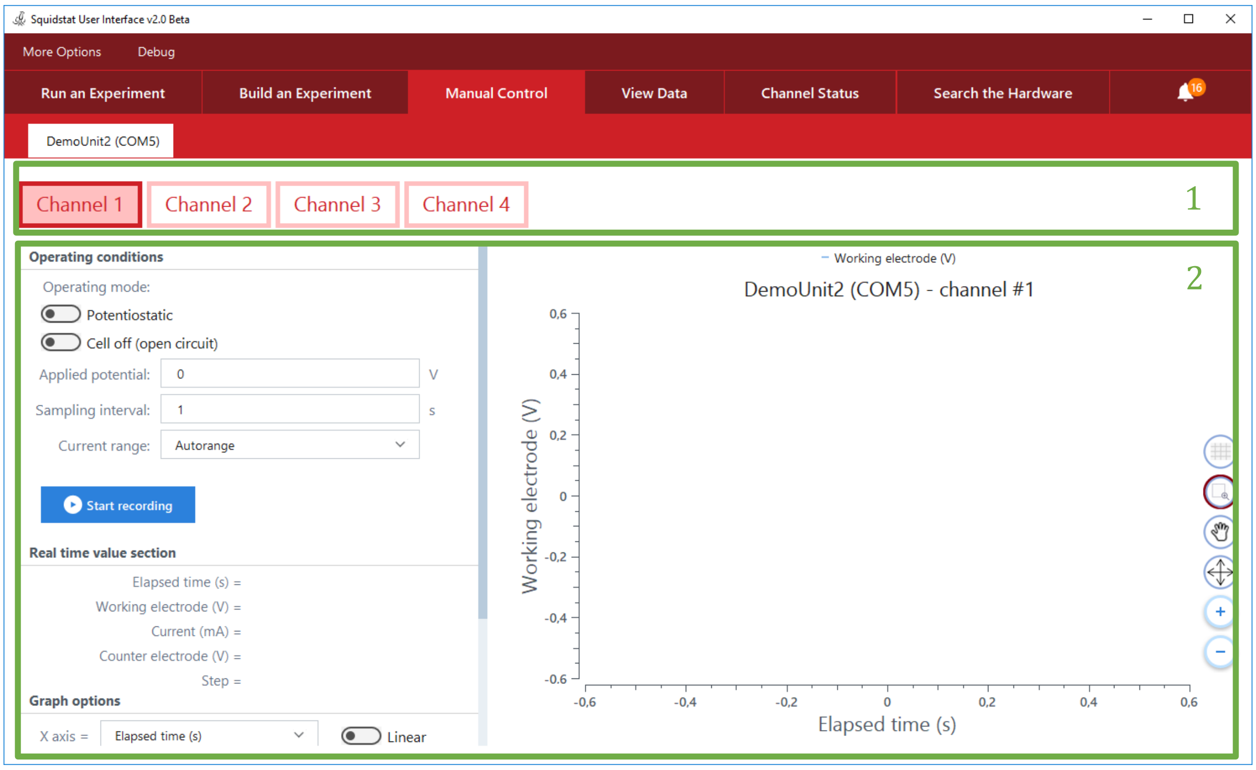
The "Manual Control" tab is created through the executing the "MainWindowUI::GetManualControlTab" method.

Sub-tabs represent each connected instrument. They are created and deleted by handlers for "MainWindow::AddNewInstruments" and "MainWindow:: RemoveDisconnectedInstruments" correspondently.

Each sub-tab owns the widget on a "QStackedLayout" (region № 2). This widget also has a "QStackedLayout" on it with data plot widgets (see p. 17) for each channel.

On the region № 1 there are "MAX_CHANNEL_VALUE" buttons. When sub-tab is activated last $(MAX\_CHANNEL\_VALUE\ - actual\ channel\ amount)$ buttons are disabled.

Switching sub-tabs and channel buttons leads to switching widgets inside both "QStackedLayout" and calling "MainWindow::SelectHardware".



Operating conditions

Operating mode:

Potentiostatic

Cell off (open circuit)

Applied potential: 0 V

Sampling interval: 1 s

Current range: Autorange

Real time value section

Elapsed time (s) =

Working electrode (V) =

Current (mA) =

Counter electrode (V) =

Step =

Graph options

X axis = Elapsed time (s)

Figure 17 – View of the "Manual Control" tab

17 How does the "View Data" tab work

The "View Data" tab is created through the executing the "MainWindowUI:: GetNewDataWindowTab" method.

There are four major parts of the tab (see fig. 18):

1. Sub-tabs bar.
2. Buttons layout.
3. Sub-windows area.
4. Data plot widget.

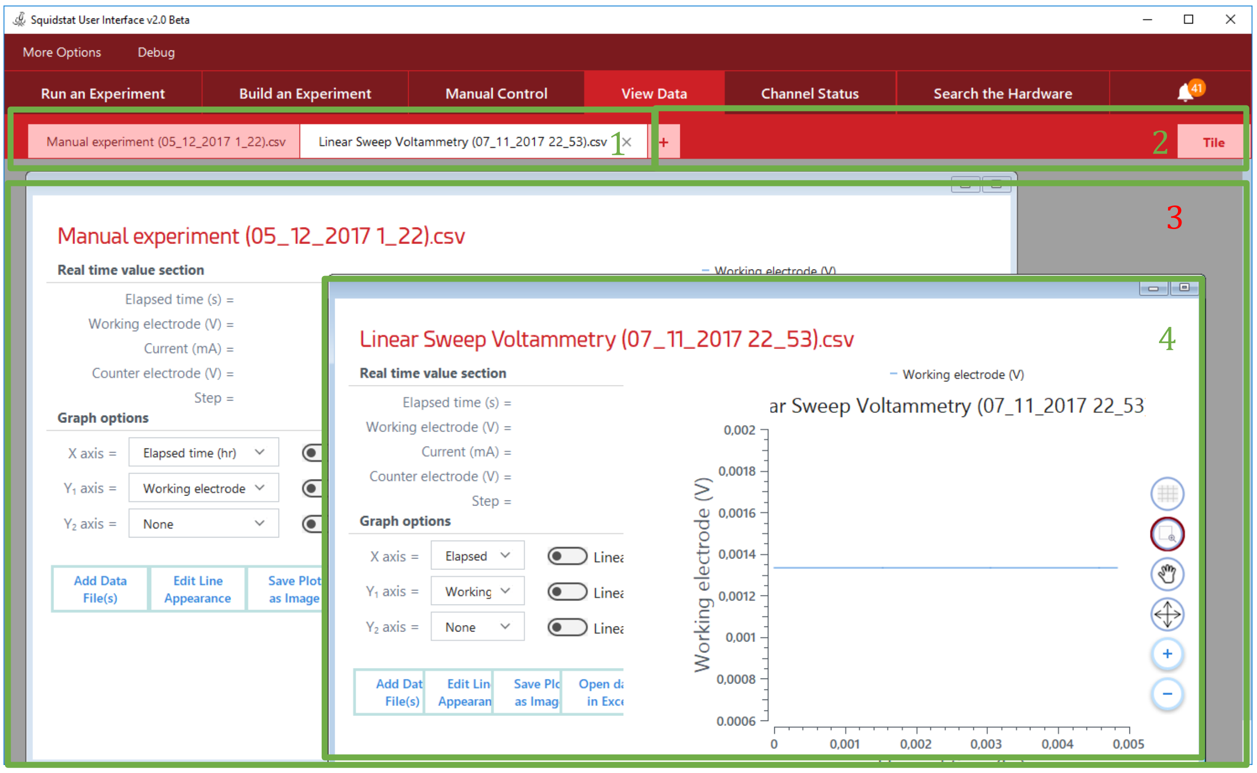


Figure 18 – View of the "View Data" tab

On the "View Data" tab at the same time can be opened data plot widgets with data loaded from the file, data from already ended experiment and data from the active experiment. Each data set is displayed as a sub-tab (region № 1).

Region № 2 is a "QHBoxLayout" with buttons and a stretch between them.

Region № 3 is a "QMdiArea" that allows to display multiple sub-windows inside the main one.

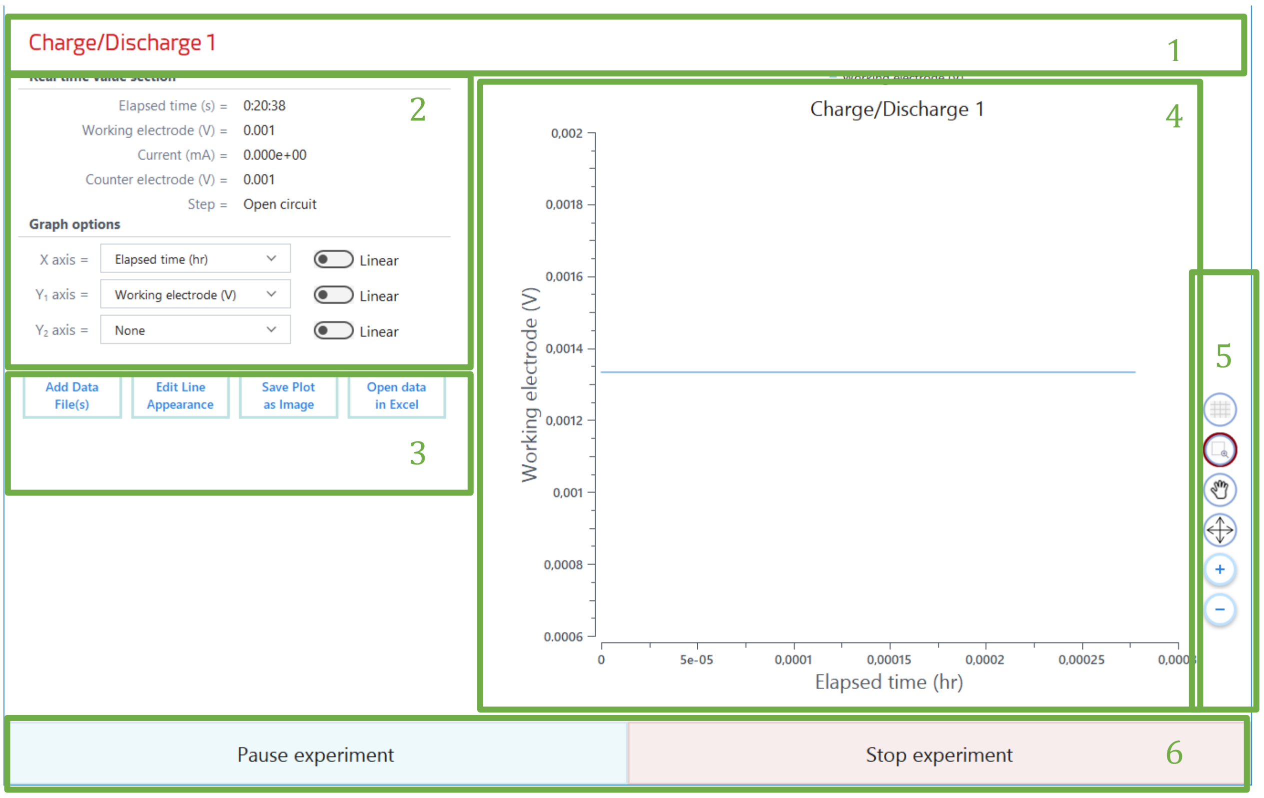
17.1 Data plot widget

The data plot widget can be created through executing the "MainWindowUI:: CreateNewDataTabWidget" method.

The data plot widget consists of the following major parts (see fig. 19):

1. The header ("QLabel").
2. Parameters and info region (set of "QGroupBox" combined by the vertical layout – "QVBoxLayout").
3. Control buttons region.

4. Plot area ("QwtPlot").
5. Plot control buttons.
6. Experiment control buttons.



Charge/Discharge 1

Elapsed time (s) = 0:20:38
Working electrode (V) = 0.001
Current (mA) = 0.000e+00
Counter electrode (V) = 0.001
Step = Open circuit

Graph options

X axis = Elapsed time (hr) — Linear
Y_1 axis = Working electrode (V) — Linear
Y_2 axis = None — Linear

Figure 19 – Structure of the data plot widget

To handle "drag and move" feature on the plot, "zoom to selection" feature (see fig. 20) and the "nearest point highlighting" (see fig. 20) there is a transparent overlay widget above the "QwtPlot".

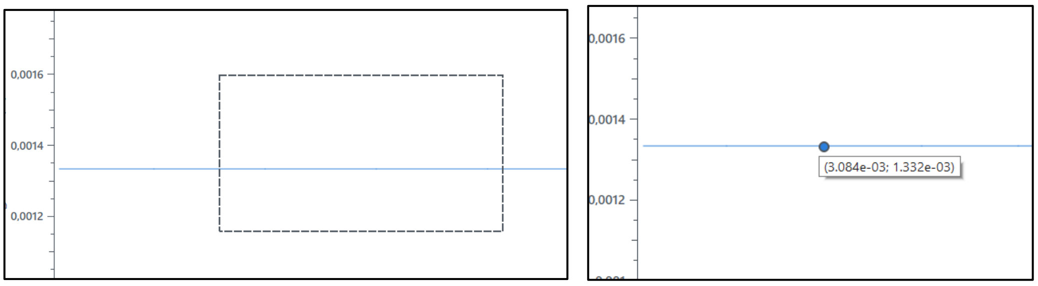
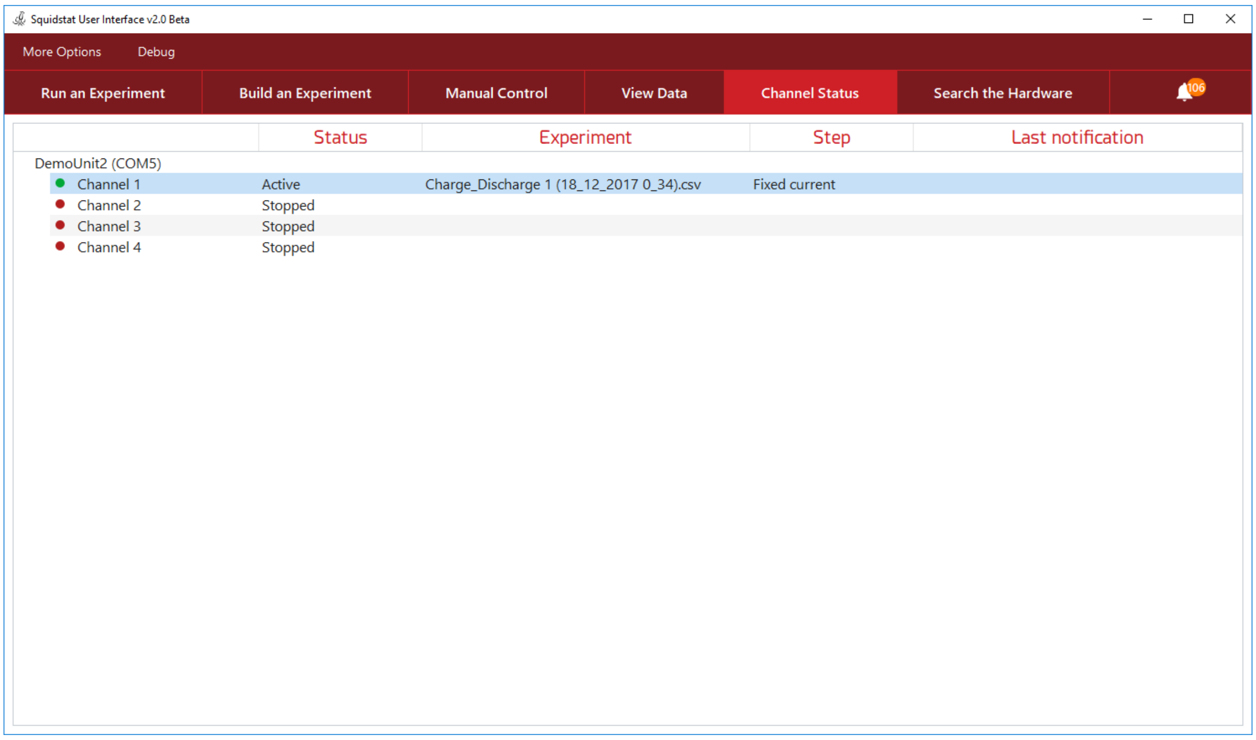


Figure 20 – Zoom to selection (left) and nearest data point highlighting (right)

18 How does the "Channel Status" tab work

The "Channel Status" tab (see fig. 21) is created through the executing the "MainWindowUI::GetChannelStatusTab" method.



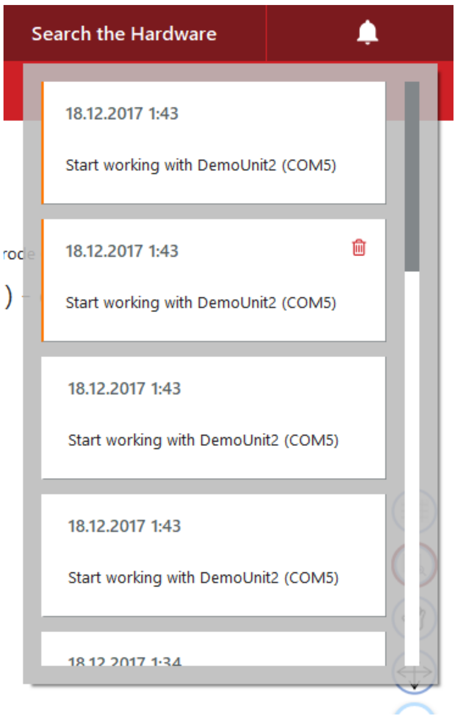
	Status	Experiment	Step	Last notification
DemoUnit2 (COM5)				
● Channel 1	Active	Charge_Discharge 1 (18_12_2017 0_34).csv	Fixed current	
● Channel 2	Stopped			
● Channel 3	Stopped			
● Channel 4	Stopped			

Figure 21 – View of the "Channel Status" tab

The tab contains the "QTreeView" widget with disabled collapse feature. First-level items represent connected instruments, the second-level – channels.

19 How does the notification area work

Notification area (see fig. 22) is a modal "QDialog" that is created throw the execution of the "MainWindowUI::ShowNotificationDialog" method.



18.12.2017 1:43

Start working with DemoUnit2 (COM5)

18.12.2017 1:43

Start working with DemoUnit2 (COM5)

18.12.2017 1:43

Start working with DemoUnit2 (COM5)

18.12.2017 1:43

Start working with DemoUnit2 (COM5)

18.12.2017 1:34

Figure 22 – View of the notification area

Dialog contains "QScrollArea" with the vertical layout inside ("QVBoxLayout"). Each message is a widget with several "QLabel" and a "QPushButton" (visible on mouse hovering).

19.1 Log message processing

The main idea in the log message processing is the installing special handler ("LogSignalEmitter") for native Qt's debug messages. Installation is performed at the "main" method.

```
    qInstallMessageHandler(LogMessageHandler);
```

This message handler redirects all debug messages to the signal of the "LogSignalEmitter" to let any object inside application perform a subscription.

So, after message handler was installed every message that was passed though the "QDebug" object will be also passed through the "LogSignalEmitter:: SendLogEmitter" signal of the singleton object (access through the "GetLogSignalEmitter" method, "Log.h").

20 How does the firmware updater work

Firmware updater (see fig. 23) is a modal "QDialog" that is created throw the execution of the "MainWindowUI::GetUpdateFirmwareDialog" method.

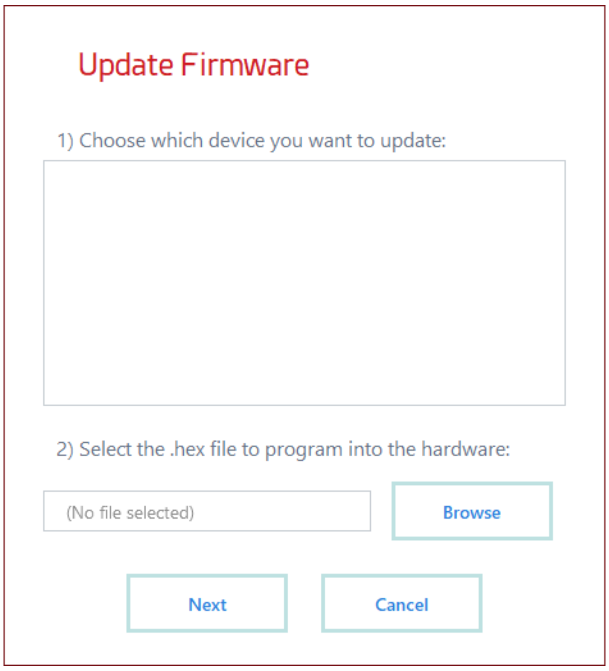

Update Firmware
1) Choose which device you want to update:
2) Select the .hex file to program into the hardware:
(No file selected)
Browse
Next
Cancel

Figure 23 – View of the "Update Firmware" dialog

Right now, updater works just with instruments that have hardware version "HardwareModel_t::PLUS_2_0" and later. On dialog loading all instruments that have unsupported hardware version skipped and not displayed on the list.

21 How to extend the firmware updater

To extend updater functionallity developer must do the following:

1. Add modular ability to update firmware in different ways for instruments with different hardware versions.
2. Add new modules for updating firmware.

All work will be done inside two handlers of the "MainWindowUI:: GetUpdateFirmwareDialog" method:

```
    dialogConn << QObject::connect(m_mainWindow,
                                   &MainWindow::CurrentHardwareList,
                                   [=](const InstrumentList &_list) {
        ...
    });

    ...

    dialogConn << QObject::connect(nextBut, &QPushButton::clicked, [=]() {
        ...
    });
```

Inside the first one developer need to update filter that skips unsupported instruments.

```
    for (int i = 0; i < list.size();) {
        if (list.at(i).hwVer.hwModel < PLUS_2_0) {
            list.removeAt(i);
        }
        else {
            ++i;
        }
    }
```

Then to save information about hardware version of the each passed instrument. The simplest way to do it is to add hardware version to the item model with data role "Qt::UserRole", like the following:

```
    for (auto it = list.begin(); it != list.end(); ++it) {
        auto *item = new QStandardItem(it->name);
        item->setData(it->hwVer.hwModel, Qt::UserRole);
        model->setItem(row++, item);
    }
```

Inside the second handler developer needs to add switch that will make a decision what method to execute depending on hardware version.

22 QSS tips and hints

There are some undocumented and unobvious tricks that developer need to know about Qt Style Sheets.

22.1 Setting border and background to enable box model

Sometimes widgets have some strange behavior that leads to ignoring stylesheet. To prevent this use the following rule: always set "border" and "background" parameters for the widget that you are going to style. Even if it will be "none" value.

22.2 QWidget vs QFrame

The "QWidget" itself have no full support of style sheets. For example, if you place a layout on the "QWidget", fill this layout with other widgets and then try to set margins or paddings for the first one they both will not work.

For this purpose use "QFrame" instead. It has full support of the box model[8].

22.3 QListView for QComboBox

When you'd like to style the pop-up of the "QComboBox" you can encounter some problems that are look as your style does not apply.

To prevent this developer need to create explicitly "QListView" and set it into the "QComboBox".

```
auto someComboBox = CMB();
someComboBox->setView(OBJ_NAME(new QListView, "combo-list"));
```

After this you can style "#combo-list" as a regular widget with no problems.

[8] Customizing Qt Widgets Using Style Sheets – http://doc.qt.io/qt-5/stylesheet-customizing.html