|  |
| --- |
| Squidstat GUI  Programmers guide |
| 2017 |

Table of contents

[1 Introduction 3](#_Toc500106839)

[2 General description of the structure 4](#_Toc500106840)

[3 Serial communicator 6](#_Toc500106841)

[4 Instrument operator 7](#_Toc500106842)

[5 Instrument enumerator 8](#_Toc500106843)

[6 What is an experiment 9](#_Toc500106844)

[7 Workflow of a regular experiment 10](#_Toc500106845)

[8 What is a custom experiment 12](#_Toc500106846)

[9 Workflow of a custom experiment 13](#_Toc500106847)

[10 What is a manual experiment 14](#_Toc500106848)

[11 Workflow of a manual experiment 15](#_Toc500106849)

[12 Experiment plugin creation 16](#_Toc500106850)

[13 Builder element plugin creation 22](#_Toc500106851)

[14 How does the "Run an Experiment" tab work – [optional] 23](#_Toc500106852)

[15 How does the "Build an Experiment" tab work – [optional] 24](#_Toc500106853)

[16 How does the "Manual Control" tab work – [optional] 25](#_Toc500106854)

[17 How does the "View Data" tab work – [optional] 26](#_Toc500106855)

[18 How does the "Channel Status" tab work – [optional] 27](#_Toc500106856)

[19 How does the notification area work – [optional] 28](#_Toc500106857)

[20 How does the firmware updater work – [optional] 29](#_Toc500106858)

[21 How to extend the firmware updater – [optional] 30](#_Toc500106859)

[22 QSS tips and hints – [optional] 31](#_Toc500106860)

[1 Estimation summary: 32](#_Toc500106861)

# Introduction

This document contains a whole description of the SquidStat GUI software.

Developers can use this guide to extend the software functionallity.

# 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.

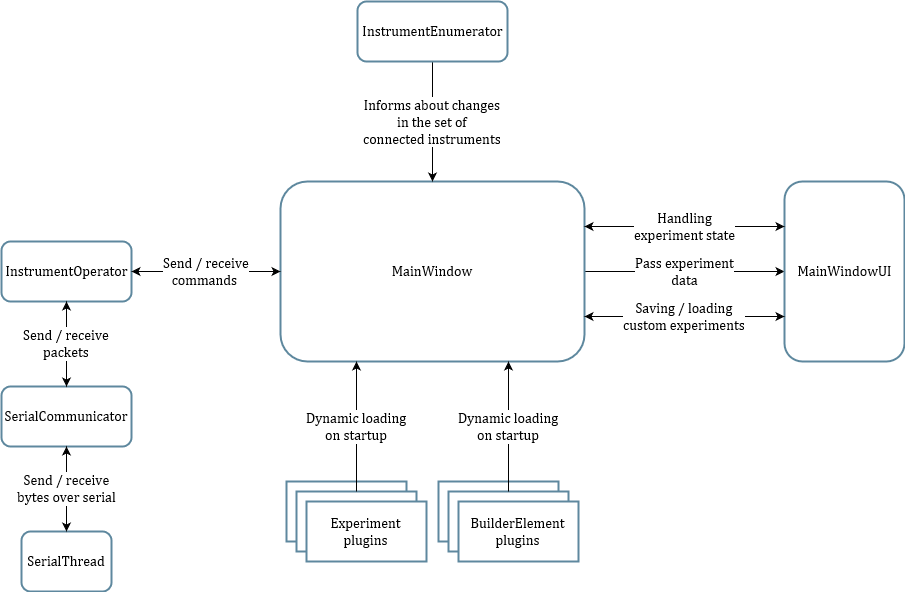


Figure 1 – General scheme of the application

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 .

# Serial communicator

For the PC every instrument is the COM-port. Every instrument supports the specific protocol[[1]](#footnote-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]](#footnote-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.

# 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 parameter types.

The "InstrumentOperator" owns the "SerialCommunicator".

# 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.

# 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]](#footnote-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.

# 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).

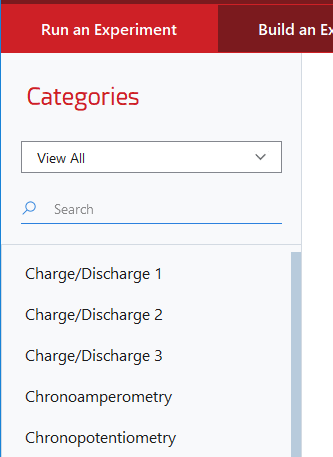


Figure 2 – List of loaded experiments

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).

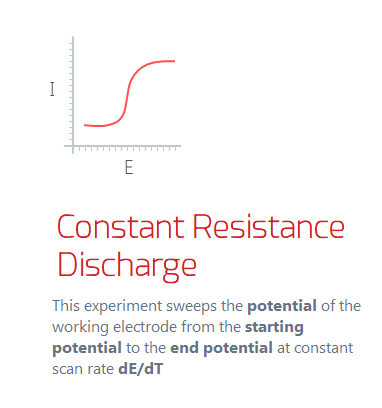


Figure 3 – Detailed experiment description

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).

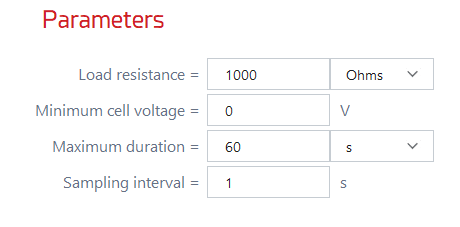


Figure 4 – UI for parameters setting of a specific experiment

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.

# What is a custom experiment

0.5 h

# Workflow of a custom experiment

0.5 h

# What is a manual experiment

0.5 h

# Workflow of a manual experiment

0.5 h

# 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[[4]](#footnote-4) and provide the "ExperimentFactoryInterface" interface.

## ExperimentFactoryInterface

The main interface exported from the plugin.

Developer needs to implement the pure virtual method:

1. 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:

1. class ExampleExperiment: public AbstractExperiment {
2. …
3. };
4. class Factory: public QObject, public ExperimentFactoryInterface {
5. …
6. };
7. AbstractExperiment\* Factory::CreateExperiment(const QVariant&) {
8. return new ExampleExperiment;
9. }

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.

## AbstractExperiment

The interface for the Experiments objects.

Developer needs to implement the following pure virtual methods:

1. virtual QString GetShortName() const = 0;
2. virtual QString GetFullName() const = 0;
3. virtual QString GetDescription() const = 0;
4. virtual QString GetCategory() const = 0;
5. virtual QPixmap GetImage() const = 0;
6. virtual QWidget\* CreateUserInput() const = 0;
7. virtual QByteArray GetNodesData(QWidget\*) const = 0;

Table 1 – 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. 5). |
| 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. 6). |
| 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. 6).  Rich text formatting allowed. |
| GetCategory | QString | 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. 5). |
| GetImage | QPixmap | Returns the image that will be displayed at the detailed description region (the center of the "Run an Experiment" tab, see fig. 6). |
| 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). |

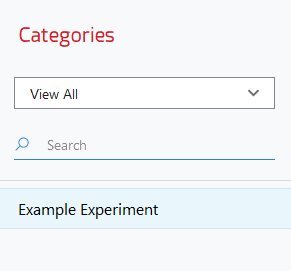


Figure 5 – List of the prebuilt experiments

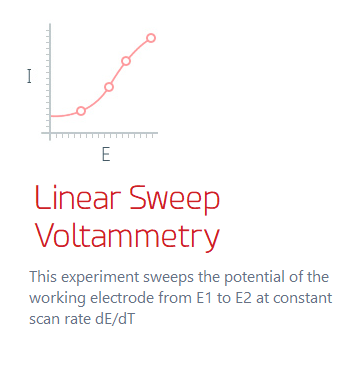


Figure 6 – Detailed experiment description

## GetShortName method

Example:

1. QString ExampleExperiment::GetShortName() const {
2. return "Example Experiment";
3. }

## GetFullName method

Example:

1. QString ExampleExperiment::GetFullName() const {
2. return "Linear Sweep Voltammetry";
3. }

## GetDescription method

Example:

1. QString ExampleExperiment::GetDescription() const {
2. return "This experiment sweeps the <b>potential</b>";
3. }

## GetCategory method

Example:

1. QString ExampleExperiment::GetCategory() const {
2. return "Example Category";
3. }

## GetImage method

Example:

1. QPixmap ExampleExperiment::GetImage() const {
2. return QPixmap(":/GUI/Resources/experiment.png");
3. }

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

## CreateUserInput method

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

1. #include <ExperimentUIHelper.h>

There are two required macros that must be used:

1. QWidget\* ExampleExperiment::CreateUserInput() const {
2. USER\_INPUT\_START("top-widget-unique-id");
3. …
4. USER\_INPUT\_END();
5. }

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:

1. \_INSERT\_RIGHT\_ALIGN\_COMMENT("Label text", row, column);
2. \_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):

1. \_INSERT\_TEXT\_INPUT("0", "start-voltage-id", row, column);

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

1. \_START\_DROP\_DOWN("drop-down-id", row, column);
2. \_ADD\_DROP\_DOWN\_ITEM("Item 1");
3. \_ADD\_DROP\_DOWN\_ITEM("Item 2");
4. \_ADD\_DROP\_DOWN\_ITEM("Item 3");
5. \_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:

1. \_START\_RADIO\_BUTTON\_GROUP("radio-button-group-id");
2. \_INSERT\_RADIO\_BUTTON("Radio 1", row, column);
3. \_INSERT\_RADIO\_BUTTON("Radio 2", row, column);
4. \_END\_RADIO\_BUTTON\_GROUP();

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

1. \_START\_RADIO\_BUTTON\_GROUP\_HORIZONTAL\_LAYOUT("radio-button-group-id", row, col);
2. \_INSERT\_RADIO\_BUTTON\_LAYOUT("Radio 1");
3. \_INSERT\_RADIO\_BUTTON\_LAYOUT("Radio 2");
4. \_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:

1. \_SET\_ROW\_STRETCH(row, 1);
2. \_SET\_COL\_STRETCH(column, 1);

## 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:

1. #include <ExperimentUIHelper.h>

There are two required macros that must be used:

1. QByteArray ExampleExperiment::GetNodesData(QWidget \*wdg) const {
2. NODES\_DATA\_START(wdg, "top-widget-unique-id");
3. …
4. NODES\_DATA\_END();
5. }

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:

1. 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:

1. PUSH\_NEW\_NODE\_DATA();

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

1. qint32 var;
2. GET\_TEXT\_INPUT\_VALUE(var, "text-input-id");

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

1. QString var;
2. GET\_SELECTED\_RADIO(var, "radio-button-id");

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

1. QString var;
2. GET\_SELECTED\_DROP\_DOWN(var, "drop-down-id");

# Builder element plugin creation

Describing the process of the builder element plugin creation.

1.5 h

# How does the "Run an Experiment" tab work – [optional]

Widget structure, widget interaction logic.

1.5 h

# How does the "Build an Experiment" tab work – [optional]

Widget structure, widget interaction logic.

1.5 h

# How does the "Manual Control" tab work – [optional]

Widget structure, widget interaction logic.

1 h

# How does the "View Data" tab work – [optional]

Widget structure, widget interaction logic.

2 h

# How does the "Channel Status" tab work – [optional]

Widget structure, widget interaction logic.

0.5 h

# How does the notification area work – [optional]

Widget structure, widget interaction logic.

0.5 h

# How does the firmware updater work – [optional]

Widget structure, widget interaction logic, protocol description.

1 h

# How to extend the firmware updater – [optional]

For example, what to do to add teensy utility.

0.5 h

# QSS tips and hints – [optional]

Some undocumented and unobvious tricks for Qt Style Sheets.

0.5 h

# Estimation summary:

Minimum estimation – 10.5 hours.

Optional paragraphs – 9 hours.

Total (minimum + optional) – 19.5 hours.

1. This document does not contain the description of the instrument protocol [↑](#footnote-ref-1)
2. Connection type is the fifth parameter of the "QObject::connect" method [↑](#footnote-ref-2)
3. This document does not contain the detailed description of the "ExperimentNode\_t" structure and its capabilities. [↑](#footnote-ref-3)
4. How to Create Qt Plugins: http://doc.qt.io/qt-5/plugins-howto.html [↑](#footnote-ref-4)