Squidstat API User's Guide

Notes on Distributions

API code was tested using QT 5.14.x, where we have verified it works best in. Included in the distributions of API for Windows and Linux are the necessary files from QT 5.14.2 to run the example projects. Mac users will need to manually install QT 5.14.x to run these same projects. The QT online installer can be found here.

Introduction

The Squidstat API has two primary classes with which users can interact, and several helper classes that set and get information from the primary classes. All of the classes have the prefix "Ais," for "Admiral Instruments" inside each class, not every public member is available to users, but only those with the macro `SQUIDSTAT_DLL_SHARED_EXPORT` in the signature.

The two primary classes are:

- **AisSquidstatStarter**
    - used to initiate the application loop that interacts with the instrument.
- **AisSquidstat**
    - used to interact with the application loop, and deals with event handling and data transfer.

The helper classes as well as a brief description are as follows:

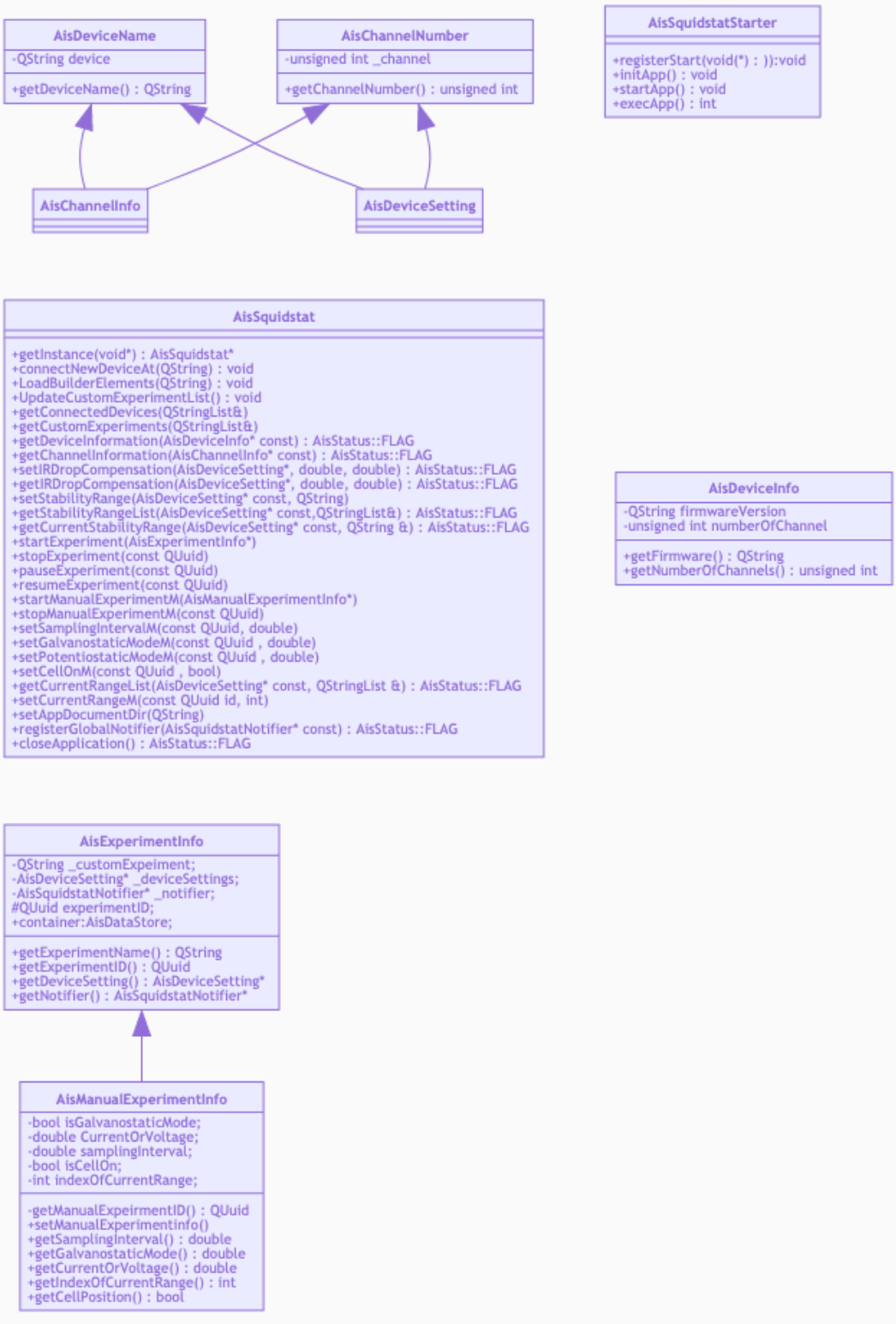
- **AisDeviceSetting**
    - a small class used to store the instrument serial name and channel number associated with an experiment.
- **AisSquidstatNotifier**
    - An abstract class whose virtual functions are used as callback functions for when important events happen during an experiment. The user must create a derived class and implement the virtual functions in order to have their callbacks fired when data arrives or when the experiment pauses, resumes, or stops.

- **AisExperimentInfo**
    - Holds pointers to an **AisDeviceSetting** object and an **AisSquidstatNotifier** object in order to pass them to the member function `startExperiment()`. It also holds the container for the experimental data.
- **AisManualExperimentInfo**
    - Holds pointers to an **AisDeviceSetting** object and an **AisSquidstatNotifier** object in order to pass them to the member function `startManualExperiment()`. It also holds the container for the manual experimental data. It is derived form **AisExperimentInfo**
- **AisDataStore**
    - Objects each hold one column's worth of data (data from a single experimental variable, e.g. current, voltage, timestamp, etc.). Several **AisDataStore** objects comprise a container for a given experiment, stored inside **AisExperimentInfo**.
- **AisDeviceInfo**
    - An object that holds information about the firmware version and number of channels of a given device.
- **AisChannelInfo**
    - An object that in future API updates will contain information about the channel status.

In order to use the Squidstat API, the user will need to include the following header files:

- AisSquidstat.h
- DataLabels.h

Class Diagram



Class: AisSquidstatStarter

Defined in **AisSquidstatStarter.h**.

Public member functions:

```
AisSquidstatStarter();
~AisSquidstatStarter();
void registerStart(void(*)() func);
void initApp();
void startApp();
int execApp();
```

AisSquidstatStarter initializes all of the Squidstat API background processes and starts the application loop that polls for events, such as data and notifications that arrive from the hardware. The important member functions (for users **not** using the QtCoreApplication class to build their application) are `initApp()` and `execApp()`. For those users who use the QtCoreApplication class, the `startApp()` function will suffice.

The **AisSquidstatStarter** class also provides the ability to register a callback app with the `registerStart()` function, which allows multi-threaded applications to coordinate asynchronously with the return of the `initApp()` function, which takes a few seconds to execute.

AisSquidstatStarter Member Functions

registerStart

```
void registerStart(void(*)() func);
```

Arguments	Returns
• Function Pointer with void return type and no arguments	• Nothing

The function `initApp()` takes a one or two seconds to execute. The function `registerStart()` can be used to register a callback function that executes at the very end of `initApp()`. This can be useful in multi- threaded applications where the user does not

want to wait for `initApp()` to return before starting other processes.

initApp

```
void initApp();
```

Arguments	Returns
• None	• Void

This function initializes important background processes that interact with the hardware. It is important to note that `initApp()` and `execApp()` must be called on the same thread. This function must be called before calling the static member function `getInstance()`, or else `getInstance()` will return a null pointer.

startApp

```
void startApp();
```

Arguments	Returns
• None	• Void

This function is for Qt application development only, and it should be called instead of `initApp()` and `execApp()`. In a Qt application, the user would first create a QtCoreApplication object and then call its `exec()` member function. In a non-Qt project, `initApp()` and `execApp()` accomplish these two tasks.

execApp

```
void execApp();
```

Arguments	Returns
• None	• Void

This function starts the control loop that polls for events related to Squidstat hardware

activity. It will handle all of the background processes that communicate with the instruments, and it also fires the callbacks in **AisSquidstatNotifier**. This function will not return until closeApplication() is called.

Class: AisSquidstat

Defined in **AisSquidstat.h**.

Public member functions:

```
~AisSquidstat();
static AisSquidstat* getInstance(void* mainWindow = nullptr);
void connectNewDeviceAt(QString comPortName = "");
void LoadBuilderElements(QString dllFilePath);
void UpdateCustomExperimentList();
AisStatus::FLAG getConnectedDevices(QStringList& connectedDevices);
AisStatus::FLAG getCustomExperiments(QStringList& customExperiments);
AisStatus::FLAG getDeviceInformation(AisDeviceInfo* const);
AisStatus::FLAG getChannelInformation(AisChannelInfo* const);
AisStatus::FLAG setIRDropCompensation(AisDeviceSetting* const,
    double UnCompenstatedResistance, double CompensatationLevel);
AisStatus::FLAG getIRDropCompensation(AisDeviceSetting* const,
    double &UnCompenstatedResistance, double &CompensatationLevel);
AisStatus::FLAG setStabilityRange(AisDeviceSetting* const, QString rangeName);
AisStatus::FLAG getStabilityRangeList(AisDeviceSetting* const, QStringList&);
AisStatus::FLAG getCurrentStabilityRange(AisDeviceSetting* const,
    QString tRangeRange);
AisStatus::FLAG startExperiment(AisExperimentInfo* experiment);
AisStatus::FLAG stopExperiment(const QUuid id);
AisStatus::FLAG pauseExperiment(const QUuid id);
AisStatus::FLAG resumeExperiment(const QUuid id);
AisStatus::FLAG startManualExperimentM(AisManualExperimentInfo* experiment);
 AisStatus::FLAG stopManualExperimentM(const QUuid id);
AisStatus::FLAG setSamplingIntervalM(const QUuid id, double seconds_S);
AisStatus::FLAG setGalvanostaticModeM(const QUuid id, double current_mA);
AisStatus::FLAG setPotentiostaticModeM(const QUuid id, double Voltage_V);
AisStatus::FLAG setCellOffM(const QUuid id, double cellPosition);
AisStatus::FLAG getCurrentRangeListM (AisDeviceSetting* const,
    QStringList &currentRageList);
AisStatus::FLAG setCurrentRangeM(const QUuid id, int indexOfCurrentRange);
AisStatus::FLAG setAppDocumentDir(QString documentDir);
AisStatus::FLAG registerGlobalNotifier(AisSquidstatNotifier* const);
void closeApplication();
```

The **AisSquidstat** class is the primary way that users will control the connected devices. From this class users can send commands to individual instruments, such as starting and stopping experiments. There should only be one **AisSquidstat** object running in a given application, and it is acquired using the static getInstance() function, and not through a constructor.

AisSquidstat Member Functions

getInstance

```
static AisSquidstat* getInstance(void* mainWindow = nullptr);
```

Arguments	Returns
• Note that a nullptr is provided as the default argument; the user should never provide any arguments to this function.	• A pointer to the AisSquidstat object that is running all of the background processes.

Note that this is a static member function. It provides a pointer to the object created by `initApp()`. This pointer gives access to the user to control connected instruments.

connectNewDeviceAt

```
void connectNewDeviceAt(QString comPortName = "");
```

Arguments	Returns
• A QString indicating the COM port at which to search for a device. If no COM port is specified, then the software will search all available COM ports for Squidstat devices.	• Void

This function initiates the search for Squidstat devices. The user can optionally specify a COM port at which to search. If this argument is omitted, then the software will poll all available COM ports. Any non-Squidstat devices at available COM ports will receive a “ping,” and the software will wait until a response has timed out before continuing on to the next device on the list. Whenever a Squidstat device is found, calibration data will automatically be downloaded if necessary, and the instrumentReadyToUse() callback will be fired.

LoadBuilderElements

```
void LoadBuilderElements(QString dllFilePath);
```

Arguments	Returns
• dllFilePath, a QString object that represents the directory path to where the Squidstat experiment “tile” library files are stored.	• Void

Here the user must provide a directory path to where they have stored the library files corresponding to the Squidstat User Interface’s “Experiment Builder tiles.” These library files are installed during a normal installation of the Squidstat User Interface in “/Admiral Instruments/Squidstat/elements”. However, a set of library files for both debug and release builds are also provided in the Squidstat API package. The path string should not terminate with a “/”. The argument is a QString object, which is the Qt library version of std::string. The directory path can be provided directly in the argument, like so:

```
auto app_handler = AisSquidstat::getInstance();
app_handler->LoadBuilderElements(
    “C:/SquidstatAPI files/SquidstatDLL/Debug/element”);
```

UpdateCustomExperimentList

```
void UpdateCustomExperimentList();
```

Arguments	Returns
• None	• Void

This function is useful for users who want to add or edit custom experiment files on-the-fly. Each custom experiment is stored as a JSON file, the text of which can be read and modified outside of the Squidstat User Interface Experiment Builder. However, the JSON files are parsed when `initApp()` is called. In order to refresh the list for new or modified files, UpdateCustomExperimentList() must be called. Note that for existing JSON files that are modified and not renamed, the API will not recognize any changes to the file unless the UUID field inside the file is changed. Users can generate a new UUID using the class QUuid, which is included in the Squidstat API. For more information on the QUuid class, see https://doc.qt.io/qt-5/quuid.html.

getConnectedDevices

```
AisStatus::FLAG getConnectedDevices(QStringList& connectedDevices);
```

Arguments	Returns
• connectedDevices, a QStringList object that is passed by reference. It should be empty upon entering the function, and filled at return.	• AisStatus::NO_ERROR

This function provides a list of serial numbers/device names of the Squidstat devices connected to the software. Note that the API can only interact with and open connections that are closed when `initApp()` is called, and so open instances of the Squidstat User Interface or other programs running the Squidstat API will interfere with device connectivity. The Squidstat API searches for newly connected devices only when the connectNewDeviceAt() function is called. For more information about the QStringList class, see https://doc.qt.io/qt-5/qstringlist.html.

getCustomExperiments

```
AisStatus::FLAG getCustomExperiments(QStringList& customExperiments);
```

Arguments	Returns
• customExperiments, a QStringList object that is passed by reference. It should be empty upon entering the function, and filled at return.	• AisStatus::NO_ERROR

This function provides a list of names of custom experiments that have been built and saved using the Squidstat User Interface Experiment Builder tab. These experiments are saved in “/Admiral Instruments/Custom Experiments”.

getDeviceInformation

```
AisStatus::FLAG getDeviceInformation(AisDeviceInfo* const);
```

Arguments	Returns
• A pointer to an AisDeviceInfo object. The object should have the device name set to the device in question before calling getDeviceInformation. Upon return, the user can check the specified device's firmware version and the number of channels.	• AisStatus::NO_ERROR if the specified device is found. • AisStatus::HANDLER_NOT_FOUND if the specified device is not connected.

This function allow the user to check the firmware version and the number of channels a given device has. The user creates an **AisDeviceInfo** object, specifying the device name/serial number in the constructor. Then the user passes the pointer to that object to getDeviceInformation(). Upon return, the user can call the **AisDeviceInfo** object's getFirmware() and getNumberOfChannels() member functions.

getChannelInformation

```
AisStatus::FLAG getChannelInformation(AisChannelInfo* const);
```

Arguments	Returns
• A pointer to an AisChannelInfo object. The object should have the device name and channel number before calling getChannelInformation.	• AisStatus::NO_ERROR if the specified device is found. • AisStatus::HANDLER_NOT_FOUND if the specified device is not connected. • AisStatus::INVALID_CHANNEL if the specified channel falls outside the valid range for the instrument.

In this release of the Squidstat API, the **AisChannelInfo** object is not yet useful to the user. In future versions users will be able to specify the desired instrument and channel and get information on the channel's status using this function, e.g. whether the instrument is idle, paused, or running an experiment.

setIRDropCompensation

```
AisStatus::FLAG setIRDropCompensation(AisDeviceSetting* const,
double UnCompensatedResistance, double CompensationLevel);
```

Arguments	Returns
• A pointer to an AisDeviceSetting object, indicating the device name and channel number. • UncompensatedResistance, a double, containing the uncompensated resistance (in Ohms) the user wishes to set. • CompensationLevel, a double, containing the compensation level (between 0 and 100, in percent) that the instrument should use.	• AisStatus::NO_ERROR if the specified device is found. • AisStatus::HANDLER_NOT_FOUND if the specified device is not connected. • AisStatus::INVALID_CHANNEL if the specified channel falls outside the valid range for the instrument.

This function allows the user to set the IR drop compensation settings, as one would in the Squidstat User Interface menu. The user selects the desired instrument and channel by creating an **AisDeviceSetting** object and passing it to this function. The user also provides the two parameters (doubles) specifying the uncompensated resistance and compensation level. For more information on IR drop compensation and compensation settings, see the Squidstat User Interface manual.

getIRDropCompensation

```
AisStatus::FLAG getIRDropCompensation(AisDeviceSetting* const,
double &UncompensatedResistance, double &CompensationLevel);
```

Arguments	Returns
• A pointer to an AisDeviceSetting object, indicating the device name and channel number. • UncompensatedResistance, a double, passed by reference. Upon return, it contains the uncompensated resistance (in Ohms) that the	• AisStatus::NO_ERROR if the specified device is found. • AisStatus::HANDLER_NOT_FOUND if the specified device is not

Arguments	Returns
channel is using. • CompensationLevel, a double, passed by reference. Upon return, it contains the compensation level (between 0 and 100 in percent) that the channel is using.	connected. • AisStatus::INVALID_CHANNEL if the specified channel falls outside the valid range for the instrument.

This function allows the user to read out the previously set IR drop compensation parameters. See “setIRDropCompensation” and the Squidstat User Interface manual for more information.

setStabilityRange

```
AisStatus::FLAG setStabilityRange(AisDeviceSetting* const, QString rangeName);
```

Arguments	Returns
• A pointer to an AisDeviceSetting object, indicating the device name and channel number. • rangeName, a QString indicating the selected stability range	• AisStatus::NO_ERROR if selection is successful. • AisStatus::HANDLER_NOT_FOUND if the specified device is not connected. • AisStatus::INVALID_CHANNEL if the specified channel falls outside the valid range for the instrument. • AisStatus::INCOMPATIBLE_MODEL if the designated device does not have stability range settings. • AisStatus::INVALID_STABILITY_RANGE if the indicated range does not exist for the specified model.

This function allows the user to set the stability range settings for a channel on a device. The name of the range must be specified by a QString passed to the function. The names of these strings can be copied from the latest version of the Squidstat User Interface under More Options. Alternatively, a list of range names valid for the given hardware model can be acquired using getStabilityRangeList().

It is highly recommended to call setStabilityRange() before starting an experiment. The hardware default stability range at power-up is usually not preferable. See the default selection used in the Squidstat User Interface for a given model. For example, for the more recent models of the Squidstat Plus, the preferred range is named “Capacitive loading (current <= 1mA)”.

For more information on stability ranges, see the Squidstat User Interface manual.

getStabilityRangeList

```
AisStatus::FLAG getStabilityRangeList(AisDeviceSetting* const, QStringList&);
```

Arguments	Returns
• A pointer to an AisDeviceSetting object, indicating the device name and channel number. • A QStringList object, passed by reference. This should be passed empty to the function, and upon return will be filled with the list of range names.	• AisStatus::NO_ERROR if selection is successful. • AisStatus::HANDLER_NOT_FOUND if the specified device is not connected. • AisStatus::INCOMPATIBLE_MODEL if the designated device does not have stability range settings.

This function allows the user to obtain a list of names of the given device’s possible stability range settings. One of these names can then be passes to setStabilityRange(). For more information about the QStringList class, see https://doc.qt.io/qt-5/qstringlist.html.

getCurrentStabilityRange

```
AisStatus::FLAG getCurrentStabilityRange(AisDeviceSetting* const,
 QString &currentRange);
```

Arguments	Returns
• A pointer to an AisDeviceSetting object, indicating the device name and channel number. • currentRange, a QString passed by reference. It should be empty when passed to the function, and upon return it contains the name of the selected stability range.	• AisStatus::NO_ERROR if there are no errors. • AisStatus::HANDLER_NOT_FOUND if the specified device is not connected. • AisStatus::INCOMPATIBLE_MODEL if the designated device does not have stability range settings.

This function enables the user to determine which stability range is currently selected for a given device and channel.

startExperiment

```
AisStatus::FLAG startExperiment(AisExperimentInfo* experiment);
```

Arguments	Returns
• A pointer to an AisExperimentInfo object, containing the device name, channel number, and experiment name.	• AisStatus::NO_ERROR if there are no errors. • AisStatus::HANDLER_NOT_FOUND if the specified device is not connected. • AisStatus::CHANNEL_BUSY if the channel is busy running another experiment. • AisStatus::EXPERIMENT_NOT_FOUND if the custom experiment is not found. • AisStatus::NODE_UPLOAD_UNSUCCESSFUL if the API encounters an error while uploading the experiment parameters to the hardware. Often cycling power to the unit or disconnecting and reconnecting the device will fix this error.

This function is called to start an experiment. The device, channel number, and experiment

name are stored inside of an **AisExperimentInfo** object passed into this function. To accomplish this, first create an **AisDeviceSetting** object, passing the instrument name and channel number into the constructor. Then create an **AisExperimentInfo** object by passing the **AisDeviceSetting** object as well as the experiment name into the constructor. A pointer to an object that implements **AisSquidstatNotifier**'s virtual functions is also required. For example:

```
mDeviceSetting = new AisDeviceSetting(_InstrumentName, _channelNum);
mExperimentInfo = new AisExperimentInfo(mDeviceSetting,
 _ExperimentName, this);
```

This code snippet is inside a member function of a class that inherits from **AisSquidstatNotifier** and implements its virtual functions, hence the keyword "this" is used to pass the object pointer to the **AisExperimentInfo** constructor.

stopExperiment

```
AisStatus::FLAG stopExperiment(const QUuid id);
```

Arguments	Returns
• Id, a QUuid object containing the experiment's associated UUID	• AisStatus::NO_ERROR if there are no errors. • AisStatus::HANDLER_NOT_FOUND if the experiment by the UUID cannot be found. • AisStatus::EXPERIMENT_NOT_RUN_ON_CHANNEL if the experiment is not running in the first place.

This function is used to prematurely end an experiment. The **AisSquidstatNotifier**::experimentStopped() is still fired when this function is called. The experiment UUID can be accessed by calling getExperimentID() from the **AisExperimentInfo** that was passed to the startExperiment() function.

setAppDocumentDir

```
AisStatus::FLAG setAppDocumentDir(QString documentDir);
```

Arguments	Returns
• documentDir, a QString specifying the path of both the “Calibration files” and the “Custom Experiments” directories	• Always returns AisStatus::NO_ERROR

This function is used to set the location of the Squidstat calibration files and custom experiment files are stored. The default location is in “/Admiral Instruments”, and this is the location that the Squidstat User Interface application uses. However, this function allows the user to specify an alternate location. Note that this function must be called before calling the connectNewDeviceAt() function because connectNewDeviceAt() reads and writes from the calibration file directory.

registerGlobalNotifier

```
AisStatus::FLAG registerGlobalNotifier(AisSquidstatNotifier* const);
```

Arguments	Returns
• A pointer to an object derived from the AisSquidstatNotifier class	• Always returns AisStatus::NO_ERROR

This function is used to register the callback functions that are called when instruments connect and disconnect from the software. The argument is a pointer to an object that implements **AisSquidstatNotifier**’s virtual functions instrumentReadyToUse() and instrumentDisconnected().

closeApplication

```
void closeApplication();
```

Arguments	Returns
• None	• Void

This function stops the execApp() loop and causes it to return. Device connections are closed, and any experiments still running are stopped, and API background processes are stopped. In Qt projects, this function will close the entire application since it calls

QCoreApplication::quit(), and therefore this function may not be appropriate to use for Qt Projects.

Class: AisDeviceSetting

Defined in **AisDeviceSetting.h**.

Public member functions:

```
AisDeviceSetting(QString deviceName, unsigned int channel);
~AisDeviceSetting();
```

Inherited member functions:

```
QString getDeviceName();
unsigned int getChannelNumber();
```

AisDeviceSetting objects are used directly or indirectly to specify an instrument and channel number when calling commands with the **AisSquidstat** class. For example, setStabilityRange() and setIRDropCompensation() pass an **AisDeviceSetting** object pointer as an argument, and startExperiment() passes an **AisExperimentInfo** object pointer, which contains an **AisDeviceSetting** object pointer as one of its members.

Note that no default constructor exists for **AisDeviceSetting**, so if you use this class as a member object inside a custom class, then you will need to either include its constructor in an initialization list or use a pointer as the member instead.

Class: AisSquidstatNotifier

Defined in **AisSquidstatNotifier.h**.

Protected functions:

```
virtual void instrumentReadyToUse(QString);
virtual void instrumentDisconnected(QString);
virtual void readDCExperimentData(QUuid);
virtual void readACExperimentData(QUuid);
virtual void experimentStopped(QUuid);
virtual void experimentPaused(QUuid);
virtual void experimentResumed(QUuid);
```

The first two functions are callbacks used for “general” events: when instruments connect and disconnect from the software. The callback instrumentReadyToUse() fires when an instrument successfully connects to the software after connectNewDeviceAt() is called. The callback instrumentDisconnected() fires whenever an instrument disconnects from the software. Both of these callbacks pass the name of the instrument as an argument. In order to use these callbacks, the **AisSquidstatNotifier** object that implements these virtual functions must be registered using registerGlobalNotifier().

The last five functions are callbacks used for events that happen during the course of an experiment. The callback readDCExperimentData() fires when DC data is sampled (voltage and current data sampled at a particular instance), and readACExperimentData fires when AC data is sampled (complex impedance and frequency data sampled during an EIS sweep). The callbacks experimentStopped(), experimentPaused(), and experimentResumed() fire when the experiment ends, when the experiment is paused, and when the experiment resumes, respectively. Each of the callbacks passes the QUuid associated with the experiment as an argument. This is useful in cases where the user has registered the same callbacks with separate experiments and needs to identify which experiment has fired the callback.

In order to use these experiment-related callbacks, **AisSquidstatNotifier** object that implements these virtual functions must be registered by creating an** AisExperimentInfo** object. This object will take a pointer to the **AisSquidstatNotifier** object in its constructor. Then a pointer to the **AisExperimentInfo** object is passed to StartExperiment().

Refer to the “Squidstat API Sample Project Documentation” manual for more explanation on how the **AisSquidstatNotifier** class and callbacks are used.

Class: AisExperimentInfo

Defined in **AisExperimentInfo.h**.

Public member functions

```
AisExperimentInfo(AisDeviceSetting* deviceSettings,QString customExperiment,
AisSquidstatNotifier* dataList);
QString getExperimentName();
QUuid getExperimentID();
AisDeviceSetting* getDeviceSetting();
AisSquidstatNotifier* getNotifier();
~AisExperimentInfo();
```

Public member objects:

```
AisDataMap container;
```

The **AisExperimentInfo** class serves two purposes: to pass the necessary info to startExperiment() and to give the user access to the experimental data, through the **AisDataMap** “container” member. AisDataMap is a typedef for QMap<QString, **AisDataStore**>. For more information, see the documentation for startExperiment() and for the **AisDataStore** class.

Each **AisExperimentInfo** object also stores a QUuid object (which holds an UUID) associated with a running experiment. This UUID is generated when the object is created. This can be useful in determining which experiment fired a callback when the same callbacks are registered to multiple experiments. The QUuid object can be accessed with the member function getExperimentID().

Note that no default constructor exists for **AisExperimentInfo**, so if you use this class as a member object inside a custom class, then you will need to either include its constructor in an initialization list or use a pointer as the member instead.

Class: AisManualExperimentInfo

Defined in **AisManualExperimentInfo.h**.

Public member functions

```
AisManualExperimentInfo(AisDeviceSetting* deviceSettings,
    AisSquidstatNotifier* notifier);
void setManualExperimentinfo(double samplingInterval = 1,
    bool isCellOn = false, int indexOfCurrentRange = 0,
    bool isGalvanostaticMode = false,double CurrentOrVoltage = 0 );
double getSamplingInterval() const;
bool getGalvanostaticMode() const;
double getCurrentOrVoltage() const;
int getIndexOfCurrentRange() const;
bool getCellPosition() const;
~AisManualExperimentInfo();
```

The **AisManualExperimentInfo** is use to start manual experiment. It is derived class of **AisExperimentInfo**.

AisManualExperimentInfo Member Functions

AisManualExperimentInfo

```
AisManualExperimentInfo(AisDeviceSetting* deviceSettings,
    AisSquidstatNotifier* notifier);
```

Arguments	Returns
`deviceSettings` assign instrument serial name and channel `notifier` assign notifier for manual experiment	• Nothing

This function is used to create the manual experiment. It takes two arguments. Pointer to a AisDeviceSetting which is to specify the instrument name and Channel number. And a pointer of AisDeviceSetting which is to specify the instrument name and Channel number. And a pointer of **AisSquidstatNotifier** is help to call back method for manual experiment.

setManualExperimentinfo

```
void setManualExperimentinfo(double samplingInterval = 1,
    bool isCellOn = false, int indexOfCurrentRange = 0,
    bool isGalvanostaticMode = false,double CurrentOrVoltage = 0 );
```

Arguments	Returns
`samplingInterval` set sampling interval `isCellOn` set the cell position isGalvanostaticMode`	• Nothing

This function helps create the experiment. It takes two arguments. Pointer to a AisDeviceSetting object, used to specify the instrument name and Channel number. And a pointer to a **AisSquidstatNotifier** object, used as a call back method.

Class: AisDataStore

Defined in **AisDataStore.h**.

Public member functions:

```
AisDataStore();
qreal getMinValue();
qreal getMaxValue();
QList<qreal> getAllDataPoints();
QStringList getAllStringDataPoints();
bool isDataListEmpty();
bool isStringDataListEmpty();
qreal firstDataPoint();
qreal lastDataPoint();
QString firstStringData();
QString lastStringData();
int numberOfDataPoints();
void removeAllDataPoints();
```

The **AisExperimentInfo** member function “container” is an **AisDataMap**, which is a typedef for QMap<QString, **AisDataStore**>. Each **AisDataStore** object inside an **AisDataMap** holds a given column of data, stored as a key-value pair. The list of keys is given in DataLabels.h, which is organized into three categories: “DC data keys,” “AC data keys,” and “Common keys.”

Each **AisDataStore** object contains a list of data, either of text or of doubles. The first type of data is accessed by the DC data keys. This is data sampled in the time domain, during a constant potential or constant current interval, a current or potential sweep, a current or

potential pulse train, a constant power or constant resistance interval, or an open circuit interval. The second type of data is accessed by the AC data keys. This is data sampled in the frequency domain, during EIS sweeps. AC and DC data are expressed as qreal's (a typedef for double). All DC data lists in a given experiment will share the same length, as will all AC data lists.

The third data category of data, accessed by the "Common keys," includes information about the experiment phase: "Step name" and "Step number." Each data point is expressed as a QString. Instead of one data point generated per AC sample or DC sample, in "Common keys" category there is one QString generated at the beginning of every experiment "substep." It is not always intuitive to the user how many substeps comprise an experiment; therefore, it is not straightforward to correlate the correct step name and number with a given AC or DC data point. The recommended strategy is to look up the latest step name and number in the **AisDataStore** list whenever an AC or DC data point arrives. For example:

```
void SquidstatAppHandler::readDCExperimentData(QUuid id) {
    qreal time = mExperimentInfo->container[DCDATA_ELAPSED_TIME_S]
        .lastDataPoint();
    qreal WE = mExperimentInfo->container[DCDATA_WORKING_ELECTRODE]
        .lastDataPoint();
    qreal current = mExperimentInfo->container[DCDATA_CURRENT]
        .lastDataPoint();
    QString ExperimentSubstepName = mExperimentInfo
        ->container[CURRENT_NODE_NAME].lastStringData();
    QString text = ExperimentSubstepName +": ";
    text += QString::number(time) + "(s), ";
    text += QString::number(WE) + "(V), ";
    text += QString::number(current) + "(mA)\n";
    cout << text.toStdString();
}
```

For more information on how to access experimental data from the **AisExperimentInfo** "container" member object, refer to the "Squidstat API Sample Project Documentation."

Here is a brief description of each of the public member functions:

getMinValue

```
void getMinValue();
```

Arguments	Returns
• None	• qreal, the furthest left data point on the ordered real number line on the condition that the **AisDataStore** contains only qreal (double) data.

getMaxValue

```
void getMaxValue()
```

Arguments	Returns
• None	• qreal, the furthest right data point on the ordered real number line on the condition that the **AisDataStore** contains only qreal (double) data.

getAllDataPoints

```
QList<qreal> getAllDataPoints();
```

Arguments	Returns
• None	• QList, all of the data points in a list, on the condition that **AisDataStore** contains only qreal (double) data.

getAllStringDataPoints

```
QStringList getAllStringDataPoints();
```

Arguments	Returns
• None	• QStringList of text data on the condition that **AisDataStore** contains QString data. For more information about the QStringList class, see https://doc.qt.io/qt- 5/qstringlist.html.

Misc Functions

These functions are provided for convenience and do as they say on the tin.

- bool isDataListEmpty() returns true if there is no qreal (double) data in the list.
- bool isStringDataListEmpty() returns true if there is no text data in the list.
- firstDataPoint() returns the first data point in the qreal (double) list.
- qreal lastDataPoint() returns the last data point in the qreal (double) list.
- QString firstStringData() returns the first data point in the QString list.
- QString lastStringData() returns the last data point in the QString list.
- int numberOfDataPoints() returns the length of the data list.
- void removeAllDataPoints() clears all data stored in the list.

Class: AisDeviceInfo

Defined in **AisDeviceInfo.h**.

Public member functions:

```
AisDeviceInfo(const QString deviceName);
~AisDeviceInfo();
QString getFirmware();
unsigned int getNumberOfChannels();
```

Inherited member functions:

```
QString getDeviceName();
```

The **AisDeviceInfo** class is used to get the firmware information and number of channels for a specified device. Create an **AisDeviceInfo** object by passing the device name to the constructor. Then pass a pointer to the **AisDeviceInfo** object to **AisSquidstat**'s member function getDeviceInformation(). When the function returns you can use getFirmware() to read the firmware version (stored as a QString) and getNumberOfChannels() to determine the number of channels the device has.

Note that no default constructor exists for **AisDeviceInfo**, so if you use this class as a member object inside a custom class, then you will need to either include its constructor in an initialization list or use a pointer as the member instead.

Class: AisChannelInfo

Defined in **AisChannelInfo.h**.

Public member functions:

```
AisChannelInfo(const QString deviceName,unsigned int channel = 0);
~AisChannelInfo();
```

Inherited members

```
QString getDeviceName();
unsigned int getChannelNumber();
```

The **AisChannelInfo** class will be used in future Squidsat API releases to get the status of a specified channel on a device. As of this release, however, **AisChannelInfo** does not contain any useful information. This class will be used in the following way: create an **AisChannelInfo** object by passing the device name and channel number to the constructor. Then pass a pointer to this object to **AisSquidstat**'s member function getChannelInformation(). When the function returns it will contain useful information about the channel's status, accessible through getter functions that have yet to be implemented.

Example Project: Single Threaded Manual Experiment

This project's full source code can be found in the ManualExperimentDemo folder. For Mac and Linux, we suggest using Qt Creator to build and launch the application using the instructions in the .pro file in the directory of the source code. Alternatively, qmake can be called directly on the .pro file. For Windows, a Visual Studio solution is provided.

Before running the executable, ensure the `#define` constants in **SquidStateHandler.cpp** have been given absolute paths to your appropriate documents folder (where custom experiments are kept in .json form), and where the dynamic libraries for the builder elements reside. Lastly, create a folder with the name "csv" on the desktop.

main.cpp

The point of entry to our application will be a single thread which will be running a single manual experiment.

```
#include "AppThread.h"

int main(int argc, char *argv[])
{
    // We are running a manual experiment on a single thread.
    std::thread app(Dowork);
    app.join();
    return 0;
}
```

AppThread.h

This is where our worker used above is defined. This worker is responsible for the bulk of the execution. To fully understand what the worker is doing, SquidStateHandler should be sufficiently traced.

```
#include "AppThread.cpp"
void Dowork() {
    AisSquidstatStarter appStarter;
    appStarter.initApp(); // Initializes the AisSquidstatStarter

    /*
     * SquidStateHandler can be traced from the constructor.
     * What it does: ExpDataNotifier is registered as the global notifier.
     * fillAppData is then called which is responsible for loading: the list
     * of custom experiments the user has saved to their documents folder;
     * the builder elements dynamic libraries which are constitutes of the
     * custom  experiments; and the Squidstat device which is connected in
     * some usb port (the port may be specified, but is not required).
     * Then the global notifier ExpDataNotifier is responsible for starting the
     * manual experiment after it registers that a device has connected to the
     * SquidStatHandler.
     */
    SquidStateHandler start;

    appStarter.execApp(); // Must be called on same thread as initApp
}
```

SquidStateHandler.cpp

SquidStateHandler is mostly full of helper routines to complement the running of simple manual experiment. Thread timers are used to give some guidance on how one might automate pausing and resuming the Squidstat device. Routines used for saving csv files are provided for additional guidance for post-experiment analysis.

```
void SquidStateHandler::startManualExperiment() {
    qDebug() << "Manual experiment started";

    auto expInfoData = expSelector->getNextExpInfo();

    if ( expInfoData.acFilePath.isEmpty() ||
        expInfoData.dcFilePath.isEmpty()) {
        handler->closeApplication();
        return;
    }

    auto deviceName = DEVICE_NAME;
    auto channelNumber = 0;

    if (!connectedDevice.contains(deviceName)) {
        qDebug() << deviceName << "is not found";
        return;
    }

    deviceSettings = new AisDeviceSetting(deviceName, channelNumber);
    exp = new Experiment(deviceSettings, eventHandler);
    exp->createACDataFile(expInfoData.acFilePath);
    exp->createDCDataFile(expInfoData.dcFilePath);

    auto errorFlag = handler->startManualExperimentM(exp->getManualExperiment());

    if (errorFlag != AisStatus::NO_ERROR) {
        delete exp;
        delete deviceSettings;
    }

    startAlltimers();
}
```

ExpDataNotifier.cpp

The ExpDataNotifier object is the connection between your program and the device. When the notifier recieves a signal from the device, it will execute a corresponding function. For

example, if the device signals that DC data is to arrive, ExpDataNotifer::readDCExperimentData is called and from the below definition, the handler executes the code to save the DC data.

```
/*
 *  The Squidstat Device outputs signals which are caught by the ExpDataNotifier
 *  which then is responsible for instructing the handler to execute
 *  code corresponding to the appropriate output by the device.
*/
void ExpDataNotifier::readDCExperimentData(QUuid id) {
    handler->DCDataExperiment(id); // saves DC data to DC data file.
}
void ExpDataNotifier::readACExperimentData(QUuid id) {
    handler->ACDataExperiment(id); // saves AC data to AC data file
                                   // (not executed in manual experiments)
}
void ExpDataNotifier::experimentStopped(QUuid id) {
    handler->StopExperiment(id);
}
void ExpDataNotifier::experimentPaused(QUuid id) {
    handler->PauseExperiment(id);
}
void ExpDataNotifier::experimentResumed(QUuid id) {
    handler->ResumeExperiment(id);
}
void ExpDataNotifier::instrumentReadyToUse(QString newDevice) {
    handler->addInstrumnets(newDevice);
}
void ExpDataNotifier::instrumentDisconnected(QString removeDevice) {
    handler->instrumentRemove(removeDevice);
}
```