

# Open Energy Profiler Toolset

Drive innovations in the field of low-powered technologies

Documentation

## GRAPHICAL USER INTERFACE

DEVELOPER GUIDE

30. JUNE 2025



## Revision History

Version	Date	Description
1.0	30.06.2025.	Initial draft



# CONTENT

<b>1. INTRODUCTION .....</b>	<b>4</b>
<b>2. ARCHITECTURE .....</b>	<b>5</b>
<b>3. SOURCE CODE ORGANIZATION .....</b>	<b>7</b>
<b>4. CLASSES .....</b>	<b>10</b>
<b>4.1. Top Level .....</b>	<b>10</b>
4.1.1. Device .....	10
4.1.2. Device container .....	12
<b>4.2. Processing .....</b>	<b>13</b>
4.2.1. File processing .....	13
4.2.2. Energy Point processing .....	14
4.2.3. Data processing .....	16
<b>4.3. Links .....</b>	<b>18</b>
4.3.1. Stream Link .....	18
4.3.2. Control Link .....	19
4.3.3. Status Link .....	21
4.3.4. Energy Debugging Link .....	22
<b>4.4. Windows .....</b>	<b>23</b>
4.4.1. Add Device .....	23
4.4.2. Console .....	24
4.4.3. Data Analyzer .....	26
4.4.4. Device .....	28
4.4.5. Data Statistics .....	30
4.4.6. Calibration .....	32
<b>5. BUILD AND RUN INSTRUCTIONS .....</b>	<b>34</b>
<b>LIST OF FIGURES .....</b>	<b>39</b>
<b>REFERENCES .....</b>	<b>40</b>



# 1. INTRODUCTION

This document serves as a detailed technical manual for understanding, building, and extending the **OpenEPT Graphical User Interface (OpenEPT GUI)**, a central software component within the **Open Energy Profiler Toolset (OpenEPT)**. It is specifically tailored for software developers, embedded systems engineers, and future contributors who will be involved in the design, development, testing, or maintenance of the OpenEPT GUI application. Readers are expected to possess foundational knowledge of C++, the Qt Creator development environment, Qt's signal-slot mechanism, and general principles related to embedded systems and graphical user interface development.

The OpenEPT GUI is a modular, high-performance application developed using the Qt framework. It enables real-time control, monitoring, and diagnostic interaction with the **Energy Profiler Probe (EPP)** [1]. The GUI provides an intuitive interface for streaming data management, hardware configuration, and execution of calibration and analysis procedures. Designed to run on both Windows and Linux platforms, the application offers cross-platform support and was built using Qt version 5.15.2, which includes several essential Qt modules to ensure flexibility and scalability.

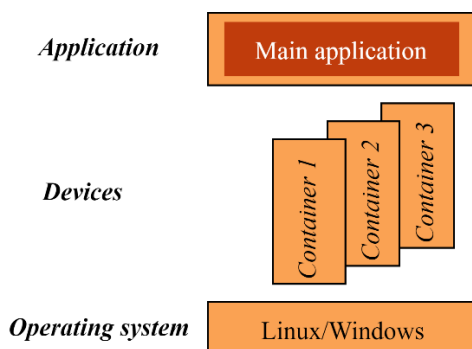
The primary responsibilities of the OpenEPT GUI include acquiring, visualizing, and processing real-time data from the EPP, as well as presenting the device's operational status and offering control over key runtime parameters. Its architecture is carefully structured to allow straightforward extension and adaptation for energy profiling use cases, making it well-suited for ongoing research and industry applications that require dynamic system interaction.

To support modularity and maintainability, the OpenEPT GUI is designed as a layered architecture, based on guidelines [2], comprising four main layers: Logic, Processing, Windows, and Link. Each of these layers is implemented through dedicated C++ classes, with clearly defined responsibilities. This separation ensures organized data flow, from raw acquisition and calibration through processing, up to user interaction, while promoting clean code structure, scalability, and testability. The application leverages Qt's event-driven programming model and multithreading features to ensure smooth user experiences and low-latency data visualization, even during high-frequency sampling.

Beyond architectural documentation, this guide provides complete instructions for setting up a compatible development environment. This includes installing the appropriate version of the Qt framework, selecting required Qt modules, configuring the toolchain, and preparing Qt Creator for building and running the application.

## 2. ARCHITECTURE

The architecture of the Acquisition Device Firmware is presented in the Figure 2.1:

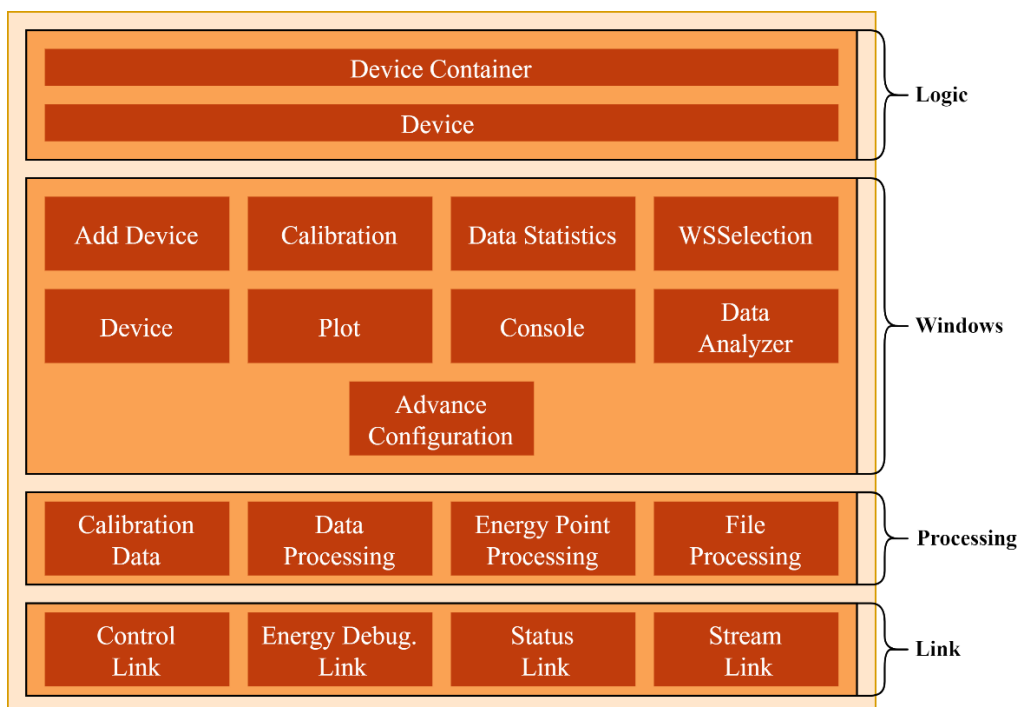


**Figure 2.1** – Acquisition Device's Firmware

The diagram presents a high-level architectural overview of the OpenEPT application, highlighting the separation of functionalities between the main application, individual device containers, and the underlying operating system.

At the topmost layer is the Main Application, which serves as the central control unit responsible for managing the overall workflow, user interface, and orchestration of all connected acquisition devices. This application operates in relation to the number of devices and abstracts complexity by handling each device through a dedicated component referred to as a container.

The Device Layer illustrates the concept of device containers, Container 1, Container 2, and Container 3. Each container encapsulates all logic, data processing, and communication mechanisms related to an individual acquisition device. This design ensures that devices operate independently from one another, allowing the application to scale horizontally as more devices are added. The containers isolate device-specific behavior, facilitate parallelism, and simplify debugging and development. All classes instantiated inside the device container are presented on Figure 2.2.



**Figure 2.2** - Device Container's elements

All classes instantiated inside Device Container are organized into four clearly defined layers: Logic Layer, Windows Layer, Processing Layer, and Link Layer. This layered design provides a modular foundation for robust development, where responsibilities are cleanly separated and encapsulated within

distinct blocks. Each block shown in the diagram corresponds to a dedicated C++ class, ensuring encapsulation, maintainability, and clear abstraction of responsibilities.

At the top of the architecture is the Logic Layer, which coordinates all high-level operations. It is composed of the *DeviceContainer* and *Device* classes. The *DeviceContainer* serves as the entry point for managing one or more devices simultaneously. Each *Device* instance maintains full ownership over its processing pipeline, user interface components, and communication links. This separation enables the system to support multiple concurrent measurement sessions while keeping their execution contexts isolated and well-managed.

Beneath the logic layer lies the Windows Layer, which comprises the GUI modules that form the visual and interactive core of the application. Each block within this layer is implemented as an independent Qt-based C++ class, presenting a self-contained graphical interface. This includes modules such as *Add Device*, for incorporating new hardware into the system, *Calibration*, for tuning measurement accuracy, and *Data Statistics*, for summarizing data trends. *Plot* and *Console* windows offer real-time visual and textual feedback, respectively, while *Data Analyzer* supports offline review. The *Advance Configuration* module provides access to expert-level settings, allowing deeper customization of system behavior.

Backend for the graphical interface is the Processing Layer, which performs critical backend tasks essential for data integrity and feature extraction. This layer includes components such as *Calibration Data*, which stores and manages device-specific calibration coefficients, and *Data Processing*, which transforms raw input signals into usable information. The *Energy Point Processing* module detects and annotates energy-related events within the data stream, while *File Processing* manages structured logging and persistent storage of collected data. All processing modules are designed to operate either synchronously or within dedicated threads, depending on system requirements.

At the foundation of architecture is the Link Layer, which manages all external communication with EPP [1]. Each link is encapsulated in its own C++ class and typically operates in a dedicated thread to ensure responsiveness and real-time performance. The *Control Link* establishes a TCP connection for command-based interaction, while the *Energy Debug Link* receives structured labels and debug events. *Status Link* functions as a TCP server that accepts periodic status reports from devices, and the *Stream Link* serves as a high-bandwidth UDP receiver for real-time acquisition of analog samples, such as voltage and current.

Altogether, this layered and modular architecture enables OpenEPT to function as a highly adaptable and efficient platform for embedded measurement and energy analysis. By decoupling system components into specialized, reusable C++ classes, the software maintains strong cohesion and extensibility. This design not only simplifies development and debugging but also positions the system well for future expansion and integration into diverse measurement environments.

At the foundation, the Operating System Layer, represented here by Linux/Windows, provides the runtime environment for both the main application and the device containers. The architecture is designed to be platform-independent and can operate on either of the supported operating systems. The containers themselves are implemented as software modules but are conceptually treated as isolated units managed by the main application.

### 3. SOURCE CODE ORGANIZATION

Complete project source code is available under *Firmware* repository on the official [OpenEPT organization on the GitHub](#) [3].

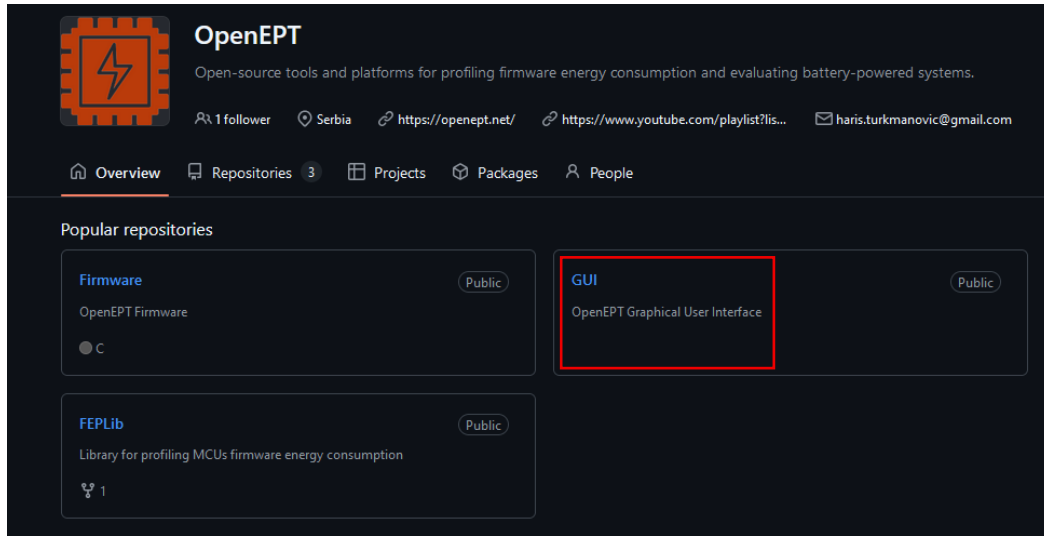


Figure 3.1 - OpenEPT Organization on GitHub and GUI repository

The GUI repository contains two main directories:

- **Documentation**  
Here are located scripts to generate documentation based on code comments (Doxygen)
- **Source**  
Where complete OpenEPT GUI source code is located

Therefore, all source code of OpenEPT GUI is located inside Source directory which structure is illustrated on Figure 3.2

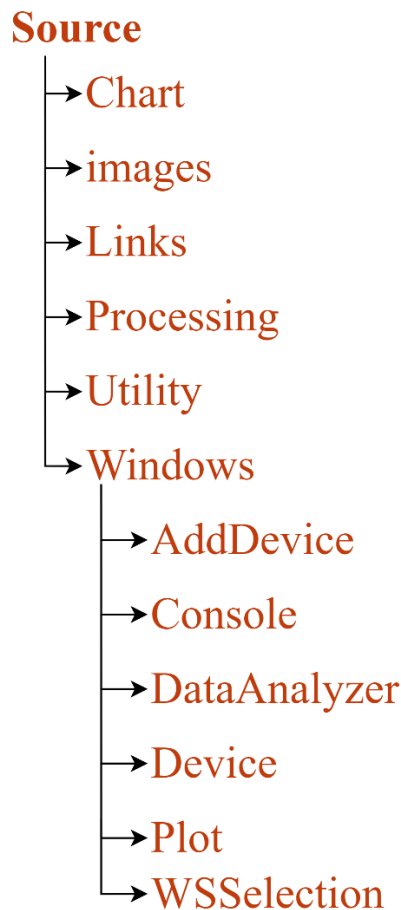


Figure 3.2 – OpenEPT Source directory structure

The Source directory serves as the root of the application’s codebase, organizing all the functional components of the OpenEPT GUI software into well-separated modules. Each subfolder within Source encapsulates a specific aspect of the application, contributing to a modular and maintainable design.

- **Chart**  
This directory contains components responsible for visualizing data. It includes custom plotting widgets, real-time chart implementations, and mechanisms for displaying voltage, current, or consumption curves. The logic here interfaces with graphical libraries like *QCustomPlot* or *QChart* to present data to the user.
- **Images**  
The images folder stores graphical assets used throughout GUI, such as icons, splash screens, button graphics, and UI embellishments. These resources are loaded into the application at runtime to support a consistent and professional interface appearance
- **Links**  
This directory manages network communication interfaces and abstractions. It includes classes for handling TCP/UDP connections to hardware devices, such as *StatusLink* and *StreamLink*, as well as connection lifecycle management (connect/disconnect, message parsing, etc.).
- **Processing**  
This folder hosts data processing modules, including filtering, analysis, transformation, or aggregation of incoming measurement streams. Components here serve as the computational backend, preparing data for visualization or further calibration.
- **Utility**





The Utility folder provides supporting services and helper classes that are shared across modules. It may include logging utilities, configuration parsers, data validators, color palettes, unit conversion helpers, or file handling logic.

- **Windows**

This high-level directory groups all GUI subcomponents into dedicated windows or widgets. Each subfolder represents a specific user interface feature of the application and encapsulates its respective UI logic.

- **AddDevice**

Contains the implementation for the dialog or window that allows users to manually add a device by IP address and port. It likely includes input validation and emits connection requests.

- **Console**

Implements the textual command interface window. It allows manual sending of control messages to devices, shows responses, supports command history, and may feature auto-completion for faster use

- **Data Analyzer**

Hosts the interface for analyzing acquired measurement data. This may include tools for exporting, segmenting, statistical analysis, or browsing logged sessions.

- **Device**

Encapsulates the main interface for managing a connected device. This includes configuration dialogs, real-time monitoring, control toggles, and integration with other subsystems like plots or logs.

- **Plot**

Provides the interactive window or panel where data is plotted in real-time or from logs. It handles rendering updates, axis scaling, plot interaction, and curve overlays.

- **WSSelection**

Likely stands for "Workspace Selection" or "Working Session Selection", offering an interface for the user to choose which device, dataset, or operating mode to engage. It may also manage workspace configurations or presets.

Each of these directories is organized to promote encapsulation and separation of concerns, making the project easier to scale, debug, and maintain.



## 4. CLASSES

### 4.1. Top Level

#### 4.1.1. Device

BLOCK SUMMARY			
Name	Device	Layer	Logic
Version	1.03		
Related files			
<i>Top source file</i>			
Source/device.cpp			
<i>Top header file</i>			
Source/device.h			

The *Device* class encapsulates the complete logic required to configure, control, and monitor a EPP. It is designed as a high-level abstraction built on top of Qt's *QObject*, and serves as a central coordinator for data streaming, analogue signal processing, and peripheral control. The class enables seamless communication between a host application and the embedded device via multiple link types: *ControlLink*, *StreamLink*, *StatusLink*, and *EDLink*. These links manage command-based interactions, real-time data transfer, status updates, and energy point processing respectively.

This class enables dynamic configuration of the ADC subsystem. Through dedicated setter and getter methods, it supports adjusting key acquisition parameters including ADC resolution, sampling period, channel sampling time, clock division factor, and hardware averaging ratio. It also manages the synchronization of these settings by computing the effective sampling time and updating dependent modules such as the *DataProcessing* and *ChargingAnalysis* components. A dedicated method is provided to automatically retrieve the current configuration from the device.

Beyond signal acquisition, the *Device* class enables real-time control over various power path elements and actuators within the system. It implements control commands for enabling or disabling the load, DAC, battery path, and power path. The current DAC setting is computed using a fitted inverse function, ensuring precise digital-to-analog translation based on calibration data. Overvoltage, undervoltage, and overcurrent conditions can be queried and monitored in real time.

The data streaming capability is enabled via the *createStreamLink* method, which sets up the stream server and connects it to the *DataProcessing* pipeline. Incoming data samples are decoded, processed, and re-emitted as Qt signals containing voltage, current, and derived energy metrics. Statistical analysis, event-based energy processing (via *EPProcessing*), and battery charging state tracking (via *ChargingAnalysis*) are also integrated within the class structure. These components interact through Qt signal-slot mechanisms, ensuring asynchronous, non-blocking data flow.

Internally, the *Device* class maintains a comprehensive set of state variables that represent the real-time configuration and operational status of the device. These include ADC characteristics, offsets, sampling configurations, current and voltage thresholds, and component states. All interactions with the firmware are performed through structured string-based command protocols, and each result is validated, parsed, and used to update internal state or propagate signals to higher-level modules.

In summary, the *Device* class is responsible for the full lifecycle management of a measurement device, from low-level ADC setup to high-level power control and data streaming. It abstracts the



complexity of embedded communication and real-time signal handling while providing a modular and extensible interface for further system development.



### 4.1.2. Device container

BLOCK SUMMARY			
<b>Name</b>	Device Container	<b>Layer</b>	Logic
<b>Version</b>	1.03		
<b>Related files</b>			
<i>Top source file</i>			
Source/devicecontainer.cpp			
<i>Top header file</i>			
Source/devicecontainer.h			

The *DeviceContainer* class functions as a high-level integration layer that connects the GUI front-end (represented by *DeviceWnd*) with the hardware interface logic (encapsulated by the *Device* class). It is designed to manage and coordinate interactions between the user interface, logging system, device control, data acquisition, and file processing mechanisms. This class enables the encapsulation of a single acquisition session, maintaining contextual state and behavior across user-driven actions and device-generated events.

This class enables initialization of all signal-slot bindings required for real-time operation of the device interface. Upon construction, it connects GUI actions (such as parameter changes, acquisition control commands, and calibration updates) to corresponding slots that translate these actions into appropriate method calls on the underlying *Device* instance. It also binds the *Device* class signals (e.g., hardware status changes, new samples received, energy point updates) back to the GUI, ensuring responsive feedback and seamless visualization. In this role, *DeviceContainer* acts as a mediator and state synchronizer between the human operator and the hardware abstraction layer.

Internally, *DeviceContainer* maintains additional services such as logging (via the *Log* class), file-based data logging (via the *FileProcessing* class), and runtime state variables like elapsed acquisition time, energy point enablement status, and file write permissions. It ensures that the correct directories and files are created for storing acquisition data, summaries, and computed statistics, and manages filename conflicts and profile overwrites with user confirmation through message boxes. When acquisition is active and saving is enabled, incoming data (voltage, current, consumption, energy points) is continuously streamed to disk.

Additionally, *DeviceContainer* enables translation and validation of textual GUI inputs into enumerated types for ADC resolution, clock dividers, sampling times, and averaging modes. This mapping ensures the consistency and correctness of configuration commands issued to the hardware. Device configuration can also be applied in bulk via the *onDeviceWndNewConfiguration* method, which parses and applies all relevant acquisition parameters from an advanced configuration structure.

This class further monitors acquisition state and protects data integrity by enforcing profile setup before writing to file. It enables acquisition start, pause, stop, and refresh operations, while logging all relevant actions, warnings, and errors. It also tracks elapsed acquisition time via a periodic timer and propagates this information to the GUI, keeping the user informed of ongoing progress.

## 4.2. Processing

### 4.2.1. File processing

BLOCK SUMMARY			
<b>Name</b>	File processing	<b>Layer</b>	Processing
<b>Version</b>	1.03		
<b>Related files</b>			
<i>Top source file</i>			
Source/Processing/fileprocessing.cpp			
<i>Top header file</i>			
Source/Processing/fileprocessing.cpp			

The *FileProcessing* class is responsible for managing all file input/output operations related to data logging and acquisition results within the OpenEPT system. It supports multiple files types and organizes output data into well-structured CSV and text formats. This class enables the storage of raw measurement samples, calculated consumption data, summary information, and extracted energy points in a structured and thread-safe manner. It also ensures data integrity and writes efficiency through a dedicated file-processing thread and synchronized operations.

Upon initialization, the class creates and opens the required output files depending on the specified *fileprocessing\_type\_t*. When operating in *FILEPROCESSING\_TYPE\_SAMPLES* mode, the class moves itself to a dedicated thread and establishes queued connections for sample, consumption, and energy point logging. This design enables safe, non-blocking data logging even when large volumes of data are streamed in real time from the acquisition device.

Each file is created with a header section that includes a user-defined title and a content-specific column structure. For instance, voltage/current samples are stored in *vc.csv* with timestamped entries, while consumption data is logged in *cons.csv*, and energy point data in *ep.csv*. The *OpenEPT.txt* summary file is used to record session metadata such as acquisition start/stop time and user-defined notes.

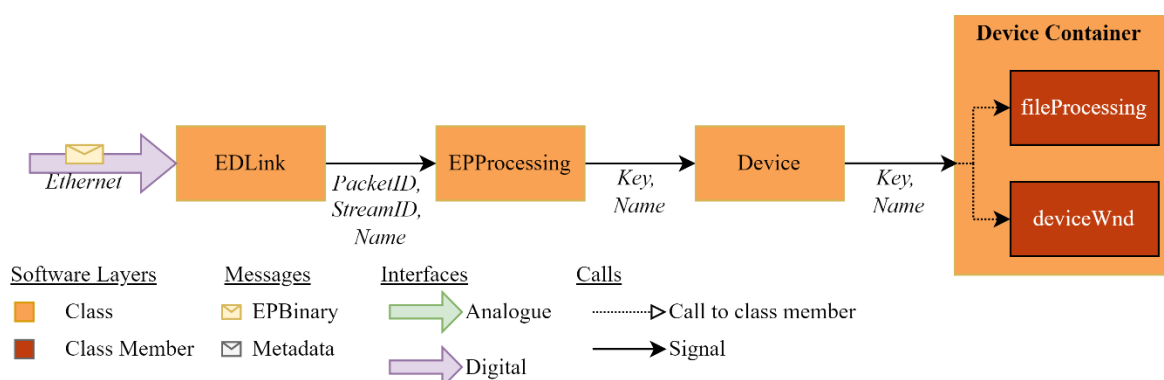
The class enables both direct and queued appending of data. Queued appending is particularly useful for multithreaded contexts where data is generated asynchronously. Corresponding signal-slot pairs such as *sigAppendSampleData* and *onAppendSampleData* ensure that sample data is written in the background thread, preventing UI blocking or race conditions. Direct appending functions (*appendSampleData*, *appendConsumptionData*, etc.) are also available for simpler or single-threaded use cases.

Internally, the *FileProcessing* class maintains key configuration state such as the file type, output paths, and header contents. It offers methods to set or reset headers, reopen files (e.g., when overwriting an existing session), and append new entries with precision formatting. For synchronization during threaded startup, a *QSemaphore* is used to delay data logging until all files are safely opened and ready.

## 4.2.2. Energy Point processing

BLOCK SUMMARY			
Name	File processing	Layer	Processing
Version	1.03		
Related files			
<i>Top source file</i>			
Source/Processing/epprocessing.cpp			
<i>Top header file</i>			
Source/Processing/epprocessing.h			

The *EPProcessing* class is responsible for managing the detection, tracking, and handling of Energy Points (EPs) within the OpenEPT measurement framework. An energy point represents a notable event or annotation in the energy profile of a system—typically associated with a specific sample, timestamp, or packet. This class enables asynchronous processing of EP data, allowing the system to handle high-frequency measurements without blocking the main application thread. Figure 4.1 illustrates classes that are, besides *EPProcessing*, included in Energy Points processing mechanism.



**Figure 4.1** - Overview of the OpenEPT GUI classes that process and visualize Energy Points

Upon instantiation, *EPProcessing* moves itself to a dedicated thread, isolating its operations from the main thread to ensure non-blocking behavior during real-time acquisition. This threading model is particularly important when managing queued data such as energy point names and values arriving independently via separate signals.

The *onNewEPValueReceived* function is triggered when a new energy point value is received. This includes the *PacketID*, the computed value, and its associated key (which typically represents a timestamp or sample index). Instead of matching incoming values with previously known names (which is commented out in the current implementation), the function directly creates a new *EPInfo* instance and appends it to the internal *epList* for later processing or reference.

Conversely, *onNewEPNameReceived* handles the case where only the energy point name is known, along with a *PacketID* and *SampleID*. It computes a *samplePosition* based on a default buffer size and emits a *sigEPProcessed* signal immediately. This allows the system to reflect named energy points on the user interface, even if the value is not yet associated.

The *EPInfo* class serves as a container for individual energy point data. It encapsulates the packet identifier (*packetID*), the assigned name (if any), the associated value and key, and flags indicating whether the name or value has been set. The class provides methods to assign names and values independently, retrieve stored attributes, and compare against a packet ID via the operator `==` overload.

Although the matching mechanism between name and value assignments is currently disabled (commented out in *onNewEPValueReceived*), the infrastructure is in place to enable full pairing between



asynchronously received EP metadata. Once re-enabled, this logic would allow EPProcessing to accumulate partial data (name or value) and emit complete EP results only when both parts are available.



### 4.2.3. Data processing

BLOCK SUMMARY			
<b>Name</b>	Data processing	<b>Layer</b>	Processing
<b>Version</b>	1.03		
<b>Related files</b>			
<i>Top source file</i>			
Source/Processing/dataprocessing.cpp			
<i>Top header file</i>			
Source/Processing/dataprocessing.h			

The *DataProcessing* class is responsible for handling the core data acquisition and signal processing functionality within the OpenEPT platform. It manages the collection, calibration, buffering, and processing of voltage and current measurements received from the acquisition device. This class operates in its own dedicated thread to ensure that all time-sensitive operations, such as high-speed sampling and FFT processing, can be executed without blocking the main GUI or interfering with other system tasks.

Upon creation, the class sets up key internal parameters such as buffer sizes, calibration coefficients, and filtering options. It allocates buffers for voltage and current samples, timestamps (keys), and energy consumption data. Depending on the selected device mode, the raw data received through the *onNewSampleBufferReceived* method is either interpreted as integer values or as packed binary words. Each sample is converted into meaningful voltage or current values using user-defined or default calibration constants, including gain, shunt resistance, voltage correction factor, and ADC reference voltage. These conversions are updated automatically if calibration data changes.

In addition to calibration, *DataProcessing* tracks a wide range of signal statistics in real time. It computes and stores minimum, maximum, and average values of both voltage and current throughout the measurement session. For energy tracking, it offers two consumption modes: one that estimates energy using instantaneous current values (current mode) and another that integrates current over time to produce a cumulative energy profile (cumulative mode), expressed in mAh. The user can switch between these modes depending on whether the focus is on momentary power draw or long-term energy usage.

Once the number of acquired buffers reaches a user-defined threshold, the class finalizes data processing and emits a variety of signals. These include *sigNewVoltageCurrentSamplesReceived* for voltage/current data, *sigNewConsumptionDataReceived* for energy data, *sigSamplesBufferReceiveStatistics* for transmission diagnostics, and *sigSignalStatistics* for updated measurement statistics. It also emits *sigAverageValues* to report average current and voltage for the completed buffer session. These signals are presented on Figure 4.2.





```
switch(consumptionMode)
{
case DATAPROCESSING_CONSUMPTION_MODE_CURRENT:

    switch(measurementMode)
    {
    case DATAPROCESSING_MEASUREMENT_MODE_VOLTAGE:
        emit sigNewVoltageCurrentSamplesReceived(voltageDataCollected, voltageDataCollectedFiltered, voltageKeysDataCollected, voltageKeysDataCollected);
        emit sigNewConsumptionDataReceived(ffftDataCollectedVoltage, fftKeysDataCollected, DATAPROCESSING_CONSUMPTION_MODE_CURRENT);

        break;
    case DATAPROCESSING_MEASUREMENT_MODE_CURRENT:
        emit sigNewVoltageCurrentSamplesReceived(currentDataCollected, currentDataCollectedFiltered, currentKeysDataCollected, currentKeysDataCollected);
        emit sigNewConsumptionDataReceived(ffftDataCollectedCurrent, fftKeysDataCollected, DATAPROCESSING_CONSUMPTION_MODE_CURRENT);

        break;
    }
    break;
case DATAPROCESSING_CONSUMPTION_MODE_CUMULATIVE:
    if(filteringEnable == 1)
    {
        emit sigNewVoltageCurrentSamplesReceived(voltageDataCollectedFiltered, currentDataCollectedFiltered, voltageKeysDataCollected, currentKeysDataCollected);
        emit sigNewConsumptionDataReceived(cumulativeConsumptionDataCollected, consumptionKeysDataCollected, DATAPROCESSING_CONSUMPTION_MODE_CUMULATIVE);
    }
    else
    {
        emit sigNewVoltageCurrentSamplesReceived(voltageDataCollected, currentDataCollected, voltageKeysDataCollected, currentKeysDataCollected);
        emit sigNewConsumptionDataReceived(cumulativeConsumptionDataCollected, consumptionKeysDataCollected, DATAPROCESSING_CONSUMPTION_MODE_CUMULATIVE);
    }
    break;
}
//emit sigEBP(ebpValue, ebpValueKey);
emit sigSamplesBufferReceiveStatistics(dropRate, dropPacketsNo, receivedPacketCounter, lastReceivedPacketID, ebpNo);
emit sigSignalStatistics(voltageStat, currentStat, consumptionStat);
emit sigAverageValues(currentStat.average, voltageStat.average);
```

Figure 4.2 - Signals inside DataProcessing class

All buffers are reset after each full acquisition cycle, ensuring that new data can be collected seamlessly. This reset is handled cleanly through modular methods like *initVoltageBuffer* and *initCurrentBuffer*, which resize and zero the internal data arrays.

## 4.3. Links

### 4.3.1. Stream Link

BLOCK SUMMARY			
Name	Stream Link	Layer	Link
Version	1.03		
Related files			
<i>Top source file</i>			
Source/Links/streamlink.cpp			
<i>Top header file</i>			
Source/Links/streamlink.h			

The *StreamLink* class is designed to manage UDP-based data streaming in the OpenEPT system. It provides a dedicated interface for receiving high-frequency measurement packets over the network and seamlessly integrating them into the data processing pipeline. The class is implemented as a *QObject* and is executed in a separate thread to ensure non-blocking communication and thread-safe operation across the application.

Upon initialization, the class assigns a default packet size (*STREAM\_LINK\_PACKET\_SIZE*) and creates a *QThread* instance dedicated to handling UDP operations. When the thread is started through the *enable* method, it triggers the *initStreamLinkThread* slot, which binds a new *QUdpSocket* to the configured port and sets an increased receive buffer size to accommodate high-throughput datagram traffic. The socket is configured to listen for incoming datagrams, and whenever data is available, the *readPendingData* slot is automatically invoked.

The class supports dynamic configuration of the network port and stream ID through the *setPort* and *setID* methods. It also provides a method *setPacketSize* for adjusting the expected number of 16-bit values per packet, giving flexibility to accommodate various hardware or firmware configurations.

When datagrams arrive, the *readPendingData* function (Figure 4.3) processes each packet in a loop. It first allocates space for the incoming byte stream and then uses *receiveDatagram* to extract the actual payload. The first 8 bytes of each packet are assumed to contain a 32-bit counter (used as packet ID) and a 32-bit magic field (used for auxiliary data such as flags or checksums). The remainder of the packet contains raw 16-bit signed integers, which are copied into a *QVector<short>* and then cast to a *QVector<double>* to standardize the format used downstream. Once the data is prepared, the class emits the *sigNewSamplesBufferReceived* signal with the converted sample array, packet counter, and magic number.

```
void StreamLink::readPendingData()
{
    unsigned int counter;
    unsigned int magic;
    while(udpSocket->hasPendingDatagrams())
    {
        QByteArray receivedData;
        QVector<short> data;
        QVector<double> data_double;

        receivedData.resize(udpSocket->pendingDatagramSize());
        QNetworkDatagram datagram = udpSocket->receiveDatagram();
        receivedData = datagram.data();

        data.resize(packetSize);
        memcpy(data.data(), receivedData.data()+8, packetSize*2);
        memcpy(&counter, receivedData.data(), 4);
        memcpy(&magic, receivedData.data()+4, 4);
        data_double.reserve(data.size());
        std::copy(data.cbegin(), data.cend(), std::back_inserter(data_double));
        emit sigNewSamplesBufferReceived(data_double, counter, magic);
    }
}
```

Figure 4.3 - Parsing Stream Message inside Stream Link



### 4.3.2. Control Link

BLOCK SUMMARY			
<b>Name</b>	Control Link	<b>Layer</b>	Link
<b>Version</b>	1.03		
<b>Related files</b>			
<i>Top source file</i>			
Source/Links/controllink.cpp			
<i>Top header file</i>			
Source/Links/controllink.h			

The *ControlLink* class implements a TCP client interface used to establish and manage a control communication link between the host application and a remote embedded device. It is designed to issue textual commands over TCP/IP and receive formatted responses, supporting command/response-based interaction with the device's firmware.

Upon initialization, the class sets up a *QTcpSocket*, enables the TCP keep-alive option, and stores initial default values such as IP address and port number. The *establishLink* method is used to initiate a connection to the target device. It takes an IP address and port number, attempts to connect using Qt's networking primitives, and returns the current link status. This method also sets up signal-slot connections for handling disconnection and reconnection events.

In the event of an unexpected disconnection, the *onDisconnected* slot is invoked. It sets the internal status to reconnecting and starts a *QTimer* that attempts to re-establish the connection periodically using the *reconnect* slot. Once reconnected, the *onReconnected* slot is triggered, at which point the TCP keep-alive options are applied at the system socket level to help detect stale or dead links in environments where continuous communication cannot be guaranteed.

The class provides the *executeCommand* method as the primary mechanism for interacting with the remote device. This method sends a command string, waits for a response within a specified timeout, and parses the response. It performs basic validation to ensure the command is properly terminated and returns the parsed result or an appropriate error message if the exchange fails. Definition of this method is presented on Figure 4.4.



```
bool Controllink::executeCommand(QString command, QString* response, int timeout)
{
    QByteArray receivedData;
    QByteArray dataToSend(command.toUtf8());
    QString receivedResponse = "";
    receivedData.clear();
    if(linkStatus != CONTROL_LINK_STATUS_ESTABLISHED)
    {
        *response = QString("Control link not established");
        return false;
    }
    tcpSocket->flush();
    tcpSocket->write(dataToSend);
    tcpSocket->waitForBytesWritten();
    if(tcpSocket->waitForReadyRead(timeout) != true)
    {
        *response = QString("Unable to read data");
        return false;
    }
    receivedData = tcpSocket->readAll();
    QString responseAsString(receivedData);
    /* Check did we receive "\r\n" */
    if(!responseAsString.contains("\r\n"))
    {
        *response = QString("End of command not detected");
        return false;
    }
    /* Split response to identify OK*/
    QStringList responseParts = responseAsString.split(" ");
    if(responseParts[0] != "OK")
    {
        *response = QString("ERROR");
        return false;
    }
    for(int i = 1; i < responseParts.size(); i++)
    {
        *response += responseParts[i];
        if((i+1) != responseParts.size())
        {
            *response += " ";
        }
    }
    /*take substring until*/
    *response = (*response).split("\r\n")[0];
}
```

Figure 4.4 - executeCommand method definition

Additionally, the *Controllink* class offers helper methods such as *getDeviceName* to retrieve identifying information from the device and *setSocketKeepAlive* to configure low-level socket options such as idle timeout, probe interval, and maximum probe count. These settings ensure reliable long-term connectivity, especially over unstable or mobile networks.

### 4.3.3. Status Link

BLOCK SUMMARY			
Name	Status Link	Layer	Link
Version	1.03		
Related files			
<i>Top source file</i>			
Source/Links/statuslink.cpp			
<i>Top header file</i>			
Source/Links/statuslink.h			

The *StatusLink* class provides a TCP-based communication channel for receiving status messages from one or more remote clients in the OpenEPT system. Its main purpose is to listen for incoming TCP connections on a specified port, accept them, and process textual status messages that are sent during the acquisition process or system operation. The class is implemented as a *QObject* and operates within its own dedicated thread to isolate network activities from the main application and ensure smooth, asynchronous message handling.

Upon construction, the *StatusLink* class sets up a *QThread* instance named "OpenEPT - Status link server" and connects its *started* signal to the internal slot *onServerStarted*. The server is started by calling *startServer*, which triggers the thread to begin execution. When the thread starts, the class initializes a *QTcpServer* and binds it to the configured port, which is set in advance using the *setPort* method. If the binding fails, a diagnostic message is printed to aid debugging.

Once the server is active and listening, any new incoming client connection is handled by the *onNewConnectionAdded* (Figure 4.5) slot. This slot is triggered whenever the server detects a new pending connection. It retrieves each connection using *nextPendingConnection*, stores the corresponding *QTcpSocket* in an internal list (*clientList*), and connects each socket's *readyRead* signal to the *onReadPendingData* slot. Additionally, a signal *sigNewClientConnected* is emitted with the IP address of the newly connected client, allowing external components (e.g., GUI or logging systems) to respond accordingly.

```
void StatusLink::onNewConnectionAdded()
{
    QTcpSocket* tmpSocket;
    while(tcpServer->hasPendingConnections())
    {
        tmpSocket = tcpServer->nextPendingConnection();
        clientList.append(tmpSocket);
        connect(tmpSocket, SIGNAL(readyRead()), this, SLOT(onReadPendingData()));
        emit sigNewClientConnected(QHostAddress(tmpSocket->peerAddress().toIPv4Address()).toString());
    }
}
```

Figure 4.5 - onNewConnectionAdded slot definition

The *onReadPendingData* slot is responsible for reading incoming messages from the connected clients. When triggered, it identifies the sending socket and reads available data into a fixed-size buffer (*STATUS\_LINK\_BUFFER\_SIZE*). Messages are read continuously in a loop until no more data remains, and each valid message is passed along through the *sigNewStatusMessageReceived* signal, which includes both the client's IP address and the received message.

In summary, the *StatusLink* class acts as a lightweight TCP server module for receiving and forwarding status updates from external devices or software components. Its multithreaded structure ensures that network I/O does not interfere with real-time data acquisition or UI responsiveness. With support for multiple simultaneous client connections, signal-based integration, and message dispatching, it forms an essential part of OpenEPT's runtime status monitoring and system coordination infrastructure.

### 4.3.4. Energy Debugging Link

BLOCK SUMMARY			
Name	Energy Debugging Link	Layer	Link
Version	1.03		
Related files			
<i>Top source file</i>			
Source/Links/edlink.cpp			
<i>Top header file</i>			
Source/Links/edlink.h			

The *EDLink* class implements a TCP server interface responsible for handling energy debugging messages sent from acquisition device unit to the OpenEPT GUI application. It listens for incoming TCP connections on a dedicated thread and processes client messages according to a predefined protocol. By default, the server binds to port 8000, although this can be configured as needed. Upon startup, *EDLink* initializes a *QTcpServer* and connects to its *newConnection* signal to detect when new clients attempt to establish a connection.

When a new client connects, the *onNewConnectionAdded* slot accepts the connection, adds the socket to an internal list of active clients, and connects the socket's *readyRead* signal to the *onReadPendingData* handler. This design ensures that incoming messages from all clients are handled asynchronously and independently.

The message format is defined with a simple structure: each message must end with a carriage return and newline (`\r\n`) and must include a minimum of 8 bytes of header data—comprising a 32-bit identifier and a 32-bit DMA ID—followed by an optional UTF-8 encoded payload. The *onReadPendingData* method, which definition is presented on Figure 4.6, accumulates data from the socket into a static buffer and parses messages in a loop, ensuring that fragmented or incomplete transmissions are handled. Valid messages are parsed to extract the header and payload, and a signal (*sigNewEPNameReceived*) is emitted to notify other components about the received data.

```
void EDLink::onReadPendingData()
{
    static QByteArray buffer; // Buffer to store incomplete messages
    QTcpSocket *clientSocket = qobject_cast<QTcpSocket*>(sender());
    if (!clientSocket) return;

    buffer.append(clientSocket->readAll()); // Read all available data into buffer

    while (true) {
        int endIndex = buffer.indexOf("\r\n"); // Find message delimiter
        if (endIndex == -1) {
            // No complete message found, wait for more data
            break;
        }

        // Extract a full message
        QByteArray message = buffer.left(endIndex);
        buffer.remove(0, endIndex + 2); // Remove processed message from buffer

        if (message.size() < 8) {
            qDebug() << "Received incomplete header, discarding";
            continue;
        }

        unsigned int id = *reinterpret_cast<unsigned int*>(message.data());
        unsigned int dmaID = *reinterpret_cast<unsigned int*>(message.data() + 4);
        QString payload = QString::fromUtf8(message.mid(8));

        qDebug() << "ID:" << id << "DMA:" << dmaID << "Message:" << payload;
        emit sigNewEPNameReceived(id, dmaID, payload);
    }
}
```

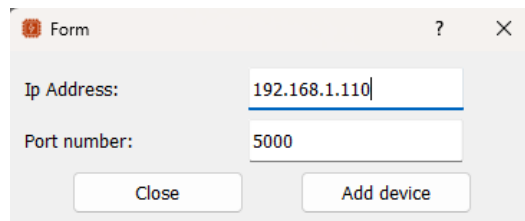
Figure 4.6 - Initial processing of energy debugging messages inside *EDLink* class

## 4.4. Windows

### 4.4.1. Add Device

BLOCK SUMMARY			
Name	Add Device	Layer	Windows
Version	1.03		
Related files			
Top source file			
Source/Windows/AddDevice/adddevicewnd.cpp			
Top header file			
Source/Windows/AddDevice/adddevicewnd.h			
Top UI file			
Source/Windows/AddDevice/adddevicewnd.ui			

The *AddDeviceWnd* class defines a graphical user interface component that enables the addition of a new acquisition device within the OpenEPT GUI application. This window provides users with a compact form to input connection parameters, specifically an IP address and port number, and then submit this data to the main application for device initialization. Layout of *AddDeviceWnd* is illustrated on **Figure 4.7**



**Figure 4.7** - Add Device Windows layout

Upon construction, the class initializes its UI elements using the *AddDeviceWnd* form definition, sets a fixed window size, and adjusts the font size for consistency with application-wide settings. Two interactive buttons are configured through Qt's signal-slot mechanism: one to close the window (*closePushb*) and another to trigger the device addition process (*addDevicePushb*).

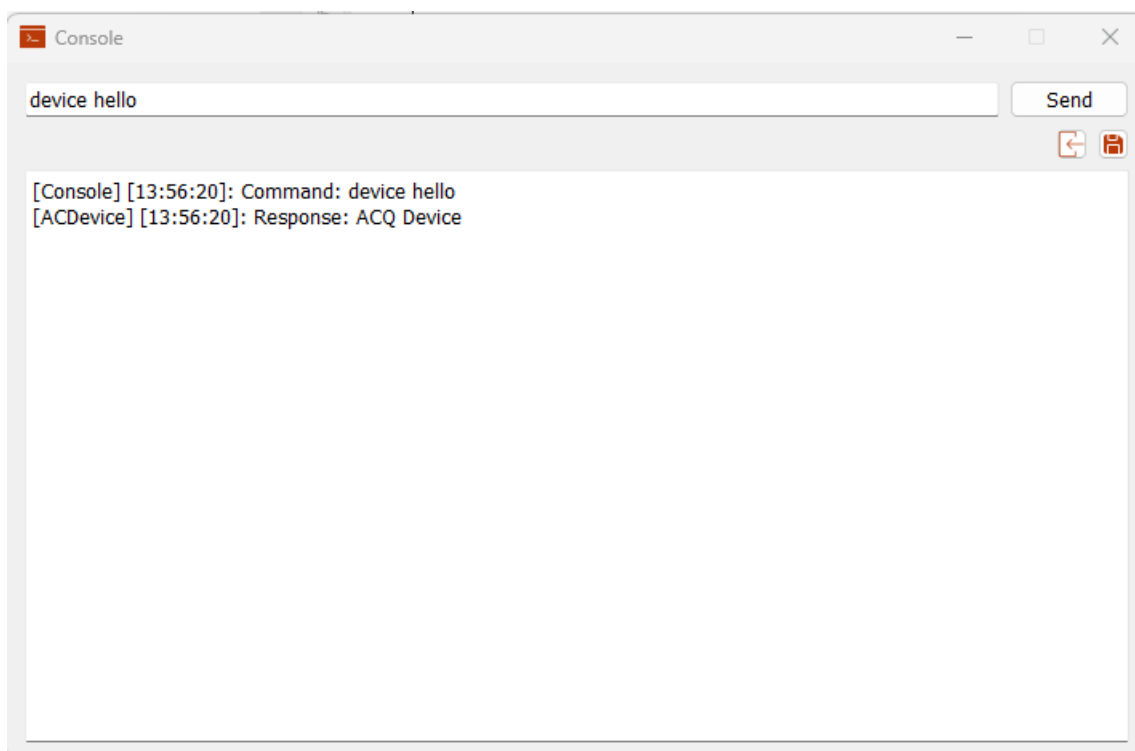
When the "Add Device" button is pressed, the widget reads the text fields for IP address and port number, emits a signal *sigAddDevice* with those parameters, and closes the window. This signal is typically connected to the part of the application responsible for managing device containers or communication links, thereby allowing dynamic runtime device registration.

The window is designed to be lightweight and modal in function, encapsulated in its own *QWidget* derived class to enable clean instantiation and destruction (delete *ui* in the destructor). This separation of concerns makes it easy to maintain, reuse, and modify the interface logic as the application evolves.

### 4.4.2. Console

BLOCK SUMMARY			
Name	Console	Layer	Windows
Version	1.03		
Related files			
Top source file			
Source/Windows/Console/consolewnd.cpp			
Top header file			
Source/Windows/Console/consolewnd.h			
Top UI file			
Source/Windows/Console/consolewnd.ui			

The *ConsoleWnd* class implements a terminal-style user interface within the OpenEPT GUI application, enabling direct communication with the connected device via command-line input. This component is especially useful for advanced users and developers who need fine-grained control or quick access to low-level device functions without navigating through the graphical configuration modules. Layout of *ConsoleWnd* is illustrated on **Figure 4.8**



**Figure 4.8** – Console Window Layout

Upon initialization, the UI is configured through the *ConsoleWnd* form. Font settings are adjusted for consistency, and signal-slot connections are established to handle user interactions. Specifically, the widget listens for clicks on the "Send" button or the press of the Enter key within the command line edit, both of which trigger the *onSendControlMsgClicked* slot.

A *QCompleter* is populated with a predefined list of supported device commands. This enhances usability by providing auto-completion suggestions as the user types, making it easier to explore the available command set and reducing the chance of input errors. The list includes various device management, ADC/DAC configuration, power path control, and charger commands. **Figure 4.9** illustrate autocomplete option within a console window.



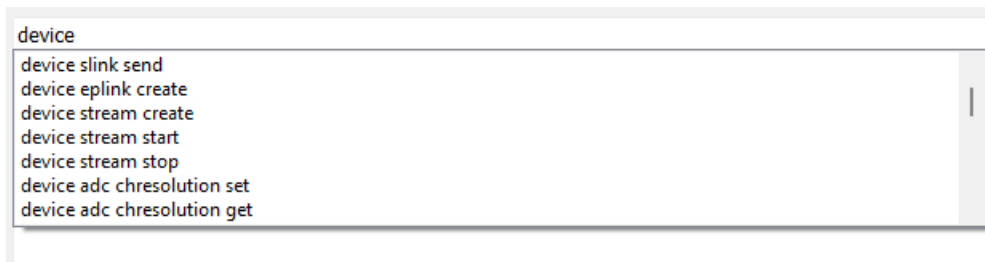


Figure 4.9 - Console autocomplete option

Internally, the console maintains a history of previously entered commands in the entries list. This history is navigable via the up and down arrow keys, with the logic implemented in the overridden *keyPressEvent* method. This behavior mimics traditional shell consoles, improving usability for power users who frequently reissue or tweak commands.

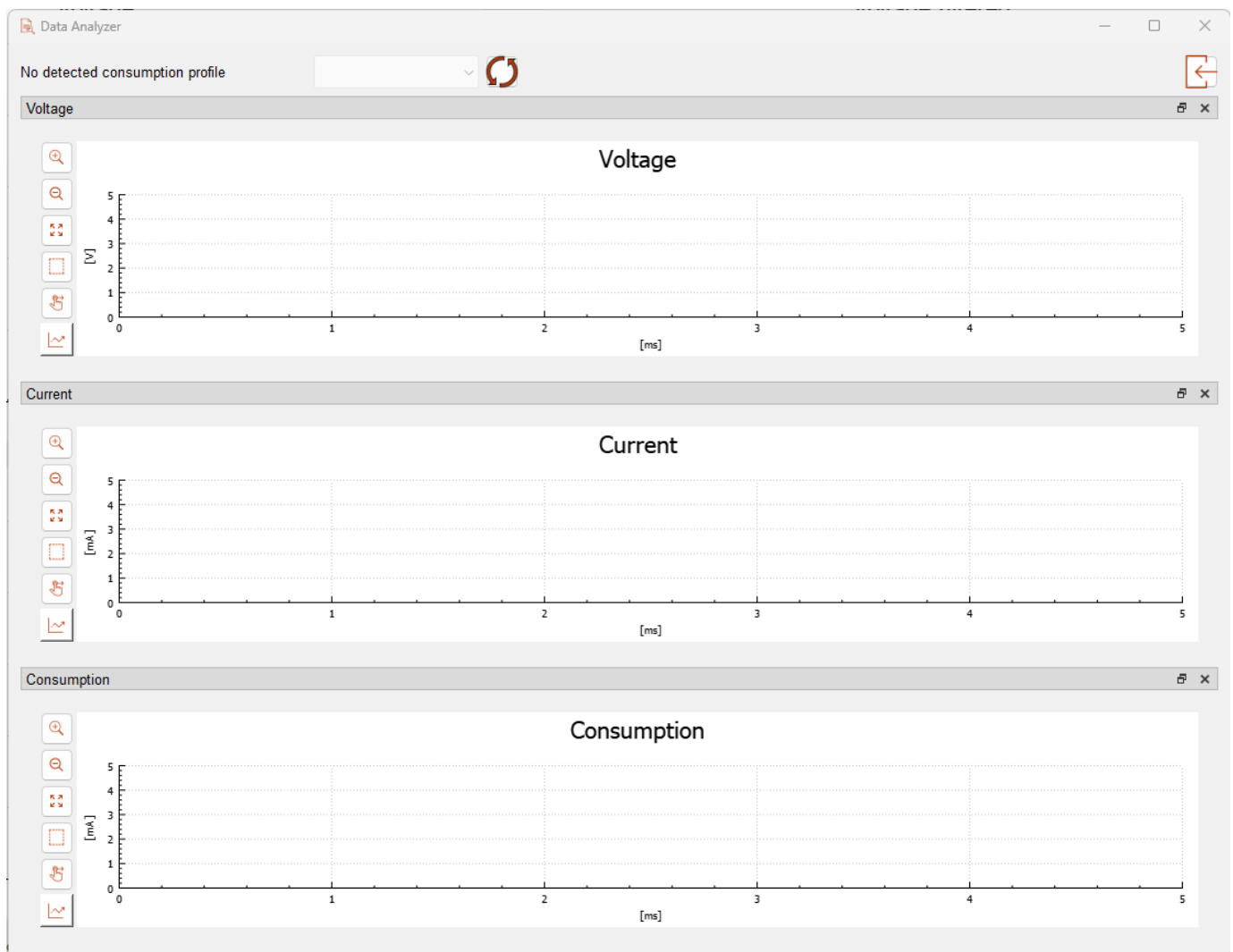
Messages exchanged with the device are logged using a *logUtil* utility. Responses are visually differentiated based on execution success, with informational messages shown for successful commands and error messages highlighted for failures. Incoming acknowledgment messages (e.g., "OK") are appended directly to the display widget.

Finally, the class emits the *sigControlMsgSend* signal whenever a new command is issued. This decouples the console from the device logic itself, allowing it to remain lightweight and focused solely on interaction.

### 4.4.3. Data Analyzer

BLOCK SUMMARY			
Name	Data Analyzer	Layer	Windows
Version	1.03		
Related files			
Top source file			
Source/Windows/DataAnalyzer/dataanalyzer.cpp			
Top header file			
Source/Windows/DataAnalyzer/dataanalyzer.h			
Top UI file			
Source/Windows/DataAnalyzer/dataanalyzer.ui			

The *DataAnalyzer* class in this implementation provides a flexible and interactive graphical component for analyzing recorded consumption profile data, particularly voltage, current, and derived consumption metrics. It is designed with usability and responsiveness in mind, integrating both user interface elements and background processing through a worker thread. Data Analyzer window layout is illustrated on **Figure 4.10**.



**Figure 4.10** - Data Analyzer Window Layout

Upon instantiation, the constructor sets up the graphical layout, including a toolbar with controls for selecting and reloading consumption profiles, and for initiating data analysis. It dynamically lists available profiles by scanning the working directory for subfolders containing a marker file (OpenEPT.txt). These



profiles are shown in a drop-down combo box, which is automatically refreshed when the reload button is pressed.

A unique feature of this class is its embedded *QMainWindow*, used solely to host dockable plotting windows for voltage, current, and consumption signals. Each plot is encapsulated in a custom Plot widget, with auto-scaling and descriptive axis labels. These plots can be rearranged and detached thanks to *QDockWidget* support, providing a highly modular and customizable user experience.

When a user triggers data processing, a separate *DataAnalyzerWorker* object, executing in a dedicated thread, is signalled to begin parsing data from CSV files corresponding to voltage/current (vc.csv), consumption (cons.csv), and optionally energy points (ep.csv). This parallel processing ensures that the user interface remains responsive even during heavy data loads.

Progress updates, including percentage complete and current processing stage, are communicated back to the UI via Qt's signal-slot mechanism. A modal progress dialog keeps users informed of the ongoing task without blocking application interaction.

Parsed data is returned as structured vectors and passed to the plotting components for immediate visualization. If the profile indicates that energy point data is available (based on metadata from OpenEPT.txt), the analyzer will proceed to load this information and overlay relevant annotations on all plots. This integration of consumption signal processing and energy event annotation gives the *DataAnalyzer* module significant analytical depth.



#### 4.4.4. Device

BLOCK SUMMARY			
Name	Device	Layer	Windows
Version	1.03		
Related files			
Top source file			
Source/Windows/Device/devicewnd.cpp			
Top header file			
Source/Windows/Device/devicewnd.h			
Top UI file			
Source/Windows/Device/devicewnd.ui			

The *DeviceWnd* class serves as a central control panel, providing the user interface and backend logic to configure, monitor, and manage a EPP. It is derived from *QWidget* and integrates a complex set of functionalities related to hardware interaction, signal visualization, and user-driven control over various parameters of the DAQ system. The widget handles user interactions through numerous buttons, combo boxes, checkboxes, and radio buttons, all of which are connected to corresponding slots using Qt's signal-slot mechanism. Device Window's layout is presented on Figure 4.11.

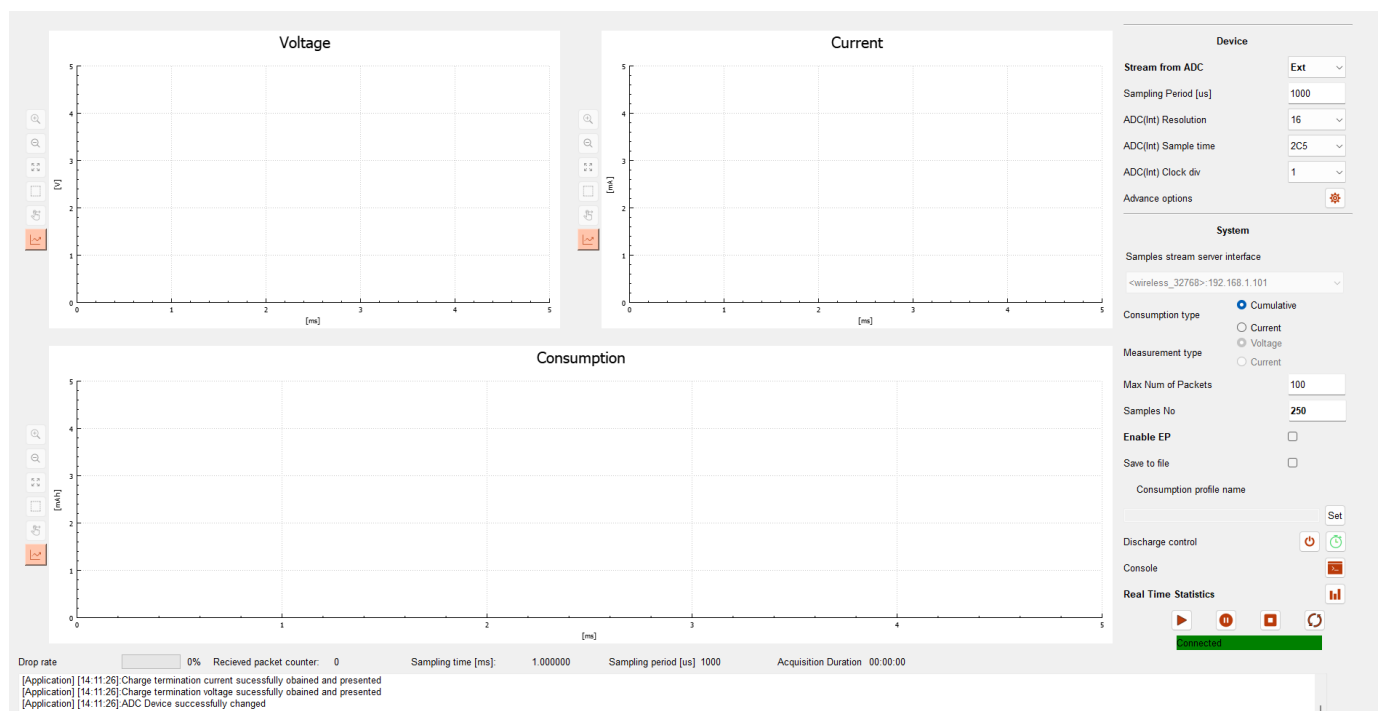


Figure 4.11 - Device Window Layout

During initialization, *DeviceWnd* constructs a wide variety of user interface components, including drop-down menus for selecting ADC sources, resolutions, sampling times, clock dividers, and averaging ratios. It also automatically detects active network interfaces using *QNetworkInterface*, populating a combo box to allow users to choose which interface will be used for streaming data. Alongside basic configuration, the window also instantiates and manages auxiliary subwindows such as *AdvanceConfigurationWnd*, *ConsoleWnd*, *CalibrationWnd*, and *DataStatistics*, each of which offers specialized functionality, like advanced ADC setup or real-time data analysis.

For visualization, *DeviceWnd* creates and embeds three Plot objects into the layout: one for voltage, one for current, and one for consumption. These plots are dynamically updated during acquisition and tailored according to the current measurement mode and data type (instantaneous or cumulative). They

provide both time-domain and frequency-domain representations, with additional support for marking key events or error boundaries using named data points. These plots are allocated minimum dimensions and are set to expand with the layout.

A significant part of the class is dedicated to managing stateful interactions with the hardware. Depending on the selected ADC mode ("Int" for internal or "Ext" for external), various fields are enabled or disabled, ensuring that the user is only allowed to modify parameters that are valid in the current mode. When users adjust settings such as resolution, clock dividers, or sampling periods, the new values are reflected not only in the GUI but also propagated to the *AdvanceConfigurationWnd* to ensure consistency across modules.

The *DeviceWnd* class also provides support for real-time acquisition control. It implements slots for handling actions like starting, pausing, stopping, and refreshing data acquisition. Each of these emits corresponding signals (*sigStartAcquisition*, *sigPauseAcquisition*, etc.) that connect to the data handling and device communication backend. Furthermore, it supports saving measurement results to disk. When this mode is enabled via a checkbox, it allows the user to enter a consumption profile name.

The class plays a key role in real-time interaction with the Acquisition Device Unite (ADU) hardware by managing various subsystems. It supports enabling/disabling hardware blocks like the load, battery path, and power path, as well as setting voltage/current offsets and DAC control. The class can update visual charging status (Charging, Discharging, Idle) in response to device feedback.

Internally, *DeviceWnd* maintains state variables like the current device state (connected, disconnected, undefined), interface selection status, measurement and consumption types, and acquisition mode. These internal states drive both GUI logic (e.g., disabling or enabling UI elements) and system behavior (e.g., switching chart titles and units). The class includes robust handling of device connection and disconnection, visually updating indicators in the GUI and resetting the state when needed.

To support debugging and control commands, *DeviceWnd* integrates a *ConsoleWnd* that allows users to send raw commands and view device responses. Received messages are logged in the console and can also be echoed to the main interface if needed. A calibration window is also embedded to allow users to interact with device-specific calibration data, and the *setCalibrationData* method ensures synchronization with the calibration engine.

### 4.4.5. Data Statistics

BLOCK SUMMARY			
Name	Data Statistics	Layer	Windows
Version	1.03		
Related files			
<i>Top source file</i>			
Source/Windows/Device/datastatistics.cpp			
<i>Top header file</i>			
Source/Windows/Device/datastatistics.h			
<i>Top UI file</i>			
Source/Windows/Device/datastatistics.ui			

The *DataStatistics* class is a *QWidget*-derived component designed to present summary statistics for key measurements, specifically voltage, current, and consumption data. It offers a compact and user-friendly interface where users can view the average, minimum, and maximum values associated with each of these signal types. This class plays a supporting role in the overall data analysis workflow by enabling real-time insights into signal quality and measurement trends without needing to manually inspect the raw plots. Data Statistics Window layout is presented on Figure 4.12.

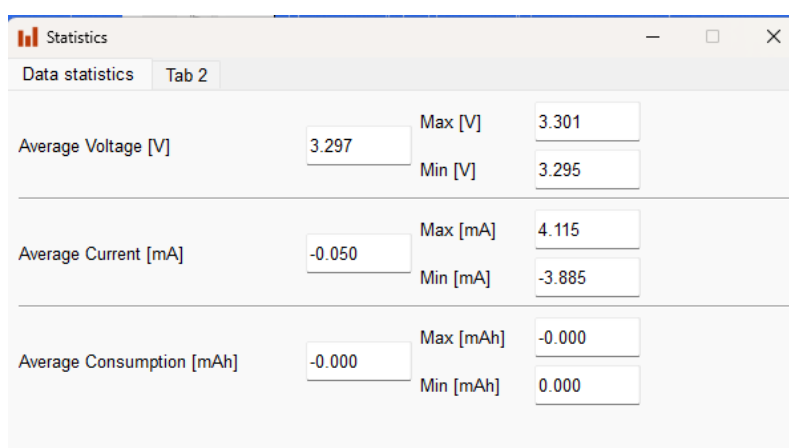


Figure 4.12 - Data Statistics Window Layout

Upon initialization, the constructor sets up the interface using a predefined UI file (*ui\_datastatistics.h*) and customizes the font to ensure consistency across all widgets. It builds a vertical layout that is anchored to the *AverageData* tab of a *QTabWidget*. Within this layout, the class creates dedicated horizontal sub-layouts for voltage, current, and consumption statistics, each comprising labelled fields for the average, maximum, and minimum values.

The voltage section is created first through the *createVoltageInfoLayout* method. It includes two vertical layout branches: one for the average voltage value and another for the maximum and minimum values. Each value is displayed in a read-only *QLineEdit* next to a descriptive *QLabel*, and spacing elements are used to ensure clean alignment and visual separation between columns.

A similar approach is applied to current data via the *createCurrentInfoLayout* method. It replicates the same structure but with labels and units specific to current (mA). Again, values are displayed using read-only *QLineEdit* widgets to ensure the interface remains non-editable, reflecting its analytical, display-only nature. This consistent layout design ensures that users can easily compare structure and values across signal types without confusion.

The third and final data section is dedicated to consumption, created with *createConsumptionInfoLayout*. Here, the fields are labelled with units in milliamperes-hours (mAh),



reflecting the cumulative nature of the consumption signal. This section includes similar components—*QLabels*, *QLineEdits*, and spacers, ensuring visual consistency with the voltage and current sections.

Between each major layout section, horizontal separator lines are added using *QFrame* widgets configured with a sunken style. These lines visually segment the data blocks, improving readability and organization. The lines are wrapped in *QHBoxLayouts* and inserted between sections in the main vertical layout.

The class provides three public setter methods, *setVoltageStatisticInfo*, *setCurrentStatisticInfo*, and *setConsumptionStatisticInfo*, to populate the fields with the calculated statistics. Each method takes average, maximum, and minimum values as arguments and updates the corresponding line edit fields using formatted floating-point strings with three decimal places of precision.

The layout is adjusted to a fixed size of 600x300 pixels to provide a compact but clear view of all relevant information. Additional UI tweaks are made using Qt's stylesheet system to remove unnecessary background decorations from the tab widget and its pane.

## 4.4.6. Calibration

BLOCK SUMMARY			
Name	Calibration	Layer	Windows
Version	1.03		
Related files			
<i>Top source file</i>			
Source/Windows/Device/calibrationwnd.cpp			
<i>Top header file</i>			
Source/Windows/Device/calibrationwnd.h			
<i>Top UI file</i>			
Source/Windows/Device/calibrationwnd.ui			

The *CalibrationWnd* class provide controls for configuring and updating calibration parameters used in the ADU. This window provides the user with a convenient way to manually enter or revise correction and gain values that are essential for accurate electrical measurements such as voltage, current, and consumption. This windows layout is presented on **Figure 4.13**.

**Figure 4.13** - Calibration Window Layout

When the window is shown via the *showWnd* method, it populates a set of editable fields with the current values stored in a *CalibrationData* structure. These include: the ADC voltage reference, current correction factor, current gain, current shunt resistance, voltage offset, voltage correction factor, and voltage-to-current offset. This structure is passed to the class using the *setCalibrationData* method, which stores a pointer to the externally managed *CalibrationData* instance, allowing the window to act directly upon the live configuration data used elsewhere in the application.

Upon pressing the “Submit” button, the window reads the content of each input field and updates the underlying *CalibrationData* structure with the new user-provided values. Once the updated values are committed, the class emits a *sigCalibrationDataUpdated* signal, notifying any connected components (such as data processors or configuration savers) that new calibration data is now in effect. This mechanism allows for a decoupled and responsive integration of UI and backend logic.

These calibration parameters are directly consumed by the *DataProcessing* class, particularly within the *onNewSampleBufferReceived* function, which is responsible for converting raw ADC data into real-world physical quantities. For example, the *adcVoltageRef* is used to calculate the voltage increment per ADC unit, while *currentShunt* and *currentGain* are involved in transforming raw current readings into calibrated current values in milliamps. The *voltageOff* parameter compensates for static voltage offsets, and *voltageCurrOffset* allows correction of voltage readings influenced by current flow.

The formula for calculating current reflects several layers of calibration logic. The raw ADC value is scaled by a gain and shunt resistance, then adjusted with the current correction factor to account for known deviations or hardware inaccuracies. This ensures that small mismatches in hardware (e.g., non-ideal resistor





values or amplifier gains) are corrected in software, yielding high-quality, consistent results regardless of board-to-board variation. Equations (E.1) and (E.2) are used to calculate final voltage value.

$$currentInc = \left( \frac{V_{ref}}{2^{Res}} \right) \quad (E.1)$$

$$I = \left( \frac{rawCurrent \cdot currentInc - V_{offset\_current}}{R_{shunt} \cdot G_{amp}} \right) \cdot 1000 \cdot K_{current} \quad (E.2)$$

Similarly, voltage data is compensated using a combination of offset and linear scaling based on the supplied correction values. The calibrated voltage is then checked against minimum and maximum values to support statistical analysis, which will later be reported or visualized. The processed values are stored in circular buffers and used to derive instantaneous as well as cumulative statistics like average voltage, peak current, and total energy consumption. Equations (E.3) and (E.4) are used to calculate final voltage value.

$$voltageInc = \left( \frac{V_{ref}}{2^{Res}} \right) \cdot K_{voltage} \quad (E.3)$$

$$V = V_{offset} + rawVoltage \cdot voltageInc \quad (E.4)$$

The calibration has a significant impact on downstream processing. For instance, the accuracy of cumulative consumption measurements (expressed in mAh) depends on how precisely the current readings are corrected and integrated over time. Inaccurate gain or shunt values would distort the integration result, leading to underestimated or overestimated energy consumption. This is particularly critical in battery-driven applications, where fine-grained consumption data is vital for energy management.

## 5. BUILD AND RUN INSTRUCTIONS

To successfully build and run the OpenEPT application, the Qt development environment must be installed and configured with the appropriate dependencies. The following steps guide you through the full setup process.

### Step 1: Download and install Qt and relevant packages

The Qt framework should be obtained from the official [Qt website](https://www.qt.io). Once the installer is downloaded and executed, users will be prompted to complete the sign-up process. Following successful registration, the installer will proceed to the component selection stage. For proper compilation and execution of the GUI application, the following components must be selected:

- |                      |   |   |
|----------------------|---|---|
| <b>Qt</b>            | - Qt 5.15.2   | - MSVC 2019-64bit<br>- Qt Debug Information Files |
| <b>- Build Tools</b> | - Qt Creator <latest version><br>- Qt Creator <latest version> CDB Debugger Support<br>- Debugging tools for Windows<br>- Qt Creator <latest version> Debug Symbols<br>- Qt Creator <latest version> Plugin Development<br>- CMake 3.30.5 |   |

The Figure 5.1 illustrates the selected components that must be enabled before proceeding to the next step in the installation process. These selections ensure that all necessary libraries and tools required for building and running the GUI application are properly installed.

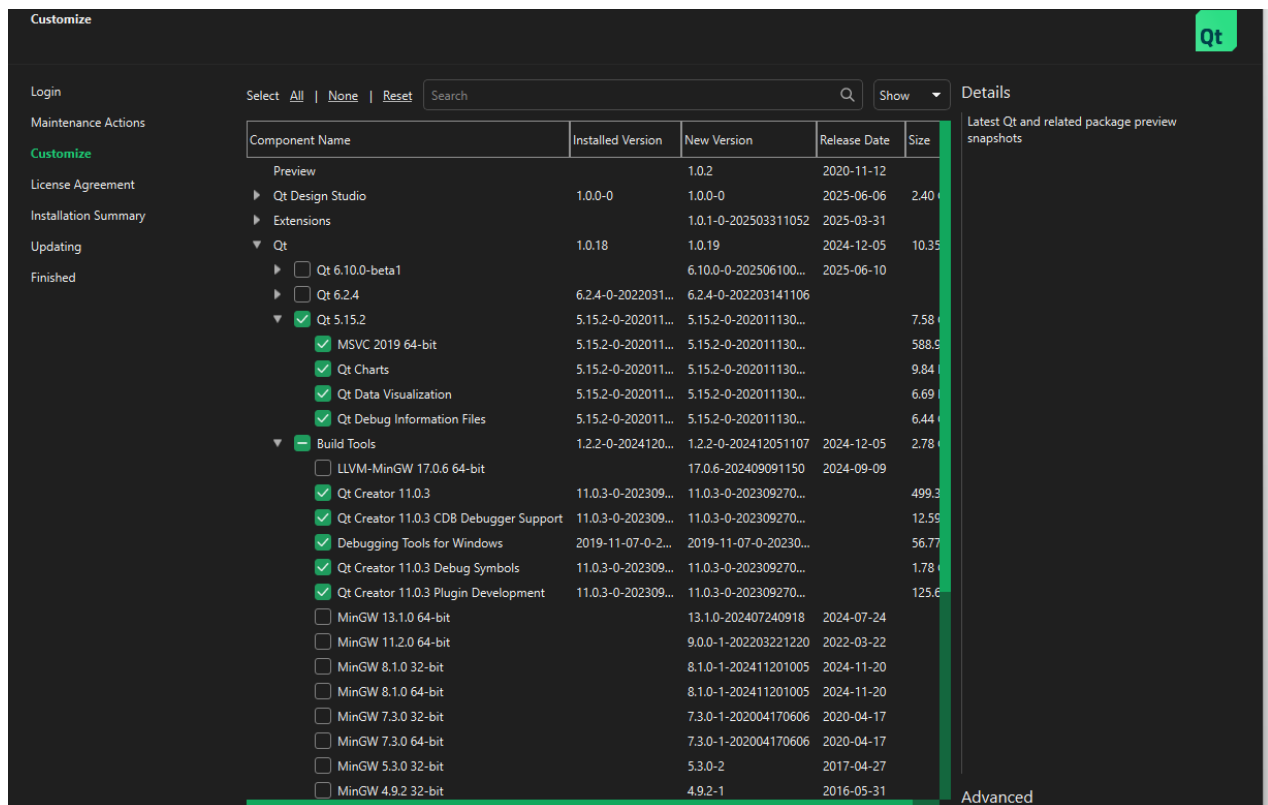


Figure 5.1 - Qt Installer Component Selector with selected components required for OpenEPT GUI application

If Qt has been successfully installed, the application named **Qt Creator** should be launchable without issues. Upon starting Qt Creator, Welcome page will be displayed, which provides quick

access to recent projects, examples, tutorials, and version control options. This interface should resemble what is shown in the corresponding Figure 5.2.

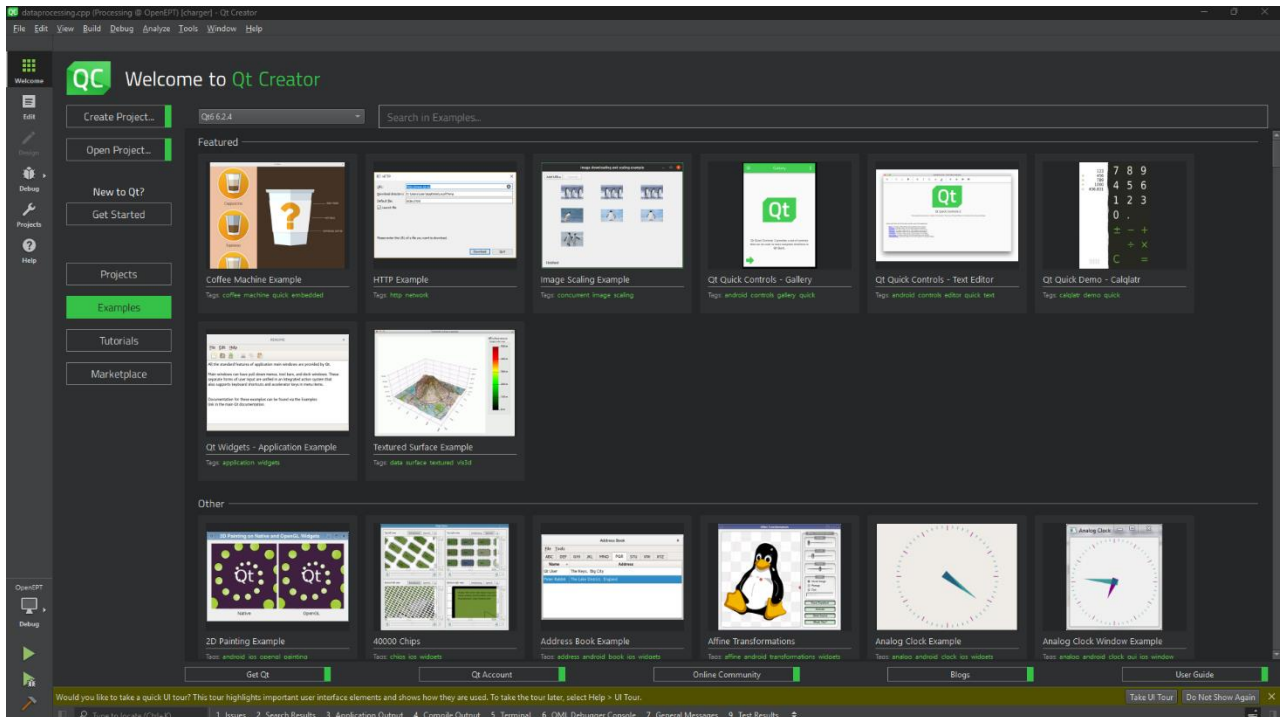


Figure 5.2 - Qt Creator Welcome Page

## Step 2: Clone project from official GitHub account

There are two primary methods for downloading the project from the [official GitHub repository](#). The first method is using a dedicated Git console, such as [Git Bash](#), which allows cloning the repository directly via the command line. The second method involves navigating to the repository's GitHub page and clicking the green Code button, then selecting the **Download ZIP** option. This will download the project files as a compressed archive. The latter method is visually illustrated in the corresponding Figure 5.3.

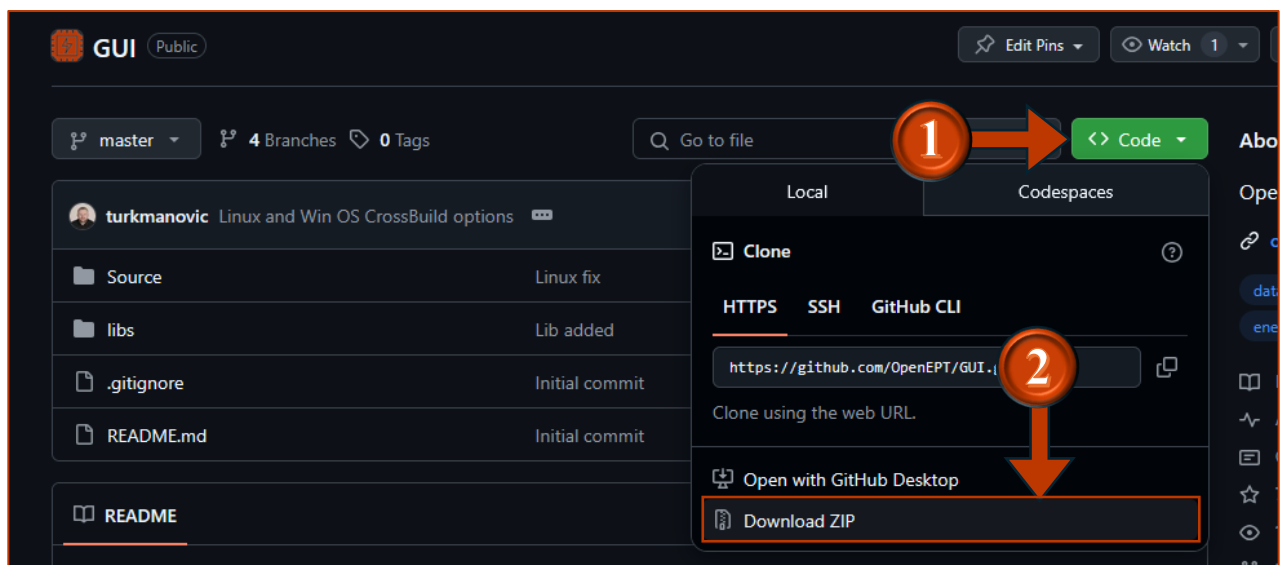


Figure 5.3 - Download project from official GitHub repository

### Step 3: Import and configure project inside QtCreator

After successfully launching Qt Creator, the first step is to open the project. On the left-hand sidebar, click on the *Welcome* tab (1) to access the start page. From there, click the *Open Project* button (2). These two steps are presented on Figure 5.4.

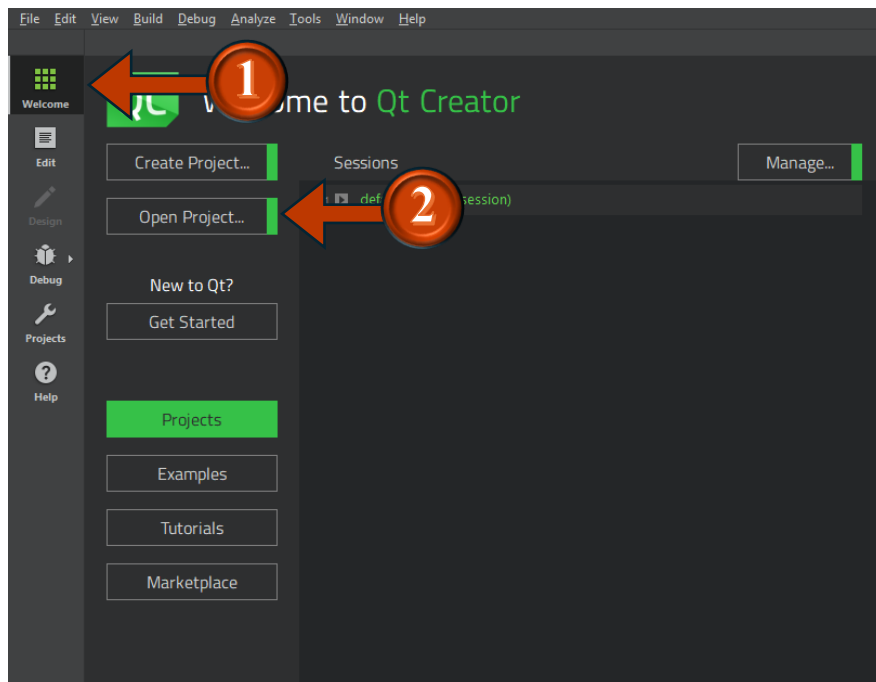


Figure 5.4 – Welcome page and Open project button

After clicking the Open Project button, a file browser window will appear, allowing you to navigate to the directory where the OpenEPT project was downloaded. Inside this directory, locate the main project file with the .pro extension (*Source/OpenEPT.pro*). Selecting this file and confirming the action will prompt Qt Creator to load the project. Open File dialog with selected project's .pro file is presented on Figure 5.5.

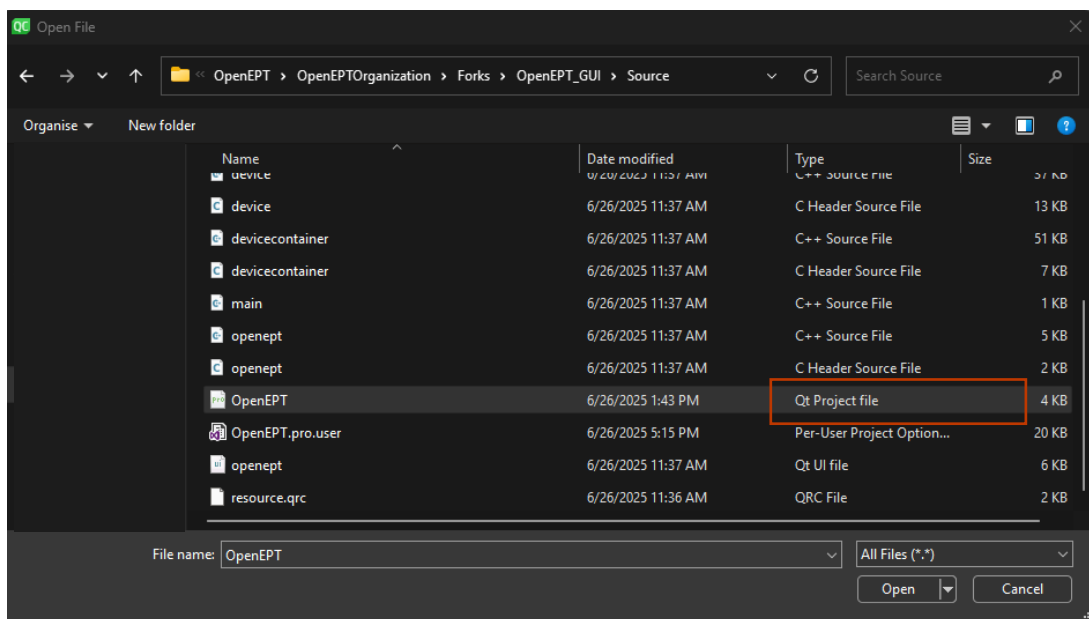


Figure 5.5 - Navigate to .pro file within Open File dialog

When the project is successfully loaded for the first time, Qt Creator will automatically open the initial project configuration dialog. In this step, you are required to choose an appropriate build kit and associated settings, which define how the project will be compiled and run. For the OpenEPT project, it is essential to select the kit labelled "**Qt 5.12.2 MSVC 2019 64-bit**", as this version ensures

compatibility with the codebase and required libraries. Selecting the correct kit at this stage is critical to avoid compilation errors and ensure stable application behavior. Once the correct kit is selected, just as illustrated in Figure 5.6, click on *Configure Project* to proceed. This action finalizes the setup and prepares the development environment for building and running the OpenEPT application.

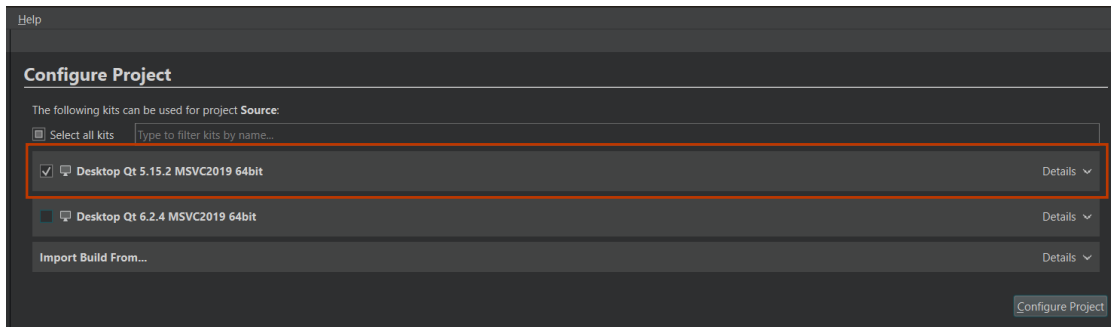


Figure 5.6 - Development Kit Selection Prompt Upon First Project Load

### Step 4: Build and Run the project

Once the project has been successfully loaded into Qt Creator, the next step involves selecting the desired build configuration (1). At the bottom-left of the Qt Creator interface, users can choose between Debug and Release modes. The Debug configuration is typically used during development for troubleshooting and includes additional debugging symbols, while the Release configuration is optimized for performance and is generally used for deployment.

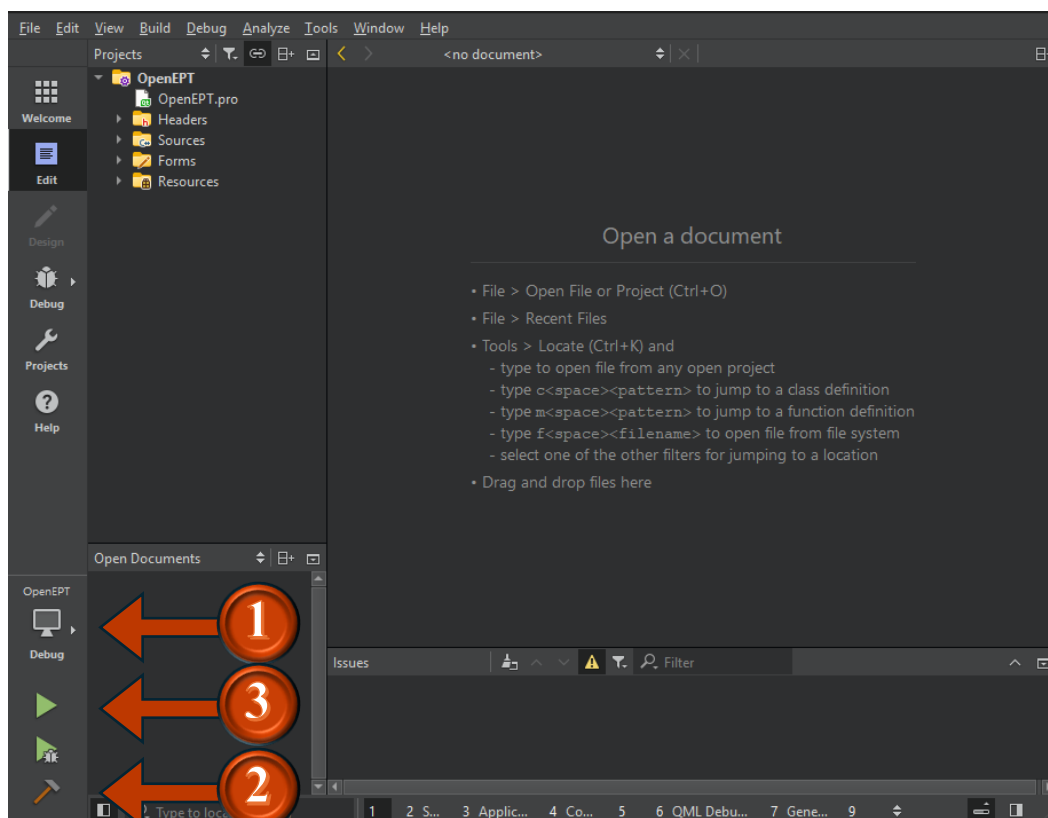


Figure 5.7 - Select Build configuration, Build and Run

After selecting the desired build configuration (either Debug or Release), the next step is to compile the project (2). This is achieved by clicking the *Build* button (represented by a hammer icon) located in the lower-left corner of the Qt Creator interface, or by navigating through the top menu via *Build* > *Build Project*. Qt Creator will then begin compiling all necessary project files according to the selected configuration and the chosen development kit.



Once the build process completes successfully and without errors, the application can be launched (3). If the Debug configuration is selected, the application should be started by clicking the *Run* button that includes a small briefcase icon, which initiates a full debugging session. For a Release configuration, the standard *Run* button (green play icon) next to the configuration selector should be used, or alternatively via *Build > Run* from the menu.



## LIST OF FIGURES

<b>FIGURE 2.1</b> – ACQUISITION DEVICE’S FIRMWARE .....	5
<b>FIGURE 2.2</b> - DEVICE CONTAINER'S ELEMENTS .....	5
<b>FIGURE 3.1</b> - OPENEPT ORGANIZATION ON GITHUB AND GUI REPOSITORY .....	7
<b>FIGURE 3.2</b> – OPENEPT SOURCE DIRECTORY STRUCTURE.....	8
<b>FIGURE 4.1</b> - OVERVIEW OF THE OPENEPT GUI CLASSES THAT PROCESS AND VISUALIZE ENERGY POINTS.....	14
<b>FIGURE 4.2</b> - SIGNALS INSIDE DATAPROCESSING CLASS.....	17
<b>FIGURE 4.3</b> - PARSING STREAM MESSAGE INSIDE STREAM LINK .....	18
<b>FIGURE 4.4</b> - EXECUTECOMMAND METHOD DEFINITION.....	20
<b>FIGURE 4.5</b> - ONNEWCONNECTIONADDED SLOT DEFINITION .....	21
<b>FIGURE 4.6</b> - INITIAL PROCESSING OF ENERGY DEBUGGING MESSAGES INSIDE EDLINK CLASS .....	22
<b>FIGURE 4.7</b> - ADD DEVICE WINDOWS LAYOUT.....	23
<b>FIGURE 4.8</b> – CONSOLE WINDOW LAYOUT.....	24
<b>FIGURE 4.9</b> - CONSOLE AUTOCOMPLETE OPTION .....	25
<b>FIGURE 4.10</b> - DATA ANALYZER WINDOW LAYOUT.....	26
<b>FIGURE 4.11</b> - DEVICE WINDOW LAYOUT.....	28
<b>FIGURE 4.12</b> - DATA STATISTICS WINDOW LAYOUT.....	30
<b>FIGURE 4.13</b> - CALIBRATION WINDOW LAYOUT.....	32
<b>FIGURE 5.1</b> - QT INSTALLER COMPONENT SELECTOR WITH SELECTED COMPONENTS REQUIRED FOR OPENEPT GUI APPLICATION .....	34
<b>FIGURE 5.2</b> - QT CREATOR WELCOME PAGE .....	35
<b>FIGURE 5.3</b> - DOWNLOAD PROJECT FROM OFFICIAL GITHUB REPOSITORY .....	35
<b>FIGURE 5.4</b> – WELCOME PAGE AND OPEN PROJECT BUTTON.....	36
<b>FIGURE 5.5</b> - NAVIGATE TO .PRO FILE WITHIN OPEN FILE DIALOG .....	36
<b>FIGURE 5.6</b> - DEVELOPMENT KIT SELECTION PROMPT UPON FIRST PROJECT LOAD .....	37
<b>FIGURE 5.7</b> - SELECT BUILD CONFIGURATION, BUILD AND RUN.....	37



## REFERENCES

- [1] OpenEPT, *Energy Profiler Probe – Firmware Developer Guide*, 2025
- [2] [Turkmanović, H.; Karličić, M.; Rajović, V.; Popović, I. High Performance Software Architectures for Remote High-Speed Data Acquisition. \*Electronics\* \*\*2023\*\*, \*12\*, 4206.](#)
- [3] [OpenEPT Organization Code Repository, GitHub.](#)