# Data Acquisition and Scripting Documentation

# Introduction

A data acquisition capability is a requirement for the TEDDI project. 2Easy initially provided the TEDDI visualisation capabilities for data collected from the Hexitec detectors. 2Easy has been extended to implement flexible data acquisition and scripting environments.

# Configuration

Various parameters that control the behaviour of 2Easy at runtime can be configured using the Qt application Configure2Easy. Parameters can be set as follows via the interface shown in Figure 1. Once set up save the settings via the Settings menu. This menu also gives the ability to delete all settings if required. On a windows system the settings are probably stored in the system registry.

* MATLAB : if MATLAB is available on the host system a MATLAB engine will be started. Access to this is made available from the 2Easy scripting environment. By default 2Easy assumes MATLAB isn’t available.
* Up to eight motors can be controlled from 2Easy, as listed in Figure 1. These can be Dummy motor or accessed via Galil motion controllers. Dummy motors simply appear to move. By default 2Easy does not create any motors.

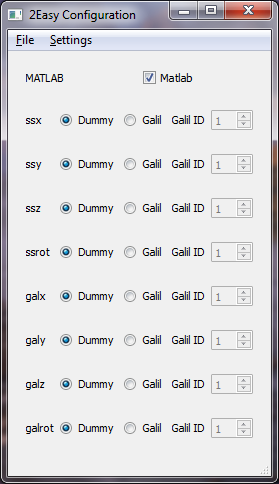


Figure 1: The Qt application ConfigureEasy

# Software Development

Visualisation and data acquisition are distinct processes. 2Easy has a single panel providing the visualisation capability. For clarity and ease of use a tabbed interface was introduced to separate the user input and implementation of the different processes.

Any data acquisition capability must be easy to use and flexible. Users have a number of different requirements ranging from basic well defined procedures to complex and as yet unknown requirements. The most appropriate way to address these different requirements is to provide a basic data acquisition capability and a scripting capability to allow users to define their own data acquisition procedures.

## Tabbed Interface

Three tabbed panes have been created in 2Easy as follows:

* Visualisation
* Data Acquisition
* Scripting

## Data Acquisition

The first priority for the data acquisition capability is to be able to control motors, but the system can be extended to control other types of hardware required for data acquisition (e.g. detectors, sensors, etc.).

It is the responsibility of the DataAcquisitionFactory class to create the objects required for hardware control. The DataAcquisitionFactory creates a MotorFactory which then creates the motors used for data acquisition and adds them to the list of Motors in the MotorModel. Both factories are implemented as singletons. Currently the TEDDI experiment uses Galil motors to control 8 axes. Further axes, not necessarily Galil, may be added to the system in the future. It is also necessary to be able to develop code without motor hardware being available so a motor simulator (DummyMotor) has been developed.

To enable this flexibility the Motor class is an abstract class. The Motor interface is implemented by the concrete classes DummyMotor and GalilMotor. Motor provides the following interface:

* *Q\_INVOKABLE virtual void stop() = 0;*
* *Q\_INVOKABLE virtual void begin();*
* *Q\_INVOKABLE virtual void reset();*
* *Q\_INVOKABLE void setDesiredMove(int move, int mode);*
* *Q\_INVOKABLE int getPosition();*
* *Q\_INVOKABLE QString getStatusString();*
* *Q\_INVOKABLE bool isMoving();*
* *virtual void run() = 0;*
* *virtual void configure() = 0;*
* *void setMoveInput(int moveInput, bool doEmit);*
* *int getMoveInput();*
* *void setMode(int mode, bool doEmit);*
* *int getMode();*
* *void setSeletected(bool selected);*
* *bool isSelected();*
* *bool willGo(bool doLock);*
* *bool release(QString releaser);*

Other concrete motor classes can be created by extending motor and implementing the virtual methods provided by the Motor class. The following virtual methods must be implemented by any concrete class which inherits Motor:

* *Q\_INVOKABLE virtual void stop() = 0;*

Stops the move in progress.

* The protected method *virtual void begin(int desiredPosition) = 0;*

Begins a motor move.

* *virtual void run() = 0;*

Executed in a separate thread to monitor motor position and status.

* *virtual void configure() = 0;*

Set and initial configuration parameters required by a motor (eg. name, speed, etc.).

The GalilMotor class interfaces to the Galil motors through the GalilMotorController which uses the GalilTools STL library (C/C++) to connect to and communicate with the motors over a network connection. GalilMotorController is the only point of access to the hardware and is implemented as a Singleton. This is an initial implementation of a GalilMotor class which doesn’t provide all the functionality available, more development is required.

The basic data acquisition tab, see Figure **2**, provides the capability to select a number of motors to move in absolute or relative step units and reports the resulting position and status.

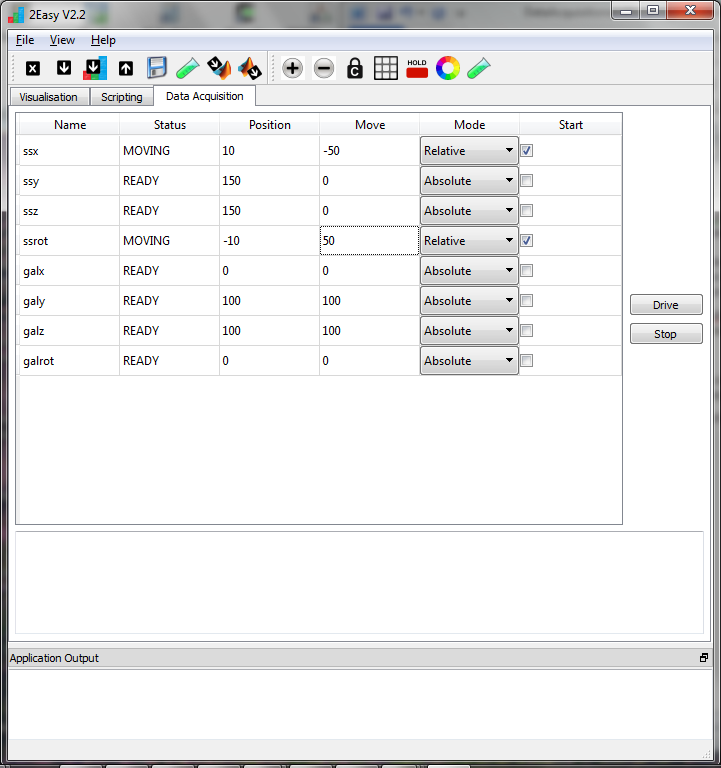


Figure 2: 2Easy Data Acquisition tab

The data acquisition tab has been created using the QT Creator designer to layout the widgets and produce dataacquisitionform.ui. Running qmake will generate the appropriate C code.

This screen can be extended to provide a complete range of facilities required to drive motors and provide the basic data acquisition capability. Adding new types of motor to the system will only require a new concrete class to be written to implement the low level motor control.

Currently the configuration of motors in the DataAcquisitionFactory (eg. name, speed, etc) is hardcoded. A future enhancement will be to make this parameterised via Configure2Easy.

## Scripting

Providing a scripting interface and making objects available in the scripting environment enables users to create more complex procedures to control their experiment.

The scripting tab, see Figure 3, provides the capability to type scripts in to the editor, load, execute and save a script. Scripts can be swapped between MATLAB and the ECMAScript scripting environment implemented by Qt. This can be swapped via a background menu on the script editor or the shortcut keys Alt-M and Alt-S in that editor. ECMA and MATLAB cannot currently be mixed in a single script. A single line script command window is also available to test script commands; script command history is available through the up and down keys in this window.

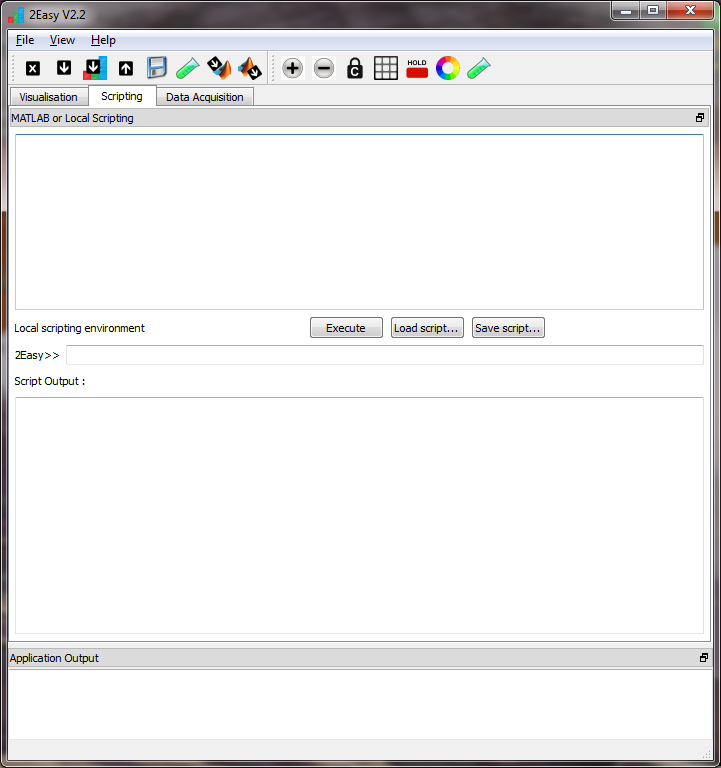


Figure 3: 2Easy Scripting tab

Any classes created in 2Easy can be made available to the scripting environment. The following is an example of the mechanism as for a motor created in 2Easy:

* The class Motor extends QThread and hence QObject in the following way:
  + *#include <QThread>* (motor.h)

*#include “reservable.h”*

*class Motor : public QThread, public Reservable*

*{*

*Q\_OBJECT*

*public:*

*explicit Motor(QObject \*parent = 0);*

* Any methods which need to be available to the scripting environment are declared as Q\_INVOKABLE. In the case of Motor these methods may be pure virtual where they have to be implemented in concrete implementations.
  + *Q\_INVOKABLE virtual void stop() = 0;* (motor.h)

*Q\_INVOKABLE virtual void begin(int desiredPosition) = 0;*

*Q\_INVOKABLE void setDesiredMove(int move, int mode);*

*Q\_INVOKABLE int getPosition();*

*Q\_INVOKABLE bool isMoving();*

*Q\_INVOKABLE QString getStatusString();*

* The ScriptRunner class has a slot addObject used to connect Objects to it.
* Motors are created in MotorFactory. MotorFactory emits a signal addObject to make each motor known to the ScriptRunner object and thus available to ECMA scripts.
  + *void addObject(QObject \*object);* (motorfactory.h)
* The connection between the signal (MotorFactory) and slot (ScriptingWidget) is made as follows:
  + *connect(this, SIGNAL(addObject(QObject\*)), ScriptingWidget::getScriptRunner(), SLOT(addObject(QObject\*)));* (MotorFactory constructor)
* An object name for use by the scripting environment is set.
  + *motor->setProperty("objectName", name);* (MotorFactory::createMotor())
* A signal is emitted to make the Object available in the scripting environment.
  + *emit addObject(motor);* (MotorFactory::createMotor())

Other objects can be made available to the scripting environment in the same way.

Scripts are executed via standard Qt event handling. Scripts that involve long delays (e.g. data processing or waiting for motors to finish) would interrupt event handling and stop the GUI responding. Therefore scripts are run in a separate thread. As a result of this there are two important consequences that need to be taken into account in making objects available to scripting:

* Scripts cannot use QT classes to access 2Easy GUI elements directly, signals should be used.
* Methods of classes that need to be accessed from script should be thread safe.

The implementations of motors in 2Easy should provide examples of how to deal with these.