



## **University of Huddersfield Repository**

Morris, Ezra

The Design and Deployment of Cross-Platform User Interface for the Analysis, Control and Management of Disparate Embedded Systems

### **Original Citation**

Morris, Ezra (2015) The Design and Deployment of Cross-Platform User Interface for the Analysis, Control and Management of Disparate Embedded Systems. Masters thesis, University of Huddersfield.

This version is available at <http://eprints.hud.ac.uk/id/eprint/29199/>

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: [E.mailbox@hud.ac.uk](mailto:E.mailbox@hud.ac.uk).

<http://eprints.hud.ac.uk/>

THE DESIGN AND DEPLOYMENT OF  
CROSS-PLATFORM USER  
INTERFACE FOR THE ANALYSIS,  
CONTROL AND MANAGEMENT OF  
DISPARATE EMBEDDED SYSTEMS

EZRA PETER CHRISTOPHER MORRIS

A thesis submitted to the University of Huddersfield in partial fulfilment of  
the requirements for the degree of Master of Science by Research

The University of Huddersfield

December 2015

## Copyright statement

- i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and s/he has given The University of Huddersfield the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.
- ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the University Library. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made
- iii. The ownership of any patents, designs, trademarks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any reproductions of copyright works, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

## Declaration of prior work

The author of this thesis hereby declares that this work builds upon work previously completed by the author, namely “Real-Time Graphical Analysis of RS-232 Serial Data Using Computer Software” as part of the award of BEng (Hons) Electronic Engineering at Sheffield Hallam University.

The software output of this work, including software architecture, graphical user interface (GUI) and application programmable interface (API) was designed independently of the original work, to meet the aims and objectives of this project. However, portions of the source code from the original application were re-used and adapted during implementation of this work. The features of the original application are listed in section 2.2.1.5, and the changes made are clearly identified in sections 3 and 4.

With regards to the software applications produced, references to “original SerialPlot”, “original application”, “existing software” etc. refer to the prior work. Specifically, sections 2.2.1.5 and 3.3 refer exclusively to the original application. The application created is referred to as “the application” throughout this document, however references to “SerialPlot” may be observed in screenshots and code from the application, as it was still referred to as “SerialPlot” internally.

## Abstract

Embedded computers contribute to almost every aspect of 21<sup>st</sup> century living, from communication and transport to utilities and manufacturing. With such a prevalence of these systems, the need to efficiently monitor, analyse and control them is paramount. Often a bespoke software application is used, but by using a general purpose application, less resources are required. Several commercial and free software utilities are available which can be used to meet this need to some extent, however most are designed for specific use-cases or contain a number of deficiencies which makes their use difficult.

As such, a graphical user interface (GUI) application was produced which aimed to meet this need. The Python programming language was utilised alongside the Qt framework, resulting in a cross-platform application. The software package was designed to be modular, allowing parts of the application to be re-used when a full-featured GUI was not required. A key feature of the application was the ability to define data protocols to extract individual data fields from a stream of data, and construct output data from user-entered values. In addition, a flexible GUI was created, allowing the user to display the data in various formats, such as textual or graphical, simultaneously.

The application produced made good strides in meeting this need, and received positive feedback from users. Some minor performance improvements were identified, however the application generally performed smoothly and efficiently. It is envisaged that through increased usage of suitable, performant and intuitive generic tools such as the one produced, the efficient use of human, time and financial resources used for developing and monitoring embedded systems can be greatly improved.

# Contents

1	Introduction .....	15
1.1	Project aims and objectives .....	16
2	Background .....	18
2.1	Serial communications .....	18
2.2	Existing solutions .....	19
2.2.1	RS-232 solutions .....	19
2.2.1.1	Basic applications .....	19
2.2.1.2	Realterm .....	20
2.2.1.3	Serialtest .....	22
2.2.1.4	qSerialTerm .....	24
2.2.1.5	SerialPlot .....	26
2.2.2	Other solutions .....	29
2.2.2.1	Wireshark .....	29
2.2.2.2	USBlyzer .....	31
2.2.3	Comparison .....	31
3	System design .....	34
3.1	Programming language .....	34
3.2	A modular approach .....	37
3.3	Deficiencies of SerialPlot .....	38
3.3.1	GUI design .....	38
3.3.2	Graph issues .....	38
3.3.3	Application state .....	39
3.3.4	Data transmission inflexibility .....	40
3.3.5	Extendibility .....	40
3.4	Features .....	40
3.4.1	User interface concept .....	40
3.4.2	Textual display .....	43
3.4.3	Graphing .....	44
3.4.4	File capture .....	45
3.4.5	Protocol editor .....	46

3.5	Tools .....	47
3.5.1	Serial port access .....	47
3.5.2	Graphical user interface .....	47
3.5.3	Mathematical parsing .....	51
4	Implementation .....	53
4.1	Program structure .....	53
4.1.1	File structure .....	53
4.1.2	Application logic .....	54
4.1.3	Hardware managers .....	55
4.1.3.1	Serial manager .....	55
4.1.4	The data handler .....	56
4.1.4.1	Data storage .....	57
4.1.4.2	Data counter .....	59
4.1.5	Protocols .....	59
4.1.5.1	Protocol handlers .....	60
4.1.5.2	Protocol objects .....	62
4.2	Graphical user interface .....	63
4.2.1	Main window .....	63
4.2.2	Views .....	65
4.2.2.1	Base classes .....	65
4.2.2.2	Basic views .....	66
4.2.2.3	Textual display views .....	68
4.2.2.4	Graphing .....	70
4.2.2.5	File capture .....	72
4.2.3	Data transmission .....	73
4.2.4	Protocol editor .....	75
4.2.5	Data persistence .....	76
5	Outcomes .....	77
5.1	Final application .....	77
5.1.1	Menus .....	78
5.1.2	Error messages .....	79
5.2	Usage example .....	80

5.3	Command line interface example .....	82
5.4	Performance issues .....	83
5.4.1	Bit manipulation efficiency .....	83
5.4.2	Protocol validation efficiency .....	83
5.4.3	Programming language.....	84
6	Conclusion .....	85
7	References .....	86
Appendix A	Application directory structure.....	88
Appendix B	Serial receive tests.....	89
	Buffer overflow testing .....	90
Appendix C	Example View code .....	92
Appendix D	Example workspace file.....	93
Appendix E	Application main window .....	94
Appendix F	CLI example .....	95
Appendix G	Program code .....	97
	__init__.py .....	97
	dataformats.py .....	97
	datahandler.py .....	99
	file_capture.py .....	102
	protocols.py .....	104
	serialplot_gui.py .....	110
	settings.py .....	111
	gui/__init__.py .....	113
	gui/docks.py .....	113
	gui/mainwindow.py.....	119
	gui/prefs_dialog.py .....	124
	gui/prefs.py .....	129
	gui/protocol_dialog.py .....	132
	gui/protocol_editor.py .....	135
	gui/protocol_widgets.py .....	139
	gui/resources.py.....	143
	gui/ui_state.py.....	143

gui/util.py .....	144
gui/views/__init__.py.....	144
gui/views/console.py .....	145
gui/views/current_value.py .....	147
gui/views/example_view.py .....	148
gui/views/file_capture.py.....	149
gui/views/graph.py .....	151
gui/views/subwindow_multifield.py .....	155
gui/views/subwindow.py.....	157
gui/views/table.py .....	160
hw/__init__.py.....	162
hw/manager.py .....	162
hw/serialmanager.py .....	163

Word count, excluding appendices: 18347

## List of tables

Table 1: Communication analyser product comparison .....	32
Table 2: Comparison of programming languages.....	35
Table 3: GUI framework features.....	51
Table 4: Key protocolHandler methods .....	61

# List of figures

Figure 1: Example transmission of 1 byte over RS-232 (Ktnbn & Tardieu, 2009) .....	19
Figure 2: Realterm main window .....	21
Figure 3: Serialtest interface.....	24
Figure 4: qSerialTerm application.....	25
Figure 5: Original SerialPlot application, showing graph tab and protocol selection areas .....	27
Figure 6: SerialPlot's protocol editor .....	29
Figure 7: Wireshark main window .....	30
Figure 8: SerialPlot graph plotting errors.....	39
Figure 9: Comparison of MDI (left) and SDI interfaces.....	41
Figure 10: Concept GUI layout.....	43
Figure 11: Console view concept.....	44
Figure 12: Table view concept .....	44
Figure 13: Graph view concept .....	45
Figure 14: IDLE settings dialog .....	48
Figure 15: Application architecture .....	54
Figure 16: The data queue .....	58
Figure 17: Example protocol representation .....	63
Figure 18: views package <code>__init__.py</code> .....	65
Figure 19: Application main window, with Views menu open .....	65
Figure 20: Application main window, displaying a Current Value View .....	68
Figure 21: Application main window, with several Console Views open.....	69
Figure 22: Table View and field selection dialog.....	70
Figure 23: The Graph View, showing three data fields .....	71
Figure 24: File Capture View .....	73
Figure 25: CSV file imported into Microsoft Excel.....	73

Figure 26: Send Data dock in floating mode .....	75
Figure 27: Protocol Editor dialog .....	75
Figure 28: Main window .....	77
Figure 29:Main window menus .....	78
Figure 30: Settings dialog .....	79
Figure 31: Port not found error .....	80
Figure 32: Error when failed to read from serial port.....	80
Figure 33: Application running on a BeagleBone Black .....	81
Figure 34: Screenshot of the application running on BeagleBone Black .....	81
Figure 35: Example ADC and push switch protocol.....	82
Figure 36: Example console application output.....	83
Figure 37: Control (application running, connected to port, no data transfer) .....	89
Figure 38: Interval = 0 (data read whenever application idle) .....	89
Figure 39: Interval = 10 ms .....	90
Figure 40: Interval = 50 ms .....	90
Figure 41: Interval = 100 ms .....	90

This work is dedicated to the memory of my brother Jason who, despite having been taken many years before his time, led a life bursting with love and laughter, and will forever remain a constant inspiration.

## Acknowledgements

I wish to offer my thanks to the following people and organisations whose support has contributed greatly to this project:

To my employer, Smart Component Technologies, for fully supporting my efforts and allowing the time to work on this project.

To Dr Greg Horler, my primary supervisor, for his advice, encouragement and willingness to be involved with all aspects of the project, as well as the recommendation to pursue this work initially.

To Jack Hughes, for his extended use of SerialPlot throughout its development, and constructive and detailed feedback.

To the open source software teams and mailing list, forum and IRC users who generously volunteer their time to help others get the most out of their software.

To my parents, Chris and Chris Morris, for continued support, encouragement and understanding throughout the length of the project.

To Amy Buchanan, for constant encouragement and the provision of various edible items during the writing of this thesis.

# Nomenclature

ASCII	American Standard Code for Information Interchange. A common character encoding which utilises 7 bits to represent various printable characters and control codes.
CPU	Central processing unit. The primary, general purpose microprocessor in a computer system.
checksum	A small datum calculated from a block of data, used to detect errors in said data.
class	In object oriented programming: a template for creating a specific type of object, which defines the various attributes and methods it contains.
free software	Software licensed in a way that allows users to use, modify and distribute it with few restrictions. A free software package will also generally be open source.
GUI	Graphical User Interface.
hex	Hexadecimal. A representation of integer values in base 16, typically using characters 0-9 and A-F.
object	In object oriented programming: a specific instance or occurrence of a data type (class).
open source	Software that allows users access to the application source code. Open source software is often also free software.
windows	(Small 'w') A graphical control element in a GUI; a rectangular visual area which can be moved.
Windows	(Capital 'W') The Microsoft Windows operating system.

Text written in monospaced font with a grey background (`as such`) represents the name of a Python file, package, module, or code element.

# 1 Introduction

Much of the world today, in some shape of form, uses embedded computers and microcontrollers to assist with our lives. In the western world in particular, such devices have come to dominate nearly every aspect of life, from the smartphones, washing machines and cars individuals use every day, to large-scale systems controlling transportation, utilities and manufacturing. ARM (2014) – a major embedded microprocessor designer – reported that 12 billion chips based on their architecture alone had been shipped in 2014.

With such a large industry dedicated to embedded systems, naturally it now plays a large part in education from university down to primary, as well as being a leading field in both academic and commercial research and development. Low cost products such as the Arduino development board and Raspberry Pi single board computer have further introduced this trend to the hobbyist field.

Most embedded systems offer a user interface to some extent, be it a keypad and 7-segment display on a microwave oven, the numerous controls of a modern automobile, or a high resolution touch screen on a jukebox. However, these systems often offer a secondary communication interface to allow initial configuration, continuous monitoring, fault finding and re-configuration. These frequently use common standards such as RS-232, CAN bus, USB and Ethernet. In order to communicate with the devices over such interfaces, either an application-specific software application must be created, or existing generic software used.

There are several disadvantages to developing a bespoke piece of software for every application, in particular the requirement to have skilled software developers available, the time required to develop the software, and the cost

involved with doing so. It is clear, therefore, that it is preferential to re-use existing software for this purpose.

Several such software applications exist, however few offer a complete package of tools required to view and analyse the data effectively, as well as communicate back to the system, in a flexible, configurable manner. As such, there is an apparent need for a solution that will allow parties interested in monitoring and testing serial communications to do so easily and effectively.

The author, in a previous research project, developed a basic computer application (known as SerialPlot) which displayed RS-232 serial data in both textual and graphical forms, and provided the ability to capture data to a file. Functionality was added which allowed the user to specify “protocols” to extract data from incoming data streams, and display only the particular data field of interest. (Morris, 2012) Whilst useful for certain basic scenarios, the features available were relatively limited.

As such, it was clear that there was still a need for a more flexible application which could (a) be configurable to match the particular use-case, (b) allow support for communication interfaces other than RS-232, and (c) be able to be integrated into other existing applications or new software products.

## 1.1 Project aims and objectives

The over-arching aim of this project was to design a software interface to allow a user to analyse, control and manage various embedded systems. To achieve this, the following project objectives were defined:

- Investigate and define the need for such a system.
- Investigate the features and deficiencies of existing software applications – including the original SerialPlot application – and hence define the features of the new application.

- Investigate what code, if any, could be re-used from the original SerialPlot application.
- Design and implement a cross-platform application to meet the project aim:
  - Develop a high performance, modular application core to efficiently process data streams and present them to user interfaces.
  - Develop a flexible, extensible graphical user interface (GUI), allowing user-configurable views and layouts.
  - Develop a graph view, allowing multiple data fields to be plotted on the same graph.
  - Develop additional views, allowing data fields to be viewed in formats other than graphically.
  - Develop a flexible hardware backend, to allow communication with RS-232, but also to allow support for other hardware interfaces to be easily added.
- Test the application with a number of use-cases, to determine if it meets the project aim.

## 2 Background

### 2.1 Serial communications

Embedded systems have a need to communicate with the outside world.

Although many have direct human interfaces, such as screens, buttons and LEDs, there is often a need to obtain machine-readable outputs and control, in order to integrate the equipment into larger systems, or for programming, debugging and testing purposes. Three common protocols that facilitate this are RS-232, Ethernet and USB.

In particular, RS-232 is popular due to its electrical, timing and wiring simplicity. An RS-232 based serial port was once a ubiquitous occurrence on desktop computers and laptops for communication with various peripherals, modems and directly to other systems, however today has been almost completely replaced with USB, Ethernet and wireless communication for general purpose computing. Nevertheless, this communication format has remained a common interface on embedded systems. The Arduino, Raspberry Pi and BeagleBone boards – three popular (as of 2015) and low cost development boards – all contained a universal asynchronous receiver/transmitter (UART), similar to RS-232, but with voltage levels compatible with transistor-transistor logic (TTL).

RS-232 communication links send raw data bytes over the line, with very little protocol overhead. A typical RS-232 transmission of a byte can be seen in Figure 1. Whilst the packet structure is not defined in the standard, a common pattern is as follows:

- One start bit
- Between five and eight data bits (8 in this example)

- An optional parity bit, which would represent either odd or even parity (none in this case)
- One, two, or one-and-a-half stop bits (one in this case)

The format used in the figure (“8N1”) is the most frequent used today, most likely due to 8-bit bytes used in most computer systems, as well as the ASCII and UTF-8 character sets.

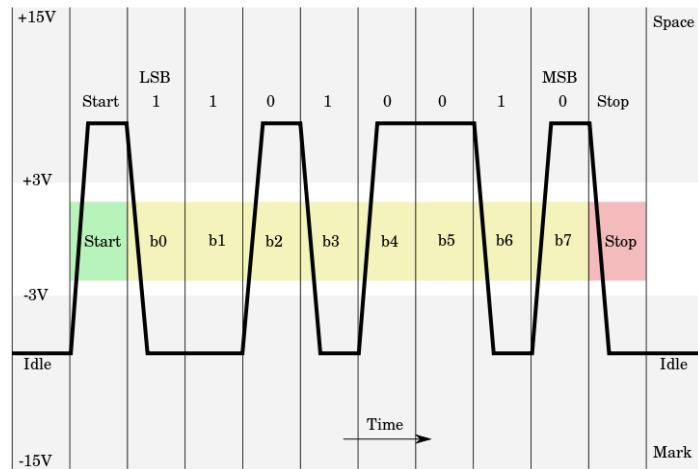


Figure 1: Example transmission of 1 byte over RS-232<sup>1</sup> (Ktnbn & Tardieu, 2009)

## 2.2 Existing solutions

A number of solutions, ranging from simple hobbyist applications to full commercial solutions, had previously been created, which attempted to implement some of the requirements to create a system to monitor and test embedded systems.

### 2.2.1 RS-232 solutions

#### 2.2.1.1 Basic applications

A number of very common serial communication utilities exist, which provide basic serial port access, but lack the additional functionality that may be required.

---

<sup>1</sup> Licensed under Creative Commons Share-Alike 1.0 license (<http://creativecommons.org/licenses/sa/1.0/>)

In Microsoft Windows up to Windows XP, the HyperTerminal application was included, and is still available to purchase for later Windows versions from Hilgraeve (n.d.). Despite being a relatively basic tool, a few useful features are available, for instance the ability to transmit files using several protocols, and assign keyboard shortcuts to send certain character combinations. The application is particularly useful for debugging modem connections and communicating with certain legacy systems, however for general purpose serial debugging, it does have its limitations.

Another common Windows program is PuTTY, which allows various types of internet protocol (IP) network connections, as well as serial connections. Again, this provides a relatively limited set of features, with a standard terminal display, albeit highly configurable. Due to its large number of configuration options, a useful feature is the ability to save configurations to the Windows registry, which can later be reloaded.

### 2.2.1.2 Realterm

Realterm is an open source terminal application which, according to its authors, is "specially designed for capturing, controlling and debugging binary and other difficult data streams." (Realterm: Serial Terminal, 2014) The main interface is split into two main sections, as shown in Figure 2. The top half is a traditional terminal view, and the bottom area consists of a number of tabs, which contain a large amount of settings, controls and status information.

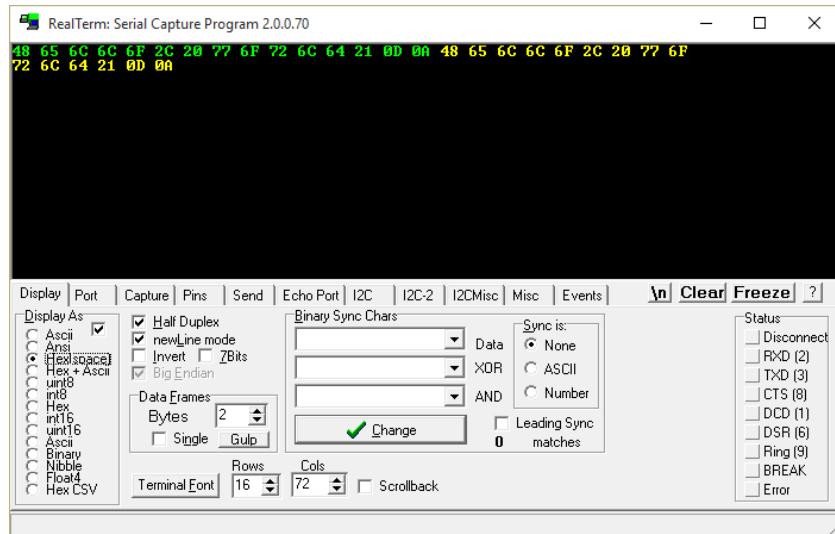


Figure 2: Realterm main window<sup>2</sup>

The “Display” tab enables the user to control a number of options pertaining to the terminal area. The data can be displayed in fourteen different formats including ASCII text, decimal, hexadecimal and binary, and there is an option to group data into fixed byte length “frames”, which will keep the data together when it wraps to a new line in the terminal. In addition, new lines can be triggered when a particular string of characters or bytes is matched.

The “Capture” tab enables received data to be captured to a file. By default, data is stored as the raw binary data, however options are available to encode as hex characters and prepend a timestamp. The data can append to or overwrite an existing file, and can be stopped manually or after a certain amount of data has been received or time has passed.

Data can be typed directly into the terminal area, or can be sent with more control using the “Send” tab. The “Send” tab allows data to be entered into input fields, which is temporarily saved to allow the same data to be easily re-sent later. Data entry can either be ASCII text, or in numerical formats.

---

<sup>2</sup> Screenshot from software released under a permissive free software license (BSD license).

There are various options available to transform the input, for example adding end of line characters, cyclic redundancy checks (CRCs) of various formats, and repeating the input a number of times. Data can also be sent using data loaded from a file, which can be repeated a number of times, with optional delays between characters, lines and iterations.

Clearly, Realterm contains a large number of options and functions. As well as the commonly used ones listed above, there are additional tabs containing functionality to communicate with certain specific devices over serial ports. Although this level of functionality does add a considerable amount of flexibility, this does result in a rather crowded interface. Coupled with the fact that certain controls are poorly labelled, and the documentation is somewhat limited, it can make the software unintuitive for a new user, and can be tricky to work with when used in earnest. It is also not possible to save different configurations, which means that for a complex setup, the settings have to be re-entered each time the application is launched. Whilst a number of the options can be passed as command line parameters – allowing a Windows shortcut to be made to relaunch the configuration – this is not as straightforward to manage as a simple save/load interface.

### 2.2.1.3 Serialtest

Serialtest is a commercial hardware and software serial analyser developed by Frontline Test Equipment. The system consists of a probe which connects to a PC over USB, which runs the Serialtest software, and can analyse either RS-422 and RS-485 data, or RS-232, depending on which probe is used. The system costs around £850 for the RS-232 version. (BCD Microelectronics, 2015)

A key feature of the Serialtest system is that, rather than the PC used for analysis being one end of a serial connection, the supplied probe can sit in

line between two devices, which are communicating, and monitor the data being transferred between them.

Serialtest contains built in support for some common protocols, such as Modbus, the Serial Line Internet Protocol (SLIP) and the Point-to-Point Protocol (PPP). In addition, it contains a scripting language that allows users to create their own protocol decoders, as well as being able to filter data.  
(Frontline Test Equipment, Inc, 2012)

The software interface of Serialtest – as shown in Figure 3 – is a single document interface, with multiple windows for various views. These include the “Event Display” view, which shows the raw data in various formats, and the “Frame Display” view that allows the user to select a data frame (defined by the protocol) and view the information decoded from the data. For instance, when using the Modbus protocol, this includes details such as which direction the frame was travelling, the Modbus function name and parameters, and whether any errors occurred, for example the checksum failing. Additional views are available to view the status of the various signals on the serial port, and view statistics for the currently captured data.

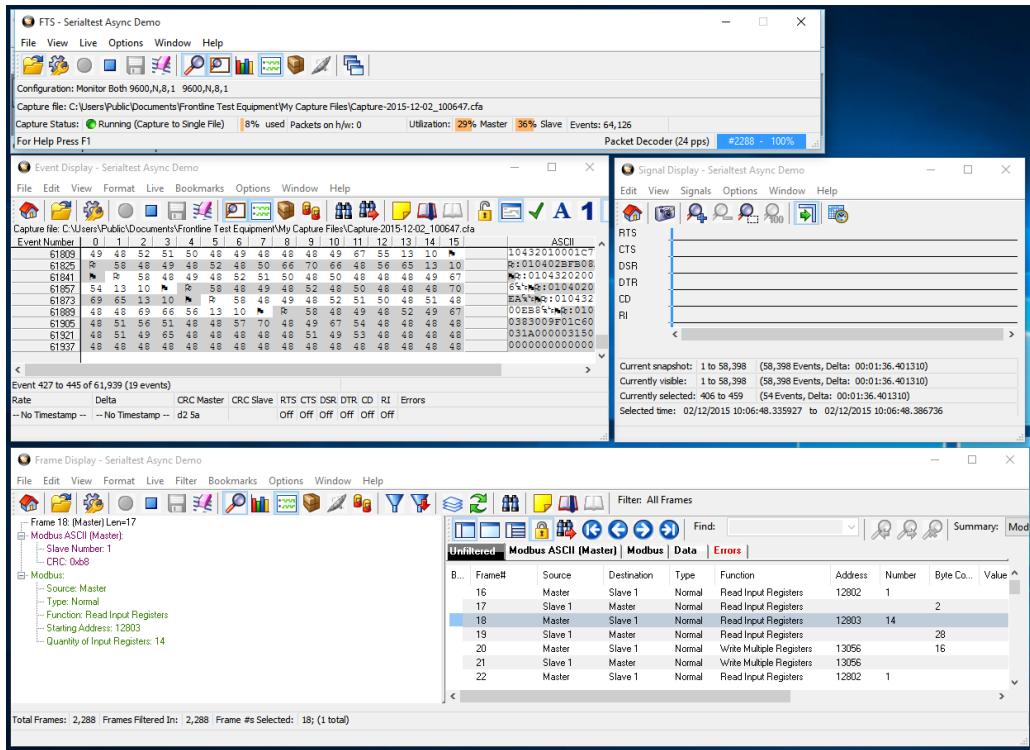


Figure 3: Serialtest interface<sup>3</sup>

A useful feature of the GUI is the ability to duplicate certain windows – such as the Event Display view – and configure them differently. This can allow the same data to be displayed in different formats simultaneously. One slight disadvantage of the GUI design is the use of the multiple windows approach. With several windows open, it can become arduous to manage all the windows without losing track. However, like similarly functioning applications, windows can be brought to the foreground using a menu, and windows can be arranged in a cascading fashion.

#### 2.2.1.4 qSerialTerm

qSerialTerm is an open source, cross platform serial port utility developed by Jorge Aparicio. Although designed to be a simple tool to meet a few specific requirements rather than a fully-fledged application, it does provide interesting solutions to some unique problems. qSerialTerm is developed

---

<sup>3</sup> Used with permission from Frontline Test Equipment.

using the Qt framework, which allows GUI applications to be written which are cross platform. Qt is discussed in detail in section 3.5.2.

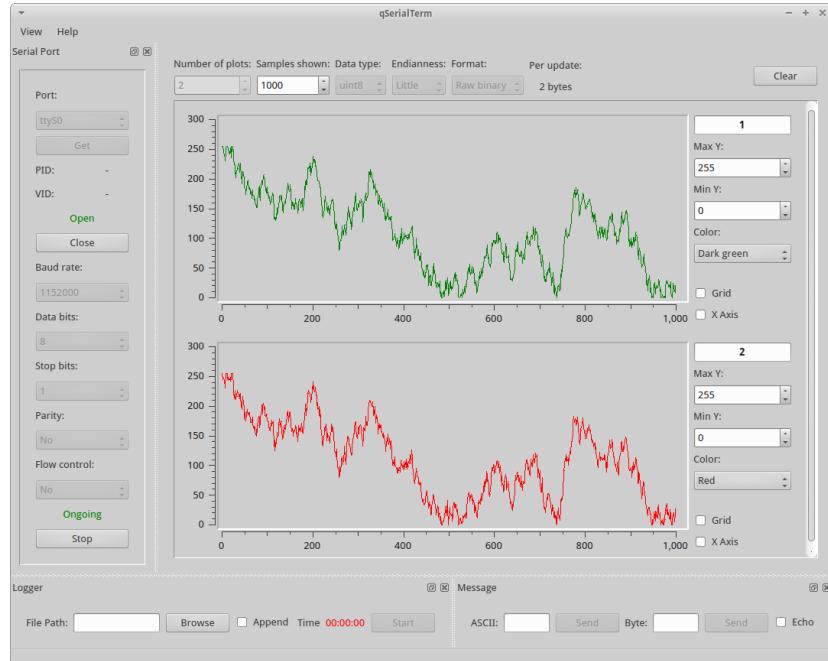


Figure 4: qSerialTerm application<sup>4</sup>

The main qSerialTerm window – as shown in Figure 4 – consists of a main data display area, as well as several "dock widgets" surrounding it that allow application configuration and interaction with the data. The use of dock widgets makes the display of the application quite flexible, without enforcing a number of windows or sub windows. In Figure 4, three dock widgets are displayed: the serial port configuration to the left, and the logger and message configurations at the bottom. The docks can be dragged to different sides of the window, stacked on top of each other to create a tabbed interface, "floated" to act as an independent window, or closed completely. The "View" menu contains options to show and hide each dock.

The main data display area has three configurations: a basic terminal view that displays incoming data as ASCII text, a data acquisition mode that

---

<sup>4</sup> Screenshot from software released under permissive, free software license (GNU General Public License)

displays one or more real-time graphs, and an image-plotting mode, which can parse incoming data to display an image. Of these, the graph view provides the most flexible interface. The number of plots and the data type can be selected, which effectively allows a simple protocol to be configured. The data type is configured once for all the fields, and can be signed or unsigned integers of 8, 16 or 32 bits wide; or floating-point values. For instance, groups of three 16-bit integers could be transmitted from a device, and qSerialTerm would be able to plot these values onto three graphs. Some features of the graphs – such as scale, number of samples shown and colour – can be configured, however the data specification can be quite limiting, for instance it is not possible to have fields of differing sizes, or mask unused fields. In addition, whilst the graphs are scaled horizontally, they have a fixed vertical size, which means that (depending on the screen resolution) typically only two to three graphs can be displayed without needing to scroll.

One issue was observed when experimenting with the program. qSerialTerm was tested using a virtual serial port, and a basic Python program that transmitted a pseudo-random data packet in a loop, with a configurable delay. When using a sample rate of approximately 10 kHz, the program handled the data relatively well, but doubling this to around 20 kHz caused the graph to stall, and the whole application to lock up until the data stream was stopped.

#### 2.2.1.5 SerialPlot

SerialPlot was a GUI application written by the author for an undergraduate degree project. The program was written using the Python programming language and using the Qt GUI framework.

SerialPlot began as a project to create a software application to plot RS-232 serial data in real-time; however, the project developed into the beginnings of a system with the ability to decode and interpret user-defined protocols.  
(Morris, 2012)

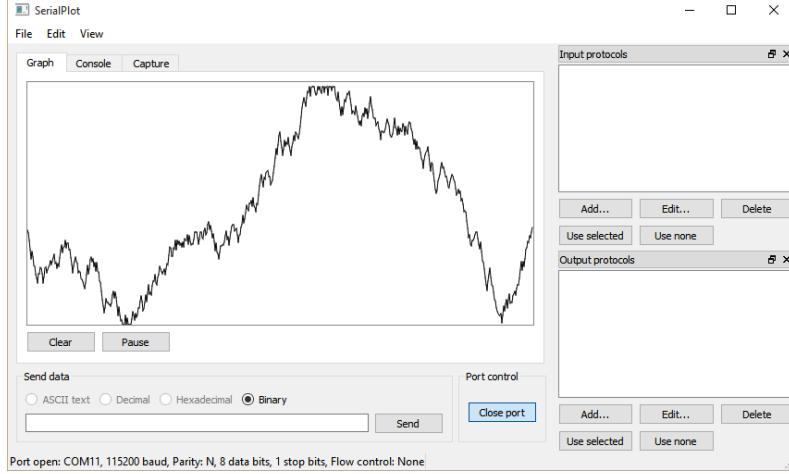


Figure 5: Original SerialPlot application, showing graph tab and protocol selection areas

The main window, shown in Figure 5, consisted of three main areas:

- a set of three tabs, which could display a graph, a console view displaying each datum as a decimal number, or file capture display,
- a *send data* area, allowing data to be sent in various formats (although only binary had been implemented), and
- two *protocol editors*; one for incoming data, and the other for outgoing data.

Like some existing basic tools, such as qSerialTerm, and Eli Bendersky's (2009) data monitor, SerialPlot had a real-time graph display. This was relatively basic, having a fixed range, and a single trace. Options were available to clear and pause the graph display. In addition, the console view was extremely simple, displaying values only in decimal form, and one per line. A data capture tool was also available, which allowed incoming data to be saved to a comma separated values (CSV) file, preserving the field ordering.

SerialPlot's primary unique selling point, however, was the ability to set bitwise protocols. Whereas applications like Serialtest and – to an extent – qSerialTerm allow some elements of protocol creation, SerialPlot allowed a much finer control, for instance allowing a 10-bit value spanning two bytes to be displayed. This allowed values encapsulated in an incoming data packet to be displayed or captured as the correct fields, as well as encapsulating transmitted values into data packets understood by the connected device.

Protocols were edited using the “protocol editor” dialog. A sample protocol can be seen in Figure 6, which consists of:

- A start byte. This is defined as a “fixed” field, which would result in SerialPlot ignoring data until a byte of that specific value was matched. This is a good way to ensure that a particular data stream is correctly aligned with the start of the protocol.
- Six padding bits, followed by a 10-bit field. In this case, the 10-bit value spans two bytes, but not all the bits of the first byte are used; therefore, these are set as an “ignored” field, which would cause SerialPlot to discount these bits completely. These two fields could be swapped if the data was aligned to the start of the first byte. The “plot/console” checkbox is checked, which would result in the values displaying on the graph and console tabs in SerialPlot.
- A final 8-bit data field. In this case, the “plot/console” checkbox is not checked, so SerialPlot would not display this value, but still store the value when using the data capture mode.

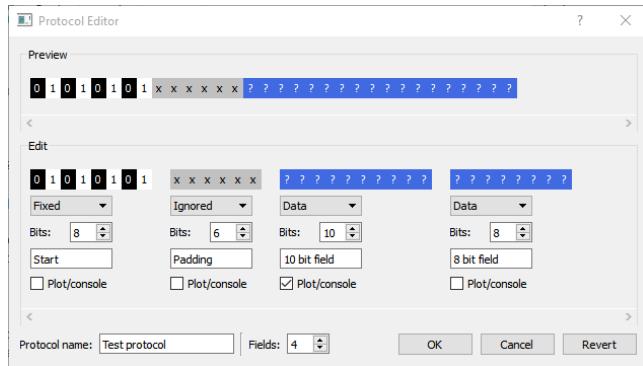


Figure 6: SerialPlot's protocol editor

Through using SerialPlot both during and after development, it proved to be a useful tool, and filled several use cases missed by other similar applications, and feedback from other users reflected this. As such, it was seen as a logical candidate to use as a foundation for the project. However, there were a number of issues and deficiencies noted in the application that needed to be addressed to turn it into a more powerful and flexible tool suitable for scientific and engineering use. These are discussed in section 3.3.

### 2.2.2 Other solutions

A number of popular software applications exist, which are designed to monitor data streams other than RS-232.

### 2.2.2.1 Wireshark

Wireshark is an open source, cross platform GUI application for network packet capture and analysis. It is an extremely popular piece of software for IT professionals, and – according to a press release – is downloaded over 1.5 million times per month. (Riverbed Technology, 2015)

The Wireshark main window, as shown in Figure 7, consists of three main panes. Upon starting a capture, Wireshark displays IP packets which are entering and leaving the PC in the top pane – a tabular view. Selecting a packet allows the user to drill down through the various layers of the Internet Protocol Suite in the middle pane. In the case of Figure 7, Ethernet

(Link Layer), Internet Protocol (Internet Layer), Transmission Control Protocol (Transport Layer), Hypertext Transfer Protocol (Application Layer) and Extensible Markup Language (data being transmitted) are shown. The raw packet data is displayed in the bottom pane, and selecting a layer in the middle pane highlights the data relevant to that particular protocol.

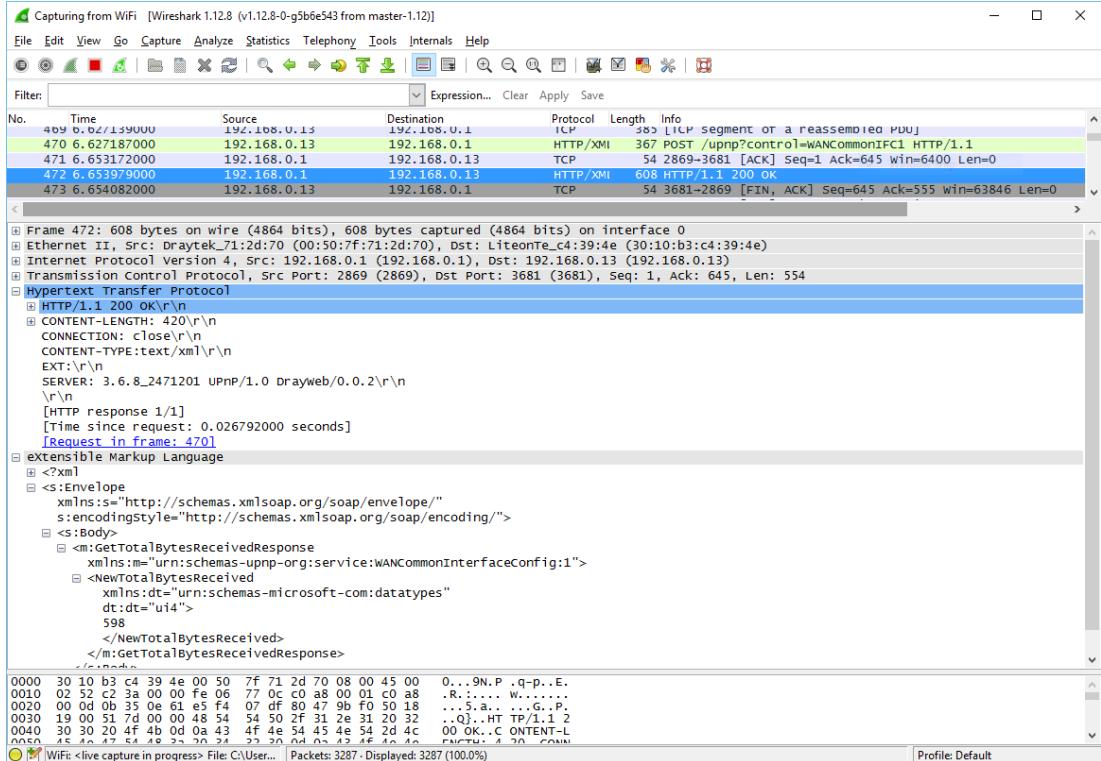


Figure 7: Wireshark main window<sup>5</sup>

Wireshark supports around 1700 protocols (Combs & contributors, 2015) and “dissectors” can be written in C code to support new protocols. The Wireshark interface is clean and intuitive, is designed around the expectation that there will be multiple protocol layers encapsulated within one another, and allows filtering of packets at all protocol layers.

<sup>5</sup> Screenshot from software released under a permissive, free software license (GNU General Public License)

### 2.2.2.2 USBlyzer

USBlyzer is a commercial tool for analysing Universal Serial Bus (USB) communications from the system on which the software is running. It provides a tree view which lets the user navigate through the various USB ports and hubs connected to the system. Upon selecting a device, detailed information is shown in a “USB Properties” pane.

Two other panes provide functionality similar to Wireshark, in that USB packets can be seen in a tabular view in one, and selected to view data analysis of the various fields in the other.

One interesting feature of the user interface is the presence of four “screen sets”. These allow the user to lay out the various panes to their liking in four different configurations, then use toolbar buttons to switch between them. This makes it simple to set up the application for different scenarios or users, and quickly switch between them.

### 2.2.3 Feature comparison and rationale

A comparison of the various features of the applications can be seen in Table 1.

Key:

- Y: Yes, does support the feature.
- N: No, does not support the feature.
- P: Has partial support for feature.

Application	HyperTerminal	PuTTY	Realterm	Serialtest	qSerialTerm	Original SerialPlot	Wireshark	USBlyzer	New application
<b>RS-232 support</b>	Y	Y	Y	Y	Y	Y	N	N	Y
<b>USB support</b>	N	N	N	N <sup>6</sup>	N	N	Y <sup>7</sup>	Y	N
<b>Internet protocol (IP) support</b>	Y	Y	Y	N	N	N	Y	N	N
<b>View textual data</b>	Y	Y	Y	Y	Y	P <sup>8</sup>	Y	Y	Y
<b>View data graphically</b>	N	N	N	N	Y	P <sup>9</sup>	N	N	Y
<b>View data in a table</b>	N	N	P <sup>10</sup>	Y	N	N	P <sup>11</sup>	P <sup>12</sup>	Y
<b>Specify byte level protocols</b>	N	N	P <sup>13</sup>	Y	P <sup>14</sup>	N	Y	P <sup>15</sup>	N
<b>Specify bit level protocols</b>	N	N	N	N	N	Y	Y	N	Y
<b>Intuitive user interface</b>	Y	Y	N	Y	P	Y	Y	Y	Y
<b>Configurable user interface</b>	N	N	N	Y	N	N	N	Y	Y
<b>Send data</b>	Y	Y	Y	Y	Y	P <sup>16</sup>	N	N	Y
<b>File transmission</b>	Y	N	Y	Y	N	N	N	N	Y
<b>File capture</b>	Y	N	Y	Y	Y	Y	Y	Y	Y
<b>Save application configuration</b>	Y	Y	N	Y	N	N	Y	Y	Y
<b>Good performance at high data rates</b>	Y	Y	Y	Y	N	P <sup>17</sup>	Y	Y	Y
<b>Application programmable interface (API)</b>	N	N	Y	N	N	N	Y	N	Y
<b>Cross-platform</b>	N	Y	N	N	Y	Y	Y	N	Y

Table 1: Communication analyser product comparison

<sup>6</sup> Available in a different product by the same company.

<sup>7</sup> With use of a third party tool.

<sup>8</sup> Only decimal format.

<sup>9</sup> Only one data field.

<sup>10</sup> Can set up terminal to display in a tabular format to an extent.

<sup>11</sup> Data is viewed in a table at the IP packet level.

<sup>12</sup> Data is viewed in a table at the USB level.

<sup>13</sup> Can specify data frame widths and/or sync characters, which start a new line when matched.

<sup>14</sup> Can specify the number of data fields, all of which must be the same width.

<sup>15</sup> Standard USB protocols used; not higher level.

<sup>16</sup> Only binary format.

<sup>17</sup> Generally, yes, but issues with graph display in some cases.

Table 1 presents the various features in common applications used for analysing and controlling embedded systems. Therefore, the aim is to include all of the features listed in the proposed application with the exception of USB support, IP support and the ability to specify byte level protocols. The rationale for this was:

- The complexity of USB means that support is difficult to achieve in a way which would be flexible yet user-intuitive. USBlyzer implements this by providing an interface to drill down through the various devices and protocol layers, however this would be difficult to achieve whilst maintaining support to interrogate low level protocols such as RS-232. In addition, it would likely require hardware drivers to be written, which would add an additional level of complexity, and would lock the application to specific platforms.
- The IP stack contains a large number of protocols, which build on top of one another, which again makes this difficult to support. However, with the introduction of a generic hardware interface, access to specific high level protocols could be implemented. For instance, support for a specific protocol on top of IP could be supported as an add-on to the application.
- The support for byte level protocols would not be implemented, as the same functionality would available using bit level protocols.

As such, the development of a system providing the fourteen features is described in the following sections.

### 3 System design

#### 3.1 Programming language

One of the first decisions to be made when developing a software solution is the programming language and associated execution environment. Whilst most modern high-level languages offer similar feature sets and could be used to produce a similar result, a number of factors impact on the language choice, and indeed the choice of language has an impact on application design and implementation specifics.

SerialPlot had originally been written using the Python programming language and as such, it appeared to be a good candidate for continued development. However, it was necessary to assess whether another language had any major advantages that would outweigh the additional work required to re-implement the existing functionality into the new application.

Table 2 outlines a comparison of the most popular programming languages that were deemed suitable candidates for use in this application, and the features are discussed in more detail below.

Language	Code reuse	Cross platform	Performance	Support	Speed of development
C	N	Source – limited Compiled – no	Native	Many libraries available. Very mature language.	Low
C++	N		Native	Many libraries available.	Low
C# (.NET/Mono)	N	Limited	Virtual machine	Some libraries freely available. Commercial support available.	Medium
Java (JVM)	N		Virtual machine		Medium
Perl	N	Y	Interpreted		High

<b>Python (CPython)</b>	Y	Y	Interpreted/ virtual machine	Very active community. Central repository of libraries available.	High
-----------------------------	---	---	------------------------------------	--	------

Table 2: Comparison of programming languages

An important requirement of the application was that it should be cross platform. To this end, a language was required which was supported on a number of commonly used operating systems and processor architectures. Languages that are compiled to machine code—such as C and C++—are in a sense cross platform, however they must be compiled for each operating system and hardware architecture combination, and often for different library versions running on those systems. As such, a language that can be run directly through use of an interpreter, or one that compiles to cross-platform byte code would be preferable.

A related factor is the performance of the language to be used. To an extent, this is becoming less critical as the performance of computers increases, however it can still be an important consideration in processor intensive applications. There are broadly three categories of programming languages. The first is the traditional languages that compile to machine code, such as C and C++. With a modern compiler, these languages are highly optimised for the hardware and operating system for which they are built.

At the other end of the spectrum are interpreted languages. Programs written in such languages cannot be executed directly, and instead utilise an interpreter which utilises the source code at run time. Falling in the centre of the spectrum are languages such as Java, which are compiled to a platform independent byte-code. In order for the program to be executed, it must be run in a "virtual machine" or "runtime environment". This procedure

compiles the byte-code to machine code on the fly, which results in a compromise between performance and cross-platform compatibility.

C<sub>Python</sub> – the default Python implementation – is a notable exception to these categories; it could be seen as a hybrid between interpreted and virtual machine based languages. Although not requiring an explicit compilation phase, running Python code using C<sub>Python</sub> generates an intermediate byte-code, which is then implemented in a virtual machine. The byte-code is then cached to a file, which allows faster start up time in consecutive executions.

(Python Software Foundation, 2015)

One further consideration is the level of support available for development. C and C++ are very mature languages, and there has been much reference material written about them. Java and the .NET languages are often used commercially, and hence often have commercial support available. However, modern, high level languages such as Perl and Python often have a strong community backing. Both Perl and Python have mailing lists, chat rooms and user groups, as well as providing a repository of community maintained libraries.

A final consideration is the speed of development. Generally, the higher level the language, the quicker the speed of development, as one does not need to be concerned about low-level features such as memory management.

In this particular case, Python was chosen as the programming language for a number of reasons. Firstly, the Python language style is very flexible and succinct, allowing rapid development and prototyping of new features. In addition, the existing SerialPlot application was written in Python, so keeping the same language would ease the additional development work. In addition, a large number of libraries (nearly 70000 as of November 2015 (Python Software Foundation, 2015)) are available through the Python

Package Index, so harnessing existing libraries to add additional features would be simplified. Although there was likely to be a reduction in performance, this was considered an acceptable compromise against the advantages the language offered.

### 3.2 A modular approach

From the onset, it was clear that the application should be as modular as possible, easing future development and the addition of new features. With a high level of modularity and good documentation, it is possible to implement a system that allows a third party developer to add features without requiring an in-depth understanding of the code.

Two features of Python that lend themselves to modular design are modules and classes. Python modules are Python code files that define functions, classes and attributes, and can be imported into and re-used in other modules. This allows an application to be built using component parts; with those parts able to be used both in the application, and in other applications.

Although not a strict object oriented language, Python does lend itself to such a programming style. Object oriented programming is based on the concept of "objects", which are similar to data structures and contain "attributes" – which store data – and callable "methods". A key feature of object oriented programming is the ability to have a hierarchical structure of object types (known as "classes"), allowing a base class to be written from which child classes can be created. This allows objects to be created which have a common interface, but perform different actions as a result. In Python, however, this common interface is not strictly enforced, so must be documented accordingly.

One key advantage of writing an application in such a way is that the individual parts of the application could be re-used in other applications. For instance, the GUI could be replaced with a simpler but less flexible one, or a command line application written for automated use.

### 3.3 Deficiencies of SerialPlot

Although the original SerialPlot application could act as a good starting point for a new, fully featured application, it had a number of deficiencies, which would need to be rectified in order to produce a fully functioning application.

#### 3.3.1 GUI design

SerialPlot had been designed with a few core objectives, and the GUI layout reflected this. The tabbed interface made it difficult to perform multiple tasks simultaneously. For example, it was not possible to view numerical values and the graph view at the same time, and the user was not able to start and stop data capture whilst monitoring the data.

In addition, the graph view supported only a single trace, which meant that it was impossible to differentiate between multiple fields plotted from the same protocol. Similarly, the console view supported displaying the values from each field – one per line – but failed to provide any means to identify to which field each value was associated.

#### 3.3.2 Graph issues

During the development of the original application, an issue was encountered in which the graph would not update correctly, when a fast stream of data was received. (Morris, 2012) This was seemingly resolved by buffering the data and plotting at defined intervals, however on further testing, this did not fully resolve the issue.

Figure 8 shows this effect. To the right of the dashed line, the data was plotted correctly, in near real time, however, to the left; the plot would seem to stall, with parts of the plot only updating every few seconds. The extent to which this occurred depended on the width of the window, and hence the size of the graph. At the time, the cause of this issue and potential solutions were unclear, but this would need to be investigated further in order provide a good user experience.

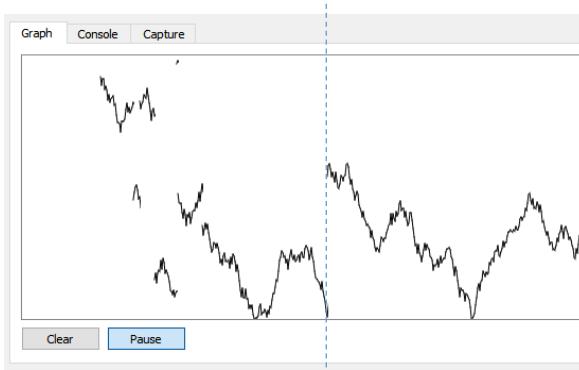


Figure 8: SerialPlot graph plotting errors

Another – albeit minor – issue with the graph was that of scaling. The graph widget had been designed to scale to fill available space; however, the data that was currently displayed was not retained in memory after it had been plotted. This left the only realistic option to clear the graph when it was resized. For fast, continuous data transmission, this was not an issue, however when receiving slow or intermittent data, or if the graph had been paused, this could result in the user not being able to view the data which they were interested in.

### 3.3.3 Application state

The original version of SerialPlot had no way to save or otherwise export the application state. This meant, for instance, that every time the application was started, the user had to re-enter the serial port settings. Furthermore, although multiple input and output protocols could be created, these were only stored in memory, and hence would be lost upon application exit.

In addition, it was envisaged that a potentially useful feature would be the ability to share potentially complex protocols with others, rather than it having to be communicated in another manner, then re-entered manually.

### 3.3.4 Data transmission inflexibility

As previously mentioned, the application had an extremely limited data transmission feature. The only implemented data format was binary, and this could only be sent as raw data, or in the first data field of the protocol.

This resulted in difficulties sending data using protocols other than the most trivial: as data for only one field could be specified in the main window, any further fields would have to be specified as “fixed” in the protocol editor, and edited every time they needed to be changed. In addition, manually entering all data as binary was unlikely to be the most useful format for many use cases.

### 3.3.5 Extendibility

Finally, since SerialPlot had been built to meet a small number of narrow requirements, adding any new features often required restructure and rewriting of large sections of the program. In order to create a system that was flexible and simple to develop further, taking the time to restructure the code to meet this aim was seen as essential to ease continued development, and maintain a stable application.

## 3.4 Features

### 3.4.1 User interface concept

Despite creating an application which could be flexible in its usage, a graphical user interface (GUI) would be supplied which would act as the primary, default interface. A well-designed GUI would allow novice users to

use the application intuitively, without the need for a programming background or to engage in a steep learning curve.

Qt allows for a number of different GUI types to be created, including a multiple document interface (MDI). An MDI is an application that consists of a main window, which contains multiple child windows that can be moved, resized and managed within the constraints of the main window. MDIs were traditionally designed to work with multiple documents (such as text files and spreadsheets) however their use for that purpose is now discouraged, in favour of a single document interface (SDI) or multiple frame interface (MFI) – displaying each document in a separate window. (MacDonald, 2006) The shift from MDIs to SDIs has been seen through the early-to-mid-2000s with high profile software such as Microsoft Office and Adobe Acrobat favouring an SDI/MFI. (Microsoft, 2007) (Adobe Systems, 2006)

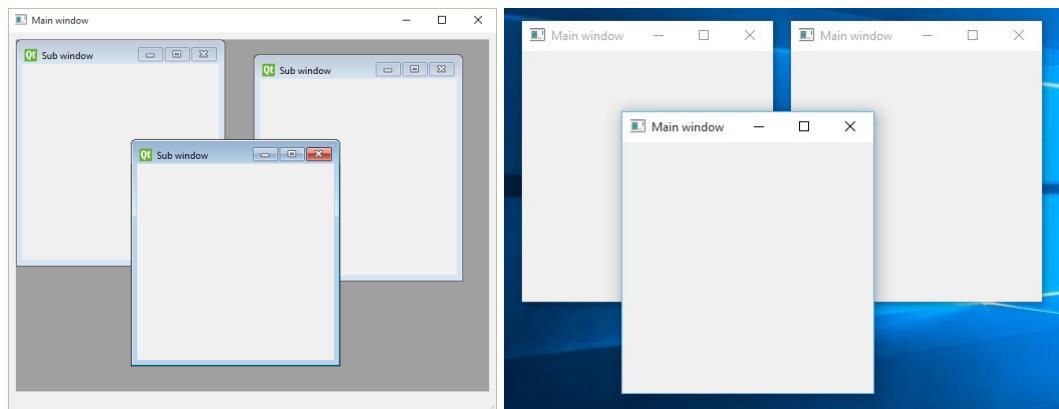


Figure 9: Comparison of MDI (left) and SDI interfaces

Despite this trend, it was believed that in this scenario, an MDI would be the most suitable interface. In applications such as word processing programs, each document is an unrelated object, however the data represented in each sub-window in our interface would be closely coupled, sharing the same sources. As such, it would be easier for a user to manage and interpret as well as simpler and more efficient to implement.

With the flexibility an MDI allows, such interfaces lend themselves to not only allowing the user to control the size and positioning of the sub-windows, but also the selection of which windows are currently open, even allowing multiple windows of the same type to be opened. This allows for an extremely configurable layout, for instance, two graphs could be shown, with only some data fields plotted on each graph.

The following views were determined to be the most useful in order to provide a flexible application. These are described in more detail in the subsequent sections.

- Console view: display of a single data field, with the ability to select from a number of display types.
- Table view: display of one or more fields in a tabular layout.
- Graph view: real-time graphing of one or more data fields overlaid.
- File capture view: allow capturing all data fields to a file.

In addition to the display views, a means to send data over the serial port, encapsulated in various protocols, was required. As this was likely to be frequently required, this feature was designed to be encapsulated into a dock widget, similar to those used in qSerialTerm.

Figure 10 shows a concept drawing of the proposed application main window, containing a number of sub windows, and the dock widget used for sending data.

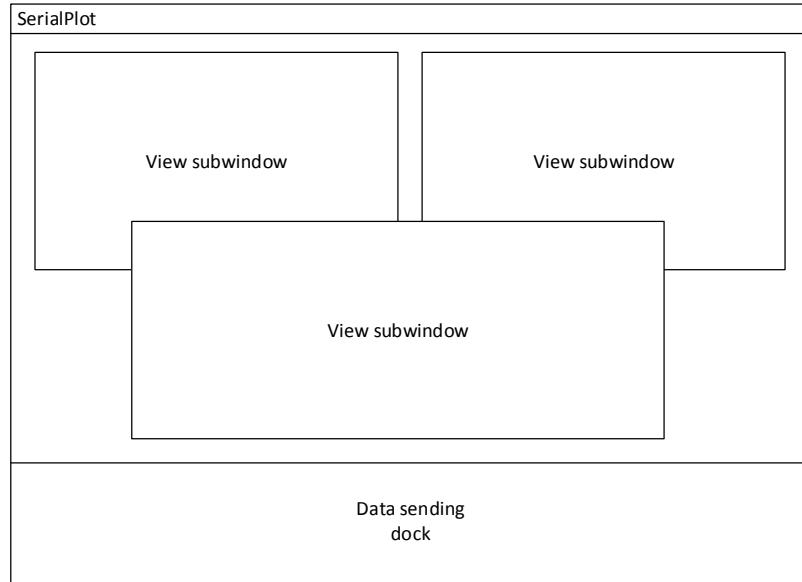


Figure 10: Concept GUI layout

### 3.4.2 Textual display

Two textual display views were designed, to allow end users to view the values of the data fields as they were being received.

The first of these was the console view. This was designed with a similar aim to the original SerialPlot's console view: to provide a simple mechanism to view incoming data in near real-time. However, a specific aim of the new version was to allow a simple method of selecting which data field to display, as well as the format to display it.

A concept drawing of the console view can be seen in Figure 11. This included three basic elements: the console data display, a dropdown box to select a field to display, and a second dropdown box to select the display type.

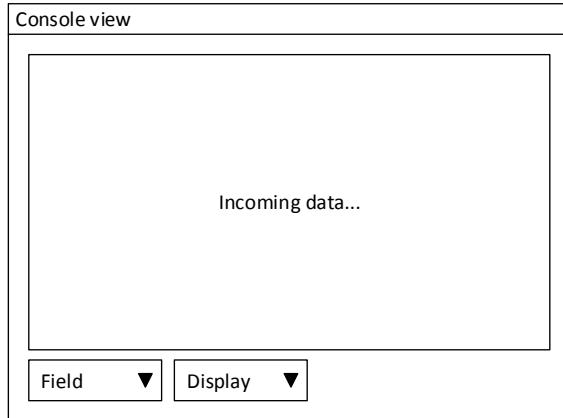


Figure 11: Console view concept

A table view was designed to allow a user to view data from multiple fields simultaneously, a concept drawing of which can be seen in Figure 12. In place of a textual widget was a table widget, and – since multiple fields needed to be selected – a button to launch a selection dialog was used instead of a dropdown box.

Table view				
Header	Field A	Another field	Test field	Field B
Select fields...				

Figure 12: Table view concept

### 3.4.3 Graphing

A graph tab was available in the original SerialPlot application, however this only supported a single trace, and had a fixed scaling. This limited plotting to a single data field, which was one byte in length, or less.

A new graph view was designed, which supported multiple traces and the ability to show or hide. Like the table view, this would require the more complex dialog to select multiple fields, rather than the dropdown box.

In order to display the data in the most user-friendly manner, consideration was required as to the horizontal and vertical scaling of the graph. In the vertical dimension, the range could be calculated based on the specified protocol: each field in the protocol would have a fixed number of bits, and hence a known maximum value. The range of the graph will therefore be 0 to the maximum value of the largest selected field. There were a number of options for the horizontal dimension: either a fixed (potentially configurable) number of points could be plotted however wide the sub window is, or at a fixed rate, e.g. one point per screen pixel. It was decided that the latter option would be most preferable, as the data window can be adjusted simply by changing the width of the GUI sub window. This would likely be the most intuitive option to the casual user.

Figure 13 shows a concept drawing of the graph view.

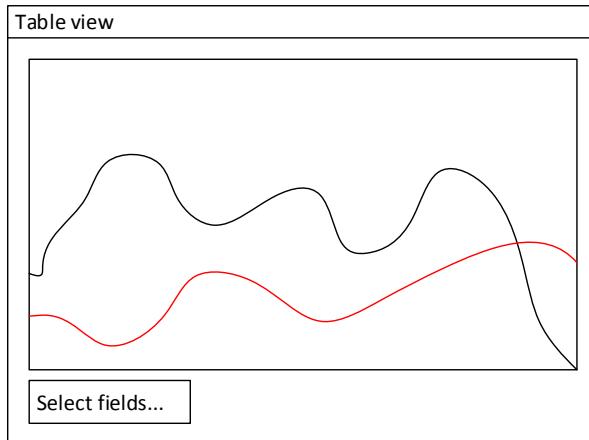


Figure 13: Graph view concept

#### 3.4.4 File capture

File capture to a CSV file had been a well utilised feature of SerialPlot, and was seen as an important one to retain. With the ability to save potentially large datasets to a file, this would allow a user to retrieve the historical data from an application such as MATLAB or Excel with little effort, allowing data to be scrutinised at a later point in time.

### 3.4.5 Protocol editor

SerialPlot contained a protocol editor which allowed users to define incoming and outgoing data protocols, in order to extract data fields from a stream. Multiple protocols could be created, and these were stored in dock widgets at the side of the application. However, this took up an unnecessary amount of screen area, and – as previously discussed – did not allow saving of the protocols to disk, meaning they had to be recreated every time the application was started, and could not be shared with others.

As such, it was determined that the protocol editor should remain in a separate window, but with options added to that window to store and load the protocols in a file, rather than in memory.

One additional feature which was felt to be lacking was the ability to validate fields. Although “fixed” fields could be defined, which would ensure that certain bytes would be required to match in order for a packet to be accepted, the ability to have finer validation control was seen as an advantage. For instance, a checksum field could be used, and this compared to a checksum generated after the data was received by the application. However, as described in section 3.5.3, by harnessing the power of SymPy’s parsing functions, the protocol validation would not be limited to a single algorithm, but rather open to the user to match their requirements.

In line with the modular application approach, it also seemed prudent to separate the logic of storing and utilising the protocols away from the GUI protocol editor. This would open the prospect for using the protocol functionality in third party applications, with or without the GUI functionality.

## 3.5 Tools

In addition to the Python programming language and the modules that ship with it, a number of third party modules were considered which could ease development and provide additional features to the application.

### 3.5.1 Serial port access

Serial ports behave differently on different hardware and operating systems. Port naming conventions differ, and the methodology for opening and configuring the ports vary considerably. As such, it is a complex task to utilise serial ports in a cross platform manner utilising only the Python standard modules.

As such, an extremely popular Python module was used: pySerial. pySerial encapsulates the code required for serial port access in different operating systems and provides a common interface. It supports Windows and POSIX based systems (Linux, BSD, Mac OS X etc.). (Liechti, 2013) Usage is then as simple as constructing a `Serial` object with the relevant settings, and calling methods to read and write bytes from the serial port.

### 3.5.2 Graphical user interface

Whilst a rudimentary text based user interface can be created using Python's inbuilt functions, in order to create a graphical user interface (GUI) it is again necessary to use additional tools. Popular GUI frameworks for Python tend to have one thing in common, in that they rely on existing toolkits written in languages other than Python, and provide the bindings and necessary conversions to allow interaction from Python modules.

Python includes a GUI package called Tkinter, which uses the third party Tk GUI toolkit. Although being the de-facto GUI toolkit in Python (and hence most likely having good support from the Python community), the features

are relatively basic and create a quite dated looking interface. This is demonstrated in Figure 14; the settings dialog for IDLE, an editor built with wxWidgets which is supplied with Python. Although a themes package is available – Ttk – the results are less aesthetically pleasing as the other options discussed. Tk also does not have support for complex GUIs such as multi-document interfaces. However, its shortcomings offer an advantage in that the framework is relatively lightweight in terms of resource usage.

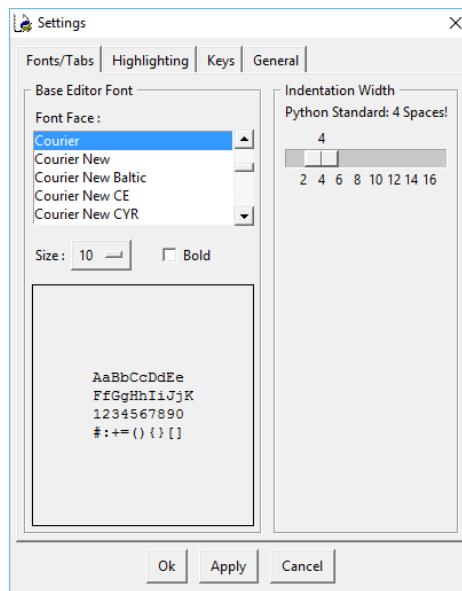


Figure 14: IDLE settings dialog<sup>18</sup>

An alternative to Tk is wxWidgets. wxWidgets utilises the native user interface controls across the platforms that it supports, but provides a standard interface to utilise them. wxWidgets is written in C++ but Python bindings are available and well used, in the form of the wxPython package. wxWidgets includes a large amount of pre-defined user interface widgets allowing many common high-level GUI application features, including multi-document interfaces, the ability to create layouts that adjust according to the window size, creating and editing of tabular data, and selecting items

---

<sup>18</sup> Screenshot of software released under a permissive, open source license (Python Software Foundation License).

such as colours and dates. A major advantage of using wxWidgets over other frameworks is the use of native controls, which gives an appearance and behaviour in keeping with the look and feel of the operating system. On the other hand, such use of native controls can bring a disadvantage – if controls have different behaviours, it can be difficult to determine whether a particular function will operate consistently across all platforms.

Another popular GUI framework is Qt. Qt, being dual licensed with proprietary and open source licenses, is popular in both commercial software and large open source projects, and has backing from The Qt Company; a commercial organisation. Instead of using native platform controls, Qt provides its own, which are themed to appear the same as native controls. This has the advantage of ensuring a consistent behaviour across platforms, but in some cases may behave differently on a particular platform to native applications. Qt also has support for additional functionality such as data manipulation, network access and hardware access.

The original version of SerialPlot used Qt for its GUI framework, which can be seen in Figure 5 on page 27.

A unique feature of the Qt framework is its event loop implementation. GUI frameworks are commonly implemented through use of an event loop. In contrast to traditional procedural programming – where code is executed in a fixed order – an event loop allows code to be triggered when certain events occur, for instance a key being pressed or an on-screen button being clicked, all whilst keeping the GUI up-to-date and responsive. Qt's implementation uses a "signals and slots" mechanism. Signals are triggered when a particular event occurs, and these can be connected to one or more slots, which define the actions to be taken. Using this mechanism, much of the underlying details of dealing with events can be factored out, and signals and slots

connected together as required. For instance, an "undo" button click and a press of the Ctrl-Z shortcut keys can both be connected to a slot, which undoes the last user action.

There are two primary Python bindings for Qt: PyQt and PySide. PyQt is developed by Riverbank Computing, and allows Python to access almost all of the functionality of Qt. PySide is a project which was later developed by The Qt Company (and predecessors) themselves, due to the inability to agree license terms with Riverbank. (PySide Frequently Asked Questions, 2015)

When SerialPlot was originally developed, PyQt was the de-facto implementation; however, throughout the length of the project, PySide grew in popularity. Due to its support from the same company that produced Qt, and licensing terms that are more flexible, it would perhaps be the package of choice for new applications.

Table 3 shows a comparison between the frameworks discussed. After reviewing the various options, it was decided that PyQt was the most suitable for the application. As well as offering the largest feature set, Qt is widely used for both commercial and open source projects, and therefore has a large community following, meaning it was likely that sources of help would be available, as well as third party tools and add-ons.

Application	Cross-platform	Modern interface	Uses native widgets	Good selection of pre-defined widgets	MDI support	Very lightweight	Support
<b>Tkinter</b>	Y	N	N	N	N	Y	Python community (default module)
<b>wxWidgets</b>	Y	Y	Y	Y	Y	N	Less active
<b>Qt</b>	Y	Y	N	Y	Y	N	Commercial support available

Table 3: GUI framework features

### 3.5.3 Mathematical parsing

In order to implement the protocol validation feature, there was a need to parse user entered mathematical expressions from text into a meaningful mathematical expression. The simplest method of achieving this is to use Python's inbuilt `eval` function, which evaluates a text string input as a Python expression. However, it is seen as bad practice to use the `eval` function in this manner, as it could allow a user to inadvertently input dangerous code, potentially crashing the program or causing damage to the system, for instance deleting files. (Hetland, 2008) (Maruch & Maruch, 2006)

A second method of approaching this problem was to use the regular expression (regex) parser in the Python standard library. Regular expressions allow text strings to be searched for particular pattern matches, and as such, actions can be performed based on user input. However, allowing enough flexibility for anything other than trivial mathematical expressions would require a large amount of complex code. As such, it was necessary to turn to third party modules to complete this task.

The pyparsing package is designed for parsing a text input, and allows results to be matched to actions. It provides a number of classes to facilitate

common parsing actions, such as `Word` to match particular groups of characters, and `OneOrMore` to match a repetition one or more times of a given expression. However, after evaluating the package, it was clear that a large amount of code would be required to implement a sufficiently complex parser for this task.

SymPy is symbolic mathematics package for Python, which aims to implement a full-featured computer algebra system. It contains a large amount of modules for completing various mathematical functions, including equation solving, geometric algebra and cryptography. Critically, it also has a number of parsing functions, which allow a text string to be interpreted as a mathematical expression. Not only would this achieve the objective of parsing a user's validation expression, it would also allow such expressions to use the additional functionality provided by SymPy. However, one disadvantage of using SymPy is that – due to its large collection of functions – the package size is relatively large, which would normally be a dependency for installation.

Method	Safe	Easy to implement	Sufficiently efficient	Size of module
<code>eval</code>	N	Y	Y	(Inbuilt)
<code>Regex</code>	Y	N	Y	(Inbuilt)
<code>pyparsing</code>	Y	N	Y	156 kB
<code>SymPy</code>	Y	Y	Y	30 MB

After analysing each option, SymPy was deemed the most appropriate option for the task. However, rather than enforcing the user to install the full package, or supplying it alongside the application, the decision was made to supply only the modules required to implement the parsing functionality, which was a small subset of the package – 176 kB instead of 30 MB.

## 4 Implementation

### 4.1 Program structure

#### 4.1.1 File structure

As discussed in section 3.2, a key objective of the application design was for it to be highly modular, allowing components to be added, removed and replaced, rather than creating a tightly coupled system. Python's support for a hierarchical package and module structure facilitated this. Each Python file is known as a *module* and can be used to implement a specific part of an application. Modules can be grouped together in folders to form *packages*, which can also be placed in other packages. All Python files are automatically modules, but in order for a folder to be used as a package, it must contain a file named `__init__.py`, which can either be empty, or used for package wide functions and configuration.

The file and folder structure eventually created for the application is shown in Appendix A. All files were contained in a parent package called `serialplot`, which kept the code together and would allow re-use of code in other programs without the risk of conflicting package names. The `serialplot` package contained a number of sub-package, as well as modules which made up the core of the application, or otherwise did not fit in to any specific sub-package. The sub-packages consisted of:

- `gui`, which contained the various components of the GUI, including the main window and protocol editor. In addition, this also contained a sub-package to house the various sub-window views.
- `hw`, for current and future *hardware manager* modules, responsible for hardware communication.

- `util`, for miscellaneous and third party utility functions, such as functions for detecting serial ports, and the required SymPy modules.

Figure 15 gives an overview of the various primary components of the application, and how they relate to one another. The various modules are described in more detail in the following sections.

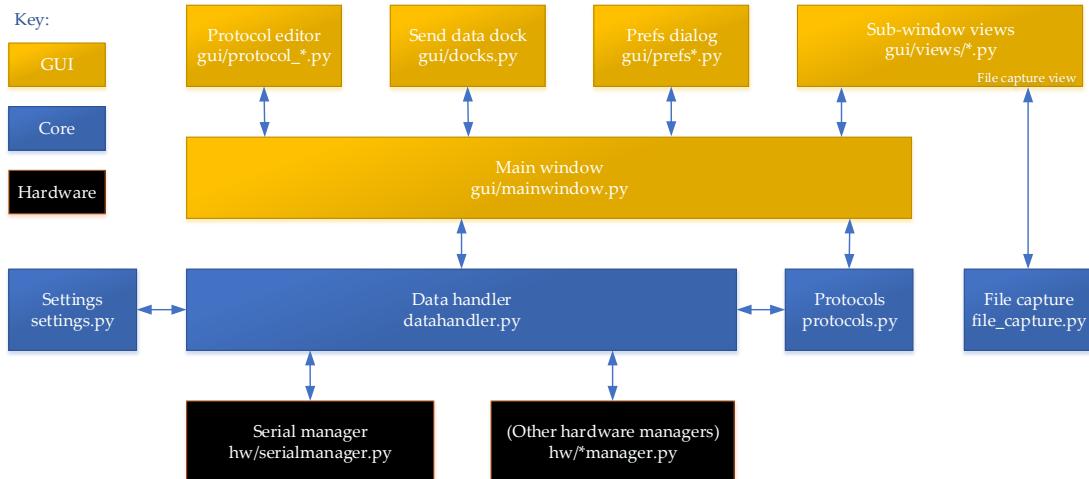


Figure 15: Application architecture

#### 4.1.2 Application logic

In the original version of SerialPlot, incoming data was processed and immediately sent to the various views. This not only made the program highly inefficient, but also created a strong coupling between the data processing and the GUI widgets.

In order to make the application more modular as well as more efficient, it was determined that a suitable implementation would process the data as it was received, but store it in a way which would allow consumers (such as GUI views) to access the processed data only as required. As such, the concept of a *data handler* module was introduced. This would act as a core part of the application, where the other modules would send and receive data. For instance:

- hardware managers would send incoming data to the data handler,

- the data handler would process the data once and store the processed data, and
- views, or other consumers, would retrieve the data from the data handler as required.

#### 4.1.3 Hardware managers

In keeping with the aim of modularity, the concept of a hardware manager was formed. A hardware manager was a module responsible for controlling access and data flow to and from a particular type of hardware device. For the purposes of the project, a serial manager was created to communicate with RS-232 devices, however the idea being that other managers could be created with the same interface, to allow a common mode of communication with the data handler.

A base `HardwareManager` class was created, which defined a minimal implementation which real hardware managers could derive from. Three key methods were provided:

- `setup`: run initialisation code the first time the manager is used.
- `updateSettings`: method which accepts an application settings object and applies the relevant settings.
- `sendData`: used to send data to the device.

##### 4.1.3.1 Serial manager

A `SerialManager` class was created as a sub-class of `HardwareManager`, as an implementation to enable communication using a serial port.

During the development of the original application, an issue had been observed which resulted in a high CPU usage when a continuous stream of data was received into the serial port. (Morris, 2012) It had been determined that a more efficient solution was to check the serial port for new data at a

fixed frequency, and receive all data currently waiting in the UART buffer, regardless of how many bytes – if any – were present. This functionality was re-implemented in the serial manager in the new application. To ensure the optimum timer interval was selected, the results of experimentation can be seen in Appendix B, which demonstrates it to be approximately 50 ms. As such, a Qt timer was utilised to trigger the reading of data. This was instantiated in the `setup` method, and was started and stopped when the port was opened and closed respectively.

The `updateSettings` method was implemented to read settings for the ‘serial’ settings group, and set up the port accordingly. These included the name of the port to open, the baud rate, parity, bit settings, and flow control.

`sendData` simply sent the required data on to the serial port, checking for any errors when doing so. Any errors were broadcast to the rest of the application using a Qt signal, which could then be connected to a slot to notify the user.

In addition to implementing the methods defined in `HardwareManager`, two additional methods were defined which were specific to serial ports:

`openPort` and `closePort`. As the names suggest, these opened and closed the serial ports respectively, and also managed the starting and stopping of the incoming data timer. When a port had been successfully opened or closed, a Qt signal was emitted, to inform the application that the state of the serial port had changed. This could also be triggered if and when the port closed unexpectedly, such as the user unplugging a USB serial port.

#### 4.1.4 The data handler

The data handler was designed to be one of the fundamental building blocks of the application. This would manage the processing and temporary storage

of data, and the movement of it throughout the application. As such, it would have the following main requirements:

- to accept raw input data,
- to process the data using the defined protocols,
- to allow both raw and processed data to be accessed by the views,
- and vice versa for output data.

One particular advantage of the data handler approach was that it removed the necessity for individual GUI components to store temporary data. This paved the way for re-usable views, which could display the same data in multiple formats.

#### 4.1.4.1 Data storage

As computer memory does not have unlimited availability, it was determined that a fixed amount of data would be stored.

The de facto ordered sequence type in Python is the “list”. Whilst highly efficient for random read/write operations at each index, as the list grows, adding and removing items from the start of the list becomes less efficient. (Python Software Foundation, 2015). In order to store the most recent data, there is a requirement to efficiently insert new data onto the end of the sequence whilst removing the oldest data from the start, hence retaining the sequence’s maximum size.

Python’s standard library provides an alternative sequence type known as a “deque” (short for double-ended queue). Unlike a list, this is efficient for adding and removing data from both ends, sacrificing efficiency of retrieving data from the middle. (Python Software Foundation, 2015) The deque has an additional feature which allows the setting of a maximum length; if data is added to one end of the deque, and this length is exceeded, data is

automatically removed from the opposite end. As such, this appeared to be the ideal method to store the data. The usage of the deques in the application is shown in Figure 16. The queue implementation can be viewed as a physical pipe with an access window. When new data arrives into the data handler, it is appended to the right or end of the queue, and when the queue is full, data is automatically removed from the left or start of the queue. Therefore, the newest data is known to always be at the end of the queue, so a section of data, either from a specified location or from the newest, can be obtained as required.

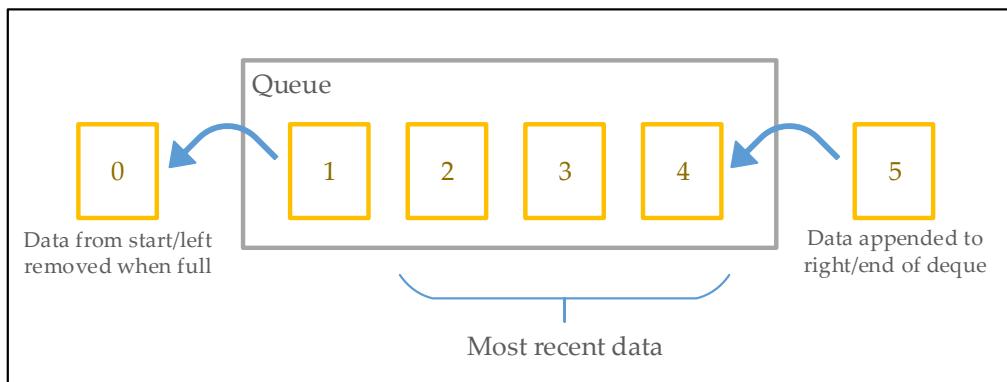


Figure 16: The data queue

The data handler was defined to have several dequeues:

- A raw data queue, to store the unprocessed data as it was received. This allowed views to utilise the incoming raw data.
- A queue for each field, to allow views who required access to specific fields to do so efficiently. The number of these would be changed based on the currently defined protocol.
- A queue to hold data structures representing complete packets.

It should be noted that the third type of queue listed above duplicated data in the second. Although this increased the amount of memory required to store the data, this was done for two related reasons. Firstly, there was a need to store complete packets of data as one. This was of particular

importance to the data capture utility, as correlation between the fields in each packet was critical. However, for views such as the console and graph views, correlation was not as critical, but rather the efficiency of obtaining the required data fields.

#### 4.1.4.2 Data counter

Each consumer (for example, a GUI view) would need to keep track of the most recent data it had received, and hence the data which was required on the next timer iteration.

An initial idea was to tag each data point in a queue with a unique identifier (ID), however this was deemed unnecessary and inefficient for data lookups. Instead, an incrementing counter was used, which effectively represented the ID of the last (most recent) datum in the queue. The current value of this counter (referred to as a “data pointer”) would then be returned alongside the data returned to a consumer, and the consumer could then use this value to request the starting point for the next data request.

A typical lifetime of the view is as follows:

1. The view is created, and requests all data currently in the queue for the selected data field.
2. The data handler returns the data, complete with a data pointer.
3. Upon the next iteration, again the view requests data from the data handler, but passes the pointer to define the first datum to return.
4. The data handler returns the data, starting with the pointer provided.
5. The cycle continues until the view is destroyed.

#### 4.1.5 Protocols

Although much of the protocol logic remained the same, one key development was to decouple this from the GUI. For instance, the original

protocol editor dialog had an option for each field to toggle display on the graph and console views. From a user experience perspective, this was poorly designed, as the protocol editor did not seem the logical place for an option which affected the viewing of data. In addition, moving the control of this option to the individual views would allow further control over which fields were shown on each individual view, increasing the flexibility of the application.

As such, the concept of a protocol handler was developed. Much like the data handler, this provided a key interface to which other application components could communicate, as demonstrated in Figure 15, page 54. Whilst not being definitive, the intention was that two protocol handlers would be used in an application – one to handle incoming data, and the other outgoing.

#### 4.1.5.1 Protocol handlers

A `protocolHandler` object was simply an object which would have a particular protocol assigned to it, and handle the conversion of input data into an encoded data stream, and vice versa. The primary reason that this was introduced was related to the splitting of incoming data over multiple bytes: there was a high possibility that data could have been received by the data handler which may only contain the first part of a data packet.

Therefore, there was a need to keep track of the current position in the parsing sequence, so that when the full packet had been received, the packet could be validated, and parsed fields returned together.

A `protocolHandler` object contained a number of methods for working with the attached protocol, but two were particularly critical, as detailed in Table 4. Hence, the key functionality of the protocol utilisation was made available to the application.

Method name	Input	Output
<code>match</code>	Single data byte from raw data.	If the input byte was the last of a protocol, and the data matched, return the decoded data fields. Otherwise, store the data received so far and return a null value.
<code>create</code>	Mapping of fields to values.	Raw data encoded using the protocol and specified field values.

Table 4: Key protocolHandler methods

An additional check performed by the protocol handler before returning data was to check the decoded packet against the optional validator. This was implemented by the use of two attributes in the protocol; left and right validation expressions. When the validator was configured, the user input was taken as a string, and converted to a SymPy expression, which allowed complex mathematic expressions to be used. The placeholders `f1...fn` were used to represent the field values. At validation time, the `f` placeholders were replaced with the packet's field values, and then compared to see if they were equal. For instance:

- A theoretical protocol exists which consists of three fields, and the third field must be the result of an exclusive-or operation of the first two fields.
- The left expression is set to `f1 ^ f2`. (`^` is the Python symbol for exclusive-or.)
- The right expression is set to `f3`.
- The data 5, 6, 3 is received. These values are substituted into the expressions, which are then compared.
- As  $5 \oplus 6 = 3$ , the validation passes, and the data is passed to the application.

- Next, the data 7, 8, 5 is received. Again, these are substituted in place of the f placeholders, and validated.
- As  $7 \oplus 8 \neq 5$ , the validation fails, and the data is not passed to the application.

#### 4.1.5.2 Protocol objects

In order to define a protocol for the protocol handler to utilise, the concept of a `protocol` object was introduced. Rather than to process data (the job of the protocol handler) the protocol object's methods were focussed around storing the field definitions and other attributes of the protocol.

The `protocol` object was based around a standard Python list object, which is simply a variable sized, ordered collection of other objects; in this specific case, `protocolField` objects. `protocolField` objects in turn were also derived from lists, but here containing integers representing each specific bit:

- 0: Fixed field, 0 bit
- 1: Fixed field, 1 bit
- 2: Data field bit

In practice, for a specific field, only fixed field bits or data field bits would be used, and this would define the overall field type in the user interface.

Figure 17 shows the construction of an example protocol object, to match a protocol with the following definition:

- 1 fixed 8-bit start byte, containing the bit sequence 01010101
- 1 16-bit value, spanning two bytes.

It should be noted that the concept of bytes as transferred over the serial line is masked, to allow full flexibility when defining and implementing protocols.

Protocol																								
Field: start byte, 8 bits								Field: value, 16 bits																
0	1	0	1	0	1	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Figure 17: Example protocol representation

The advantage of storing the protocol in this format is that it is very easy to extract the raw protocol bits in order to match incoming data, but also to differentiate between the individual fields.

## 4.2 Graphical user interface

### 4.2.1 Main window

As discussed in section 3.4.1, the GUI for the application was to be heavily based around the MDI concept, allowing a user to open and manage a number of views in order to display the data in the most suitable format. In the Qt ecosystem, any type of widget can be displayed as a window with no additional wrapper objects, however the supplied `QMainWindow` provides a number of useful features as standard, such as a status bar at the bottom of the window, a menu bar at the top, and the ability to add docked sub-windows to all sides of the window.

Although the application was designed to be modular, the main window module – `mainwindow` – not only managed the appearance of the main window, but also acted as the interface between the application core, and the GUI frontend as a whole. In summary, `mainwindow` was responsible for the following roles:

- Initialising the main window.
- Loading the various view modules, and allowing the user to create and destroy them.

- Notifying the application core when certain events occurred, such as the user choosing to open or close the port, or modifying the protocols.
- Updating the GUI when the application state changed, for instance when the serial port is successfully opened or closed.
- Managing the status bar messages.
- Creating the menu items and managing their various actions.
- Saving and restoring application state upon start-up and shut-down.
- Handling shortcut keys.

In order to add the MDI functionality, a `QMdiArea` widget was set at the main window's central widget, and a "Views" menu added to allow management of the views. To populate the Views menu, a specific sub-package format was used, which simplified the addition of different views:

As shown in Appendix A, all GUI related code was located under the `gui` sub-package. Under this, a further `views` sub-package was created to contain the various sub-window views. The package's `__init__.py` file, as shown in Figure 18, contained a list of views which should be available to the user. Upon application start-up, `mainwindow` iterated through the list of enabled views, and attempted to import each one. If this was successful, then the view would be made available to the user, and an entry added into the Views menu to create a view of that type.

The top part of the main window can be seen in Figure 19, which demonstrates the Views menu having been populated, and the empty MDI area (dark grey colour).

```

enabled_views = [
    'current_value',
    'console',
    'graph',
    'table',
    'file_capture'
]

```

Figure 18: views package `__init__.py`

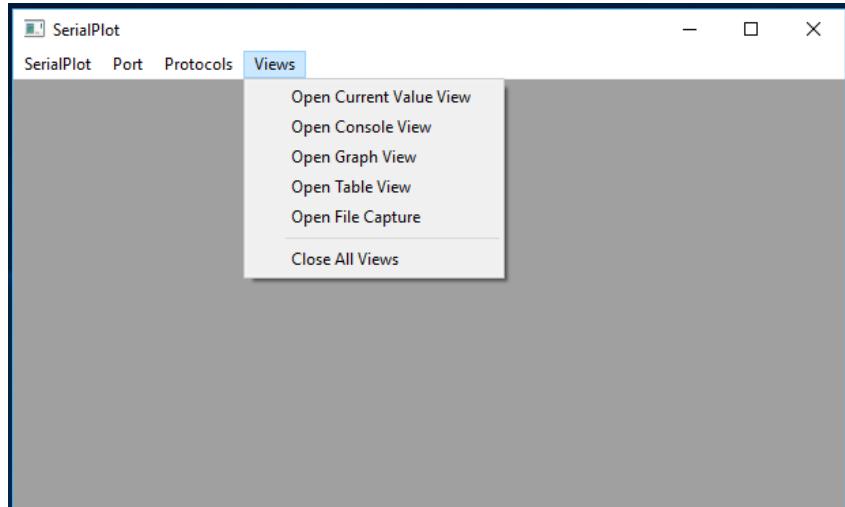


Figure 19: Application main window, with Views menu open

## 4.2.2 Views

### 4.2.2.1 Base classes

Whilst developing the concept of an MDI with multiple types of views, it was clear that most views would have certain aspects in common with one another. Therefore, to reduce code duplication, and simplify creation of new views, base classes were created to implement the shared functionality. Two modules were created for this purpose: `subwindow`; for views which only showed one field, and `subwindow_multifield`; for views which would allow the user to select more than one field. The latter in turn inherited some functionality from the former.

Both base classes implement the following functionality:

- Acceptance of some basic settings into its initialisation method, such as a `dataHandler` object (so the view can retrieve and send data), and the update interval to set how often the view will update.

- Creation of a periodic timer to update the view.
- Methods to get new data, or a specified data set, and – in the former case – manage the data pointer.

Both base classes additionally provided a widget for configuring the displayed field(s). In the case of the single field view, this was simply a dropdown box, however in the multi field view, this was a button which launched a dialog to allow the user to select which fields they wished to be displayed. How the button or drop-down is positioned in the particular sub-window is determined by the specific view's setup code. In cases where the concept of selected fields was not relevant (such as the file capture view), this widget could simply be hidden and ignored.

In order to provide a generic interface for the `mainwindow` module to load arbitrary views, they were defined to adhere to a common standard. This specified that view modules must be a Python module in the `gui.views` package, containing at least the following items:

- A string named `menu_text`, which defined the name of the view in the Views menu.
- A class named `ViewWidget`, which was a subclass of `SubWindowViewWidget` (either the single or multi field version) and defined the view behaviour.

#### 4.2.2.2 Basic views

One of the first views to be developed was a simple view to show the most recent value from a particular field. Whilst not intended to be enabled in a final application, this could be used as an example for other developers wishing to create views, and demonstrated the minimal amount of code required to do so.

Appendix C shows the code used to implement the Example View. As required, the module included a `menu_text` attribute, and a `ViewWidget` class. The menu text contained an ampersand character, which informs Qt that the next character (in this case an ‘x’) should be used as the keyboard shortcut when browsing the menus using the keyboard. The `ViewWidget` class contained two methods, which are special to views: `setup`, which would be called when a view is opened to perform initial configuration, and `updateCallback`, which would be called on every timer tick; by default, every 30 ms.

In this case, `setup` performed a typical initial configuration: setting the sub-window title (displayed in its title bar), and creating a basic layout, with a label (non-editable text field), and the field selection dropdown provided by the base class. The label by default displayed the text “Waiting for data...”, which would later be replaced.

`updateCallback` was constructed to retrieve the most recent datum from the specified field, and display it in the label widget, with the text “Most recent value:” preceding it. This example implementation, although basic, demonstrates two crucial aspects of the application design. Firstly, a timer was used to update the view on a certain frequency. Data could theoretically be received at a much higher rate, however a user is not able to observe changes faster than 30 ms, and updating for every new datum could be highly resource intensive, reducing the efficiency and performance of the application. Also demonstrated is the design decision to let individual views retrieve the data themselves, rather than being passed this in the timer callback from the base class. This is due to the fact that almost every view type had different requirements, for instance the graph view would only require a specific snapshot of data, whereas the table view would need to

ensure that data was contiguous, to allow scrolling back through. In the case of the example view, only the last data point was of interest, so returning any additional data would be unnecessary.

Despite only having been designed for demonstration purposes, feedback from users who had been experimenting with the application indicated that a simple view showing the latest value of a field was a useful tool to provide a status indication, or for demonstration purposes. As such, the example view was developed into the Current Value View. The logic of this view was extremely similar to that of the Example View, the main difference being that the value was displayed with no prefix, and matched the height of the sub-window. This resulted in a view, shown in Figure 20, which could be resized to display the value at different sizes for different use cases.

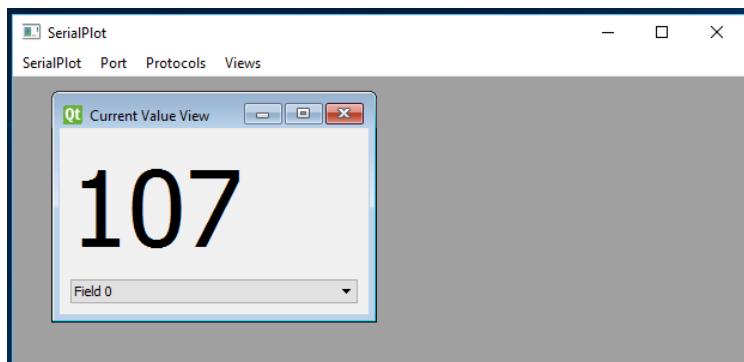


Figure 20: Application main window, displaying a Current Value View

#### 4.2.2.3 Textual display views

The console view was the first view developed which fully exercised the data handler's data pointer functionality. As users may wish to scroll back through data that has been received, it was essential that all data was made available. As such, the `getNewData` method provided by the base class was utilised, which returned any data not yet received by the view. This was configured to be called on each timer tick.

The console view consisted of a read-only text region, where the data would be displayed, and two drop-down buttons at the bottom: the usual field

selection widget, and a drop-down to select the data display type from ASCII, hex, decimal, octal or binary. The option selected was then mapped to an option for Python's built in string formatter, which converted each data value into the correct format. Finally, the formatted data was added to the text region.

Figure 21 shows three Console Views open, displaying the tail end of the same data stream, in three different formats.

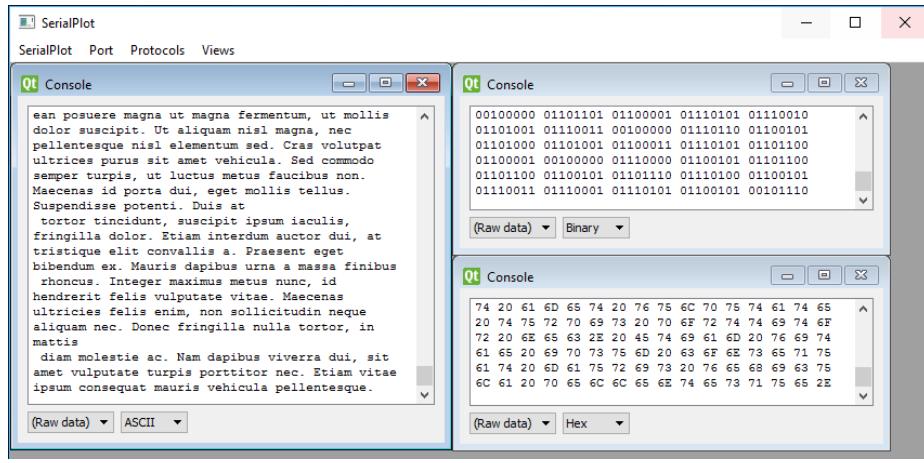


Figure 21: Application main window, with several Console Views open

The second textual view implemented was the Table View. The Table View introduced support for the display of multiple fields in the same view, and utilised a `QTableWidget` to display the data. The logic was much the same as the Console View, but with the added complexity of managing the various table rows and columns. This was much simplified by completely removing all rows and columns when the field selection was changed, then adding the selected columns and re-populating the data. The number of displayed rows were limited by checking the row count, and if exceeding a threshold, removing the oldest rows. The resulting view can be seen in Figure 22, alongside the field selection dialog.

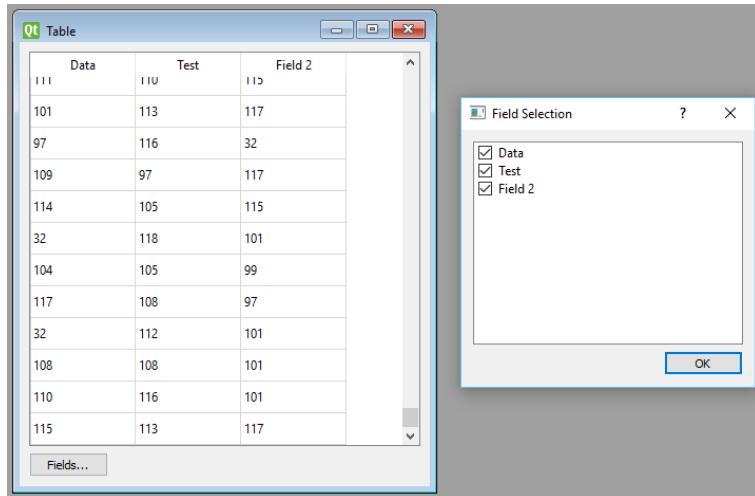


Figure 22: Table View and field selection dialog

#### 4.2.2.4 Graphing

The Graph View was implemented by using a custom subclass of a `QWidget` – the generic, empty widget provided by Qt – and manually drawing the data points onto it. Although some existing third party graphing packages exist for Qt, it was found that implementing the functionality manually provided more flexibility for the needs of the application, and excluded unused features which reduced efficiency.

The graphing implementation in the original application stored the current state of the graph in a pixmap (akin to a virtual drawing surface), and modified this when new data was received, by adding this to the right side, and removing old data from the left. As noted in section 3.3.2, the graphing functionality of the original application suffered visual defects at high data rates, and whilst investigative work was inconclusive, it was highly likely that the pixmap-shifting methodology was a key contributor to this issue.

Having moved the temporary storage of data away from the views and to the data handler, it was no longer necessary to maintain a copy of the data (graphically or otherwise) within the view itself. As such, the graph pixmap could simply be redrawn from scratch on every update. It was found that this method, despite appearing to be counter-intuitive in terms of resource

usage, was in fact highly efficient, and allowed faultless graph plotting even at high speeds.

The colours and key functionality for multiple trace plotting was implemented by assigning particular colours in sequence to the “user data” attribute of the view’s model of each protocol field. When the graph was updated, the required data and colour were retrieved from each field enabled by the user, and the traces drawn accordingly. Likewise, when the sub-window was launched, the protocol changed, or the selected fields changed, the colour information was retrieved from each field to construct the key.

The final implementation of the Graph View can be seen in Figure 23, which demonstrates the real-time plotting of three data fields. The graph widget was implemented to automatically scale the height of the graph to fill the available area, based on the size of the largest field. In this case, all three fields have a size of 8 bits, therefore the range of the graph is automatically set to 0 through 255 ( $2^8 - 1$ ).

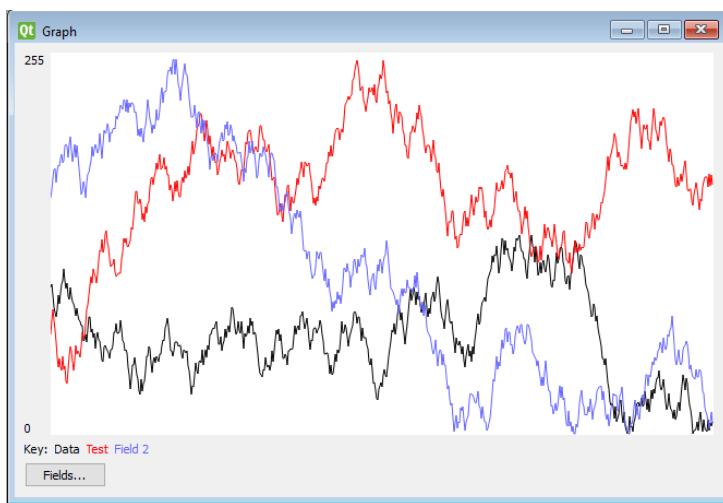


Figure 23: The Graph View, showing three data fields

An additional feature which was added to the Graph View was the ability to scroll through the data; to view the historical data which had scrolled off the

screen. One method to achieve this would have been to use a traditional scroll bar, however this would have resulted in the need to utilise an extremely long pixmap and to plot all available data on every timer iteration. Instead, the arrow keys were utilised to set the data offset. By default, the most recent data was displayed, i.e. the data located at the end of the data handler's queue. By pressing the left arrow key, an offset was added, calculated at a quarter of the currently visible data. On the following graph updates, this offset was used to request the same amount of data, but offset. Likewise, the right arrow key was configured to move forward though the data, and the home and end keys to move to the oldest and newest data respectively. For example:

1. A Graph View is loaded and sized such that 100 data points are visible.
2. Initially, the most recent 100 data points are displayed. If we refer to sample 1 as the most recent sample, then samples 1-100 are displayed.
3. The user presses the left arrow key. This sets an offset of 25.
4. The next time the graph is plotted, samples 26-125 are displayed.
5. The user presses the end key. This resets the offset to 0.
6. The next time the graph is plotted, the current samples 1-100 are displayed.

#### 4.2.2.5 File capture

A File Capture View was created, which added the functionality to save data to a comma separated values (CSV) file using the new interface. The user interface of the view itself, as shown in Figure 24, was very minimal and compact, so as not to use unnecessary screen space. Figure 25 shows an example of data previously exported from the application later imported into

Microsoft Excel and plotted. The column headers are visible, and were automatically added when the file was created.

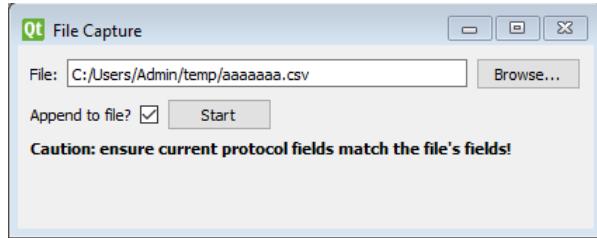


Figure 24: File Capture View

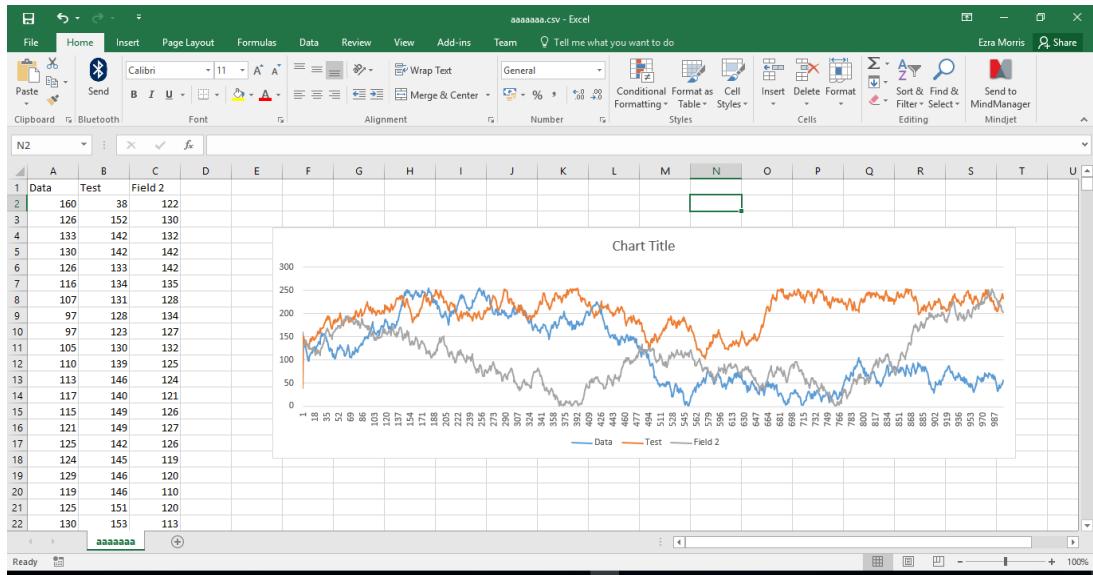


Figure 25: CSV file imported into Microsoft Excel<sup>19</sup>

#### 4.2.3 Data transmission

To provide functionality to send data over the serial port, a **QDockWidget** was used. A dock widget can either be “docked” to the edge of the main window – akin to a toolbox – or “floated” as a separate window. This provides flexibility to the user, as in docked mode, the tools are readily accessible, but can be moved away or closed completely if more space is required.

Initially, the ability to transmit raw data (not encoded using a protocol) in various formats was implemented (Figure 26, A). The process of validating

---

<sup>19</sup> Used with permission from Microsoft.

and converting from the various types was performed by the `dataformats` module. This used regular expressions to validate the data, ensuring only permitted characters had been entered. For example, the regular expression `^[01 ]*$` matched only the characters zero, one and space for the binary data type – the space character was later ignored during the conversion. After validation, the data entered was converted to bits, then sent to the data handler.

Through continued use of the send data functionality, user feedback suggested that having to frequently re-type commonly used data packets was tedious, and a need for an option to save and re-send data was required. As such, the idea of “quick buttons” was introduced. A series of twelve buttons was added to the send data dock (Figure 26, C), plus a mechanism to save the currently entered data to a particular button, optionally with a name. The button text was then changed to the name, if provided, or a representation of the stored data. Upon clicking on the button, the stored data would then be sent. Additionally, the keyboard F-keys (F1 to F12) were assigned to each button, allowing rapid sending of the stored data.

To enable utilisation of output protocols from the GUI, a second means to send data was added. This, like the raw data, had a dropdown box to select the data format, but then had a number of labelled input fields to match the selected protocol, as shown in Figure 26, B.

Figure 26 shows the Send Data dock, in an undocked state, complete with raw data entry and quick button assignment, protocol data entry, a button to toggle visibility of the editing functions (hence only showing the quick buttons), and the quick buttons themselves.

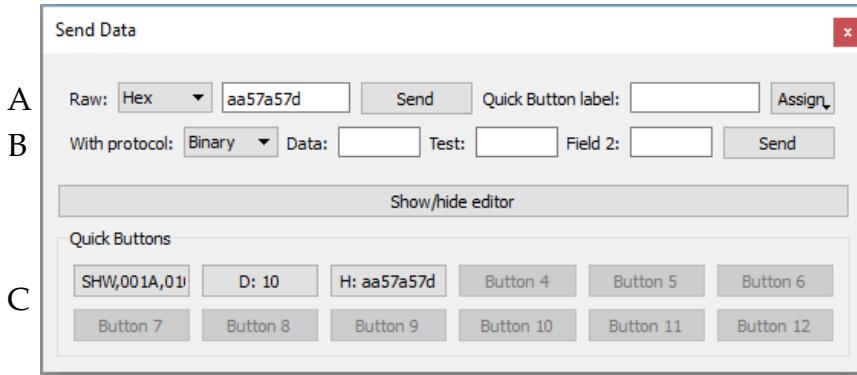


Figure 26: Send Data dock in floating mode

#### 4.2.4 Protocol editor

The protocol editor dialog, as shown in Figure 27, had much the same appearance as the original application, but with two key additional features: the ability to add a validation expression, and provision to save and open protocols to a file.

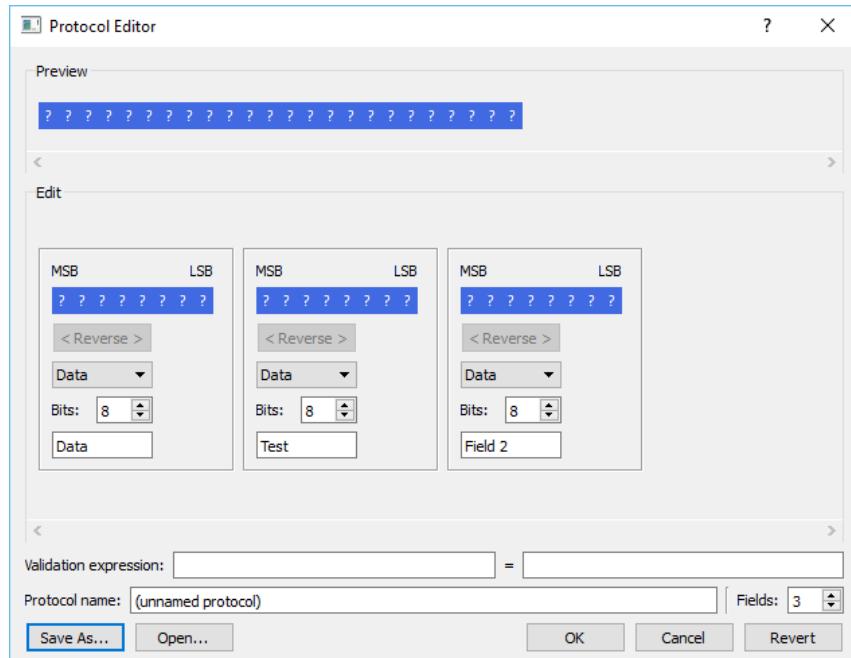


Figure 27: Protocol Editor dialog

The validation expression inputs, as well as the majority of the interface widgets, are simply a front-end to the underlying `protocol` objects.

Upon clicking on the “save as” and “open” buttons, a file selection dialog was shown, which allowed the user to select a protocol file to open or save

to. The file name returned was then used to create a `QSettings` object, which provides a cross platform way to save application settings; in this case the protocol. The `QSettings` methods `setValue()` and `value()` were then used to store and retrieve the protocol settings respectively.

#### 4.2.5 Data persistence

As the application was now very modular, a consistent way to store application state persistently was required. The need for this was two-fold: to maintain the state of the application when it was closed and re-opened, and to allow the saving and loading of the application state to a file (referred to in the GUI as “workspaces”). In order to achieve this, a `StateSaveableUI` base class was created, which provided a common interface to save and restore arbitrary GUI elements. This was used for the main window, the send data dock, and all the sub-windows. Examples of saved settings included the type, size and positioning of sub-windows, the selected field(s) in each sub-window, and the application preferences, such as serial port settings.

By default, on application exit, the settings were saved to the operating system-specific default location, for instance the Windows registry, Mac OS X preferences files, or INI files on Linux. This data was then re-loaded when the application was next started.

For exporting the workspace, INI files were used for all platforms, as this would mean that they would be cross platform if shared across devices. Appendix D shows an example workspace settings file which was exported from the application. Although some attributes have been converted from a binary format and are unreadable, the `prefs`, `subwindows` and `send_dock` sections in particular are human-readable and would be editable.

# 5 Outcomes

## 5.1 Final application

The primary outcome of the project, in line with the project objectives, was the production of a suite of tools, in the form of a software application, to allow users to monitor and control embedded systems.

A modular Python package was created consisting of a suite of core application modules, a serial hardware manager, and a graphical user interface. The full code listing can be seen in Appendix G.

The final GUI can be seen in Figure 28, with a larger copy in Appendix E. In this particular screen shot, the user has opened one of each of the available views, positioned them, and has docked the “send data” dock to the bottom of the window. There is a protocol defined to have three 8-bit fields, named “Data”, “Test” and “Field 2” which are displayed on the graph as well as the other views. Note that the last row of the table view, and hence the most recent data, reads Data = 91, Test = 57, Field 2 = 106. This correlates to the other views, i.e. the current value view is set to Test, and is displaying 57, and the console view is set to Data and is displaying 91 as the final value.

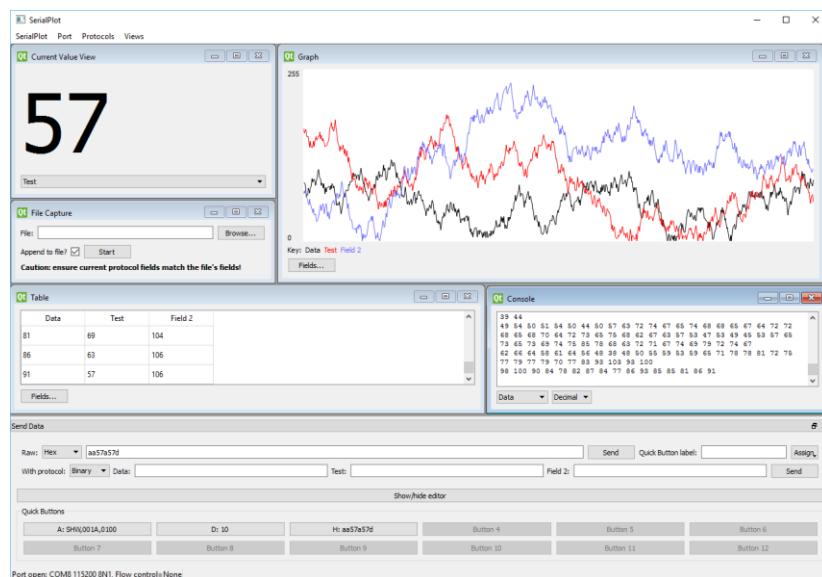


Figure 28: Main window

### 5.1.1 Menus

The application had four menus on the top menu bar: SerialPlot, Port, Protocols and Views, shown in Figure 29, and described below.

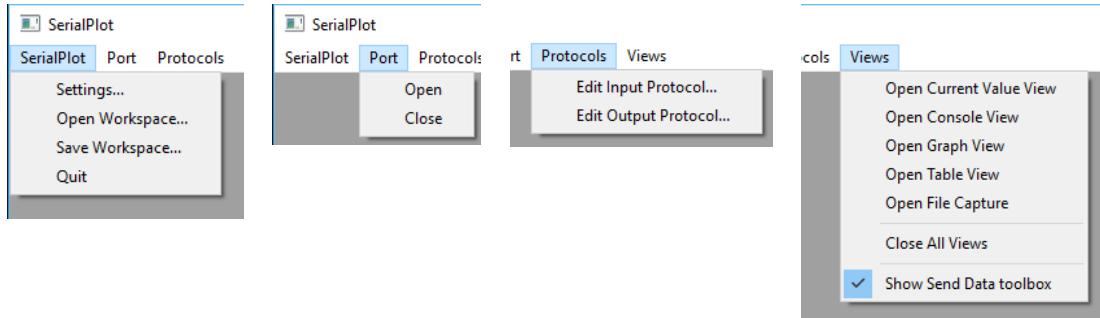


Figure 29: Main window menus

- SerialPlot
  - Settings: opened the settings dialog, as seen in Figure 30. The GUI for this was re-used from the original application, but used a new back-end implementation, to enable settings to be persistent.
  - Open Workspace: displayed a standard operating system open dialog, which allowed a workspace file (as described in section 4.2.5) to be loaded to restore previous settings.
  - Save Workspace: allowed a workspace (GUI layout, settings, protocols etc.) to be saved out to a file.
  - Quit: quit the application.
- Port
  - Open: attempted to open the serial port.
  - Close: closed the serial port.
- Protocols
  - Edit Input Protocol: displayed a protocol editor window (as described in section 4.2.4) to allow the user to change the input protocol.

- Edit Output Protocol: displayed a protocol editor window to allow the user to change the output protocol.
- Views
  - Open ### View: created a new sub-window with the requested view.
  - Close All Views: closed all view sub-windows.
  - Show Send Data toolbox: allowed toggling of the “send data” dock. This was essential because, if the user was to close the dock, they would otherwise have no means to re-open it.

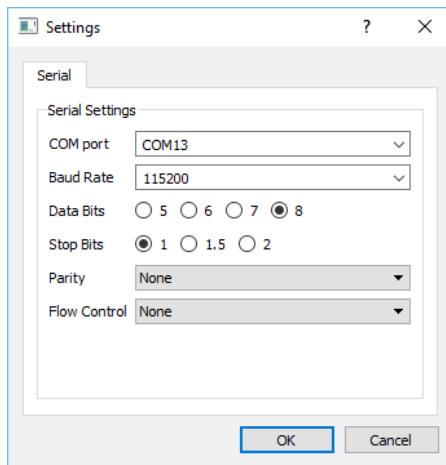


Figure 30: Settings dialog

### 5.1.2 Error messages

A number of error messages could be displayed in the application, to provide a user friendly response to an unexpected event. The two primary error messages were associated with the serial port. If the user tried to open an invalid serial port, an error similar to that in Figure 31 was shown. This displayed the actual error returned from pySerial and the operating system – which could potentially assist the user to identify the source of the problem – as well as a fixed message asking the user to check their settings.

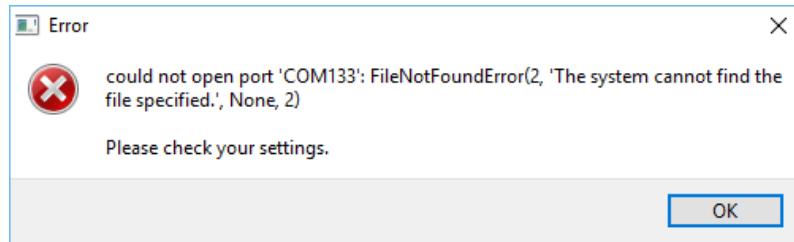


Figure 31: Port not found error

An error similar to that shown in Figure 32 was shown when reading from or writing to the serial port failed. The most likely occurrences of this error were if a USB to serial converter was disconnected whilst in use, or if data was attempted to be sent from the application when the port was closed, but it could also occur if there was a hardware failure. Again, a generic error message is printed (“An error occurred whilst reading from/writing to the serial port”) as well as the error message returned from pySerial and the operating system.

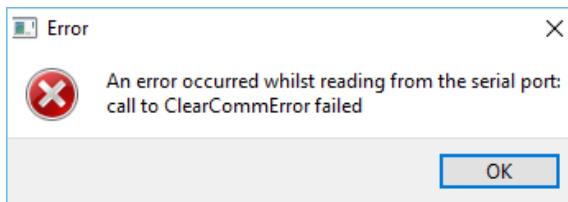


Figure 32: Error when failed to read from serial port

## 5.2 Usage example

One of the objectives of this work was to design a *cross-platform* application. Whilst most of the screenshots in this report have focussed on Windows, the application could run on other operating systems without alteration.

Figure 33 shows the application running on a BeagleBone black – a low cost ARM based development board. The board – shown to the left of the image – has the Debian Linux distribution installed, and a low-resource-intensive desktop environment. The board is connected to a “lapdock” – a device which resembles a laptop but provides a dumb screen, keyboard and track pad which can be connected to an external device – and a mouse. As such,

the video output from the board can be seen on the screen, and keyboard and mouse commands sent to it.

After installing the application pre-requisites as well as the code itself, the `serialplot_gui.py` script was executed, which started the GUI. This can be seen on the screen in Figure 33, a screenshot of which (albeit with different data) can be seen in Figure 34. The GUI layout is identical to when run on Windows. The theming is much more basic, which demonstrates how Qt matches its styling according to the operating system environment.



Figure 33: Application running on a BeagleBone Black

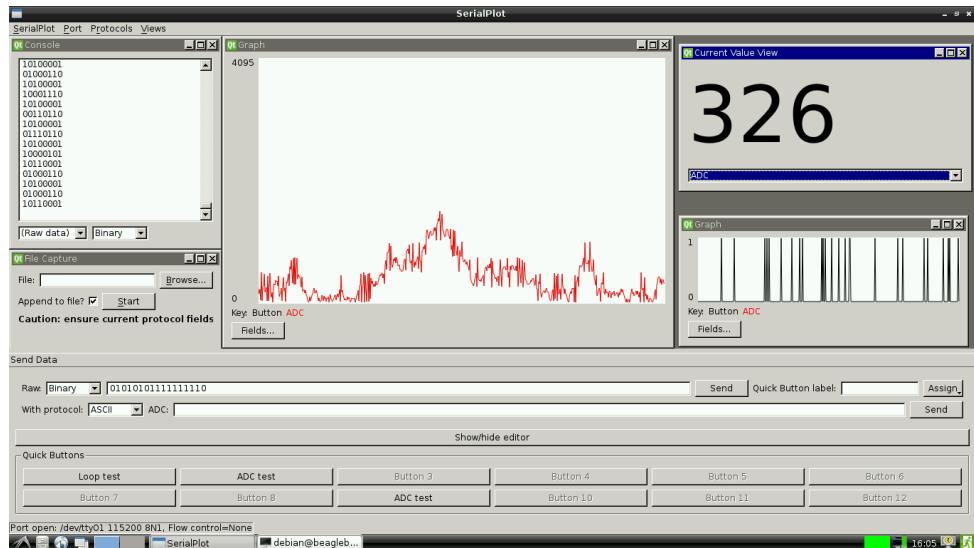


Figure 34: Screenshot of the application running on BeagleBone Black

One of the BeagleBone's UARTs is connected to a device simulating data transmission from a hardware device containing an analogue-to-digital converter (ADC) and a push switch. The protocol is defined in Figure 35, in which three fields are defined, spanning two bytes: the start field – fixed 3 bits of 0, 1, 0; the switch – a single bit; and a 12-bit ADC field.

Protocol												
Start			Switch	ADC								
0	1	0	Switch bit	ADC MSB								ADC LSB
Byte 0									Byte 1			

Figure 35: Example ADC and push switch protocol

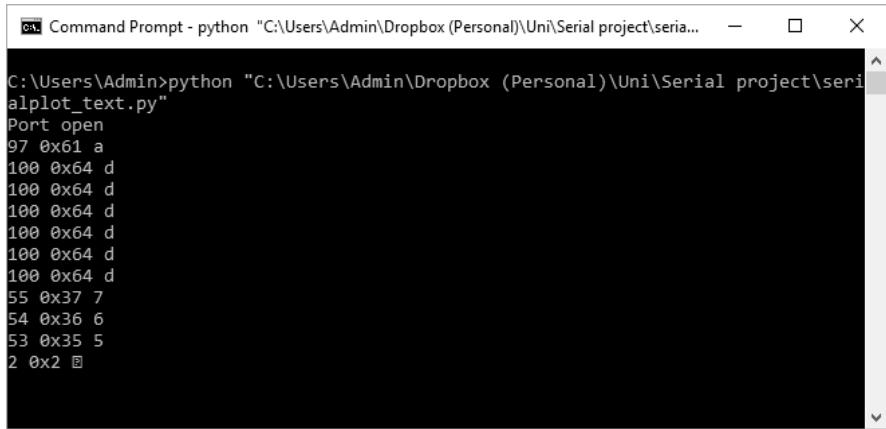
The console view in Figure 34 – displaying raw data – shows that, as expected, the first byte in each pair begins with 010. Two graph views are displayed, showing the ADC and switch fields respectively; this demonstrates the graph scaling: the ADC graph is scaled to 0 to 4095 (as  $2^{12} - 1 = 4095$ ), and the switch field is scaled to 0 to 1 (as  $2^1 - 1 = 1$ ).

### 5.3 Command line interface example

As the application had been written in a modular fashion, the individual components of the application were re-usable without the need for the GUI. Appendix F shows example code for a command line interface (CLI) using the application modules as a back-end. Although the script does not use a GUI, it still makes use of Qt's event loop architecture to connect the signals and slots of the various parts of the modules to the CLI application.

After initial configuration, the script sends each of the characters a, b, c, d, e and f to the serial port, and then sets a timer to check for and print new incoming data every second. This is then printed to the console in various formats.

Figure 36 shows the script running in the Windows command prompt.

A screenshot of a Windows Command Prompt window titled "Command Prompt - python". The command run is "python "C:\Users\Admin\Dropbox (Personal)\Uni\Serial project\serialplot\_text.py"". The output shows several lines of binary data being processed. The first few lines are: "Port open", "97 0x61 a", "100 0x64 d", "55 0x37 7", "54 0x36 6", "53 0x35 5", "2 0x2 ".

```
C:\Users\Admin>python "C:\Users\Admin\Dropbox (Personal)\Uni\Serial project\serialplot_text.py"
Port open
97 0x61 a
100 0x64 d
55 0x37 7
54 0x36 6
53 0x35 5
2 0x2 
```

Figure 36: Example console application output

## 5.4 Performance issues

Although the application had an acceptable level of performance for most scenarios, in some cases performance issues were observed.

### 5.4.1 Bit manipulation efficiency

The protocol mechanism centred around the conversion of all protocols and data into data structures containing the individual bits. Whilst this made the comparison code straightforward, it perhaps was not the most efficient. When using long protocols with many data fields, the application became quite resource intensive, and would often lock up.

Although the protocol editor had been designed to allow the user to specify individual bits, this was perhaps unnecessary for most scenarios. Instead, fields could potentially be specified as multiples of bytes, rather than bits, and bit masks used to ignore specific bits, or ensure certain bits matched. This would likely be more efficient than looping over each bit of each field, which was the method used in the final application.

### 5.4.2 Protocol validation efficiency

Protocols had validation attributes, to allow incoming data to be validated. Whilst this worked for periodic data, for fast data streams this would often freeze the GUI, making the application impossible to work with.

More work would need to be carried out to determine if (a) the protocol validation method itself could be made more efficient and (b) moving the validation activity (and perhaps the protocol decoding in general) to a separate thread would improve matters. The second point is most likely to improve performance, as performing a large amount of resource intensive work on the GUI thread will reduce the resources available to update the GUI. In addition, most modern computers contain multiple core CPUs; a thread can only run on a single core, so offloading processing to a separate thread would harness the multiple core advantages.

#### 5.4.3 Programming language

Python had been selected to enable rapid prototyping and development, and due to the ease of producing a cross-platform application. However, for cases where extremely high performance is required, particularly for lower level, embedded uses, a compiled language may be preferred to meet more stringent performance requirements.

## 6 Conclusion

The outcome of this research resulted in a cross-platform graphical user interface application to allow hardware and firmware developers, engineers, maintenance teams and researchers to monitor, analyse and control a range of embedded systems.

Previous work had been found to have a number of deficiencies, whether inflexible, inefficient, or lacking an intuitive user interface, hence identifying the need for a new solution.

The software produced generally had a high degree of stability, however did face some performance issues when stretching the limits of the application. These could be addressed in future work by reviewing the algorithms used, making better use of multi-core CPUs, or re-implementing some aspects with a compiled programming language.

The application proved to be a useful tool through usage by a number of parties, with very positive feedback, which re-iterates the identified demand for such a tool.

Through the development of software tools such as the one created, which are flexible enough to enable a wide variety of applications, but are simple and intuitive to use and offer an acceptable level of performance, it is envisaged that future development and analysis of embedded systems and communication with such systems can make more efficient use of time, money and human resources.

## 7 References

- Adobe Systems. (2006, August 28). *The Single Document Interface (SDI) for Acrobat 8 for Windows*. Retrieved November 13, 2015, from [http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/single\\_doc\\_interface.pdf](http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/single_doc_interface.pdf)
- ARM Holdings plc. (2014). *Shaping the Connected World: ARM Holdings plc Strategic Report 2014*. Cambridge.
- BCD Microelectronics. (2015). *Serialtest: Asynchronous RS-232 Serial Protocol Analyzer and Packet Sniffer*. Retrieved November 28, 2015, from BCD Microelectronics Online Shop: <http://www.bcdmicro.co.uk/shop.php?pg=4#/Serialtest®-Asynchronous-RS-232-Serial-Protocol-Analyzer-and-Packet-Sniffer/p/43973713/category=10032017>
- Bendersky, E. (2009, August 7). A "live" data monitor with Python, PyQt and PySerial. Retrieved December 12, 2015, from Eli Bendersky's website: <http://eli.thegreenplace.net/2009/08/07/a-live-data-monitor-with-python-pyqt-and-pyserial>
- Combs, G., & contributors. (2015). Enabled Protocols. *Wireshark (application)*. Retrieved from <https://www.wireshark.org/download.html>
- DeBill, E. (n.d.). *Module Counts*. Retrieved November 8, 2015, from <http://www.modulecounts.com/>
- Frontline Test Equipment, Inc. (2012). *Serialtest RS-232/422/485 Serial Analyzer datasheet*. Retrieved November 29, 2015, from [http://www.fte.com/docs/Serialtest\\_datasheet.pdf](http://www.fte.com/docs/Serialtest_datasheet.pdf)
- Hetland, M. L. (2008). *Beginning Python: From Novice to Professional*. New York: Apress.
- Hilgraeve, Inc. (n.d.). *HyperTerminal Private Edition*. Retrieved November 11, 2015, from <http://www.hilgraeve.com/hyperterminal/>
- Ktnbn, & Tardieu, S. (2009). *Diagram of RS232 signalling as seen when probed by an Oscilloscope for an uppercase ASCII "K" character (0x4b) with 1 start bit, 8 data bits, 1 stop bit*. Wikimedia Commons. Retrieved December 20, 2015, from [https://commons.wikimedia.org/wiki/File:Rs232\\_oscilloscope\\_trace.svg](https://commons.wikimedia.org/wiki/File:Rs232_oscilloscope_trace.svg)
- Lamping, U., Ontanon, L. E., & Bloice, G. (2014, December 28). *Adding a basic dissector*, 1.1. Retrieved December 18, 2015, from Wireshark Developer's Guide: [https://www.wireshark.org/docs/wsdg\\_html\\_chunked/ChDissectAdd.html](https://www.wireshark.org/docs/wsdg_html_chunked/ChDissectAdd.html)
- Liechti, C. (2013). *pySerial*. Retrieved November 9, 2015, from <http://pythonhosted.org/pyserial/pyserial.html>

- MacDonald, M. (2006). *Pro .NET 2.0 Windows Forms and Custom Controls in C#*. New York: Apress.
- Maruch, S., & Maruch, A. (2006). *Python For Dummies*. Indianapolis: Wiley Publishing, Inc.
- Microsoft. (2007, January 24). *A new instance of Word appears to run when you create or open an additional document in Word 2000 and in later versions of Word*. Retrieved November 13, 2015, from Microsoft Support: <https://support.microsoft.com/en-us/kb/291313>
- Microsoft. (n.d.). *Multiple Document Interface*. Retrieved November 13, 2015, from Windows Dev Center: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms632591\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms632591(v=vs.85).aspx)
- Morris, E. (2012). *Real-Time Graphical Analysis of RS-232 Serial Data Using Computer Software*. Undergraduate dissertation, Sheffield Hallam University, Faculty of Arts, Computing, Engineering and Sciences, Sheffield.
- PySide Frequently Asked Questions*. (2015, March 26). Retrieved November 12, 2015, from Qt Wiki: [https://wiki.qt.io/PySide\\_FAQ](https://wiki.qt.io/PySide_FAQ)
- Python Software Foundation. (2015, December 21). *collections - Container datatypes*. Retrieved December 23, 2015, from Python 3.4 Documentation: <https://docs.python.org/3.4/library/collections.html>
- Python Software Foundation. (2015, October). *Glossary*. Retrieved November 7, 2015, from Python 3.5.0 Documentation: <https://docs.python.org/3/glossary.html>
- Python Software Foundation. (2015, November 15). *PyPI - the Python Package Index*. Retrieved November 15, 2015, from <https://pypi.python.org/pypi>
- Realterm: Serial Terminal*. (2014, May 7). Retrieved November 22, 2015, from <http://realterm.sourceforge.net/>
- Riverbed Technology. (2015, November 3). *Riverbed Announces Wireshark 2.0*. Retrieved November 29, 2015, from <http://gb.riverbed.com/about/news-articles/press-releases/Riverbed-Announces-Wireshark-2.html>

## Appendix A Application directory structure

```
serialplot
├── gui
│   ├── views
│   │   ├── __init__.py
│   │   ├── console.py
│   │   ├── current_value.py
│   │   ├── example_view.py
│   │   ├── file_capture.py
│   │   ├── graph.py
│   │   ├── subwindow.py
│   │   ├── subwindow_multifield.py
│   │   └── table.py
│   ├── __init__.py
│   ├── docks.py
│   ├── mainwindow.py
│   ├── prefs.py
│   ├── prefs_dialog.py
│   ├── protocol_dialog.py
│   ├── protocol_editor.py
│   ├── protocol_widgets.py
│   ├── resources.py
│   ├── ui_state.py
│   └── util.py
├── hw
│   ├── test
│   │   ├── __init__.py
│   │   └── test_stdmodel.py
│   ├── __init__.py
│   ├── manager.py
│   └── serialmanager.py
└── util
    ├── sympy
    │   ├── core
    │   │   ├── basic.py
    │   │   ├── cache.py
    │   │   ├── compatibility.py
    │   │   ├── decorators.py
    │   │   ├── numbers.py
    │   │   └── sympify.py
    │   └── parsing
    │       ├── __init__.py
    │       ├── mathematica.py
    │       ├── sympy_parser.py
    │       └── sympy_tokenize.py
    ├── __init__.py
    ├── list_ports.py
    ├── list_ports_posix.py
    └── list_ports_windows.py
    ├── __init__.py
    ├── dataformats.py
    ├── datahandler.py
    ├── file_capture.py
    ├── protocols.py
    └── serialplot_gui.py
    └── settings.py
```

## Appendix B Serial receive tests

The application was executed on a computer with a dual core Intel Atom CPU running Windows 7. The third party 'RealTerm' application was used to send a file filled with pseudo-random data to a virtual COM port, which was connected to another virtual COM port from which this data was read. The baud rate was set to 115200, with 8 data bits, 1 stop bit, no parity and no flow control. The timer interval was set to several different values, and the CPU usage was monitored using the Windows Task Manager.

Figure 37 shows the application running in an idle state, which demonstrates a typical system CPU level in normal usage.

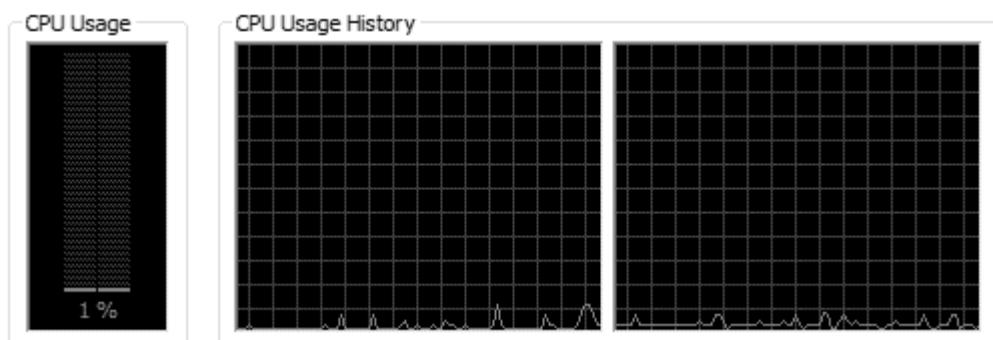


Figure 37: Control (application running, connected to port, no data transfer)

As shown in Figure 38, when the interval was 0, the CPU usage was relatively high, but suddenly increased to 100% on both CPUs part way through the transfer, presumably because the Qt application settled to an idle state.

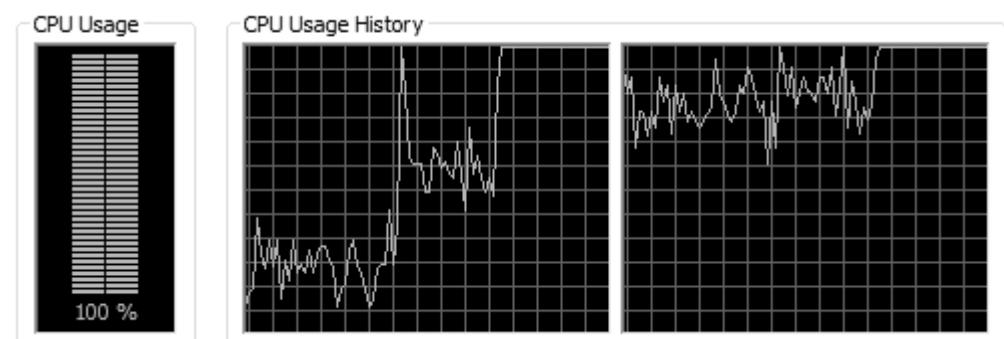


Figure 38: Interval = 0 (data read whenever application idle)

Figure 39 and Figure 40 demonstrate that, up to a limit, as the timer interval was increased, the CPU usage decreased.

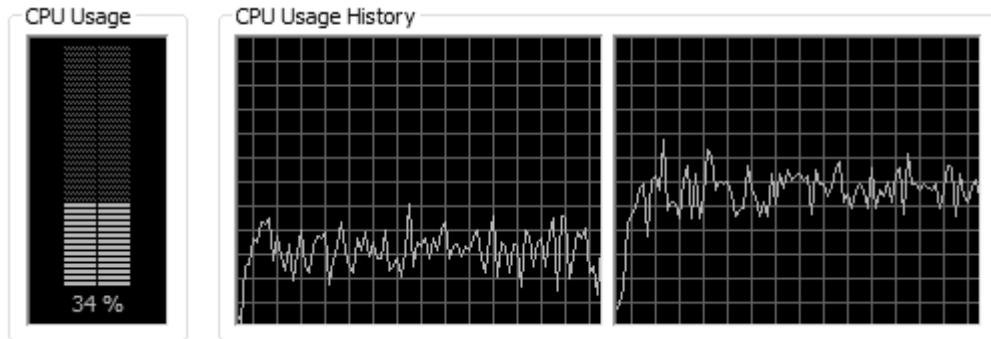


Figure 39: Interval = 10 ms

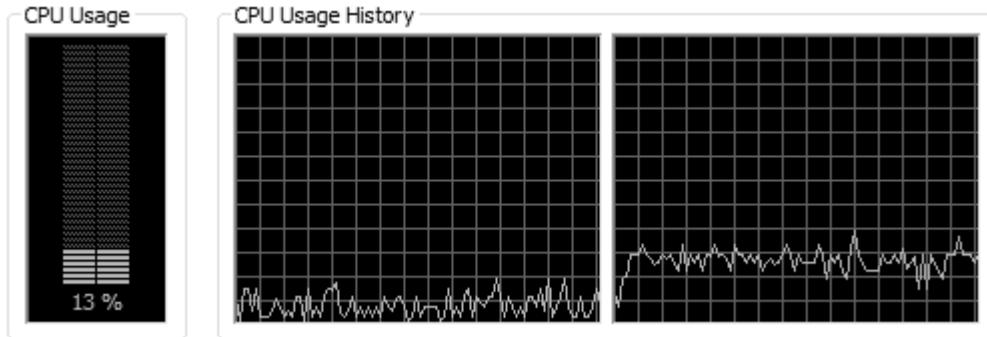


Figure 40: Interval = 50 ms

As shown in Figure 41, at 100 ms, the CPU usage fluctuated more, but was generally higher. This was most likely due to the fact that more data had to be processed on each tick, but the ticks were further apart.

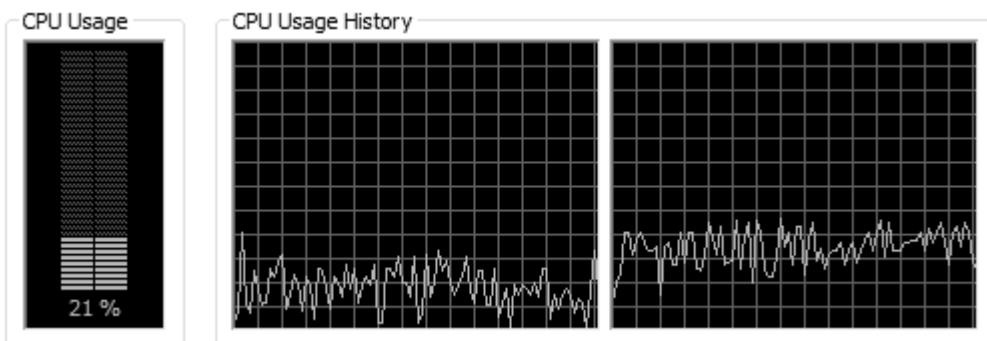


Figure 41: Interval = 100 ms

## Buffer overflow testing

A small command line Python application was written, which printed the number of bytes in the serial buffer, and, if the buffer was full, the string

“OVERFLOW!”. The contents of the buffer were then read and discarded, and the program slept for a specified time. A file of random data was sent using RealTerm.

Interval = 0:

The buffer is almost always empty before the read, indicating that the interval is too short.

Interval = 10 ms:

Interval = 100 ms

180	360	360	360	360	180	180	180	180	360	180	180	180	180	180	360	360	360	300	180	360	180	180
309	180	180	180	180	360	360	180	180	360	360	180	360	360	180	296	180	180	180	360	360	360	360
180	180	360	360	300	180	360	360	180	180	360	360	360	186	180	180	180	360					

Interval = 1 s;

2880	2700	2700	2700	2700	2700	2700	2880	2160	2340	1980	2880	2880	2520	2700	1980	2520
2880	2880	2880	2880	2700	2880	2700	2700	2700	2880	2700	2880	2700	2520	2520	2820	
2880	2880	2880	2760	1260	2700	2700	2880	2820	2880	2880	2880	2880	2700	2700	2880	

The buffer is about half full before each read. This is acceptable, but if extra load is placed on the system, this can cause the buffer to overflow.

Interval = 2 s;

The buffer is always full, so the application never “catches up” until the transfer is stopped. This leads to a lag, which increases the longer the data is continuously sent.

## Appendix C Example View code

```
#!/usr/bin/python3

from PyQt4 import QtCore, QtGui

from .subwindow import SubWindowViewWidget

menu_text = "E&xample View"

class ViewWidget (SubWindowViewWidget):
    def setup(self):
        self.setWindowTitle("Example View")

        layout = QtGui.QVBoxLayout()

        self.valueLabel = QtGui.QLabel("Waiting for data...")

        layout.addWidget(self.valueLabel)
        layout.addWidget(self.fieldDropdown)

        self.setLayout(layout)

    def updateCallback(self):
        # Get most recent data point (returns an iterator or None)
        data = self.getData(1)

        # No data yet
        if data is None: return

        try:
            # Just get the actual data item
            data = next(data)
            self.valueLabel.setText("Most recent value: " + str(data))
        # There was no data item
        except StopIteration:
            pass
```

## Appendix D Example workspace file

```

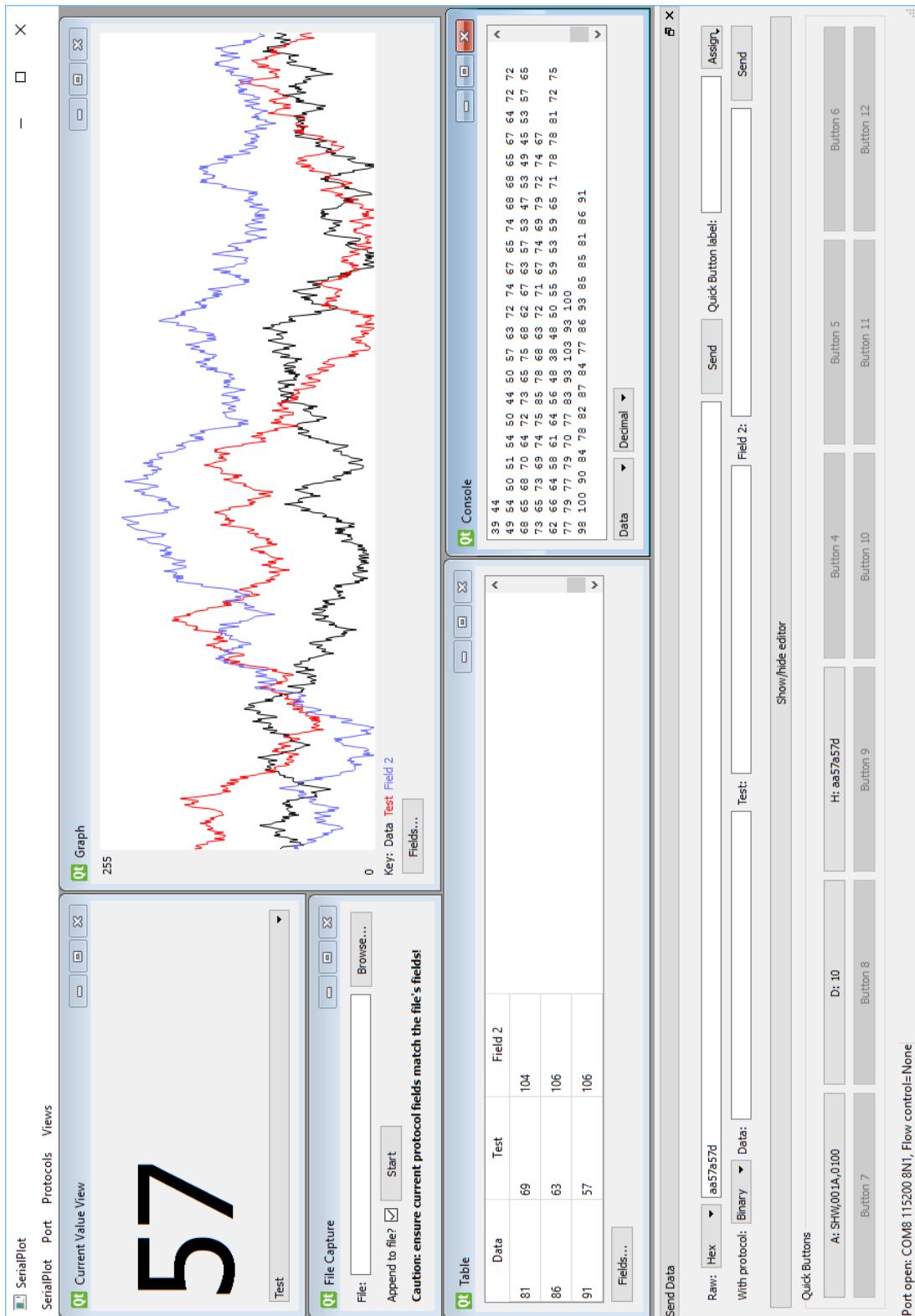
[General]
geometry=@ByteArray(\x1\xd9\xd0\xcb\0\x1\0\0\0\0\x5N\xff\xff\xff\xf8\0\0\0]\0\0\x3\xe9\0\0\x5\xdb\0\0\0\xac\0\0\t\x85\0\0\x3m\0\0\0\x1\x2\0)
state=@ByteArray(\0\0\0\xff\0\0\0\xfd\0\0\0\x1\0\0\x3\0\0\xab\0\0\0\xe6\xfc\x1\0\0\0\x1\xfb\0\0\0\x1c\0\x65\0n\0\x64\0\0\x44\0\x61\0t\0\x61\0\x64\0o\0\x63\0k\x3\0\0\x5\x63\0\0\x1\xc9\0\0\x2\x19\0\0\xd0\0\0\x5\0\0\x3\x2\0\0\0\x4\0\0\x4\0\0\b\0\0\b\xfc\0\0\0\b)
inputProtocol=@Variant(\0\0\0\x7f\0\0\xePyQt_PyObject\0\0\x1\x1e\x80\x3\x63serialplot.protocols\nprotocol\nq\0)\x81q\x1(cserialplot.protocols\nprotocolField\0\nq\x2)\x81q\x3(K\x2K\x2K\x2K\x2K\x2K\x2K\x2K\x2\x65}q\x4X\x4\0\0\0nameq\x5X\x4\0\0\x44\x61taq\x6sbh\x2)\x81q\aa(K\x2K\x2K\x2K\x2K\x2K\x2K\x2K\x2\x65}q\xbh\x5X\x4\0\0\0Testq\xtsbh\x2)\x81q\hn(K\x2K\x2K\x2K\x2K\x2K\x2K\x2K\x2\x65}q\xvh\x5X\aa\0\0\x46ield2q\xfsbe}q\r(X\x4\0\0\xvalLq\xeNX\x4\0\0\x_fIdq\xfK\x3h\x5X\x12\0\0\xunnamed protocol)q\x10X\x4\0\0\xvalRq\x11Nub.)
outputProtocol=@Variant(\0\0\x7f\0\0\xePyQt_PyObject\0\0\x1\x14\x80\x3\x63serialplot.protocols\nprotocol\nq\0)\x81q\x1(cserialplot.protocols\nprotocolField\0\nq\x2)\x81q\x3(K\x2K\x2K\x2K\x2K\x2K\x2K\x2K\x2\x65}q\x4X\x4\0\0\0nameq\x5X\x4\0\0\x44\x61taq\x6sbh\x2)\x81q\aa(K\x2K\x2K\x2K\x2K\x2K\x2K\x2K\x2\x65}q\xbh\x5X\x4\0\0\xTes tq\xtsbh\x2)\x81q\hn(K\x2K\x2K\x2K\x2K\x2K\x2K\x2K\x2\x65}q\xvh\x5X\aa\0\0\x46ield2q\xfsbe}q\r(X\x4\0\0\xvalLq\xeNX\x4\0\0\x_fIdq\xfK\x3h\x5X\b\0\0\x33 fieldsq\x10X\x4\0\0\xvalRq\x11Nub.)
[prefs]
serial\baud=115200
serial\parity=N
serial\flowControl=None
serial\stopBits=1
serial\dataBits=8
serial\port=COM8

[subwindows]
1\class=<class 'serialplot.gui.views.file_capture.ViewWidget'>
1\geometry=@Rect(-2 247 412 139)
1\field=-1
2\class=<class 'serialplot.gui.views.current_value.ViewWidget'>
2\geometry=@Rect(0 0 413 240)
2\field=-1
3\class=<class 'serialplot.gui.views.graph.ViewWidget'>
3\geometry=@Rect(441 8 830 355)
3\field=-1
3\fields=0, 1, 2
4\class=<class 'serialplot.gui.views.table.ViewWidget'>
4\geometry=@Rect(577 373 687 284)
4\field=-1
4\fields=0, 1, 2
size=4

[send_dock]
format=2
data{text=aa57a57d}
quickbuttons\1\data="SHW,001A,0100"
quickbuttons\1\type=0
quickbuttons\1\use_prot=false
quickbuttons\1\label="A: SHW,001A,0100"
quickbuttons\2\data=10
quickbuttons\2\type=3
quickbuttons\2\use_prot=false
quickbuttons\2\label=D: 10
quickbuttons\3\data=aa57a57d
quickbuttons\3\type=2
quickbuttons\3\use_prot=false
quickbuttons\3\label=H: aa57a57d
quickbuttons\size=3

```

## Appendix E Application main window



## Appendix F CLI example

```
#!/usr/bin/python3

import sys

# Import serial module to use constants
import serial
# Import QtCore module to run a Qt command line application
from PyQt4 import QtCore

# Import the required serialplot components
from serialplot.datahandler import DataHandler
from serialplot.hw.serialmanager import SerialManager
from serialplot.settings import Settings

pointer = None

def main():
    # Create Qt CLI application
    app = QtCore.QCoreApplication(sys.argv)

    # Improves debugging
    QtCore.pyqtRemoveInputHook()

    # Create data handler and settings objects
    dh=DataHandler()
    settings = Settings()

    # Create a hardware manager using the above
    sm=SerialManager(dh, settings)

    # Create a dictionary of serial settings, then update the manager
    d={'port': 'COM13',
       'baud': 115200,
       'dataBits': serial.EIGHTBITS,
       'stopBits': serial.STOPBITS_ONE,
       'parity': serial.PARITY_NONE,
       'flowControl': "None"
      }
    settings.updateSettingsGroup('serial', d)
    sm.updateSettings()

    # Connect signals, so data from data handler goes to serial manager,
    # and errors are printed on the command line
    dh.dataSent.connect(sm.sendData)
    sm.error.connect(print)

    # Open the serial port
    sm.openPort()

    # Send some data using data handler and default protocol
    s="abcdef"
    for c in s:
        dh.sendData(c.encode('ascii'), False)

    # Start timer to call print_data every second
    t = QtCore.QTimer()
    t.timeout.connect(lambda: print_data(dh, app))
    t.start(1000)

    sys.exit(app.exec_())
```

```
def print_data(dh, app):
    global pointer

    # Get all (raw) data in data handler
    tagged = dh.getNewData(pointer)
    data = tagged.data
    pointer = tagged.pointer

    # No data; return
    if data is None:
        return

    # Print various representations of each data value
    for val in data:
        print(val, hex(val), chr(val))

# Run main function when called directly
if __name__ == '__main__':
    main()
```

## Appendix G Program code

File paths are given in relation to the serialplot package root

`__init__.py`

(Empty file)

`dataformats.py`

```
import re

from serialplot import protocols

DATA_TYPE_ASCII=0
DATA_TYPE_BINARY=1
DATA_TYPE_HEX=2
DATA_TYPE_DEC=3

# Exception when invalid data (e.g. "5" when binary specified) is used
class InvalidDataException(Exception):
    pass

# Exception when an unspecified type is used
class InvalidTypeException(Exception):
    pass

# Check if data (and type) is valid
# data: data to validate (str)
# typ: data type to validate against (int - use constants)
# returns: True if data is valid for type, else False (bool)
def validate_data(data, typ):
    if (typ==DATA_TYPE_ASCII): return True
    if (typ==DATA_TYPE_BINARY and re.match(r"^[01 ]*$", data)): return True
    if (typ==DATA_TYPE_HEX and re.match(r"^[0-9A-Fa-f ]*$", data)): return True
    if (typ==DATA_TYPE_DEC and re.match(r"^[0-9 ]*$", data)): return True

    return False

# Convert data to binary
# data: data to validate (str)
# typ: data type to validate against (int - use constants)
# width: for decimal type, specifies number of bits per value. Other types this is
# ignored
# returns: binary representation of data (generator of ints where each int={0,1})
def to_binary(data, typ, width=8):
    if (not(validate_data(data, typ))):
        raise InvalidDataException

    # For decimal (not aligned to binary), we use space to split values
    if typ == DATA_TYPE_DEC:
        for val in data.split(' '):
            for bit in protocols.byteToBits(int(val), width):
                yield bit

    else:
        for c in data:
            try:
                # Ignore spaces if not ASCII
```

```

        if typ != DATA_TYPE_ASCII and c == " ":
            continue

        # If binary, just convert to bit
        if (typ == DATA_TYPE_BINARY):
            yield (int(c, 2))

        # If ascii, convert to bits then yield
        elif (typ == DATA_TYPE_ASCII):
            # Find integer value of char
            val=ord(c)
            for bit in protocols.byteToBits(val, 8):
                yield bit

        # If hex, convert to int, then to bits, then yield
        elif (typ==DATA_TYPE_HEX):
            val=int(c, 16)
            for bit in protocols.byteToBits(val, 4):
                yield bit

        else:
            raise InvalidTypeException

    except ValueError:
        raise InvalidDataException

# For testing
if __name__ == "__main__":
    def _test(data, typ):
        print (list(map(str, to_binary(data, typ)))))

    print ("Testing binary 01010101")
    _test('01010101', DATA_TYPE_BINARY)

    print ("Testing ASCII U=>01010101")
    _test('U', DATA_TYPE_ASCII)

    print ("Testing hex 0x55=>01010101")
    _test('55', DATA_TYPE_HEX)

    print ("Testing hex 0x10=>10000101")
    _test('85', DATA_TYPE_HEX)

```

## datahandler.py

```
from collections import deque
from itertools import islice

from PyQt4 import QtCore

from serialplot import protocols

class DataHandler(QtCore.QObject):
    dataSent = QtCore.pyqtSignal(object)
    _queue_maxlen=10000

    def __init__(self):
        super().__init__()

        # "queue" holds the raw incoming data
        self.queue = self._makeQueue()

        # "procqueue" holds tuples of the fields for each packet
        self.procqueue = self._makeQueue()

        # Each "fieldqueue" holds the data for a particular field
        self.fieldqueue = None

        self.bitsize=8

        self.default_prot = protocols.protocol()

    self.default_prot.append(protocols.protocolField([protocols.bitTypeReq]*self.bitsize))

    self.setInputProtocol(self.default_prot)
    self.setOutputProtocol(self.default_prot)

    def setInputProtocol(self, protocol):
        if not(protocol):
            protocol = self.default_prot

        self._in_prot_h = protocols.protocolHandler(protocol, self.bitsize)

        num_fields = len(self.inputProtocol())
        self.fieldqueue = list(self._makeQueue() for _ in range(num_fields)) # Add queue for each field

    def inputProtocol(self):
        return self._in_prot_h.protocol

    def setOutputProtocol(self, protocol):
        if not(protocol):
            protocol = self.default_prot

        self._out_prot_h = protocols.protocolHandler(protocol, self.bitsize)

    def outputProtocol(self):
        return self._out_prot_h.protocol

    def sendData(self, data, use_prot):
        if use_prot:
            processed_data = self._out_prot_h.createEx(data)
        else:
            processed_data = protocols.bitsToBytes(data)
        self.dataSent.emit(processed_data)
```

```

def addNewItem(self, data):
    self.queue.append(data)

    processed_data = self._in_prot_h.match(data)
    if processed_data is not None:
        self.procqueue.append(processed_data)
        for item in processed_data:
            # Add each item to the queue for its field
            self.fieldqueue[item.field].append(item)

def _makeQueue(self):
    return DataQueue(maxlen = self._queue maxlen)

def clearQueue(self):
    self.queue.clear()

def queueLength(self):
    return len(self.queue)

def queueMaxLength(self):
    return self._queue maxlen

def _getQueue(self, field):
    if(field is None):
        return self.queue
    elif field == -1:
        return self.procqueue
    else:
        return self.fieldqueue[field]

# "start" and "end" are counted from the right (most recent) of the queue
def getData(self, start=None, end=None, field=None):
    """
    NB: If field is None, raw data will be returned.
        If field is -1, tuple of all fields will be returned.
    """
    q = self._getQueue(field)

    # Start not specified => return an iterator over the full queue
    if(start==None): return iter(q)

    # End not specified => return all to most recent
    if(end==None): end=0

    q_len=len(q)

    try:
        return islice(q, q_len-start, q_len-end)
    except ValueError:
        return iter(q)

def getNewData(self, latest=None, field=None, limit=None):
    """
    limit, if specified, gives the maximum number of points to be returned.
    If limit > num of available points, the latest _limit_ points will be
    returned.
    NB: If field is None, raw data will be returned.
        If field is -1, tuple of all fields will be
    """
    q = self._getQueue(field)
    itemid = q._itemid

```

```

        if(latest is None):
            if (limit is None):
                data = self.getData(field=field)
            else:
                data = self.getData(limit, field=field)
        elif(latest>=itemid):
            data = None
        else:
            if limit is None or limit>(itemid-latest):
                data = self.getData(itemid-latest, field=field)
            else:
                data = self.getData(limit, field=field)

    return TaggedData(data, itemid)

class DataQueue(deque):
    def __init__(self, maxlen):
        super().__init__(maxlen = maxlen)
        self._itemid = 0

    def append(self, x):
        super().append(x)
        self._itemid+=1

    def extend(self, it):
        try:
            length=len(it)
        except TypeError: # Will be raised if 'it' has no len
            it = list(it)
            length = len(it)
        super().extend(it)
        self._itemid+=length

    def appendleft(self, *args):
        raise NotImplementedError("Appending to the left would mess up _itemid")

    def extendleft(self, *args):
        raise NotImplementedError("Extending the left would mess up _itemid")

    def pop(self, *args):
        raise NotImplementedError("Popping from the right would mess up _itemid")

    def remove(self, *args):
        raise NotImplementedError("Removing random items would mess up _itemid")

    class TaggedData(object):
        def __init__(self, data, pointer):
            self.data=data
            self.pointer=pointer

    class DataPointer(int):
        def __repr__(self):
            return ("DataPointer(" + str(self) + ")")

```

## file\_capture.py

```
import itertools

from PyQt4 import QtCore

# Object to manage data capture to file
class capturer(QtCore.QObject):
    # Signals to indicate stopping and starting of capture
    # Stopped, especially, should be connected, as it may stop without being
    requested
        # (e.g. IOError on write())
    started=QtCore.pyqtSignal()
    stopped=QtCore.pyqtSignal()
    fileOpenError=QtCore.pyqtSignal(str)

    def __init__(self):
        super().__init__()

        # File handle
        self.file=None

    # Begin capture to specified file
    # fileName: name of file to capture to (str)
    # append: append instead of overwrite? (bool) (default False)
    def start_capture(self, fileName, headings, append=False):
        # File mode - overwrite or append
        if append:
            mode='a'
        else:
            mode='w'

        try:
            # Open file in correct mode
            self.file=open(fileName, mode=mode)
            #
            # We are at the start of the file
            if (not append) or (self.file.tell() == 0):
                self.write_headings(headings)
        except IOError as s:
            self.fileOpenError.emit(str(s))
        else:
            self.started.emit()

    # Stop file capture
    def stop_capture(self):
        if(self.file is not None):
            self.file.close()
            self.file=None

        self.stopped.emit()

    # Is the capture running?
    # returns: (bool)
    def running(self):
        if(self.file is None):
            return False
        else:
            return True

    # Wrapper for write_data, replacing empty strings with good guess name
    # headings:
    def write_headings(self, headings):
        self.write_data(heading if heading else "Data "+str(i)
                       for heading, i in zip(headings, itertools.count(0)))
```

```
# 'data' should be an iterable of strings
def write_data(self, data):
    if(self.file is None):
        return
    try:
        # Print items separated by commas
        self.file.write(", ".join(map(str, data)) + "\n")
    except IOError:
        self.stop_capture()
        raise

# For testing
if(__name__ == "__main__"):
    import sys
    c=capturer()
    # Can't call open() on stdout
    c.file=sys.stdout
    c.write_headings(("Foo", "Bar", "", "Baz", None))
    c.write_data(("33423", "345345", "0", "3453", "145"))
```

## protocols.py

```
#!/usr/bin/python3

import copy
from itertools import count

from PyQt4 import QtCore
from sympy.parsing.mathematica import mathematica as parse_math

bitTypeZero=0
bitTypeOne=1
bitTypeReq=2

# Converts byte to bits
# b - byte
# n - number of bits
def byteToBits(b, n):
    for i in range(n-1, -1, -1):
        yield (b>>i & 1)

# and vice versa
def bitsToByte(bits):
    b=0
    for i, bit in enumerate(reversed(bits)):
        b+=bit<<i

    return b

def bitsToBytes(bits, bits_per_byte=8):
    bits=tuple(bits)
    for i in count(0, bits_per_byte):
        curbits = bits[i:i+bits_per_byte]
        if curbits:
            yield bitsToByte(curbits)
        else:
            break

# Protocol is a list of protocol fields
class protocol(list):
    def __init__(self, fields=[]):
        super().__init__(fields)

        # Placeholder for name
        self.name=None

        # Auto-incremented field ID
        self._fId = 0

        # Validation
        self.valL=None
        self.valR=None

        for field in fields:
            self._fixFieldName(field)

    # Block append being called directly
    def append(self, field):
        self._fixFieldName(field)

        # Calls parent's append()
        # Important to actually add to list!
        super().append(field)
```

```

# Disable some other list functions
def extend(self, *args):
    raise NotImplementedError()
def insert(self, *args):
    raise NotImplementedError()

def _fixFieldName(self, field):
    if (not field.name):
        field.name = "Field " + str(self._fId)

    self._fId+=1

# Yields a raw stream of protocol bits
def rawProtocol(self):
    for field in self:
        for bit in field:
            yield bit

# Returns number of bits in protocol
def numBits(self):
    return sum(len(n) for n in self)

# Returns an iterable of data field names
def dataFieldNames(self):
    for field in self:
        if(field[0]==bitTypeReq):
            yield field.name

def setValidator(self, valLStr, valRStr):
    if(valLStr is not None and valRStr is not None):
        self.valL=parse_math(valLStr)
        self.valR=parse_math(valRStr)

# Protocol field is a list of bits
class protocolField(list):
    def __init__(self, *args, name=None):
        super().__init__(*args)

        self.name=name

    def isDataField(self):
        if self[0] == bitTypeReq:
            return True
        else:
            return False

# This class is used when returning matched data
# Field contains the field ID
class protocolMatch(int):
    def __init__(self, field, value):
        self.field=field

    def __new__(cls, field, value):
        return int.__new__(cls, value)

class protocolHandler(QtCore.QObject):
    newData=QtCore.pyqtSignal(int)

    def __init__(self, protocol=None, byteSize=8):
        super().__init__()

        self.setProtocol(protocol)
        self.byteSize=byteSize

```

```

# Empty list to hold matched data bits
self._req=[]

# Reset location markers to 0
self._fieldLoc=0
self._bitLoc=0

self._data=list()

def setProtocol(self, protocol):
    self.protocol=protocol
    self.reset()

def reset(self):
    # *_Loc represents the current location in the protocol arrays
    self._fieldLoc=0
    self._bitLoc=0
    # Reset _req
    self._req=[]

    # _data will contain a list of ProtocolMatch()es
    self._data=list()

def _currentBit(self):
    return self.protocol[self._fieldLoc][self._bitLoc]

# Returns True if we're on the last bit of the field
def _endOfField(self):
    return ((self._bitLoc+1)>=len(self.protocol[self._fieldLoc]))

# Returns True if we're on the last bit of the protocol
def _endOfProtocol(self):
    return ( self._endOfField() and (self._fieldLoc+1)>=len(self.protocol) )

# Increment the current bit locations
def _incLoc(self):
    self._bitLoc+=1
    if(self._bitLoc>=len(self.protocol[self._fieldLoc])):
        self._bitLoc=0
        self._fieldLoc+=1

# Validate data
# Only applicable when full protocol has been matched
def validate(self):
    # No validation set? - return true
    if(self.protocol.vall is None or self.protocol.valR is None):
        return True

    # Iterator of field replacements
    repl=({"f"+str(i+1), data} for i, data in enumerate(self._data))

    repl=tuple(repl)
    vL=self.protocol.vall.subs(repl)
    vR=self.protocol.valR.subs(repl)

    if(vL==vR):
        return True
    else:
        return False

def match(self, b):
    # If protocol not set, just return the data
    if (self.protocol is None):
        #return (protocolMatch(b),)

```

```

        raise Exception("Protocol not set!")

    # Return value - None by default
    data=None

    for bit in byteToBits(b, self.byteSize):
        # Get current bit
        pb=self._currentBit()

        # Reset (failed match) if a fixed bit doesn't match
        if ((pb==bitTypeZero and bit!=0) or (pb==bitTypeOne and bit!=1)):
            self.reset()
            break
        # (We don't need to do anything for ignored)
        # Required/data bits
        elif(pb==bitTypeReq):
            # Append the bit
            self._req.append(bit)

            # Check to see if we are at the end of the field, if so, append the
            data to the list
            if(self._endOfField()):
                self._data.append(protocolMatch(self._fieldLoc,
                    bitsToByte(self._req)))
                self._req=[]

            # If we've reached the end of the protocol, return the data, reset then
            skip the rest of the byte
            if (self._endOfProtocol()):
                data=tuple(self._data)
                # Validate data
                if(not(self.validate())): data=None
                self.reset()
                break

            # Increment protocol pointers
            self._incLoc()

    return data

# Create data using the protocol and a given input
# data should be a list of bits
def create(self, data, asbits=False):
    if(self.protocol is None):
        finalBits=list(data)
    else:
        field=None
        # Local copy of protocol
        toSend=copy.copy(self.protocol)
        # Select the (first) data field, if one exists
        for i in range(len(toSend)):
            # Check the bit type of the first bit in the field
            if(toSend[i][0]==bitTypeReq):
                field=i
                break

        if(field is not None):
            # Replace the field with 0s (as padding if not enough data)
            toSend[field]=[0]*len(toSend[field])

            # Loop round the bits of the field, using the data, or 0 if not
            enough
            # Replace the data in place
            i=0
            for bit in data:

```

```

        toSend[field][i]=bit
        i+=1

    finalBits=list(toSend.rawProtocol())

    # Replace erroneous bits with 0
    for i in range(len(finalBits)):
        if(finalBits[i]>1): finalBits[i]=0

    if asbits:
        for bit in finalBits:
            yield bit
    else:
        # Take chunks of data of the byte size, convert to bytes then yield
        while(len(finalBits)>=self.byteSize):
            b=bitsToByte(finalBits[0:self.byteSize])
            yield b
            del finalBits[0:self.byteSize]

        # Yield anything left over
        if(finalBits):
            yield bitsToByte(finalBits)

def createEx(self, data, asbits=False):
    # Extended create function which allows multiple fields to be specified
    # Specified in a dictionary of field offset (int)->field (protocolField)
    toSend=copy.copy(self.protocol)

    for (fId, fData) in data.items():
        toSend[fId] = fData

    finalBits=list(toSend.rawProtocol())
    for i in range(len(finalBits)):
        if(finalBits[i]>1): finalBits[i]=0

    if asbits:
        for bit in finalBits:
            yield bit
    else:
        # Take chunks of data of the byte size, convert to bytes then yield
        while(len(finalBits)>=self.byteSize):
            b=bitsToByte(finalBits[0:self.byteSize])
            yield b
            del finalBits[0:self.byteSize]

        # Yield anything left over
        if(finalBits):
            yield bitsToByte(finalBits)

# For testing, when called directly
if (__name__ == "__main__"):
    print()
    print("Testing byte to bits, A->01000001")
    bits=[bit for bit in byteToBits('A'.encode("ascii")[0], 8)]
    print("".join(repr(bits)), "which is a", type(bits), "of", type(bits[0]))

    print()
    print("And back again...")
    byte=bitsToByte(bits)
    print(byte, "or", "{:c}".format(byte), "which is a", type(byte))

    print("Protocol = 01??01?????1?10")
    prot=protocol(
        protocolField((0,1)),

```

```

        protocolField((2,2)),
        protocolField((0,1)),
        protocolField((2,)*6),
        protocolField((1,)),
        protocolField((2,)),
        protocolField((1,0)),
    )))
ph=protocolHandler(prot)

print(str(prot))
print()

print("Running 'FJ' (01001010 01000110) through protocol. Should match, and
output 36 (001001):")
data='FJ'.encode("ascii")
for c in data:
    data=ph.match(c)
    if(data is not None):
        print(str(data[1]))
    else:
        print("No match")
print()

print("Running 'AA' (01000001 01000001). Should fail:")
data='AA'.encode("ascii")
for c in data:
    data=ph.match(c)
    if(data is not None):
        print(str(data[0].data))
    else:
        print("No match")
print()

print("Embedded match (twice), 'AAJAAFAJFAJF':")
data='AAJAAFAJFAJF'.encode("ascii")
for c in data:
    data=ph.match(c)
    if(data is not None):
        for item in data:
            print(str(item.field), ":", str(item))
print()

print("Field names:")
for field in prot:
    print(field.name)

```

## serialplot\_gui.py

```
import sys

from PyQt4 import QtGui
from PyQt4.QtGui import QApplication

from serialplot.gui import resources

app = QApplication(sys.argv)

# Display splash screen before loading most of modules
pm = QtGui.QPixmap(":/splash.png")
splash = QtGui.QSplashScreen(pm)
splash.show()
app.processEvents()

from PyQt4 import QtCore

from serialplot.gui/mainwindow import MainWindow
from serialplot.datahandler import DataHandler
from serialplot.hw.serialmanager import SerialManager
from serialplot.settings import Settings
from serialplot.gui.util import showError

# Improves debugging
QtCore.pyqtRemoveInputHook()

dh=DataHandler()
settings = Settings()
sm=SerialManager(dh, settings)
win = MainWindow(dh, settings)

dh.dataSent.connect(sm.sendData)
sm.error.connect(showError)

win.openPort.connect(sm.openPort)
win.closePort.connect(sm.closePort)
sm.portOpen.connect(win.setPortStatusOpen)
sm.portClosed.connect(win.setPortStatusClosed)
win.inputProtocolChanged.connect(dh.setInputProtocol)
win.outputProtocolChanged.connect(dh.setOutputProtocol)

win.setWindowTitle("SerialPlot")
win.show()

# Finish splash screen when window finished displaying
splash.finish(win)

sys.exit(app.exec_())
```

## settings.py

```
#!/usr/bin/python3

import serial

from PyQt4 import QtCore

class Settings(QtCore.QObject):
    """Holds program settings.

    Attributes:
        (Setting groups are stored in dictionaries of setting:value pairs.)
        general -- General settings.
        serial -- Serial settings.
    """

    settingsChanged = QtCore.pyqtSignal()

    def __init__(self):
        super().__init__()

        self.data = dict()

        self.data['general'] = dict(
            # Any general settings would go here
        )

        # Serial specific settings
        # If more hardware types are added,
        # this may need to be more tightly integrated with the module
        self.data['serial'] = dict(
            port = None,
            baud = 9600,
            dataBits = serial.EIGHTBITS,
            stopBits = serial.STOPBITS_ONE,
            parity = serial.PARITY_NONE,
            flowControl = "None",
        )

    def updateSettings(self, settings_dict):
        """Update settings using a nested dictionary.

        Keyword arguments:
            settings_dict -- A nested dictionary with groups and settings.

        Example:
            s=Settings()
            d={
                'group1': {
                    'setting1': 'value',
                    # ...
                },
                # ...
            }
            s.updateSettings(d)

        NB: This method should be used rather than modifying the internal data
        manually, otherwise settings won't actually be changed.
        """
        for group, group_settings in settings_dict.items():
            self.data[group].update(group_settings)
        self.settingsChanged.emit()
```

```

def getSettings(self):
    return self.data

def updateSettingsGroup(self, group, settings_dict):
    """Update a particular group's settings using a dictionary.

    Keyword arguments:
        group -- The group to update.
        settings_dict -- A dictionary of setting: value pairs.

    Example:
        s=Settings()
        d={'setting1': 'value', 'setting2': 123}
        s.updateSettings(d)
    """

    self.data[group].update(settings_dict)
    self.settingsChanged.emit()

def getSettingsGroup(self, group):
    return self.data[group]

def saveToQSettings(self, settings):
    for groupname, groupdata in self.data.items():
        settings.beginGroup(groupname)
        for name, value in groupdata.items():
            if value is None:
                value = -1
            settings.setValue(name, value)
        settings.endGroup()

def loadFromQSettings(self, settings):
    for groupname in settings.childGroups():
        if groupname in self.data:
            settings.beginGroup(groupname)
            for name in settings.childKeys():
                if name in self.data[groupname]:
                    value = settings.value(name)
                    # Error loading
                    if value is None:
                        continue
                    # Try to convert to int
                    try:
                        value = int(value)
                    except ValueError:
                        pass
                    # The "-1" special case
                    if value == -1:
                        value = None
                    self.data[groupname][name] = value
            settings.endGroup()

    self.settingsChanged.emit()

if __name__ == "__main__":
    from pprint import pprint

    settings = Settings()
    pprint(vars(settings))
    settings.updateSettingsGroup('serial', {'baud': 115200})
    pprint(vars(settings))
    pprint(settings.getSettingsGroup("serial"))

```

## gui/\_\_init\_\_.py

(Empty file)

## gui/docks.py

```
from itertools import count

from PyQt4 import QtCore, QtGui

from serialplot import dataformats, protocols
from serialplot.gui.util import showError
from serialplot.gui.ui_state import StateSaveableUI

class SendDataDock(StateSaveableUI, QtGui.QDockWidget):
    # 2 arguments - binary data iterator and bool
    dataSent = QtCore.pyqtSignal(object, bool)

    def __init__(self, dh, parent=None):
        super().__init__()
        self.dh = dh
        self.setParent(parent)
        self.setWindowTitle("Send Data")

        self.setAllowedAreas(QtCore.Qt.TopDockWidgetArea |
                             QtCore.Qt.BottomDockWidgetArea)

        self.sendDataContainer = QtGui.QWidget(self)
        self.sendDataContainer.setSizePolicy(QtGui.QSizePolicy.Preferred,
                                             QtGui.QSizePolicy.Fixed)
        self.setWidget(self.sendDataContainer)

        main_layout = QtGui.QVBoxLayout(self.sendDataContainer)
        self.sendDataContainer.setLayout(main_layout)

        self.editorContainer = QtGui.QWidget(self.sendDataContainer)
        editor_layout = QtGui.QVBoxLayout(self.editorContainer)
        self.editorContainer.setLayout(editor_layout)

        main_layout.addWidget(self.editorContainer)

        toggleEditorButton = QtGui.QPushButton("Show/hide editor", self)
        toggleEditorButton.clicked.connect(self.toggleEditor)
        main_layout.addWidget(toggleEditorButton)

    ### Raw data section ###

    rawLabel = QtGui.QLabel("Raw:", self.editorContainer)
    self.formatCombo = FormatCombo(self.editorContainer)
    self.formatCombo.currentIndexChanged.connect(self.validateData)

    self.dataText = QtGui.QLineEdit(self.editorContainer)
    self.dataText.textChanged.connect(self.validateData)

    self.dataTextValidPalette = self.dataText.palette()
    self.dataTextInvalidPalette = self.dataText.palette()
    invalid_col=QtGui.QColor(255, 170, 170)
```

```

        self.dataTextInvalidPalette.setColor(self.dataText.backgroundRole(),
invalid_col)

        self.sendButton = QtGui.QPushButton("Send", self.editorContainer)
        self.sendButton.setCheckable(False)
        self.sendButton.clicked.connect(self.sendTextData)

        self.dataText.returnPressed.connect(self.sendButton.click)
        self.sendButton.clicked.connect(self.dataText.selectAll)
        self.sendButton.clicked.connect(self.dataText.setFocus)

        labelLabel = QtGui.QLabel("Quick Button label:", self.editorContainer)
        self.labelText = QtGui.QLineEdit(self.editorContainer)
        self.labelText.setSizePolicy(QtGui.QSizePolicy.Preferred,
                                     QtGui.QSizePolicy.Fixed)
        self.assignDropdown = AssignDropdown(self.editorContainer)

        data_layout = QtGui.QHBoxLayout()
        data_layout.addWidget(rawLabel)
        data_layout.addWidget(self.formatCombo)
        data_layout.addWidget(self.dataText)
        data_layout.addWidget(self.sendButton)
        data_layout.addWidget(labelLabel)
        data_layout.addWidget(self.labelText)
        data_layout.addWidget(self.assignDropdown)
        editor_layout.addLayout(data_layout)

    ### Protocol data section ###

    protLabel = QtGui.QLabel("With protocol:", self.editorContainer)

    self.pformatCombo = FormatCombo(self.editorContainer)
    self.pformatCombo.currentIndexChanged.connect(self.validateData)

    # self.dataText = QtGui.QLineEdit(self.editorContainer)
    # self.dataText.textChanged.connect(self.validateData)

    self.psendButton = QtGui.QPushButton("Send", self.sendDataContainer)
    self.psendButton.setCheckable(False)
    self.psendButton.clicked.connect(self.sendFieldData)

    #self pdataText.returnPressed.connect(self.psendButton.click)
    #self.psendButton.clicked.connect(self pdataText.selectAll)
    #self.psendButton.clicked.connect(self pdataText.setFocus)

    self.pfield_layout = QtGui.QHBoxLayout()

    self.passignDropdown = AssignDropdown(self.sendDataContainer)
    # TODO: Make Quick Button work with protocol
    self.passignDropdown.hide()

    pdata_layout = QtGui.QHBoxLayout()
    pdata_layout.addWidget(protLabel)
    pdata_layout.addWidget(self.pformatCombo)
    #pdata_layout.addWidget(self.dataText)
    pdata_layout.addLayout(self.pfield_layout)
    pdata_layout.addWidget(self.psendButton)
    pdata_layout.addWidget(self.passignDropdown)
    editor_layout.addLayout(pdata_layout)

    # Dict to store the text boxes so we can access the correct field
    self.pFields = {}

### Quick Button section ###

```

```

        quickbutton_groupbox = QtGui.QGroupBox("Quick Buttons",
self sendDataContainer)
        quickbutton_layout = QtGui.QGridLayout(quickbutton_groupbox)

        self.quickButtonData = [] # To store data and type of quick buttons
        self.quickButtons = QtGui.QButtonGroup(quickbutton_groupbox)
        self.quickButtons.setExclusive(False)
        self.quickButtons.buttonClicked[int].connect(self.sendQuickButtonData)
        for i in range(12):
            text = "Button {}".format(i+1)
            button = QtGui.QPushButton(text, quickbutton_groupbox)
            button.setEnabled(False)
            button.setSizePolicy(QtGui.QSizePolicy.Ignored,
QtGui.QSizePolicy.Fixed)
            self.quickButtons.addButton(button, i)
            quickbutton_layout.addWidget(button, i//6, i%6)

            self.quickButtonData.append(_QuickButtonData())

            self.assignDropdown.menu().addAction(text, self._makeAssignLambda(i))
main_layout.addWidget(quickbutton_groupbox)

        self.addSettingsItem("format", int,
            self._makeComboGetLambda(self.formatCombo),
            self._makeComboSetLambda(self.formatCombo))
    )
    self.addSettingsItem("datatext", str,
        self.dataText.text,
        self.dataText.setText
    )

def toggleEditor(self):
    if self.editorContainer.isVisible():
        self.editorContainer.hide()
    else:
        self.editorContainer.show()

def currentData(self):
    return self.dataText.text()

def currentType(self):
    return self.formatCombo.itemData(self.formatCombo.currentIndex())

def currentPType(self):
    return self.pformatCombo.itemData(self.pformatCombo.currentIndex())

def validateData(self):
    if dataformats.validate_data(self.currentData(), self.currentType()):
        self.dataText.setPalette(self.dataTextValidPalette)
    else:
        self.dataText.setPalette(self.dataTextInvalidPalette)

def sendData(self, data, typ, use_prot):
    try:
        if(use_prot):
            binary = {}
            prot = self.dh.inputProtocol()
            for fId in data:
                width = len(prot[fId])
                binary[fId] = dataformats.to_binary(data[fId], typ, width)
        else:
            binary = dataformats.to_binary(data, typ, 8)

    except dataformats.InvalidDataException:
        showError("Invalid data entered for specified format type.")

```

```

        else:
            self.dataSent.emit(binary, use_prot)

    def sendTextData(self):
        data = self.currentData()
        typ = self.currentType()
        self.sendData(data, typ, False)

    def updateProtocolFields(self):
        # Clear all existing widgets
        item = self.pfield_layout.itemAt(0)
        while(item is not None):
            w = item.widget()
            self.pfield_layout.removeWidget(w)
            w.setParent(None)

            item=self.pfield_layout.itemAt(0)
        # Clear the dict
        self.pFields = {}

        for field, i in zip(self.dh.outputProtocol(), count()):
            if(field[0] == protocols.bitTypeReq):
                l = QtGui.QLabel(field.name+":", self.sendDataContainer)
                self.pFields[i] = QtGui.QLineEdit(self.sendDataContainer)
                self.pfield_layout.addWidget(l)
                self.pfield_layout.addWidget(self.pFields[i])

    def sendFieldData(self):
        data = {}
        for fId, fWidget in self.pFields.items():
            data[fId] = fWidget.text()

        self.sendData(data, self.currentPType(), True)

    def assignQuickButton(self, buttonid, data, typ, type_text, use_prot,
label=None):
        bd = self.quickButtonData[buttonid]
        bd.data = data
        bd.typ = typ
        bd.use_prot = use_prot

        button = self.quickButtons.button(buttonid)
        if label:
            text = label
        elif (self.labelText.text()):
            text = self.labelText.text()
        else:
            text = "{0}: {1}".format(type_text[0], data)
        button.setText(text)
        bd.label = text
        button.setEnabled(True)

    def assignQuickButtonFromText(self, buttonid):
        data = self.currentData()
        typ = self.currentType()
        typeText = self.formatCombo.currentText()

        if data:
            self.assignQuickButton(buttonid, data, typ, typeText, False)

    def sendQuickButtonData(self, buttonid):
        data = self.quickButtonData[buttonid].data
        typ = self.quickButtonData[buttonid].typ
        use_prot = self.quickButtonData[buttonid].use_prot
        if data is None or typ is None: return

```

```

        self.sendData(data, typ, use_prot)

    def fKeyPressed(self, key):
        self.quickButtons.button(key).click()

    # Extend saveUi to save quickbuttons
    def saveUi(self, settings):
        super().saveUi(settings)

        settings.beginWriteArray("quickbuttons")
        for i, buttondata in enumerate(self.quickButtonData):
            if buttondata.data is not None:
                settings.setArrayIndex(i)
                settings.setValue("data", buttondata.data)
                settings.setValue("type", buttondata.typ)
                settings.setValue("use_prot", buttondata.use_prot)
                settings.setValue("label", buttondata.label)
        settings.endArray()

    # Extend restoreUi to restore quickbuttons
    def restoreUi(self, settings):
        super().restoreUi(settings)

        for i in range(settings.beginReadArray("quickbuttons")):
            settings.setArrayIndex(i)
            data = settings.value("data")
            if data is None: continue
            typ = settings.value("type", type=int)
            type_text = self.formatCombo.itemText(self.formatCombo.findData(typ))
            use_prot = settings.value("use_prot", type=bool)
            label = settings.value("label")
            self.assignQuickButton(i, data, typ, type_text, use_prot, label)
        settings.endArray()

    def _makeAssignLambda(self, i):
        return lambda: self.assignQuickButtonFromText(i)

    def _makeComboGetLambda(self, obj):
        return lambda: obj.itemData(obj.currentIndex())

    def _makeComboSetLambda(self, obj):
        return lambda d: obj.setCurrentIndex(obj.findData(d))

    class _QuickButtonData(QtCore.QObject):
        def __init__(self, data=None, typ=None, use_prot=True):
            super().__init__()

            self.data = data
            self.typ = typ
            self.use_prot = use_prot

        class FormatCombo(QtGui.QComboBox):
            def __init__(self, parent=None):
                super().__init__(parent)
                self.setEditable(False)
                self.addItem("ASCII", dataformats.DATA_TYPE_ASCII)
                self.addItem("Hex", dataformats.DATA_TYPE_HEX)
                self.addItem("Binary", dataformats.DATA_TYPE_BINARY)
                self.addItem("Decimal", dataformats.DATA_TYPE_DEC)

        class AssignDropdown(QtGui.QToolButton):
            def __init__(self, parent=None):
                super().__init__(parent)
                self.setText("Assign")
                self.setToolTip("Assign current data to a Quick Button")

```

```
self.setPopupMode(QtGui.QToolButton.InstantPopup)
self.setToolButtonStyle(QtCore.Qt.ToolButtonTextOnly)
self.setSizePolicy(QtGui.QSizePolicy.Fixed, QtGui.QSizePolicy.Minimum)
self.setMenu(QtGui.QMenu(self))

# Show a window with docks in place if called directly
if __name__ == "__main__":
    import sys

    app = QtGui.QApplication(sys.argv)
    win = QtGui.QMainWindow()

    win.setCentralWidget(QtGui.QLabel(" (main widget)", win))

    sdd = SendDataDock(win)
    sdd.dataSent.connect(print)
    win.addDockWidget(QtCore.Qt.BottomDockWidgetArea, sdd)

    win.show()
app.exec_()
```

## gui/mainwindow.py

```
from importlib import import_module

from PyQt4 import QtCore, QtGui

from serialplot import protocols
import serialplot.gui.views
from serialplot.gui import protocol_editor, prefs, ui_state
from serialplot.gui.docks import SendDataDock

class MainWindow (ui_state.StateSaveableUI, QtGui.QMainWindow):
    inputProtocolChanged = QtCore.pyqtSignal(list)
    outputProtocolChanged = QtCore.pyqtSignal(list)
    openPort = QtCore.pyqtSignal()
    closePort = QtCore.pyqtSignal()

    # Used for config settings etc.
    companyName = "jjeg"
    productName = "serialplot"

    defaultUpdate = 30

    _wkspaceFileExt = ".workspace"
    _wkspaceFileFilter = "Workspace file (" + _wkspaceFileExt + ");All files (*)"

    def __init__(self, dataHandler, settings):
        super().__init__()

        self.dataHandler = dataHandler
        self.settings = settings

        self.defaultSubWindowSize=QtCore.QSize(640, 480)
        self.defaultSubWindowMinSize=QtCore.QSize(100, 100)

        self.resize(800, 600)

        self.mdiArea=QtGui.QMdiArea(self)
        self.setCentralWidget(self.mdiArea)

        menubar=QtGui.QMenuBar(self)
        self.setMenuBar(menubar)

        self.portStatus = QtGui.QLabel(self)
        self.setPortStatusClosed()
        self.statusBar().addWidget(self.portStatus)

        self.sendDataDock = SendDataDock(self.dataHandler, self)
        self.sendDataDock.setObjectName("Send Data dock")
        self.sendDataDock.dataSent.connect(self.dataHandler.sendData)
        self.addDockWidget(QtCore.Qt.BottomDockWidgetArea, self.sendDataDock)

        progMenu=QtGui.QMenu("&SerialPlot", menubar)
        menubar.addMenu(progMenu)

        progSettingsAction = QtGui.QAction("&Settings...", progMenu)
        progMenu.addAction(progSettingsAction)
        progSettingsAction.triggered.connect(self.editSettings)

        progOpenWsAction = QtGui.QAction("&Open Workspace...", progMenu)
        progMenu.addAction(progOpenWsAction)
        progOpenWsAction.triggered.connect(lambda: self.loadWorkspace(True))
```

```

progSaveWsAction = QtGui.QAction("&Save Workspace...", progMenu)
progMenu.addAction(progSaveWsAction)
progSaveWsAction.triggered.connect(lambda: self.saveWorkspace(True))

progQuitAction = QtGui.QAction("&Quit", progMenu)
progMenu.addAction(progQuitAction)
progQuitAction.triggered.connect(self.close)

portMenu=QtGui.QMenu("&Port", menubar)
menubar.addMenu(portMenu)

portOpenAction=QtGui.QAction("&Open", portMenu)
portMenu.addAction(portOpenAction)
portOpenAction.triggered.connect(self.openPort)

portCloseAction=QtGui.QAction("&Close", portMenu)
portMenu.addAction(portCloseAction)
portCloseAction.triggered.connect(self.closePort)

protMenu=QtGui.QMenu("P&rotocols", menubar)
menubar.addMenu(protMenu)

protEditInputAction = QtGui.QAction("Edit &Input Protocol...", protMenu)
protEditInputAction.triggered.connect(self.editInputProtocol)
protMenu.addAction(protEditInputAction)

protEditOutputAction = QtGui.QAction("Edit &Output Protocol...", protMenu)
protEditOutputAction.triggered.connect(self.editOutputProtocol)
protMenu.addAction(protEditOutputAction)

windowMenu=QtGui.QMenu("&Views", menubar)
menubar.addMenu(windowMenu)

# Dict of available subwindow types
# Each item = name: class
self.subWindowClasses = {}
# Add menu items for opening subwindow views
for mod_name in serialplot.gui.views.enabled_views:
    try:
        mod = import_module("." + mod_name, "serialplot.gui.views")
        action = QtGui.QAction("Open " + mod.menu_text, windowMenu)
    except ImportError as e:
        # TODO: notify user
        print("Could not import view module:", mod_name)
        print(e)
    except AttributeError as e:
        print("Invalid module format; please see the dev docs:", mod_name)
        print(e)
    else:
        self.subWindowClasses[repr(mod.ViewWidget)] = mod.ViewWidget
        action.triggered.connect(self._makeLambda(mod.ViewWidget))
        windowMenu.addAction(action)

windowMenu.addSeparator()

closeAllViewsAction = QtGui.QAction("&Close All Views", windowMenu)
closeAllViewsAction.triggered.connect(self.closeAllSubwindows)
windowMenu.addAction(closeAllViewsAction)

windowMenu.addSeparator()

showSendAction = self.sendDataDock.toggleViewAction()
showSendAction.setText("Show &Send Data toolbox")
windowMenu.addAction(showSendAction)

```

```

        self.addSettingsItem("geometry", QtCore.QByteArray,
                             self.saveGeometry, self.restoreGeometry)
        self.addSettingsItem("state", QtCore.QByteArray,
                             self.saveState, self.restoreState)
        self.addSettingsItem("inputProtocol", protocols.protocol,
                             self.dataHandler.inputProtocol, self.dataHandler.setInputProtocol)
        self.addSettingsItem("outputProtocol", protocols.protocol,
                             self.dataHandler.outputProtocol,
                             self.dataHandler.setOutputProtocol)

        # Restore UI settings
        self.loadWorkspace()

        self.sendDataDock.updateProtocolFields()

    def closeEvent(self, ev):
        """Save UI settings on window close."""

        self.saveWorkspace()

    def keyPressEvent(self, ev):
        key = ev.key()
        F1 = QtCore.Qt.Key_F1
        F12 = QtCore.Qt.Key_F12
        if key>=F1 and key<=F12:
            self.sendDataDock.fKeyPressed(key-F1)

    # Creates a lambda function to open a sub window with the correct widget
    def _makeLambda(self, obj):
        return lambda: self.openSubWindow(obj)

    def openSubWindow (self, widgetClass):
        widget = widgetClass(self, self.dataHandler, self.defaultUpdate)
        win = self.mdiArea.addSubWindow(widget)
        win.resize(self.defaultSubWindowSize)
        win.setMinimumSize(self.defaultSubWindowMinSize)
        win.show()
        return win

    def setPortStatusOpen(self):
        s = self.settings.getSettingsGroup("serial")
        self.portStatus.setText("Port open: {} {} {}{}{}, Flow control={}".format(
            s["port"],
            s["baud"],
            s["dataBits"],
            s["parity"],
            s["stopBits"],
            s["flowControl"]
        ))
    def setPortStatusClosed(self):
        self.portStatus.setText("Port closed")

    def editSettings (self):
        self.closePort.emit()
        dialog = prefs.PrefsEditor(self.settings)
        dialog.exec_()

    def editInputProtocol(self):
        pc = protocol_editor.protocolCreator()
        pc.setProtocol(self.dataHandler.inputProtocol())
        prot = pc.run()
        if prot is not None:
            self.dataHandler.setInputProtocol(prot)

```

```

        self.inputProtocolChanged.emit(prot)
        print("Changed input protocol")

    def editOutputProtocol(self):
        pc = protocol_editor.protocolCreator()
        pc.setProtocol(self.dataHandler.outputProtocol())
        prot = pc.run()
        if prot is not None:
            self.dataHandler.setOutputProtocol(prot)
            self.outputProtocolChanged.emit(prot)
            print("Changed output protocol")

        self.sendDataDock.updateProtocolFields()

    def restoreSubwindows(self, subwindows):
        for subwindow in subwindows:
            self.mdiArea.addSubWindow(subwindow)

    def closeAllSubwindows(self):
        self.mdiArea.closeAllSubWindows()

    def saveWorkspace(self, toFile=False):
        """Save the current workspace as the default or to a file."""

        if toFile:
            filename = QtGui.QFileDialog.getSaveFileName(
                parent = self, caption="Save workspace",

directory=QtGui.QDesktopServices.storageLocation(QtGui.QDesktopServices.HomeLocatio
n),
                filter=self._wkspaceFileFilter
            )
            if(not(filename.endswith(self._wkspaceFileExt))):
                filename += self._wkspacefileExt
            settings = QtCore.QSettings(filename, QtCore.QSettings.IniFormat)
        else:
            settings = QtCore.QSettings(self.companyName, self.productName)
        settings.clear()
        self.saveUi(settings)

        settings.beginGroup("prefs")
        self.settings.saveToQSettings(settings)
        settings.endGroup()

        settings.beginWriteArray("subwindows")
        subwindows = self.mdiArea.subWindowList(order=QtGui.QMdiArea.StackingOrder)
        if subwindows is not None:
            for i, subwindow in enumerate(subwindows):
                settings.setArrayIndex(i)
                settings.setValue("class", repr(type(subwindow.widget())))
                settings.setValue("geometry", subwindow.geometry())
                subwindow.widget().saveUi(settings)
        settings.endArray()

        settings.beginGroup("send_dock")
        self.sendDataDock.saveUi(settings)
        settings.endGroup()

    def loadWorkspace(self, fromFile=False):
        """Restore the workspace or load from file."""

        if fromFile:
            filename = QtGui.QFileDialog.getOpenFileName(
                parent = self, caption="Open workspace",

```

```

directory=QtGui.QDesktopServices.storageLocation(QtGui.QDesktopServices.HomeLocatio
n),
                filter=self._wkspaceFileFilter
            )
        settings = QtCore.QSettings(filename, QtCore.QSettings.IniFormat)
else:
    settings = QtCore.QSettings(self.companyName, self.productName)

self.closePort.emit()

self.closeAllSubwindows()

self.restoreUi(settings)

settings.beginGroup("prefs")
self.settings.loadFromQSettings(settings)
settings.endGroup()

numitems = settings.beginReadArray("subwindows")
if numitems:
    for i in range(numitems):
        settings.setArrayIndex(i)
        widget_class = settings.value("class", type=str)
        try:
            widget = self.subWindowClasses[widget_class](self,
self.dataHandler, self.defaultUpdate)
        except KeyError:
            print("View module not found:", widget_class)
        else:
            win = self.mdiArea.addSubWindow(widget)
            try:
                geo = settings.value("geometry", type=QtCore.QRect)
            except TypeError:
                pass
            else:
                if geo is not None:
                    win.setGeometry(geo)
            widget.restoreUi(settings)
            win.show()
# Open some default subwindows
else:
    for widget_class in self.subWindowClasses.values():
        widget = widget_class(self, self.dataHandler, self.defaultUpdate)
        win = self.mdiArea.addSubWindow(widget)
        win.resize(300, 300)
        win.show()

settings.endArray()

settings.beginGroup("send_dock")
self.sendDataDock.restoreUi(settings)
settings.endGroup()

```

## gui/prefs\_dialog.py

(This file was generated automatically from a Qt Designer project by PyQt UI code generator.)

```
from PyQt4 import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    _fromUtf8 = lambda s: s

class Ui_settingsDialog(object):
    def setupUi(self, settingsDialog):
        settingsDialog.setObjectName(_fromUtf8("settingsDialog"))
        settingsDialog.resize(345, 327)
        self.verticalLayout = QtGui.QVBoxLayout(settingsDialog)
        self.verticalLayout.setObjectName(_fromUtf8("verticalLayout"))
        self.prefsTabs = QtGui.QTabWidget(settingsDialog)
        self.prefsTabs.setObjectName(_fromUtf8("prefsTabs"))
        self.serialTab = QtGui.QWidget()
        self.serialTab.setObjectName(_fromUtf8("serialTab"))
        self.horizontalLayout = QtGui.QHBoxLayout(self.serialTab)
        self.horizontalLayout.setObjectName(_fromUtf8("horizontalLayout"))
        self.serialGroup = QtGui.QGroupBox(self.serialTab)
        self.serialGroup.setObjectName(_fromUtf8("serialGroup"))
        self.formLayout = QtGui.QFormLayout(self.serialGroup)
        self.formLayout.setObjectName(_fromUtf8("formLayout"))
        self.portLabel = QtGui.QLabel(self.serialGroup)
        self.portLabel.setObjectName(_fromUtf8("portLabel"))
        self.formLayout.setWidget(0, QtGui.QFormLayout.LabelRole, self.portLabel)
        self.baudLabel = QtGui.QLabel(self.serialGroup)
        self.baudLabel.setObjectName(_fromUtf8("baudLabel"))
        self.formLayout.setWidget(1, QtGui.QFormLayout.LabelRole, self.baudLabel)
        self.dataBitsLabel = QtGui.QLabel(self.serialGroup)
        self.dataBitsLabel.setObjectName(_fromUtf8("dataBitsLabel"))
        self.formLayout.setWidget(2, QtGui.QFormLayout.LabelRole,
                               self.dataBitsLabel)
        self.stopBitsLabel = QtGui.QLabel(self.serialGroup)
        self.stopBitsLabel.setObjectName(_fromUtf8("stopBitsLabel"))
        self.formLayout.setWidget(3, QtGui.QFormLayout.LabelRole,
                               self.stopBitsLabel)
        self.baudCombo = QtGui.QComboBox(self.serialGroup)
        self.baudCombo.setEditable(True)
        self.baudCombo.setObjectName(_fromUtf8("baudCombo"))
        self.baudCombo.addItem(_fromUtf8(""))
        self.baudCombo.addItem(_fromUtf8(""))
        self.baudCombo.addItem(_fromUtf8(""))
        self.baudCombo.addItem(_fromUtf8(""))
        self.baudCombo.addItem(_fromUtf8(""))
        self.baudCombo.addItem(_fromUtf8(""))
        self.baudCombo.addItem(_fromUtf8(""))
        self.baudCombo.addItem(_fromUtf8(""))
        self.formLayout.setWidget(1, QtGui.QFormLayout.FieldRole, self.baudCombo)
        self.parityLabel = QtGui.QLabel(self.serialGroup)
        self.parityLabel.setObjectName(_fromUtf8("parityLabel"))
        self.formLayout.setWidget(4, QtGui.QFormLayout.LabelRole, self.parityLabel)
        self.flowLabel = QtGui.QLabel(self.serialGroup)
        self.flowLabel.setObjectName(_fromUtf8("flowLabel"))
        self.formLayout.setWidget(5, QtGui.QFormLayout.LabelRole, self.flowLabel)
        self.dataBitsButtons = QtGui.QHBoxLayout()
        self.dataBitsButtons.setObjectName(_fromUtf8("dataBitsButtons"))
```

```

        self.dataBits5Button = QtGui.QRadioButton(self.serialGroup)
        self.dataBits5Button.setObjectName(_fromUtf8("dataBits5Button"))
        self.dataBitsGroup = QtGui.QButtonGroup(settingsDialog)
        self.dataBitsGroup.setObjectName(_fromUtf8("dataBitsGroup"))
        self.dataBitsGroup.addButton(self.dataBits5Button)
        self.dataBitsButtons.addWidget(self.dataBits5Button)
        self.dataBits6Button = QtGui.QRadioButton(self.serialGroup)
        self.dataBits6Button.setObjectName(_fromUtf8("dataBits6Button"))
        self.dataBitsGroup.addButton(self.dataBits6Button)
        self.dataBitsButtons.addWidget(self.dataBits6Button)
        self.dataBits7Button = QtGui.QRadioButton(self.serialGroup)
        self.dataBits7Button.setObjectName(_fromUtf8("dataBits7Button"))
        self.dataBitsGroup.addButton(self.dataBits7Button)
        self.dataBitsButtons.addWidget(self.dataBits7Button)
        self.dataBits8Button = QtGui.QRadioButton(self.serialGroup)
        self.dataBits8Button.setChecked(True)
        self.dataBits8Button.setObjectName(_fromUtf8("dataBits8Button"))
        self.dataBitsGroup.addButton(self.dataBits8Button)
        self.dataBitsButtons.addWidget(self.dataBits8Button)
        spacerItem = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.dataBitsButtons.addItem(spacerItem)
        self.formLayout.setLayout(2, QtGui.QFormLayout.FieldRole,
self.dataBitsButtons)
        self.stopBitsButtons = QtGui.QHBoxLayout()
        self.stopBitsButtons.setObjectName(_fromUtf8("stopBitsButtons"))
        self.stopBits1Button = QtGui.QRadioButton(self.serialGroup)
        self.stopBits1Button.setChecked(True)
        self.stopBits1Button.setObjectName(_fromUtf8("stopBits1Button"))
        self.stopBitsGroup = QtGui.QButtonGroup(settingsDialog)
        self.stopBitsGroup.setObjectName(_fromUtf8("stopBitsGroup"))
        self.stopBitsGroup.addButton(self.stopBits1Button)
        self.stopBitsButtons.addWidget(self.stopBits1Button)
        self.stopBits15Button = QtGui.QRadioButton(self.serialGroup)
        self.stopBits15Button.setObjectName(_fromUtf8("stopBits15Button"))
        self.stopBitsGroup.addButton(self.stopBits15Button)
        self.stopBitsButtons.addWidget(self.stopBits15Button)
        self.stopBits2Button = QtGui.QRadioButton(self.serialGroup)
        self.stopBits2Button.setObjectName(_fromUtf8("stopBits2Button"))
        self.stopBitsGroup.addButton(self.stopBits2Button)
        self.stopBitsButtons.addWidget(self.stopBits2Button)
        spacerItem1 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.stopBitsButtons.addItem(spacerItem1)
        self.formLayout.setLayout(3, QtGui.QFormLayout.FieldRole,
self.stopBitsButtons)
        self.parityCombo = QtGui.QComboBox(self.serialGroup)
        self.parityCombo.setObjectName(_fromUtf8("parityCombo"))
        self.parityCombo.addItem(_fromUtf8(""))
        self.parityCombo.addItem(_fromUtf8(""))
        self.parityCombo.addItem(_fromUtf8(""))
        self.parityCombo.addItem(_fromUtf8(""))
        self.parityCombo.addItem(_fromUtf8(""))
        self.parityCombo.addItem(_fromUtf8(""))
        self.formLayout.setWidget(4, QtGui.QFormLayout.FieldRole, self.parityCombo)
        self.flowCombo = QtGui.QComboBox(self.serialGroup)
        self.flowCombo.setObjectName(_fromUtf8("flowCombo"))
        self.flowCombo.addItem(_fromUtf8(""))
        self.flowCombo.addItem(_fromUtf8(""))
        self.flowCombo.addItem(_fromUtf8(""))
        self.flowCombo.addItem(_fromUtf8(""))
        self.formLayout.setWidget(5, QtGui.QFormLayout.FieldRole, self.flowCombo)
        self.portEdit = QtGui.QComboBox(self.serialGroup)
        self.portEdit.setEditable(True)
        self.portEdit.setObjectName(_fromUtf8("portEdit"))
        self.formLayout.setWidget(0, QtGui.QFormLayout.FieldRole, self.portEdit)

```

```

        self.horizontalLayout.addWidget(self.serialGroup)
        self.prefsTabs.addTab(self.serialTab, _fromUtf8(""))
        self.graphTab = QtGui.QWidget()
        self.graphTab.setObjectName(_fromUtf8("graphTab"))
        self.formLayout_2 = QtGui.QFormLayout(self.graphTab)

        self.formLayout_2.setFieldGrowthPolicy(QtGui.QFormLayout.AllNonFixedFieldsGrow)
        self.formLayout_2.setObjectName(_fromUtf8("formLayout_2"))
        self.graphUpdateRateLabel = QtGui.QLabel(self.graphTab)
        self.graphUpdateRateLabel.setObjectName(_fromUtf8("graphUpdateRateLabel"))
        self.formLayout_2.setWidget(0, QtGui.QFormLayout.LabelRole,
self.graphUpdateRateLabel)
        self.horizontalLayout_2 = QtGui.QHBoxLayout()
        self.horizontalLayout_2.setSpacing(3)
        self.horizontalLayout_2.setObjectName(_fromUtf8("horizontalLayout_2"))
        self.graphUpdateRateEdit = QtGui.QLineEdit(self.graphTab)
        self.graphUpdateRateEdit.setObjectName(_fromUtf8("graphUpdateRateEdit"))
        self.horizontalLayout_2.addWidget(self.graphUpdateRateEdit)
        self.msLabel = QtGui.QLabel(self.graphTab)
        self.msLabel.setObjectName(_fromUtf8("msLabel"))
        self.horizontalLayout_2.addWidget(self.msLabel)
        self.formLayout_2.setLayout(0, QtGui.QFormLayout.FieldRole,
self.horizontalLayout_2)
        self.plainTextEdit = QtGui.QPlainTextEdit(self.graphTab)
        self.plainTextEdit.setReadOnly(True)
        self.plainTextEdit.setObjectName(_fromUtf8("plainTextEdit"))
        self.formLayout_2.setWidget(1, QtGui.QFormLayout.FieldRole,
self.plainTextEdit)
        self.prefsTabs.addTab(self.graphTab, _fromUtf8(""))
        self.verticalLayout.addWidget(self.prefsTabs)
        self.mainLayout = QtGui.QVBoxLayout()
        self.mainLayout.setObjectName(_fromUtf8("mainLayout"))
        spacerItem2 = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        self.mainLayout.addItem(spacerItem2)
        self.okButton = QtGui.QPushButton(settingsDialog)
        self.okButton.setObjectName(_fromUtf8("okButton"))
        self.mainLayout.addWidget(self.okButton)
        self.cancelButton = QtGui.QPushButton(settingsDialog)
        self.cancelButton.setObjectName(_fromUtf8("cancelButton"))
        self.mainLayout.addWidget(self.cancelButton)
        self.verticalLayout.addWidget(self.mainLayout)

        self.retranslateUi(settingsDialog)
        self.prefsTabs.setCurrentIndex(0)
        self.baudCombo.setCurrentIndex(2)
        QtCore.QObject.connect(self.okButton,
QtCore.SIGNAL(_fromUtf8("clicked()")), settingsDialog.accept)
        QtCore.QObject.connect(self.cancelButton,
QtCore.SIGNAL(_fromUtf8("clicked()")), settingsDialog.reject)
        QtCore.QMetaObject.connectSlotsByName(settingsDialog)

    def retranslateUi(self, settingsDialog):

        settingsDialog.setWindowTitle(QtGui.QApplication.translate("settingsDialog",
"Settings", None, QtGui.QApplication.UnicodeUTF8))
        self.serialGroup.setTitle(QtGui.QApplication.translate("settingsDialog",
"Serial Settings", None, QtGui.QApplication.UnicodeUTF8))
        self.portLabel.setText(QtGui.QApplication.translate("settingsDialog", "COM
port", None, QtGui.QApplication.UnicodeUTF8))
        self.baudLabel.setText(QtGui.QApplication.translate("settingsDialog", "Baud
Rate", None, QtGui.QApplication.UnicodeUTF8))
        self.dataBitsLabel.setText(QtGui.QApplication.translate("settingsDialog",
>Data Bits", None, QtGui.QApplication.UnicodeUTF8))

```

```

        self.stopBitsLabel.setText(QtGui.QApplication.translate("settingsDialog",
"Stop Bits", None, QtGui.QApplication.UnicodeUTF8))
        self.baudCombo.setItemText(0,
QtGui.QApplication.translate("settingsDialog", "2400", None,
QtGui.QApplication.UnicodeUTF8))
        self.baudCombo.setItemText(1,
QtGui.QApplication.translate("settingsDialog", "4800", None,
QtGui.QApplication.UnicodeUTF8))
        self.baudCombo.setItemText(2,
QtGui.QApplication.translate("settingsDialog", "9600", None,
QtGui.QApplication.UnicodeUTF8))
        self.baudCombo.setItemText(3,
QtGui.QApplication.translate("settingsDialog", "19200", None,
QtGui.QApplication.UnicodeUTF8))
        self.baudCombo.setItemText(4,
QtGui.QApplication.translate("settingsDialog", "38400", None,
QtGui.QApplication.UnicodeUTF8))
        self.baudCombo.setItemText(5,
QtGui.QApplication.translate("settingsDialog", "57600", None,
QtGui.QApplication.UnicodeUTF8))
        self.baudCombo.setItemText(6,
QtGui.QApplication.translate("settingsDialog", "115200", None,
QtGui.QApplication.UnicodeUTF8))
        self.parityLabel.setText(QtGui.QApplication.translate("settingsDialog",
"Parity", None, QtGui.QApplication.UnicodeUTF8))
        self.flowLabel.setText(QtGui.QApplication.translate("settingsDialog", "Flow
Control", None, QtGui.QApplication.UnicodeUTF8))
        self.dataBits5Button.setText(QtGui.QApplication.translate("settingsDialog",
"5", None, QtGui.QApplication.UnicodeUTF8))
        self.dataBits6Button.setText(QtGui.QApplication.translate("settingsDialog",
"6", None, QtGui.QApplication.UnicodeUTF8))
        self.dataBits7Button.setText(QtGui.QApplication.translate("settingsDialog",
"7", None, QtGui.QApplication.UnicodeUTF8))
        self.dataBits8Button.setText(QtGui.QApplication.translate("settingsDialog",
"8", None, QtGui.QApplication.UnicodeUTF8))
        self.stopBits1Button.setText(QtGui.QApplication.translate("settingsDialog",
"1", None, QtGui.QApplication.UnicodeUTF8))

self.stopBits15Button.setText(QtGui.QApplication.translate("settingsDialog", "1.5",
None, QtGui.QApplication.UnicodeUTF8))
        self.stopBits2Button.setText(QtGui.QApplication.translate("settingsDialog",
"2", None, QtGui.QApplication.UnicodeUTF8))
        self.parityCombo.setItemText(0,
QtGui.QApplication.translate("settingsDialog", "None", None,
QtGui.QApplication.UnicodeUTF8))
        self.parityCombo.setItemText(1,
QtGui.QApplication.translate("settingsDialog", "Even", None,
QtGui.QApplication.UnicodeUTF8))
        self.parityCombo.setItemText(2,
QtGui.QApplication.translate("settingsDialog", "Odd", None,
QtGui.QApplication.UnicodeUTF8))
        self.parityCombo.setItemText(3,
QtGui.QApplication.translate("settingsDialog", "Mark", None,
QtGui.QApplication.UnicodeUTF8))
        self.parityCombo.setItemText(4,
QtGui.QApplication.translate("settingsDialog", "Space", None,
QtGui.QApplication.UnicodeUTF8))
        self.flowCombo.setItemText(0,
QtGui.QApplication.translate("settingsDialog", "None", None,
QtGui.QApplication.UnicodeUTF8))
        self.flowCombo.setItemText(1,
QtGui.QApplication.translate("settingsDialog", "XON/XOFF", None,
QtGui.QApplication.UnicodeUTF8))

```

```

        self.flowCombo.setItemText(2,
QtGui.QApplication.translate("settingsDialog", "RTS/CTS", None,
QtGui.QApplication.UnicodeUTF8))
        self.flowCombo.setItemText(3,
QtGui.QApplication.translate("settingsDialog", "DSR/DTR", None,
QtGui.QApplication.UnicodeUTF8))
        self.prefsTabs.setTabText(self.prefsTabs.indexOf(self.serialTab),
QtGui.QApplication.translate("settingsDialog", "Serial", None,
QtGui.QApplication.UnicodeUTF8))

self.graphUpdateRateLabel.setText(QtGui.QApplication.translate("settingsDialog",
"Update Rate", None, QtGui.QApplication.UnicodeUTF8))

self.graphUpdateRateEdit.setText(QtGui.QApplication.translate("settingsDialog",
"30", None, QtGui.QApplication.UnicodeUTF8))
        self.msLabel.setText(QtGui.QApplication.translate("settingsDialog", "ms",
None, QtGui.QApplication.UnicodeUTF8))

self.plainTextEdit.setPlainText(QtGui.QApplication.translate("settingsDialog", "NB:
This tab is currently unused and therefore disabled in serialplot.gui.prefs.",
None, QtGui.QApplication.UnicodeUTF8))
        self.prefsTabs.setTabText(self.prefsTabs.indexOf(self.graphTab),
QtGui.QApplication.translate("settingsDialog", "Graph", None,
QtGui.QApplication.UnicodeUTF8))
        self.okButton.setText(QtGui.QApplication.translate("settingsDialog", "OK",
None, QtGui.QApplication.UnicodeUTF8))
        self.cancelButton.setText(QtGui.QApplication.translate("settingsDialog",
"Cancel", None, QtGui.QApplication.UnicodeUTF8))

```

## gui/prefs.py

```
#!/usr/bin/python3

import re

import serial
from PyQt4 import QtCore, QtGui

from serialplot.gui import prefs_dialog
from serialplot.util.list_ports import comports as get_ports

class PrefsEditor(QtGui.QDialog):
    """Graphical editor for editing program settings."""

    def __init__(self, settings):
        """Initialise editor.

        Keyword arguments:
        settings -- a serialplot.settings.Settings object which
                    settings will be read from and written to.
        """

        super().__init__()

        self.settings = settings

        # Pull in Qt Designer generated code
        self.ui = prefs_dialog.Ui_settingsDialog()
        self.ui.setupUi(self)

        # Remove graph tab as not currently used
        self.ui.graphTab.setParent(None)

        # Finds all serial ports and adds to the "Port" combo box
        # TODO: Provide ability to find which ports are actually available
        self.ui.portEdit.addItems(sorted((port[0] for port in get_ports())),
key=_getPortSortKey)

        self.accepted.connect(self.applyPrefs)
        self.loadPrefs()

    def applyPrefs(self):
        # Dictionaries to map options => serial vars etc
        # TODO: Store the mappings in the combobox items's UserData
        dataBitMap = {
            "5": serial.FIVEBITS,
            "6": serial.SIXBITS,
            "7": serial.SEVENBITS,
            "8": serial.EIGHTBITS
        }

        stopBitMap = {
            "1": serial.STOPBITS_ONE,
            "1.5": serial.STOPBITS_ONE_POINT_FIVE,
            "2": serial.STOPBITS_TWO
        }

        parityMap = {
            "None": serial.PARITY_NONE,
            "Even": serial.PARITY_EVEN,
            "Odd": serial.PARITY_ODD,
            "Mark": serial.PARITY_ODD,
            "Space": serial.PARITY_SPACE
        }
```

```

        }

    serialsettings = dict(
        port = self.ui.portEdit.currentText(),
        baud = self.ui.baudCombo.currentText(),
        dataBits =
    dataBitMap[self.ui.dataBitsGroup.checkedButton().text()],
        stopBits =
    stopBitMap[self.ui.stopBitsGroup.checkedButton().text()],
        parity = parityMap[self.ui.parityCombo.currentText()],
        flowControl = self.ui.flowCombo.currentText(),
    )
    self.settings.updateSettingsGroup('serial', serialsettings)

def loadPrefs(self):
    # TODO: Store the mappings in the combobox items's UserData
    dataBitMap = {
        serial.FIVEBITS: self.ui.dataBits5Button,
        serial.SIXBITS: self.ui.dataBits6Button,
        serial.SEVENBITS: self.ui.dataBits7Button,
        serial.EIGHTBITS: self.ui.dataBits8Button
    }

    stopBitMap = {
        serial.STOPBITS_ONE: self.ui.stopBits1Button,
        serial.STOPBITS_ONE_POINT_FIVE: self.ui.stopBits15Button,
        serial.STOPBITS_TWO: self.ui.stopBits2Button
    }

    parityMap = {
        serial.PARITY_NONE: self.ui.parityCombo.findText("None"),
        serial.PARITY_EVEN: self.ui.parityCombo.findText("Even"),
        serial.PARITY_ODD: self.ui.parityCombo.findText("Odd"),
        serial.PARITY_ODD: self.ui.parityCombo.findText("Mark"),
        serial.PARITY_SPACE: self.ui.parityCombo.findText("Space")
    }

    settings = self.settings.getSettingsGroup('serial')

    if settings['port'] is None:
        self.ui.portEdit.setCurrentIndex(0)
    else:
        self.ui.portEdit.setEditText(settings['port'])
    self.ui.baudCombo.setEditText(str(settings['baud']))
    dataBitMap[settings['dataBits']].setChecked(True)
    stopBitMap[settings['stopBits']].setChecked(True)
    self.ui.parityCombo.setCurrentIndex(parityMap[settings['parity']])
    flowindex = self.ui.flowCombo.findText(settings['flowControl'])
    self.ui.flowCombo.setCurrentIndex(flowindex)

def _getPortSortKey(port):
    """Returns a sort key intended to be used in sorted()'s "key" argument.
    This use will result in ports being sorted numerically.

    Keyword arguments:
        port -- The name of the port to obtain a sort key for.
    """

    m=re.search(r"\d+$", port)
    if(m):
        key=int(m.group(0))
    else:
        key=0

    return key

```

```
if (__name__ == "__main__"):
    import sys
    from pprint import pprint
    from serialplot.settings import Settings
    app = QtGui.QApplication(sys.argv)
    settings = Settings()
    dialog = PrefsEditor(settings)
    dialog.exec_()
    pprint(vars(settings))
```

## gui/protocol\_dialog.py

(This file was generated automatically from a Qt Designer project by PyQt UI code generator.)

```
from PyQt4 import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    _fromUtf8 = lambda s: s

class Ui_ProtocolEdit(object):
    def setupUi(self, ProtocolEdit):
        ProtocolEdit.setObjectName(_fromUtf8("ProtocolEdit"))
        ProtocolEdit.resize(631, 460)
        self.verticalLayout = QtGui.QVBoxLayout(ProtocolEdit)
        self.verticalLayout.setObjectName(_fromUtf8("verticalLayout"))
        self.groupBox = QtGui.QGroupBox(ProtocolEdit)
        self.groupBox.setObjectName(_fromUtf8("groupBox"))
        self.horizontalLayout_4 = QtGui.QHBoxLayout(self.groupBox)
        self.horizontalLayout_4.setMargin(0)
        self.horizontalLayout_4.setObjectName(_fromUtf8("horizontalLayout_4"))
        self.viewScrollArea = QtGui.QScrollArea(self.groupBox)
        sizePolicy = QtGui.QSizePolicy(QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
        sizePolicy.setHorizontalStretch(0)
        sizePolicy.setVerticalStretch(0)

        sizePolicy.setHeightForWidth(self.viewScrollArea.sizePolicy().hasHeightForWidth())
        self.viewScrollArea.setSizePolicy(sizePolicy)
        self.viewScrollArea setFrameShape(QtGui.QFrame.NoFrame)

        self.viewScrollArea.setVerticalScrollBarPolicy(QtCore.Qt.ScrollBarAlwaysOff)

        self.viewScrollArea.setHorizontalScrollBarPolicy(QtCore.Qt.ScrollBarAlwaysOn)
        self.viewScrollArea.setWidgetResizable(True)
        self.viewScrollArea.setObjectName(_fromUtf8("viewScrollArea"))
        self.viewScrollAreaContents = QtGui.QWidget()
        self.viewScrollAreaContents.setGeometry(QtCore.QRect(0, 0, 605, 63))

        self.viewScrollAreaContents.setObjectName(_fromUtf8("viewScrollAreaContents"))
        self.viewScrollArea.setWidget(self.viewScrollAreaContents)
        self.horizontalLayout_4.addWidget(self.viewScrollArea)
        self.verticalLayout.addWidget(self.groupBox)
        self.groupBox_2 = QtGui.QGroupBox(ProtocolEdit)
        self.groupBox_2.setObjectName(_fromUtf8("groupBox_2"))
        self.horizontalLayout_3 = QtGui.QHBoxLayout(self.groupBox_2)
        self.horizontalLayout_3.setMargin(0)
        self.horizontalLayout_3.setObjectName(_fromUtf8("horizontalLayout_3"))
        self.editScrollArea = QtGui.QScrollArea(self.groupBox_2)
        self.editScrollArea setFrameShape(QtGui.QFrame.NoFrame)
        self.editScrollArea.setVerticalScrollBarPolicy(QtCore.Qt.ScrollBarAsNeeded)

        self.editScrollArea.setHorizontalScrollBarPolicy(QtCore.Qt.ScrollBarAlwaysOn)
        self.editScrollArea.setWidgetResizable(True)
        self.editScrollArea.setObjectName(_fromUtf8("editScrollArea"))
        self.editScrollAreaContents = QtGui.QWidget()
        self.editScrollAreaContents.setGeometry(QtCore.QRect(0, 0, 605, 206))

        self.editScrollAreaContents.setObjectName(_fromUtf8("editScrollAreaContents"))
```

```

    self.editScrollArea.setWidget(self.editScrollAreaContents)
    self.horizontalLayout_3.addWidget(self.editScrollArea)
    self.verticalLayout.addWidget(self.groupBox_2)
    self.horizontalLayout_2 = QtGui.QHBoxLayout()
    self.horizontalLayout_2.setObjectName(_fromUtf8("horizontalLayout_2"))
    self.validationLabel = QtGui.QLabel(ProtocolEdit)
    self.validationLabel.setObjectName(_fromUtf8("validationLabel"))
    self.horizontalLayout_2.addWidget(self.validationLabel)
    self.valLEdit = QtGui.QLineEdit(ProtocolEdit)
    self.valLEdit.setObjectName(_fromUtf8("valLEdit"))
    self.horizontalLayout_2.addWidget(self.valLEdit)
    self.label = QtGui.QLabel(ProtocolEdit)
    self.label.setObjectName(_fromUtf8("label"))
    self.horizontalLayout_2.addWidget(self.label)
    self.valREdit = QtGui.QLineEdit(ProtocolEdit)
    self.valREdit.setObjectName(_fromUtf8("valREdit"))
    self.horizontalLayout_2.addWidget(self.valREdit)
    self.verticalLayout.addLayout(self.horizontalLayout_2)
    self.horizontalLayout = QtGui.QHBoxLayout()
    self.horizontalLayout.setObjectName(_fromUtf8("horizontalLayout"))
    self.nameLabel = QtGui.QLabel(ProtocolEdit)
    self.nameLabel.setObjectName(_fromUtf8("nameLabel"))
    self.horizontalLayout.addWidget(self.nameLabel)
    self.nameEdit = QtGui.QLineEdit(ProtocolEdit)
    self.nameEdit.setObjectName(_fromUtf8("nameEdit"))
    self.horizontalLayout.addWidget(self.nameEdit)
    self.line = QtGui.QFrame(ProtocolEdit)
    self.line.setFrameShape(QtGui.QFrame.VLine)
    self.line.setFrameShadow(QtGui.QFrame.Sunken)
    self.line.setObjectName(_fromUtf8("line"))
    self.horizontalLayout.addWidget(self.line)
    self.segmentsLabel = QtGui.QLabel(ProtocolEdit)
    self.segmentsLabel.setObjectName(_fromUtf8("segmentsLabel"))
    self.horizontalLayout.addWidget(self.segmentsLabel)
    self.segmentsSpinBox = QtGui.QSpinBox(ProtocolEdit)
    self.segmentsSpinBox.setKeyboardTracking(False)
    self.segmentsSpinBox.setMaximum(128)
    self.segmentsSpinBox.setObjectName(_fromUtf8("segmentsSpinBox"))
    self.horizontalLayout.addWidget(self.segmentsSpinBox)
    self.verticalLayout.addLayout(self.horizontalLayout)
    self.horizontalLayout_5 = QtGui.QHBoxLayout()
    self.horizontalLayout_5.setObjectName(_fromUtf8("horizontalLayout_5"))
    self.saveAsButton = QtGui.QPushButton(ProtocolEdit)
    self.saveAsButton.setObjectName(_fromUtf8("saveAsButton"))
    self.horizontalLayout_5.addWidget(self.saveAsButton)
    self.openButton = QtGui.QPushButton(ProtocolEdit)
    self.openButton.setObjectName(_fromUtf8("openButton"))
    self.horizontalLayout_5.addWidget(self.openButton)
    spacerItem = QtGui.QSpacerItem(40, 20, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
    self.horizontalLayout_5.addItem(spacerItem)
    self.okButton = QtGui.QPushButton(ProtocolEdit)
    self.okButton.setAutoDefault(False)
    self.okButton.setObjectName(_fromUtf8("okButton"))
    self.horizontalLayout_5.addWidget(self.okButton)
    self.cancelButton = QtGui.QPushButton(ProtocolEdit)
    self.cancelButton.setAutoDefault(False)
    self.cancelButton.setObjectName(_fromUtf8("cancelButton"))
    self.horizontalLayout_5.addWidget(self.cancelButton)
    self.revertButton = QtGui.QPushButton(ProtocolEdit)
    self.revertButton.setAutoDefault(False)
    self.revertButton.setObjectName(_fromUtf8("revertButton"))
    self.horizontalLayout_5.addWidget(self.revertButton)
    self.verticalLayout.addLayout(self.horizontalLayout_5)

```

```

        self.retranslateUi(ProtocolEdit)
        QtCore.QObject.connect(self.okButton,
QtCore.SIGNAL(_fromUtf8("clicked()")), ProtocolEdit.accept)
        QtCore.QObject.connect(self.cancelButton,
QtCore.SIGNAL(_fromUtf8("clicked()")), ProtocolEdit.reject)
        QtCore.QMetaObject.connectSlotsByName(ProtocolEdit)

    def retranslateUi(self, ProtocolEdit):
        ProtocolEdit.setWindowTitle(QtGui.QApplication.translate("ProtocolEdit",
"Protocol Editor", None, QtGui.QApplication.UnicodeUTF8))
        self.groupBox.setTitle(QtGui.QApplication.translate("ProtocolEdit",
"Preview", None, QtGui.QApplication.UnicodeUTF8))
        self.groupBox_2.setTitle(QtGui.QApplication.translate("ProtocolEdit",
>Edit", None, QtGui.QApplication.UnicodeUTF8))
        self.validationLabel.setText(QtGui.QApplication.translate("ProtocolEdit",
"Validation expression:", None, QtGui.QApplication.UnicodeUTF8))
        self.label.setText(QtGui.QApplication.translate("ProtocolEdit", "=", None,
QtGui.QApplication.UnicodeUTF8))
        self.nameLabel.setText(QtGui.QApplication.translate("ProtocolEdit",
"Protocol name:", None, QtGui.QApplication.UnicodeUTF8))
        self.segmentsLabel.setText(QtGui.QApplication.translate("ProtocolEdit",
"Fields:", None, QtGui.QApplication.UnicodeUTF8))
        self.saveAsButton.setText(QtGui.QApplication.translate("ProtocolEdit",
"Save As...", None, QtGui.QApplication.UnicodeUTF8))
        self.openButton.setText(QtGui.QApplication.translate("ProtocolEdit",
"Open...", None, QtGui.QApplication.UnicodeUTF8))
        self.okButton.setText(QtGui.QApplication.translate("ProtocolEdit", "OK",
None, QtGui.QApplication.UnicodeUTF8))
        self.cancelButton.setText(QtGui.QApplication.translate("ProtocolEdit",
"Cancel", None, QtGui.QApplication.UnicodeUTF8))
        self.revertButton.setText(QtGui.QApplication.translate("ProtocolEdit",
"Revert", None, QtGui.QApplication.UnicodeUTF8))

```

## gui/protocol\_editor.py

```
from PyQt4 import QtCore, QtGui

from serialplot.gui import protocol_dialog
from serialplot.gui import protocol_widgets
from serialplot import protocols
from serialplot.protocols import bitTypeOne, bitTypeZero, bitTypeReq

# Interface to creator dialog
# Allows setting the protocol, running the dialog and getting the protocol
class protocolCreator(QtCore.QObject):
    # Signal emitted when protocol changed
    protocolChanged = QtCore.pyqtSignal()

    _protFileExt = ".prot"
    _fileFilter = "Protocol file (" + _protFileExt + ");All files (*)"

    def __init__(self):
        super().__init__()

        # List of segments
        self.segments=list()

        # Original protocol to allow reverting
        self.originalProtocol=None

        # Set up dialog
        self.protocolUi=protocol_dialog.Ui_ProtocolEdit()
        self.dialog=QtGui.QDialog()
        self.protocolUi.setupUi(self.dialog)

        # Create a layout in the scroll area containing the segment layout and a
        spacer
        segmentLayoutContainer=QtGui.QHBoxLayout()
        self.segmentLayout=QtGui.QHBoxLayout()
        segmentLayoutContainer.addLayout(self.segmentLayout)
        segmentLayoutContainer.addItem(QtGui.QSpacerItem(0, 0,
QtGui.QSizePolicy.Expanding, QtGui.QSizePolicy.Minimum))
        self.protocolUi.editScrollAreaContents.setLayout(segmentLayoutContainer)

        # Create a layout for the preview area
        previewLayoutContainer=QtGui.QHBoxLayout()
        self.previewLayout=QtGui.QHBoxLayout()
        self.previewLayout.setSpacing(0)
        previewLayoutContainer.addLayout(self.previewLayout)
        previewLayoutContainer.addItem(QtGui.QSpacerItem(0, 0,
QtGui.QSizePolicy.Expanding, QtGui.QSizePolicy.Minimum))
        self.protocolUi.viewScrollAreaContents.setLayout(previewLayoutContainer)

        # Connect signals
        self.protocolUi.segmentsSpinBox.valueChanged.connect(self._setNumSegments)
        self.protocolUi.revertButton.clicked.connect(self.revert)
        self.protocolUi.saveAsButton.clicked.connect(self.saveProtocol)
        self.protocolUi.openButton.clicked.connect(self.loadProtocol)
        self.protocolChanged.connect(self.updateViewArea)

    def updateViewArea(self):
        # Clear current view area
        for _ in range(self.previewLayout.count()):
            w=self.previewLayout.itemAt(0).widget()
            self.previewLayout.removeWidget(w)
            w.setParent(None)
```

```

# Generate all bits based on raw protocol
for bit in self.protocol().rawProtocol():
    w=protocol_widgets.protocolBitWidget(bit, self.dialog, True)
    if(bit==protocols.bitTypeOne): w.setBitType(bitTypeOne)
    self.previewLayout.addWidget(w)

# Sets the protocol
# This is only designed to be ran before the dialog is executed
# as it will lose any changes made
def setProtocol(self, prot):
    self.clearAllSegments()

    # Loop over fields
    for field in prot:
        # Create new segment using the type of the first bit
        seg=self._newSegment(field[0], field.name)
        seg.setNumBits(len(field))

        # Loop over protocol bits and toggle if '1'
        for bit, i in zip(field, range(len(field))):
            if(bit==bitTypeOne):
                seg.bits[i].toggleBit()

    # Set name
    if(prot.name): self.protocolUi.nameEdit.setText(prot.name)

    # Set validation fields
    if(prot.valL): self.protocolUi.valLEdit.setText(str(prot.valL))
    if(prot.valR): self.protocolUi.valREdit.setText(str(prot.valR))

    # Set original protocol (for reset, cancel etc)
    self.originalProtocol=prot

# Gets the protocol
def protocol(self):
    # Create a new protocol object
    p=protocols.protocol()

    # Loop round the protocol segment widgets
    for segw in self.segments:
        # Create new field and set name
        field=protocols.protocolField()
        field.name=segw.nameEdit.text()

        # Loop round the bits in the segment
        for bit in segw.bits:
            # Append bit to protocol
            field.append(bit.bitType)

        # Append field to protocol
        p.append(field)

    # Set protocol name
    name=self.protocolUi.nameEdit.text()
    if(not(name)):
        name='(unnamed protocol)'
    p.name=name

    # Get validation fields
    valL=self.protocolUi.valLEdit.text()
    if(valL==""): valL=None
    valR=self.protocolUi.valREdit.text()
    if(valR==""): valR=None
    p.setValidator(valL, valR)

```

```

        # Return the protocol
        return p

    def clearAllSegments(self):
        self.setNumSegments(0)

        # Increments the number of segments by 1, sets the given type
        # and returns the last segment widget
        # TODO: Allow changing type by type rather than index
        # Shouldn't be called directly - call setNumSegments()
    def _newSegment(self, t=None, name ""):
        self.setNumSegments(self.numSegments() + 1)
        seg = self.segments[-1]

        if(t == bitTypeZero or t == bitTypeOne):
            seg.changeType(0)
        elif(t == bitTypeReq):
            seg.changeType(1)

        seg.nameEdit.setText(name)

        return seg

    def setNumSegments(self, n):
        self.protocolUi.segmentsSpinBox.setValue(n)

        # Should only be called from the spin box
    def _setNumSegments(self, n):
        m = len(self.segments)
        if (n < m):
            # Delete the last (m-n) bits
            for _ in range(m-n):
                # Remove from layout
                self.segmentLayout.removeWidget(self.segments[-1])
                # Set parent to None
                self.segments[-1].setParent(None)
                # Deletes the item from the list
                del self.segments[-1]
        elif (n > m):
            # Add (n-m) bits
            for _ in range(n-m):
                # Create widget and add to layout
                w = protocol_widgets.protocolSegmentWidget(self.dialog)
                self.segments.append(w)
                self.segmentLayout.addWidget(w)

                # The segment has changed, so the protocol has
                # so join the signals
                w.segmentChanged.connect(self.protocolChanged)

        # Emit signal as protocol has changed
        self.protocolChanged.emit()

    def numSegments(self):
        return len(self.segments)

    def run(self):
        # Execute dialog and check result
        if(self.dialog.exec_() == QtGui.QDialog.Accepted):
            return self.protocol()
        else:
            return None

```

```

# Method for revert button
def revert(self):
    if(self.originalProtocol is not None):
        self.setProtocol(self.originalProtocol)
    # Most likely if we are adding a new protocol
    else:
        # Delete all segments
        self.clearAllSegments()
        # Clear the name box
        self.protocolUi.nameEdit.clear()

def saveProtocol(self):
    (filename, filter_) = QtGui.QFileDialog.getSaveFileNameAndFilter(
        parent=self.dialog, caption="Save to file",
        directory=QtGui.QDesktopServices.storageLocation(QtGui.QDesktopServices.HomeLocatio
n),
        filter=self._fileFilter
    )
    if (not(filename.endswith(self._protFileExt)) and
        filter_.endswith(self._protFileExt, 0, -1)):
        filename += self._protFileExt

    settings = QtCore.QSettings(filename, QtCore.QSettings.IniFormat)
    settings.setValue("protocol", self.protocol())

def loadProtocol(self):
    filename = QtGui.QFileDialog.getOpenFileName(
        parent = self.dialog, caption="Load from file",
        directory=QtGui.QDesktopServices.storageLocation(QtGui.QDesktopServices.HomeLocatio
n),
        filter=self._fileFilter
    )
    settings = QtCore.QSettings(filename, QtCore.QSettings.IniFormat)
    self.setProtocol(settings.value("protocol"))

# Run a sample window for testing
def _test():
    import sys
    app=QtGui.QApplication(sys.argv)

    pc=protocolCreator()
    p_out=pc.run()
    print("Protocol out:", p_out)

if (__name__=="__main__"):
    _test()

```

## gui/protocol\_widgets.py

```
from PyQt4 import QtCore, QtGui

# Import bit type definitions
from serialplot.protocols import bitTypeZero, bitTypeOne, bitTypeReq

_bitWidgetWidth=15
_bitWidgetHeight=20

# Combo box set up for bit type
class bitTypeComboBox(QtGui.QComboBox):
    def __init__(self, parent=None):
        super().__init__(parent)

        self.itemTypes=dict()
        for s, t in zip((self.tr("Fixed"), self.tr("Data")),
                         (bitTypeZero, bitTypeReq)):
            # Add item string and associated bit type to the dict
            self.itemTypes[s]=t
            # Add item to the combo box
            self.addItem(s)

    # Current bit type
    def currentBitType(self):
        return self.itemTypes[self.currentText()]

class protocolSegmentWidget(QtGui.QFrame):
    # Signal for when segment is changed
    segmentChanged=QtCore.pyqtSignal()

    def __init__(self, parent=None):
        super().__init__(parent)

        self.setFrameShape(QtGui.QFrame.Box)
        self.setFrameShadow(QtGui.QFrame.Sunken)

        # Empty list to hold the bits
        self.bits=list()

        # Make widget fixed size
        self.setSizePolicy(QtGui.QSizePolicy.Fixed, QtGui.QSizePolicy.Fixed)

        # Set a grid layout
        mainLayout=QtGui.QGridLayout()
        self.setLayout(mainLayout)

        # Row counter to aid adding additional widgets
        # Should be incremented after a new row added
        row=0

        # First row contains labels for MSB and LSB, and a spacer to keep them at
the end
        # These are put into a separate layout to stop them interfering with
        endianLayout=QtGui.QHBoxLayout()
        lsbLabel=QtGui.QLabel("LSB", self)
        msbLabel=QtGui.QLabel("MSB", self)
        endianLayout.addWidget(msbLabel)
        endianLayout.addItem(QtGui.QSpacerItem(0, 0, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum))
        endianLayout.addWidget(lsbLabel)

        # Add a vertical layout in top row spanning all cols containing
```

```

# another vertical layout and a spacer
# Inner layout will contain bits
bitsLayoutContainer=QtGui.QGridLayout()
bitsLayoutContainer.setLayout(endianLayout, 0, 0)
self.bitsLayout=QtGui.QHBoxLayout()
self.bitsLayout.setSpacing(0)
bitsLayoutContainer.setLayout(self.bitsLayout, 1, 0)
bitsLayoutContainer.addItem(QtGui.QSpacerItem(0, 0,
QtGui.QSizePolicy.Expanding, QtGui.QSizePolicy.Minimum),
                           0, 1, 2, 1)
mainLayout.addLayout(bitsLayoutContainer, row, 0, 1, 3)
row+=1

# Button to reverse order of fixed bits
self.reverseButton=QtGui.QPushButton(self)
self.reverseButton.setText("< Reverse >")
self.reverseButton.clicked.connect(self.reverseBits)
mainLayout.addWidget(self.reverseButton, row, 0, 1, 2)
row+=1

# Add a combo for segment type row 2
self.typeCombo=bitTypeComboBox(self)
mainLayout.addWidget(self.typeCombo, row, 0, 1, 2)
row+=1

# Add label and spin box for num of bits
numBitsLabel=QtGui.QLabel(self.tr("Bits:"), self)
mainLayout.addWidget(numBitsLabel, row, 0)

self.numBitsSpinBox=QtGui.QSpinBox(self)
self.numBitsSpinBox.setMinimum(1)
self.numBitsSpinBox.setMaximum(128)
self.numBitsSpinBox.setValue(1)
self.numBitsSpinBox.setKeyboardTracking(False)
mainLayout.addWidget(self.numBitsSpinBox, row, 1)
row+=1

# Add name field
self.nameEdit=QtGui.QLineEdit(self)
self.nameEdit.setPlaceholderText(self.tr("Field name"))
# Let width match parent widget width (size hint ignored)
self.nameEdit.setSizePolicy(QtGui.QSizePolicy.Ignored,
QtGui.QSizePolicy.Fixed)
mainLayout.addWidget(self.nameEdit, row, 0, 1, 2)
row+=1

self._setNumBits(1)

# Add a spacer to keep settings tidy
settingsSpacer=QtGui.QSpacerItem(0, 0, QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Minimum)
mainLayout.addItem(settingsSpacer, 1, 2, 2, 1)

# Connect signals to functions,
self.numBitsSpinBox.valueChanged.connect(self._setNumBits)
self.typeCombo.currentIndexChanged.connect(self._changeType)

# Ensure elements are set up correctly for default type
self._changeType()

# TODO: Allow reverse lookup, i.e. specifying the type and the combo box is set
def changeType(self, i):
    self.typeCombo.setCurrentIndex(i)

# Combo box changed - change the type of all the bits

```

```

def _changeType(self):
    t=self.typeCombo.currentBitType()
    for b in self.bits:
        b.setBitType(t)
    # Disable/enable elements only relevant to particular bit types
    if(t==bitTypeZero or t==bitTypeOne):
        self.reverseButton.setEnabled(True)
    else:
        self.reverseButton.setEnabled(False)

    # Emit changed signal
    self.segmentChanged.emit()

def segType(self):
    return self.typeCombo.currentBitType()

def setNumBits(self, n):
    self.numBitsSpinBox.setValue(n)

def _setNumBits(self, n):
    m=len(self.bits)
    if (n<m):
        # Delete the last (m-n) bits
        for _i in range(m-n):
            # Remove from layout
            self.bitsLayout.removeWidget(self.bits[-1])
            # Set parent to None
            self.bits[-1].setParent(None)
            # Deletes the item from the list
            del self.bits[-1]
    elif (n>m):
        t=self.typeCombo.currentBitType()
        # Add (n-m) bits
        for _ in range(n-m):
            w=protocolBitWidget(t, self)
            # When the bit has been changed, the segment is considered to have
been changed
            w.bitChanged.connect(self.segmentChanged)
            self.bits.append(w)
            self.bitsLayout.addWidget(w)

        # Emit changed signals
        self.segmentChanged.emit()

    # Reverse all bits in a field
    # Creates a list of current bit types, reverses it, then toggles a bit if not
now correct
    def reverseBits(self):
        for bit, newType in zip(self.bits, reversed(list(b.bitType for b in
self.bits))):
            if(bit.bitType!=newType):
                bit.toggleBit()

    def numBits(self):
        return len(self.bits)

class protocolBitWidget(QtGui.QWidget):
    bitChanged=QtCore.pyqtSignal()

    # ro = don't allow toggling of bit
    def __init__(self, t=bitTypeZero, parent=None, ro=False):
        super().__init__(parent)
        self.setFixedSize(_bitWidgetWidth, _bitWidgetHeight)
        self.bitType=None
        self.setBitType(t)

```

```

        self.ro=ro

    def setBitType(self, t):
        if (self.bitType!=t):
            self.bitType=t
            self.update()

    def toggleBit(self):
        if (self.bitType==bitTypeZero):
            self.setBitType(bitTypeOne)
        elif (self.bitType==bitTypeOne):
            self.setBitType(bitTypeZero)

        self.bitChanged.emit()

    # For bit toggling
    def mousePressEvent(self, event):
        if (not(self.ro) and event.button()==QtCore.Qt.LeftButton):
            self.toggleBit()

    def paintEvent(self, event):
        t=self.bitType
        # Set variables depending on bit type
        if (t==bitTypeZero):
            ch="0"
            col="black"
            txtcol="white"
        elif (t==bitTypeOne):
            ch="1"
            col="white"
            txtcol="black"
        elif (t==bitTypeReq):
            ch ="?"
            col="royalblue"
            txtcol="white"

        # Create painter
        p=QtGui.QPainter(self)

        # Fill with specified colour
        p.setBrush(QtGui.QBrush(QtGui.QColor(col)))
        p.setPen(QtCore.Qt.NoPen)
        p.drawRect(0, 0, _bitWidgetWidth, _bitWidgetHeight)

        # Draw character in centre
        p.setBrush(QtGui.QBrush(QtCore.Qt.NoBrush))
        p.setPen(QtGui.QColor(txtcol))
        t=QtGui.QStaticText(ch)
        size=t.size().toSize()
        x=(_bitWidgetWidth/2)-(size.width()/2)
        y=(_bitWidgetHeight/2)-(size.height()/2)
        p.drawStaticText(x, y, t)

    # For testing; launches a dialog with several of the widgets
    import sys
    if (__name__=="__main__"):
        app = QtGui.QApplication(sys.argv)
        dialog=QtGui.QDialog()
        layout=QtGui.QHBoxLayout(dialog)
        for _i in range(4):
            widget=protocolSegmentWidget(dialog)
            layout.addWidget(widget)
        dialog.exec_()

```

## gui/resources.py

(File consisted of primarily large binary data generated by PyQt Resource Compiler, for image resources e.g. a splash screen. Not included for brevity.)

## gui/ui\_state.py

```
from PyQt4 import QtCore

class StateSaveableUI(object):
    """Provide methods for saving and restoring the UI state.

    """

    def __init__(self):
        super().__init__()

        self.uiSettingsItems = []

    def addSettingsItem(self, *args):
        """Add an item to save/restore.
        NB: Objects which inherit QtGui cannot be saved.

        Arguments:
            1 -- String to be used as the key.
            2 -- Object type to be saved/restored.
            3 -- Function to be called to retrieve the object to save.
                Will be called with no arguments.
            4 -- Function to be called to apply the setting.
                Will be passed an object with type 'type_'.
        """

        self.uiSettingsItems.append(args)

    def saveUi(self, settings):
        """Save the current state of the UI.

        Keyword arguments:
            settings -- QSettings object to save the settings to.
        """

        for (key, _, func, _) in self.uiSettingsItems:
            settings.setValue(key, func())

    def restoreUi(self, settings):
        """Retore the UI state from a settings object.

        Keyword arguments:
            settings -- QSettings object to save the settings to.
        """

        for (key, type_, _, func) in self.uiSettingsItems:
            try:
                obj = settings.value(key, type=type_, defaultValue=None)
            except TypeError:
                print("TypeError raised while loading UI setting:", key)
                pass
            else:
                if obj is not None:
                    func(obj)
```

## gui/util.py

```
from PyQt4 import QtCore, QtGui

def showError(text):
    msgbox = QtGui.QMessageBox(QtGui.QMessageBox.Critical, "Error", text)
    msgbox.exec_()
```

## gui/views/\_\_init\_\_.py

```
#!/usr/bin/python3

# List of enabled view modules
# Add names of modules here to appear in the View menu
enabled_views = [
    # Uncomment next line to enable example view
    #'example_view',
    'current_value',
    'console',
    'graph',
    'table',
    'file_capture'
]
```

## gui/views/console.py

```
#!/usr/bin/python3

from fractions import Fraction
from math import ceil

from PyQt4 import QtCore, QtGui

from .subwindow import SubWindowViewWidget

menu_text = "&Console View"

class ViewWidget (SubWindowViewWidget):
    def setup(self):
        self.setWindowTitle("Console")

        self._numCols = 1
        self._curCol = 0

        # Presentation type of data
        self._presType = "d"

        # Create layout for console view at top and row of buttons under
        main_layout = QtGui.QVBoxLayout(self)
        self.setLayout(main_layout)

        # Create text area
        self._consoleOutput = con_output = QtGui.QPlainTextEdit(self)
        con_output.setReadOnly(True)
        con_output.setMaximumBlockCount(1000)
        font=QtGui.QFont("Courier New")
        con_output.setFont(font)
        main_layout.addWidget(con_output)

        # Layout for controls
        controls_layout = QtGui.QHBoxLayout()
        main_layout.addLayout(controls_layout)

        controls_layout.addWidget(self.fieldDropdown)

        # Drop down list for selection of formatting
        self._formatDropdown = format_dropdown = QtGui.QComboBox(self)
        format_dropdown.setEditable(False)
        controls_layout.addWidget(format_dropdown)

        # Add items to drop down
        # Presentation type character passed as user data
        for item in (
            ("ASCII", "c"),
            ("Hex", "X"),
            ("Decimal", "d"),
            ("Octal", "o"),
            ("Binary", "b")
        ):
            format_dropdown.addItem(*item)

        # Set dropdown to current (default) type
        format_dropdown.setCurrentIndex(format_dropdown.findData(self._presType))
        format_dropdown.currentIndexChanged.connect(self._dataFormatChanged)

        controls_layout.addStretch()

    def updateCallback(self):
```

```

        self.printNewData()

    def _dataFormatChanged(self, itemid):
        pt = self._formatDropdown.itemData(itemid)
        self.setPresType(pt)

    def setPresType(self, t):
        self._presType = t
        self._reset()

    def _reset(self):
        self.stopUpdate()
        self._consoleOutput.clear()
        super().__reset__()

    pt = self._presType
    flen = self.getFieldLen()

    # Default to no padding, and a space between each value
    w = ""
    sep = " "
    # ASCII representation => no space between characters
    if pt == "c":
        sep = ""
    # Other formats => use a fixed width based on field length
    elif pt == "b":
        w = flen
    else:
        w = len('{{:' + pt + '}}').format((1 << flen) - 1))

    s = '{0:'
    if w: s += '0'
    s += str(w) + pt + '}'

    self._presFormat = s, sep

    self.printNewData()
    self.resumeUpdate()

def printNewData(self):
    data = self.getNewData()

    if data is not None:
        self.printData(data)

def printData(self, data):
    if data is None:
        return

    # Alternative implementation which doesn't create a new line each time
    # This, however, means the number of "blocks" may never exceed 1, so the
    # view will continue to grow indefinitely.

    #self._consoleOutput.moveCursor(QtGui.QTextCursor.End)
    #self._consoleOutput.insertPlainText(sep.join(s.format(datum) for datum in
data) + sep)
    #sb = self._consoleOutput.verticalScrollBar()
    #sb.setValue(sb.maximum())

    fmt, sep = self._presFormat
    self._consoleOutput.appendPlainText(sep.join(fmt.format(datum) for datum in
data))

```

## gui/views/current\_value.py

```
#!/usr/bin/python3

from PyQt4 import QtCore, QtGui

from .subwindow import SubWindowViewWidget

menu_text = "&Current Value View"

class ViewWidget (SubWindowViewWidget):
    def setup(self):
        self.setWindowTitle("Current Value View")
        layout = QtGui.QVBoxLayout()
        self.valueLabel = StretchedLabel("Waiting for data...")
        layout.addWidget(self.valueLabel)
        layout.addWidget(self.fieldDropdown)
        self.setLayout(layout)

    def updateCallback(self):
        # Get most recent data point (returns an iterator or None)
        data = self.getData(1)

        try:
            # Just get the actual data item
            data = next(data)
        # There was no data item
        except StopIteration:
            pass
        else:
            self.valueLabel.scaleText = True
            self.valueLabel.setText(str(data))

    class StretchedLabel(QtGui.QLabel):
        def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)

            self.setSizePolicy(QtGui.QSizePolicy.Ignored, QtGui.QSizePolicy.Ignored)

            font = self.font()
            self.defaultPixelSize = font.pixelSize()

            self.scaleText=False

        def rescaleText(self):
            font = self.font()
            if self.scaleText:
                h = self.height()
                ps = h * 0.8
            else:
                ps = self.defaultPixelSize

            # Check if valid; this happens if the object has not finished init'ing
            if ps > 0:
                font.setPixelSize(ps)
            self.setFont(font)

        def setText(self, *args, **kwargs):
            super().setText(*args, **kwargs)
            self.rescaleText()

        def resizeEvent(self, evt):
            self.rescaleText()
```

## gui/views/example\_view.py

```
#!/usr/bin/python3

from PyQt4 import QtCore, QtGui

from .subwindow import SubWindowViewWidget

menu_text = "&Current Value View"

class ViewWidget (SubWindowViewWidget):
    def setup(self):
        self.setWindowTitle("Current Value View")
        layout = QtGui.QVBoxLayout()
        self.valueLabel = StretchedLabel("Waiting for data...")
        layout.addWidget(self.valueLabel)
        layout.addWidget(self.fieldDropdown)
        self.setLayout(layout)

    def updateCallback(self):
        # Get most recent data point (returns an iterator or None)
        data = self.getData(1)
        try:
            # Just get the actual data item
            data = next(data)
        # There was no data item
        except StopIteration:
            pass
        else:
            self.valueLabel.scaleText = True
            self.valueLabel.setText(str(data))

    class StretchedLabel(QtGui.QLabel):
        def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)

            self.setSizePolicy(QtGui.QSizePolicy.Ignored, QtGui.QSizePolicy.Ignored)

            font = self.font()
            self.defaultPixelSize = font.pixelSize()

            self.scaleText=False

        def rescaleText(self):
            font = self.font()
            if self.scaleText:
                h = self.height()
                ps = h * 0.8
            else:
                ps = self.defaultPixelSize

            # Check if valid; this happens if the object has not finished init'ing
            if ps > 0:
                font.setPixelSize(ps)

            self.setFont(font)

        def setText(self, *args, **kwargs):
            super().setText(*args, **kwargs)
            self.rescaleText()

        def resizeEvent(self, evt):
            self.rescaleText()
```

## gui/views/file\_capture.py

```
#!/usr/bin/python3

from PyQt4 import QtCore, QtGui

from .subwindow import SubWindowViewWidget
from serialplot.file_capture import capturer

menu_text = "&File Capture"

class ViewWidget (SubWindowViewWidget):
    appendWarning = "Caution: ensure current protocol fields match the file's
fields!"
    overwriteWarning = "Caution: will overwrite any existing file!"

    def setup(self):
        self.setWindowTitle("File Capture")

        self.fieldDropdown.hide()

        self.cap = capturer()

        mainlayout = QtGui.QVBoxLayout()

        filelayout = QtGui.QHBoxLayout()
        filelabel = QtGui.QLabel("File:", self)
        filelayout.addWidget(filelabel)
        self.fileName = QtGui.QLineEdit(self)
        filelayout.addWidget(self.fileName)
        self.browseButton = QtGui.QPushButton("&Browse...", self)
        filelayout.addWidget(self.browseButton)
        mainlayout.addLayout(filelayout)

        controllayout = QtGui.QHBoxLayout()
        appendLabel = QtGui.QLabel("Append to file?", self)
        controllayout.addWidget(appendLabel)
        self.appendCheck = QtGui.QCheckBox(self)
        self.appendCheck.setChecked(True)
        self.appendCheck.setToolTip("If checked, will append to an existing file
rather than overwriting.")
        controllayout.addWidget(self.appendCheck)
        self.startButton = QtGui.QPushButton("&Start", self)
        controllayout.addWidget(self.startButton)
        self.progressBar = QtGui.QProgressBar(self)
        self.progressBar.setMinimum(0)
        self.progressBar.setMaximum(0)
        self.progressBar.hide()
        controllayout.addWidget(self.progressBar)
        # Add a small widget to use up the space of the hidden progress bar
        self.controlStretch = QtGui.QWidget(self)
        self.controlStretch.setSizePolicy(QtGui.QSizePolicy.MinimumExpanding,
QtGui.QSizePolicy.Fixed)
        controllayout.addWidget(self.controlStretch)
        mainlayout.addLayout(controllayout)

        self.statusText = QtGui.QLabel(self.appendWarning, self)
        font = self.statusText.font()
        font.setWeight(QtGui.QFont.Bold)
        self.statusText.setFont(font)
        mainlayout.addWidget(self.statusText)

        mainlayout.addStretch()
        self.setLayout(mainlayout)
```

```

# Connect stuff
self.cap.started.connect(self.captureStarted)
self.cap.stopped.connect(self.captureStopped)
self.cap.fileOpenError.connect(self.fileOpenError)

self.browseButton.clicked.connect(self.selectFile)
self.appendCheck.toggled.connect(self.appendChangedState)
self.startButton.clicked.connect(self.startStopCapture)

def updateCallback(self):
    data = self.getNewData(-1)
    if data is not None:
        for item in data:
            self.cap.write_data(item)

def selectFile(self):
    ext = ".csv"
    filtertext = "CSV files (*"+ext+")"
    (fname, ffilter) = QtGui.QFileDialog.getSaveFileNameAndFilter(
        self, "Save data to...", "", filtertext+";;All Files (*)")
    if(ffilter == filtertext and not(fname.endswith(ext))):
        fname += ext
    self.fileName.setText(fname)

def appendChangedState(self, checked):
    if checked:
        self.statusText.setText(self.appendWarning)
    else:
        self.statusText.setText(self.overwriteWarning)

def startStopCapture(self):
    if self.cap.running():
        self.cap.stop_capture()
    else:
        headings = self.dataHandler.inputProtocol().dataFieldNames()
        self.cap.start_capture(self.fileName.text(), headings,
                               self.appendCheck.isChecked())

def captureStarted(self):
    self.setWidgetsEnabled(False)
    self.startButton.setText("&Stop")
    self.progressBar.show()
    self.controlStretch.hide()

def captureStopped(self):
    self.setWidgetsEnabled(True)
    self.startButton.setText("&Start")
    self.progressBar.hide()
    self.controlStretch.show()

def fileOpenError(self, s):
    QtGui.QMessageBox.critical(self, "Error", "An error occurred whilst opening
the file:\n"+s)

# True/False to enable/disable widgets when capture in progress
def setWidgetsEnabled(self, enabled):
    self.fileName.setEnabled(enabled)
    self.browseButton.setEnabled(enabled)
    self.appendCheck.setEnabled(enabled)

```

## gui/views/graph.py

```
#!/usr/bin/python3

from itertools import cycle

from PyQt4 import QtCore, QtGui

from .subwindow_multifield import SubWindowViewWidget

menu_text = "&Graph View"

class ViewWidget (SubWindowViewWidget):
    colours = (
        (0, 0, 0),      # Black
        (255, 0, 0),    # Red
        (100, 100, 255),# Blue
        (0, 180, 0),    # Green
        (255, 140, 0),  # Orange
        (255, 0, 255),  # Magenta
    )

    def setup(self):
        super().setup()

        self.setWindowTitle("Graph")

        self.setFocusPolicy(QtCore.Qt.StrongFocus)

        # Allows user to browse back
        self._browseOffset=0

        self.setMinimumHeight(50)
        self.setMinimumWidth(50)

        main_layout = QtGui.QVBoxLayout(self)
        graph_layout = QtGui.QHBoxLayout()
        yaxis_layout = QtGui.QVBoxLayout()
        self.maxValLabel = QtGui.QLabel(" ", self)
        yaxis_layout.addWidget(self.maxValLabel)
        yaxis_layout.addStretch()
        yaxis_layout.addWidget(QtGui.QLabel("0", self))
        graph_layout.addLayout(yaxis_layout)

        self.plotWidget = PlotWidget(self)
        graph_layout.addWidget(self.plotWidget)

        main_layout.addLayout(graph_layout)

        # For the colour key
        self.key_layout = QtGui.QHBoxLayout()
        main_layout.addLayout(self.key_layout)

        controls_layout = QtGui.QHBoxLayout()
        controls_layout.addWidget(self.fieldsButton)

        controls_layout.addStretch()
        main_layout.addLayout(controls_layout)

        self.updateFields()

        # Update autoscale when selected fields changed
```

```

        self.selectedFieldsUpdated.connect(self.updateAutoScale)

# Callback from "update" timer
def updateCallback(self):
    self._plotPendingDataPoints()

def updateFields(self):
    super().updateFields()

    # Delete items from current key
    while self.key_layout.count():
        i = self.key_layout.takeAt(0)
        try:
            i.widget().deleteLater()
        except AttributeError:
            pass

    self.key_layout.addWidget(QtGui.QLabel("Key:"))

    # Loop round all fields, assigning colour and adding to key
    for i, colour in zip(range(self.fields.rowCount()), cycle(self.colours)):
        item = self.fields.item(i)
        c = QtGui.QColor(*colour)

        # Assign colour as the "User Data" in the FieldItem
        item.setData(c)

        # Add to key
        l = QtGui.QLabel(item.text(), parent=self)
        p = l.palette()
        p.setColor(l.foregroundRole(), c)
        l.setPalette(p)
        self.key_layout.addWidget(l)

    self.key_layout.addStretch()

    self.updateAutoScale()

def updateAutoScale(self):
    prot = self.dataHandler.inputProtocol()
    maxval = 0

    for field in self.getSelectedFields():
        f_len = len(prot[field.fieldid])
        size = (2**f_len)-1 # Max value for a given number of bits

        if(size>maxval):
            maxval = size

    self.maxValLabel.setText(str(maxval))
    self.plotWidget.setMaxVal(maxval)

def _plotPendingDataPoints(self):
    self._reset()

    self.plotWidget.plot(self.getAllData())

def getAllData(self):
    numpoints=self.plotWidget.numPoints() # Number of points which can be plotted

    for field in self.getSelectedFields():
        points=self.dataHandler.getData (start=numpoints+self._browseOffset,
                                         end=self._browseOffset, field=field.fieldid)
        if (points is not None):

```

```

        # field.data() returns the colour as assigned in updateFields()
        yield DataSet(points, field.data())

# TODO: tidy up next 4 functions to avoid repetition
def browseBack(self):
    w = self.plotWidget.width()
    self._browseOffset += w // 4
    qlen = self.dataHandler.queueLength()
    if (self._browseOffset > qlen - w):
        self._browseOffset = qlen - w
    self._plotPendingDataPoints()

def browseForwards(self):
    w = self.plotWidget.width()
    self._browseOffset -= w // 4
    if (self._browseOffset < 0): self._browseOffset = 0
    self._plotPendingDataPoints()

def browseStart(self):
    w = self.plotWidget.width()
    self._browseOffset = self.dataHandler.queueLength() - w

def browseEnd(self):
    self._browseOffset = 0

# TODO: implement into widgets
def keyPressEvent(self, ev):
    k=ev.key()
    if(k==QtCore.Qt.Key_Left):
        self.browseBack()
    elif(k==QtCore.Qt.Key_Right):
        self.browseForwards()
    elif(k==QtCore.Qt.Key_Home):
        self.browseStart()
    elif(k==QtCore.Qt.Key_End):
        self.browseEnd()
    else:
        super().keyPressEvent(ev)

# On resize, reset graph size
def resizeEvent (self, event):
    self.plotWidget.resetGraphSize()

class PlotWidget(QtGui.QWidget):
    def __init__(self, parent):
        super().__init__(parent)

        self.setSizePolicy(QtGui.QSizePolicy.Expanding,
QtGui.QSizePolicy.Expanding)

        # Default maximal value to plot
        self.maxVal = 255

        self._penWidth = 1
        self._pointSpacing = 0

        self.resetGraphSize()
        self._reset()

    # Reimplement size hint
    def sizeHint(self):
        return QtCore.QSize(50, 50)

    def _reset(self):
        # Overwrite pixmap

```

```

        self.graphPixmap = QtGui.QPixmap(self.width(), self.height())
        self.yVal = None

        p = QtGui.QPainter(self.graphPixmap)
        p.setBrush(QtGui.QBrush(QtGui.QColor("white")))
        p.setPen(QtCore.Qt.NoPen)
        p.drawRect(0, 0, self.graphPixmap.width(), self.graphPixmap.height())

        self.update()

    def resetGraphSize(self):
        self._reset()

    # Get number of points shown on graph
    def numPoints(self):
        return self.graphPixmap.width()*self._penWidth

    def plot(self, datasets):
        self._reset()

        p=QtGui.QPainter(self.graphPixmap)

        # Turn on antialiasing
        p.setRenderHint(QtGui.QPainter.Antialiasing)

        for dataset in datasets:
            # Set up pen for drawing lines
            pen = QtGui.QPen(dataset.colour)
            pen.setWidth(self._penWidth)
            pen.setCapStyle(QtCore.Qt.FlatCap)
            pen.setJoinStyle(QtCore.Qt.RoundJoin)
            p.setPen(pen)
            y1 = None
            for i, point in enumerate(dataset.data):
                y2 = self.height() - (self.scalePoint(point))

                # Only plot if y1 has been set, else this is the first point
                if y1 is not None:
                    p.drawLine(i, y1, i+self._penWidth, y2)

                y1=y2

            self.update()

    def setMaxVal(self, val):
        self.maxVal=val

    # Scales the given value to the graph height
    def scalePoint(self, val):
        return (val*(self.height()-2)/self.maxVal+1)

    def copyGraph(self):
        p = QtGui.QPainter(self)
        p.drawPixmap(0, 0, self.graphPixmap)

    # Copy pixmap to widget when requested
    def paintEvent (self, event):
        self.copyGraph()
        pass

    class DataSet(object):
        def __init__(self, data, colour):
            self.data = data
            self.colour = colour

```

## gui/views/subwindow\_multifield.py

```
#!/usr/bin/python3

from PyQt4 import QtCore, QtGui

from . import subwindow

class SubWindowViewWidget (subwindow.SubWindowViewWidget):
    # Emitted when user (or otherwise) updates selected fields
    selectedFieldsUpdated = QtCore.pyqtSignal()

    def setup(self):
        super().setup()

        # We don't use this...
        self.fieldDropdown.hide()

        # Create button to select fields
        self.fieldsButton = QtGui.QPushButton("Fields...", self)
        self.fieldsButton.setCheckable(False)
        self.fieldsButton.clicked.connect(self.selectFields)

        # NB: We are actually storing a list, but "int" is passed as the type
        # as we specify the type of the container's items
        # See http://pyqt.sourceforge.net/Docs/PyQt4/pyqt_qsettings.html
        self.addSettingsItem("fields", int, self.getSelectedFieldIds,
                             self.setSelectedFieldIds)

    # Override field update method
    def reloadFieldDropdown(self):
        self.updateFields()

    def selectFields(self):
        # Field selection dialog
        d = FieldSelectDialog(self)
        d.exec_()
        self.selectedFieldsUpdated.emit()

    # TODO: reset data counter so all current data is received
    #self.resetDataCounter()

    def updateFields(self):
        dh = self.dataHandler
        protocol = dh.inputProtocol()

        # Fields model
        self.fields = QtGui.QStandardItemModel(self)

        for fieldid, field in enumerate(protocol):
            if field.isDataField():
                item = FieldItem(field.name)
                item.setCheckable(True)
                item.setCheckState(QtCore.Qt.Checked)
                item.fieldid = fieldid
                self.fields.appendRow(item)

        self.selectedFieldsUpdated.emit()

    def getSelectedFields(self):
        for i in range(self.fields.rowCount()):
            item = self.fields.item(i)
            if item.checkState() == QtCore.Qt.Checked:
                yield item
```

```

def getSelectedFieldIds(self):
    ids = list()
    for field in self.getSelectedFields():
        ids.append(field.fieldid)

    return ids

def setSelectedFieldIds(self, ids):
    # Loop round each item and check the ID is in the list provided
    for i in range(self.fields.rowCount()):
        item = self.fields.item(i)
        if item.fieldid in ids:
            item.setCheckState(QtCore.Qt.Checked)
        else:
            item.setCheckState(QtCore.Qt.Unchecked)

    self.selectedFieldsUpdated.emit()

class FieldSelectDialog(QtGui.QDialog):
    """ Allows a user to select visible fields. """

    def __init__(self, parent):
        super().__init__(parent)

        self.setWindowTitle("Field Selection")

        main_layout = QtGui.QVBoxLayout(self)

        self.list = QtGui.QListView(self)
        self.list.setModel(parent.fields)
        main_layout.addWidget(self.list)

        buttons = QtGui.QDialogButtonBox.Ok
        button_box = QtGui.QDialogButtonBox(buttons, parent=self)
        main_layout.addWidget(button_box)

        button_box.accepted.connect(self.accept)
        button_box.rejected.connect(self.reject)

    class FieldItem(QtGui.QStandardItem):
        def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)

            # Used to store the field ID
            self.fieldid = None

```

## gui/views/subwindow.py

```
#!/usr/bin/python3

from PyQt4 import QtCore, QtGui

from serialplot.gui import ui_state

# Provides a base class for the sub window views internal widgets
class SubWindowViewWidget(ui_state.StateSaveableUI, QtGui.QWidget):
    def __init__(self, parent, dataHandler, updateInterval=30):
        super().__init__()

        # Can't do this as part of the __init__ as we inherit 2 classes
        self.setParent(parent)

        self.dataHandler = dataHandler

        self.fieldDropdown = FieldDropdown(self)

        self.window().inputProtocolChanged.connect(self.reloadFieldDropdown)
        self.fieldDropdown.currentIndexChanged.connect(self._reset)

        # Timer to update the view
        self._updateTimer=QtCore.QTimer()
        self._updateTimer.timeout.connect(self.updateCallback)
        self.setUpdateInterval(updateInterval)
        self._updateTimer.start()

        self._latestDataPointer = None

        self.addSettingsItem("field", int, self._currentFieldInt,
self._setCurrentFieldInt)

        self.setup()

        self.reloadFieldDropdown()

    # Function for submodule setup code
    def setup(self):
        pass

    # Callback to update widget to latest data
    # Note this is not the same thing as Qt's GUI update()
    def updateCallback(self):
        pass

    def closeEvent(self, ev):
        self._updateTimer.stop()

    # Callback to reset the view
    def _reset(self):
        self._latestDataPointer = None

    def _currentFieldInt(self):
        """Return the current field, but -1 instead of None as
        the raw data field.
        """

        f = self._currentField()
        if f is None:
            f = -1
        return f
```

```

def _setCurrentFieldInt(self, f):
    """Set the current field, but interprets -1 as the
    raw data field.
    """

    if f == -1:
        f = None
    self._setCurrentField(f)

def _currentField(self):
    index = self.fieldDropdown.currentIndex()
    f = self.fieldDropdown.itemData(int(index))
    return f

def _setCurrentField(self, fieldId):
    index = self.fieldDropdown.findData(fieldId)
    self.fieldDropdown.setCurrentIndex(index)

def getData(self, *args):
    return self.dataHandler.getData(*args, field=self._currentField())

def getNewData(self, field=None, limit=None):
    if field is None:
        field=self._currentField()
    tagged_data = self.dataHandler.getNewData(self._latestDataPointer,
field=field, limit=limit)
    self._latestDataPointer=tagged_data.pointer
    return tagged_data.data

def getFieldLen(self, field=None):
    if field is None:
        field=self._currentField()

    # TODO: check byte width setting
    if field is None:    # Raw data
        return 8

    return len(self.dataHandler.inputProtocol()[field])

def resetDataCounter(self):
    self._latestDataPointer=None

def updateInterval(self):
    return self._updateInterval

def setUpdateInterval(self, interval):
    self._updateInterval=interval
    self._updateTimer.setInterval(interval)

def reloadFieldDropdown(self):
    fields = self.dataHandler.inputProtocol()
    fieldnames = map(lambda f: f.name, fields)

    self.fieldDropdown.reload(zip(
                                fieldnames,
                                range(len(fields)), # Create field "ID"
                                ))
    # Stop/pause view update
def stopUpdate(self):
    self._updateTimer.stop()

def resumeUpdate(self):
    self._updateTimer.start()

```

```
# Creates a dropdown list of fields
class FieldDropdown(QtGui.QComboBox):
    def __init__(self, parent):
        super().__init__(parent)

        self.setEditable(False)

    # Fields should be in a tuple of tuple name-id pairs
    def reload(self, fields=[]):
        # Clear current items
        self.clear()

        self.addItem("(Raw data)", None)

        for field in fields:
            self.addItem(*field)
```

## gui/views/table.py

```
#!/usr/bin/python3

from PyQt4 import QtCore, QtGui

from .subwindow_multifield import SubWindowViewWidget

menu_text = "&Table View"

class ViewWidget(SubWindowViewWidget):
    numrows = 1000

    def setup(self):
        super().setup();

        self.setWindowTitle("Table")

        main_layout = QtGui.QVBoxLayout(self)

        self.table = QtGui.QTableWidget(self)
        self.table.verticalHeader().setVisible(False)
        main_layout.addWidget(self.table)

        controls_layout = QtGui.QHBoxLayout()
        controls_layout.addWidget(self.fieldsButton)
        controls_layout.addStretch()
        main_layout.addLayout(controls_layout)

        # Used to create a map between the table columns and data field IDs
        # So you can do self._colmap[2] to get the col num for field 2
        self._colmap = None

        self.updateFields()

        # Reset the table view when selected fields changed
        self.selectedFieldsUpdated.connect(self.reset)

    def reset(self):
        # Reset data pointer
        self._reset()

        # Clear the table
        self.table.clear()
        self.table.setRowCount(0)
        self.table.setColumnCount(0)

        # Reset column map
        self._colmap = {}

        for col, field in enumerate(self.getSelectedFields()):
            self.table.insertColumn(col)
            header = QtGui.QTableWidgetItem(field.text())
            self.table.setHorizontalHeaderItem(col, header)

            self._colmap[field.fieldid] = col

    def updateCallback(self):
        data = self.getNewData(-1, self.numrows)

        if data is not None:
            self.addData(data)

    def addData(self, data):
```

```

# Iterate through each data item
for item in data:
    while(self.table.rowCount()>=self.numrows):
        self.table.removeRow(0)

    row = self.table.rowCount()
    self.table.insertRow(row)

    # Iterate through each piece of data (i.e. from each field) in the data
item
    for piece in item:
        try:
            col = self._colmap[piece.field]
        except KeyError:
            # Ignore if not found => column not shown
            pass
        else:
            text = str(piece)
            cell = QtGui.QTableWidgetItem(text)
            self.table.setItem(row, col, cell)

# Scroll to bottom if was at bottom before adding
scrollbar = self.table.verticalScrollBar()
if scrollbar.value() == scrollbar.maximum():
    self.table.repaint()      # Effectively forces a redraw, adding the new
rows
    scrollbar.setValue(scrollbar.maximum())

```

## hw/\_\_init\_\_.py

(Empty file)

## hw/manager.py

```
from PyQt4 import QtCore

class HardwareManager(QtCore.QObject):
    """Generic hardware manager base class. Should be inherited by
    other hardware manager classes.

    Attributes:
        data_handler -- The application's data handler object.
        settings -- The application's settings object.
    """

    def __init__(self, data_handler, settings):
        """Initialise the hardware manager.
        For subclasses, setup() should be used for setup code rather than
        __init__().
        """

        Keyword arguments:
            data_handler -- The application's serialplot.datahandler object.
            settings -- The application's serialplot.settings object.
        """

        super().__init__()

        self.data_handler = data_handler
        self.settings = settings
        self.settings.settingsChanged.connect(self.updateSettings)

        self.setup()

    def setup(self):
        """Run hardware manager setup code.
        This should be overloaded in subclasses.
        """

        pass

    def updateSettings(self):
        """Update manager settings.
        This should be overloaded in subclasses if settings functionality is
        required.
        """

        pass

    def sendData(self, data):
        """Should be overloaded in subclasses to send data bytes."""

        pass
```

## hw/serialmanager.py

```
import serial
from serial.serialutil import SerialException
from PyQt4 import QtCore

from serialplot.hw.manager import HardwareManager

class SerialManager(HardwareManager):
    error = QtCore.pyqtSignal(str)
    portOpen = QtCore.pyqtSignal()
    portClosed = QtCore.pyqtSignal()

    def setup(self):
        self.ser=serial.Serial()

        self.incoming_timer = QtCore.QTimer()
        self.incoming_timer.timeout.connect(self.incoming_timer_tick)

    def updateSettings(self):
        settings = self.settings.getSettingsGroup('serial')
        self.ser.port=settings['port']
        self.ser.baudrate=settings['baud']
        self.ser.timeout=0
        self.ser.parity=settings['parity']
        self.ser.bytesize=settings['dataBits']
        self.ser.stopbits=settings['stopBits']
        self.ser.xonxoff=False
        self.ser.rtscts=False
        self.ser.dsrdtr=False

        # Set flow control
        fc = settings['flowControl']
        if (fc == 'XON/XOFF'):
            self.ser.xonxoff = True
        elif (fc == 'RTS/CTS'):
            self.ser.rtscts = True
        elif (fc == 'DSR/DTR'):
            self.ser.dsrdtr = True

    def openPort(self):
        timer_interval = 50

        try:
            self.ser.open()
            # Flush buffers so we don't get a mass of data coming in!
            self.ser.flushInput()
            self.ser.flushOutput()
            self.incoming_timer.start(timer_interval)
            self.portOpen.emit()
            print("Port open")
            return True
        except (serial.SerialException, ValueError) as s:
            self.error.emit(str(s) + "\n\nPlease check your settings.")
            print("Port open fail")
            self.closePort()
            return False

    def closePort(self):
        self.incoming_timer.stop()
        self.ser.close()
        self.portClosed.emit()
        print("Port closed")
```

```
def incoming_timer_tick(self):
    buf_size = 4096
    try:
        buf = self.ser.read(buf_size)
    except ValueError as s:
        self.error.emit(str(s))
    except SerialException as s:
        self.closePort()
        self.error.emit("An error occurred whilst reading from the serial port:
\n" + str(s))
    else:
        for item in buf:
            self.data_handler.addDataItem(item)

def sendData(self, data):
    try:
        self.ser.write(bytes(data))
    except IndexError:
        self.error.emit("Sent data is too long for protocol.")
    except ValueError as s:
        self.error.emit(str(s))
    except AttributeError:
        self.error.emit("Failed to send data. The port is not open.")
    except SerialException as s:
        self.closePort()
        self.error.emit("An error occurred whilst writing to the serial port:
\n" + str(s))
```

## util/

The `util` sub-package contained various files from third party sources:

- `__init__.py`: empty file, to declare `util` as a package.
- `list_ports*.py`: files from pySerial. Although included with the pySerial source code, in some distributions, these files are not included, so they are included with the application to ensure they are available.
- `sympy`: selected files from the SymPy core and parsing sub-packages, to avoid the need to depend on the full SymPy package.