ISC-2425-25+
Quick Start Companion

Software version: 1.1.1

AN-50-008

www.minicircuits.com

Table of Contents

1. Introduction 1
2. Graphical User Interface 2
   - 2.1 GUI Layout 2
   - 2.2 Using the GUI 3
3. Source Code 5

The source code of the GUI consists of the following files: 5

- 3.1 Core Concepts 5
  - 3.1.1 Slots and signals 5
  - 3.1.2 MainWindow class 5
  - 3.1.3 QDebug 6
- 3.2 Constructor 6
  - 3.2.1 Preparing the serial port 6
    - 3.2.1.1 Error handling 7
    - 3.2.1.2 Setting the port name 7
  - 3.2.2 Setting up port polling 8
    - 3.2.2.1 Updating the port list 9
  - 3.2.3 UI start-up state 10
- 3.3 Button management – Serial Connections 11
  - 3.3.1 Ports comboBox 11
  - 3.3.2 Connect 11
  - 3.3.3 Disconnect 12
  - 3.3.4 Auto-Detect & Connect 12
- 3.4 Button management – Command Buttons 13
  - 3.4.1 WriteRead 14
  - 3.4.2 WriteRead_OK 16
  - 3.4.3 Printing communications to the log 16
- 3.5 Sweeping 17
  - 3.5.1 Executing a sweep and parsing the data 18
  - 3.5.2 Drawing a plot from the sweep data 20
  - 3.5.3 Plot notation buttons 21
- 3.6 About button 22

1. Introduction

The Quick Start Companion application is a simple graphical user interface (GUI) application that can be used side-by-side with the Quick Start Guide. More specifically, it's best used alongside the latter half of the guide, which focuses on the command line interface.

This document serves to provide additional information about the codebase of the Quick Start Companion application offered as an example for software integration of the ISC-2425-25+ small signal generator.

The Quick Start Companion application has been created using the Qt framework for C++. It is recommended to interact with the source code using the Qt Creator application.

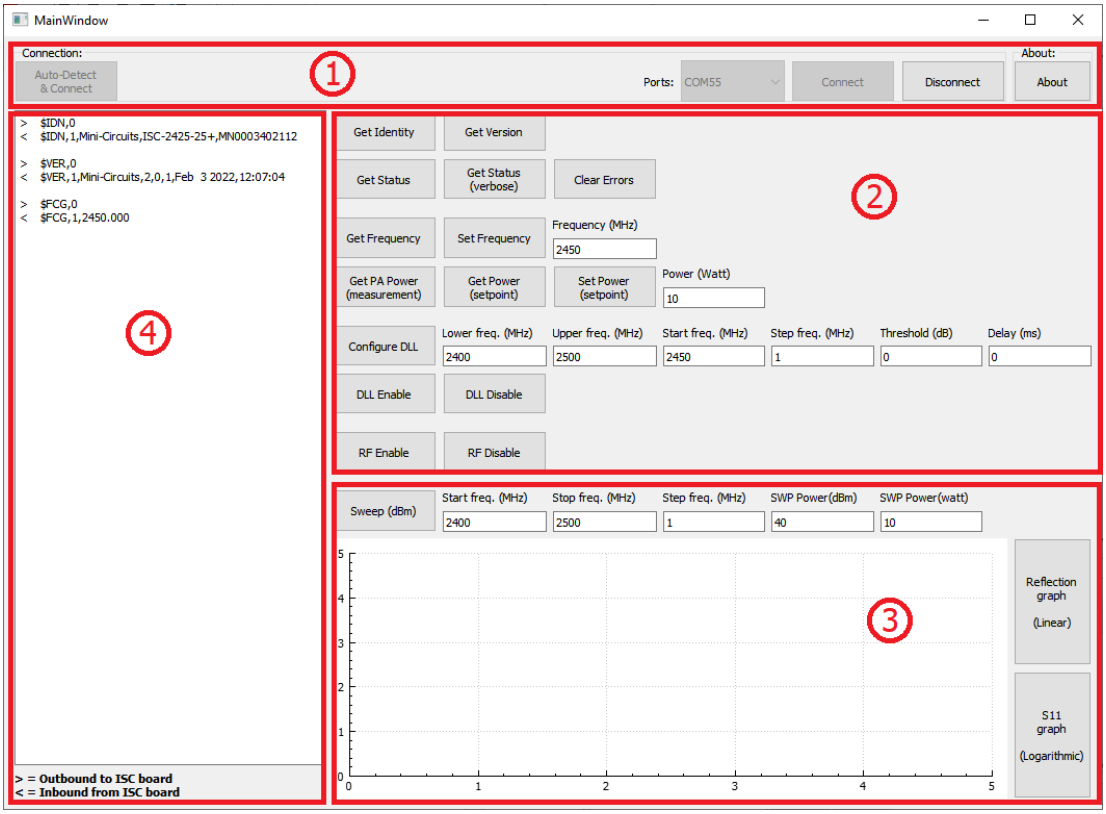
Qt is available for download here: https://www.qt.io/download

2. The Graphical User Interface

Explanation of the GUI

2.1 GUI Layout

The GUI can be divided up into 4 sections



① Top bar

Contains information for establishing a connection with the ISC board. It also contains the 'about' button.

② Command buttons section

Contains buttons and lineEdits for sending commands to the ISC board.

③ Sweep section

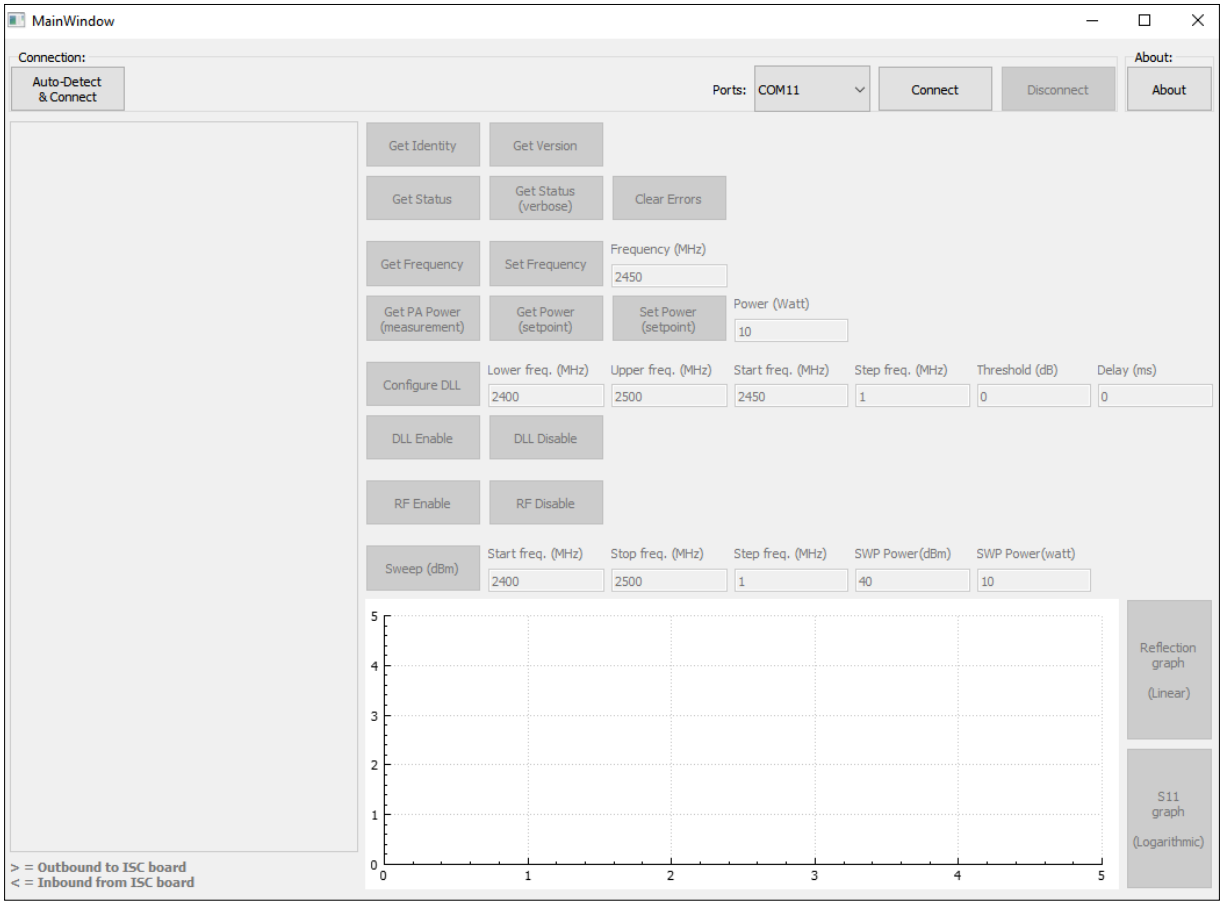
Contains buttons and lineEdits for executing a sweep with the ISC board. Also contains a plot area to displaythe results of the sweep, and two additional buttons to switch between two ways to plot the data.

④ Communications log

Displays all the inbound and outbound communication between the GUI and the ISC board.

2.2 Using the GUI

At startup of the GUI, only the top bar is enabled, everything else disabled and greyed out. The user isrequired to establish communication with an ISC board before continuing.



There are two ways to go about this:

Manual

- 1. Find the serial port of your ISC board (for example via the device manager) and manually select thename of its port from the Ports comboBox.
- 2. Press the ‘Connect’ button to try opening the connection.

Automatic

- Simply press the ‘Auto-Detect & Connect’ button in the top left corner. The GUI will connect to the firstISC board it detects. If no ISC board is detected the GUI will display a pop-up message and exit.

After connecting successfully to an ISC, the rest of the GUI becomes available.

The command buttons are laid out in an order meant to intuitively align with the instructions of the QuickStart Guide.

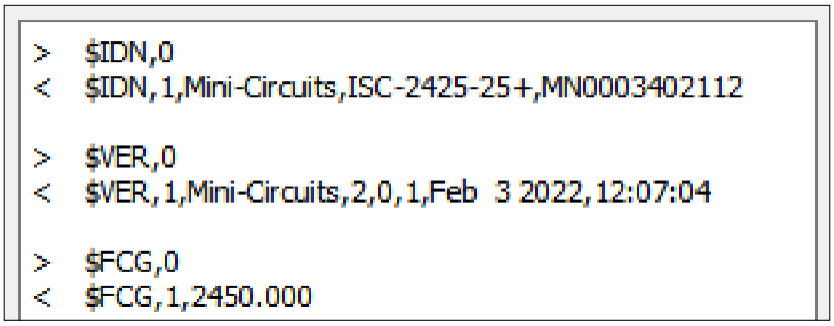
The following buttons are present:

Button	Command String	Comment
Get Identity	$IDN,0	
Get Version	$VER,0	
Get Status	$ST,0	
Get Status (verbose)	$ST,0,1	The extra argument tells the ISC board to return a verboselist of the errors instead of an error code.
Clear Errors	$ERRC,0	
Get Frequency	$FCG,0	
Set Frequency	$FCS,0,?	Fills ? with value(s) from the adjacent lineEdit(s).
Get PA Power		
(measurement)	$PPG,0	
Get Power (setpoint)	$PWRG,0	Fills ? with value(s) from the adjacent lineEdit(s).
Set Power	$PWRS,0,?	Fills ? with value(s) from the adjacent lineEdit(s).
Configure DLL	$DLES,0,?,?,?,?,?,?	
DLL Enable	$DLES,0,1	
DLL Disable	$DLES,0,0	
RF Enable	$ECS,0,1	
RF Disable	$ECS,0,0	
Sweep (dBm)	$SWPD,0,?,?,?,?,0	Fills ? with value(s) from the adjacent lineEdit(s).

When a button is pressed, the associated command string is sent to the ISC board, after which it returns areply.

Every command that is sent and received is displayed in the communications log on the left side of the GUI.

- Outbound communication (to the ISC) is marked with a '>' symbol
- Inbound communication (from the ISC) is marked with a '<' symbol



It's the user's responsibility to interpret the messages that go back and forth, according to availableinformation from the Quick Start Guide and Command Language Manual.

Notably the Sweep command has some additional handing. The results of the ISC's reply are parsed andvisualized in a graph/plot at the bottom of the GUI.

The two buttons right of the plotting area which read 'Reflection graph (Linear)' and 'S11 graph (Logarithmic)'will redraw the data of the plot in linear (%) or logarithmic (dB) notation respectively.

3. Source Code

The source code of the GUI consists of the following files:

File	Description
QSComp.pro	Project file which is opened with Qt creator.
main.cpp	Main file which starts the GUI.
mainwindow.h	Header of the GUI code.
mainwindow.cpp	Implementation of the GUI code.
mainwindow.ui	File generated with the UI designer that describes the layout of the UI.
qcustomplot.h	Header of plotting library used for sweep.
qcustomplot.cpp	Implementation of plotting library used for sweep.
about.qrc	A Qt resource file.
about.txt	Text file with contents for the 'about' button.

Among these files, the only ones of real interest are **mainwindow.cpp** and its header file **mainwindow.h**, as they contain all the interactions between the GUI and the ISC board.

3.1 Core Concepts

The following are a few core concepts the reader should keep in mind while looking at the code.

3.1.1 Slots and signals

The Qt framework has a mechanism called signals and slots, which allows different objects to communicate with one another. One object 'emits' a signal, and another object reacts to it via a slot function. This is relationship is typically set up through the 'connect' function.

```
Counter a, b;
QObject::connect(&a, &Counter::valueChanged,
                 &b, &Counter::setValue);
```

Qt Signals & Slots documentation: https://doc.qt.io/qt-5/signalsandslots.html

3.1.2 MainWindow class

The GUI is contained within **mainwindow.cpp** and mainwindow.h and **mainwindow.ui.**

mainwindow.ui is simply a file that describes layout of the GUI form and contains things like the names of UI elements. All the interactions, both UI and background stuff, are handled by the MainWindow class of mainwindow.h and -cpp.

Notably UI elements (e.g., buttons) also emit signals which are connected to the respective slot functions, though for these UI elements Qt auto-generates files at compilation which describe those particularconnections.

3.1.3 QDebug

Several lines can be found throughout the code that reference the QDebug class of Qt

```
312             qDebug() << "Port Opened";
```

These lines simply output text and/or values to the console/terminal during runtime. They are only there forthe convenience of the developer, and can otherwise be ignored entirely.

3.2 Constructor

The constructor is executed upon the instantiation of the MainWindow class at startup. It sets everythingup for use in the rest of the program.

3.2.1 Preparing the serial port

To establish communication between the software and ISC board, the QSerialPort class is used from theQt framework.

Qt QSerialPort documentation: https://doc.qt.io/qt-5/qserialport.html

The object pointer of the QSerialPort 'SG_port' for the signal generator is declared in mainwindow.h:

```
22      /* Create a new instance of the SG_port serialport object and connect its SerialPort::error
23      SG_port = new QSerialPort;
24      connect(SG_port, &QSerialPort::errorOccurred, this, &MainWindow::serialport_error_handler);
25
26      /* Set hard-coded defaults for the serialport */
27      SG_port->setBaudRate(QSerialPort::Baud115200);
28      SG_port->setDataBits(QSerialPort::Data8);
29      SG_port->setParity(QSerialPort::NoParity);
30      SG_port->setFlowControl(QSerialPort::NoFlowControl);
31      SG_port->setStopBits(QSerialPort::OneStop);
32
33      /* Set default portname for SG_port */
34      update_port_list();                                                 //Populate the list of
35      on_comboBox_ports_activated(ui->comboBox_ports->currentText());     //Set the portname to t
```

This involves:

- Setting up error handling (line 24)
- Setting up the serial parameters (baudrate, databits, etc.) (line 27-31)
- Setting up the port name. (line 34-35)

3.2.1.1 Error handling

The Qt framework has a mechanism called signals and slots, which allows different objects to communicatewith one another. In this case it is used for the error-handling of the serial port.

Here the 'errorOccurred' signal of the QSerialPort 'SG_port' is connected to the slot function 'serialport_ error_handler' of 'this' instance of the MainWindow class.

```
23      SG_port = new QSerialPort;
24      connect(SG_port, &QSerialPort::errorOccurred, this, &MainWindow::serialport_error_handler);
```

The error handler function looks like this:

```
101 void MainWindow::serialport_error_handler(QSerialPort::SerialPortError error)
102 {
103     if (error == QSerialPort::NoError || error == QSerialPort::TimeoutError)
104     {
105         return;
106     }
107     else
108     {
109         QMessageBox message;
110         message.critical(0, "Serialport Error", SG_port->errorString());
111         on_pushButton_disconnect_clicked();
112     }
113     qDebug() << error;
114 }
```

If the error type of the serial port is 'NoError' or 'TimeoutError', it is ignored and the error handler function exits early. NoError means everything is all good, and the TimeoutError can occasionally be a false alarm raised by function like Sweep, which simply requires a bit of time to respond.

All other errors spawn a pop-up message with the error and close the serial port. This is done by re-using the 'on_pushButton_disconnect_clicked' slot function which would normally be called when pressing the disconnect button in the GUI.

3.2.1.2 Setting the port name

Setting the default port name is a two-part action.

```
34      update_port_list();
35      on_comboBox_ports_activated(ui->comboBox_ports->currentText());
```

First a list of ports is generated by the 'update_port_list' function. This fills the ports comboBox in the GUI with viable port names. This is covered in detail in chapter 3.2.2.1 – Updating the port list .

Then, as a default value, the first item in the comboBox is set as the (placeholder) port name parameter for the serial port, so that it has something to work with from the get-go. This part re-uses the on_comboBox_ ports_activated slot function, which is called whenever the user chooses an item in the comboBox. This is covered in a bit more detail in chapter 3.3.1 - Ports comboBox.

3.2.2 Setting up port polling

The next part of the constructor sets up the port polling of the GUI.

The name of a serial port is something that is flexible. For example, if two signal generators are plugged into the same computer, each will have a different name/number, and the user should be able to connect to the particular one they need.

To account for devices appearing and disappearing, the list of available ports is kept up to date continuously using a timer (QTimer object).

Qt QTimer documentation: https://doc.qt.io/qt-5/qtimer.html

```
37      /* Set up polling of COM ports and periodic updating of the ports comboBox
38       * Create a new instance of the port_poll_timer QTimer object and connect its QTi
39      port_poll_timer = new QTimer();
40      connect(port_poll_timer, &QTimer::timeout, this, &MainWindow::update_port_list);
41
42      /* Start the timer. Everytime it expires, the timeout() signal is emitted, which t
43      port_poll_timer->start(POLL_TIMER);
44
45      /* Configure visuals correctly */
46      ui->plainTextEdit->setTabStopDistance(20);  //Configure the Tab length in pixels t
47      show_connection_buttons(true);              //Enable the serial port management bu
48      show_main_buttons(false);                   //Disable the messaging buttons
49  }
```

The timer 'port_poll_timer' has its timeout signal connected to the update_port_list slot function.

Afterwards, the timer is started with the value POLL_TIMER provided as the argument.

```
7   #define POLL_TIMER 1000
```

'POLL_TIMER' is defined near the top of mainwindow.cpp as 1000. So, the timer operates at intervals of 1000ms, or 1 second. Every time it expires, the timeout() signal is emitted, which triggers the update_port_list slot function.

3.2.2.1 Updating the port list

This is the update_port_list slot function:

```
/* Check available serial ports, if the port contents have changed (either the
 * A timer (port_poll_timer) calls this function every second. */
void MainWindow::update_port_list()
{
    QList<QSerialPortInfo> port_info = QSerialPortInfo::availablePorts();
    bool ports_changed = false;

    if (port_info_old.count() != port_info.count())
    {
        ports_changed = true;
    }
    else
    {
        for (int i = 0; i < port_info_old.count(); i++)
        {
            if (port_info_old.at(i).portName() != port_info.at(i).portName())
            {
                ports_changed = true;
            }
        }
    }

    if (ports_changed)
    {
        ui->comboBox_ports->clear();
        for (int i = 0; i < port_info.count(); i++)
        {
            ui->comboBox_ports->addItem(port_info.at(i).portName());
        }
    }

    port_info_old = port_info;
}
```

This function collects a list of available port information ‘port_info’ and compares it against the previous set of port information stored in ‘port_info_old’. If a difference is detected in the number of ports or the names of the ports, the boolean ‘ports_changed’ is set to true.

If ‘ports_changed’ is true, the contents of the ports comboBox in the UI is updated (cleared and repopulated) with the new list of names from port_info.

This approach lets the list stay up to date as needed, but avoids interrupting the user experience by constantly resetting the contents of the comboBox while the user is interacting with it.

3.2.3 UI start-up state

The last part of the constructor configures the (visible) state of the GUI at startup.

```
46      ui->plainTextEdit->setTabStopDistance(20);
47      show_connection_buttons(true);
48      show_main_buttons(false);
49  }
```

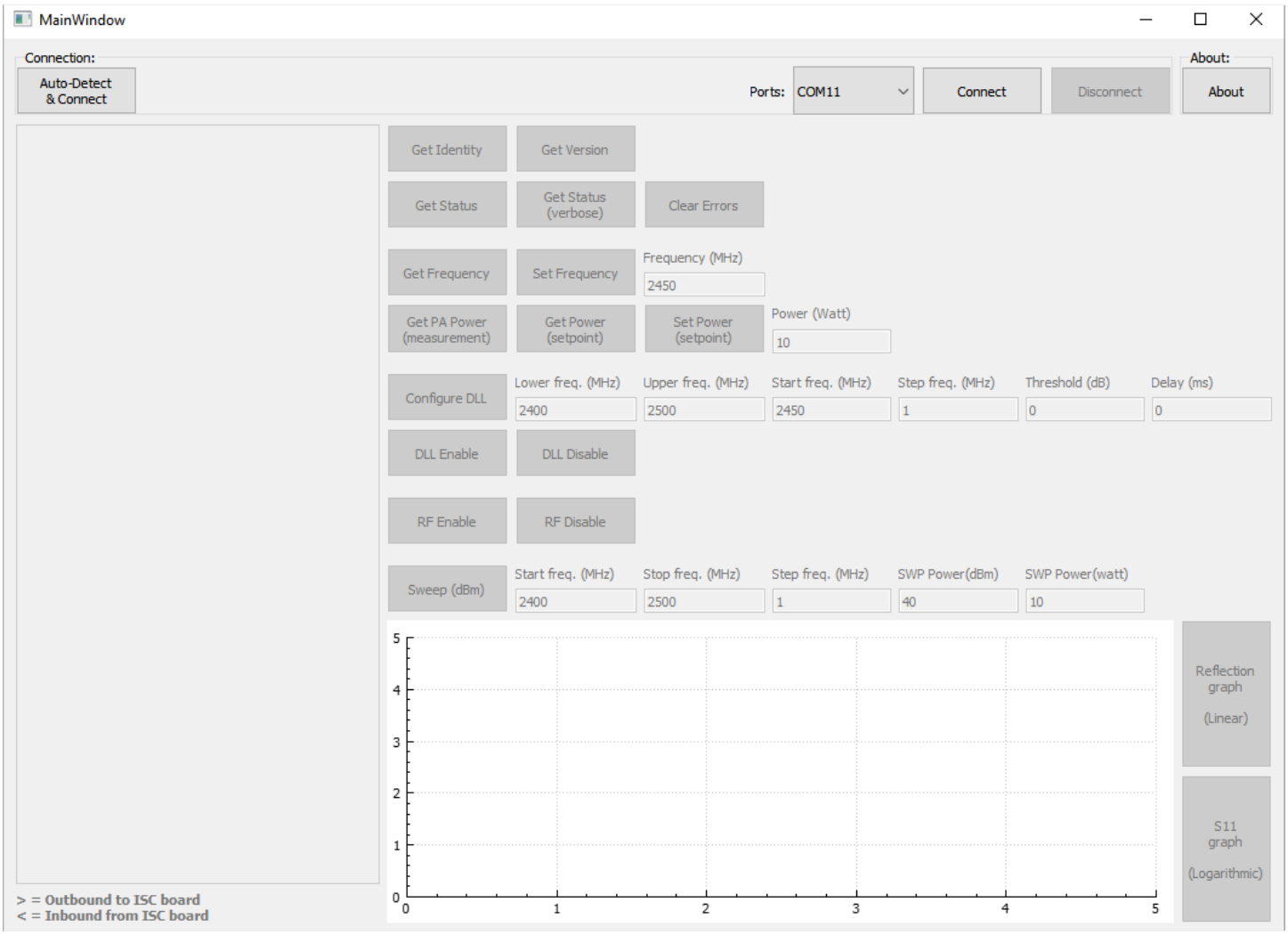
The tab length of the communications log is set to 20 pixels width, purely for aesthetic reasons.More importantly, various parts of the GUI are enabled or disabled.

At start-up, the connection to the signal generator is not yet opened, so the connection buttons are enabled(and the disconnect button is disabled) using the show_connection_buttons function:

```
63  /* Enable/Disable UI elements pertaining to establish
64  void MainWindow::show_connection_buttons(bool enable)
65  {
66      ui->comboBox_ports->setEnabled(enable);
67      ui->pushButton_connect->setEnabled(enable);
68      ui->pushButton_autoconnect->setEnabled(enable);
69      ui->pushButton_disconnect->setEnabled(!enable);
70  }
71
72  /* Enable/Disable UI elements pertaining to sending c
73  void MainWindow::show_main_buttons(bool enable)
74  {
75      ui->frame->setEnabled(enable);
76  }
```

And all the buttons for sending commands (and pretty much every other part of the UI) are disabled using the show_main_buttons function, which disables all the UI elements residing within a container in the UI called 'frame':

This results in a start-up state of the GUI that looks like this:



3.3 Button management – Serial Connections

The top of the GUI contains 4 UI elements related to serial connections:

- Ports comboBox
- Connect button
- Disconnect button
- Auto-Detect & Connect button

3.3.1 Ports comboBox

Whenever the user chooses an item in the comboBox, this function reconfigures the port name parame-ter of QSerialPort 'SG_port'.

```
300 void MainWindow::on_comboBox_ports_activated(const QString &arg1)
301 {
302     SG_port->setPortName(arg1);
303     qDebug() << "Port name:" << SG_port->portName();
304 }
```

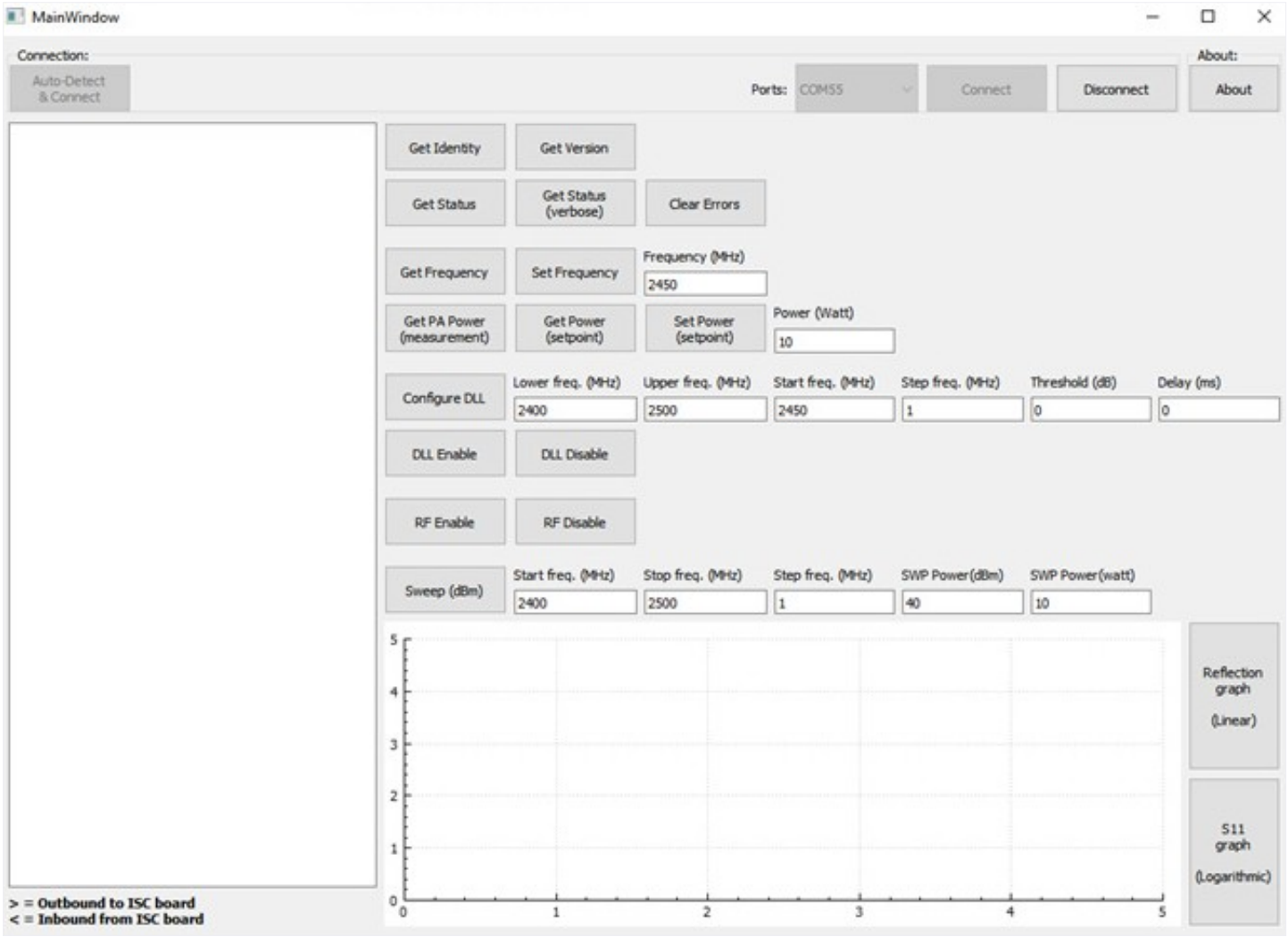
3.3.2 Connect

When the connect button is pressed, the GUI attempts to open the configured serial port.

If the port is opened successfully, the state of the GUI is updated. The connection buttons are disabled(disconnect is enabled), and all the other elements of the UI are enabled for the user to interact with.

```
307 void MainWindow::on_pushButton_connect_clicked()
308 {
309     if (SG_port->open(QIODevice::ReadWrite))
310     {
311         show_connection_buttons(false);
312         show_main_buttons(true);
313         qDebug() << "Port Opened";
314     }
315 }
```

The resulting state of the GUI looks like this:



3.3.3 Disconnect

When the disconnect button is pressed, the serial port connection is closed, and the UI is reverted to thesame state as at start-up.

```
318  void MainWindow::on_pushButton_disconnect_clicked()
319  {
320      if (SG_port->isOpen())
321      {
322          SG_port->close();
323          show_connection_buttons(true);
324          show_main_buttons(false);
325          qDebug() << "Port Closed";
326      }
327  }
```

3.3.4 Auto-Detect & Connect

The auto-detect button automatically recognizes and connects to a signal generator board.

```
330 void MainWindow::on_pushButton_autoconnect_clicked()
331 {
332     ui->comboBox_ports->setCurrentText(autodetect_SG_port().at(0).portName());
333     on_comboBox_ports_activated(ui->comboBox_ports->currentText());
334     on_pushButton_connect_clicked();
335 }
```

First the selection of the ports comboBox is updated with a value returned from the function ‘autodetect_SG_port’. Then the ‘on_pushButton_connect_clicked’ function is reused to connect to the port.

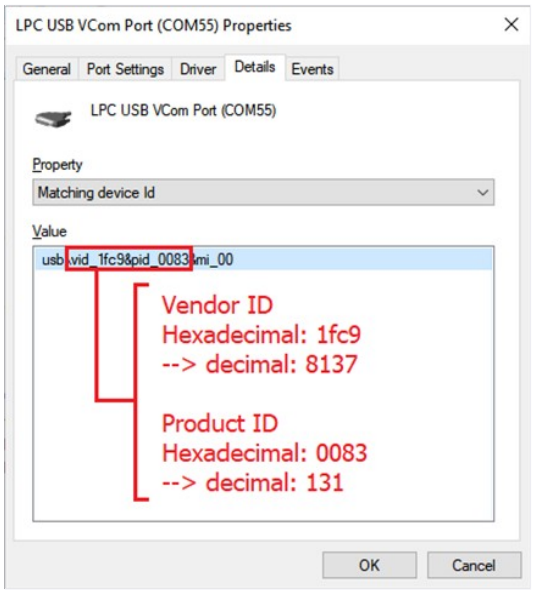
The selection of the ports occurs using the ‘autodetect_SG_port’ function:

```
151 QList<QSerialPortInfo> MainWindow::autodetect_SG_port()
152 {
153     QList<QSerialPortInfo> infolist;
154     for (const QSerialPortInfo& info : QSerialPortInfo::availablePorts())
155     {
156         if ((info.vendorIdentifier() == 8137 && info.productIdentifier() == 131))
157         {
158             infolist += info;
159         }
160     }
161
162     if (infolist.size() > 1)
163     {
164         qDebug() << "Multiple signal generator boards found";
165     }
166
167     /* If the list comes up empty show a pop-up message and exit the program
168      * This is done for the sake of simplicity, as the function that requests the l
169     if (infolist.isEmpty())
170     {
171         QMessageBox message;
172         message.critical(0, "COULD NOT AUTO-DETECT SIGNAL GENERATOR BOARD",
173                             "No signal generator board auto-detected at any port."
174                             "\nSerial connection cannot be established.\n"
175                             "\n1. Ensure signal generator board is connected."
176                             "\n2. Try again."
177                             "\n3. If problem persist try manual connect");
178         exit(-1);
179     }
180
181     return infolist;
182 }
```

Similarly, to the update_port_list function, this function also collects a list of ports. However, the differenceis that this one filters the contents.

An ISC signal generator board can be detected by looking at its ‘vendor identifier’ and ‘product identifier’.For this board model, the values of these identifiers are 8137 and 131 respectively.

These ID’s can be retrieved from the properties menu of the device, in the device manager:



If no board is detected and the ‘infolist’ comes up empty after going through all the available ports, the GUIpops an error message that no board was detected and promptly exits.

- Remark: This is done to keep things simple. Were an empty list to be returned instead, it would result inout-of-scope list references in other parts of the code and the software would crash.

Assuming a viable signal generator port was found, infolist is returned, and the first entry in the list is usedto update the comboBox selection and port name of the SG_port.

3.4 Button management – Command Buttons

Each button in the GUI has a slot function which is triggered by the respective ‘clicked’ signal. When thebutton is clicked a command string is sent to the ISC board through the serial port interface.

```
371  void MainWindow::on_pushButton_get_frequency_clicked()
372  {
373      writeRead("$FCG,0");
374  }
```

For example when the ‘Get Frequency’ button is pressed, the string “$FCG,0” is sent.

Some commands require additional input from the user. For example, the ‘Set Frequency’ command takesa numeric input for the desired frequency value.

```
376  void MainWindow::on_pushButton_set_frequency_clicked()
377  {
378      writeRead("$FCS,0," + ui->lineEdit_frequency->text());
```

When the ‘Set Frequency’ button is pressed, the string “$FCS,0,” is concatenated with the contents of thelineEdit ‘lineEdit_frequency” and is sent to the ISC board. E.g., $FCS,0,2450.

Most commands reply with only one line. For example:

- > $IDN,0

  < $IDN,1,Mini-Circuits,ISC-2425-25+,MN0003402112
- > $PWRG,0

  < $PWRG,1,10.000000
- > $ECS,0,1

  < $ECS,1,OK

However, a small handful of commands return more than a single line. For example, the verbose version ofthe ‘Status Get’ command:

- > $ST,0,1

  < $ST,1,RESET_DETECTED

  < $ST,1,EXTERNAL_SHUTDOWN_DETECTED

  < $ST,1,OK

To handle these two different types of responses, there are two separate functions for sending commandsand receiving responses:

- writeRead
- writeRead_OK

3.4.1 writeRead

The ‘writeRead’ function is used for commands with single-line responses.Here’s a

breakdown down of the function:

1. It takes the command string for the signal generator as an argument, checks whether the SG_port is stillopen at all before continuing, and then prepares a QString ‘rx’ in which to store a response to the command.

```
193  QString MainWindow::writeRead(QString tx)
194  {
195      if (!SG_port->isOpen())
196      {
197          return "";
198      }
199
200      QString rx = "";
```

2. It attempts to write the command string to the ISC through the serial port interface, and waits up to 250ms for an indication that the transmission has begun. If the transmission has begun, the outbound message is also printed in the communications log.

Notably a 'carriage return' (\r) and 'line feed' (\n) are appended to the command string when writing it out. These are treated as an end of message indicator by the ISC board. Without them the board would wait indefinitely for additional inputs and not respond.

```
204     SG_port->write((const char*)(tx+"\r\n").toUtf8());
205
206     if (SG_port->waitForBytesWritten(250))
207     {
208         print_message(direction::outbound, tx);
209     }
```

3. In a while loop, the function awaits a response by the ISC that contains "\r\n", indicating the end of the response. The function waits for up to 500ms at a time until SG_port indicates with a 'readyRead' signal that there is something to be read from the read buffer of SG_port. When the SG_port is ready to be read (or if the timer expires), the contents of the read buffer are appended to the QString 'rx'. 'rx' is then parsed for a possible error, indicated by an "ERR" string. The open state of SG_port is also checked to make sure the serial port is still working while it is stuck in the while loop. If it turns out there's an error response from the SG or the serial port has closed for some reason, the while loop is broken prematurely, otherwise it keeps looping till a "\r\n" is detected in the contents of rx.

```
212     while (!rx.contains("\r\n"))
213     {
214         SG_port->waitForReadyRead(500);      //Wait up
215         rx += QString::fromUtf8(SG_port->readAll());
216
217         if (rx.contains("ERR") || !SG_port->isOpen())
218         {
219             break;
220         }
221     }
```

4. After breaking out of the while loop, the response string is printed in the communications log, and 'rx' is returned to the function which called writeRead, allowing further interaction with the response string if necessary.

```
224     print_message(direction::inbound, rx);
225
226     return rx;
227 }
```

3.4.2 writeRead_OK

The writeReady_OK function is used for commands with multi-line responses. Commands with multi-line responses return several lines in one go, each ending with “\r\n” and finally finish up with an “OK\r\n”.

The function ‘writeRead_OK’ behaves largely identical to ‘writeRead’. The only noteworthy difference is in the loop that waits for a complete response from the ISC board. In this case the function waits until the response string rx contains the string “OK\r\n”, indicating the multi-line response has been completed.

```
253     while (!rx.contains("OK\r\n"))
254     {
255         SG_port->waitForReadyRead(500);     //Wait up to
256         rx += QString::fromUtf8(SG_port->readAll());
257         if (rx.contains("ERR") || !SG_port->isOpen())
258         {
259             break;
260         }
261     }
```

3.4.3 Printing communications to the log

The GUI logs all the communication to and from the ISC board.

This occurs in the writeRead commands on the lines that call the print_message function.

- print_message(direction::outbound, tx);
- print_message(direction::inbound, rx);

The print_message function simply appends to the tx and rx lines to the contents of the text editor on theleft side of the GUI.

```
270 void MainWindow::print_message(MainWindow::direction dir, QString text)
271 {
272     QString str = "";
273     if (dir == direction::inbound)
274     {
275         str += ("<\t");
276 
277         //Adjusted cosmetics for multi-line responses
278         if (text.count("\r\n") > 1)
279         {
280             text.replace("\r\n", "\r\n<\t");
281             text.chop(2);   //Chop the last <\t
282         }
283     }
284     else
285     {
286         str += (">\t");
287     }
288 
289     ui->plainTextEdit->appendPlainText(str + text);
290 }
```

It takes a direction argument (dir), and a string argument (text).

If the direction of the message is inbound from the ISC board, “<\t” is inserted before the string, as well asafter the “\r\n” of each line of the response (if there are multiple), except the last.

If the direction of the message is outbound to the ISC board, “>\t” is inserted before the string.Outbound commands are always a single line, so that is sufficient.

These are purely cosmetic edits to make the transmissions a bit easier to visually parse for the user.

3.5 Sweeping

Sweeping is one of the more complicated features of the ISC board.

It gives the user a glimpse at the matching quality of their load across a range of frequencies.

The ‘Sweep (dBm)’ button performs a sweep using dBm as the preferred unit of power for both inputs andoutputs. This is done with the $SWPD command, which also takes arguments from the adjacent lineEdits.

For an improved experience the user is provided with both a dBm and watt input, though only the dBmvalue is actually used in the end. When the value in either lineEdit changes, the other is also updated:

```
429 void MainWindow::on_lineEdit_SWP_4_textEdited(const QString &arg1)
430 {
431     ui->lineEdit_SWP_5->setText(QString::number(convert_dbm_to_watt(arg1.toDouble())));
432 }
433
434 void MainWindow::on_lineEdit_SWP_5_textEdited(const QString &arg1)
435 {
436     ui->lineEdit_SWP_4->setText(QString::number(convert_watt_to_dbm(arg1.toDouble())));
437 }
```

This uses a pair of conversion functions provided for convenience.

convert_dbm_to_watt: A value in dBm is provided as the argument, and a value in watt is returned.

```
79 double MainWindow::convert_dbm_to_watt(double value_in_dBm)
80 {
81     double value_in_Watt = 0.001 * pow(10, 0.1 * value_in_dBm);
82     return value_in_Watt;
83 }
```

convert_watt_to_dbm: A value in dBm is provided as the argument, and a value in dBm is returned.

```
86 double MainWindow::convert_watt_to_dbm(double value_in_watt)
87 {
88     double value_in_dBm = (10 * log10(value_in_watt)) + 30;
89     return value_in_dBm;
90 }
```

The actions of the sweep button are two-part:

- Execute the sweep and parse the resulting data
- Draw a plot from the data (If the sweep was successful)

```
451 /* Execute an S11 Sweep */
452 void MainWindow::on_pushButton_execute_sweep_clicked()
453 {
454     if (SWP_run_sweep() == true)    //Only draw a plot
455     {
456         SWP_draw_plot(MainWindow::logarithmic);
457     }
458 }
```

3.5.1 Executing a sweep and parsing the data

Execution and parsing of the sweep data occurs in the 'SWP_run_sweep' function.

First the sweep command is sent to the signal generator. The writeRead_OK function is used and the response, which potentially spans hundreds of lines depending on the provided parameters, is stored as one long string in QString 'SWP_raw_data'.

As mentioned before, the $SWPD command is used, which takes power in dBm as an argument from 'lineEdit_SWP_4'. The power in watt from 'lineEdit_SWP_5' goes unused, and is only there for the convenience of the user.

```
462 bool MainWindow::SWP_run_sweep()
463 {
464     QString SWP_raw_data =  writeRead_OK("$SWPD,0," + ui->lineEdit_SWP_1->text()
465                                                     + "," + ui->lineEdit_SWP_2->text()
466                                                     + "," + ui->lineEdit_SWP_3->text()
467                                                     + "," + ui->lineEdit_SWP_4->text()
468                                                     + ",0");
```

Next the response's contents are checked. If the response contains the strings "$SWPD," and "OK\r\n", it indicates the sweep has completed successfully.

If the sweep is successful, the data gets divided up into more manageable chunks.

This is done by splitting the giant Qstring 'SWP_raw_data' at all of the "\r\n" bits, and storing each line as an individual item into a QStringList called 'SWP_data'. The two last items in the list (OK message and an empty line) are promptly deleted, as they are of no further use, and would only get in the way when parsingthe data.

If the sweep fails (for example because of an invalid input in the sweep lineEdits), it instead pops an error message and returns false, letting the program know it shouldn't try to draw a plot from this data.

```
469     QStringList SWP_data;
470
471     /* If the data checks out, split the long message up by the individual line
472     if(SWP_raw_data.contains("$SWPD,") && SWP_raw_data.contains("OK\r\n"))
473     {
474         SWP_data = SWP_raw_data.split("\r\n");
475         //Remove the OK entry + the empty line that comes after.
476         SWP_data.removeLast();
477         SWP_data.removeLast();
478     }
479     else
480     {
481         QMessageBox message;
482         message.warning(0, "Sweep Error", "Sweep data invalid / incomplete.");
483         return false;
484     }
```

The data from the sweep will be saved in a set of QVectors of the type double.Qt

QVector documentation: https://doc.qt.io/qt-5/qvector.html

- Remark: This data will be reused by other functions, so these QVectors have been declared ahead of time in mainwindow.h, making them persistent and accessible to any function in the MainWindow class.

```
66      QVector<double> SWP_freq_data;
67      QVector<double> SWP_fwd_data;
68      QVector<double> SWP_rfl_data;
69      QVector<double> SWP_s11_dbm_data;
70      QVector<double> SWP_s11_watt_data;
```

The QVectors are resized according to the number of lines available for parsing in 'SWP_data'.

```
486     /* Resize the QVectors for the correct amount of data */
487     SWP_freq_data.resize(SWP_data.count());
488     SWP_fwd_data.resize(SWP_data.count());
489     SWP_rfl_data.resize(SWP_data.count());
490     SWP_s11_dbm_data.resize(SWP_data.count());
491     SWP_s11_watt_data.resize(SWP_data.count());
```

Next the string data from the QStringList 'SWP_data' is processed. Every item from 'SWP_data' is done oneby one.

The contents of each line, which look something like "$SWP,1,2400.00,47.23,35.53\r\n", are split at the comma character and stored in a new QStringList called 'data'.

Starting from 0, items number 2,3 and 4 of QStringList 'data' contain the frequency, forward power, and reflected power respectively. These values are converted from QStrings to doubles and stored at appropriate index in their respective QVectors

```
493     /* Process the data */
494     for (int i = 0; i < SWP_data.count(); i++)
495     {
496         /* Split each line of SWP data by the comma's */
497         QStringList data = SWP_data.at(i).split(",");
498
499         /* Fill the QVectors with data, calculate S11 */
500         if (data.contains("$SWPD") && data.count() == 5)
501         {
502             SWP_freq_data[i] = data.at(2).toDouble();
503             SWP_fwd_data[i] = data.at(3).toDouble();
504             SWP_rfl_data[i] = data.at(4).toDouble();
```

From the forward and reflected power measurements the RF matching values are also calculated.

- The S11 values are calculated in dB.
- The reflection values, after converting the power values from dBm to watts, are calculated in %.Both are

stored their respective QVectors as well.

```
506             /* Calculate S11 (dB) / Reflection (%) using the above forward and r
507             SWP_s11_dbm_data[i] = SWP_rfl_data[i] - SWP_fwd_data[i];
508             SWP_s11_watt_data[i] = (convert_dbm_to_watt(SWP_rfl_data[i])
509                                     /
510                                     convert_dbm_to_watt(SWP_fwd_data[i])) * 100;
511         }
512     }
513     return true;
514 }
```

Remark: the code here has been slightly adjusted to keep the screenshot sufficiently legible.

Finally, the 'SWP_run_sweep' function returns true, indicating the sweep has been executed successfullyand the data is ready to use.

3.5.2 Drawing a plot from the sweep data

After the sweep has been executed successfully, the data can be used to draw a plot and give the user amore easily digestible overview than a bunch of text.

This is done in the 'SWP_draw_plot' function.

The SWP_draw_plot function uses the open source QCustomPlot library by Emanuel Eichhammer, availablefor download here: https://www.qcustomplot.com/index.php/download

Documentation of the QCustomPlot is available here:

- https://www.qcustomplot.com/index.php/tutorials/settingup
- https://www.qcustomplot.com/documentation/index.html

First the contents of the sweep QVectors are checked. They mustn't be empty.

```
516 void MainWindow::SWP_draw_plot(S11_notation notation)
517 {
518     if (SWP_freq_data.isEmpty() || SWP_s11_dbm_data.isEmpty() || SWP_s11_watt_data.isEmpty())
519     {
520         return;
```

Next the X-axis of the plot is configured to display the frequency data saved in the frequency QVector.Additionally, the Y-axis, which will show S11 data, is configured for 2 decimals of precision.

```
523     ui->SWP_plot->addGraph();
524     ui->SWP_plot->xAxis->setLabel("Frequency (MHz)");
525     ui->SWP_plot->xAxis->setRange(SWP_freq_data[0],SWP_freq_data[SWP_freq_data.size()-1]);
526     ui->SWP_plot->yAxis->setNumberFormat("f");
527     ui->SWP_plot->yAxis->setNumberPrecision(2);
```

Now it is time to deal with the S11 data.

Two variables of the type double are prepared, to store the smallest and largest values of the S11 data, tobe used later for setting the range of the Y-axis.

The 'SWP_draw_plot' function takes an 'S11_notation' argument. This argument can be either 'logarithmic'or 'linear', and the plot is drawn differently depending on the choice.

If the notation is logarithmic, the plot uses data from the 'SWP_s11_dbm_data' QVector.

The smallest and largest values from the QVector are stored in 'min_val' and 'max_val', though they areignored if they exceed 0, as an S11 graph is expected to consist of only negative values.

The visible limits of the plot's Y-axis are set to smallest and largest S11 values multiplied by a factor of 1.1for improved visibility.

```
528     double min_val, max_val;
529
530     if(notation == S11_notation::logarithmic)
531     {
532         ui->SWP_plot->graph(0)->setData(SWP_freq_data,SWP_s11_dbm_data);
533
534         min_val = *std::min_element(SWP_s11_dbm_data.constBegin(),SWP_s11_dbm_data.constEnd());
535         if (min_val > 0){
536             min_val = 0;
537         }
538         max_val = *std::max_element(SWP_s11_dbm_data.constBegin(),SWP_s11_dbm_data.constEnd());
539         if(max_val < 0){
540             max_val = 0;
541         }
542
543         ui->SWP_plot->yAxis->setRange(min_val*1.1, max_val*1.1);
544         ui->SWP_plot->yAxis->setLabel("S11 (dB)");
```

If the notation is linear, the plot instead uses data from the 'SWP_s11_watt_data' QVector.

The largest value from the QVector is stored in 'max_val', though it is capped to 100, as a reflection graph is expected to at most have total reflection (100%). The visible limits of the plot's Y-axis are set to 0 and 'max_val'.

```
546     else
547     {
548         ui->SWP_plot->graph(0)->setData(SWP_freq_data,SWP_s11_watt_data);
549
550         max_val = *std::max_element(SWP_s11_watt_data.constBegin(),SWP_s11_watt_data.constEnd());
551
552         if(max_val <= 100)
553         {
554             max_val = 100;
555         }
556         ui->SWP_plot->yAxis->setRange(0,max_val);
```

Finally the interaction permissions for the plot are configured and the plot drawn in the GUI.

```
560     ui->SWP_plot->setInteractions(QCP::iRangeDrag | QCP::iRangeZoom | QCP::iSelectItems);
561     ui->SWP_plot->replot();
562 }
```

3.5.3 Plot notation buttons

Lastly are the two buttons located right of the plotting area.

When either of these is pressed, they activate the 'SWP_draw_plot' function again, with either the linear or logarithmic notation. SWP_draw_plot re-uses the last known measurements stored in the QVectors to re- draw the plot.

```
441 void MainWindow::on_pushButton_SWP_notation_linear_clicked()
442 {
443     SWP_draw_plot(S11_notation::linear);
444 }
445
446 void MainWindow::on_pushButton_sweep_notation_logarithmic_clicked()
447 {
448     SWP_draw_plot(S11_notation::logarithmic);
449 }
```

3.6 About button

When pressed, the about button pops a message with legal information for the application.

The text of the message is retrieved from a file called ‘about.txt’, which is loaded into the project as a‘resource’ and compiled into the executable.

```
567  void MainWindow::on_pushButton_about_clicked()
568  {
569      QMessageBox message;
570      QFile file(":/about.txt");
571      if(!file.open(QIODevice::ReadOnly))
572      {
573          message.critical(0,"Error", "Could not open about text");
574      }
575      else
576      {
577          message.about(0,"About", file.readAll());
578      }
579      file.close();
580  }
```