Complete Measurement System for Measuring High Voltage and Electrical Field Using Slab-Coupled Optical Fiber Sensors

Nikola Stan
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/7053

This Dissertation is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
Complete Measurement System for Measuring High Voltage and Electrical Field

Using Slab-Coupled Optical Fiber Sensors

Nikola Stan

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Stephen M. Schultz, Chair
Gregory P. Nordin
Aaron Hawkins
David Long
Daniel Smalley

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2018 Nikola Stan
All Rights Reserved
ABSTRACT

Complete Measurement System for Measuring High Voltage and Electrical Field Using Slab-Coupled Optical Fiber Sensors

Nikola Stan
Department of Electrical and Computer Engineering, BYU
Doctor of Philosophy

A slab-coupled optical fiber sensor (SCOS) falls into a narrow class of all-dielectric optical fiber electric field sensors, which makes it a perfect candidate for measurements of high electric fields in environments where presence of conductors is highly perturbing to the system under test. Its nonlinear response to high fields requires a new nonlinear calibration technique. A nonlinear calibration method is explained and demonstrated to successfully measure high electric fields, as well as high voltages with dynamic range up to 50 dB. Furthermore, a SCOS can be fitted into narrow spaces and make highly localized measurements due to its small size. This allows a SCOS to be integrated inside a standard high voltage coaxial cable, such as RG-218. Effects of partial discharge and arcing is minimized by development of a fabrication method to avoid introduction of impurities, especially air-bubbles, into the cable during SCOS insertion. Low perturbation of the measured voltage is shown by simulating the introduced voltage reflections to be on the order of −50 dB. It is also shown that a SCOS can be inserted into other cables without significant perturbation to the voltage.

A complete high voltage and high electric field measurement system is built based on the high-voltage modifications of the SCOS technology. The coaxial SCOS is enhanced for robustness. Enhancements include packaging a SCOS into stronger ceramic trough, strengthening the fiber with kevlar reinforced furcation tubing and protecting the sensor with metal braces and protective shells. The interrogator is protected from electromagnetic interference with an RF-shielded box. Reduction in power losses introduced by the new PANDA-SCOS technology allows interrogator bandwidths to be increased up to 1.2 GHz. The whole measurement process is streamlined with dedicated software, developed specifically for high voltage and electric field measurements with support for the nonlinear calibration.

Keywords: high electric field measurement, high voltage measurement, optical fiber sensor, lithium-niobate, slab-coupled optical fiber sensor, SCOS, corona, arcing, coaxial cable, pulsed power, fiber Bragg grating, FBG, feedback loop, strain, vibration
ACKNOWLEDGMENTS

Thanks to my family for their lasting support: mom and dad, Mirjana and Dušan, my sister Katarina, who listened to me talk about my research both in times of frustration and success, and my brother Aleksandar.

Dr. Stephen Schultz, thank you for your never-ending patience and support. And of course, Dr. Richard Selfridge, thank you for making research and learning fun and productive at the same time. You have been great mentors. Thanks to the rest of the amazing BYU faculty I have had the fortune to learn from.

Frederick Seng, thank you for being a collaborator and a friend, and likewise Spencer Chadderdon. Thank you also to: LeGrand Shumway, Rex King, Kevin Cuzner and many other students who I worked with.

There is a great number of people outside of BYU who supported me on my journey, most specially, Billy Saunders, Ron and Annette Densley.
# TABLE OF CONTENTS

**LIST OF FIGURES** .................................................................................. vii

**Chapter 1** **Introduction** ................................................................. 1
  1.1 Sensor selection ........................................................................... 1
  1.2 Contributions and dissertation outline ........................................ 6

**Chapter 2** **Background** ................................................................. 12
  2.1 Sensitivity .................................................................................... 13
  2.2 Linear calibration ........................................................................ 16

**Chapter 3** **Nonlinear calibration for measuring high electric fields with a SCOS** ......................................................... 19
  3.1 High field adaptations ................................................................. 19
    3.1.1 Materials ............................................................................. 19
    3.1.2 Invariance of the spectral shift rate .................................... 21
    3.1.3 Nonlinear calibration .......................................................... 23
  3.2 High field measurements ............................................................ 26
  3.3 Summary ..................................................................................... 31

**Chapter 4** **Non-perturbing voltage measurements in a coaxial cable** ........................................................................ 34
  4.1 Potential system perturbation ...................................................... 34
    4.1.1 Voltage reflections in the coaxial cable ............................. 34
    4.1.2 Dielectric breakdown .......................................................... 41
  4.2 Inserting a SCOS in a coaxial cable ............................................ 45
    4.2.1 Charge build-up measurement ........................................... 46
  4.3 The SCOS voltage measurements in a coaxial cable .................. 49
    4.3.1 Minimum detectable voltage .............................................. 49
    4.3.2 Maximum voltage .............................................................. 50
    4.3.3 Oscillations ........................................................................ 52
    4.3.4 Measurement accuracy ....................................................... 56
    4.3.5 Frequency response ............................................................ 58

**Chapter 5** **The complete high voltage and high electric field measurement system - Hardware** ......................................................... 60
  5.1 Interrogator ................................................................................ 60
    5.1.1 Increasing interrogator bandwidth .................................... 60
    5.1.2 Electromagnetic interference .............................................. 62
  5.2 Partial discharge and unwanted arcing ...................................... 66
    5.2.1 Corona, streamer discharges and unwanted arcing ........... 67
  5.3 200 kV high voltage generator .................................................... 72
  5.4 Improving robustness of the coaxial cable SCOS ....................... 77
    5.4.1 Bend test ............................................................................ 79
    5.4.2 Drop test ............................................................................ 80
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.3 Vibration and shipping test</td>
<td>81</td>
</tr>
<tr>
<td>Chapter 6 The complete high voltage and high electric field measurement system - Software</td>
<td>83</td>
</tr>
<tr>
<td>6.1 Running the application</td>
<td>83</td>
</tr>
<tr>
<td>6.2 Preparation for measurements</td>
<td>84</td>
</tr>
<tr>
<td>6.2.1 Resonance scanning</td>
<td>87</td>
</tr>
<tr>
<td>6.3 Calibration and results window</td>
<td>90</td>
</tr>
<tr>
<td>Chapter 7 Other SCOS contributions</td>
<td>98</td>
</tr>
<tr>
<td>7.1 SCOS technology with PANDA side-polished optical fibers</td>
<td>98</td>
</tr>
<tr>
<td>7.2 Transformer-oil resistant SCOS</td>
<td>101</td>
</tr>
<tr>
<td>7.3 Pushpull SCOS for measurements in a vibrational environment</td>
<td>103</td>
</tr>
<tr>
<td>7.4 SCOS sensitivity direction rerouting and enhancement using a passive integrated dipole antenna</td>
<td>105</td>
</tr>
<tr>
<td>Chapter 8 Increasing dynamic range of Fiber Bragg grating interrogator</td>
<td>109</td>
</tr>
<tr>
<td>8.1 Background</td>
<td>110</td>
</tr>
<tr>
<td>8.2 System overview</td>
<td>112</td>
</tr>
<tr>
<td>8.3 System characterization</td>
<td>113</td>
</tr>
<tr>
<td>8.3.1 Components</td>
<td>114</td>
</tr>
<tr>
<td>8.3.2 Low frequency strain sensitivity and range</td>
<td>114</td>
</tr>
<tr>
<td>8.3.3 Tuning rate</td>
<td>116</td>
</tr>
<tr>
<td>8.3.4 Control loop performance</td>
<td>117</td>
</tr>
<tr>
<td>8.4 Simultaneous low and high frequency strain measurement</td>
<td>121</td>
</tr>
<tr>
<td>8.4.1 Simultaneous HF and LF strain measurement</td>
<td>122</td>
</tr>
<tr>
<td>8.5 Summary</td>
<td>125</td>
</tr>
<tr>
<td>Chapter 9 Summary</td>
<td>126</td>
</tr>
<tr>
<td>9.1 Contributions</td>
<td>126</td>
</tr>
<tr>
<td>9.1.1 Nonlinear calibration</td>
<td>127</td>
</tr>
<tr>
<td>9.1.2 Coaxial cable SCOS</td>
<td>127</td>
</tr>
<tr>
<td>9.1.3 SCOS interrogator system for measuring high electric fields</td>
<td>128</td>
</tr>
<tr>
<td>9.1.4 Increasing dynamic range of FBG edge-filtering demodulation method</td>
<td>128</td>
</tr>
<tr>
<td>9.1.5 PANDA-SCOS</td>
<td>129</td>
</tr>
<tr>
<td>9.1.6 Transformer-oil-resistant SCOS</td>
<td>129</td>
</tr>
<tr>
<td>9.1.7 Push-pull SCOS</td>
<td>130</td>
</tr>
<tr>
<td>9.1.8 Antenna SCOS</td>
<td>130</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>131</td>
</tr>
<tr>
<td>Appendix A Interrogator software code</td>
<td>140</td>
</tr>
<tr>
<td>A.1 Algorithms</td>
<td>140</td>
</tr>
<tr>
<td>A.1.1 Classes</td>
<td>140</td>
</tr>
<tr>
<td>A.1.2 Functions</td>
<td>147</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

1.1 Voltage divider. .................................................. 2
1.2 D-dot electric field sensor. .................................. 3
1.3 Coaxial SCOS concept. ......................................... 4
1.4 Air presence distorts high voltage measurements. ....... 5
1.5 Calibration failure. .............................................. 5
1.6 Damage caused by arcing. .................................... 6

2.1 Coupled LN crystal. ............................................. 13
2.2 SCOS spectrum and interrogator. ............................ 14
2.3 Wavelength shift. ............................................... 15
2.4 Operating wavelength on the spectrum edge. ............. 17

3.1 SCOS materials. .................................................. 20
3.2 Mapping interrogator voltage to spectral shift. ......... 23
3.3 Slope of the SCOS resonance. ................................. 24
3.4 SCOS excitation and interrogation block diagram. .... 25
3.5 Schematic of the testing configuration. .................... 26
3.6 Simulated field distribution. .................................. 26
3.7 Electrodes with an inserted SCOS. ......................... 27
3.8 The SCOS spectrum measured with a laser. ............. 28
3.9 Mapping the calibration waveform onto the spectrum. 29
3.10 Electric field measurements using nonlinear and linear calibration method. 32
3.11 Peak values using nonlinear and linear calibration. 33

4.1 Cross section of a coaxial cable with a SCOS. ............ 35
4.2 TDR setup. ...................................................... 35
4.3 TDR measurements. ........................................... 37
4.4 Two-port network. ............................................. 38
4.5 S11 simulation models. ........................................ 38
4.6 S11 simulation results. ....................................... 39
4.7 S11 simulation. ................................................ 40
4.8 Lumped element with a SCOS. ............................... 41
4.9 The charge build-up effect. .................................. 41
4.10 Effects of the damage to the cable insulation. .......... 42
4.11 Electric field in coaxial cable as a function of distance. 43
4.12 Field in the coaxial cable with a SCOS. .................... 44
4.13 Simulated electric field with and without a SCOS. .... 45
4.14 S11 simulation models. ........................................ 46
4.15 Measurement setup. .......................................... 47
4.16 Charge build up measurements. ............................ 48
4.17 Minimum voltage measurement. ............................. 49
4.18 Minimum voltage measurements. ........................... 50
4.19 Maximum voltage measurement. ............................ 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.20</td>
<td>Discharge circuit model</td>
<td>52</td>
</tr>
<tr>
<td>4.21</td>
<td>Simulation of the 100 kV impulse</td>
<td>57</td>
</tr>
<tr>
<td>4.22</td>
<td>SCOS: Low corner frequency</td>
<td>58</td>
</tr>
<tr>
<td>5.1</td>
<td>Removing TIA from the interrogator</td>
<td>61</td>
</tr>
<tr>
<td>5.2</td>
<td>Dual interrogator schematic</td>
<td>62</td>
</tr>
<tr>
<td>5.3</td>
<td>Interrogator components</td>
<td>63</td>
</tr>
<tr>
<td>5.4</td>
<td>RF-Box interference test</td>
<td>64</td>
</tr>
<tr>
<td>5.5</td>
<td>Dual interrogator system in RF-isolated box</td>
<td>65</td>
</tr>
<tr>
<td>5.6</td>
<td>Corona discharge illustration</td>
<td>67</td>
</tr>
<tr>
<td>5.7</td>
<td>Van de Graaf generator measurement</td>
<td>68</td>
</tr>
<tr>
<td>5.8</td>
<td>Unwanted arcing</td>
<td>69</td>
</tr>
<tr>
<td>5.9</td>
<td>Conductor curvature effect on surface charge</td>
<td>71</td>
</tr>
<tr>
<td>5.10</td>
<td>Corona discharge with different electrode shapes</td>
<td>72</td>
</tr>
<tr>
<td>5.11</td>
<td>High voltage generator schematics</td>
<td>73</td>
</tr>
<tr>
<td>5.12</td>
<td>Capacitor charge hold</td>
<td>74</td>
</tr>
<tr>
<td>5.13</td>
<td>High voltage generator and components</td>
<td>75</td>
</tr>
<tr>
<td>5.14</td>
<td>High voltage generator output characteristic</td>
<td>76</td>
</tr>
<tr>
<td>5.15</td>
<td>Improving the coaxial cable SCOS robustness</td>
<td>77</td>
</tr>
<tr>
<td>5.16</td>
<td>Power loss in the SCOS after on-site visit</td>
<td>78</td>
</tr>
<tr>
<td>5.17</td>
<td>Bend test</td>
<td>80</td>
</tr>
<tr>
<td>5.18</td>
<td>Drop test</td>
<td>81</td>
</tr>
<tr>
<td>5.19</td>
<td>Vibration test</td>
<td>81</td>
</tr>
<tr>
<td>5.20</td>
<td>Shipping damage test</td>
<td>82</td>
</tr>
<tr>
<td>6.1</td>
<td>Initial view</td>
<td>84</td>
</tr>
<tr>
<td>6.2</td>
<td>Selecting sensor and connecting laser and DAQ</td>
<td>85</td>
</tr>
<tr>
<td>6.3</td>
<td>Laser low-noise feature</td>
<td>86</td>
</tr>
<tr>
<td>6.4</td>
<td>Wide-band spectrum scan</td>
<td>87</td>
</tr>
<tr>
<td>6.5</td>
<td>Single resonance scan</td>
<td>88</td>
</tr>
<tr>
<td>6.6</td>
<td>Fine frequency tuning inaccuracy</td>
<td>89</td>
</tr>
<tr>
<td>6.7</td>
<td>Maximum sensitivity wavelength</td>
<td>90</td>
</tr>
<tr>
<td>6.8</td>
<td>Calibration</td>
<td>91</td>
</tr>
<tr>
<td>6.9</td>
<td>Post-processing from previously saved data</td>
<td>91</td>
</tr>
<tr>
<td>6.10</td>
<td>Signal type algorithm</td>
<td>92</td>
</tr>
<tr>
<td>6.11</td>
<td>Main results window</td>
<td>94</td>
</tr>
<tr>
<td>6.12</td>
<td>Results window tools</td>
<td>95</td>
</tr>
<tr>
<td>6.13</td>
<td>Calibration preview window</td>
<td>96</td>
</tr>
<tr>
<td>6.14</td>
<td>Two instances running simultaneously</td>
<td>97</td>
</tr>
<tr>
<td>7.1</td>
<td>Fast and slow axes of PANDA fiber</td>
<td>99</td>
</tr>
<tr>
<td>7.2</td>
<td>PANDA SCOS compared to D-SCOS</td>
<td>100</td>
</tr>
<tr>
<td>7.3</td>
<td>TO-SCOS fabrication</td>
<td>100</td>
</tr>
<tr>
<td>7.4</td>
<td>Air-bubbles in a TO-SCOS</td>
<td>101</td>
</tr>
<tr>
<td>7.5</td>
<td>TOSCOS oil immersion test</td>
<td>102</td>
</tr>
</tbody>
</table>
CHAPTER 1. INTRODUCTION

High voltages are used in a variety of applications such as high voltage pulsed generators for biological cell research [1], plasma generation [2–4], lightning simulation [5], etc. High voltage applications often require generation and measurement of high electric fields, like in defense or nuclear research programs [6–10]. An example is defense against high power microwave (HPM) and electromagnetic pulse (EMP) weapons. These weapons emit short, high-power, high-frequency pulses that couple with conductive lines in electronics, inducing large currents that destroy the electronics. These weapons have the ability to quickly disable any electronics within the range [11–14]. To protect them from these attacks, electronic devices are shielded by enclosing them within a metal box or mesh to block the external field [15]. The efficacy of the shielding is determined by characterizing electric fields near the electronics with minimum perturbation to the measured field and without affecting the electronics and shielding. This kind of field characterization requires the sensor to be non-perturbing, small in sensing area for highly localized field measurement, and directionally sensitive to measure the full three-axis field direction and amplitude [16]. Another example is ignition coil and similar plasma discharge systems. Their effectiveness is partially determined by the duration and voltage of a plasma discharge, also known as a plasma arc. Plasma arcs typically occur on the order of several microseconds and their characterization requires non-perturbing sensors capable of high speed measurements. Finally, there is a need for non-perturbing voltage measurements inside of a cable, for example, for measuring partial discharge inside a cable [17], reading the data in power-line communications [18], etc.

1.1 Sensor selection

The simplest most common method of measuring high voltage is using a voltage divider probe [19]. Figure 1.1 shows the schematic used to measure high voltages (left) and voltage divider probe (right). Resistor divider network scales down a high voltage to a value small enough for
measurement with a voltmeter. Capacitive dividers are also used for high voltage measurements when power lost in resistors as heat becomes a concern. This method is simple and useful in a lot of cases, but it modifies the voltage generator circuitry by altering the overall circuit impedance and capacitance. Consequently, the RC time constant is affected and the discharge measurement is perturbed. Another disadvantage to this method is that the voltage probe is much too large to reach tight spots within complex circuitry.

Electric field sensors are an alternative method for taking voltage measurements, since potential field $\phi(\vec{r})$ and electric field $\vec{E}(\vec{r})$ are merely two alternative ways of describing the same phenomenon. Relationship between the two is given by

$$\vec{E}(\vec{r}) = -\nabla \phi(\vec{r}),$$

(1.1)

where $\vec{r}$ is the location in space. When the geometry of the electrical system is fixed, the electric field magnitude $E$ and voltage are proportional.

One example of a state-of-the-art electric field sensor with high sensitivity is the D-dot sensor, as shown in Figure 1.2. D-dot sensor measures the time derivative of the electric field, which is in turn related to the voltage [20]. However, typical D-dot sensor dimensions are around 40 x 6 x 2 cm, which means it’s field localization is poor and it is too large to place inside many electronic devices. It is also composed of metal, which interferes with the measured electric field and can induce unwanted arcing and perturb the measured electric field.

A slab-coupled optical fiber sensor (SCOS) is an electric field sensor made entirely of dielectric materials, which makes it a good choice for sensing high electric fields since it doesn’t perturb the measured field like metallic sensors do by increasing corona discharge or inducing dielectric breakdown of the system. It has previously been used to measure electric fields within high
power microwave systems [9, 21] and fields up to 100 kV/m within ion traps [22]. Other research studies report using similar optical electric field sensors for measuring electric fields below the dielectric strength of air at low frequencies in the 50–60 Hz range [23, 24] and frequencies up to 1 GHz [25]. An optical fiber sensor based on electro-optic effect of the bismuth germanate (BGO) crystal is reported measuring fields from 100 V/m to 10 MV/m with bandwidth up to 1 GHz [26]. There is a need to extend the capabilities of SCOS technology to measure electric fields beyond air breakdown.

Measuring voltage using an electric field sensor requires an electrode structure with fixed geometry for proper calibration. Because of this, voltage measurement methods commonly imply attaching a capacitively-coupled voltage sensor with metallic parts to the system under test [7, 27–29]. Voltage sensors based on the electro-optic Pockels effect generally use separate metallic electrodes connected to the system under test [30–34]. Measuring voltage in a coaxial cable is even more demanding because it requires compact sensor size. Previously reported optical voltage sensors based on Pockels effect in bismuth-germanate (BGO) show good performance in terms of bandwidth and voltage range, however dimensions of their sensing elements range from 1 cm [35] and 1.5 cm [36] to 12 cm [37], which is inadequate for embedding inside most coaxial cables. Researchers in Asea Brown Boveri Limited Company (ABB) have developed a commercial voltage
SCOS’s are compact enough to fit inside a coaxial cable so that it can use cable’s own conductors as electrodes for the voltage-field conversion and calibration. Figure 1.3 shows the cross-section of an RG-218 coaxial cable with a SCOS placed next to it for size comparison. The small size of the electro-optic crystal on the fiber allows exceptional field localization. The small size, however, does limit the overall SCOS sensitivity. A SCOS is not as sensitive as D-dot sensors, but can measure fields on the order of 100 V/m [39].

The problems initially encountered with the concept of SCOS voltage measurements inside of a coaxial cable are shown in Figure 1.4. Voltages up to 30 kV (red) measure with correct shape, but increasing the voltage further distorts the measurements, as shown with the 36 kV measurement (blue) in Figure 1.4(a). The problem of air bubbles being introduced into the coaxial cable during SCOS insertion is illustrated in Figure 1.4(b). Figure 1.5 illustrates further observed inconsistency with high voltage measurements. The measurements increase non-linearly for linear increase in the applied voltage. Moreover, the measured waveform shape becomes more deformed as voltages increase.
This dissertation introduces new fabrication and measurement methods extending the capabilities of SCOS technology to correctly measure high electric field and high voltage. The improvements in SCOS technology are used as a core technology to build a complete non-perturbing measurement system for high electric field and voltage. Testing the high voltage measurement sys-
Figure 1.6: Damage caused by uncontrolled arcing in high voltage measurement setups.

This kind of work has its own specific challenges like sporadic arcing and high electrical power requirements. Figure 1.6(a) shows a SCOS sensor broken in two after a dielectric breakdown between the electrodes. Figure 1.6(b) illustrates the strength of wayward arcs, which are able to break a thick piece of plexiglass, as shown in Figure 1.6(b), and blow up a power resistor into pieces, as shown in Figure 1.6(c).

1.2 Contributions and dissertation outline

In total, my contributions have been published in 13 journal papers and 14 conference proceedings.
My major contributions have been published in four peer-reviewed journals, three of which I am the first author, and four conference proceedings, three of which I am the first author. My major contributions can be divided in three groups, given as follows:

I developed the non-linear calibration technique enabling SCOS to measure high electric fields:


I developed a system for non-perturbing voltage measurement in a coaxial cable using SCOS:


I developed an interrogation technique that enables edge-filtering fiber-Bragg-grating demodulation in high vibration environment:


My other contributions include: I contributed to the development of the PANDA-SCOS technology (solving together fabrication challenges, changing from TM to TE polarization to improve sensitivity [49], using APC connectors to reduce phase noise, helping with splicing, fabrication, testing and SEM imaging):


I contributed to the development of the push-pull-SCOS technology (discussing challenges, helping with measurements):


I contributed to the development of the antenna-SCOS (helping with the measurements):

I contributed to the development of phase noise reduction method (helping with measurements):


I contributed to the measurement of arc dynamics (taking measurements and high voltage setup):


I contributed by providing ongoing support for a broad array of research applying high-speed full-spectrum fiber Bragg grating interrogation system in structural health monitoring and material properties characterization:


I contributed to writing of the paper for dynamic shape sensing with FBG sensors and helped with taking necessary measurements:


I contributed to the application of fiber-Bragg grating sensor for the purpose of characterizing SCOS strain sensitivity reduction of various sensor packages:


This dissertation can be separated into three parts: Introduction, main contributions and other contributions. Background about the SCOS technology is given in Chapter 2. Chapters 3 - 6 align with the three major parts of the complete measurement system for high voltage and high electric field:

1. SCOS calibration and fabrication improvements required for high electric field measurement are discussed in Chapters 3 and 4;

2. Interrogator system design and build are discussed in Chapter 5; and

3. Software development are discussed in Chapter 6.

Other SCOS-related contributions are given in Chapter 7. My major non-SCOS related contribution involves development of a fiber Bragg grating interrogator system, which is presented in Chapter 8. Summary of the entire dissertation is given in Chapter 9.
SCOS is an electric field sensor created by coupling two waveguides: a polarization-maintaining (PM) fiber and an electro-optic slab waveguide. SCOS relies on the linear electro-optic effect (Pockels effect) as well as evanescent waveguide coupling to detect electric field. Traditionally, D-fibers with an elliptical core are used [68]. However, side-polished Panda fiber has also been used for SCOS fabrication [50, 69]. General principle is the same: close proximity between fiber core and the crystal waveguide allows evanescent coupling between the two. Figure 2.1(a) shows a cross-sectional view of a side-polished PANDA fiber. The slab waveguide is attached to the flat side of the fiber, as shown in Figure 2.1(b).

As light propagates through the fiber, it couples out of the fiber and into the electro-optic crystal slab waveguide. The waveguide used in this research is LiNbO3 lithium-niobate (LN), but other electro-optic materials can be used such as KTP, LiTaO3, or electro-optic polymer [70]. In order for the light to couple out of the fiber via the slab waveguide, the core of the fiber and the slab waveguide must be close enough for evanescent coupling to occur. A short section in the middle of the fiber is stripped of its plastic jacket and then etched in hydrofluoric acid. The part of the cladding is etched away exposing the evanescent field and allowing better optical coupling between the core of the fiber and the crystal. LN slab is placed on top of the flat surface of the fiber, and because of crystals close proximity to the core, light from the core couples into the slab around specific resonant wavelengths, as given by [71, 72]

\[
\lambda_m = \frac{2t}{m} \sqrt{n_0^2 - N_f^2},
\]

(2.1)

where \(\lambda_m\) is the resonant wavelength corresponding to the integer mode \(m\), \(t\) is the slab waveguide thickness, \(n_0\) is the refractive index of the slab waveguide, and \(N_f\) is the effective index of refraction of the fiber mode [21].
Figure 2.1 shows the SCOS transmission spectrum with resonant dips due to light coupling from the fiber into the LN slab as described by Eq. 2.1. Figure 2.2(b) shows a block diagram of a SCOS interrogator, used to record the SCOS transmission spectrum and for measurements. The interrogator consists of a tunable laser (TL) connected to the SCOS input, a photodiode (PD) connected to the SCOS output, converting the transmitted optical power into an electrical current signal, a trans-impedance amplifier (TIA), converting the current into a voltage signal, and an oscilloscope (OSC), capturing and displaying the measured waveform. The spectrum is captured by tuning the laser wavelength in small increments and recording the oscilloscope voltage at each step.

### 2.1 Sensitivity

Lithium niobate is a uniaxial crystal with principal refractive indices $n_x = n_y = 2.211$ and $n_z = 2.138$, $z$ axis being its optic axis. The crystal has Trigonal 3m symmetry with electro-optic
Figure 2.2: Transmitted optical spectrum of a SCOS (a) and SCOS interrogation block diagram (b).

The tensor \( r \) [70]

\[
r = \begin{bmatrix} 0 & -r_{22} & r_{13} \\ 0 & r_{22} & r_{13} \\ 0 & 0 & r_{33} \\ 0 & r_{51} & 0 \\ r_{51} & 0 & 0 \\ -r_{22} & 0 & 0 \end{bmatrix}
\]

(2.2)

where \( r_{13} = 8.6 \text{ pm/V} \), \( r_{33} = 31 \text{ pm/V} \), \( r_{22} = 3.4 \text{ pm/V} \) and \( r_{51} = 28 \text{ pm/V} \). This tensor describes the change to the slab’s refractive index when it is exposed to electric field in accordance with the
Figure 2.3: Resonance edge before (solid) and after (dotted) the shift causing a change in the interrogator voltage.

\[ \Delta \left( \frac{1}{n^2} \right)_i = \sum_j r_{ij}E_j, \quad (2.3) \]

where \( r_{ij} \) is the element from the refractive index tensor \( r \) and \( E_j \) is one of three components of the electric field \( E \). For tangential component of the electric field and x-cut slab the change in refractive index \( n_0 \) is

\[ \Delta n_0 = \frac{1}{2} n_0^3 r_{33} E. \quad (2.4) \]

Since refractive index \( n_0 \) is proportional to the electric field applied across the slab, the entire transmission spectrum shifts as a function of the applied field [68]. Figure 2.3 shows the resonance edge before and after the spectral shift. If the laser light transmitted through the SCOS has a wavelength that falls on the resonance edge, the spectral shift results in a change in the interrogator voltage \( \Delta V \).

Figure 2.3 shows that the spectral shift \( \Delta \lambda \), observed at a fixed wavelength \( \lambda_0 \) causes the interrogator voltage to change from \( V_0 \) to \( V_1 \). The same spectral shift can be interpreted as a shift in the operating laser wavelength on a fixed spectrum edge.
The shift in wavelength $\Delta \lambda$ divided by the change in the applied electric field $\Delta E_{\text{ext}}$ is the spectral shift rate. The detected voltage change $\Delta V$ is, therefore, proportional to the electric field, whereby the proportionality constant depends on both the resonance slope and the spectral shift rate. The same is true for the transmitted optical power.

The power transmitted through the SCOS is given by [22]

$$ P(t) = P_0 + \left( \frac{\partial P}{\partial \lambda} \right) \left( \frac{\partial \lambda}{\partial E} \right) E_{\text{ext}}(t), \quad (2.5) $$

where $P_0$ is the power transmitted through the SCOS when no electric field is applied, $E_{\text{ext}}(t)$ is the external electric field applied across the SCOS, $\partial P/\partial \lambda$ is the resonance slope in terms of optical power and $\partial \lambda/\partial E$ is the spectral shift rate. Similarly, the interrogator voltage $V(t)$ can be expressed as

$$ V(t) = V_0 + \left( \frac{\partial V}{\partial \lambda} \right) \left( \frac{\partial \lambda}{\partial E} \right) E_{\text{ext}}(t), \quad (2.6) $$

where $V_0$ is the voltage DC offset corresponding to the optical power level $P_0$ and $\partial V/\partial \lambda$ is the resonance slope in terms of voltage. The two terms in parenthesis are the SCOS sensitivity terms, which determine the rate at which the interrogator voltage changes with the change in the applied electric field.

### 2.2 Linear calibration

The SCOS sensitivity terms in Eq.2.6 are approximated by applying a known external electric field $E_{\text{ext, cal}}$ and measuring the corresponding interrogator voltage $V_{\text{cal}}$ as given by

$$ \left( \frac{\partial V}{\partial \lambda} \right) \left( \frac{\partial \lambda}{\partial E} \right) \approx \frac{V_{\text{cal}}}{E_{\text{ext, cal}}}. \quad (2.7) $$

Using the approximation in Eq.2.7, the value of any applied external electric field $E_{\text{ext}}(t)$ can be found from the measured interrogator voltage $V(t)$ as given by

$$ E_{\text{ext}}(t) \approx \frac{E_{\text{cal}}}{V_{\text{cal}}} (V(t) - V_0). \quad (2.8) $$
Since voltage is proportional to electric field in a system with fixed geometry, a SCOS can also be used for voltage measurements. The interrogator voltage depends on the applied external voltage $V_{\text{app}}$ similar to Eq.2.6, only with the sensitivity term scaled by a constant, as given by

$$V(t) = V_0 + C \left( \frac{\partial V}{\partial \lambda} \right) \left( \frac{\partial \lambda}{\partial E} \right) V_{\text{ext}}(t),$$

(2.9)

where $C$ is the proportionality constant between the voltage and electric field. The sensitivity term in Eq.2.9 is approximated by applying a known external voltage $V_{\text{ext}, \text{cal}}$ and measuring the corresponding interrogator voltage $V_{\text{cal}}$, as given by

$$C \left( \frac{\partial V}{\partial \lambda} \right) \left( \frac{\partial \lambda}{\partial E} \right) \approx \frac{V_{\text{cal}}}{V_{\text{ext}, \text{cal}}}. $$

(2.10)

Using the approximation in Eq.2.10, the value of the measured external voltage $V_{\text{ext}}(t)$ can be found from the measured interrogator voltage $V(t)$ as given by

$$V_{\text{ext}}(t) \approx \frac{V_{\text{ext}, \text{cal}}}{V_{\text{cal}}} (V(t) - V_0).$$

(2.11)

The conversion of the measured interrogator voltage into external electric field, as described by Eq.2.8, and into external voltage, as described by Eq.2.11, is called linear calibration. Figure 2.4
illustrates how linear calibration method approximates the resonance slope with a constant, which corresponds to approximating the resonance edge with a tangent at the operating wavelength. The tangent matches the resonance edge well near the operating wavelength, which means that linear interrogation is a good approximation for small spectral shifts, i.e. for small electric fields. The mismatch between the tangent and the resonance edge for large spectral shifts in Figure 2.4 means that linear calibration can only measure electric fields below a certain threshold with a preset maximum measurement error. Furthermore, linear calibration only accounts for one side of the resonance and therefore cannot measure large fields with spectral shifts that span to the other side.
CHAPTER 3. NONLINEAR CALIBRATION FOR MEASURING HIGH ELECTRIC FIELDS WITH A SCOS

This chapter describes the development and testing of SCOS’s for high bandwidth measurement of electric fields below and above air breakdown. SCOS technology is adapted for high electric field measurement by changing the standard linear calibration method into nonlinear calibration method and by ensuring that all materials used in fabrication have a high dielectric breakdown point, specifically that no air-bubbles are introduced in the epoxy during fabrication. The nonlinear calibration uses SCOS transmission spectrum to map voltage to spectral wavelength shift and then to correlate the wavelength shift to the measured electric field. The system is demonstrated by measuring electric fields up to 18 MV/m.

3.1 High field adaptations

Measuring high electric fields requires the use of sensor materials that do not break down under the applied electrical field. Otherwise, the interrogator voltage no longer correlates with the electric field. In addition, high electric fields invalidate the linear calibration and a nonlinear method is required. Nonlinear calibration method requires the spectral shift rate as referred to in Eq.2.5 to be constant with electrical field and wavelength, which makes the sensitivity dependent only on the spectrum.

3.1.1 Materials

The SCOS must be made of materials whose dielectric strengths exceed the measured field. Figure 3.1 shows a block diagram of SCOS and its materials and dimensions. The main SCOS parts are the slab, made of lithium-niobate (LN), and the optical fiber, made of fused silica. Other materials that make up the SCOS are PLA plastic or ceramic, used for the trough, and 5-min and
low-index (LS-6140 from Nusil) epoxies used in the packaging process. Dimension of the LN slab in the direction of the optics axis is 0.3 mm, which is also the spatial resolution of the sensor.

Researchers performed periodic poling of LN using 100 µs electric field pulses with 24 MV/m magnitude [73], providing evidence that electric field inside LN can safely reach this value without breaking down. Here, the electric field is not measured directly with an optical sensor, but rather estimated based on the known voltage applied across the slab with a known thickness. Fused silica has dielectric strength of 30–50 MV/m, varying between manufacturers, and relative permittivity of 3.8 at 1 MHz. PLA plastic has a dielectric strength exceeding 500 MV/m and relative permittivity of 2.7 at room temperature [74]. The low-index epoxy LS-6140 from Nusil has a dielectric strength of 20.5 MV/m and 5-minute epoxy by Devcon has dielectric strength of 19.2 MV/m according to specification sheets. Epoxies commonly have relative permittivity around 6 in the 1 kHz–1 MHz frequency range. Based on the listed material dielectric strengths, the SCOS can safely measure electric fields up to the lowest of the listed values, which is 19 MV/m. The SCOS packaging
materials limit the maximum electric field. Changing the packaging materials can increase the maximum electric field to at least 24 MV/m.

### 3.1.2 Invariance of the spectral shift rate

At high electric fields, higher order nonlinear effects become observable. Pockels and Kerr effects are analyzed in the spectral shift rate by taking the derivative of Eq. 2.1 in terms of electric field $E$, as given by

$$\frac{d\lambda}{dE} = \frac{\lambda n_0}{n_0^2 - N_f^2} \frac{\partial n_0}{\partial E} - \frac{\lambda N_f}{n_0^2 - N_f^2} \frac{\partial N_f}{\partial E},$$

where the first term describes electro-optic effects in the LN slab and the second term the electro-optic effects in the optical fiber. Eq. 3.1 is simplified as

$$\frac{d\lambda}{dE} = \frac{\lambda n_2}{n_0^2 - N_f^2} \left( n_0 \frac{\partial n_0}{\partial E} - N_f \frac{\partial N_f}{\partial E} \right).$$

Both Pockels and Kerr effects exist in the LN slab, so the first term in the parenthesis in Eq. 3.2 is written as

$$n_0 \frac{\partial n_0}{\partial E} = \frac{1}{2} n_0^3 r_{33} \gamma + n_2 \frac{n_0^2 \gamma^2 E_{\text{ext}}}{\eta_0},$$

where $\eta_0$ is the impedance of air, $n_2$ is the nonlinear index of refraction in the slab, $r_{33}$ is the effective electro-optic coefficient for TE polarization in the LN slab [49]. Electric field inside the slab $E$ is proportional to the external electric field $E_{\text{ext}}$, as given by

$$\gamma = \frac{E}{E_{\text{ext}}},$$

where $\gamma$ depends on the given material properties and geometry [49].

Substituting in Eq. 3.3 the values for nonlinear index of refraction $n_2 = 5.3 \times 10^{-15} \text{ cm}^2/\text{W}$ [75] for LN, $\gamma = 0.25$ [49], $n_0 = 2.155$ for TE polarization and $r_{33} = 34 \text{ pm/V}$ [76] gives a value on the order of $100 \text{ pm/(MV/m)}$ for the linear term and a value on the order of $0.01 \text{ pm/(MV/m)}$ for the nonlinear term at $\lambda = 1550 \text{ nm}$ and $E_{\text{ext}} = 19 \text{ MV/m}$. The contribution from the nonlinear electro-optic effect in LN is four orders of magnitude smaller than the linear electro-optic effect and therefore can be neglected.
Fused silica exhibits only second order electro-optic effect and the second term in the parenthesis in Eq. 3.2 can be written as

\[ N_i \frac{\partial N_i}{\partial E} = n_{2f} \frac{N_i \gamma_f E_{ext}}{\eta_0}, \]  

(3.5)

where \( n_{2f} \) is the nonlinear index of refraction in the fiber and \( \gamma_f \) is the proportionality constant between the field inside the fiber \( E \) and external electrical field \( E_{ext} \). Substituting the value for nonlinear index for fused silica \( n_{2f} = 3.2 \times 10^{-16} \text{ cm}^2/\text{W} \) [76] in Eq. 3.5 and keeping the other parameters similar as in the case of LN, gives the value on the order of 1 fm/(MV/m), five orders of magnitude smaller than the dominant linear term in LN, so the nonlinear term for fused silica can also be neglected. Eq. 3.2 is, therefore, simplified as

\[ \frac{d\lambda}{dE} = \frac{\lambda n_0^3 r_{33} \gamma}{2(n_0^2 - N_f^2)}. \]  

(3.6)

Eq. 3.6 does not depend on electric field, meaning that spectral shift rate can be considered invariant with electric field for field values below \( E = 19 \text{ MV/m} \).

Besides invariance with electric field, the spectrum shift also needs to be invariant in terms of wavelength. According to Eq. 3.6 the spectral shift rate and wavelength are proportional. This means their relative changes are equal. Changing the wavelength \( \lambda \) by 5 nm, the spectral width of a single SCOS resonance at the operating wavelength of 1550 nm results in 0.3% change of the spectral shift rate, which is negligible. It can, therefore, be assumed that spectral shift rate is constant with wavelength for spectral shifts on the order of SCOS resonance width. This allows the use of edge filtering interrogation on any resonant valley as shown in Figure 2.2(a) and described by Eq. 2.1 without affecting measurement sensitivity. Moreover, because the spectral shift rate \( \partial \lambda / \partial E \) can be considered constant in terms of electric field and wavelength change, the only part of the total SCOS sensitivity in Eq. 2.6 that depends on electric field and wavelength is the spectrum slope. Sensitivity is, therefore, maximized by maximizing the resonance slope, \( \partial V / \partial \lambda \), that is by tuning the operating wavelength to the steepest point on the spectrum.
3.1.3 Nonlinear calibration

Nonlinear calibration maps the interrogator voltage change to a spectral shift, which, in turn, is proportional to the applied electric field. This is illustrated in Figure 3.2. The spectrum edge is shown before and after the spectral shift with the corresponding voltage and wavelength values. Looking only at the spectrum before the shift (solid), the change in interrogator voltage from $V_0$ to $V_1$ maps to the wavelength shift from $\lambda_0$ to $\lambda_1$, for a total spectral shift of $\Delta \lambda$. It can, therefore, be said that the effects of a spectral shift $\Delta \lambda$ at a fixed operating wavelength $\lambda_0$ are equivalent to moving the laser by $\Delta \lambda$ in the opposite direction along a fixed spectrum. Because of this equivalency, the applied electric fields in further text are treated as causing a shift in the operating wavelength along a fixed SCOS spectrum, instead of shifting the spectrum observed at a fixed operating wavelength.

High electric field measurements requires the use of nonlinear calibration since linear calibration breaks down for high electric fields. The nonlinear calibration measurement process consists of three stages: (1) measuring the SCOS spectrum, (2) applying a known electric field to measure spectral shift rate and (3) mapping the interrogator voltage to electric field values using the spectral shift rate and SCOS spectrum.
First, the SCOS transmission spectrum is measured by varying the laser wavelength in the interrogator setup shown in Figure 2.2(b) and recording the corresponding interrogator voltage at each wavelength increment. The operating wavelength is then set at the steepest point on the spectrum edge, the location of the maximum spectrum slope, in order to maximize the sensor sensitivity. Figure 3.3 shows the SCOS spectrum and its slope with the location of the operating wavelength and the corresponding interrogator voltage at the point of maximum slope. The spectral slope reaches high values only in a very narrow range around the operating wavelength and drops rapidly when moving away from the operating wavelength in either direction.

Second, the system measures the interrogator voltage while applying a known electric field. Figure 3.4 shows that the electric field generation circuit is made up of a voltage source $V_S$ applied to parallel plate electrodes separated by a distance $d$, creating electric field $E = V_S/d$ between the plates. The SCOS is placed between the plates with its sensitive axis aligned with the field lines.

When the known electric field $E_{\text{cal}}$ is applied across the SCOS, the interrogator produces the voltage $V_{\text{cal}}$. The interrogator voltage $V_{\text{cal}}$ is mapped into wavelength shift $\lambda_{\text{cal}}$ using the measured spectrum. The spectral shift rate $\partial \lambda / \partial E$ is then calculated as given by

$$
\frac{\partial \lambda}{\partial E} = \frac{\lambda_{\text{cal}}}{E_{\text{cal}}}.
$$

(3.7)
The spectral shift rate is assumed to be constant for all values of measured electric field and allows a simple way to convert any wavelength shift $\Delta \lambda$ into the corresponding electric field $E$ as in

$$E = \frac{E_{\text{cal}}}{\lambda_{\text{cal}}} \Delta \lambda. \quad (3.8)$$

The third step in nonlinear calibration involves applying an unknown electrical field waveform $E(t)$ across the SCOS and measuring the corresponding interrogator voltage waveform $V(t)$. Every point in the interrogator voltage waveform $V(t)$ is mapped to a wavelength waveform $\lambda(t)$ using the measured spectrum as expressed by

$$V(t) \overset{\text{spectrum}}{\longrightarrow} \lambda(t). \quad (3.9)$$

The wavelength shift is obtained by subtracting the operating wavelength $\lambda_0$ from $\lambda(t)$ and, based on Eq.3.8, the unknown electric field $E(t)$ is calculated as [40,41]

$$E(t) = \frac{E_{\text{cal}}}{\lambda_{\text{cal}}} (\lambda(t) - \lambda_0). \quad (3.10)$$

Although the nonlinear calibration method is harder to implement than linear calibration, it correctly calibrates both low and high $E$ fields, eliminating the errors produced by the linear calibration.
3.2 High field measurements

Figure 3.5 shows the schematic of a measurement system comprised of a 17.5 nF capacitor \( C \), charged by a high voltage supply \( V \) through the switch SW1, and connected in parallel via a switch SW2 to a parallel-plate electrode structure containing a SCOS \( C_{SCOS} \) and four 50 k\( \Omega \) resistors \( R \) in parallel, with the equivalent resistance of 12.5 k\( \Omega \). First, \( C \) is charged by closing SW1 and opening SW2. Then, SW1 is opened and an exponential electric field pulse is created across \( C_{SCOS} \) by shorting SW2 at time \( t_0 \). The electric field pulse corresponds to the discharge of \( C \) into the four parallel resistors \( R \), which are connected in parallel to increase the equivalent maximum power rating.

Figure 3.6 shows the simulated electrical field magnitude between electrodes with 7 mm spacing and with a SCOS modeled as a block of plastic laying on the high potential electrode. Simulation is done using the software Maxwell 2D. The two subfigures show simulated field with 10 kV and 100 kV applied across the electrodes. After inserting the SCOS in the field, the field
is not as homogeneous anymore, but as long as the geometry is unchanged, changing only the voltage proportionally changes the electric field. Figures 3.6(a) and 3.6(b) show that for both voltage values, the field distribution remains identical, with only the field magnitude changing proportionally. The relationship between the field and voltage can be calculated from the simulated field values as

\[ E = \frac{V}{6.7 \times 10^{-3}}, \]  

(3.11)

where the constant \( 6.7 \times 10^{-3} \) m is obtained by dividing the applied voltage and the simulated electric field magnitude at the point in the middle between the electrodes. The obtained proportionality constant is very close to the actual distance between the electrodes.

Figure 3.7 shows the parallel plate structure with the SCOS inserted between the plates. The optics axis is oriented normal to the plates so that the positive electrical field gives negative interrogator voltage change. The width of the SCOS packaging is less than 3 mm. The electrodes are two copper discs with a 70 mm diameter and 7 mm separation. The whole setup is immersed in transformer oil for electrical insulation, allowing the generated fields to reach values above the dielectric strength of air, 3 MV/m, without arcing between the plates. The capacitance of the parallel-plate structure in transformer oil is around \( C_{\text{SCOS}} \approx 11 \) pF. With this setup it is possible to generate electric fields up to 18 MV/m by applying voltages up to 125 kV without breaking
down the transformer oil. The generated electric field is a pulse that starts at time $t_0$ and decays exponentially with a time constant $\tau = 17.5 \text{nF} 12.5 \text{k}\Omega = 219\mu\text{s}$.

As explained in the previous section the nonlinear calibration method requires the measurement of both the SCOS spectrum and the spectral shift rate. Figure 3.8 shows the SCOS spectrum measured using a tunable laser with 1 pm wavelength increments. The same electronics are used for both spectrum and electric field measurements to ensure accurate calibration.

The calibration measurement is performed with a 6.3 kV$_{pp}$, 60 Hz sinusoid voltage, corresponding to a calibration signal of $E_{\text{cal}} = 0.9 \text{ MV/m}$. The interrogator measured a $V_{\text{cal}} = 28 \text{ mV}_{pp}$ signal. Figure 3.9(a) shows the measured calibration waveform. Mapping the calibration voltage to the spectrum gives $\lambda_{\text{cal}} = 45 \text{ pm}$. Figure 3.9(b) illustrates this mapping process with a spectrum edge and lines showing the extent of the mapped calibration voltages and wavelengths. Here, the voltages are mapped onto a fixed spectrum to obtain the wavelength shifts instead of observing a shifting spectrum with fixed wavelength as previously explained with Figure 3.2. According to Eq.3.7, the measured spectral shift sensitivity is

$$\frac{\partial \lambda}{\partial E} = \frac{\lambda_{\text{cal}}}{E_{\text{cal}}} = \frac{45 \text{ pm}}{0.9 \text{MV/m}} = 50 \frac{\text{pm}}{\text{MV/m}}.$$  \hspace{1cm} (3.12)

Figure 3.10 shows three characteristic electric field pulse measurements calibrated using linear and nonlinear calibration method with the corresponding wavelength shifts along the SCOS
Figure 3.9: Calibration measurement shown as a voltage waveform (a), which is then mapped into wavelength using the measured spectrum (b). The operating wavelength and the corresponding voltage are indicated with dotted vertical and horizontal lines, respectively. The extent of the calibration voltages and mapped wavelengths are indicated with solid horizontal and vertical lines, respectively.

The rise times of the measured pulses are on the order of 100 ns and the fall time constants range from 180 µs to 240 µs.

Figure 3.10(a) shows the results for 1.5 MV/m electric field. Linear and nonlinear calibration give very similar results for small field like this. However, Figure 3.10(c) shows that with a peak field amplitude of 8 MV/m there is obvious discrepancy between the two calibration methods. Linear calibration predicts a peak field of 6.5 MV/m because it interprets all the field magnitudes with the same high sensitivity as used for lower fields. For the particular SCOS used in the measurements, the sensitivity starts significantly dropping for fields above 5 MV/m. Nonlinear calibration, however, predicts the 8 MV/m peak field more correctly since it interprets different field magnitudes with variable sensitivity.

Figure 3.10(e) shows large discrepancy between the two calibration methods for the electric field of 18 MV/m. Because the resonance slope becomes zero at the bottom of the resonance, the SCOS sensitivity drops to zero in this region (see Figure 3.3). For 18 MV/m the wavelength shift is large enough to make the operating wavelength cross over the zero-sensitivity region to the opposite side of the resonance. This is evident in Figure 3.10(f). Because linear calibration
sees only one side of the resonance, it results in a non-exponential, rounded pulse shape with an increased time constant. Nonlinear calibration, on the other hand, applies to both sides of the resonance and predicts 17.3 MV/m peak field with time constant of 240 µs, which is very close to the expected 220 µs. The discontinuity around the point t = 40 µs is due to the transition across the zero-sensitivity region at the resonance bottom. It also falls right on top of the rounded peak obtained with linear calibration.

Measurements are made with voltages ranging from 11 kV to 125 kV in increments of around 11 kV. Because voltage is applied across a parallel plate structure with 7 mm gap, the generated fields approximately range from 1.5 to 18 MV/m in increments of 1.5 MV/m. Different fields could also be achieved by keeping the voltage constant and tuning the distance between electrodes. Measurement results should be the same, however. In this case the distance is kept constant and voltage is changed for better voltage control.

Figure 3.11 shows the peak values of the whole set of measurements as a function of the actual applied electric fields calibrated with nonlinear and linear calibration method. The actual electric field is calculated from the known applied voltage and the measured gap distance between the parallel plates. Expected results with an ideal calibration fall on the dotted line. It can be seen that nonlinear calibration results fall very close to the ideal line, while the linear calibration method exceeds 10% relative error compared to nonlinear calibration for electric fields above 5 MV/m. The increasing divide between the linear and nonlinear calibration curves corresponds to the drop in slope sensitivity illustrated by the increasing mismatch between the spectrum edge and its tangent illustrated in Figure 2.4. The saturation of the linear calibration at 9 MV/m for large applied field values results from the operating wavelength reaching the bottom of the resonance dip and crossing over to the opposite side of the resonance.

The dynamic range is the ratio of the maximum and minimum measurable electric field. Maximum measured electric field is 18 MV/m and previous measurements resulted in minimum detectable field of 15.8 V/m [68]. This gives the maximum dynamic range exceeding 1,000,000, or 60 dB. It is possible to decrease the interaction length between the slab and D-fiber during the SCOS fabrication process, resulting in sensitivity reduction. The advantage of this is the increased range for linear operation, but the drawback is a higher minimum field, downgrading the maximum dynamic range.
3.3 Summary

Application of the SCOS technology in non-perturbing, directional and localized measurements of high electric fields is described. Measuring high electric fields requires the use of materials with high dielectric strength and a nonlinear calibration method. Based on the dielectric strengths of materials used in the SCOS fabrication, a SCOS is safe to use for electric fields up to 19 MV/m. Successful measurements of electric fields up to 18 MV/m are reported.

Nonlinear calibration method is explained and compared with the standard linear calibration. Measuring electric fields beyond a sensor-specific threshold level (presented measurements have a threshold of about 5 MV/m) requires nonlinear calibration. Nonlinear calibration accurately interprets measurements of both small and high fields with variable sensitivity, while linear calibration works accurately only for fields below the threshold.

The two calibration methods are compared by applying them to the same set of electric field pulse measurements ranging from 1.5 MV/m to 18 MV/m, with rise times on the order of 100 ns and time constants of 220 μs. The nonlinear calibration gives more accurate predictions for all measured electric field values, while linear calibration becomes increasingly inaccurate for fields above the 5 MV/m threshold and saturates for fields above 9 MV/m. For a SCOS with 50 pm/(MV/m) shift sensitivity and 4.5 nm wide resonance, like the one used for measurements, nonlinear calibration could theoretically work for calibration of electric fields up to 90 MV/m, limited only by the dielectric strength of the used materials.
Figure 3.10: Electric field measurements using nonlinear (solid) and linear (dashed) calibration methods for electric field amplitudes of (a) 1.5 MV/m, (c) 8 MV/m, and (e) 18 MV/m. Next to each measurement is the corresponding wavelength shift indicated with a solid bold line on the SCOS spectrum (b, d and f). The operating wavelength $\lambda_0$ and the corresponding voltage $V_0$ are indicated with dotted vertical and horizontal lines, respectively.
Figure 3.11: Peak values of the calibrated electric field measurements using nonlinear (solid, diamond) and linear (dashed, asterisk) calibration.
CHAPTER 4. NON-PERTURBING VOLTAGE MEASUREMENTS IN A COAXIAL CABLE

Measuring coaxial cable voltage using a SCOS is possible by exploiting the proportionality between electric field and voltage in a fixed cable structure [42–45]. This proportionality allows a SCOS to be calibrated to a known cable voltage. A SCOS is small enough to be embedded inside the coaxial cable dielectric between the inner and outer conductor. This is illustrated in Figure 4.1. The SCOS directional sensitivity requires its optical axis to be oriented in the radial direction to align with the radial electric field lines of the coaxial structure.

Commonly, voltage measurement methods using electric field sensors involve adding metallic electrodes to the system under test in order to convert voltage into electric field [7, 27–34]. Adding electrodes changes the capacitance of the system and inevitably perturbs the system under test. Advantage of embedding the all-dielectric SCOS in the dielectric of the cable is that no metallic parts are added to the system, reducing the perturbation to the coaxial cable.

4.1 Potential system perturbation

Inserting a SCOS into the coaxial cable can potentially perturb the system under test in two ways. First, a SCOS can cause voltage reflections at the point of insertion. Secondly, the dielectric strength of the cable can be compromised by the materials introduced into the cable.

4.1.1 Voltage reflections in the coaxial cable

Inserting a SCOS in a coaxial cable changes the characteristic impedance of the cable at the point of insertion and forms an impedance mismatch, resulting in voltage reflections. Voltage reflections in systems are commonly measured using time-domain reflectometry technique. The magnitude of voltage reflections can also be estimated with a computer simulation of the line’s S11 parameter.
Figure 4.1: Cross section of a coaxial cable with a SCOS inserted in it.

Figure 4.2: Setup for measuring voltage reflections using the time-domain reflectometry technique.

**Measuring reflections using time-domain reflectometry**

Time-domain reflectometry (TDR) is a technique for measuring voltage reflections in a transmission line by sending a voltage pulse down the transmission line and observing the voltage waveform at a characteristic point along the line. The TDR setup used for measuring reflections off of a 3 mm hole drilled through the coaxial cable is shown in Figure 4.2. Voltage waveform is observed with a SCOS inserted in the coaxial cable about 1 m from the generator side of the cable. Details of the SCOS insertion into a coaxial cable are discussed later in Section 4.2. The SCOS signal is interrogated with a tunable laser (TL) connected to the SCOS input and a photodiode (PD) and oscilloscope (OSC) at the SCOS output. The photodiode bandwidth is 1 GHz and oscilloscope
bandwidth is 2 GHz with the maximum sample rate of 10 GS/s. Reflections are caused by a 3 mm hole drilled in the middle of the 11 m long cable about 5.8 m away from the inserted SCOS. A 10 kV voltage pulse is generated with a high voltage generator V and closing the switch SW starts the propagation of the voltage pulse down the 11 m long RG-218 coaxial cable. The terminal end of the coaxial cable is left open in order to act as a point of total reflection.

Pulse propagation times are given in parenthesis for each cable section indicated in Figure 4.2. Propagation times are calculated from the distances and the propagation speed \( v \) of the electric perturbation in a coaxial cable, as given by

\[
v = \frac{c}{\sqrt{\varepsilon_{rPE}}},
\]

where \( c \) is the speed of light in vacuum, and \( \varepsilon_{rPE} \) is the relative permittivity of polyethylene, with a value of \( \varepsilon_{rPE} = 2.25 \).

The voltage waveform is measured using the inserted SCOS while the coaxial cable is excited with a 10 kV pulse. Figure 4.3(a) shows five repeated measurements to show the measurement consistency. The consistency indicates that the measured waveform has a deterministic nature and does not represent noise in the system. This means that the waveform can be analyzed in search for the reflections caused by the hole drilled in the middle of the cable. It should be noted that the waveforms converge towards the 10 kV levels as expected, which is indicated by the horizontal dotted lines.

Figure 4.3(b) shows a zoomed-in TDR waveform measurement. Only first few reflections are shown since the reflection off of the cable hole should be observable in the initial reflection. From the calculated propagation times in Figure 4.2 it can be concluded that the time required for the voltage pulse propagation from the SCOS to the hole and back is \( 2 \times 23 \text{ ns} = 46 \text{ ns} \) and the time required for propagation from the SCOS to the end of the cable and back is \( 101 \text{ ns} = 2 \times 50.5 \text{ ns} \). The impulse rise time \( \tau_{\text{rise}} \) is about 50 ns. The voltage reflection caused by the hole should be observed as a rise in voltage between \( t = 46 \text{ ns} \) to \( t = 92 \text{ ns} \), but since the waveform is flat with a slight slight negative slope in this region, no significant observable voltage reflection can be observed caused by the reflection off of the hole. The reflection off of the cable’s open end can clearly be observed at \( t = 101 \text{ ns} \).
Since time-domain reflectometry measurements are not able to reveal any significant reflections caused by a hole in the coaxial cable, a reflection caused by a hole with the SCOS is expected to be smaller or of similar magnitude. Computer simulation is used to estimate the magnitude of this reflection more precisely.

**Computer simulation of the S11 parameter**

Electrical reflections in a system are commonly described by scattering parameters, or S-parameters. S-parameters are the elements of the S-matrix connecting the incident and reflected power waves at the system ports. For a 2-port network, as shown in Figure 4.4, the S-matrix is given as

\[
\begin{bmatrix}
V_1^r \\
V_2^r
\end{bmatrix} =
\begin{bmatrix}
S_{11} & S_{12} \\
S_{21} & S_{22}
\end{bmatrix}
\begin{bmatrix}
V_1^i \\
V_2^i
\end{bmatrix},
\]

(4.2)

where \(V_1^i\) and \(V_2^i\) are the incident and \(V_1^r\) and \(V_2^r\) are the reflected voltage waves on ports 1 and 2, respectively. From Eq.4.2, \(S_{11}\) parameter is equal to the ratio of the reflected voltage on port 1 \(V_1^r\) and...
and the incident voltage $V^i_1$ on port 2, when the incident voltage wave on port 2 $V^i_2$ is zero, as in

$$S_{11} = \frac{V^r_1}{V^i_1 \bigg|_{V^i_2=0}}.$$  \tag{4.3}

To determine the magnitude of voltage reflections caused by the SCOS-insertion in the RG-218 coax-cable, S11 parameter is simulated using ANSYS Maxwell software. S11 simulation is performed on a 50 cm long RG-218 coaxial cable with and without a SCOS and compared to S11 parameter simulated in a cable with a comparably sized cut-out. Cable length of 50 cm is arbitrarily chosen to be significantly larger than the size of the SCOS. The simulated models are shown in Figure 4.5. The simulation of the cut cable is used to obtain the upper bound on S11 by removing most of the dielectric and shielding on a cable section matched in size to the SCOS hole. Because cutting the cable is by nature more intrusive than drilling a small hole, it is assumed that the impedance mismatch of the the cut model surpasses the impedance mismatch of the model with a SCOS. The simulation of the cable without a SCOS is used to obtain the lower bound on S11 because in this case there is no impedance mismatch.
Figure 4.6: S11 parameter simulated for a cut cable (dashed), original unmodified cable (dotted) and cable with an inserted SCOS (solid).

The S11 simulation results in the frequency range from 0.1 MHz to 3 GHz are shown in Figure 4.6 [42]. The cut cable model predicts significant reflections with S11 parameter exceeding $-10$ dB for frequencies above 1 GHz. The models of the coaxial cable with and without a SCOS have comparably small reflections with S11 peak values below $-30$ dB. This means that inserting a SCOS into the coaxial cable is not expected to introduce significant voltage reflections and impedance mismatch.

**S11 parameter dependence on impedance mismatch and the SCOS hole size**

SCOS can be inserted into any cable for the purpose of voltage measurements. In a general case, perturbations are estimated by simulating the S11 parameter using a lumped element transmission-line model. In this simulation, the section of the cable with an inserted SCOS is modeled as a transmission line with a mismatched characteristic impedance $Z_1$ of length $L$, equal to the size of the SCOS hole, inserted in between two transmission lines with characteristic impedance $Z_0$. Figure 4.7(a) illustrates this. Red line is the incident voltage pulse rising edge. Blue dashed lines are the reflections off of the first and second impedance mismatch on the path of the voltage pulse. The first edge causes a negative reflection for $Z_1 < Z_0$, while the second edge causes opposite reflection. These two reflections are delayed by a small amount of time that it takes for the pulse to travel across the mismatched line and the two reflections superimpose giving a total reflection.
shown in the solid blue line. The blue arrow indicates that the reflection is traveling towards the voltage source $V$, and the red arrow indicates that the incident voltage is traveling away from the source. These reflections happen even to the reflected pulses, so a more realistic model includes multiple reflections. Simulation is performed for 100 multiple reflections and a range of length $L$ and mismatched impedance $Z_1$ values, with the assumption that $Z_0 = 50\,\Omega$. The $S_{11}$ parameter is calculated as the ratio of the amplitude of the total reflected voltage pulse and the incident pulse, as given in Eq.4.3. Results are presented in Figure 4.7(b). In the extreme case of $48\,\Omega$ mismatched characteristic impedance $Z_1$ and the SCOS hole size of $L = 10\,\text{mm}$, the simulated $S_{11}$ parameter increases to a maximum value of $-36.4\,\text{dB}$. This suggests that a SCOS can be inserted in a variety of different cables without the concern of perturbing the voltage through large voltage reflections.

The $S_{11}$ parameter is simulated for a $4\,\text{mm}$ SCOS hole in the RG-218 coaxial cable using the lumped element transmission-line model. First, the characteristic impedance value $Z_1 = 49.892\,\Omega$ for a lumped element with an inserted SCOS, as shown in Figure 4.8, is obtained through a simulation using HFSS®. Using these values and assuming $Z_0 = 50\,\Omega$, multiple reflection simulation is performed using MATLAB® and value $S_{11} = -52.72\,\text{dB}$ is obtained. This value verifies the simulation method since it is roughly in accordance with the $S_{11}$ simulation shown in Figure 4.6.
4.1.2 Dielectric breakdown

To estimate the voltage range that causes the charge buildup effect, Eq. 4.4 is used to calculate the field magnitude in a coaxial structure

\[ E(r_n) = \frac{V}{(a + r_n(b - a)) \ln(b/a)}, \]  

(4.4)

where \( E(r_n) \) is the magnitude of electrical field at a normalized radial distance \( r_n = (r - a)/(b - a) \) between the inner and outer conductor, measured at a distance \( r \) from the inner conductor, \( a \) and \( b \) are the radii of the inner and outer conductors, respectively, and \( V \) is the applied voltage. The formula shows that electric field in a coaxial cable depends only on geometry and voltage, but not the material properties. The formula is valid when the material between the conductors is
Figure 4.10: Damage to the cable insulation by drilling the SCOS-hole too close to the inner conductor.

homogenous and it allows, therefore, only the calculation of fields in the coaxial cable before the SCOS insertion, but not afterwards when the material becomes heterogeneous. The formula is still useful to demonstrate the field attenuation dependence as $1/r$ and approximate the threshold breakdown voltages. Dielectric breakdown of air is reached at $r_n = 0$ and 0.5 for voltages of $V = 9.3$ kV and $V = 20.8$ kV, respectively. Figure 4.11 shows the magnitude of electric field as a function of the normalized radial distance $r_n$, calculated using Eq.4.4 for the aforementioned voltage values. Field is the greatest in the region right around the inner conductor and placing the SCOS in this region is best for signal-to-noise ratio (SNR) improvement and measuring smaller voltages. However, drilling too close to the inner conductor presents a risk of damaging the inner conductor and increases chance of dielectric breakdown, so it’s safer to place the SCOS between $r_n = 0.25$ and 0.5. Figure 4.10 shows distortions in the voltage measurements (Figure 4.10(a)) made by a sensor with a damaged insulation to the inner conductor (Figure 4.10(b)). For the coaxial cable RG-218, according to Eq.4.4, the breakdown of air at $r_n = 0$ and 0.5 is achieved for voltages $V = 9.3$ kV and $V = 20.8$ kV, respectively.

The field in the coaxial cable after the SCOS insertion is analyzed using numerical methods with ANSYS Maxwell electric field simulation software. Figure 4.12 shows the simulated electric field magnitude inside the dielectric of an RG-218 coaxial cable at 20.8 kV (the value chosen based
on the previous discussion) with the SCOS inserted in a 3 mm hole filled with air and located at \( r_n = 0.5 \). As expected, the field attenuates with the distance of the hole from the inner conductor and is zero inside the conductors. However, field is significantly distorted inside the SCOS-hole and only negligibly distorted around the hole. Voltage measurement method, nevertheless, is unaffected by this distortion because the proportionality of voltage and electric field in a fixed electrode structure remains.

The presence of the SCOS hole distorts the electric field significantly because homogeneous dielectric material becomes replaced with heterogeneous materials. Material heterogeneity causes spikes in electric field, especially around sharp edges, as can be seen in Figure 4.12. The marker m1, placed at \( r_n = 0.5 \) and just above the SCOS slab, shows an increased field value of 3.9 MV m\(^{-1}\) compared to the expected 3 MV m\(^{-1}\) at the same location according to Eq.4.4. The simulated field in the cable dielectric outside of the SCOS hole is consistent with Eq.4.4. This is illustrated by the field value at marker m2, placed at \( r_n = 0.5 \), which is close to the predicted 3 MV m\(^{-1}\). Further simulation with ANSYS Maxwell shows that the value at marker m1 drops from 3.9 MV m\(^{-1}\) to 1.9 MV m\(^{-1}\) after air is replaced with epoxy, which is below the value from Eq.4.4. The reason why the field increases when the SCOS-hole is filled with air and decreases when filled with low-index epoxy is because relative permittivity of air (\( \varepsilon_r=1 \)) is smaller and relative permittivity of low-index epoxy (\( \varepsilon_r=6 \)) is greater than that of polyethylene (\( \varepsilon_{rPE}=2.25 \)).
Figure 4.12: Simulated electric field in a cross-section of the coaxial cable with a SCOS inserted through an air filled hole in the coaxial cable.

Coaxial cable dielectric. The simulated electric fields along the radius going through the SCOS slab (red) and going through the cable dielectric outside of the hole (black) are plotted in Figure 4.13 for comparison. The simulated field values outside of the SCOS hole (black) match very well with the prediction of Eq.4.4. The simulated field values along the line going through the SCOS slab show that inserting a SCOS into an air-filled hole in the cable causes large spikes and drops in the electric field, especially in the air filled regions (red). This is expected since relative permittivity of air is approximately one, which is much smaller than the permittivity of other materials in the SCOS and the coaxial cable. The spikes and drops are more balanced when air in the SCOS hole is replaced epoxy with relative permittivity of $\varepsilon_r=6$ (blue).

Sealing the SCOS hole with epoxy restores the dielectric strength of the cable. The dielectric strength of the coaxial cable is compromised when the holes are filled with air because the dielectric strength of air is much lower than the dielectric strength of all other materials in the
Figure 4.13: Simulated electric field magnitude along a radial line of the coaxial cable without a SCOS (black), and with a SCOS in a hole filled with air (red) and epoxy (blue).

cable. Dielectric strength of the LS-6140 epoxy from Nusil, used for sealing the SCOS hole, is about 20.5 MV m\(^{-1}\), comparable to the dielectric strength of PE(LE) (21.7 MV m\(^{-1}\)), the dielectric of the RG-218 coaxial cable. Based on the Maxwell simulation results, the voltage needs to exceed 164 kV = 20.5 MV m\(^{-1}\)/1.9 MV m\(^{-1}\) \* 20.8 kV for the dielectric breakdown of the sealing epoxy to occur.

4.2 Inserting a SCOS in a coaxial cable

A SCOS is fabricated by placing a 0.5 mm x 1 mm LN crystal on a side-polished Panda fiber [50] and sealing it in a 2.6 mm by 1.8 mm ceramic trough with low-index epoxy for better optical confinement and electrical insulation. The SCOS is made entirely of dielectric materials: MACOR ceramics (trough), fused silica (fiber), lithium-niobate (electro-optic slab), low-index epoxy LS-6140 by Nusil and 5-min epoxy. The packaged SCOS is inserted into a 3 mm hole drilled perpendicular to the axis of an RG-218 coaxial cable as shown in Figure 4.14 (left). The SCOS is positioned so that the LN crystal is aligned with the inner conductor of the cable and its optic axis is oriented in the radial direction along the field lines. The SCOS is then fixed to the bottom of the cable hole by modeling clay. The clay also closes the bottom side of the hole so the epoxy doesn’t drip out during the pouring process. The epoxy is poured from the top side of
the hole using a syringe with a needle through the millimeter wide opening. The low viscosity of the LS-6140 epoxy allows it to flow through the syringe needle and fill out the hole within hours. The epoxy is left for about 24 hours to cure before modeling clay is removed and both ends of the cable hole are sealed with fast-setting epoxy. This process ensures that no air remains in the hole to cause the charge build-up effect, as discussed in the section 4.1.2, and compromise dielectric strength of the coaxial cable.

4.2.1 Charge build-up measurement

To test the charge build-up effect, coaxial cable voltage is simultaneously measured with two sensors: one in a hole filled with air (air-SCOS), as shown in Figure 4.14(a), and the other sealed with epoxy (epoxy-SCOS), as shown in Figure 4.14(b). Figure 4.15 shows the setup for simultaneous measurements. Sensors are interrogated using tunable lasers (TL1 and TL2) at fiber inputs and photodiodes (PD1 and PD2) at fiber outputs. Signals from both sensors are displayed on the oscilloscope (OSC). The measured voltage pulse is generated by a high voltage generator (V), charging the capacitor $C = 30 \text{nF}$ through the input resistance $R_g = 200 \text{ M}\Omega$ when switch SW1 is closed. After capacitor C is charged, the switch SW1 is opened sending the voltage pulse into the coaxial cable by closing the switch SW2 and discharging into the resistor $R_L = 200 \text{k}\Omega$. 
Figure 4.16 shows select voltage pulse measurements with the two SCOS’s. Figures 4.16(a) and 4.16(b) show measurements of the 8 kV and 10 kV voltage discharges, respectively. Measurements with the air-SCOS and the epoxy-SCOS match well at 8 kV, but at 10 kV the air-SCOS measures reduced voltage at the peak as compared to the epoxy-SCOS, which measures the expected 10 kV. This voltage drop between 8 kV and 10 kV is caused by the charge buildup as discussed earlier in section 4.1.2 and matches the 9.3 kV - 20.8 kV voltage range for air breakdown shown in Figure 4.11.

Charge buildup effect is more prominent at higher voltages because at higher electric fields the dielectric breakdown of air is more intensive. This is illustrated by the 16 kV measurement shown in Figure 4.16(c). The air-SCOS measurement shows significant signal attenuation in the first 7 ms, measuring even negative voltages caused by the opposing electric field of the built-up charge. The epoxy-SCOS measurement peaks at the expected 17 kV and always stays positive. There is a negative voltage offset in the air-SCOS voltage caused by charges in air forming an opposing local electric field as illustrated in Figure 4.9 only around the air-SCOS and not around the epoxy-SCOS. To verify the effectiveness of epoxy-sealing method in preventing charge buildup, the same measurement is repeated after sealing the air-SCOS with epoxy. Figure 4.16(d) shows this measurement for a 20 kV discharge. No voltage attenuation can be seen in the measurement indicating that sealing a SCOS with epoxy eliminates the observed charge buildup effect.

The measured time constants in Figure 4.16 vary from the expected value, decreasing with voltage. For the 8 kV and 10 kV measurements (Figures 4.16(a) and 4.16(b), respectively) the time constants are \( \tau \approx 4.9 \) ms, for the 16 kV and 20 kV epoxy-SCOS measurements (black line on Figure 4.16(c) and 4.16(d), respectively) \( \tau \approx 4.2 \) ms and for the 20 kV sealed air-SCOS measurement (grey line on Figure 4.16(d)) \( \tau \approx 2.5 \) ms. These values differ from the expected value of \( \tau \approx 6.3 \) ms, which is calculated by multiplying 200 k\( \Omega \), the output resistance \( R_o \) in Figure 4.15, with 31.4 nF.
Figure 4.16: Simultaneous SCOS measurements using a SCOS sealed with epoxy (epoxy-SCOS, black line) and with air (air-SCOS, gray line), showing the charge buildup effect and how sealing the SCOS with epoxy fixes it.

the output capacitance comprised of the capacitor $C = 30 \text{nF}$ in parallel with the capacitance of the coaxial cable, which is $1.4 \text{nF}$ for a $13.4 \text{m}$ long RG-218 cable with $101 \text{pF/m}$ capacitance per unit length. The time constant calculation assumes a perfect exponential pulse, while the measured exponential pulse has superimposed oscillations due to the inductances and capacitances of the impulse voltage generator [77]. The measured time constants are reduced because they fall on the troughs of the superimposed oscillation. This reduction in the measured time constant value also increases with voltage as the oscillations become more pronounced at higher voltages.
4.3 The SCOS voltage measurements in a coaxial cable

Different voltage measurements in the coaxial cable are shown to describe a SCOS performance in terms of minimum and maximum voltage and high and low corner frequency.

4.3.1 Minimum detectable voltage

In Section 4.1.2 charge buildup effect is observed in voltage measurements exceeding 10 kV. In this section it is shown that a SCOS is able to measure much lower voltages inside the coaxial cable. Measurements are shown for 100 V and down to 1 V.

Figure 4.17 shows measurements of 500 V\textsubscript{pp} and 100 V\textsubscript{pp} 2 kHz sine waveforms. The setup used is similar to Figure 4.15, with the difference that $R_0$ is removed, leaving the coaxial cable open, and the AC voltage generator is directly connected to the coaxial cable input. The two sine measurements are phase-matched by matching the generator’s voltage monitor signals. The 500 V\textsubscript{pp} measurement is significantly above the noise-level of the system and shows up as a clean sinusoid. However, the 100 V\textsubscript{pp} measurement has the SNR $\approx$ 1 and measuring lower voltages in time domain becomes impossible due to system noise.

Voltages down to 1 V are detected by running the uncalibrated SCOS signal through an electrical spectrum analyzer (ESA) and observing the signal in the frequency domain. ESA mea-
sures only the uncalibrated SCOS voltage, and not the actual applied voltage, because the SCOS voltage is correlated to the known applied voltage only in the later calibration step in the post-processing stage. Figures 4.18(a) and 4.18(b) show the screenshots of the electrical spectrum analyzer measuring the uncalibrated SCOS voltage when applying $10 \text{V}_{\text{pp}}$ and $1 \text{V}_{\text{pp}}$ 2 kHz sinusoid signals, respectively. The central peak corresponds to the measured 2 kHz sine signal and the surrounding peaks are part of the system noise. The frequency of the 2 kHz sine is chosen so that it falls in the noise-free frequency range. Figure 4.18(a) shows that when $10 \text{V}_{\text{pp}}$ 2 kHz sine is applied to the SCOS, the uncalibrated SCOS signal gets a $44 \mu\text{V}$ component at 2 kHz. Figure 4.18(b) shows that when the sine signal is attenuated ten times to $1 \text{V}_{\text{pp}}$, the 2 kHz component proportionally drops to one tenth of the corresponding value to $4 \mu\text{V}$.

4.3.2 Maximum voltage

Measurement of a high voltage discharge is demonstrated using similar setup as the one shown in Figure 4.15 in Section 4.2, except with corresponding high-power components. The capacitance is $C = 17.5 \text{nF}$ and output resistance is $R_o = 25 \text{k}\Omega$. The time constant of the output stage changes to $\tau \approx 0.5 \text{ ms}$. Figure 4.19(a) shows the measurement of discharge of a capacitor charged to 100 kV. The oscillations at the beginning of the discharge reach a maximum voltage of 157 kV,
which is shown in detail on the inset. According to Eq.4.4, the electric field at 100 kV ranges from 15 MV/m to 20 MV/m when the SCOS is located between $r_n = 0.25$ and 0.5. Measuring high electric fields like these requires the SCOS signal to be calibrated using non-linear calibration for accurate results [40, 41]. The discontinuity around 40 kV cannot be accurately calibrated because it corresponds to the zero-sensitivity region in the SCOS spectrum. Consequently, the exponential shape of the pulse is distorted in this part of the signal causing the measured time constant of 0.4 ms to slightly differ from the calculated 0.44 ms = 17.5 nF * 25 kΩ.

The oscillations at the beginning of the discharge reach a maximum voltage of 157 kV, which is shown in detail on the inset. Using the non-linear calibration technique the 157 kV voltage causes a $\Delta \lambda = 1.78$ nm maximum wavelength shift. The SCOS transmission spectra shown in Figure 2.2(a) has periodic resonances. The wavelength spacing between the resonances, called the free spectral range, is about 6 nm. If the wavelength shift exceeds the free spectral range the measurement is inaccurate. Therefore, the maximum voltage is less than $6 \text{nm} / 1.78 \text{nm} * 157 \text{kV} = 529 \text{kV}$. However, the voltage limitation caused by the free spectral range of the SCOS is much larger than the voltage limitation caused by the material dielectric strength. In Section 4.1, the maximum voltage due to dielectric breakdown is estimated to be 164 kV. This corresponds to a
dynamic range of 52 dB, which is close to the 60 dB dynamic range for electric field measurements reported in Chapter 3.

The total energy of the measured voltage pulse equals the energy $E$ stored in the capacitor before the discharge $E = 87.5 \, \text{J} = \frac{1}{2} \times 17.5 \, \text{nF} \times (100 \, \text{kV})^2$. The corresponding pulse power is, therefore, approximately $P = 175 \, \text{kW} = E/\tau$. Implications of measuring voltage signals with higher power is a potential subject of future research.

### 4.3.3 Oscillations

The oscillations in the measurement in 4.19(a) are common in high impulse voltage measurements and occur due to inductances and capacitances in the measurement setup [77]. The setup is analyzed analytically using circuit theory and the observed oscillations are demonstrated with an LTSpice simulation.

The measurement setup in Figure 4.15 is modeled with the circuit shown in Figure 4.20. The circuit models only the discharge part of the setup, after the switch SW1 is opened with the capacitor $C$ ($C = 17.5 \, \text{nF}$) at full charge and after the switch SW2 is closed. Therefore, the initial condition for the circuit is that $V_C(0) = 100 \, \text{kV}$. Series inductance of the charge capacitor is modeled as $L = 40 \, \text{nH}$, where the value is obtained from the technical specifications of the capacitor. Resistor $R$ models the total series resistance of the setup, including cables, switches and connections. Load resistor is $R_L = 25 \, \text{k} \Omega$ and the transmission line is modeled with $C_{\text{coax}}$ and $L_{\text{coax}}$. 
Coaxial cable is modeled as a lossless transmission line with a lumped-element model with one element [78]. Values for the model capacitance $C'_{\text{coax}}$ and inductance $L'_{\text{coax}}$ are given by

$$C_{\text{coax}} = C'_{\text{coax}} \times l,$$  \hspace{1cm} (4.5)

and

$$L_{\text{coax}} = L'_{\text{coax}} \times l,$$  \hspace{1cm} (4.6)

where $C'_{\text{coax}}$ is the capacitance per unit length, $L'_{\text{coax}}$ is the inductance per unit length and $l$ is the length of the coaxial cable. Value $C'_{\text{coax}} = 101.05 \text{ pF/m}$ is obtained from the technical specifications for the RG-218 coaxial cable used in the experiment. Value $L'_{\text{coax}} = 252.625 \text{ nH/m}$ is calculated from the characteristic impedance equation for a lossless transmission line, as given by

$$Z_0 = \sqrt{\frac{L'_{\text{coax}}}{C'_{\text{coax}}}}.$$  \hspace{1cm} (4.7)

Kirchhoff’s circuit laws give three independent equations for the model circuit in Figure 4.20 based on the reference directions of the currents $i, i_x$ and $i_L$, namely

$$i = i_x + i_L,$$ \hspace{1cm} (4.8)

$$\frac{1}{C_x} \int_0^t i_x(\tau)d\tau = i_L \times R_L,$$ \hspace{1cm} (4.9)

and

$$\frac{1}{C} \int_0^t i(\tau)d\tau + (L+L_x)i' + Ri + \frac{1}{C_x} \int_0^t i_x(\tau)d\tau = 0.$$ \hspace{1cm} (4.10)

In simplest terms the measured voltage in Figure 4.19 corresponds to the voltage across the load resistor $R_L$. Equations 4.8-4.10 can be transformed into a homogenous second order differential equation with constant coefficients in terms of the current across the load resistor $i_L$, which is proportional to the measured voltage as $V = R_L i_L$. Differentiating Eq.4.9 and expressing
\[ i_x \text{ gives} \]
\[ i_x = R_L C x i_L'. \quad (4.11) \]

Substituting Eq. 4.11 in Eq. 4.8 gives
\[ i = R_L C i_L'' + i_L . \quad (4.12) \]

Differentiating Eq. 4.12 two more times gives
\[ i' = R_L C i_L'' + i_L', \quad (4.13a) \]
and
\[ i'' = R_L C i_L'' + i_L''. \quad (4.13b) \]

Differentiating Eq. 4.10 gives
\[ \frac{1}{C} i + (L + L_x) i'' + Ri' + \frac{1}{C} i_x = 0 . \quad (4.14) \]

Substituting Eq. 4.12 and Eq. 4.13 into Eq. 4.14 gives the second order differential equation in terms of \( i_L \)
\[ i_L''(L + L_x) R_L C_x + i_L''(L + L_x + R_L C_x R) + i_L' \left( R + R_L + \frac{R_L C_x}{C} \right) + \frac{1}{C} i_L = 0 . \quad (4.15) \]

The solution to Eq. 4.15 has the form
\[ i_L(t) = C_1 e^{s_1 t} + C_2 e^{s_2 t} + C_3 e^{s_3 t} , \quad (4.16) \]
where \( C_1, C_2 \) and \( C_3 \) are constants determined by the initial conditions, and \( s_1, s_2 \) and \( s_3 \) are the solutions to the corresponding characteristic equation
\[ s^3 (L + L_x) R_L C_x + s^2 (L + L_x + R_L C_x R) + s \left( R + R_L + \frac{R_L C_x}{C} \right) + \frac{1}{C} = 0 . \quad (4.17) \]
Substituting the occurrences of $L_x$ with $z_0^2C_x$ based on Eq.4.7, the characteristic equation becomes

$$s^3 \left( L + z_0^2C_x \right) R_L C_x + s^2 \left( L + z_0^2C_x + R_L C_x R \right) + s \left( R + R_L + \frac{R_L C_x}{C} \right) + \frac{1}{C} = 0 \quad . \quad (4.18)$$

Since the measurement suggests the circuit gives an under-damped response, two of the solutions to the characteristic equation are a complex-conjugate pair

$$s_{1,2} = -\alpha \pm i\omega \quad , \quad (4.19)$$

and the third one is real

$$s_3 = -\frac{1}{\tau} \quad , \quad (4.20)$$

where $\alpha$ is the decay rate, $\omega$ is the oscillations’ frequency and $\tau$ is the time constant of the impulse. These parameters can be obtained from the measurement in Figure 4.19. Solutions $s_1$, $s_2$ and $s_3$ are related through Viète’s formulas as given by

$$2\alpha\tau + 1 = C \left( \alpha^2 + \omega^2 \right) \left( L + z_0^2C_x + R_L C_x R \right) \quad , \quad (4.21a)$$

and

$$\frac{\alpha^2 + \omega^2}{\tau} = \frac{1}{C \left( L + z_0^2 \right) R_L C_x} \quad . \quad (4.21b)$$

Solving the system of equations 4.21 for $C_x$ and $R$ gives

$$C_{x1,2} = \frac{-R_L L \pm \sqrt{R_L^2 L^2 + 4 \frac{z_0^2 R_L}{C(\alpha^2 + \omega^2)}}}{2z_0^2 R_L} \quad , \quad (4.22a)$$

and

$$R = \frac{2\alpha\tau - 1}{R_L C_x C(\alpha^2 + \omega^2)} - \frac{L}{R_L C_x} - \frac{z_0^2}{R_L} \quad . \quad (4.22b)$$

The solution for $C_{x2}$, which corresponds to the minus sign in the Eq.4.22a, is not physical since it gives a negative value for capacitance. Parameters $\tau$, $\omega$ and $\alpha$ are obtained from the measurement in Figure 4.19. Time constant $\tau = 409\mu$s is the time needed for the exponential part of the voltage impulse to drop to $1/e$ of the maximum value, oscillations exponential decay $\alpha = \frac{1}{3\mu}s$ is obtained
by visually matching the oscillations decay rate of the model to the measurement. Measured oscillation frequency is 2.93 MHz, whereby $\omega$ is calculated as $\omega = 2\pi \times 2.93$ MHz. Plugging all these values in Eq.4.22 gives

$$C_x = 1042.3 \text{ pF} \quad , \quad (4.23a)$$

$$L_x = 2.606 \mu\text{H} \quad . \quad (4.23b)$$

and

$$R = 1.656 \Omega \quad . \quad (4.23c)$$

Based on the RG-218 rating for capacitance and inductance per unit length and the solved values for $C_x$ and $L_x$, the length of the cable $l$ can be approximated to be $l = C_x/C'_x = 10.314$ m.

Figure 4.21 shows the voltage across $R_L$ simulated in LTSpice (red) against the measured waveform (black). The modeled capacitances and inductances explain the frequency and decay of the undamped oscillations observed in the waveform, as well as the time constant of the impulse. However, the model does not predict the rise time of the oscillations $\tau_0$, which has a value about $\tau_0 = 550$ ns in the measured waveform. More research is needed to explain the cause of this and it is likely that the increase in circuit complexity would make the mathematical analysis significantly more complex.

The falling edge of the simulated pulse mostly matches the measured waveform, except for the region around 40 kV, which corresponds to the crossing between the SCOS resonance edges and the region of zero sensitivity, as mentioned previously. This is the consequence on using non-linear calibration to measure extremely high voltages [40]. The simulated waveform also goes back to zero faster than the measured waveform, which indicates a slower depletion of charges from the system and could, again, be due to the existence of some unmodeled parasitic capacitance, requiring further research to explain.

### 4.3.4 Measurement accuracy

The accuracy of a sensor is the maximum difference between the actual value and the value indicated by the sensor output, expressed as a percentage of the full range. The SCOS accuracy is limited by intensity fluctuations of the optical signal. Some of these fluctuations can
be seen in the ESA screenshots in Figure 4.18. There, the tenfold reduction of the applied voltage proportionally reduces the uncalibrated SCOS signal, but the optical signal fluctuations, the noise observed around 1.7 kHz, remain unattenuated. Assuming the fluctuations are randomly positive and negative, the SCOS accuracy can be calculated as half of the value of the maximum peak to peak voltage variation in the flat part of the signal divided by the measured voltage range.

Since the SCOS measurements are performed at different optical power levels, oscilloscope sampling rates and voltage ranges depending on the measured signal, the accuracy calculation varies between measurements. Also, the accuracy of the calibration measurement contributes to the total accuracy of each measurement since these measurements combine in the post-processing calibration stage. The calibration measurement is the 6.4 kV voltage measurement shown in Figure 4.22 and its accuracy is calculated from the flat part of the signal before $t = 0$, where the maximum voltage variation in the flat part before the pulse is 29 V, which is $0.5\% = 29 \text{V} / 6.4 \text{kV}$ of the maximum voltage. The accuracy of the 8 kV epoxy-SCOS measurement in Figure 4.16(a) is calculated from the 76 V$_{pp}$ maximum voltage variation before $t = 0$, which is $0.9\% = 76 \text{V}_{pp} / 8.417 \text{kV}$ of the measured peak voltage, by adding the calibration measurement accuracy to get the total accuracy of $1.4\% = 0.9\% + 0.5\%$. Accuracy of the air-SCOS measurement in Figure 4.16(a) calculated the
same way is $1.2\% = 60\, V_{pp}/8.507\, kV + 0.5\%$. The accuracy of the 10 kV epoxy-SCOS measurement in Figure 4.16(b) is $1.1\% = 63\, V_{pp}/10.32\, kV + 0.5\%$ and the accuracy of the 16 kV epoxy-SCOS measurement in Figure 4.16(c) is $1.2\% = 111\, V_{pp}/16.13\, kV + 0.5\%$. The accuracies for the 20 kV measurements in Figure 4.16(d) are $1\% = 120\, V_{pp}/20.72\, kV + 0.5\%$ for the epoxy-SCOS and $1.25\% = 157\, V_{pp}/20.72\, kV + 0.5\%$ for the air-SCOS. The accuracy of the 100 kV voltage pulse measurement, as shown in Figures 4.19(a) and 4.19(b), is $0.7\% = 323\, V/157\, kV + 0.5\%$. The minimum voltage measurement, as shown in Figure 4.17, is a sinusoid waveform with no flat parts, so its accuracy is calculated by the maximum variation of the sinusoid peaks and troughs from the actual values. The exact applied voltages are known, so no calibration is done in the post-processing stage. Accuracy for the $500\, V_{pp}$ measurement is $28\% = 71\, V/250\, V$, and for the $100\, V_{pp}$ measurement the accuracy is $128\% = 64\, V/50\, V$. Such big values are expected here since $\text{SNR} \approx 1$ for such small applied voltages.

### 4.3.5 Frequency response

A SCOS has a low corner frequency and behaves like a high pass filter for the measured signal. Figure 4.22 shows the SCOS measurement (solid line) of the 6.4 kV step voltage (dashed...
The dotted lines indicate the point at which the measured signal drops to $1/e$ of the peak value, indicating a time constant of $\tau = 0.6\text{s}$ and a low corner frequency of $f_{\text{low corner}} = 0.26\text{Hz}$.

More research is yet to be done to determine the high corner frequency of the SCOS since the speed-limiting factors of the existing SCOS-measurements originate from the limited bandwidth of the interrogator and generated signals. Figure 4.19(b) shows the zoomed-in rising edge with 90 ns rise time of the 100 kV measurement in Figure 4.19(a). The rise time is measured as duration in which the rising edge increases from 10% to 90% of the 157 kV peak voltage value. The discontinuity in the rising edge around 40 kV corresponds to the zero-sensitivity region of the SCOS spectrum, like the discontinuity in the falling edge in Figure 4.19(a). SCOS technology has previously been used for voltage pulse measurements with rising edges as fast as 6 ns.
CHAPTER 5. THE COMPLETE HIGH VOLTAGE AND HIGH ELECTRIC FIELD MEASUREMENT SYSTEM - HARDWARE

Measuring high voltages and high electric fields has its challenges related to the sensing technology, but also other general challenges of working with high voltage systems. The challenges specific to the SCOS technology are discussed in Chapters 3 and 4 and include increasing dynamic range with nonlinear calibration and reducing system perturbation, respectively. Other high voltage system challenges include electro-magnetic interference (EMI), dielectric breakdown, unwanted corona discharges and arcing. These affect the electrical components of the system such as interrogator, voltage generator and electrodes. This chapter lays out the encountered challenges and their overcoming in building a complete, fully functional high voltage and high electric field measurement system.

5.1 Interrogator

Interrogator is the second most important part of the measurement system after the sensor. Interrogator is discussed in general in Chapter 2. Here, an actual realization of an interrogator is given with details of each component.

5.1.1 Increasing interrogator bandwidth

The interrogator schematic as shown in Figure 2.2(b) is shown again in Figure 5.1(a) for convenience. The complete measurement system discussed in this chapter uses a modified version of this interrogator where the trans-impedance amplifier stage (TIA) is removed, as shown in Figure 5.1(b). This modification increases bandwidth from 200 MHz to 1.2 GHz with a marginal trade-off in signal strength. The modified interrogator achieves signal amplification in the optical signal stage by using more powerful lasers and exploiting the reduced power loss of the new Panda-SCOS technology [50, 69]. The effects of using optical signal amplification rather than electrical signal
amplification on noise and signal strength are shown in Figures 5.1(c) and 5.1(d). Figure 5.1(c) shows a comparison of idle signal traces of two interrogator configurations AC-coupled to the oscilloscope to compare the noise levels in time-domain. The interrogator configuration without the TIA exhibits less noise than the configuration with TIA, but this corresponds to the slightly lower overall voltage signal as seen in Figure 5.1(d).

The complete measurement system for high voltage and electric-field uses the dual configuration of the interrogator shown in Figure 5.1(b) designed for two simultaneous measurements. Figure 5.2 shows the diagram of the complete measurement system. Optical connections are in blue and electrical connections in black. All optical fibers use Angled Physical Contact (APC) connectors for reduced optical reflections, which are critical for phase noise [54]. Electrical connectors are labeled in green. There is a total of two interrogators with four optical connections, two from the lasers to the SCOS and two from the SCOS to photodiodes. The oscilloscope can be connected either internally to the PD1/2 connections directly, or externally after shorting the EX1-PD1 and EX2-PD2 connectors. The DAQ connects in parallel to the photodiode for real-time
computer acquisition of the SCOS signal for the needs of the interrogator software. Interrogator software also has achieves remote control of the lasers through the labeled USB connections.

Figure 5.3 shows the physical components of the interrogator. Two lasers are used: CoBrite-DX1 by IDPhotonics, as shown in Figure 5.3(a), and Oclaro iTLA as shown in Figure 5.3(b). Both lasers are tunable, however, only CoBrite-DX1 has adjustable optical power in the range from 6 mW to 40 mW, while Oclaro has a fixed optical power output of 20 mW. Figure 5.3(c) shows the data acquisition device NI USB-6002 by National Instruments. The maximum sample rate of NI USB-6002 is 50 kS/s, but the interrogator software software doesn’t require sample rate above few samples per second. Figure 5.3(d) shows the Thorlabs InGaAs photodiode DET01CFC with bandwidth from DC to 1.2 GHz and responsivity around 1 A/W around the optical wavelength 1550 nm.

5.1.2 Electromagnetic interference

High voltage generators create a lot of electromagnetic noise in their environment due to high static charge buildup in the materials, corona, streamer discharges and arcing, where air
breaks down and becomes conductive to electricity. The emitted electromagnetic waves, when picked up by surrounding electronic devices, can cause malfunction or even great damage. The best protection from electromagnetic interference (EMI) is achieved by using a Faraday cage. Faraday cage can be used to protect enclosed devices from external radiation as well as protect the outside devices from the harmful radiation contained inside.
Figure 5.4: RF-Box electromagnetic interference test inside of a Faraday cage.

Figure 5.4(a) shows the high voltage setup inside of a room-sized Faraday cage. Metal walls of the cage can be seen in the background as well as a window made with a fine metal mesh, which lets light through but blocks radio frequency radiation. The high voltage generator is in a maroon plexiglass box on the table in the back and the interrogator is placed inside a military grade radio-frequency (RF) shielded box, which itself is a type of a Faraday cage. All the electronic devices placed inside of the box are protected from the EMI created by the pulse generator.

To test the efficiency of the RF-shielded box, measurements of a 70 kV impulse are made with the interrogator inside the open and closed RF-shielded box. The measurements are shown side by side in Figure 5.4(b). As expected, the closed box removes the EMI from the measured signal very well. The EMI in the open box measurements is many times larger than the signal. The EMI voltage exceeds the voltage range in both positive and negative direction during the first 5 µs of the pulse, completely obscuring the signal value in that time window.
Shielding the interrogator from EMI is required for any high voltage measurement, so the interrogator components are packed in the RF-shielded box. The SCOS is unaffected by EMI because it is made entirely out of dielectric materials. Figure 5.5(a) shows all the interrogator components fit on a board and secured with cable ties. Figure 5.5(b) shows the inside of the box when the board pushed inside. Box has a vent option with two openings and one fan creating air
flow to keep the box cool. The vent openings do not allow the radiation in because they are also RF-shielded with metal mesh. The RF-shielded box includes a filtered AC power strip rated at 20 A to power up the interrogator components as well as the oscilloscope and laptop (not shown in the figure). The power strip, the USB and coaxial cables are accessible in the empty space under the interrogator reserved for internal oscilloscope and laptop running the interrogator software. Figure 5.5(c) shows the connector panel on the outside of the box. These connectors correspond to the four optical connections with two SCOS’s and two external electrical connections to the oscilloscope as described in Figure 5.2. Figure 5.5(d) shows an early version of the interrogator packed in the RF-shielded box being applied to measure the output of a high voltage discharge.

5.2 Partial discharge and unwanted arcing

As shown in the previous section, the measurement system can be protected from EMI to a great degree using a Faraday cage, however electric discharges other than the desired, controlled, discharge of the pulse generator cause distortions in the voltage pulse, noise in the signal and, in the worst case scenario, damage to the system and/or injury to staff. Besides, corona discharges and streamers act as stray resistances, or shorts, in the high voltage system and drain the system energy. An ideal high voltage system doesn’t allow any discharges besides the controlled discharge of the pulse generator. To test a high voltage measurement system one must be able to generate a high voltage impulse with controlled pulse duration and size, which can be used to determine the correctness of the measurements. This section discusses the practices used to suppress unwanted discharges in the high voltage system under test.

Partial discharge is a localized breakdown of a portion of dielectric under high voltage stress, which does not bridge the gap between two conductors. Partial discharge can occur in fluid dielectrics as well as solids. In air, partial discharge manifests itself as corona, which is seen as a steady purple glow. Partial discharge in solids is invisible and is detected by material damage. Complete breakdown of the dielectric bridges the gap between two conductors and creates an arc.
Figure 5.6: Occurrence of corona discharge between a finger and the electrode of a Van de Graaf generator. Photo is made in a dark room using a few seconds long exposure and a flash.

5.2.1 Corona, streamer discharges and unwanted arcing

Figure 5.6 illustrates partial discharges by showing a picture of a finger approaching the electrode of a Van de Graaf generator. Streamer discharges can be seen as the broken bright lines that start from the finger and fade away towards the electrode. Corona discharge is seen as the purple glow along the path between the tip of the finger and the electrode. What is happening is that air particles get ionized and move along the electric field lines, but the field is not strong enough to close the conductive path and the charged particles stray away.

A SCOS can be placed within a region of high electric field with minimum perturbation to the measured field and low risk of damaging the sensor because it is completely made of dielectric materials. This allows measurement of the waveform and magnitude of fields associated with streamer and corona partial discharges. Figure 5.7(a) shows the setup where a SCOS is placed between two electrodes of the Van de Graaf high voltage generator. Left is the high potential electrode and on the right is the ground electrode. Picture is a composite of two photos, one taken in a dark room to show the streamer discharges and another in the light to show the objects. The
Figure 5.7: A SCOS between electrodes of the Van de Graaf generator. Measurements shown are the uncalibrated and AC-coupled (120 Hz corner) calibrated electric field measurements.

streamers in the picture are not all simultaneous, but the accumulated streamers over a period of a few seconds as the picture is taken with a long exposure. The SCOS measurement shown in Figure 5.7(b) reveal that streamers occur once about every half a second, since streamer discharges
Figure 5.8: Unwanted arcing due to short spacing between conductors (a) and due to conductive impurities on the surface of the dielectric (b).

correspond to the sudden drops in electric field. This measurement is uncalibrated measurement of the electric field close to the high potential electrode (red), in the middle (blue) and close to the ground electrode (black). For this measurement the TIA is set to DC-coupled mode, so the low corner frequency of the measurement corresponds to the 500 mHz SCOS low corner. The calibrated electric field measurement is shown in Figure 5.7(c). For this measurement TIA is set to AC-coupled mode, which raised the low corner frequency to about 100 Hz. The spikes represent streamer discharges and the corresponding drops of electric field range from 0.7 MV/m to 1.3 MV/m.

Figure 5.8(a) shows an arc in air between the inner and outer conductor of a coaxial cable with about 40 kV potential difference. Both partial discharges, like streamers and corona, and arcing can be prevented by placing the conductors in a better insulator such as transformer oil or sulphur hexafluoride (SF6). This works because discharges are initiated by local dielectric breakdown of the medium and by increasing the medium dielectric strength, the conditions for this breakdown won’t be met at the same voltage. In practice, SF6 is the most commonly used gas in HV systems, and extensive scientific research has been done to determine its characteristics and dielectric properties [79]. Liquid dielectrics, especially oils, can also be used to decrease corona loss and prevent arcing. Much is still unknown about fluid dielectric breakdown compared to gas or solid dielectrics. Wadhwa describes their behavior as being so erratic that even two samples of oil taken from the same container do not behave identically [80]. The most commonly used oil is
transformer oil. Its dielectric strength ranges from 20 kV/m to 50 kV/m depending on different properties such as purity, oxidation, etc.

Discharges can also be reduced with solid insulation. Solid insulators often have superior dielectric strength compared to gaseous and liquid dielectrics. These materials could be inorganic such as glass, enamel, or ceramics, as well as organic such as resin, paper, or plastic [79]. Solid insulating materials often have very high dielectric strength, sometimes even higher than liquid or gaseous dielectrics. However, unlike liquid or gaseous dielectrics, damage in solid materials is usually permanent.

Since electric breakdown is caused by large electric fields, unwanted discharges happen around the points with the largest electric field. Since electric field $E$ is proportional to the surface charge density $\rho_s$, as given in

$$E = \frac{\rho_s}{\varepsilon_0}, \quad (5.1)$$

discharges happen around the points with the largest surface charge density $\rho_s$. These points are located along the sharp edges of the object. For dielectric objects this conclusion follows from the fact that the bound surface charge density $\sigma_b$ is

$$\sigma_b = \vec{P} \cdot \vec{n}, \quad (5.2)$$

where $\vec{P}$ is the polarization vector and $\vec{n}$ is a unit surface normal vector. Because $\vec{P}$ is proportional to electrical field as $\vec{P} = \chi \varepsilon_0 \vec{E}$ and dot product of orthogonal vectors is zero, the surface charges move away from the object sides parallel to the electric field $\vec{E}$ and towards the sides normal to it. High surface charge density in this area causes large electric fields.

Similar claim goes for conductive objects. The largest fields are located around sharp edges of conductors. If two conductors are charged with the same total charge $Q$ and one conductor has a spherical shape with a radius $R$ (Figure 5.9(a)) while the other is egg-shaped with radii $R$ and $r$ (Figure 5.9(a)), their potential $V$ can be expressed as

$$V = \frac{1}{4\pi \varepsilon_0} \frac{Q}{R}, \quad (5.3)$$
The surface charge density $\sigma$ is

$$\sigma = \frac{Q}{4\pi R^2} = \frac{V}{4\pi R^2},$$ \hspace{1cm} (5.4)

which gives

$$\sigma = \frac{V\varepsilon_0}{R}. \hspace{1cm} (5.5)$$

The egg-shaped conductor is at the same potential as the spherical conductor, but has a smaller radius of curvature $r < R$ on its sharper end, which means that the surface charge density is higher on its sharper end [81].

The effect of geometry on the intensity and distribution of corona discharges is verified by applying high voltage to electrodes with different shapes. The photos of electrodes at high potential are taken in the light and dark and two pics are superimposed to show the exact location of corona discharges. Figure 5.10 shows that corona discharges are most prominent around sharp electrode edges. The square electrode in Figure 5.10(a) has very bright corona discharge located at the very edges. Disc electrode in Figure 5.10(b) has much dimmer corona, but some bright spots occur around the edge. The rounded and polished electrode in Figure 5.10(c) shows the most even and dim corona glow.

Figure 5.9: The charge density of spherical and egg-shaped conductors.
Figure 5.10: Effect of electrode shape on the intensity of corona discharge.

5.3 200 kV high voltage generator

The excitations for testing the high voltage measurement system are generated by a 200 kV pulse generator. The schematic of the generator is detailed in Figure 5.11. The input stage of the generator has two power supplies, Vdc1 and Vdc2, daisy-chained with ground as the middle point and providing \( \pm 50 \text{kV} \) on the output. This symmetric voltage configuration achieves lower corona losses during the charge stage, while still providing a total potential difference of 100 kV. The variable transformer (VAC), or variac, controls the output of both power supplies by controlling the fraction of the grid voltage that is available on their input. The capacitors C1 and C2 are CSI-100W329 plastic pulsed capacitors with 35 nF capacitance, 100 kV maximum voltage and 175 J maximum energy rating. Capacitors of this size require 5-10 seconds for a full charge when
charged through input resistance of $R_{\text{in}} = 200 \text{ M}\Omega$. The function of $R_{\text{in}}$ is to protect the power supplies in case of a short circuit.

The generator operates in two stages: 1. the charge stage (parallel configuration), and 2. the discharge stage (series configuration), with equivalent schematics shown in Figure 5.11(a). In the charge stage capacitors are connected in parallel to each other and charged up to 100 kV. After enough time is given for a complete charge, the capacitors are reconfigured into a series configuration with the negative terminal grounded and the positive terminal becoming the high voltage output. Since each capacitor is charged up to 100 kV, in series their output potential doubles, reaching up to 200 kV. The complete high voltage generator schematic is shown in Figure 5.11(b).
Since switching between charging and discharging stage is done manually and, therefore, not very fast, the capacitors are tested for how well they hold a charge over an extended period of time when immersed in transformer oil. The result of this test is shown in 5.12. Each capacitor is first charged up to 50 kV. Capacitor voltage is then measured with a 1 GΩ-100 kΩ voltage divider, which is designed and built to measure voltages up to 50 kV. The scaling factor of the voltage divider is $10^{-4}$, which means that it shows 5 V measurement when connected to 50 kV. The measurement in Figure 5.12 shows that the act of measuring the capacitor voltage automatically initiates the capacitor discharge, with the peak measured voltage of 5 V, corresponding to 50 kV (point A). The voltage divider is then disconnected after a few seconds (point B) after which the measured voltage quickly drops from 4.36 V to zero. After 45 s the voltage divider connects to the capacitor again recording the peak voltage (point C) at 4.26 V, about 100 mV lower than the last measurement (point B). The 100 mV drop corresponds to 1 kV drop in the actual capacitor voltage, which is 2.3% of the initial value (point B). Since voltage divider measurements with a large resistance have very poor bandwidth, the observed voltage drop could possibly be much smaller than 1 kV and it is safe to say that the observed 3% per minute is the maximum voltage drop that can be expected.

Since the four switches SW1-4 cannot be instantaneously changed from positions 1 to 2, switching has to be performed in the following order: first SW1 and SW3 (in any order) and then
Figure 5.13: High voltage generator and components. Parallel electrodes and spark gaps with adjustable distance and rounded for low-corona.

SW2 and SW4 (in any order). If the pairs of switches are closed in the wrong order internal arcing may occur and generator output might be lower than expected. A switch box with two switches is shown in Figure 5.13(b). The switch box design is developed with corona reduction and minimum contact resistance in mind, so the electrodes are polished to have smooth surface and maximum contact area. Electrodes are spaced far enough apart to prevent arcing at a maximum potential
difference of 100 kV. The generator contains two switch boxes, one connected as SW1/SW3, and the other as SW2/SW4.

The entire generator is immersed in transformer oil for corona suppression and arcing prevention, metal connectors are polished to round sharp edges and thick wires are used to reduce wire resistance and self-inductance. The generator is shown in Figure 5.13(a). The two DC voltage supplies are at the bottom, capacitors come on top and the two switch boxes in the middle. A trigger switch on top of the capacitors is the SW0 switch in the output stage in Figure 5.11(b), which has the function to send the voltage impulse into the load $R_L$. The spark gap that functions as the switch SW0 is shown up close in Figure 5.13(c). It is made to have adjustable spacing and rounded arcing surface for better control of the arc. The high voltage output of the two capacitors in series is connected to one side of the spark gap and the load $R_L$ is connected to the other side. Shorting the spark gap sends the high power impulse into the load. Spark gap is shorted manually using a plastic insulating stick with a rounded electrode on its far end that creates the short. These rounded electrodes are also used for the parallel plate structure, as shown in Figure 5.13(d), used for conversion of the generated voltage into electric field for electric field calibration. The enclosure is designed to keep the electrode surfaces parallel with adjustable distance. The diameter of the electrodes is 7.6 cm and the maximum electrode distance is 7 cm.

The output characteristic of the 200 kV high voltage generator is given in Figure 5.14. High voltage generator output is measured using the 1 GΩ-100 kΩ voltage divider. Input RMS voltage
is measured by a multimeter and represents the RMS voltage provided by the Variac to the two 50 kV supplies. Blue line represents the voltage measured on the output in the discharge mode. Red line is the expected ideal characteristic provided to verify the linearity of the output. The maximum 200 kV voltage output shouldn’t be exceeded since that could lead to damage to the capacitors.

5.4 Improving robustness of the coaxial cable SCOS

The most convenient way to connect a SCOS to the system under test for high voltage measurements is using the coaxial cable SCOS [42, 43], as described in the Chapter 4. It is convenient
for calibration, there is a negligible change in the SCOS position relative to the electrode structure since the SCOS is glued into place and the cable is shielded and only the cable ends should be insulated against arcing. However, a SCOS inserted in a coaxial cable like this is very sensitive to handling and physical damage, so extra steps must be taken to improve robustness.

Physical damage and wear of a SCOS inserted in the coaxial cable commonly manifest as spectrum shift or power attenuation due to decoupling of the LN crystal and very often a total loss of transmitted power due to fiber breakage. Figure 5.16 shows an example of the optical resonance of a coaxial-SCOS sensor measured before and after a visit to a site for high voltage measurement. The power dropped by about 15 dB, even though the resonance is largely preserved.

To mitigate the possible causes of the observed power loss, different precautions are taken against cable flexion, physical impacts and careless optical fiber handling. To prevent flexion of the SCOS trough and LN crystal decoupling, a SCOS is packaged in a MACOR-ceramics trough, specifically used for this purpose because of the extraordinary material hardness in small sizes. Previously used plastic troughs, even though easy to fabricate in small sizes, are very flexible and bend with the cable during normal handling causing the SCOS to lose its resonances and become dysfunctional. The ceramic trough is manufactured from a 1.59 mm thick sheet of MACOR-ceramics by precision milling, as shown in Figure 5.15(a). Precision milling requires constant air cooling in order for the mill-bit not to overheat and break. The SCOS is further protected from
breaking due to fall by protective shells made of hard ABS plastic firmly attached around the coaxial cable and the inserted SCOS as shown in Figure 5.15(b). Only one shell is shown attached with metal cable ties to the top of the coaxial cable around the SCOS. The second shell attaches from the bottom and completely encloses the SCOS. Cable bending is reduced by enclosing the cable in metal braces placed between the cable and the shell as shown in Figure 5.15(c). The most common place for a fiber break to occur is just outside the trough edge. These fiber breaks are discovered by shining a red test light on one end of the fiber and carefully looking for a sign of red light leaking from within the fiber, which is usually seen as a small red dot, as shown in Figure 5.15(c). To prevent the fiber from breaking in the critical region around the trough edge, the fiber is reinforced with furcation tubing and kevlar as shown in Figure 5.15(d).

The described steps to make the coaxial-SCOS more robust are verified with a series of robustness tests. The four different robustness tests are designed to simulate intense everyday handling and transportation and include:

1. bend test, aimed at testing the effectiveness of metal braces;
2. drop test, aimed at testing the effectiveness of protective shells;
3. vibration test, aimed at testing the exposure of sensor to strain over extended periods of time; and
4. shipping test, aimed at testing the effects of shipping and handling of the sensor by postal service.

5.4.1 Bend test

Bend test is performed on two coaxial-SCOS’s, interrogated in the same way and with similar optical power, but one protected with a metal brace around the cable as shown in Figure 5.17(a) and the other without the added bend protection as shown in Figure 5.17(b). Peak-to-peak value of the measured voltage with the sensor with metal braces is four times smaller than the corresponding peak-to-peak value for the sensor without braces. This indicates that the metal braces reduce the negative effect of bending on the SCOS signal by about 4 times and it is therefore expected that metal braces significantly prolong the sensor lifetime in the long run.
5.4.2 Drop test

Drop test is performed by monitoring the general optical power level of the SCOS with protective shells before and after dropping it from different heights. Figure 5.18(a) shows a picture of the drop test being conducted and 5.18(b) shows the comparison of the optical spectra before and after dropping the sensor onto a hard floor from the heights of 12, 24 and 36 inches. No significant drop in the overall power levels can be observed.
5.4.3 Vibration and shipping test

Vibration test is designed to be an in-lab test of the effect transportation and shipping has on the SCOS optical power. Test is performed by placing the SCOS on a vibration stage as shown in Figure 5.19(a). Test took 17 hours straight. The before and after optical spectra are shown in Figure 5.19(b) and surprisingly show a slight increase in the overall optical power through
the sensor. This increase can probably be attributed to better fiber connections during the post-test measurement, because it’s less likely that insertion loss would be reduced with vibration. The conclusion, however, is that no power loss is detected after an extended exposure to strong vibrations.

Finally, effects of transportation are measured by actually shipping the improved robust sensor to the testing site and back. Results are shown in the Figure 5.20 and show about 3 dB - 4 dB power loss in the returned sensor. This marks an improvement compared to the 15 dB power loss initially observed in the coaxial SCOS without the additional protections.

The described robustness tests are performed on a SCOS with very weak resonances, so the test results can only be used to make conclusions about the overall power loss and not on the change in resonances. However, this doesn’t detract from the relevance of these tests since the overall power loss observed in the SCOS in Figure 5.16 is dominant over the resonance attenuation.
CHAPTER 6. THE COMPLETE HIGH VOLTAGE AND HIGH ELECTRIC FIELD MEASUREMENT SYSTEM - SOFTWARE

Software for the high voltage measurement system allows the user to initialize the interrogator for operation, perform calibration and process the measured data. This chapter gives an overview of the software presented in a user-manual-like form. The application is written in MATLAB® by MathWorks and the complete code is included in the Appendix A: Interrogator software code.

6.1 Running the application

The application is started from MATLAB® main window by typing its name “interrogator” in the command window or opening up its main file interrogator.m and clicking “Run” button in the top of MATLAB® window. This step is shown in Figure 6.1. Note that the current folder setting has to be set to the application’s root directory where interrogator.m is located.

The main window of the application contains two parts: the two spectrum display frames on the right and five panels on the left, each dedicated to a specific device or stage in the measurement process. Panels are ordered from top to bottom to follow the natural flow of the measurement process. Status bar at the bottom of the window indicates the current status of the application. It defaults to “Ready” whenever the application main window is responsive and changes to “Busy” when the application is performing tasks such as communicating with a device or processing data. If the application’s main window is closed, the application runs a shutdown procedure safely disconnecting from all devices with open communication ports. Messages about all running tasks show up in the MATLAB® command window and this is the first place to go to for troubleshooting errors.
Application requires that MATLAB® has the NI-DAQmx Support from Data Acquisition Toolbox installed. This package can be found at: https://www.mathworks.com/hardware-support/nidaqmx.html.

### 6.2 Preparation for measurements

Preparation for measurements starts by initializing the interrogator hardware and connecting it to the computer running the application. The sensor is then selected from a popup menu in the sensor panel in the top left of the main window, as shown in Figure 6.2(a). The default option in the sensor panel is “Add New Sensor”. This option allows adding a previously unused sensor to the application’s database. Sensors are stored in the database with name and the E-shift value in pm/(MV/m) units. This value is important for electric field calibration and is measured for each sensor separately. Commonly measured E-shift values fall between 150 pm/(MV/m) and 300 pm/(MV/m). Typing in the name and E-shift value and clicking the “Set” button adds the given sensor to the database and sets it as the active sensor. A previously saved sensor can be selected by clicking on the popup menu and selecting its name from the list. This populates the name
and E-shift fields with the values retrieved from the database and sets it as the active sensor. At this point the “Delete” button activates and allows the selected sensor to be removed from the database. Alternatively, the sensor database can be managed by simply editing the text file “sensors.txt” in the application’s root path, where each sensor corresponds to a line of text in the format [sensor name], [E-shift value]. Deleting a line with sensor’s details removes the sensor from the database and adding a new line adds it to the database. The application does not need to be restarted for these changes to take effect.

Next step is to select a laser to be used in the measurement and initialize the DAQ device. Order of initializing laser and DAQ is irrelevant. DAQ can be initialized independent of the laser and upon successful connection the DAQ panel reports the status of DAQ as “On” and the current voltage reading, as shown in Figure 6.2(c). The voltage value updates every few seconds with a force refresh by a mouse click on the value. Laser is selected from a popup list in the laser panel, as shown in Figure 6.2(b). Selecting a laser initiates the communication sequence and turning on the optical output of the laser. It is important not to select a laser before it is properly connected and powered on.

Before attempting the communication sequence with the laser, the application prompts the user to select the correct serial communication port, as shown in Figure 6.3(a). The COM
port selection dialog box presents the user with available COM ports as seen by the operating system. If the user is unsure which com port corresponds to the laser, a simple test can be used to determine this: comparing lists of available ports when the laser is connected and connected from the computer USB port reveals the right COM port. This sequence takes about 5 s to complete and upon completion the laser panel updates with the current laser power and wavelength values, as shown in Figure 6.3(b). The “Power” and “Wavelength” fields are both indicator and control fields, meaning that they update to show the actual value, but also take input from the user to set a new value.

The “Auto Power” feature automates the selection of laser power which corresponds to the SCOS voltage signal as seen on the DAQ to 1 V, or maximize the laser power, if 1 V can’t be achieved. The radio button for “Low Noise” allows the user to put the laser in a low-noise mode by turning off the laser dithering mode. Dithering is used for improved laser stability, but is achieved at the expense of superimposing dithering high frequency noise on the laser signal. Dithering can
be turned off for a short period of time, achieving a superior low-noise quality of the laser signal, however allowing the laser to become unstable over time. This is why the application requires the user to set a timer after which the user is prompted to turn the low-noise mode off, as shown in Figure 6.3(c). The low-noise mode timer is shown in Figure 6.3(d).

6.2.1 Resonance scanning

Resonance panel has two parts to it, the wide-band and single resonance spectrum scanning part. These two correspond to the two frames on the right side of the main window: rough and fine spectrum, respectively. When sensor, laser and DAQ are properly selected and initiated, the “Scan WB Spectrum” button activates and allows a rough scan of the entire tunable wavelength range of the laser, which is automatically queried and obtained from the laser during initialization. When completed, the scanned spectrum graph appears in the top right of the window and the
Figure 6.5: Fine scanning of a single resonance with the ability to select the resonance of interest detected by the wide-band scan, select desired resolution and see the progress bar with estimated time of completion.

popup menu in the resonance panel gets populated with the auto-detected resonances, as shown in Figure 6.4(a). Application defaults to the resonance closest to 1550 nm. Also, every completed scan is auto-saved as a .mat file in a special data folder in the application’s root folder, whereby the filenames are formed with the type of scan, sensor name and date and time stamp. This scan takes about five minutes and the progress is updated in real time based on the calculated number of data points and the average scan time of the ongoing scan. The progress bar as shown in Figure 6.4(b).

After selecting one of the resonances from the resonance panel pop-up menu, the user is given the option to set the scan resolution for the fine scan. Entering a value in the “Scan resolution” field the time estimate of the scan is updated with number of data points. Figure 6.5 shows the application window with fine resonance scanning in progress. Depending on the scanning resolution, fine scan can last anywhere from 5 to over 30 minutes. Better resolution aids nonlinear calibration accuracy, but comes at a cost of time. To reduce the scanning time, wavelength tuning is optimized to using seconds-long fine frequency tuning when the subsequent data point wavelength is in the same laser optical mode. PurePhotonics laser has a fine tuning frequency span of $\pm 30\,\text{GHz}$ around the center frequency, CoBrite laser has a fine tuning frequency span of $\pm 12\,\text{GHz}$ and Oclaro laser doesn’t support fine tuning. However, using fine tuning too far from the center frequency introduces an increasing wavelength inaccuracy. This effect is shown
in Figure 6.6. The fine tuning inaccuracy is visible as the periodically broken spectrum line. The periodic segment span the entire spectrum, but the wavelength inaccuracy is most visible in the more horizontal part of the spectrum towards the bottom. The break corresponds to the mismatch in the opposite fine tuning in adjacent optical modes of the laser. At 1550 nm a 12 GHz frequency difference corresponds to 96 pm. The mismatch is fixed by limiting the fine tuning span to ±5 GHz, which speeds up scan times for resolutions finer than 80 pm. The scanning progress bar takes into consideration durations of both the fast fine frequency tuning and normal frequency tuning when estimating the time to completion.

The final step in the fine scan process is to analyze the data in order to determine the point with the steepest slope on the right resonance edge and tune the laser to this wavelength as the operating wavelength. This calculated value is the maximum sensitivity wavelength and is indicated at the bottom of the resonance panel. Clicking on the fine spectrum graph with the mouse shows the data tip with measured voltage/wavelength values at the clicked point. In rare cases when the algorithm cannot positively identify the extent of the resonance edge, as in the case shown in Figure 6.7(a), the maximum sensitivity wavelength can be calculated incorrectly. The maximum sensitivity wavelength can be changed by double-clicking on the “Max. Sensitivity
Wavelength” text bar at the bottom of the resonance panel. Clicking once on it immediately tunes the laser to this value. The current operating wavelength is indicated on the fine spectrum graph with a vertical line, which turns red when the laser is on and gray when it’s off.

At this point, the interrogator system is ready to take measurements. Alternatively, the entire spectrum scanning process can be skipped altogether by loading previously saved spectra from a file. Because parameters like loss at optical connections and temperature vary between measurement setups, using saved resonance scans is only recommended during the same measurement session. Rough spectrum, on the other hand, can be reloaded any time after initial scan because the resonance locations are not likely change. Buttons “Load WB Spectrum” and “Load Single Resonance” allow loading rough and fine spectra from previously saved files. Before any measurement are taken, DAQ should be disconnected from the oscilloscope and photodiodes because it introduces noise in the data.

6.3 Calibration and results window

The interrogator application offers extensive features for previewing and processing the measured data. To access this post-processing stage it’s required to enter the calibration voltage value and load the calibration data file, until that is done the button “Calibrate Trace File” remains
inactive. This is done in the calibration panel, as shown in Figure 6.8. Calibration panel also contains the x1 and x10 radio buttons, which are used to specify the additional voltage gain applied to the calibration measurement relative to other measurements. Selecting the x10 button applies a 1/10 scaling factor to the loaded calibration data, while the default x1 button doesn’t do anything.

The measured data can be previewed immediately upon measurement by clicking “Cali-
brate Trace File” in the calibration panel and loading the measured file, which performs calibration and displays the calibrated measurement in a new window. However, data processing doesn’t have to be done immediately after taking measurements nor does it require the interrogator to be on or connected to the computer running the software. In fact, after taking the measurements, interrogator can be turned off by first clicking the “Shutdown” buttons in the DAQ and laser panels. At this point the entire high voltage setup can be disconnected and interrogator can be powered off. Data processing can be performed independently from the measurement stage and the interrogator using the saved measurement files.
Figure 6.10: Algorithm that recognizes whether the calibration signal is harmonic or transient.

When the user first starts the interrogator software, the calibration panel remains inactive until the resonance spectrum file is loaded, as shown in Figure 6.9(a), and sensor is set in the sensor. After this is done, the “Load Calibration File” button and “Peak-to-peak voltage” field become active, as shown in Figure 6.9(b). From this point on calibration is performed in three simple steps: set the calibration voltage, load the calibration file and select the trace file to be calibrated. This triggers the execution of the calibration algorithm. The first thing the algorithm does is to determine the peak to peak value of the measured calibration data. Calibration algorithm is smart enough to determine the type of calibration signal it’s been given. Figure 6.10 shows the algorithm used to determine whether calibration measurement is harmonic or transient. Peak to peak value of a harmonic signal is calculated using Fourier transform, which is more accurate method for harmonic signals, however, peak to peak value of a transient signal is determined as a difference between the minimum and maximum value of the waveform.
Voltage calibration ratio $R_{\text{cal},V}$ is calculated in accordance with Eq.2.10 and Eq.2.11 as

$$R_{\text{cal},V} = \frac{V_{\text{ext,cal}}}{V_{\text{cal}}},$$

(6.1)

where $V_{\text{ext,cal}}$ is the peak-to-peak calibration voltage entered by the user in the calibration panel and $V_{\text{cal}}$ is the peak to peak interrogator voltage value calculated by the calibration algorithm. The calibration algorithm then multiplies the measurement voltage $V(t)$ by the voltage calibration ratio $R_{\text{cal},V}$ to obtain the linear calibration of the measured voltage, as given by

$$V_{\text{lin}}(t) = R_{\text{cal},V} V(t).$$

(6.2)

Linear calibration of the measured electrical field is obtained by first mapping the calibration measurement to the resonance edge. The process of mapping interrogator voltage to resonance edge is described in detail in Figure 3.2 in Section 3.1.3. The calibration voltage waveform is thus converted into a spectral shift waveform. The calibration algorithm determines the peak to peak value of the spectral shift waveform $\Delta \lambda_{\text{cal,pp}}$ and divides it by the sensor’s E-shift value to obtain $E_{\text{ext,cal}}$. The electric field calibration ratio $R_{\text{cal,E}}$ is then calculated in accordance with Eq.2.7, as

$$R_{\text{cal,E}} = \frac{E_{\text{ext,cal}}}{V_{\text{cal}}}. $$

(6.3)

Linear calibration of the electrical field is then obtained by multiplying the measurement data $V(t)$ with the calibration ratio $R_{\text{cal,E}}$, as given by

$$E_{\text{lin}}(t) = R_{\text{cal,E}} V(t).$$

(6.4)

Nonlinear calibration of electrical field is obtained by mapping the measured waveform $V(t)$ to the resonance edge to obtain the spectral shift waveform $\Delta \lambda(t)$ and then dividing this waveform by the E-shift parameter [40, 41], as given by

$$E_{\text{nonlin}}(t) = \frac{\Delta \lambda(t)}{\text{E shift}}.$$ 

(6.5)
Nonlinear voltage calibration is obtained as

\[ V_{\text{nonlin}}(t) = \frac{V_{\text{cal,pp}}}{\Delta \lambda_{\text{cal,pp}}} \Delta \lambda(t) \]  \hspace{1cm} (6.6)

After performing calibration, a new results window pops up with all four calibrated waveforms \((V_{\text{lin}}(t), V_{\text{nonlin}}(t), E_{\text{lin}}(t) \text{ and } E_{\text{nonlin}}(t))\), as can be seen in Figure 6.11. Each plot’s axes are auto scaled to the proper units, for example measurement in the 10V-40,000 range is automatically shown in kV units and measurement in the 100,000 V/m - 600,000 V/m range is shown in kV/m units. Each of the graph can be enlarged across the entire graph space when clicked. Clicking on the enlarged graph with no tools selected returns to the initial view with all four graphs. In the enlarged mode, the application offers four tools to preview and manipulate the data:

1. Zoom in and out tools, as illustrated in Figure 6.12(a);

2. Data tip tool, as illustrated in Figure 6.12(b);

3. Reference level correction tool, as illustrated in Figure 6.12(c);
4. Reset tool; and

5. Dual edge nonlinear calibration workspace, as illustrated in Figure 6.12(d).

Zoom in and out tools and data tip tool are the standard MATLAB® tools. The reference level correction tool is specially developed for the interrogator app to correct the signal reference level. The tool allows the user to select points which should be made reference level. The tool then finds the average voltage for all the selected points and subtracts that value from the entire waveform. Dual edge nonlinear calibration tool is used in the case of large voltages and electric fields where the spectral shift is so large that the operating wavelength crosses over to the opposite side of the resonance, like the one shown in Figure 3.10(f). Selection should be made to the data-points which cross to the opposite edge. Clicking on the “Save Changed Data” button in the bottom
left corner saves the modified data in a .mat file with the name formed using the sensor name and the current time and date. The reset tool located on the far right removes any changes made to the data.

Clicking on the “Calibration Preview” button on the left opens up the calibration preview space where the calibration data can be previewed. Zoom and data tip tools are available at the top. The calibration data plot is annotated with the DC voltage level (black) and the minimum and maximum voltage levels (red dashed), as shown in Figure 6.13(a). Figure 6.13(b) illustrates the use of data tip and zoom in tools, whereby it can be seen that the annotated lines are zoomed in with the plot. User can exit the calibration preview space by “un-clicking” the “Calibration Preview” button.

More than one interrogator application instance can be run simultaneously on the same computer as can be seen in Figure 6.14. This is important since the interrogator described in Chapter 5 is a dual interrogator system and runs two lasers. Scanning resonances has to be done at different times since DAQ cannot be connected to two instances of MATLAB® due to software restrictions.
Figure 6.14: Two instances of the application running simultaneously for dual SCOS measurements.
CHAPTER 7. OTHER SCOS CONTRIBUTIONS

Other contributions, related to SCOS technology and the high voltage and high electric field measurement system, include the development of the PANDA-SCOS technology and the transformer-oil resistant SCOS. A PANDA-SCOS is used in the final version of the high-voltage and field measurement system, along with a transformer-oil resistant SCOS. Other SCOS-related research, unrelated to high voltage, includes the development of the noise reducing push-pull-SCOS technology and the method for SCOS sensitivity direction rerouting and enhancement using an integrated dipole antenna.

7.1 SCOS technology with PANDA side-polished optical fibers

Side-polished PANDA fibers are used for SCOS fabrication similar to D-fibers [50, 68]. In both cases the coupling between LN crystal and fiber core is achieved by placing the crystal on the flat fiber surface close to the core, as described in the Chapter 2. PANDA fibers provide several advantages over traditional D-fibers. They interface easier and exhibit less coupling loss with lab equipment because of their standard optical core shape and size. Namely, PANDA fibers have circular core with a diameter of approximately 10 µm while D-fibers have an elliptical core with the dimensions of 2 µm by 4 µm. This results in 1 dB - 3 dB more power going through each PANDA fiber per junction than a D-fiber. PANDA fiber is also easier to splice than D-fiber. The automatic nature of the PANDA to PANDA splices provide a great deal more accuracy than the manual method used to splice D-fibers. This results in up to 3 dB less loss per splice in a PANDA fiber than in a D-fiber. This increase in power allows for higher measurement bandwidth using PANDA SCOS technology.

Side-polished PANDA fiber is etched with hydrofluoric acid to allow better coupling between the crystal and the light in the core. Figure 7.1(a) shows a sample of an over-etched fiber. This side-polished PANDA fiber is polished along the fast axis, as labeled in the Figure 7.1(b).
Figure 7.1: Fast and slow axes shown on a microscope picture of a PANDA-fiber cross-section.

Since the industry standard polarization-maintaining (PM) fiber connectors and laser outputs are keyed to the slow axis, splicing a side-polished PANDA fiber with the polished side aligned with the fast axis, as shown in Figure 7.1(a), results in exciting the transverse-magnetic (TM) modes within the coupled LN crystal. This is not the optimal configuration for sensitivity. It has been shown that exciting transverse-electric (TE) modes achieves superior sensitivity and directionality [49]. To improve sensitivity, the PANDA-SCOS needs to be fabricated using a side-polished PANDA fiber polished along the slow axis.

Figure 7.2 shows a comparison of the optical resonances and sensitivity of a SCOS fabricated using a D-fiber (D-SCOS) and a PANDA fiber (PANDA-SCOS). Comparison of the transmission spectra, as shown in Figure 7.2(a), shows that the extinction ratio of the resonances is comparable, somewhere between 10 dB - 15 dB, however, the insertion loss in the D-SCOS is about 10 dB greater than in the PANDA-SCOS. Figure 7.2(b) shows frequency spectra of 10 kHz signal measurements using the two SCOS’s. The ordinate axis shows interrogator voltage, but since both SCOS’s are interrogated with the same interrogator, the two signals can be directly compared. It can be concluded that the measurement sensitivity of the D-SCOS and the PANDA-SCOS is comparable.

Higher optical power in the PANDA-SCOS, due to lower insertion losses, makes the PANDA-SCOS very susceptible to interferometric noise. This noise can be reduced using APC fiber con-
Figure 7.2: Comparison of the PANDA-SCOS (dashed orange) and the D-SCOS (solid blue) optical spectra and measurement sensitivity at 2.5 kV/m.

Figure 7.3: TO-SCOS fabricated with round and square glass tube.

nectors, which eliminate reflections at the fiber-coupling points. Another good method is to use a phase modulator to up-convert interferometric noise to higher frequency bands [54].
7.2 Transformer-oil resistant SCOS

Testing the measurement system with electric fields above 3 MV/m requires the generation and sensing of electric fields in a medium that is much better insulator than air. Chapter 3 shows measurements of electric fields with peak magnitude of 8 MV/m and 18 MV/m in Figures 3.10(c) and 3.10(e), respectively. These electric fields are generated by electrodes at high electric potential immersed in transformer oil. Consequently, the SCOS used to measure the field had to be exposed to transformer oil. Prolonged exposure of the sensor to transformer oil caused its resonances to significantly shrink only after one day of persistent contact with the oil. The transformer-oil resistant SCOS technology is developed to address the need for lasting sensors for high field measurements in such a reactive environment like transformer oil.

Because transformer oil reacted only with the low-index epoxy part of the damaged SCOS without affecting the hard fast-setting epoxy, the transformer oil resistant SCOS (TO-SCOS) is protected by enclosing it in a compact glass tube and sealing it with fast-setting epoxy at the ends,
while the middle part is filled with low index epoxy. This is shown in Figure 7.3. The fabrication process is performed in three simple steps: 1) a SCOS placed in the glass tube, temporarily sealed with modeling clay at the bottom, is filled about one third with fast setting epoxy (enough to cover the transition between the furcation tubing and the trough) and left for about 15 minutes to cure; 2) the low-index epoxy is filled to cover the middle third of the tube, ensuring it covers the section around the LN crystal, and left to cure for 24 hours; 3) the glass tube is topped up with fast-setting epoxy and cured. Care needs to be taken that no air bubbles remain within the epoxy since this could significantly reduce the dielectric strength of the sensor and break the sensor, as is shown in Figure 7.4.

To test the TO-SCOS resistance to transformer oil, sensor is immersed in a transformer-oil bath, as shown in Figure 7.5(a) while monitoring its resonances. The results of this test are shown in Figure 7.5(b). The results show that spectrum doesn’t change even after 4 days of being immersed in transformer oil, while an unprotected SCOS loses its resonances within a day of exposure. In normal use, the TO-SCOS should be taken out of the transformer oil when not in use.
7.3 Pushpull SCOS for measurements in a vibrational environment

A push-pull SCOS is a vibration-insensitive SCOS created by placing in close proximity two LN crystals with oppositely oriented optical axes on the same optical fiber [51, 52]. This way, both SCOS’s exhibit the same response to stress, but an opposite response to the electric field, which allows the two signals to be subtracted to reduce strain effect while increasing electric field sensitivity. Figure 7.6(a) shows a diagram of the push-pull SCOS concept. Optical axes of the crystals are oriented perpendicular to the fiber core and parallel to the flat side with one slab rotated 180° in the plane of the flat side of the fiber. Figure 7.6(b) shows a fabricated push-pull SCOS as seen through a microscope. The distance between the two crystals is less than 1 mm.

Figure 7.7(a) shows the superimposed individual resonances of each crystal, and Figure 7.7(b) shows their composite spectrum. The two crystals are coupled to the fiber in such a way as to maximize the spectral distance between the resonances of the other crystal. This allows the two crystals to be interrogated simultaneously by two lasers tuned to different wavelengths corresponding to the resonance edges of different crystals. The opposite orientation of the optic axes makes the spectrum resonances of different crystals shift proportional to the applied electric field, but in the opposite direction. However, when the measurement stage is subject to vibration, both optical spectra shift proportional to the strain in the same direction. This allows the vibrational
noise to be reduced by subtraction of the two scaled signals. On the other hand, parts of the signals that correspond to electric field add, increasing the overall electric field sensitivity.

The push-pull SCOS principle is demonstrated with a measurement of a 10 kV/m electric field while the measurement stage is subject to slow (100 Hz range) and fast (kHz range) strain.
Figure 7.9: Canceling the vibration effect in the SCOS measurements using the push-pull approach.

Slow strain is applied by shaking the stage with a hand, while the fast strain is applied by an impact to the stage with a hard object. Figure 7.8(a) shows the two separate signals in slow strain measurements. The shape of the electric field waveform is a train of impulses, while the slow strain is very harmonic. When the two signals are appropriately scaled and subtracted, most of the slow strain signal is canceled out and the electrical field signal portion is amplified. Combined signal is shown in Figure 7.8(b). Both measurements are shown in uncalibrated interrogator voltage for easier comparison and to show that the push-pull SCOS technology increases the useful signal while decreasing the effect of strain on the signal.

Figure 7.9 shows the measurement of the same electric field waveform as in Figure 7.8, except with an impact vibration strain. Figure 7.9(a) shows the two separate signals and Figure 7.9(b) shows the two signals scaled and subtracted. The push-pull SCOS suppresses even the strain in kHz frequency range.

7.4 SCOS sensitivity direction rerouting and enhancement using a passive integrated dipole antenna

Passive dipole antenna is integrated onto the SCOS trough to enhance the directional sensitivity of electro-optic electric field measurements parallel to the fiber axis. The dipole antenna
effectively flips the directional sensitivity of the sensor by rerouting it from the longitudinal into transverse direction [53]. Figure 7.10(a) illustrates this mechanism. The effective length of the antenna is along the fiber direction, while the antenna gap is across the transverse sensing direction of the sensor. The change in the field orientation is verified with an electric field simulation using HFSS® software by ANSYS. Inside the antenna gap, where the LN crystal of the SCOS is placed, the field is much stronger (red arrows) than outside (green arrows) and almost entirely transverse, matching the SCOS sensitive direction. The measured external field has a longitudinal direction. Simulation also predicts flat response of this antenna in the bandwidth from DC - 10 MHz. Figure 7.10(c) shows the described structure fabricated by depositing aluminum onto the ceramic trough.

Figure 7.10: Sensitivity direction rerouting using a passive integrated dipole antenna.
walls to act as antenna conductors. Electro-optic slab made of lithium tantalate is centered in the antenna gap.

Sensitivity rerouting is tested by measuring a applying a 5 kHz electric field signal applied in different directions using a SCOS with and without the integrated passive antenna. The results of these measurements are given in Figure 7.11. The SCOS with integrated antenna detects a strong signal when the external field is applied in the fiber direction, while the SCOS without antenna doesn’t detect this field at all. Field applied in the optic axis direction is detected by both the
SCOS with and without integrated antenna with equal sensitivity and neither sensor are sensitive to the electric field in the up-down direction.
CHAPTER 8. INCREASING DYNAMIC RANGE OF FIBER BRAGG GRATING INTERROGATOR

This is a non-SCOS related chapter and it covers the development of a method for increasing dynamic range of fiber Bragg grating (FBG) interrogator by a control feedback loop between the signal and the laser [47, 48]. However, the overlap with the SCOS-related research is the fact that both FBG and SCOS are optical fiber sensors, which use edge-filtering interrogation method for highly sensitive measurements. This research is motivated by the need for highly sensitive strain measurements in a strong vibration environment for structural health monitoring of wind-mill blades, as they are exposed to strong winds.

Fiber Bragg grating (FBG) strain sensors have found wide use in structural health monitoring over the past couple of decades [82–85]. More recently, FBG sensors have been applied to the detection of ultrasonic waves [86, 87]. Ultrasonic wave detection can be from an external source [87] or passive acoustic detection [88, 89]. Ultrasonic waves propagating through a material get transformed or reflected off of any non-homogeneity, so detecting these waves can indicate the location and size of cracks, holes and other kind of material damage. Detection of small defects is used for early indication of material breakdown [90]. Fiber-optic sensors are widely used for monitoring the structure health in aerospace and wind-energy industries [91]. Strains as small as $10^{-13}\varepsilon Hz^{1/2}$ have been measured using a fiber Bragg-grating resonator with a diode-laser source stabilized against a quartz-disciplined optical frequency comb [92].

Detecting ultrasonic waves requires sensor sensitivity to small strain at high repetition-rates, which FBG sensors can achieve with edge-filtering demodulation technique. The trade-off is a very limited strain range [93]. The strain range limitation becomes a problem when this method is used in strong vibrating systems. The limitation can be overcome with a broadband optical source and a tunable optical filter in combination with a micro-controller to lock the interrogator to the linear part of the FBG reflection spectrum [47, 48]. The voltage signal produced by the micro-controller then correlates to the large-amplitude slow-varying strain, while the interrogator signal
Figure 8.1: FBG reflection spectrum with no strain (solid) and with strain (dashed). FBG spectral shift results in a change of the reflected power $\Delta P$ if the optical source has a fixed wavelength. The shaded portion indicates the usable linear part of the spectrum edge for reflectivity between 20-80%.

correlates to the low-amplitude fast-varying strain, measuring both the slow and fast vibration simultaneously.

Previous research demonstrates similar wavelength locking schemes with a control loop used for laser output stabilization [94], stabilizing FBG sensor output due to temperature drifts [95], measuring quasi-static strain in the sub-Hz range [96], stabilizing FBG sensitivity at strains up to about 20 Hz [97], etc. More recent research demonstrates wavelength locking at higher speeds up to 4kHz using a ring laser configuration and a PID (proportional-integral-derivative) controller [98]. In this research a simpler and cheaper solution is given using an inexpensive micro-controller board in a proportional control loop to achieve wavelength locking at frequencies up to 1 kHz, which is adequate for isolating the wind-induced vibrations.

8.1 Background

FBG sensors are optical fiber sensors that reflect light in a narrow wavelength range, which shifts as a function of the applied strain at a rate of 1.2 pm/µε [99]. This change can be measured in a number of ways described in detail in the literature [100]. Here, FBG sensor is interrogated using
the edge-filtering technique, where FBG reflection spectrum is filtered in a narrow range along the spectrum edge. Figure 8.1 shows the shift in the FBG reflection spectrum caused by the applied strain. The FBG spectral shift results in a proportional change of reflected optical power when observed at a single fixed wavelength. The largest change in power corresponds to highest strain sensitivity and is achieved at the middle of the reflection spectrum edge, i.e. the mid-reflection point.

For large shifts, the observed wavelength falls outside of the spectrum edge and the interrogated optical power stops being proportional to the wavelength shift. The useful part of the spectrum can be estimated as the part where the reflectivity is between 20% and 80%, as illustrated by the shaded region on Figure 8.1. The spectral width of this band corresponds to the maximum wavelength shift allowed, proportional to the maximum strain range that can be measured. For the FBG in Figure 8.1, the spectral width of the shaded linear part of the spectrum edge is 50 pm, corresponding to a maximum strain of about 40 $\mu$e. This range can be increased by choosing FBGs with a less steep edge slope, but at the cost of reduced strain sensitivity.

Figure 8.2 shows a typical edge-filtering interrogator setup. Light from a broadband optical source (ASE) is amplified by the optical amplifier (EDFA) and reflected off of the FBG sensor into the fiber Fabry-Perot tunable filter (FFP). Light transmitted through the FFP is then converted into electric current with a photo detector (PD) and then into a voltage signal with a trans-impedance amplifier (TIA). The output signal is recorded on the oscilloscope (OSC). Dashed lines indicate the path and direction of the incident and reflected optical signals of interest. The incident light at the two terminated fiber ends is completely scattered out, so no light is reflected back into the system at these two points.
8.2 System overview

The objective of the developed FBG interrogation system is to enable measurement of large shifts in the FBG reflection spectrum at lower repetition-rate with lower sensitivity, while simultaneously measuring smaller shifts at higher repetition-rate with higher sensitivity. The large shift measurement is attained by tuning the wavelength of the optical filter to the FBG mid-reflection point using a proportional control loop. The smaller shift at high repetition-rate is measured using the previously described edge-filtering technique.

Figure 8.3 shows the implementation of the edge-filtering interrogator with the control loop. The configuration is similar to the basic edge-filtering interrogator in Figure 8.2, except now the micro-controller board ($\mu$C), voltage divider (VDIV) and the voltage supply (VDC) are added in a control loop (dash-dot line) between the interrogator output and the FFP filter. VDC is used to tune the FFP to the initial wavelength at the mid-reflection point when no strain is applied. The resistive voltage divider (VDIV) decreases the output voltage ($V_{out}$) to attain finer FFP tuning, while the differential amplifier (AMP) matches the TIA signal to the input voltage range of the micro-controller board. A tunable laser could be used instead of the ASE and the FFP, but common tunable lasers are usually slower in the same tunable range.

The purpose of the control loop is to keep the FFP filter tuned to the wavelength at the mid-reflection point during applied dynamic strain. The control loop is fast enough to track the low-frequency strain, but too slow to track the high-frequency strain. The micro-controller output
Read interrogator voltage $V_{\text{edge}}$

Calculate the error $\Delta V = V_{\text{edge}} - V_{\text{mid}}$

Multiply by a coefficient $C \cdot \Delta V$

Increment the output $V_{\text{out}} = V_{\text{out}} + C \cdot \Delta V$

Figure 8.4: Wavelength tracking algorithm. $V_{\text{fig:edge}}$ is the interrogator output signal, $V_{\text{mid}}$ is the voltage required to tune the FFP filter to the mid-reflection point when no strain is applied, and $V_{\text{out}}$ is the voltage required to tune FFP back to the mid-reflection point.

($V_{\text{out}}$) is therefore correlated to the low-frequency signal and measured on the second channel of the oscilloscope (C2). The control loop takes out the low-frequency content from the optical signal and the remaining high frequency content of the signal ($V_{\text{fig:edge}}$) is measured on the first channel of the oscilloscope (C1).

Figure 8.4 shows the proportional control loop functionality summarized in a simple algorithm: (1) interrogator signal is sampled, (2) error is calculated by subtracting the sampled value from the pre-set mid-reflection value, (3) error is multiplied by a scaling coefficient, (4) output voltage is incremented by the error factor, and (5) output signal is generated. The optimal value of the scaling coefficient depends on system parameters like ASE output power, EDFA gain and voltage divider ratio, but using simple proportional control loop allows easy finding of this optimum by systematically changing the coefficient until the best performance is achieved. Changing the sign of the coefficient locks the interrogator to the edge on the opposite side of the FBG spectrum.

8.3 System characterization

The system characterization is related to how well the system can lock onto the low-frequency (LF) strain. The main characteristics of the LF tuning are the tuning rate, the resolution, and the maximum tuning range.
8.3.1 Components

It is instructive to first look at the individual components. The two primary components of the system are the FFP and the micro-controller.

The filter is the FFP-TF2 fiber Fabry-Perot tunable optical filter from Micron Optics. This filter transmits a very narrow wavelength band with full-width at half-maximum of about 20 pm and full-spectrum scan-rates up to 800Hz. The transmission wavelength of the filter is controlled by the applied tuning voltage. The wavelength tuning is characterized by transmitting an ASE source broadband spectrum through the filter for a range of tuning voltages and measuring the peak wavelength of the transmitted spectrum with an optical spectrum analyser. Figure 8.6 shows that the peak wavelength changes linearly with the tuning voltage at the rate of -6.9 pm/mV.

The micro-controller board used in the setup is Arduino Due. This board is equipped with a 32-bit ARM core CPU with an 84 MHz clock, a built-in 12-bit analogue to digital converter (ADC), and a 12-bit digital to analogue converter (DAC). The reference voltage for both ADC and DAC is 3.3 V.

8.3.2 Low frequency strain sensitivity and range

Algorithm in Figure 8.4 shows that the micro-controller reads in a voltage, compares the voltage to a set value and accordingly adjusts the output voltage sent to the FFP. Setup shown in

![Figure 8.5: FFP transmission wavelength versus tuning voltage.](image-url)
Figure 8.3 shows that the read voltage depends on the photodiode (PD), amplifiers (TIA and AMP), and the ADC built into the micro-controller. The gain of the amplifiers is set in such a way so that ADC could read the entire linear range of the FBG spectrum edge. FBG used in this project has the linear-range spectral-width of about 50 pm, as shown in Figure 8.1. The resulting minimum readable change in the wavelength is thus 50 pm/2^{12} = 12 fm. The other wavelength related system parameter is the spectral width of the FFP tunable filter. The 20 pm line-width of the FFP filter does not diminish the systems ability to distinguish such small wavelength changes as long as the filter is tuned to the linear part of the FBG spectrum, however it does reduce the effective width of the linear part of the FBG edge.

Even though the system can distinguish a change in wavelength of 12 fm, this is not the sensitivity of the system, because step size of the micro-controller output is limited. The 12-bit DAC of the micro-controller produces a voltage step of $\Delta V_{DAC}=3.3 \text{ V}/2^{12} = 0.8 \text{ mV}$. The resulting wavelength step size is determined by multiplying the voltage step by the FFP slope measured in Figure 8.6. The resulting wavelength step size is $\Delta \lambda = (6.9 \text{ pm/mV}) \times (0.8 \text{ mV}) = 5.5 \text{ pm}$. This wavelength step can be converted into a strain step using the FBG conversion factor of $1.2 \text{ pm/\mu\varepsilon}$, giving a value of $4.6 \mu\varepsilon$.

This step size can be made smaller by using a voltage divider labelled VDIV in Figure 8.3. The implication of the voltage divider is that it also limits the maximum voltage output of the micro-controller, thus limiting the wavelength tuning range of the FFP and the maximum LF strain range.

Without the voltage divider, maximum tuning range is $\Delta \lambda = (6.9 \text{ pm/mV}) \times (3.3 \text{ V}) = 23 \text{ nm}$. This wavelength tuning range can be converted into a strain range of 19 m\varepsilon. This strain value is unrealistically high since optical fibers have been observed to break at as low as 6 m\varepsilon and experience plastic deformation at 4 m\varepsilon, so this maximum LF strain-range reduction is justified and doesn’t diminish the results.

Using a voltage divider with $R1 = 22 \text{ k}\Omega$ and $R2 = 1 \text{ k}\Omega$, wavelength step size of 0.24 pm is achieved, corresponding to a strain step of 0.2 \mu\varepsilon and a maximum strain range of 819 \mu\varepsilon.

The system wavelength step size is a combination of the input (ADC) and output (DAC) of the micro-controller. Since the output step size is larger than the input step size, its the output side that limits the entire system to a low-frequency strain resolution of 0.2 \mu\varepsilon.
8.3.3 Tuning rate

Similar to the strain sensitivity, the tuning rate is also a combination of the micro-controller and the FFP. Figure 8.6 shows the FFP frequency response measurement setup and results. The setup is the same as the interrogator setup in Figure 8.3, except the control loop is removed and the micro-controller with the voltage divider is used to generate a sine waveform. The sine waveform is applied to the FFP and the interrogator signal measured as the sine frequency is swept in the 10 Hz - 5 kHz range. The sine amplitude is adjusted such that at 10 Hz the FFP wavelength covers the entire linear range of the FBG reflection spectrum edge (20% - 80% reflectivity). The supply voltage (VDC) is chosen so the FFP filter is centered on the left FBG reflection spectrum edge, which in this case is for 14.192 V.

Figure 8.6b shows the normalized peak-to-peak values of the measured interrogator signal in decibels as a function of frequency. The measured data is fit to the model function in the form of the magnitude of a transfer function with one zero and one pole, as given by
\[ V_N(f) = \sqrt{\frac{1+(f/f_0)^2}{1+(f/f_1)^2}}, \]  

(8.1)

where \( V_N \) is the normalized measured voltage, \( f \) is the frequency, \( f_0=835 \) Hz and \( f_1=6.6 \) kHz are the fitting parameters.

Running a test code with and without functions that read input voltages and generate output voltages and then comparing the observed micro-controller output on the oscilloscope can be used to estimate the code execution speed of the micro-controller. Thus estimated duration of the input voltage read function is 35 \( \mu \)s and less than 5 \( \mu \)s for the output voltage generation. Since the Arduino board has a 32-bit ARM core CPU with an 84 MHz clock, the data processing is expected to be much faster than I/O operations. The total code execution time for the algorithm in Figure 8.4 can thus be estimated to be less than 50 \( \mu \)s, corresponding to a rate of 20 kHz. Since the micro-controller response is faster than the FFP, the tuning rate is limited by the FFP.

### 8.3.4 Control loop performance

The combination of the micro-controller and the filter have a 3dB corner frequency of 835 Hz, a maximum LF strain range of 819 \( \mu \varepsilon \), and a LF strain sensitivity of 0.2 \( \mu \varepsilon \). However, the system response changes when it is operated in a feedback configuration. Rather than present the feedback analysis, the system response is measured.

Figure 8.7 shows the setup used to characterize the performance of the control loop. In this setup the function generator (FGEN) and power amplifier (AMP) are driving a speaker with a sinusoid voltage. The membrane on the front end of the speaker is removed and three light suspension wires are glued to the voice coil of the speaker. The other end of these wires is then glued to the fiber on one side of the FBG. The fiber on the other side of the FBG is attached to a fixed post with another wire. The wires are glued to the points close to the FBG sensor in order to maximize the strain induced across the grating. With this setup sinusoidal strain is induced with reasonably strong magnitudes at frequencies up to 1.2 kHz. The strain amplitude is chosen to match the maximum strain range of the interrogator, which is about 40 \( \mu \varepsilon \). Such strain magnitude levels at frequencies up to 1 kHz can be achieved with a speaker by applying voltages up to 10 V.
Piezoelectric actuator isn’t used in this test, because it requires voltages from 100 V to 1000 V, or a very long stack of crystals, to achieve comparable displacement.

Figure 8.8 illustrates tracking of the strain at 40 Hz and 820 Hz. The plot shows only the FBG signal and not the feedback signal, whereby the dashed line represents the FBG signal with tracking off and the solid line represents the FBG signal with tracking on. At low frequency strain, with tracking on, the FBG measurement should ideally look flat, showing the lack of high frequency components in the signal. Figure 8.8a shows that at 40 Hz the feedback loop performs close to ideal, while at 820 Hz (Figure 8.8.b) tracking only partially flattens the signal due to the inability of the feedback loop to suppress higher frequency components. The reason why solid line in Figure 8.8.b looks noisy is because of the imperfection of the speaker setup, which generates higher order harmonics when driven at frequencies above 500 Hz, which are harder to track by the system. Besides the increased presence of higher order harmonics, the generated strain magnitude also starts falling below the pre-set initial value of 40 $\mu\varepsilon$.

In order to characterize the system performance in spite of these imperfections in the induced strain, the pass factor $P$ is defined as

$$P(f) = \left| \frac{V_{ppON}(f)}{V_{ppOFF}(f)} \right|,$$

(8.2)

where $V_{ppON}(f)$ is the fundamental frequency component of the interrogator signal at frequency $f$ when tracking is turned on, and $V_{ppOFF}(f)$ is the same when tracking is turned off.

The pass factor quantifies how much of the strain at a certain frequency is passed to the edge-filtering part of the interrogator because it isn’t compensated for by the control loop. Strain
Figure 8.8: Tracking strain at (a) 40 Hz and (b) 820 Hz. The dashed line shows interrogator signal without tracking and the solid line is with tracking.

at the frequency with pass factor of one is not tracked by the control loop at all, while strain with zero pass factor is completely tracked and therefore entirely measured by the control signal.

Figure 8.9a shows the 750 Hz strain measurement in time domain with tracking off (dashed) and on (solid). Figure 8.9b shows the peak-to-peak value of these same two signals in the frequency domain. The pass factor P is calculated by dividing these two values at the fundamental frequency.

By definition the pass factor is always positive and has a value less than one because the interrogator has a negative feedback control loop. In other words, a pass factor greater than one
Figure 8.9: (a) Interrogator signal for dynamic strain at 750 Hz with tracking on (solid) and off (dashed), and (b) frequency spectrum of the peak-to-peak value of the same signal with (solid) and without (dashed) tracking.

means the control loop increases signal variation, which is contrary to the tracking algorithm given on Figure 8.4. Figure 8.10 shows the measured values of the pass factor in the frequency range 10 Hz - 1.2 kHz. The data is fit to the model function in the form of the magnitude of a transfer function with three identical zeros and three identical poles, as given by

\[ P_{\text{fit}}(f) = \left( \frac{f_L (1 + (f/f_L)^2)}{f_H (1 + (f/f_H)^2)} \right)^{3/2}, \]  

where \( f \) is the frequency and \( f_L = 83 \) Hz and \( f_H = 780 \) Hz are the fitting parameters.

The control system is supposed to track the low frequency strain while simply not reacting to the high frequency components. To summarize the features of the system it is instructive to
look at an example. Lets assume that the FBG is exposed to a strain that has two components. The first component is the LF signal with the amplitude of 800 $\mu\varepsilon$ at the frequency of 10 Hz and the second component is the HF signal with the amplitude of 1 $\mu\varepsilon$ at the frequency of 10 kHz. The amplitude of each signal is multiplied by the pass factor for each frequency. The reduction in the amplitude for the LF signal is $P(f=10)=4.14E-4$ and for the HF signal is $P(f=1E4)=0.99$. The resulting wavelength shift for the LF signal is $\Delta \lambda=800\mu\varepsilon\times1.2\text{pm}/\mu\varepsilon\times4.14E-4=0.4\text{pm}$. The resulting shift for the HF signal is $\Delta \lambda=1\mu\varepsilon\times1.2\text{pm}/\mu\varepsilon\times0.99=1.2\text{pm}$. Both of these shifts are within the linear band of the FBG (50pm) allowing the use of edge detection demodulation technique. This makes the HF component of the edge filtering signal $3\times$ larger than the LF component even though the applied LF strain has $800\times$ larger magnitude.

### 8.4 Simultaneous low and high frequency strain measurement

Figure 8.11 shows the two independently driven actuators attached to the ends of a long acrylic board with the FBG glued to the centre of the board. LF actuation is implemented with an electromagnet and HF actuation with a piezoelectric crystal actuator; both are driven by a sinusoid voltage from function generators with power amplifiers. The FBG is glued to the board in the longitudinal direction by applying a specialized strain gauge adhesive M-Bond AE-10 at two points on the stripped fiber region right next to the grating.
8.4.1 Simultaneous HF and LF strain measurement

Figure 8.12 shows the simultaneous measurement of high amplitude sinusoidal strain at 50 Hz (LF strain) and small amplitude strain in the form of regular sinusoidal bursts at 32 kHz (HF strain). The strain is captured over two full periods of LF strain using standard edge-filtering interrogator without a control loop. The LF strain amplitude is larger than the 40 $\mu \varepsilon$ maximum strain limit of the edge-filtering interrogator. High strain sensitivity is achieved only along curve edges, so HF strain remains undetected along flat parts. Even when HF strain is detected, it is measured with varying sensitivity so the measured signal cannot be consistently converted to strain value.

Figure 8.13 shows the interrogator signal under the same simultaneous HF and LF dynamic strain, except now measured with edge-filtering interrogator with the control loop. The control loop stabilizes the interrogator signal around the mid-reflection point thereby maximizing the strain sensitivity for the entire HF measurement. Tracking the LF strain reduces variation in strain sensitivity and makes the HF measurements consistent so the strain can be consistently extracted from the interrogator measurements.

Figure 8.14 shows the extracted values of strain from the measurements of the edge-filtering interrogator with the control loop for LF strain (a) and HF strain (b). Knowing the FFP filter...
Figure 8.12: Signal measured by the edge-filtering interrogator without the control loop. The FBG is subjected to combined strain of 50 Hz LF and bursts of 32 kHz HF strain.

Figure 8.13: Simultaneous measurements of LF strain at 50 Hz and HF strain at 32 kHz. (a) The signal produced by the micro-controller after the voltage divider and (b) the signal produced by the photo detector after the amplifiers.

tuning voltage rate to be 6.9 pm/mV (Figure 8.6) and FBG strain sensitivity to be 1.2 pm/µε, the LF strain sensitivity of the control signal is calculated as 6.9/1.2=5.75µε/mV. Figure 8.14a shows the resulting LF strain obtained by multiplying the measured voltage from Figure 8.13a by the calibration factor of 5.75 µε/mV.
Figure 8.14: Simultaneous measurements of a) LF dynamic strain at 50 Hz, and b) HF dynamic strain at 32 kHz using edge-filtering interrogator with control loop.

The gain of the amplifiers is set such that the measurement range of the ADC covers the total FBG spectrum edge. In this case the slope of the FBG spectrum edge is 45.3 mV/pm, whereby the HF strain sensitivity can be calculated as \( \frac{1}{45.3 \times 1.2} = 0.0184 \mu\varepsilon/mV \). Figure 8.14b shows the resulting HF strain obtained by multiplying the measured voltage shown in Figure 8.13b by the calibration factor of 0.0184 \( \mu\varepsilon/mV \). Filtering is done with a 20 kHz high pass filter.

Figure 8.14a shows that the measured LF strain has the magnitude of about 450 \( \mu\varepsilon \), which is over 11 times greater than the maximum strain range limit imposed by the width of the linear part of the FBG spectrum edge. Figure 8.14b shows that the magnitude of the measured HF strain is about 30 times smaller than that of LF strain. The similarity of the two consecutive HF strain waveforms indicates that these indeed are the measurements of an ultrasonic mechanical waves propagating through the material. These simultaneous measurements of HF and LF strain therefore show the potential of the presented interrogator for measuring strain in situations where conventional edge filtering does not give useful data.
8.5 Summary

A solution is presented to the problem of limited dynamic range of edge-filtering fiber Bragg grating interrogation by adding a proportional control loop to the interrogator, which locks the operating wavelength to the middle reflection point on the FBG spectrum. This interrogator can separate low frequency strain components and measure them with extended dynamic range, while at the same time measuring high frequency strain without loss in strain sensitivity. This solution can be used for in-situ health monitoring of structures experiencing large slowly varying strain, such as wind turbine blades exposed to strong winds, where this strain can easily take the FBG out of the usable range.

Implementation of this kind of interrogator is described using an inexpensive micro-controller board in a proportional control loop and a narrow-linewidth tunable optical filter. The combination of the micro-controller and the filter in the tested interrogator exhibit a 3dB corner frequency at 835 Hz, maximum LF strain range of 819 $\mu \varepsilon$ and LF strain sensitivity of 0.2 $\mu \varepsilon$.

Simultaneous measurement of low-magnitude 32 kHz high-frequency strain superimposed on high-magnitude 50 Hz low-frequency strain is demonstrated. The measured low-frequency strain magnitude of 450 $\mu \varepsilon$ is over 11 times greater than the maximum allowed strain range of the standard edge-filtering method, thus exhibiting an increase in the dynamic range of the proposed interrogator. The interrogator also locks the wavelength to the mid-reflection point on the FBG spectrum and therefore preserves the high sensitivity of the high-frequency measurements.
CHAPTER 9. SUMMARY

Optical fiber electric field sensors open a new door into high electric field measurements because of their inherent immunity to electromagnetic interference and high dielectric strength. The small profile of optical fibers and electro-optic crystals, used as sensing elements, allows their placement in narrow spaces for highly localized electric field measurements. When placed within a fixed electrode structure, electric field sensors can also be calibrated to measure voltage. This allows their application in measuring high and low voltage in cables with practically no perturbation to the measured voltage. This dissertation gives a detailed account of a complete measurement system, based on slab-coupled optical fiber sensing technology, built to take advantage of this opportunity. The result is a nonperturbing measurement system with very high dynamic range.

9.1 Contributions

As mentioned in the introduction, the main contributions in this work are outlined as follows:

- I developed the SCOS nonlinear calibration method for high electric field measurement [40, 41, 45, 46, 56].

- I developed the coaxial SCOS technology for nonperturbing high voltage measurements in a coaxial cable [42–44, 46],

- I built and tested a complete measurement system for high electric fields and high voltages, and

- I developed a method to increase dynamic range of FBG edge-filtering demodulation [47, 48].

Minor contributions are:
• I contributed to the development of the side-polished PANDA-SCOS technology by helping solve fabrication challenges, advising on the change from TM to TE polarization for improved sensitivity and reduction of phase noise with APC connectors, sensor fabrication, assisting measurements and creating microscope images for publication [50],

• I helped develop, build and test the transformer-oil-resistant SCOS,

• I contributed to the development of the push-pull SCOS by helping to solve design challenges and to design, set up and take necessary measurements [51, 52, 54, 55],

• I contributed to the development of the antenna-SCOS by helping with measurements [53],

9.1.1 Nonlinear calibration

The standard SCOS calibration is linear because it utilizes only a narrow range on the spectrum resonance centered around the operating wavelength. It makes the assumption of constant sensitivity, by approximating the SCOS resonance with a tangent to the spectrum at the point where calibration measurement is taken. SCOS sensitivity, however, has two components: one that corresponds to the SCOS resonance slope and the other corresponding to the spectral shift [40, 41]. It is shown that the sensitivity component corresponding to the spectral shift can be considered constant for a much wider range of electric fields than are typically encountered in practice. The sensitivity component corresponding to the resonance slope, on the other hand, is constant only for a very narrow range of electric fields. It is shown that for values above dielectric strength of air, measurements made with linear calibration become increasingly inaccurate. Nonlinear calibration takes into account the variability of sensitivity by mapping the measured voltage to the spectrum. In this way nonlinear calibration uses the full information about the change in sensitivity as it is contained in the shape of the spectrum resonance.

9.1.2 Coaxial cable SCOS

Electric field sensors can be calibrated with a known voltage and used for voltage measurements in the case when the geometry of the system is fixed. This allows a SCOS to be inserted into a coaxial cable and used for voltage measurements [42–44, 46]. This is primarily motivated by
the nonperturbing nature of a SCOS, as it is made entirely of dielectric materials and its presence inside of an electric field minimally alters the measured field. Nonperturbing operation of the sensor is shown by computer simulation showing negligibly small values of reflections caused by the SCOS insertion. Simulation is done using HFSS software and using a lumped element transmission line model, which makes it safe to conclude that the coaxial cable SCOS concept can be used in a variety of cables without significant increase in reflections. Dielectric strength of the cable is not compromised by the SCOS insertion if proper care is taken to not introduce any impurities, especially air bubbles, into the cable.

### 9.1.3 SCOS interrogator system for measuring high electric fields

A complete measurement system for measuring high electric field and high voltage is built based on the technology of nonlinear calibration and the coaxial cable SCOS. The measurement system is designed to be immune to EMI, utilize added gain in laser optical power to increase interrogator bandwidth and be able to perform dual simultaneous measurements [45,56]. The system is tested for EMI immunity and for electric fields up to 18 MV/m and for voltages up to 150 kV. Dynamic range of the measurement system is shown to exceed 50 dB. A dedicated software application is developed with automated calibration tasks, streamlining the entire measurement process and tools for previewing calibrated data (linear and nonlinear) and post processing. Software is written entirely in MATLAB®.

### 9.1.4 Increasing dynamic range of FBG edge-filtering demodulation method

The edge filtering demodulation method of FBG strain sensors offers superb sensitivity, which is why FBG is used for detection of ultrasonic waves for structural health monitoring. However, the dynamic range of this method is very narrow. This work utilizes a negative feedback to stabilize the operating wavelength of the interrogator to the middle section of the resonance edge. A feedback loop is achieved with an Arduino micro-controller, which monitors the FBG signal in real time and locks it to the middle of the resonance edge by controlling a tunable filter, which determines the operating wavelength [47,48]. The mode-locking is successful only for slow changes because of the limited speed of the micro-controller, while the fast changes of the ultrasonic wave
are measured as usual. The feedback signal corresponds, therefore, to the slow strain that shakes the system and is measured simultaneously with the high speed ultrasonic wave. High sensitivity of the ultrasonic measurement is preserved, while the measurement of the slow strain is limited due to the micro-controller voltage resolution.

9.1.5 PANDA-SCOS

The PANDA-SCOS is manufactured by coupling electro-optic crystal to the fiber core of a side-polished PM PANDA fiber [50]. PANDA fibers have about 10 µm diameter core, which is much larger than D-shaped fibers, previously used for SCOS fabrication. This significantly reduces the power loss in the PANDA-SCOS, which results in about 10 dB improvement in off-resonance power loss. Since PANDA fibers have circular core with standardized size, it is very straightforward to splice PANDA fiber to any other standard fiber, unlike D-shaped fiber, which is very hard to splice to another fiber. This allows PANDA to be spliced to APC pigtails for reduction of phase noise and minimum loss at the connector-side. Side-polished PANDA fibers come in slow and fast axis polish option. Since standard laser outputs and commercial PM pigtails are commonly keyed to the slow axis, splicing fast axis side-polished PANDA with slow axis standardized pigtails would excite the orthogonal TM polarization of the coupled lithium niobate slab in a fabricated SCOS. However, it is shown that exciting TE polarization offers a significant improvement in sensitivity, so it is relevant to use match the polished PANDA axis or properly key the spliced pigtails and lasers for greater sensitivity.

9.1.6 Transformer-oil-resistant SCOS

The transformer-oil resistant SCOS is developed for measurement of electrical fields within transformer oil, since transformer oil proved to be very corrosive to the low-index epoxy used in the SCOS packaging. When left in transformer oil, the SCOS packaged in a regular trough loses its power and resonances in about 10 hours. The TO-SCOS is tested by continued immersion in transformer oil without losing power or resonance quality for over 4 days. The TO-SCOS is fabricated by placing a SCOS inside a glass tube and sealing it with fast setting epoxy on the ends, while the middle part around the slab is filled with low-index epoxy.
9.1.7  **Push-pull SCOS**

The push-pull SCOS technology is based on the concept of two oppositely oriented crystals placed very close to each other and subtracting the two simultaneous measurements to improve SNR \([51, 52]\). Crystals are placed very close to each other so that the localization of the field measurement is not affected. Coupling of each crystal to the core has to be done so that their resonances do not overlap. Measurement of the same excitation is measured by both crystals simultaneously and measurements are subtracted to remove the common noise, while the useful signal is being amplified. It is shown that push-pull is very effective at removing low and high-speed vibration impacts up to 10 kHz.

9.1.8  **Antenna SCOS**

A passive dipole antenna is integrated onto the SCOS trough to improve directional sensitivity of electric field measurements parallel to the fiber axis \([53]\). Computer simulation is done to show that the antenna effectively reroutes the external longitudinal field to transverse direction within the antenna gap. An electro-optic crystal with transverse sensitivity is placed in the antenna gap to benefit from this gain in the signal. Computer simulation has shown that such a dipole antenna has a flat response up to 10 MHz. It is shown by measurement that the dipole antenna rotates the direction of SCOS sensitivity from transverse to longitudinal as predicted by simulation.
REFERENCES


APPENDIX A. INTERROGATOR SOFTWARE CODE

Main file: ../interrogator.m

```matlab
% Interrogator Program Entry Point
close all;
clear all;
set(0,'DefaultFigureWindowStyle','normal')
GUI.MainGUI()
```

The root folder needs to contain audio file 'StopLN.mp3', which is played when the low-noise timer is triggered.

A.1 Algorithms

A.1.1 Classes

../+algorithms/Algorithm.m

```matlab
classdef Algorithm < interfaces.IAlgorithm
    properties (Constant)
    RoughPoints = 30; %30
    %RoughPoints needs to be at least 20 to see all resonances properly in 1530–1575nm range
    %Recommended 25
    PurePhotonic laser has a wider wavelength range so the
    %RoughPoints is better increased to 35
    max_time=45; %max time to wait after tuning [s]
end
```

```matlab
methods
    function obj = Algorithm(laserObj, daqObj, sensorObj)
        obj.laserObj = laserObj;
        obj.daqObj = daqObj;
        obj.sensorObj = sensorObj;
    end
end
```

```matlab
function resonances = GetRoughSpectrum(obj)
```

140
%returns an array of spectrums

%Calculating the wavelengths
    obj.laserObj.WavelengthStep
    wavelengths = linspace(obj.laserObj.WavelengthMin, obj.laserObj.WavelengthMax,
                          obj.RoughPoints);
else
                          laserObj.WavelengthMax);
end

voltages = zeros(1, length(wavelengths));
i = 1; time1 = tic;
clear timeCumulative;
timeCumulative = zeros(obj.RoughPoints,1);
if exist('defaultSettings.mat', 'file')
    load('defaultSettings.mat', 'WavelengthScanDuration');
    tRemaining = round((obj.RoughPoints-i)*WavelengthScanDuration);
    h_w = waitbar((i)/(obj.RoughPoints),{'Fine scan...','...','Scan %i of %i. Time remaining: %3.2f min',
                                          i, obj.RoughPoints, tRemaining/60});
else
    h_w = waitbar(0,{'Fine scan...','...','Scan %i of %i, Estimating time...','i, obj.RoughPoints});
end
set(h_w, 'pointer', 'watch');
for lambda = wavelengths
    timeI = tic;
    obj.laserObj.RequestWavelength(lambda);
    [~, v] = obj.daqObj.RequestVoltageMean(0.05);
    voltages(i) = v;
    timeCumulative(i) = toc(timeI);
    tRemaining = round((obj.RoughPoints-i)*mean(timeCumulative(1:i)));
    i = i + 1;
    if i == 23
        disp('');
    end
    if h_w.isvalid; resonances=[]; return; end
    if strcmp(h_w.BeingDeleted, 'off')
        if tRemaining>60
            waitbar((i)/(obj.RoughPoints), h_w, {'Rough scan...','...','Scan %i of %i. Time remaining: %3.2f min',
                                                i, obj.RoughPoints, tRemaining/60});
        else
            waitbar((i)/(obj.RoughPoints), h_w, {'Rough scan...','...','Scan %i of %i. Time remaining: %i s',
                                                i, obj.RoughPoints, tRemaining});
        end
    else
        return;
    end
end
close(h_w); tic(time1)
[~, locs] = findpeaks(voltages);
i = 1;
resonances = {};
for jj = locs
    spectrum = [wavelengths(i;jj); voltages(i;jj)];
    resonances{end + 1} = interfaces.Resonance(spectrum);
    i = jj;
end
if isempty(obj.sensorObj)
    sensorName = 'NoName';
else
    sensorName = obj.sensorObj.Name;
end

141
end
save ( {'WBSpectrum' datesr ( now , 'yy-mm-dd_HH-MM-SS') } , ' sensorName ' . mat );

WavelengthScanDuration = mean ( timeCumulative );
if exist (' defaultSettings.mat ' , ' file ' ) == 2
  save ( ' defaultSettings.mat ' , ' WavelengthScanDuration ' , ' -append ' );
else
  save ( ' defaultSettings.mat ' , ' WavelengthScanDuration ' );
end

end

function spectrum = GetFineSpectrum ( obj , resonance , scanResolution )
  % Capturing the Narrow Resonance Spectrum
  % Calculating the wavelengths
  if scanResolution <= obj . laserObj . WavelengthStep
    scanResolution = obj . laserObj . WavelengthStep;
    disp (' resolution too small for the laser, setting to minimum ' );
  end

  wavelengths = resonance . WavelengthMin : ( 1 e12 * scanResolution ) : resonance . WavelengthMax;
  FinePoints = length ( wavelengths );
  voltages = zeros ( 1 , length ( wavelengths ) );
  i = 1 ; time1 = tic;
  clear timeCumulative;
  timeCumulative = zeros ( FinePoints , 1 );
  if exist (' defaultSettings.mat ' , ' file ' );
    load (' defaultSettings.mat ', ' WavelengthScanDuration ' );
    h_w = waitbar ( ( i ) / ( FinePoints ) , { ' Fine scan ... ' , ....
      sprintf ( ' Scan %i of %i. Time remaining: %3.2f min ' , ....
        i , FinePoints , tRemaining / 60 ) });
  else
    h_w = waitbar ( 0 , ....
      { ' Fine scan ... ', sprintf ( ' Scan %i of %i. Estimating time ... ', i , FinePoints )} );
  end

  set ( h_w , ' pointer ' , ' watch ' );
  wvRead = zeros ( 1 , length ( wavelengths ) );
  for i = 1 : length ( wavelengths )
    time1 = tic;
    [ ~ , wvRead ( i ) ] = obj . laserObj . RequestWavelength ( wavelengths ( i ) );
    [ ~ , v ] = obj . daqObj . RequestVoltageMean ( 0.5 );
    voltages ( i ) = v;
    timeCumulative ( i ) = toc ( time1);
    tRemaining = round ( ( FinePoints – i ) / mean ( timeCumulative ( 1 : i ) ) );
    i = i + 1;
    if ~ h_w . isvalid ; spectrum = []; return ; end
    if strcmp ( h_w . BeingDeleted , ' off ' )
      if tRemaining > 60
        waitbar ( ( i ) / ( FinePoints ) , h_w , ....
          { ' Fine scan ... ', sprintf ( ' Scan %i of %i. Time remaining: %3.2f min ' , ....
            i , FinePoints , tRemaining / 60 ) });
      else
        waitbar ( ( i ) / ( FinePoints ) , h_w , ....
          { ' Fine scan ... ', sprintf ( ' Scan %i of %i. Time remaining: %i s ' , ....
            i , FinePoints , tRemaining )});
      end
    end
  end
  beep;
end

close ( h_w ) ; toc ( time1 )
if isempty ( obj . sensorObj )
  sensorName = ' NoName ' ;
end
else
    sensorName = obj.sensorObj.Name;
end

% Get rid of the duplicate values
k=1;
wvs(k)=wvRead(1);
vs(k) = voltages(1); k=k+1;
for i = 2:length(wvRead)
    dwv = wvRead(i)-wvRead(i-1);
    dV = voltages(i) - voltages(i-1);
    if dwv
        wvs(k)=wvRead(i);
        vs(k) = voltages(i);
        k=k+1;
    end
end

spectrum = algorithms.Spectrum(obj.laserObj, obj.daqObj, [wvs.*1e-9; vs]);
save(fullfile(now, 'yy-mm-dd_HH-MM-SS') '. sensorName '. mat', 'spectrum');
WavelengthScanDuration = mean(timeCumulative);
save('defaultSettings.mat', 'WavelengthScanDuration', '-append');
load defaultSettings.mat;
sound(y, Fs);
end
end
end

../+algorithms/Calibration.m

classdef Calibration
%CALIBRATION object

    properties (SetAccess = private)
        CalibrationFactorV
            % number that when multiplied by the measured sensor voltage [mV] to give the
            % measured voltage V [V]
        CalibrationFactorE
            % number that when multiplied by the measured sensor voltage [mV] to give the
            % measured electric field E [V/m]
        SensorObj
            % Sensor object used to get the calibration
        tCalibration
        vCalibration
            % t and V of the calibration measurement
        smallSignalGain
            % this value tells how much gain is added to the small signal
            % compared to the signal used to capture the spectrum
        operatingV
            % used to calculate the DC coupled data
            % DC_coupled value of the vcal used for mapping
        ACDCFlag
            % is the calibration signal AC coupled, 1 for AC, 0 for DC
        vPP
            % peak to peak calibration voltage
        vOffset
            % DC offset in calibration voltage
        lambdaPP
            % peak to peak mapped wavelength
        lambdaOffset
            % DC offset in mapped wavelength

    end

    methods
        function obj = Calibration(calV, calE, sensorObj, t, v, sGain, vDC, ...
            acdFlag, vPP, v0, lambdaPP, lambda0)
            obj.CalibrationFactorV = calV;
            obj.CalibrationFactorE = calE;
            obj.SensorObj = sensorObj;
    end
classdef Results
  % RESULTS — results object
  properties (SetAccess = private)
    filename; % filename with the measured data
    t; % time vector
    v; % measured voltage data
    sGain; % small signal gain
    operatingV; % the DC offset at the operating wavelength
    operatingLambda; % operating wavelength at which the measurements were taken
  end
  methods
    function obj = Results(filename, t, v, sGain, operatingV, operatingLambda, ...
        vLin, vNonLin, eLin, eNonLin)
      obj.filename = filename;
      obj.t = t; obj.v = v;
      obj.sGain = sGain;
      obj.operatingV = operatingV;
      obj.operatingLambda = operatingLambda;
      obj.vLin = vLin; obj.vNonLin = vNonLin;
      obj.eLin = eLin; obj.eNonLin = eNonLin;
    end
  end
end

../+algorithms/Spectrum.m

classdef Spectrum < interfaces.ISpectrum
  properties (SetObservable)
    RightLeft
    % variable which has ones for points that belong on the right spectrum edge and zeros for left edge
  end
  properties (Access = private)
    operatingLambda; % wavelength at max sens spot
    voltageMaxSensSpectrum; % voltage at max sens spectrum
    calibration; % calibration factor
  end
  lasObj;
function obj = Spectrum(lasObj, daqObj, spectrum)
    addlistener(obj, 'OperatingLambda', 'PostSet', @obj.onOpLambdaChanged);
    obj.Spectrum = spectrum;
    obj.lasObj = lasObj;
    obj.daqObj = daqObj;
    [maxR, ~, ~] = algorithms.AnalyzeSpectrum(obj.Spectrum);
    obj.LambdaMaxSensitivity = maxR(1);
    obj.VMaxSensitivity = ... 
    algorithms.MapThisLambda(obj.LambdaMaxSensitivity, obj.Spectrum);
    if lasObj.State == interfaces.LaserState.Off
        obj.OperatingLambda = obj.LambdaMaxSensitivity;
    else
        obj.OperatingLambda = obj.lasObj.Wavelength;
    end
end

function calObj = Calibrate(obj, tCal, vCal, sGain, appliedVoltage, sensor)
    oldPointer = obj.startWatchCursor();
    EShiftSI = sensor.EShift*1e-12*1e-6; % E shift converted to m/(V/m)
    [~, ~, acdcFlag] = algorithms.findPeakToPeak(tCal, vCal);
    if acdcFlag
        disp('AC coupled');
        vCal = vCal + obj.OperatingV;
    else
        disp('DC coupled');
    end
    % Find linear calibration coefficient for Voltage
    [vCalSCOSPP, vCalSCOS0, acdcFlag] = algorithms.findPeakToPeak(tCal, vCal);
    disp('Calculate Offset: %3.3f mV, operating Voltage: %3.3f', ...
         1e3*vCalSCOS0, 1e3*obj.OperatingV);
    calFactorV = appliedVoltage/vCalSCOSPP;
    fprintf('V Calibration ratio is %2.3f V/m\n\n', 1e-3*calFactorV);
    % Find linear calibration coefficient for E field
    lambdaCal = algorithms.MapThisV(vCal, obj.Spectrum);
    [lambdaCalPP, lambdaCal0] = algorithms.findPeakToPeak(tCal, lambdaCal, 2);
    calFactorE = (lambdaCalPP/EShiftSI)/vCalSCOSPP;
    fprintf('E Calibration ratio is %2.3f (kV/m)/m\n\n', 1e-6*calFactorE);
    calObj = algorithms.Calibration(calFactorV, calFactorE, sensor....
                                     tCal, vCal, sGain, obj.OperatingV, acdcFlag, vCalSCOSPP, vCalSCOS0....
                                     lambdaCalPP, lambdaCal0);
    obj.endWatchCursor(oldPointer);
end

function resultsObj = ProcessData(obj, t, v, sGain, fileN, sensor, ... rightEdgeMapping)
    switch nargin
        case 6
            rightEdgeMapping = ones(length(v),1);
            % 1 for right edge
            % 0 for left edge
        end
    oldPointer = obj.startWatchCursor();
E_{ShiftSI} = \text{sensor}.E_{Shift}*1e^{-12}*1e^{-6}; \text{ # E shift converted to } m/(V/m)

[\sim, \sim, \text{acdeFlag}] = \text{algorithms}.\text{findPeakToPeak}(t, v);

\%Linear V
\begin{align*}
\text{disp( sprintf(}'\text{operating voltage is } %3.3f \text{ mV', 1e3.*obj.OperatingV));}
\end{align*}
\begin{align*}
v_{\text{Shifted}} &= v - \text{obj.OperatingV};
\end{align*}
\begin{align*}
v_{\text{Lin}} &= v_{\text{Shifted}}.*\text{obj.Calibration}.\text{CalibrationFactorV};
\end{align*}

\%Linear E
\begin{align*}
e_{\text{Lin}} &= v_{\text{Shifted}}.*\text{obj.Calibration}.\text{CalibrationFactorE};
\end{align*}

\%NonLinear E
\begin{align*}
\text{if } \text{acdeFlag} \\
v_{\text{Shifted}} &= v + \text{obj.OperatingV};
\end{align*}
\begin{align*}
\lambda_{\text{Mapped}} &= \text{algorithms}.\text{MapThisV}(v_{\text{Shifted}}, \ldots \text{obj.Spectrum}, \text{rightEdgeMapping});
\end{align*}
\begin{align*}
\text{end}
\lambda_{\text{MappedShift}} &= \lambda_{\text{Mapped}} - \text{obj.OperatingLambda};
\end{align*}
\begin{align*}
e_{\text{NonLin}} &= \lambda_{\text{MappedShift}}/E_{\text{ShiftSI}};
\end{align*}

\%NonLinear V
\begin{align*}
E_{\text{VC}} &= \text{obj.Calibration}.\text{CalibrationFactorV}/\text{obj.Calibration}.\text{CalibrationFactorE};
\end{align*}
\begin{align*}
v_{\text{NonLin}} &= e_{\text{NonLin}}.*E_{\text{VC}};
\end{align*}

\begin{align*}
[\sim, \text{fileName, } \sim] &= \text{fileparts}(\text{fileName});
\end{align*}
\begin{align*}
\text{resultsObj} &= \text{algorithms}.\text{Results}(\text{fileName}, t, v, \text{sGain}, \ldots \text{obj.OperatingV}, \text{obj.OperatingLambda}, v_{\text{Lin}}, v_{\text{NonLin}}, e_{\text{Lin}}, e_{\text{NonLin}});
\end{align*}
\begin{align*}
\% \text{Saving all the calibrated data}
\end{align*}
\begin{align*}
\text{newFolder} &= '\text{\dataFiles}';
\end{align*}
\begin{align*}
\text{if } \text{exist(newFolder)}; \text{mkdir(newFolder); end}
\end{align*}
\begin{align*}
\text{save}([\text{newFolder fileN datesStr( now, 'yy-mm-dd_HH-MM-SS')}', '_calibrated.mat'], \ldots'
\end{align*}
\begin{align*}
't', 'e\text{Lin}', 'v\text{Lin}', 'e\text{NonLin}', 'v\text{NonLin}');
\end{align*}
\begin{align*}
\text{obj.\text{endWatchCursor}(oldPointer)};
\end{align*}
\begin{align*}
\end{align*}
\begin{align*}
\text{methods (Access = private)}
\end{align*}
\begin{align*}
\text{function onOpLambdaChanged(obj, \sim, \sim)}\end{align*}
\begin{align*}
\text{if obj.OperatingLambda == obj.LambdaMaxSensitivity;}
\end{align*}
\begin{align*}
\text{obj.OperatingV = obj.VMaxSensitivity;}
\end{align*}
\begin{align*}
\text{else}
\end{align*}
\begin{align*}
\text{if obj.OperatingLambda >=obj.LambdaMin && obj.OperatingLambda==obj.LambdaMax}
\end{align*}
\begin{align*}
\text{obj.OperatingV = ...}
\end{align*}
\begin{align*}
\text{algorithms}.\text{MapThisLambda}(\text{obj.OperatingLambda}, \text{obj.Spectrum});
\end{align*}
\begin{align*}
\text{else}
\end{align*}
\begin{align*}
[\text{daqOK, obj.OperatingV}] &= \text{obj.daqObj.RequestVoltage};
\text{assert(daqOK, 'warning: voltage could not be read on the DAQ');}
\end{align*}
\begin{align*}
\end{align*}
\begin{align*}
\text{end}
\end{align*}
\begin{align*}
\text{function oldPointer = startWatchCursor(\sim)}
\end{align*}
\begin{align*}
\text{fHandle} &= \text{gcf};
\end{align*}
\begin{align*}
\text{oldPointer} &= \text{get(fHandle, 'Pointer')};
\end{align*}
\begin{align*}
\text{set(fHandle, 'Pointer', 'watch');}
\end{align*}
\begin{align*}
\text{end}
\end{align*}
\begin{align*}
\text{function endWatchCursor(\sim, oldPointer)}
\end{align*}
\begin{align*}
\text{fHandle} &= \text{gcf};
\end{align*}
\begin{align*}
\text{set(fHandle, 'Pointer', oldPointer)};
\end{align*}
\begin{align*}
\text{end}
\end{align*}
A.1.2 Functions

../algorithms/CaptureSpectrum.m

function VSpectrum = CaptureSpectrum ( lambdaGrid, laserSettings, daqSessionObj)

% Convert wavelength grid to frequency grid
fGrid = algorithms.LambdaToF ( lambdaGrid);

% Mapping the frequency grid to Fine Tuning slots
[fGridSlotted, fGridSlottedFineTuning, fFineTuningSlotsGridCenters] =
    algorithms.SlotTheGridForFineTuning (fGrid, laserSettings);
if abs (max(max(fGridSlottedFineTuning)))>12e9; warning ('Fine Tuning grid exceeds limits'); end

% Tuning and measuring voltage loop
h_w = waitbar (0, 'wait ...');
nn=1;
VSpectrum = zeros (size (fGrid));
T= all (isnan (fGridSlottedFineTuning), 1);
T((T==0))=[];
if length(T)==1 % Only Rough Tuning
    for kk =1:size (fGridSlotted, 1)
        waitbar (kk/size (fGridSlotted, 1), h_w); drawnow;
        % Set frequency
        % fGridSlotted (kk), the middle frequencies
        % CBMX_set_port_freq (laserSettings.serialObject, laserSettings.Port.1e-12.*
        % fGridSlotted (kk));
        algorithms.SetFreq (laserSettings.serialObj, fGridSlotted (kk)); % Laser
        pause (1); algorithms.wait_tuning (laserSettings.serialObject, laserSettings.Port, max_time);
        % Read the voltage from DAQ
        data = daqSessionObj.startForeground;
        VSpectrum (nn)=mean (data); nn=nn+1;
        fprintf ('%i /%i : Wavelength : %4.3f nm
', kk, size (fGridSlotted, 1), 1e9.*FToLambda (fGridSlotted (kk)));
        wait_tuning (laserSettings.serialObject, laserSettings.Port, max_time);
        pause (2);
    end
else % Rough and Fine Tuning
    disp ('Capturing spectrum with fine tuning ');
    for kk=1:size (fGridSlotted, 2)
        waitbar (kk/size (fGridSlotted, 2), h_w); drawnow;
        if isnan (fGridSlotted (1,kk))
            % Set frequency to the center frequency of the optical mode
            CBMX_set_port_config (laserSettings.serialObject, laserSettings.Port, ...
            le-12.*fGridSlotted (kk), round (le-9.*fGridSlottedFineTuning (kk, 1), 3), ...
            laserSettings.Properties.conf.read(3), laserSettings.Properties.conf.read(4);
            fprintf ('%.0f of %.0f : Wavelength : %4.3f mm\n', nn,...
            size (fGrid, 2), 1e9.*algorithms.FToLambda (fGridSlotted (kk)+
            fGridSlottedFineTuning (kk, 1)));
            wait_tuning (laserSettings.serialObject, laserSettings.Port, max_time);
            pause (2);
        end
        % Read the voltage from DAQ
        data = daqSessionObj.startForeground;
        VSpectrum (nn)=mean (data); nn=nn+1;
        % Fine tune to all frequencies within the same mode
        M=fGridSlottedFineTuning (kk,:);
        for nm=2:length (M)
            if isnan (M(nm))
                % Fine tune to mn-th frequency within the same optical mode
                % How fr left or right with in a mode you are
            end
        end
    end
%CBMX_set_port_FTF(laserSettings.serialObject, laserSettings.Port, round(1e-9.*
            fGridSlottedFineTuning(kk.mm,.3));
      algorithms.SetFreq(obj,fGridSlottedFineTuning(kk.mm,.3));
      fprintf(’%.0f of %.0f : Wavelength: %.4f nm’n’, nn, ...
                size(fGrid.2), 1e9.+algorithms.FToLambda(fGridSlotted(kk)+
                fGridSlottedFineTuning(kk.mm)));
      algorithms.wait_tuning(laserSettings.serialObject, laserSettings.Port, 
          max_time);
      %read the voltage from DAQ
      data = daqSessionObj.startForeground;
      VSpectrum(nn)=mean(data); nn=nn+1;
      elseif ~all(isnan(M(mm: end)))
      warning(’Fine Tuning Grid Error – not all tailing NaNs’);
      end
      end
  end
end
end
end
end
close(h_w);

../+algorithms/DBmToMW.m

function PmW = DBmToMW(PdBm)
  PmW = 10^(dBm./10);

../+algorithms/FindResonance.m

function resonance = FindResonance(lambdaInt, VInt)
  % Creates a user dialog to select portions of resonances. Arguments are
  % the lambda spectrum and the voltage spectrum, all with matched
  % indices.
  VInt = 1e3.*VInt;
  resonances = [ ];
  threeDB = max(VInt) * 0.5; % 3dB threshold
  [highs, iHigh] = findpeaks(VInt);
  [lows, iLow] = findpeaks(-VInt, ’MinPeakHeight’, -threeDB);
  % find the following pattern: A max followed by a min no larger than
  % the 10% of the max followed by another max within 10% of the first
  % max
  percentageWidening = 0.1;
  for i = iLow
    leftMaxes = iHigh(iHigh < i);
    rightMaxes = iHigh(iHigh > i);
    if (length(leftMaxes) < 1 || length(rightMaxes) < 1)
        continue
    end
    leftMax = leftMaxes(end);
    rightMax = rightMaxes(1);
    width = lambdaInt(rightMax) – lambdaInt(leftMax);
    set = [lambdaInt(leftMax)–percentageWidening*width, lambdaInt(i),
           lambdaInt(rightMax)+percentageWidening*width ];
    resonances = [ resonances set ’];
  end
  resonance = struct ([]);
  top = max(VInt);
  bottom = min(VInt);
  f = figure;
plot(lambdaInt, VInt);
names = {};
for r = resonances
    rectangle('Position', [r(1) bottom r(3)-r(1) top-bottom], 'Curvature', [0.5, 0.5])
    name = num2eng(r(2));
    text(r(1), bottom, name, 'VerticalAlignment', 'top');
    names{end+1} = name;
end
if isempty(names)
    messagebox('No resonances were found');
else
    [selection, ok] = listdlg('PromptString', 'Select a resonance:', ....
        'SelectionMode', 'single', ....
        'ListString', names)
    if ok
        resonance = struct('Left', resonances(1, selection), ....
            'Center', resonances(2, selection), ....
            'Right', resonances(3, selection));
    end
end

../+algorithms/FToLambda.m

function lambda = FToLambda (f)
lambda = 299792458./f;
if length(f)>1; disp('Warning frequency should be a scalar, not vector');end

../+algorithms/isfread.m

function data = isfread (pattern)
%ISFREAD Read a Tektronix ISF file.
% DATA = ISFREAD(FILENAME) reads the Tektronix ISF file given by FILENAME
% and returns a structure containing the data and the header.
% DATA = ISFREAD(DIRECTORY) reads all files with the extension '.ISF' or
% '.isf' in the specified directory and returns a structure array
% containing the data and the headers. If n files are read, the structure
% array will have size 1xn.
% DATA = ISFREAD(PATTERN) reads the files matched by the specified
% PATTERN. PATTERN can be a wildcard, such as '*.isf'. It returns a
% structure array containing the data and the headers. There is one
% element for each file read. If n files are read, the structure array
% will have size 1xn.
% ISF is the name of an internal data format used by Tektronix
% oscilloscopes. The files consist of a short text header followed by
% binary data.
% This function was written to read data saved by a model TDS 360
% oscilloscope, but is likely to work with other models that use the ISF
% format. It was tested by saving the same waveform on the oscilloscope
% in both comma-separated values (CSV) format and ISF, then comparing CSV
% data to that returned by ISFREAD.
% The DATA structure contains the following fields:
% filename
% datetime
30 % header
31 % x
32 % y
33 % The field datetime is file's modification date and time. This date and
34 % time may originate from the oscilloscope's internal clock, or from a
35 % computer subsequently used to copy the files. The value will be
36 % locale-dependent.
37 %
38 % EXAMPLES:
39 % 1. data = isfread('TEK00000.ISF');
40 % plot(data.x,data.y);
41 %
42 % 3. data = isfread('*.ISF');
43 % plot([data.x], [data.y]);
44 % legend([data.datetime]);
45 %
46 % This function is based on <a href="http://www.mathworks.com/matlabcentral/fileexchange/6247">isfread</a> by
47 % John Lipp.
48 %
49 % See also
50 % Programmer Manual for Tektronix TDS 340A, TDS 360 and TDS 380 Digital
51 % Real-Time Oscilloscopes (070–9442–02).
52 %
53 % Iain Robinson, School of Geosciences, University of Edinburgh.
54 % Last updated: 2012–09–11
55
56 if ~exist( 'pattern', 'var' )
57     error('No file name, directory or pattern was specified.');
58 end
59
60 % Check whether pattern is a folder.
61 if ( exist( pattern , 'dir' ) )
62     folder = pattern;
63     % Get a list of all files that have the extension '.isf' or '.ISF'.
64     files = [ dir( fullfile( folder , '*.ISF') ); dir( fullfile( folder , '*.isf') ) ];
65 else
66     % The pattern is not a folder, so must be a file name or a pattern with
67     % wildcards (such as 'Traces/TEK0*.ISF').
68     [ folder , '*' ] = fileparts( pattern );
69     % Get a list of all files and folders which match the pattern...
70     filesAndFolders = dir(pattern);
71     % ...then exclude the folders, to get just a list of files.
72     files = filesAndFolders( [filesAndFolders.isdir] );
73 end
74
75 % Get the file names (without folders), and modification datetimes.
76 fileNames = { files.name };
77 datetimes = { datesstr([files.datenum]) };
78
79 if numel(fileNames)==0
80     error('The pattern did not match any file or files: %s', pattern);
81 end
82
83 for s=1:numel(fileNames)
84     fileName = fileNames{s};
85     fullFileName = fullfile(folder, fileName);
86
87     % Check the file exists.
88     if ( ~exist(fullFileName, 'file') )
89         error('The file does not exist: %s', fullFileName);
90     end
91
92     % Open the file.
93     fid = fopen( fullFileName, 'r' ) ;
94     if ( fid == -1 )
95         error('The file exists, but could not be opened: %s', fullFileName);
96 end
97
98 150
% Read the text header into a variable called 'h'. The loop reads the
% file character-by-character into h until h finishes with the
% characters ':CURVE #' or ':CURV #'.
% h = '';
% while ( isempty( regexp(h, ':CURVE? #', 'once') ) )
%   % If the end of the file has been reached something is wrong.
%   if (feof(fileID) )
%     error( 'The end of the file %s was reached whilst still reading the header. This
%           suggests that it is not a Tektronix ISF file.', fileName);
%   end
%   c = char( fread(fileID, 1, 'char'));
%   h = [h, c];
end

% Parse the header
header.BYT_NR = str2double( regexp(h, 'BYT_NR?\s+\(\d+\)', 'once', 'tokens') ); % Binary field
  data width
header.BIT_NR = str2double( regexp(h, 'BIT_NR?\s+\(\d+\)', 'once', 'tokens') ); % Number of
  bits per binary waveform point
header.ENCDG = char( regexp(h, 'ENCDG?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); % Type of
  encoding for the waveform data
header.BN_FMT = char( regexp(h, 'BN_FMT?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); % Format of
  the binary data
header.BYT_OR = char( regexp(h, 'BYT_OR?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); % Byte order
  of the binary data
header.WFID = char( regexp(h, 'WFID?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); % Waveform
  identifier
header.NR_PT = str2double( regexp(h, 'NR_PT?\s+\(\d+\)', 'once', 'tokens') ); % Number of
  waveform points
header.PT_FMT = char( regexp(h, 'PT_FMT?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); % Point
  format: 'ENV' for envelope pairs or 'Y'
header.XUNIT = char( regexp(h, 'XUNIT?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); % Horizontal
  units
header.XINCR = str2double( regexp(h, 'XINCR?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); %
  Horizontal sampling interval
header.PR_OFF = str2double( regexp(h, 'PR_OFF?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); %
  Trigger point within the waveform record
header.YUNIT = char( regexp(h, 'YUNIT?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); % Vertical
  units of waveform data
header.YMULT = str2double( regexp(h, 'YMULT?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); %
  Vertical scale factor
header.YOFF = str2double( regexp(h, 'YOFF?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); % Offset
  of the vertical component
header.YZERO = str2double( regexp(h, 'YZERO?\s+\(.*?\)\s+\[.\:\]', 'once', 'tokens') ); %
  Offset

% In addition, some header fields are described in the Programmer
% Manual, but do not seem to appear in any of my files: XMULT, XOFF,
% XZERO, ZMULT, ZOFF, ZUNIT and ZZERO.

% Check that at least some part of the header was parsed.
if isempty( header.BYT_NR )
  warning( 'Failed to read some part of, or possibly all of, the header in the file %s.',
            fileName);
end

% The next few characters in the file give the number of bytes in the
% waveform data. The first digit, referred to as 'x' on page 2–60 of
% the Programmer Manual, gives the number of bytes that immediately
% follow giving the value 'y', where 'y' is the number of bytes in the
% waveform. The manual explains it better than I can.
xBYTES = str2double( char( fread(fileID, 1, 'char') ) );
yBYTES = str2double( char( fread(fileID, xBYTES, 'char') ) );

% For some reason there is an offset of 1 byte in reading the data
% files. I don't know why, but I found I could fix it by moving the
% file position back by one byte.
fsseek(fileID, -1, 'cof');

% Read the waveform.
% For some oscilloscopes it may be necessary to add 'ieee–be' to the
% fread statement below. See the comments here:
[f binaryData, count] = fread(fileID, yBytes/2, 'int16');

% Check that the expected number of points have been read.
if (count ~= header.NR_PT)
    error('According to the header, the file %s contains %d points, but only %d were read.',
         fileName, header.NR_PT, count);
end

% Check that there is no leftover data. I found that there generally
% is.
if (feof(fileID))
    warning('All expected data was read from %s, but there still appears to be data remaining
            from the header.', fileName);
end

% Calculate the horizontal (x) and vertical (y) values. These equations
% are given on page 2–171 of the Programmer Manual.
n = (1:header.NR_PT)';
x = header.XINC * (n - header.PT_OFF);
y = header.YZERO + header.YMULT * (binaryData - header.YOFF);

% Close the file.
fclose(fileID);

% Copy the data into a structure.
data(s).filename = fileNames{s};
data(s).datetime = datetimes(s,:);
data(s).header = header;
data(s).x = x;
data(s).y = y;
end
end

../+algorithms/LambdaToF.m

function f = LambdaToF (lambda)

f = 299792458./lambda;

../+algorithms/LaserOFF.m

function [] = LaserOFF (ser_obj, Port)

% turning off the laser
CBMX_set_port_state(ser_obj, Port, 0); % switch port off
pause(0.5);
% closing the laser port
fclose(ser_obj);
function [ser_obj, Port, port] = LaserON (com_port)

delete (instrfind); % first clean up before doing anything
% Parameters
% look up serial port number in Device Manager of Windows
ser_obj = serial (com_port,'BaudRate',115200);
% use the following line to access chassis via Ethernet/Telnet and comment
% out the command for serial connection
% ser_obj = tcpip (IP_address, 10001); % open telnet connection via Ethernet
ptime = 2; % pause time used after setting a value
% enter here ports to test
Port = [1 1 1]; % do not use wildcards here
% note: Echo needs to be off to make code function

% Opening communication port
clear port;
fo pen (ser_obj); % open serial port
fprintf (ser_obj, ';'); % flush of Laser chassis buffer by sending command termination

% Checking the temperature
disp ('testing monitor values');
port.wav.mon = CBMX_query_monitor_values (ser_obj, Port);
fprintf (' LaserDiode chip temp: %2.2f C, LaserDiode base temp %2.2f C, LaserDiode current %3.1f mA, TEC current %3.1f mA
', port.wav.mon);
% Reading laser Power limits
port.power.lim = CBMX_query_port_power_limit (ser_obj, Port);
% Reading laser Frequency limits
port.frequency.lim = CBMX_query_port_frequency_limit (ser_obj, Port);
lambdaCBand = 299792458./(flipr (port.frequency.lim).*1e12);
% Reading laser Wavelength limits
port.wav.lim = CBMX_query_port_wavelength_limit (ser_obj, Port);

function laserStatus = LaserStatus (ser_obj, Port)
laserStatus = CBMX_query_port_config (ser_obj, Port);
fprintf ('Frequency: %2.4f THz
', laserStatus (1));
fprintf ('FTF: actual: %2.3f GHz
', laserStatus (2));
fprintf ('Power: %2.2dBm
', laserStatus (3));
fprintf ('Output state : %d, Tuning state : %d, Dither state : %d
', laserStatus (4:end));

function [lambdaInt, vInt] = LoadSpectrumResonanceNB (resonance, lambdaStep, laserSettings,daqSessionObj)
% Loads a resonance value by either capturing a new spectrum or loading
% the last spectrum

function [lambdaGridNB, VSpectrumNB] = get_spectrum ()
% Single Resonance Scan
% Scan Parameters
lambdaGridNB = resonance.Left:lambdaStep:resonance.Right;
% Time the spectrum capture function execution

```matlab
t1 = clock; t1 = t1([4 5 6]);
V Spectrum NB = algorithms.Capture Spectrum (lambda Grid NB, laser Settings, daq Session Obj);
t2 = clock; t2 = t2([4 5 6]);
dt = t2 - t1;
if dt(2) < 0; dt(2) = 60 + dt(2); dt(1) = dt(1) - 1; end;
if dt(3) < 0, dt(3) = 60 + dt(3); end;
disp(sprintf('Time elapsed %02.0f:%02.0f:%02.0f (hrs:min:sec)', dt));
```

```matlab
dt = t2 - t1;
if dt(2) < 0; dt(2) = 60 + dt(2); dt(1) = dt(1) - 1; end;
if dt(3) < 0, dt(3) = 60 + dt(3); end;
disp(sprintf('Time elapsed %02.0f:%02.0f:%02.0f (hrs:min:sec)', dt));
```

```matlab
dt = t2 - t1;
if dt(2) < 0; dt(2) = 60 + dt(2); dt(1) = dt(1) - 1; end;
if dt(3) < 0, dt(3) = 60 + dt(3); end;
disp(sprintf('Time elapsed %02.0f:%02.0f:%02.0f (hrs:min:sec)', dt));
```

```matlab
[lambda Int, v Int] = getspectrum();
```

```matlab
dt = t2 - t1;
if dt(2) < 0; dt(2) = 60 + dt(2); dt(1) = dt(1) - 1; end;
if dt(3) < 0, dt(3) = 60 + dt(3); end;
disp(sprintf('Time elapsed %02.0f:%02.0f:%02.0f (hrs:min:sec)', dt));
```

```matlab
[lambda Int, v Int] = getspectrum();
```

```matlab
t1 = clock; t1 = t1([4 5 6]);
V Spectrum WB = algorithms.Capture Spectrum (lambda Grid WB, laser Settings , daq Session Obj);
t2 = clock; t2 = t2([4 5 6]);
dt = t2 - t1;
if dt(2) < 0; dt(2) = 60 + dt(2); dt(1) = dt(1) - 1; end;
if dt(3) < 0, dt(3) = 60 + dt(3); end;
disp(sprintf('Time elapsed %02.0f:%02.0f:%02.0f (hrs:min:sec)', dt));
```

```matlab
[lambda Int, v Int] = getspectrum();
```

```matlab
beep;
end
[lambda Int, v Int] = getspectrum();
```

```matlab
function resonance = LoadSpectrumResonanceWB (lambda Step, laser Settings, daq Session Obj)
% Loads a resonance value by either capturing a new spectrum or loading % the last spectrum
```

```matlab
filename = 'SpectrumWideband.mat';
load('SpectrumWideband.mat')
function [lambda Grid WB, VSpectrum WB] = getspectrum()
if exist(filename, 'file') == 2
    choice = questdlg('Would you like to load previous spectrum, saved one or create a new one? ....
    ' Spectrum Selection ....
    'Load Previous', 'Load Saved', 'Create New', 'Load Previous');
    switch choice
    case 'Load Previous'
        vars = load (filename);
        lambda Grid WB = vars.lambda Int;
        VSpectrum WB = vars.v Int;
        return
    case 'Load Saved'
        filename = uigetfile;
        load (filename);
        lambda Grid WB = vars.lambda Int;
        VSpectrum WB = vars.v Int;
        return
    end
    Properties.wav.lim(2);
    t1 = clock; t1 = t1([4 5 6]);
    VSpectrum WB = algorithms.Capture Spectrum (lambda Grid WB, laser Settings, daq Session Obj);
    t2 = clock; t2 = t2([4 5 6]);
    dt = t2 - t1;
    if dt(2) < 0; dt(2) = 60 + dt(2); dt(1) = dt(1) - 1; end;
    if dt(3) < 0, dt(3) = 60 + dt(3); end;
    disp(sprintf('Time elapsed %02.0f:%02.0f:%02.0f (hrs:min:sec)', dt));
    end
    [lambda Int, v Int] = getspectrum();
    beep;
    hold figure('PaperUnits', 'inches'); pos = get(h0, 'PaperPosition');
    w = 3.5; set(h0, 'PaperPosition', [pos(1), pos(2) + w, pos(3), pos(3) + w]);
    plot(1e9 + lambda Int, 1e3 + v Int);
    xlabel('Wavelength [nm]', 'FontSize', 12); ylabel('V [mV]', 'FontSize', 12);
    axis tight; grid on
    proxy = 1e3 + v Int; margin1 = 0.1; margin2 = 0.1;
    ylim([min(proxy) - margin1 .* (max(proxy) - min(proxy)) max(proxy) + margin2 .* (max(proxy) - min
    (proxy))]);
```

154
    % Returns indices between which the spectrum is monotonous

    % Find the bottom
    [botPks indBot] = findpeaks(-vRes, 'SortStr', 'descend', 'NPeaks', 1);
    lambdaBottom = lambdaRes(indBot);

    % Trim the Right edge
    for kk = (indBot + floor(length(vRes)/50)) : (length(vRes) - 1)
        if vRes(kk+1) - vRes(kk) < 0
            indCropB = kk; break
        end
    end

    % Trim the Left edge
    for kk = (indBot - floor(length(vRes)/50)) : -1:1
        if vRes(kk+1) - vRes(kk) > 0
            indCropA = kk; break
        end
    end

    vResTrimmed = vRes(indCropA : indCropB);
    lambdaResTrimmed = lambdaRes(indCropA : indCropB);

    h0 = figure('PaperUnits', 'inches'); pos = get(h0, 'Position');
    w = 3.5; set(h0, 'Position', [pos(1) pos(2) w pos(3)/pos(3)*w]);
    plot(1e9.*lambdaResTrimmed, 1e3.*vResTrimmed);
    grid on
    xlabel('Wavelength [nm]', 'FontSize', 12); ylabel('V [mV]', 'FontSize', 12);
    title('Monotonous part of the spectrum');

function vMapped = MapThisLambda(lambda0, spectrum)
    % MAPTHIS maps the points in the vector lambda
    % to corresponding voltage vector using
    % the spectrum lambda, spectrumVoltage

    lambda = spectrum(:, :);
    spectrumVoltage = spectrum(2, :);

    % by default the first point is found on the right side of the resonance

    % find the splitting point between left and right edge
    indMid = find(min(spectrumVoltage) == spectrumVoltage);
    assert(length(indMid) == 1, 'multiple minimum points in the spectrum found');

    % initialize the lambdaMapped value to all-zeros
    vMapped = zeros(size(lambda0));

    % check the segment for monotonicity before interpolation
    dlambda = diff(lambda);
    if all(dlambda > 0) || all(dlambda < 0)
        % map the lambda0
        vMapped = interp1(lambda, spectrumVoltage, lambda0);
    else
        warnldg('Mapping non-monotonic vector');
    end
function lambdaMapped = MapThisV (v, spectrum, rightleft)
% MAPTHIS maps the points in the vector v to the corresponding wavelength vector using
% the spectrum lambda and spectrumVoltage.
% By default the first point is found on the right side of the resonance
switch margin
  case 2
    rightleft = ones(length(v),1);
    % right edge for a value 1
    % left edge for value of 0
  end
oldPointer = startWatchCursor();
lambda = spectrum(1,:);
spectrumVoltage = spectrum(2,:);
% find the splitting point between left and right edge
indMid = find(spectrumVoltage==min(spectrumVoltage));
assert (length(indMid)==1, 'multiple minimum points in the spectrum found');
% find the maxima on the left and right and cut them off
indLeft = max(find(spectrumVoltage(1:indMid-1)>... spectrumVoltage(indMid)+0.9*(max(spectrumVoltage(1:indMid-1))-spectrumVoltage(indMid)) ) );
indRight = indMid - 1 +... min(find(spectrumVoltage(indMid:end)>... spectrumVoltage(indMid)+0.9*(max(spectrumVoltage(indMid:end))-spectrumVoltage(indMid)) ) );
% forming the cropped left and right edge of the spectrum  
% only non-monotonous parts on the ends of the resonance are cut off, not in the middle
spectrumLeft = spectrumVoltage(indLeft:indMid-1);
lambdaLeft = lambda(indLeft:indMid-1);
spectrumRight = spectrumVoltage(indMid:indRight);
lambdaRight = lambda(indMid:indRight);
smtValues = 1:2:300;
% initialize the lambdaMapped value to all-zeros
lambdaMapped = zeros(size(v));
% find the blocks of zeros and ones in the rightleft vector
blocks = diff(rightleft);
blocksIndices = find(blocks~=0);
blocksIndices = [1 (blocksIndices'+1); blocksIndices' length(v)];
% blocksIndices columns are the end-indices of each interchanging 1 and 0-block
h_w = waitbar(0, 'wait...');
set(h_w, 'pointer', 'watch');
oldPointer = startWatchCursor();
tic
i0 = size(blocksIndices, 2);
nonmonotonic = 1; smt = 1;
for i=1:i0
  realIndex = blocksIndices(i, 2);
vBlock = v(blocksIndices(1, i):blocksIndices(2, i));
if rightleft(blocksIndices(1, i))
  % look on the right edge
  k = 1;
  while nonmonotonic
    spectrumRight = smooth(spectrumRight, smtValues(k));
    nonmonotonic = ~ all(diff(spectrumRight)>0);
    k = k + 1;
  end
  spectrumRight = smooth(spectrumRight, smtValues(k));
end

dv = diff(spectrumRight);
if all(dv>0) || all(dv<0)
  % map the block
  vBlock(vBlock>max(spectrumRight)) = max(spectrumRight);
  vBlock(vBlock<min(spectrumRight)) = min(spectrumRight);
  lambdaMapped(blocksIndices(1, i):blocksIndices(2, i)) = ...
  interp1(spectrumRight, lambdaRight, vBlock);
else
  warndlg('Mapping non-monotonic vector');
end
else
  % look on the left edge
  k = 1;
  while nonmonotonic
    spectrumLeft = smooth(spectrumLeft, smtValues(k));
    nonmonotonic = ~ all(diff(spectrumLeft)<0);
    k = k + 1;
    spectrumLeft = smooth(spectrumLeft, smtValues(k));
  end
end

dv = diff(spectrumLeft); smt = 0;
spectrumLeftNew = spectrumLeft;
while ~ (all(dv>0) || all(dv<0))
  % find out first if the non-monotonicity is only in the first
  % third of the edge, in that case crop from the left
  dvFirstThird = diff(floor(length(spectrumLeftNew)/3));
  isMonotonicFirstThird = all(dvFirstThird > 0) || all(dvFirstThird < 0);
  isMonotonicRest = all(dvRest > 0) || all(dvRest < 0);
  if ~isMonotonicFirstThird && isMonotonicRest
    spectrumLeftNew = spectrumLeftNew(2:end);
  else
    smt = smt + 1;
    spectrumLeftNew = smooth(spectrumLeftNew, smt);
  end
  dv = diff(spectrumLeftNew);
end

vBlock(vBlock>max(spectrumLeftNew)) = max(spectrumLeftNew);
vBlock(vBlock<min(spectrumLeftNew)) = min(spectrumLeftNew);
lambdaMapped(blocksIndices(1, i):blocksIndices(2, i)) = ...
interp1(spectrumLeftNew, lambdaLeft, vBlock);
waitbar(i/i0, h_w); drawnow;

% things can possibly go wrong here on the non-monotonous parts of the spectrum
end
toc
close(h_w);
endWatchCursor(oldPointer);
end

function oldPointer = startWatchCursor()
    oldPointer = get(gcf, 'Pointer');
    set(gcf, 'Pointer', 'watch');
end

function endWatchCursor(oldPointer)
    set(gcf, 'Pointer', oldPointer);
end

../+algorithms/MWToDBm.m

function PdBm = MWToDBm(PmW)
% converts values of power from mW to dBm
PdBm = 10.*log10(PmW);

../+algorithms/num2eng.m

function str=num2eng(num);

% NUM2ENG Convert numbers to engineering notation strings.
% str = num2eng(num) converts the number NUM into a engineering
% notation string using International System of Units (SI) prefixes.

suffix_str='yzafpmkMGTPEZY';
k=1;
while abs(num)>=10^(3*(k-8)) && k<=16,
k=k+1;
end
if k==9,
suff=suffix_str(k);
else
suff=''
end
str=[num2str(num/10^(3*(k-9))), suff];

../+algorithms/ReadLeCroyBinaryWaveform.m

function waveform = LeCroyBinaryWaveform(FILENAME)
% waveform = LeCroyBinaryWaveform(FILENAME) loads the waveform file into the workspace variable W. 
% FILENAME can either be a variable or a string constant enclosed by quotes.
% The return value "waveform" is a record containing four elements:
% waveform.INFO Waveform information, in readable formats. For example Oscilloscope ID, 
% sampling time and settings
% waveform.DESC Waveform information used for further calculations. For example Sampling 
% rate
% waveform.Y Values sampled by the oscilloscope
% waveform.X Array of time values corresponding to waveform.Y. Time '0' marks the 
% trigger event
The routine was tested with files generated by a WavePro7300 and WaveRunner64Xi. Template LECROY_2_3

See also LCPlot LCPlotText

Original version (c)2001 Hochschule fur Technik+Architektur Luzern
% Fachstelle Elektronik
% 6048 Horw, Switzerland
% Slightly modified by Alan Blankman, LeCroy Corporation, 2006
% July 2007; added decoding of TrigTime_array so that sequence mode waveforms decode correctly
% Slightly modified by Jean-Daniel Deschnes, Universite Laval, 2010
% Added support for sequence mode, even if RECORD_TYPE says "single_sweep"
% (seen on a WavePro 7 Zi). Output will be a MxN matrix, where M is the number of points in each segment, and N is the number of segments. X vector also update to include offset of each trigger.

function wave=ReadLeCroyBinaryWaveform ( fn )

% Open File
% fid=fopen(fn,'r');
if fid==−1
    disp ( sprintf('ERROR: file not found: %s', fn) );
    return
end;

% Seek offset in the header block
% data=fread(fid,50);
WAVEDESC=findstr( 'WAVEDESC', char(data(1:50)) )−1;
% subtract 1 because:
% −1st byte in the File = Index [0]
% −1st byte in the Matlab−Array = Index[1]

% Define the addresses of the various informations in the file
% These addresses are valid for the template LECROY_2_3 and are subject to change in
% future revisions of the LeCroy firmware. Testing to this template version
% has been disabled to facilitate decoding of waveforms from scopes
% utilizing version 2.2 of the template.

TESTED_TEMPLATE = 'LECROY_2_3';

% Addresses (WAVEDESC + address as stated in the LECROY template.)
aTEMPLATE_NAME = WAVEDESC+ 16;
aCOMM_TYPE = WAVEDESC+ 32;
aCOMM_ORDER = WAVEDESC+ 34;
aWAVE_DESCRIPTOR = WAVEDESC+ 36; % length of the descriptor block
aUSER_TEXT = WAVEDESC+ 40; % length of the usertext block
aTRIGTIME_ARRAY = WAVEDESC+ 48;
aWAVE_ARRAY_1 = WAVEDESC+ 60; % length (in Byte) of the sample array
aINSTRUMENT_NAME = WAVEDESC+ 76;
aINSTRUMENT_NUMBER = WAVEDESC+ 92;
aTRACE_LABEL = WAVEDESC+ 96;
aWAVE_ARRAY_COUNT = WAVEDESC+ 116;
aSUBARRAY_COUNT = WAVEDESC+ 144;
aVERTICAL_GAIN = WAVEDESC+ 156;
aVERTICAL_OFFSET = WAVEDESC+ 160;
aNOMINAL_BITS = WAVEDESC+ 172;
aHORIZ_INTERVAL = WAVEDESC+ 176;
aHORIZ_OFFSET = WAVEDESC+ 180;
aVERTUNIT = WAVEDESC+ 196;
aHORUNIT = WAVEDESC+ 244;
aTRIGGER_TIME = WAVEDESC+ 296;
aRECORD_TYPE = WAVEDESC+ 316;
aPROCESSING_DONE = WAVEDESC+ 318;
aTIMEBASE = WAVEDESC+ 324;
aVERT_COUPLING = WAVEDESC+ 326;
aPROBE_ATT = WAVEDESC+ 328;
aFIXED_VERT_GAIN = WAVEDESC+ 332;
aBANDWIDTH_LIMIT = WAVEDESC+ 334;
aVERTICAL_VERNIER = WAVEDESC+ 336;
aACQ_VERT_OFFSET = WAVEDESC+ 340;
aWAVE_SOURCE = WAVEDESC+ 344;

% determine the number storage format HIFIRST / LOFIRST (big endian / little endian)

fseek (fid, aCOMM_ORDER, 'bof');
COMM_ORDER = fread (fid, 1, 'int16');
fclose (fid);

% reopen the data file using the correct HIFIRST/LOFIRST format
if COMM_ORDER==0
    fid = fopen(fn, 'r', 'iieee-be'); % HIFIRST
else
    fid = fopen(fn, 'r', 'iieee-le'); % LOFIRST
end;

% Get the waveform information (Commented out to facilitate decoding of 2_2 files)

% Check the template revision (Commented out to facilitate decoding of 2,2 files)
TEMPLATE_NAME = ReadString (fid, aTEMPLATE_NAME);
if strcmp( deblank (TEMPLATE_NAME), TESTED-template)
disp (sprintf ('WARNING!
%s %s %s
%s %s %s', ....
'This function has been written for the LeCroy-Template' ....
TESTED_TEMPLATE, '. ...'.
The current file contains information created with the template' ....
TEMPLATE_NAME, '. ...'));
end

% Instrument
wave.info.INSTRUMENT_NAME = ReadString (fid, aINSTRUMENT_NAME);
wave.info.INSTRUMENT_NUMBER = ReadLong (fid, aINSTRUMENT_NUMBER);
wave.info_FILENAME = fn;

% Channel
wave.info.TRIGGER_TIME = ReadTimestamp (fid, aTRIGGER_TIME);
tmp=['channel 1';'channel 2';'channel 3';'channel 4';'unknown '];
wave.info.WAVE_SOURCE = tmp (1+ ReadWord (fid, aWAVE_SOURCE)) ;
tmp=['DC 50 Ohms'; 'ground '; 'DC 1Mohm '; 'ground '; 'AC 1Mohm '];
wave.info.VERT_COUPLING = deblank (tmp (1+ ReadWord (fid, aVERT_COUPLING)));
tmp=['off'; 'on '];
wave.info.BANDWIDTH_LIMIT = deblank (tmp (1+ ReadWord (fid, aBANDWIDTH_LIMIT)));
tmp=['single_sweep '; 'interleaved '; 'histogram ';
'graph '; 'filter_coefficient '; 'complex ';
'extrema '; 'sequence_obsolete '; 'centered_RIS ';
'peak_detect '];
wave.info.RECORD_TYPE = deblank (tmp (1+ ReadWord (fid, aRECORD_TYPE)));
tmp=['no_processing '; 'fir_filter '; 'interpolated '; 'spared '];
'autoscaled '; 'no_result '; 'rolling '; 'cumulative '];

wave.info.PROCESSING_DONE = deblock (tmp (1+ ReadWord(fid, aPROCESSING_DONE).));

% Vertical settings
FIXED_VERT_GAIN = ReadFixed_vert_gain(fid, aFIXED_VERT_GAIN);
PROBE_ATT = ReadFloat (fid, aPROBE_ATT);
VERTICAL_GAIN = ReadFloat (fid, aVERTICAL_GAIN);
VERTICAL_OFFSET = ReadFloat (fid, aVERTICAL_OFFSET);
wave.info.NOMINAL_BITS = ReadWord (fid, aNOMINAL_BITS);
wave.info.Gain_with_Probe = strcat (Float_to_Eng (FIXED_VERT_GAIN*PROBE_ATT), 'V/div');

% Horizontal settings
HORIZ_INTERVAL = ReadFloat (fid, aHORIZ_INTERVAL);
HORIZ_OFFSET = ReadDouble (fid, aHORIZ_OFFSET);
wave.info.TIMEBASE = strcat (Float_to_Eng (ReadTimebase (fid, aTIMEBASE)), 's/div');
wave.info.SampleRate = strcat (Float_to_Eng (1/HORIZ_INTERVAL), 'S/sec');
wave.desc Ts = HORIZ_INTERVAL;
wave.desc.fs = 1/HORIZ_INTERVAL;

% Read samples array (Plain binary ADC values)
% COMM_TYPE = ReadWord ( fid , aCOMM_TYPE ) ;
WAVE_DESCRIPTOR = ReadLong ( fid , aWAVE_DESCRIPTOR ) ;
USER_TEXT = ReadLong ( fid , aUSER_TEXT ) ;
WAVEARRAY_1 = ReadLong ( fid , aWAVEARRAY_1 ) ;
WAVEARRAY_COUNT = ReadLong ( fid , aWAVEARRAY_COUNT ) ;
TRIGTIME_ARRAY = ReadLong ( fid , aTRIGTIME_ARRAY ) ;

% returns number of segments in acquisition
wave.info.nbSegments = ReadLong ( fid , aSUBARRAY_COUNT ) ;

if wave.info.nbSegments > 1
    % for sequence mode only
    % Take from X-Stream oscilloscopes remote control manual, appendix II:
    % < 0> TRIGGER_TIME: double ; for sequence acquisitions ,
    % < 0> TRIGGER_OFFSET: double ; time in seconds from first
    % < 8> TRIGGER_OFFSET: double ; the trigger offset is in seconds
    % fseek ( fid , WAVEDESC + WAVE_DESCRIPTOR + USER_TEXT , 'bof' ) ;
    % trigtime_array_temp = fread ( fid , 2*wave.info.nbSegments , 'double' ) ;
    % wave.trigger_time = trigtime_array_temp ( 1 : 2 : end ) ;
    % wave.trigger_offset = trigtime_array_temp ( 2 : 2 : end ) ;
    if WAVEARRAY_1/wave.info.nbSegments ~= WAVEARRAY_1/wave.info.nbSegments
        % if this condition happens, MATLAB will pad the rest of the output
        % matrix with zeros, so the file is still readable, but this could
        % signify another error (in this m-file?)
        warning ('While reading segment file: Total number of points is not a multiple of the number of segments');
    end

% Read the ADC values into a matrix
% since this is a sequence acquisition, we will return a
% (WAVEARRAY_COUNT/ nbSegments ) x ( nbSegments ) matrix, where each column is one segment.
fseek ( fid , WAVEDESC + WAVE_DESCRIPTOR + USER_TEXT + TRIGTIME_ARRAY , 'bof' ) ;
if COMM_TYPE == 0 % byte
wave.y = fread(fid, [WAVE ARRAY COUNT/wave.info.nbSegments wave.info.nbSegments], 'int8');
else
    wave.y = fread(fid, [WAVE ARRAY COUNT/wave.info.nbSegments wave.info.nbSegments], 'int16');
end;

% Create corresponding matrix of time, with correction for each trigger time
wave.x = repmat(wave.trigger.time, WAVE ARRAY COUNT/wave.info.nbSegments, 1) + repmat((0:WAVE ARRAY COUNT/wave.info.nbSegments - 1)*HORIZ INTERVAL + HORIZ OFFSET, 1, wave.info.nbSegments);
else
    fseek(fid, WAVE DESC + WAVE DESCRIPTOR + USER TEXT + TRIG TIME ARRAY, 'bof');
    if COMM TYPE == 0
        wave.y = fread(fid, WAVE ARRAY COUNT, 'int8');
    else
        wave.y = fread(fid, WAVE ARRAY COUNT, 'int16');
    end;

% Create corresponding array of time
wave.x = (0:WAVE ARRAY COUNT - 1)*HORIZ INTERVAL + HORIZ OFFSET;
end

% Transform the ADC values to voltages
wave.y = VERTICAL_GAIN * wave.y - VERTICAL_OFFSET;

% close the waveform file
fclose(fid);

%============================================================================
% Support functions
%============================================================================

% Read 8 Bit signed Byte (not used)
function b = ReadByte(fid, Addr)
    fseek(fid, Addr, 'bof');
    b = fread(fid, 1, 'int8');
end

% Read 16 Bit signed Word
function w = ReadWord(fid, Addr)
    fseek(fid, Addr, 'bof');
    w = fread(fid, 1, 'int16');
end

% Read 32 Bit signed Long
function l = ReadLong(fid, Addr)
    fseek(fid, Addr, 'bof');
    l = fread(fid, 1, 'int32');
end

% Read 32 Bit IEEE Float
function f = ReadFloat(fid, Addr)
fseek(fd, Addr, 'bof');

f= fread(fd, 1, 'float32');

% Read 64Bit IEEE Double
function d=ReadDouble(fd, Addr)
    fseek(fd, Addr, 'bof');
d= fread(fd, 1, 'float64');

% Read string (up to 16 characters)
function s=ReadString(fd, Addr)
    fseek(fd, Addr, 'bof');
s=fgets(fd, 16);

% Read timestamp
function t=ReadTimestamp(fd, Addr)
    fseek(fd, Addr, 'bof');
    seconds = fread(fd, 1, 'float64');
    minutes = fread(fd, 1, 'int8');
    hours = fread(fd, 1, 'int8');
    days = fread(fd, 1, 'int8');
    months = fread(fd, 1, 'int8');
    year = fread(fd, 1, 'int16');
    t=sprintf('%i.%%i:%i:%2.0f', days, months, year, hours, minutes, seconds);

% Timebase aus dem File lesen
function t=ReadTimebase(fd, Addr)
    fseek(fd, Addr, 'bof');
e=fread(fd, 1, 'int16');
tmp=[1 2 5];
mant = tmp(1+mod(e, 3));
ex = floor(e / 3) - 12;
t = mant*10^ex;

% fixed Vertical Gain aus dem File lesen
function t=ReadFixed_vert_gain(fd, Addr)
    fseek(fd, Addr, 'bof');
e=fread(fd, 1, 'int16');
tmp=[1 2 5];
mant = tmp(1+mod(e, 3));
ex = floor(e / 3) - 6;
t= mant*10^ex;

% Transform a Float to the Engineering–Format (returns a string)
function s=Float_to_Eng(f)
    ex = floor(log10(f));
exeng = ex - mod(ex, 3);
if exeng<-18; exeng=-18; end
if exeng>18; exeng=18; end;
mant=f/10^exeng;

    prefix=('afpnum kMGPE');
    prefixs (u=micro, m=milli, k=kilo, ...)
s=sprintf('%g%s', mant, prefix((exeng+18)/3+1));
function [fGridSlotted, fGridSlottedFineTuning, fFineTuningSlotsGridCenters] = SlotTheGridForFineTuning (fGrid, laserSettings)

%FTF stands for fine tune filter
fFineLim = 1e9 .* laserSettings.Properties.FTF.lim;
fFineSpan = fFineLim(2) - fFineLim(1);

NumberOfFineSlots = ceil(fSpan/fFineSpan);

%f create a subgrid of fine tuning slot borders
fFineTuningSlotsGridBorders = fliplr((fGrid(1):fFineSpan:end)
(fGrid(1)) + fFineLim(2);
fFineTuningSlotsGridBorders = [fFineTuningSlotsGridBorders ...
fFineTuningSlotsGridBorders(end) - fFineSpan];
fFineTuningSlotsGridCenters = ...
fFineTuningSlotsGridBorders(1:end-1) - fFineLim(2);
tuningAllRough = 0;

%f creating rough and fine tuning grid slots
fGridDifference = diff(fGrid);
if min(abs(fGridDifference))<fFineSpan; tuningAllRough = 1;

%f tuning all frequencies with rough tuning method
fGridSlotted = fGrid';
fGridSlottedFineTuning = zeros(size(fGrid'));
disp('Tuning all frequencies with rough tuning method');
else

nn=1;

%f assigning frequencies into fine tuning slots
disp('Assigning frequencies into fine tuning slots');
for kk=1:length(fGrid)
    slotNo(kk) = min(find(fFineTuningSlotsGridBorders < fGrid(kk)))-1;
end
fIndices = NaN(NumberOfFineSlots, NumberOfFineSlots);
fGridSlottedFineTuning = NaN(NumberOfFineSlots, NumberOfFineSlots);
fGridSlotted = NaN(1, NumberOfFineSlots);
for kk=1:NumberOfFineSlots
    temp=find(slotNo==kk);
    if length(temp)>0:
        fIndices(kk,1:length(temp))=temp;
        fGridSlotted(kk)=fFineTuningSlotsGridCenters(kk);
        for jj=1:length(temp)
            fGridSlottedFineTuning(nn, jj) = fGrid(fIndices(kk, jj))-
            fFineTuningSlotsGridCenters(kk);
        end
        nn=nn+1;
    end
end
end

function varargout = timeoutDlg(dlg, delay, varargin)

% Dialog function with timeout property
dlg is a handle to the dialog function to be called
delay is the length of the delay in seconds
% other input arguments as required by the dialog
% EXAMPLE FUNCTION--CALL
% To display an input dialog box (REFER MATLAB HELP DOC) with a
% timeout = 6 second say, the function call would be:
% 
% [matrix, size, value, colormap_string] = timeoutDlg(@inputdlg, 6, ...
% {'Enter matrix size:' , 'Enter colormap name:' }, ...
% 'Input for peaks function', 1, {'20', 'hsv'})
% Setup a timer to close the dialog in a moment
f1 = findall(0 , 'Type' , 'figures');
t = timer('TimerFcn' , { @closeit f1 }, 'StartDelay' , delay);
start(t);

% Call the dialog
retvals = dlg(varargin{:});
if numel(retvals) == nargout
    varargout = retvals(1);
else
    varargout = cell(1, nargout);
end

% Delete the timer
if strcmp(t.Running, 'on')
    stop(t);
end
delete(t);

function closeit(src, event, f1)
disp('Time out!');

function WaitDialog(message, timeout)
% Opens a dialog which will display a countdown. This function returns
% when the dialog is closed or when the timeout in seconds expires.
% The first argument is the message to display in the dialog.
% The second argument is the time to wait before closing the dialog
% in seconds.
sz = [200 50];
screensz = get(0, 'ScreenSize');
xpos = ceil((screensz(3) - sz(1))/2);
ypos = ceil((screensz(4) - sz(2))/2);
f = figure('units', 'pixels', ...
    'position', [xpos, ypos, sz(1), sz(2)] , ...
    'toolBar', 'none');

../+algorithms/wait_tuning.m

function wait_tuning(ser_obj, Port, max_time)
% waits until laser is busy or max_time [s] is elapsed

timer_start = now;
fprintf('laser busy ...');
while ((now - timer_start)*24*3600)<=max_time
    if all('CBMx_query_tuning_state(ser_obj, Port)
        break;
    end
    pause(1);
    fprintf(' . ');
end

function WaitDialog(message, timeout)
function onTick(obj, ev)
    timeout = timeout - 1;
    set(timeText, 'String', num2str(timeout));
    if (timeout < 1)
        uiresume(f);
    end
end

function timer('Period', 1,...
    'ExecutionMode', 'fixedDelay'...,
    'StartDelay', 1,...
    'TimerFcn', @onTick);
start(t);
uiwait(f);
stop(t);
if ishandle(f)
    close(f);
end
end

../+algorithms/findPeakToPeak.m

function [peak_to_peak, offset, ACDC] = findPeakToPeak(x0, y0, units)
% findPeakToPeak -- find peak to peak value of the y0(x0) data set at the dominant
% frequency in terms of x0
% units -- is a designator of units of y0, 1 for Volts (voltage) and 2 for m (wavelength)

% CONSTANTS
noiseTolerance = 100e-3;

switch margin
    case 2
        units = 1;
        % 1 for Volts (voltage), and
        % 2 for m (wavelength)
end

oldPointer = startWatchCursor();

% 1. Determining peak-to-peak in frequency domain (best for harmonic signals)
L=length(x0);  % Number of samples
fs=L/(max(x0)-min(x0));  % Sample rate
fn=fs/2;  % Nyquist frequency
Y_f=fft(y0)/L;
frequency = (0:floor(L/2))./(L/2).*fn;
Y_fourier = 2*abs(Y_f(1:size(frequency,2)));
% find most dominant peaks
nPeaks = 50;
[ps1, loc1, wl, pl]=findpeaks(Y_fourier, 'NPeaks', nPeaks, 'SORTSTR', 'descend');
% keep only tall peaks: peaks taller than 1% of the tallest peak
nPeaks = sum(ps1>0.01*ps1(1));
[ps1, loc1, wl, pl]=findpeaks(Y_fourier, 'NPeaks', nPeaks, 'SORTSTR', 'descend');
freq=frequency(loc1(1)); ampl=pks1(1);
peak_to_peak1 = 2*pks1(1);
offset1 = Y_fourier(1)/2;

% 2. Determining peak-to-peak in time domain (best for transient signals)
nPeaks = 50;
[pks2, loc2, w2, p2]=findpeaks(y0, 'NPeaks', nPeaks, 'SORTSTR', 'descend');
%keep only wide peaks: with widths greater than 1% of the widest peak
%reason for this is to discard the spikes in signal due to RF noise
%and arcing
pks2 = pks2(w2>0.01*max(w2)); loc2 = loc2(w2>0.01*max(w2));
p2 = p2(w2>0.01*max(w2)); w2 = w2(w2>0.01*max(w2));

[pks2minus, loc2minus, w2minus, p2minus]=findpeaks(-y0, 'NPeaks', nPeaks, 'SORTSTR', 'descend');
%keep only wide peaks: with widths greater than 1% of the widest peak
pks2minus = pks2minus(w2minus>0.01*max(w2minus));
lloc2minus = loc2minus(w2minus>0.01*max(w2minus));
p2minus = p2minus(w2minus>0.01*max(w2minus));
w2minus = w2minus(w2minus>0.01*max(w2minus));

peak_to_peak2 = max(pks2) + max(pks2minus);
offset2 = mean(y0);

% 3. Determining DC offset in time domain by averaging
y0Prime = diff(y0); y0Second = diff(y0Prime);
[~, loc3] = findpeaks(y0Second, 'NPeaks', 1, 'SORTSTR', 'descend');
offset3 = mean(y0(1:loc3));

% conclusion
isHarm = isHarmonic(pks1, pks2, offset1);
if isHarm
disp('Calibration signal is harmonic');
peak_to_peak = peak_to_peak1;
offset = offset1;
ACDC = isACCoupled (offset, noiseTolerance);
else
disp('Calibration signal is transient');
peak_to_peak = peak_to_peak2;
offsetVector = [offset1 offset2 offset3];
offsetAvg = mean(offsetVector); offsetStd = std(offsetVector);
in = find((offsetVector-repmat(offsetAvg, size(offsetVector)))<=...
repmat(2*offsetStd, size(offsetVector)));
if length(in)==3
offset = offset1;
else
offset = mean(offsetVector(in));
end
ACDC = isACCoupled (offset, noiseTolerance);
end

% summary
switch units
    case 1
        if isHarm
disp(sprintf('Peak to peak = %3.2f mV, frequency %3.2f Hz, Offset = %3.2f mV',...
1e3*peak_to_peak, freq, 1e3+offset));
        else
disp(sprintf('Peak to peak = %3.2f mV, Offset = %3.2f mV',...
1e3*peak_to_peak, 1e3+offset));
        end
    case 2
        if isHarm
disp(sprintf('Peak to peak = %3.2f pm, frequency %3.2f Hz, Offset = %3.2f nm',...
1e12*peak_to_peak, freq, 1e9+offset));
        else
disp(sprintf('Peak to peak = %3.2f pm, Offset = %3.2f nm',...
1e12*peak_to_peak, 1e9+offset));
        end
end

167
function harmonic = isHarmonic ( pks1, pks2, offset )
    peakTime = pks2(1) - offset;
    peakFreq = pks1(1);
    peakDiffRel = (peakTime - peakFreq)/peakFreq;
    if peakDiffRel < 1
        % dominant peak calculated using FFT should be similar to peak calculated
        % in time domain for periodic signals,
        % the relative difference between the two is required to be less than 100%
        harmonic = true;
    else
        harmonic = false;
    end
end

function ACCoupled = isACCoupled ( offset, noiseTolerance )
    ACCoupled = abs(offset) < noiseTolerance;
end

function oldPointer = startWatchCursor ()
    oldPointer = get(gcf, 'Pointer');
    set(gcf, 'Pointer', 'watch');
end

function endWatchCursor ( oldPointer )
    set(gcf, 'Pointer', oldPointer);
end

../+algorithms/AnalyzeSpectrum.m

    lambda = spectrum(1,:);
    Vs = spectrum(2,:);
    smt = 1;
    % slope
    VsPrime = diff(Vs)./diff(lambda);
    VsPrimes = smooth(VsPrime, smt); % smoothing
    lambdaPrime = (lambda(2:end)+lambda(1:end-1))/2;
    % cropping the edges off because of discontinuities
    cropPercentage = 0.05;
    ind1 = ceil(length(lambdaPrime)*cropPercentage); lambda1 = lambdaPrime(ind1);
    ind2 = floor(length(lambdaPrime)*(1-cropPercentage)); lambda2 = lambdaPrime(ind2);
    VPrimeCropped = [zeros(ind1-1,1); VsPrimes(ind1:ind2); zeros((length(VsPrimes)-ind2)+1,1)];
    indMax = find(VPrimeCropped == max(VPrimeCropped)); lambdaMax = lambdaPrime(indMax);
    % line ([lambdaMax lambdaMax], [min(VsPrimes) max(VsPrimes)], 'Color', 'red');
    indPlus10Perc = max(find(VPrimeCropped>0.9*max(VPrimeCropped))); lambdaMinus10PercPlus = lambdaPrime(indPlus10Perc);
    % line ([lambdaMinus10PercPlus lambdaMinus10PercPlus lambdaMinus10PercPlus], [min(VsPrimes) max(VsPrimes)], 'Color', 'yellow');
```matlab
indMinus10Perc = min(find(VPrimeCropped > 0.9 * max(VPrimeCropped))); lambdaMinus10PercMinus = lambdaPrime(indMinus10Perc);

% line ([lambdaMinus10PercMinus lambdaMinus10PercMinus], [min(VsPrimes) max(VsPrimes)], 'Color ', ' yellow ');

indMin = find(VPrimeCropped == min(VPrimeCropped(VPrimeCropped~0))); lambdaMin = lambdaPrime(indMin);

MaxSensR = [lambdaMax VsPrimes(indMax)];
widthR = lambdaMinus10PercPlus - lambdaMinus10PercMinus;
MaxSensL = [lambdaMin, VsPrimes(indMin)];

fprintf('Maximum sensitivity at lambda = %4.3f nm\n', 1e9*lambdaMax);
fprintf('90\\% sensitivity width = %4.3f pm\n', 1e12*widthR);
fprintf('Minimum sensitivity at lambda = %4.3f nm\n', 1e9*lambdaMin);
```

A.2 GUI

Folder /+GUI needs to contain the following image files for tool icons:

1. ’brush_icon.png’
2. ’dataTip_icon.png’
3. ’edge_lr_icon.png’
4. ’revert_icon.png’
5. ’zoom_icon.png’
6. ’zoomout_icon.png’

A.2.1 Classes

```matlab
../+GUI/BusyBar.m

classdef BusyBar < handle
    % BusyBar is a text bar at the bottom that displays when the app is busy
    % and user shouldn’t touch anything

    properties (Constant)
        busyTextStringBusy = 'Busy'
        busyTextStringReady = 'Ready'
    end

    properties (Transient, AbortSet)
        Units % Units for this panel
        Position % On parent in Units
        Parent % Figure, uitab, or uipanel
    end
```

169
properties (Transient, SetObservable)
  Busy % Boolean value that indicates when to display Busy text
end

properties (Access = private)
busyText
end

methods
  function obj = BusyBar(varargin)
    parseInputs(obj, varargin{:});
    % Build the resonance panel
    obj.buildPanel();
    addlistener(obj.busyText, 'ObjectBeingDestroyed', @('',) obj.delete);
    addlistener(obj, 'Busy', 'PostSet', @obj.updatePanel);
  end
end

methods (Access = private)
  function buildPanel(obj)
    obj.busyText = uicontrol('Parent', obj.Parent, ... 'Style', 'text', ... 'String', obj.busyTextStringReady, ... 'HorizontalAlignment', 'left', ... 'Units', 'pixels', ... 'Position', [25, 3, 100, 15]);
  end

  function updatePanel(obj, '', '')
    % Called up update the panel
    if obj.Busy
      obj.busyText.String = obj.busyTextStringBusy;
    else
      obj.busyText.String = obj.busyTextStringReady;
    end
  end

  function parseInputs(obj, varargin)
    % Parses inputs
    parser = inputParser;
    parser.FunctionName = 'ColorPanel';
    addParameter(parser, 'Parent', get(groot, 'CurrentFigure'))
    addParameter(parser, 'Position', get(groot, 'DefaultUicontrolPosition'))
    addParameter(parser, 'Units', get(groot, 'DefaultUicontrolUnits'))
    parse(parser, varargin{:});
    parent = gcf;
  end
end
classdef CalibrationPanel < handle
% CalibrationPanel Panel for handling calibration from a spectrum
% Detailed explanation goes here

properties (Transient, AbortSet)
    Units % Units for this panel
    Position % On parent in Units
    Parent % Figure, uitab, or uipanel
end

properties (Transient, SetObservable)
    Spectrum % ISpectrum object used for calibration
    Sensor % Sensor object to use
    Calibration % Calibration data object
    Results % Results data object
    plotResultsWindowFlag = 0; % Flag that signals the time to plot results
    Busy = false;
end

properties (Access = private)
    panel
    smallGainButton1
    smallGainButton2
    loadCalibrationButton
    calibrateTraceButton
    calibrationVoltageEdit
    sGain
end

events
    PanelChanged
end

methods
    function obj = CalibrationPanel(varargin)
        parseInputs(obj, varargin{:});
        obj.sGain = 1;
        obj.buildPanel();
        addlistener(obj.panel, 'ObjectBeingDestroyed', @('', '') obj.delete);
        addlistener(obj, 'PanelChanged', @obj.updatePanel);
        addlistener(obj, 'Spectrum', 'PostSet', @obj.updatePanel);
        addlistener(obj, 'Sensor', 'PostSet', @obj.updatePanel);
        addlistener(obj, 'Calibration', 'PostSet', @obj.updatePanel);
        notify(obj, 'PanelChanged'); % make sure everything is up to date
    end

methods (Access = private)
    function buildPanel(obj)
        % Builds our main panel
        obj.panel = uipanel('Parent', obj.Parent, ...
            'Title', 'Calibration', ...
            'Units', obj.Units, ...
            'Position', obj.Position);
        obj.smallGainButton1 = uicontrol('Parent', obj.panel, ...
            'Style', 'radiobutton', ...
            'String', 'xl' ...
        end

171
'Enable', 'off', ...
'Value', 1, ...
'Units', 'pixels' ...
'HorizontalAlignment', 'right' ...
'TooltipString', 'Small signal gain' ...
'Position', [20, 35, 35, 15] ...
'Callback', @obj.onSmallGain1Clicked);

obj.smallGainButton2 = uicontrol('Parent', obj.panel, ....
'Style', 'radiobutton' ....
'String', 'x10' ....
'Enable', 'off' ....
'Units', 'pixels' ....
'HorizontalAlignment', 'right' ....
'TooltipString', 'Small signal gain' ....
'Position', [55, 35, 40, 15] ....
'Callback', @obj.onSmallGain2Clicked);

uicontrol('Parent', obj.panel ....
'Style', 'text' ....
'String', 'Peak-to-peak voltage [V]:' ....
'Units', 'pixels' ....
'TooltipString', ['Enter the peak-to-peak value of the calibration voltage in V. ....
'Sensor needs to be selected from the Sensor Panel to activate this edit field'] ....
'Position', [110, 35, 120, 15]);

obj.calibrationVoltageEdit = uicontrol('Parent', obj.panel, ....
'Style', 'edit' ....
'String', '6760' ....
'Units', 'pixels' ....
'HorizontalAlignment', 'left' ....
'Position', [235, 35, 45, 20] ....
'Callback', @obj.onCalibrationVoltageChange);

obj.loadCalibrationButton = uicontrol('Parent', obj.panel, ....
'Style', 'pushbutton' ....
'String', 'Load Calibration File' ....
'Units', 'pixels' ....
'Position', [20, 5, 130, 30] ....
'Callback', @obj.onLoadCalibrationClicked);

obj.calibrateTraceButton = uicontrol('Parent', obj.panel, ....
'Style', 'pushbutton' ....
'String', 'Calibrate Trace File' ....
'Units', 'pixels' ....
'Position', [150, 5, 130, 30] ....
'Callback', @obj.onCalibrateTraceClicked);

end

function updatePanel(obj, " ")
% Called up update the panel
obj.panel.Position = obj.Position;
obj.panel.Units = obj.Units;
obj.panel.Parent = obj.Parent;

if ~isempty(obj.Spectrum) && ~isempty(obj.Sensor) ...
 & obj.Sensor.EShift
  obj.calibrationVoltageEdit.Enable = 'on';
  obj.loadCalibrationButton.Enable = 'on';
  obj.smallGainButton1.Enable = 'on';
  obj.smallGainButton2.Enable = 'on';
  if isobject(obj.Calibration)
    obj.calibrateTraceButton.Enable = 'on';
  end
else
  obj.calibrationVoltageEdit.Enable = 'off';
  obj.loadCalibrationButton.Enable = 'off';
  obj.smallGainButton1.Enable = 'off';
  obj.smallGainButton2.Enable = 'off';
obj.calibrateTraceButton.Enable = 'off';
obj.smallGainButton1.Enable = 'off';
obj.smallGainButton2.Enable = 'off';
end
obj.Busy = false;
end

function onCalibrationVoltageChange(obj, "", "")
enteredValue = str2double(obj.calibrationVoltageEdit.String);
if isnan(enteredValue)
    warndlg('Input must be a number');
    obj.calibrationVoltageEdit.String = '6760';
end
end

function onLoadCalibrationClicked(obj, "", "")
oldPointer = obj.startWatchCursor();
obj.Busy = true;
set(obj.Parent, 'Pointer', 'watch');
if isempty(obj.Spectrum)
    file = GUI.findfile('*.trc;*.dat;*.txt');
vPPCalibration = str2double(obj.calibrationVoltageEdit.String);
    if all(file)
        %Load the calibration data from the specified file
        [~, ~, fileExt] = fileparts(file);
        switch fileExt
        case '.isf'
            data = isfread(file);
tCal=data.x; vCal=data.y; clear data;
case '.csv'
            M = csvread(file);
tCal = M(:,1); vCal=M(:,2); clear M;
case '.trc'
            M=algorithm.ReadLeCroyBinaryWaveform(file);
tCal = M.x; vCal=M.y; clear M;
case '.txt'
            M=dlmread(file);
tCal = M(:,1); vCal=M(:,2); clear M;
end
    obj.Calibration = ...
        obj.Spectrum.Calibrate(tCal, vCal./obj.sGain, obj.sGain, ...
        vPPCalibration, obj.Sensor);
end
    obj.endWatchCursor(oldPointer);
    obj.Busy = false;
end

function onCalibrateTraceClicked(obj, "", "")
oldPointer = obj.startWatchCursor();
obj.Busy = true;
if isempty(obj.Spectrum)
    file = GUI.findfile('*.trc;*.dat;*.txt');
EShift = obj.Sensor.EShift;
    if all(file)
        %Load Measurement file
        switch file(end-3:end)
        case '.isf'
            data = isfread(file);
t=data.x; v=data.y;
case '.csv'
            M = csvread(file);
t = M(:,1); v=M(:,2);
case '.trc'
            M=algorithm.ReadLeCroyBinaryWaveform(file);
t = M.x; v=M.y; clear M;
case '.txt'
            M=dlmread(file);
end
end
end
t = M(:,1); v=M(:,2); clear M;

end

obj.Results = obj.Spectrum.ProcessData(t, v./obj.sGain ... 
obj.sGain, file, obj.Sensor);
obj.plotResultsWindowFlag = 1;
end

end

obj.endWatchCursor(oldPointer);
obj.Busy = false;
end

function onSmallGain1Clicked(obj, ", ")
if obj.smallGainButton1.Value
  if obj.smallGainButton2.Value
    obj.smallGainButton2.Value = 0;
  end
  obj.sGain = 1;
else
  obj.smallGainButton1.Value = 1;
end
end

function onSmallGain2Clicked(obj, ", ")
if obj.smallGainButton2.Value
  if obj.smallGainButton1.Value
    obj.smallGainButton1.Value = 0;
  end
  obj.sGain = 10;
else
  obj.smallGainButton2.Value = 1;
end
end

function oldPointer = startWatchCursor(obj)
  fHandle = obj.Parent;
  oldPointer = get(fHandle, 'Pointer');
  set(fHandle, 'Pointer', 'watch');
end

function endWatchCursor(obj, oldPointer)
  fHandle = obj.Parent;
  set(fHandle, 'Pointer', oldPointer);
end

function parseInputs(obj, varargin)
  % Parses inputs
  
  parser = inputParser;
  parserFcnName = 'ColorPanel';

  addParameter(parser, 'Parent', get(groot, 'CurrentFigure'))
  addParameter(parser, 'Position', get(groot, 'DefaultUicontrolPosition'))
  addParameter(parser, 'Units', get(groot, 'DefaultUicontrolUnits'))
  addParameter(parser, 'Spectrum', [])

  parse(parser, varargin{:});

  if isempty(parser.Results.Parent)
    parent = gcf; % Use current figure for parent if none specified
  else
    parent = parser.Results.Parent;
  end

  obj.Parent = parent;
  obj.Units = parser.Results.Units;
  obj.Position = parser.Results.Position;
  obj.Spectrum = parser.Results.Spectrum;
classdef DAQPanel < handle
  %DAQPANEL Summary of this class goes here
  % Detailed explanation goes here

  properties (Transient, AbortSet)
    Units % Units for this panel
    Position % On parent in Units
    Parent % Figure, uitab, or uipanel
  end

  properties (Transient, SetAccess = private, SetObservable)
    DAQ % DAQ object selected
    Busy = false;
  end

  properties (Transient, Access = private)
    panel
    factories
    namePopup
    daqStatusText
    initButton
    shutdownButton
  end

  properties (Transient, Access = private)
    hVoltage % Current voltage handle
    hState % Current state handle
  end

  properties (Transient, Dependent)
    DAQState
  end

  events
    PanelChanged
    DAQChanged
  end

  methods
    function obj = DAQPanel(daqFactories, varargin)
      obj.DAQ = []; 
      obj.factories = daqFactories;
      parseInputs(obj, varargin{:});
      obj.buildPanel();
      addlistener(obj.panel, 'ObjectBeingDestroyed', @(~,~) obj.delete);
      addlistener(obj, 'PanelChanged', @obj.updatePanel);
      addlistener(obj, 'DAQChanged', @obj.updateDAQ);
      obj.Busy = false;
    end

    function set.Parent(obj, newParent)
      % Validate/Set/Update
      assert(ishghandle(newParent), 'New Parent is expected to be a valid handle to a figure , uitab, or uipanel')
  end
  end
end

../+GUI/DAQPanel.m
assert(ismember(newParent.Type,{'figure','uipanel','uitab'}), 'New Parent is expected to be a valid handle to a figure, uipanel, or uitab')

obj.Parent = newParent;

% Notify
notify(obj,'PanelChanged');
end

function set.DAQ(obj, daq)
    obj.DAQ = daq;
    notify(obj,'DAQChanged');
    notify(obj,'PanelChanged');
end

function s = get.DAQState(obj)
    if isempty(obj.DAQ)
        s = 'No DAQ selected';
    else
        s = sprintf('%s, Voltage: %04.1fV', char(obj.DAQ.State), GUI.num2eng(round(obj.DAQ.Voltage,4)));
    end
end

methods (Access = private)

function buildPanel(obj)
    % Builds our main panel

    obj.panel = uipanel('Parent', obj.Parent,...
    'Title', 'DAQ', ...
    'Units', obj.Units,...
    'Position', obj.Position);

    names = cellfun(@(f) f.Name, obj.factories, 'un', 0);

    obj.namePopup = uicontrol('Parent', obj.panel,...
    'Style', 'popup', ...
    'String', names, ...
    'Units', 'pixels', ...
    'Position', [20, 50, 260, 20]);

    obj.initButton = uicontrol('Parent', obj.panel,...
    'Style', 'pushbutton', ...
    'String', 'Initialize', ...
    'Units', 'pixels', ...
    'Position', [20, 15, 130, 30], ...
    'Callback', @obj.onInitClicked);

    obj.shutdownButton = uicontrol('Parent', obj.panel,...
    'Style', 'pushbutton', ...
    'String', 'Shutdown', ...
    'Units', 'pixels', ...
    'Enable', 'off', ...
    'Position', [150, 15, 130, 30], ...
    'Callback', @obj.onShutdownClicked);

    obj.daqStatusText = uicontrol('Parent', obj.panel,...
    'Style', 'text', ...
    'String', sprintf('%s', obj.DAQState), ...
    'Enable', 'inactive', ...
    'TooltipString', 'Mouse click to refresh voltage', ...
    'Units', 'pixels', ...
    'HorizontalAlignment', 'center', ...
    'Position', [20, 0, 260, 15], ...
    'ButtonDownFcn', @obj.onStatusTextClicked);
end

function updatePanel(obj, " ")
    % Called to update the panel
if isValid(obj)
    obj.panel.Position = obj.Position;
    obj.panel.Units = obj.Units;
    obj.panel.Parent = obj.Parent;
    obj.daqStatusText.String = sprintf('DAQ: %s', char(obj.DAQState));

if isObject(obj.DAQ)
    obj.namePopup.Enable = 'off';
    obj.initButton.Enable = 'off';
    obj.shutdownButton.Enable = 'on';
else
    obj.namePopup.Enable = 'on';
    obj.initButton.Enable = 'on';
    obj.shutdownButton.Enable = 'off';
end

obj.Busy = false;
end

function updateDAQ(obj, "", "")
% Performs steps for updating the DAQ
if "isempty(obj.hvoltage)
    delete(obj.hvoltage)
end
if "isempty(obj.DAQ)
    obj.hvoltage = addlistener(obj.DAQ, 'Voltage', 'PostSet', @obj.updatePanel);
    obj.hstate = addlistener(obj.DAQ, 'State', 'PostSet', @obj.updatePanel);
end

function onInitClicked(obj, "", "")
obj.Busy = true;
oldPointer = obj.startWatchCursor();
idx = obj.namePopup.Value;
factory = obj.factories{idx};
    try
daq = factory.Create();
    assert(daq.RequestVoltage == 1, 'Unable to read voltage');
    obj.DAQ = daq;
catch ME
    messagebox(ME.message, 'Error', 'modal');
    warning(getReport(ME));
end
obj.endWatchCursor(oldPointer);
obj.Busy = false;
end

function onShutdownClicked(obj, "", "")
obj.Busy = true;
oldPointer = obj.startWatchCursor();
if "isempty(obj.DAQ)
    try
    obj.DAQ.RequestShutdown;
catch ME
    messagebox(ME.message, 'Error', 'modal');
    warning(getReport(message));
end
    obj.DAQ = [];
end
obj.endWatchCursor(oldPointer);
obj.Busy = false;
end

function onStatusTextClicked(obj, "", "")
obj.Busy = true;
oldPointer = obj.startWatchCursor();

end
if isempty(obj.DAQ)
try
    assert(obj.DAQ.RequestVoltage == 1, 'Unable to read voltage');
catch ME
    msgbox(ME.message, 'Error', 'modal');
    warning(getReport(ME));
end
end
obj.endWatchCursor(oldPointer);
obj.Busy = false;
end

function oldPointer = startWatchCursor(obj)
    fHandle = obj.Parent;
    oldPointer = get(fHandle, 'Pointer');
    set(fHandle, 'Pointer', 'watch');
end

function endWatchCursor(obj, oldPointer)
    fHandle = obj.Parent;
    set(fHandle, 'Pointer', oldPointer);
end

function parseInputs(obj, varargin)
    % Parses inputs
    parser = inputParser;
    parser.FunctionName = 'ColorPanel';
    addParameter(parser, 'Parent', get(groot, 'CurrentFigure'))
    addParameter(parser, 'Position', get(groot, 'DefaultUicontrolPosition'))
    addParameter(parser, 'Units', get(groot, 'DefaultUicontrolUnits'))
    parse(parser, varargin{:});
    if isempty(parser.Results.Parent)
        parent = gcf; % Use current figure for parent if none specified
    else
        parent = parser.Results.Parent;
    end
    obj.Parent = parent;
    obj.Units = parser.Results.Units;
    obj.Position = parser.Results.Position;
end
end

..+/GUI/LaserPanel.m

classdef LaserPanel < handle
    %LASERPANEL Panel for displaying the laser
    % UI Control for controlling a laser
    % Based on http://www.mathworks.com/matlabcentral/answers/uploaded_files/22289/ColorPanel.m
properties (Transient, AbortSet)
    Units % Units for this panel
    Position % On parent in Units
    Parent % Figure, uiax, or uipanel
end

properties (Transient, SetObservable)
    Laser % Laser object
    DAQ % DAQ object
Busy = false;

end

properties (Transient, Access = private)
    panel
    factories
    namePopup
    powerEdit
    wavelengthEdit
    initButton
    shutdownButton
    autoButton
    lnButton
    hwavelength % Current wavelength handle
    hpower % Current power handle
    hstate % Current state handle
end

properties (Transient, Dependent)
    LaserState
end

events
    PanelChanged
    LaserChanged
    DaqChanged
end

methods
    function obj = LaserPanel(laserFactories, varargin)
        obj.Laser = [];
        obj.factories = laserFactories;
        parseInputs(obj, varargin{:});
        obj.buildPanel();
        addlistener(obj.panel, 'ObjectBeingDestroyed', @(), obj.delete);
        addlistener(obj, 'PanelChanged', @obj.updatePanel);
        addlistener(obj, 'LaserChanged', @obj.updateLaser);
        notify(obj, 'PanelChanged');
    end

    function setParent(obj, newParent)
        % Validate/Set/Update
        assert(ishandle(newParent), 'New Parent is expected to be a valid handle to a figure, uipanel, or uitab')
        assert(ismember(newParent.Type, {'figure', 'uipanel', 'uitab'}), 'New Parent is expected to be a valid handle to a figure, uipanel, or uitab')
        obj.Parent = newParent;
        % Notify
        notify(obj, 'PanelChanged');
    end

    function setLaser(obj, laser)
        obj.Laser = laser;
        notify(obj, 'LaserChanged');
        notify(obj, 'PanelChanged');
    end

    function setDAQ(obj, DAQ)
        obj.DAQ = DAQ;
        notify(obj, 'PanelChanged');
    end
end
methods (Access = private)

function buildPanel(obj)
% Builds our panel

    obj.panel = uipanel('Parent', obj.Parent, ...
        'Title', 'Laser', ...
        'Units', obj.Units, ...
        'Position', obj.Position);

names = cellfun(@(f) f.Name, obj.factories, 'un', 0);

    obj.namePopup = uicontrol('Parent', obj.panel, ...
        'Style', 'popup', ...
        'String', names, ...
        'Units', 'pixels', ...
        'TooltipString', 'Choose a laser', ...
        'Position', [20, 95, 260, 20]);

    obj.initButton = uicontrol('Parent', obj.panel, ...
        'Style', 'pushbutton', ...
        'String', 'Initialize', ...
        'Units', 'pixels', ...
        'TooltipString', 'Turn on the selected laser', ...
        'Position', [150, 60, 130, 30], ...
        'Enable', 'off', ...
        'Callback', @obj.onInitializeClicked);

    obj.shutdownButton = uicontrol('Parent', obj.panel, ...
        'Style', 'pushbutton', ...
        'String', 'Shutdown', ...
        'Units', 'pixels', ...
        'TooltipString', 'Shut down the selected laser', ...
        'Position', [150, 60, 130, 30], ...
        'Enable', 'off', ...
        'Callback', @obj.onShutdownClicked);

uicontrol('Parent', obj.panel, ...
    'Style', 'text', ...
    'String', 'Power:', ...
    'HorizontalAlignment', 'right' , ...
    'Units', 'pixels', ...
    'Position', [20, 30, 70, 20]);

    obj.powerEdit = uicontrol('Parent', obj.panel, ...
        'Style', 'edit', ...
        'Enable', 'off', ...
        'Units', 'pixels', ...
        'HorizontalAlignment', 'left' , ...
        'Position', [90, 30, 60, 20], ...
        'Callback', @obj.onPowerEntered);

uicontrol('Parent', obj.panel, ...
    'Style', 'text', ...
    'String', '[mW]', ...
    'HorizontalAlignment', 'left' , ...
    'Units', 'pixels', ...
    'Position', [150, 30, 30, 20]);

uicontrol('Parent', obj.panel, ...
    'Style', 'text', ...
    'String', 'Wavelength:', ...
    'HorizontalAlignment', 'right' , ...
    'Units', 'pixels', ...
    'Position', [20, 5, 70, 20]);

    obj.wavelengthEdit = uicontrol('Parent', obj.panel, ...
        'Style', 'edit', ...
        'Enable', 'off', ...
        'Units', 'pixels', ...
        'HorizontalAlignment', 'left' , ...
        'Position', [90, 5, 60, 20], ...
        'Callback', @obj.onWavelengthEntered);

uicontrol('Parent', obj.panel, ...
function onInitializeClicked (obj , "", "")
    obj.Busy = true;
    oldPointer = obj.startWatchCursor();
    idx = obj.namePopup.Value;
    factory = obj.factories {idx};
    try
        laser = factory.Create();
        if isempty(laser) ;
            return ; end
        assert(laser.UpdatePower == 1, 'Unable to set initial laser power');
        assert(laser.UpdateWavelength == 1, 'Unable to read wavelength');
        obj.Laser = laser;
        catch ME
            msgbox(ME.message, 'Error', 'modal');
            warning(getReport(ME));
        end
    obj.endWatchCursor( oldPointer );
    obj.Busy = false;
end

function onShutdownClicked (obj , "", "")
    obj.Busy = true;
    oldPointer = obj.startWatchCursor();
    if ~ isempty(obj.Laser)
        try
            obj.Laser.RequestShutdown();
            catch ME
                msgbox(ME.message, 'Error', 'modal');
                warning(getReport(ME));
            end
            obj.Laser = [];
        end
    end
    obj.endWatchCursor( oldPointer );
    obj.Busy = false;
end

function onPowerEntered (obj , "", "")
    obj.Busy = true;
    oldPointer = obj.startWatchCursor();
    if ~ isempty(obj.Laser)
        p=1e-3*str2double(obj.powerEdit.String);
        obj.Laser.SetPower(p);
assert (isnumeric(p), 'Must enter a number');
if ~obj.Laser.RequestPower(p)
    msgbox('Unable to set laser power', 'Error', 'modal');
    warning('Unable to set laser power');
end
end
obj.endWatchCursor(oldPointer);
end
obj.Busy = false;
end

function onAutoClicked(obj, "", "", "")
    obj.Busy = true;
    oldPointer = obj.startWatchCursor();
    if ~isempty(obj.Laser) & isempty(obj.DAQ) %if laser and DAQ are initialized
        autoWavelength = obj.Laser.Wavelength; % save the current wavelength
        wavelengthSelection = obj.Laser.WavelengthMin + wavelengthSpan/2 + [0 3]*1e-9;
        k = 0;
        for lambda = wavelengthSelection;
            assert (obj.Laser.RequestWavelength(lambda), 'Could not set wavelength');
            k = k + 1;
            [~, voltageSelection(k)] = obj.DAQ.RequestVoltage;
        end
        topIndex = find (voltageSelection == max(voltageSelection));
        newPower = obj.Laser.Power*0.4/voltageSelection(topIndex);
        if newPower > max(obj.Laser.Powers)
            newPower = max(obj.Laser.Powers);
        end
        disp(sprintf('New laser power is %2.3f', newPower));
    end
    if ~obj.Laser.RequestPower(newPower)
        warning('Unable to set laser power');
    end
    assert (obj.Laser.RequestWavelength(autoWavelength), 'Could not set wavelength');
    % set back the previous wavelength
    end
end
obj.endWatchCursor(oldPointer);
obj.Busy = false;
end

function onWavelengthEntered(obj, "", "", "")
    obj.Busy = true;
    oldPointer = obj.startWatchCursor();
    if ~isempty(obj.Laser)
        w = 1e-9*str2double(obj.wavelengthEdit.String);
        if ~isnumeric(w):
            warndlg('Non-numeric value entered');
        end
        if obj.Laser.RequestWavelength(w)
            msgbox('Unable to set laser wavelength', 'Error', 'modal');
            warning('Unable to set laser wavelength');
        end
        obj.Laser = obj.Laser;
    end
    obj.endWatchCursor(oldPointer);
    obj.Busy = false;
end

function onLNCClicked(obj, "", "", "")
    obj.Busy = true;
    oldPointer = obj.startWatchCursor();
    try
        if obj.lnButton.Value
            if ~obj.Laser.RequestLNSwitch(1);
                obj.lnButton.Value = 0;
            end;
        else
            if ~obj.Laser.RequestLNSwitch(0);
                obj.lnButton.Value = 1;
            end;
        end
    end
end
catch ME
    msgbox(ME.message, 'Error', 'modal');
    warning(getReport(ME));
end
obj.endWatchCursor(oldPointer);
obj.Busy = false;
end

function updatePanel(obj, "", "")
    %- Performs steps for updating the panel
    obj.panel.Position = obj.Position;
    obj.panel.Units = obj.Units;
    obj.panel.Parent = obj.Parent;
    if isobject(obj.Laser)
        obj.namePopup.Enable = 'off';
        obj.shutdownButton.Enable = 'on';
        obj.initButton.Enable = 'off';
        if obj.Laser.State == interfaces.LaserState.Off
            if strcmp(obj.namePopup.String(obj.namePopup.Value), 'OclaroTL5000')
                obj.InButton.Enable = 'on';
        end
        obj.powerEdit.Enable = 'on';
        if isobject(obj.DAQ)
            obj.autoButton.Enable = 'on';
        end
        obj.powerEdit.TooltipString = ...
            sprintf('%Enter value between %2.3f %2.3f', 1e3*obj.Laser.Powers(1), 1e3*obj.Laser.Powers(end));
        obj.powerEdit.String = num2str(1e3*obj.Laser.Power, '%3.3f');
        obj.wavelengthEdit.Enable = 'on';
        obj.wavelengthEdit.TooltipString = ...
            sprintf('%Enter value between %4.3f %4.3f', 1e9*obj.Laser.WavelengthMin, 1e9*obj.Laser.WavelengthMax);
        obj.wavelengthEdit.String = num2str(1e9*obj.Laser.Wavelength, '%4.3f');
        else
            obj.powerEdit.Enable = 'off';
            obj.autoButton.Enable = 'off';
            obj.wavelengthEdit.Enable = 'off';
            obj.namePopup.Enable = 'on';
            obj.InButton.Enable = 'off';
            obj.initButton.Enable = 'on';
            obj.powerEdit.TooltipString = '';
            obj.wavelengthEdit.TooltipString = '';
        end
        if obj.Laser.State == interfaces.LaserState.LN
            obj.InButton.Value = 1;
        else
            obj.InButton.Value = 0;
        end
    end
    obj.Busy = false;
end

function updateLaser(obj, "", "")
    %- Performs steps for updating the laser
    if isempty(obj.hwavelength)
        delete(obj.hwavelength)
    end
    if isempty(obj.hpower)
        delete(obj.hpower)
function oldPointer = startWatchCursor(obj)
    fHandle = obj.Parent;
    oldPointer = get(fHandle, 'Pointer');
    set(fHandle, 'Pointer', 'watch');
end

function endWatchCursor(obj, oldPointer)
    fHandle = obj.Parent;
    set(fHandle, 'Pointer', oldPointer);
end

function parseInputs(obj, varargin)
    % Parses inputs
    parser = inputParser;
    parser.FunctionName = 'ColorPanel';
    addParameter(parser, 'Parent', get(groot, 'CurrentFigure'))
    addParameter(parser, 'Position', get(groot, 'DefaultUicontrolPosition'))
    addParameter(parser, 'Units', get(groot, 'DefaultUicontrolUnits'))
    parse(parser, varargin{:});
    if isempty(parser.Results.Parent)
        parent = gcf; % Use current figure for parent if none specified
    else
        parent = parser.Results.Parent;
    end
    obj.Parent = parent;
    obj.Units = parser.Results.Units;
    obj.Position = parser.Results.Position;
end

end
properties (Transient, SetAccess = private, SetObservable)
RoughSpectrum % Plot data in the format [[lambda, power], ...] for main gui
ScanResolutionUser = 100
% number in pm that determines the step size of the narrow band
WavelengthStartSelected
WavelengthEndSelected
% start and end wavelengths selected in the popup menu
WavelengthScanDuration = 15
% Duration of one wavelength tuning in seconds

end

properties (Access = private, SetObservable)
panel
scanButton
loadRoughButton
resonancePopup
resolutionEdit
estimatedScanDurationText
resonanceWavelengthsText
scanFineButton
loadFineButton
laserWavelengthEdit
maxSensitivityWavelengthText
resonnaces = []
setDefaultResonance = 0

end

events
PanelChanged

end

methods
function obj = ResonancePanel(varargin)
parseInputs(obj, varargin);

obj.loadWavelengthScanDurationSetting();
% Build the resonance panel
obj.buildPanel();

addlistener(obj.panel, 'ObjectBeingDestroyed', @(~,~) obj.delete);
addlistener(obj, 'PanelChanged', @obj.updatePanel);
addlistener(obj, 'Laser', 'PostSet', @obj.updatePanel);
addlistener(obj, 'DAQ', 'PostSet', @obj.updatePanel);
addlistener(obj, 'Sensor', 'PostSet', @obj.updatePanel);
notify(obj, 'PanelChanged'); % make sure everything is up to date

end

methods (Access = private)

function buildPanel(obj)
% Builds our main panel
obj.panel = uipanel('Parent', obj.Parent, ...
    'Title', 'Resonance', ...
    'Units', obj.Units, ...
    'Position', obj.Position);

obj.scanButton = uicontrol('Parent', obj.panel, ...
    'Style', 'pushbutton', ...
    'String', 'Scan WB Spectrum', ...
    'Units', 'pixels', ...
    'TooltipString', 'Click to start wideband spectrum scan', ...
    'Position', [20, 130, 130, 30], ...
    'Callback', @obj.onScanButtonClick);
obj.loadRoughButton = uicontrol('Parent', obj(panel, ...
  'Style', 'pushbutton';
  'String', 'Load WB Spectrum';
  'Enable', 'on';
  'Units', 'pixels';
  'TooltipString', 'Click to load previously saved wideband spectrum';
  'Position', [150, 130, 130, 30];
  'Callback', @obj.onLoadRoughButtonClick);

obj.resonanceWavelengthsText = uicontrol('Parent', obj(panel, ...
  'Style', 'text';
  'String', 'Select Resonance:';
  'Enable', 'off';
  'HorizontalAlignment', 'left';
  'Units', 'pixels';
  'Position', [20, 115, 260, 15];
  'ButtonDownFcn', @obj.onResWavsClicked);

obj.resonancePopup = uicontrol('Parent', obj(panel, ...
  'Style', 'popup';
  'String', {'No scan performed'};
  'TooltipString', 'Select a resonance to enable single resonance scan (recommended closest to 1550 nm)';
  'Units', 'pixels';
  'Position', [20, 95, 260, 20];
  'Callback', @obj.onResonancePopupSelectionChanged);

ui_control('Parent', obj(panel, ...
  'Style', 'text';
  'String', 'Scan resolution [pm]:';
  'HorizontalAlignment', 'right';
  'Units', 'pixels';
  'Position', [20, 70, 210, 20];
  obj.resolutionEdit = uicontrol('Parent', obj(panel, ...
    'Style', 'edit';
    'Units', 'pixels';
    'String', num2str(obj.ScanResolutionUser, '%i');
    'HorizontalAlignment', 'left';
    'Position', [235, 70, 45, 20];
    'Callback', @obj.onResolutionChanged);

obj.estimatedScanDurationText = uicontrol('Parent', obj(panel, ...
  'Style', 'text';
  'String', 'Estimating scan time...';
  'HorizontalAlignment', 'right';
  'Units', 'pixels';
  'Position', [20, 55, 260, 15]);

obj.scanFineButton = uicontrol('Parent', obj(panel, ...
  'Style', 'pushbutton';
  'String', 'Scan Single Resonance';
  'TooltipString', 'Click to start single resonance scan';
  'Units', 'pixels';
  'Position', [20, 25, 130, 30];
  'Callback', @obj.onScanFineButtonClick);

obj.loadFineButton = uicontrol('Parent', obj(panel, ...
  'Style', 'pushbutton';
  'String', 'Load Single Resonance';
  'Enable', 'on';
  'TooltipString', 'Click to load previously saved single resonance spectrum';
  'Units', 'pixels';
  'Position', [150, 25, 130, 30];
  'Callback', @obj.onLoadFineButtonClick);

obj.maxSensitivityWavelengthText = uicontrol('Parent', obj(panel, ...
  'Style', 'text';
  'String', 'Max Sensitivity Wavelength:';
  'Enable', 'inactive';
  'HorizontalAlignment', 'left';
  'Units', 'pixels';
  'Position', [20, 5, 260, 15];
  'ButtonDownFcn', @obj.onMaxSensWavClicked);
function updatePanel(obj,˜)
% Called up update the panel
obj.panel.Position = obj.Position;
obj.panel.Units = obj.Units;
obj.panel.Parent = obj.Parent;

if˜isObject(obj.Laser)&&˜isempty(obj.Spectrum)
end

if˜isObject(obj.DAO)&&˜isObject(obj.Laser)
  obj.scanButton.Enable = 'on';
  obj.loadRoughButton.Enable = 'on';
else
  obj.scanButton.Enable = 'off';
  obj.loadRoughButton.Enable = 'off';
  obj.resonances = [];
  obj.RoughSpectrum = [];
  % clears rough spectrum when laser is turned off
end

if˜isempty(obj.resonances)
  obj.resonancePopup.String = cellfun(@(x)...
  [GUI.num2eng1e3(x.WavelengthMin) 'm' ' ' ...
  GUI.num2eng1e3(x.WavelengthMax) 'm'], ...
  obj.resonances, 'un', 0);
  obj.resonancePopup.Enable = 'on';
  obj.resonanceWavelengthsText.Enable = 'inactive';
  if obj.setDefaultResonance
    obj.resonancePopup.Value = obj.setDefaultResonance;
    obj.setDefaultResonance = 0;
  end
  obj.onResonancePopupSelectionChanged();
  obj.scanFineButton.Enable = 'on';
  obj.resolutionEdit.Enable = 'on';
  obj.resolutionEdit.String = ...        
  num2str(obj.ScanResolutionUser, '%i');
  set(obj.resolutionEdit, 'ToolTip', 'Enter an integer between 1 and 1000');
  obj.estimatedScanDurationText.Visible = 'on';
else
  obj.resonancePopup.String = {'No scan performed'};
  obj.resonancePopup.Value = 1;
  obj.resonancePopup.Enable = 'off';
  obj.resonanceWavelengthsText.Enable = 'off';
  obj.scanFineButton.Enable = 'off';
  obj.resolutionEdit.Enable = 'off';
  set(obj.resolutionEdit, 'ToolTip', ' '); 
  obj.estimatedScanDurationText.Visible = 'off';
end

if˜isempty(obj.Spectrum)
  obj.maxSensitivityWavelengthText.String = ...
  ['Max Sensitivity Wavelength: ' ...
  num2str(obj.Spectrum.LambdaMaxSensitivity*1e9, '%4.3f') ' nm'];
  obj.updateSpectrumOperatingPoint();
else
  obj.maxSensitivityWavelengthText.String = '';
end

if˜isempty(obj.Spectrum)&&˜isObject(obj.Laser)
  obj.maxSensitivityWavelengthText.Enable = 'inactive';
else
  obj.maxSensitivityWavelengthText.Enable = 'off';
end
obj.Busy = false;
function onScanButtonClicked(obj, ",", ",")
    obj.Busy = true;
    oldPointer = obj.startWatchCursor();
    if isobject(obj.Laser) & isobject(obj.DAQ)
        algo = algorithms.Algorithm(obj.Laser, obj.DAQ, obj.Sensor);
        obj.resonances = [];
        notify(obj, 'PanelChanged');
        obj.resonances = algo.GetRoughSpectrum();
        if isempty(obj.resonances); return; end;
        obj.RoughSpectrum = cat(1, cell2mat(cellfun(@(x) x.Spectrum, obj.resonances, 'un' , 0)));
        obj.setDefaultResonance = obj.findClosestResonanceTo1550;
        notify(obj, 'PanelChanged');
        end
    obj.endWatchCursor(oldPointer);
    obj.Busy = false;
end

function onLoadRoughButtonClicked(obj, ",", ",")
    file = GUI.findfile('*.mat');
    if all(file)
        z = load(file, 'resonances');
        assert('isempty(z)', 'Rough Spectrum file error!');
        obj.resonances = z.resonances;
        obj.RoughSpectrum = cat(1, cell2mat(cellfun(@(x) x.Spectrum, obj.resonances, 'un' , 0)));
        obj.setDefaultResonance = obj.findClosestResonanceTo1550;
        notify(obj, 'PanelChanged');
    end
end

function n = findClosestResonanceTo1550(obj)
    for i = 1:length(obj.resonances)
        dLambda(i) = 1e9*obj.resonances{i}.WavelengthMin - 1550;
    end
    n = find((diff(dLambda)>=0)==1);
    if isempty(n)
        n=length(dLambda);
    end
end

function onScanFineButtonClicked(obj, ",", ",")
    obj.Busy = true;
    oldPointer = obj.startWatchCursor();
    if isempty(obj.resonances)
        obj.Spectrum = [];
        notify(obj, 'PanelChanged');
        resonance = obj.resonances{obj.resonancePopup.Value};
        algo = algorithms.Algorithm(obj.Laser, obj.DAQ, obj.Sensor); % TODO: algo gets to live on this object
        obj.Spectrum = algo.GetFineSpectrum(resonance, obj.ScanResolutionUser);
        %this triggers plotting of fine spectrum by MainGUI SetSpectrum
        if isempty(obj.Spectrum); return; end;
        assert(obj.Laser.RequestWavelength(obj.Spectrum.LambdaMaxSensitivity), 'Problem setting wavelength to max sensitivity');
        obj.Spectrum = obj.Spectrum; % alternative way to refresh the plot
        notify(obj, 'PanelChanged');
    end
    obj.endWatchCursor(oldPointer);
    obj.Busy = false;
end

function onLoadFineButtonClicked(obj, ",", ",")
    file = GUI.findfile('*.mat');
    if all(file)
fileContents = whos('−file', file);
match = false;
for i=1:length(fileContents)
    match = match||strcmp(fileContents(i).name, 'spectrum');
end
if match
    x = load(file, 'spectrum');
else
    warndlg('No Narrow Band Spectrum Data found in the file');
    return;
end
try
    obj.Spectrum = x.spectrum;
    % Setting Spectrum property triggers the event
    % that refreshes fine spectrum plot by MainGUI SetSpectrum
    catch ME
        if strcmp(ME.identifier, 'MATLAB nonExistentField')
            warndlg('No Narrow Band Spectrum Data found in the file');
            return;
        end
    end
    obj.updateSpectrumOperatingPoint();
    notify(obj, 'PanelChanged');
end
end

function onMaxSensWavClicked(obj, , )
obj.Busy = true;
oldPointer = obj.startWatchCursor();
persistent clickCounter;
if isempty(clickCounter)
    clickCounter = 1;
pause(0.3);
    if clickCounter == 1
        clickCounter = [];
    end
else
    clickCounter = [];
    newLambdaMaxSensitivity = 1e−9*str2double(inputdlg('Enter New Max Sensitivity Wavelength [nm]:', ...
            'Input', 1, {sprintf('%4.3f', 1e9*obj.Spectrum.LambdaMaxSensitivity)}));
    if isempty(newLambdaMaxSensitivity)
        newLambdaMaxSensitivity = obj.Spectrum.LambdaMaxSensitivity;
    end
    obj.Spectrum.LambdaMaxSensitivity = newLambdaMaxSensitivity;
    obj.Spectrum = obj.Spectrum;
    notify(obj, 'PanelChanged');
end
if isobject(obj.Laser)&isempty(obj.Spectrum)
    obj.Laser.RequestWavelength(obj.Spectrum.LambdaMaxSensitivity);
    obj.Laser = obj.Laser;
    obj.Spectrum = obj.Spectrum;
end
obj.endWatchCursor(oldPointer);
obj.Busy = false;
end

function onResonancePopupSelectionChanged(obj, , )
    obj.updateScanDurationString();
    notify(obj, 'PanelChanged');
end

function onResWavsClicked(obj, , )
    obj.Busy = true;
    oldPointer = obj.startWatchCursor();
persistent clickCounterRes;
if isempty(clickCounterRes)
```matlab
353  clickCounterRes = 1;
354  pause(0.3);
355  if clickCounterRes == 1
356     clickCounterRes = [];
357  end
358  else
359     clickCounterRes = [];
360     % double clicked
361     prompt = {'Start \lambda : ', 'End \lambda : '};
362     dlg_title = 'Edit resonance';
363     num_lines = [1 20];
364     defaultans = {num2str(1e9*obj.resonances{obj.resonancePopup.Value}.WavelengthMin, '%4.3f'), ...}
365      num2str(1e9*obj.resonances{obj.resonancePopup.Value}.WavelengthMax, '%1.3f')};
366     options.Interpreter = 'tex';
367     answer = inputdlg(prompt, dlg_title, num_lines, defaultans, options);
368     if isempty(answer)
369        WMin = 1e-9*str2double({1}); WMax = 1e-9*str2double({2});
370        warnlg('Invalid Entry');
371     else
372        if WMin >= WMax
373           warnlg('End must be larger than start'); return;
374        end
375        obj.resonances{obj.resonancePopup.Value}.WavelengthMin = WMin;
376        obj.resonances{obj.resonancePopup.Value}.WavelengthMax = WMax;
377     end
378     notify(obj, 'PanelChanged');
379     obj.endWatchCursor(oldPointer);
380     obj.Busy = false;
381  end
382
383  function onResolutionChanged(obj, ", ", ", ")
384     num = str2double(obj.resolutionEdit.String);
385     if (isempty(num) || num<=0 || num>1000)
386        obj.resolutionEdit.String = num2str(obj.ScanResolutionUser, '%i');
387        warnlg('Resolution value is an integer value in the range 1--1000 pm');
388     else
389        obj.ScanResolutionUser = ceil(num);
390        obj.resolutionEdit.String = num2str(obj.ScanResolutionUser, '%4.0f');
391     end
392
393     obj.updateScanDurationString();
394  end
395
396  function loadWavelengthScanDurationSetting(obj, ")
397     % Load saved default values for scan duration
398     if exist('defaultSettings.mat', 'file')==2
399        vars=load('defaultSettings.mat', 'WavelengthScanDuration');
400        if isfield(vars, 'WavelengthScanDuration')
401            obj.WavelengthScanDuration = vars.WavelengthScanDuration;
402        end
403     end
404
405  end
406
407  function updateScanDurationString(obj, ")
408     r = obj.resonances(obj.resonancePopup.Value);
409     dataPointsNumberOfEstimate = ...
410        floor(1e9*(r{1}.WavelengthMax - r{1}.WavelengthMin)/(1e-3*obj.ScanResolutionUser)) + 1;
411     obj.estimatedScanDurationText.String = sprintf('%0.2f data points in %0.2f minutes'.
412        "...
```
dataPointsNumberEstimate, ceil(dataPointsNumberEstimate*obj.
WavelengthScanDuration/60));
end

function updateSpectrumOperatingPoint(obj, ~)
% when new spectrum is loaded or laser has a new wavelength,
% the operating wavelength and voltage are calculated from the
% spectrum
if isempty(obj.Spectrum.LambdaMaxSensitivity)
    [maxR, ~, ~] = algorithms.AnalyzeSpectrum(obj.Spectrum);
    obj.Spectrum.LambdaMaxSensitivity = maxR(1);
end
obj.Spectrum.VMaxSensitivity = ...
    algorithms.MapThisLambda(obj.Spectrum.LambdaMaxSensitivity,...
    obj.Spectrum.Spectrum);
if isobject(obj.Laser)
else
end
obj.Spectrum.OperatingV = ...
    algorithms.MapThisLambda(obj.Spectrum.OperatingLambda,...
    obj.Spectrum.Spectrum);
end

function oldPointer = startWatchCursor(obj)
    fHandle = obj.Parent;
    oldPointer = get(fHandle, 'Pointer');
    set(fHandle, 'Pointer', 'watch');
end

function endWatchCursor(obj, oldPointer)
    fHandle = obj.Parent;
    set(fHandle, 'Pointer', oldPointer);
end

function parseInputs(obj, varargin)
    % Parses inputs
    parser = inputParser;
    parser.FunctionName = 'ColorPanel';
    addParameter(parser, 'Parent', get(groot, 'CurrentFigure'))
    addParameter(parser, 'Position', get(groot, 'DefaultUicontrolPosition'))
    addParameter(parser, 'Units', get(groot, 'DefaultUicontrolUnits'))
    addParameter(parser, 'Laser', false)
    parse(parser, varargin{:});
    if isempty(parser.Results.Parent)
        parent = gcf; % Use current figure for parent if none specified
    else
        parent = parser.Results.Parent;
    end
    obj.Parent = parent;
    obj.Units = parser.Results.Units;
    obj.Position = parser.Results.Position;
    obj.Laser = parser.Results.Laser;
end
```matlab
classdef ResultsWindow < handle
    %ResultsWindow opens a new window to display results

    properties (Constant)
        enlargedSubplotPosition = [185, 35, 400, 300];
    end

    properties (Transient, AbortSet)
        Units % Units for this panel
        Position % On parent in Units
        Parent % Figure, uitable, or uipanel
    end

    properties (SetObservable)
        Spectrum % ISpectrum object used for calibration
        Sensor % Sensor object to use
        Calibration % Calibration data object
        Results % Results data object
        referenceLevelShift % adjustments to reference level via GUI
        referenceLevelShiftIncremental % increments of the shift
        leftEdgeSelection
        plotResultsWindowFlag = 0; % Flag that signals the time to plot results
        Busy = false;
    end

    properties (Access = private)
        infoPanel
        figureHandle % Figure Handle for the window
        zoomObj
        dcmObj
        brushObj
        dualEdgeCalibrationButton
        calibrationPreviewButton
        checkingCalibrationTurnedOff
        dualEdgingTurnedOff
        saveButton
        dataTipButton
        zoomButton
        zoomOutButton
        brushButton
        edgeButton
        revertButton
        hSubplots % Array of handles to 4 subplots
        hSubplotsAxes % Array of handles to data of 4 subplots
        calAxisHandle
        calPlotHandle
        subplotSelected
        subplotAxesProperties
        yMultiplier
        xMultiplier
        previousPosition
        ResultsNewRefLevel % Results data object with updated ref level
        subplotTitles % all the titles of the subplots
    end

    events
        WindowChanged
        RequestRecalibration
```
methods

function obj = ResultsWindow(varargin)
    parseInputs(obj, varargin {});
    addlistener(obj, 'WindowChanged', @obj.updateWindow);
    addlistener(obj, 'referenceLevelShift', 'PostSet', @obj.applyReferenceLevelShift);
    addlistener(obj, 'RequestRecalibration', @obj.onRequestRecalibration);
end

function obj = plotResultsWindow(obj)
    obj.subplotSelected = 0;
    obj.previousPosition = {(160, 240, 200, 135), [420, 240, 200, 135], ...
    [160, 35, 200, 135], [420, 35, 200, 135], [170, 50, 400, 300]};
    obj.referenceLevelShiftIncremental = [0 0 0 0];
    obj.leftEdgeSelection = [];
    obj.ResultsNewRefLevel.t = obj.Results.t;
    obj.ResultsNewRefLevel.v = obj.Results.v;
    obj.ResultsNewRefLevel.vLin = obj.Results.vLin;
    obj.ResultsNewRefLevel.vNonLin = obj.Results.vNonLin;
    obj.ResultsNewRefLevel.eLin = obj.Results.eLin;
    obj.ResultsNewRefLevel.eNonLin = obj.Results.eNonLin;

    obj.subplotTitles = {'Linear Calibration − V', ...
    'Nonlinear Calibration − V', 'Linear Calibration − E', ...
    'Nonlinear Calibration − E'};

    obj.buildWindow();
end
end

methods (Access = private)

function buildWindow(obj)
    WIDTH = 640;
    HEIGHT = 400;
    windowSize = [obj.Position(1), obj.Position(2), WIDTH, HEIGHT];
    if isempty(obj.figureHandle) || isValid(obj.figureHandle)
        f = figure('Parent', obj.Parent, 'Visible', 'on', ...
        'Units', obj.Units, 'Position', windowSize, 'Resize', 'off', ...
        'MenuBar', 'none', 'IntegerHandle', 'off', 'Name', 'Results');
        obj.figureHandle = f;
    end
end

% Getting the data and general units

obj.calibrationPreviewButton = uicontrol('Parent', obj.figureHandle, ...
'Style', 'togglebutton', ...
'String', 'Calibration Preview', ...
'Units', 'pixels', ...
'Position', [5, 95, 100, 30], ...
'Callback', @obj.onCalibrationPreviewButtonClick);

obj.dualEdgeCalibrationButton = uicontrol('Parent', obj.figureHandle, ...
'Style', 'togglebutton', ...
'String', '<html><div style="text-align:center">Dual Edge<br>Calibration</div>', ...
'Units', 'pixels', ...
'Position', [5, 50, 100, 40], ...
'Callback', @obj.onDualEdgeCalibrationButtonClick);

obj.saveButton = uicontrol('Parent', obj.figureHandle, ...
'Style', 'pushbutton', ...
'String', '<html><div style="text-align:center">Save Changed<br>Data</div>', ...
'Units', 'pixels', ...
'Position', [5, 5, 100, 40] ... 
'Callback', @obj.onsaveButtonClicked);

% Plot the subplots
obj.hSubplots(1) = subplot(2, 2, 1);
set(obj.hSubplots(1), 'Units', 'pixels' ....
'Position', obj.previousPosition{1});
[x, y, xUnitSuffix, yUnitSuffix] = obj.getSubPlotData(1);
[obj.hSubplotsAxes(1), obj.xMultiplier(1), obj.yMultiplier(1)] = ...
obj.plotLabelResults(x, y, xUnitSuffix, yUnitSuffix);
set(obj.hSubplots(1), 'ButtonDownFcn', @obj.onSubplotClicked);
title(obj subplotTitles{1}, 'FontSize', 9);

obj.hSubplots(2) = subplot(2, 2, 2);
set(obj.hSubplots(2), 'Units', 'pixels' ....
'Position', obj.previousPosition{2});
[x, y, xUnitSuffix, yUnitSuffix] = obj.getSubPlotData(2);
[obj.hSubplotsAxes(2), obj.xMultiplier(2), obj.yMultiplier(2)] = ...
obj.plotLabelResults(x, y, xUnitSuffix, yUnitSuffix);
set(obj.hSubplots(2), 'ButtonDownFcn', @obj.onSubplotClicked);
title(obj subplotTitles{2}, 'FontSize', 9);

obj.hSubplots(3) = subplot(2, 2, 3);
set(obj.hSubplots(3), 'Units', 'pixels' ....
'Position', obj.previousPosition{3});
[x, y, xUnitSuffix, yUnitSuffix] = obj.getSubPlotData(3);
[obj.hSubplotsAxes(3), obj.xMultiplier(3), obj.yMultiplier(3)] = ...
obj.plotLabelResults(x, y, xUnitSuffix, yUnitSuffix);
set(obj.hSubplots(3), 'ButtonDownFcn', @obj.onSubplotClicked);
title(obj subplotTitles{3}, 'FontSize', 9);

obj.hSubplots(4) = subplot(2, 2, 4);
set(obj.hSubplots(4), 'Units', 'pixels' ....
'Position', obj.previousPosition{4});
[x, y, xUnitSuffix, yUnitSuffix] = obj.getSubPlotData(4);
[obj.hSubplotsAxes(4), obj.xMultiplier(4), obj.yMultiplier(4)] = ...
obj.plotLabelResults(x, y, xUnitSuffix, yUnitSuffix);
set(obj.hSubplots(4), 'ButtonDownFcn', @obj.onSubplotClicked);
title(obj subplotTitles{4}, 'FontSize', 9);

notify(obj, 'WindowChanged');
end

function updateWindow(obj, " ")
% Performs steps for updating the window
if isobject(obj.figureHandle)
  if obj.calibrationPreviewButton.Value
    obj.dualEdgeCalibrationButton.Enable = 'off';
    obj.saveButton.Enable = 'off';
  else
    if obj.checkingCalibrationTurnedOff
      obj.dualEdgeCalibrationButton.Enable = 'on';
      obj.saveButton.Enable = 'on';
      obj.checkingCalibrationTurnedOff = false;
    end
  end

  if obj.dualEdgeCalibrationButton.Value
    obj.calibrationPreviewButton.Enable = 'off';
    obj.saveButton.Enable = 'off';
  else
    if obj.dualEdgingTurnedOff
      obj.calibrationPreviewButton.Enable = 'on';
      obj.saveButton.Enable = 'on';
      obj.dualEdgingTurnedOff = false;
    end
  end
end

194
if obj subplotSelected >=1&&obj subplotSelected <=4 ... &obj dualEdgeCalibrationButton.Value
obj calibrationPreviewButton.Enable = off;
obj dualEdgeCalibrationButton.Enable = off;
else if "obj subplotSelected
obj calibrationPreviewButton.Enable = on';
obj dualEdgeCalibrationButton.Enable = 'on';
else if obj subplotSelected ==5
obj calibrationPreviewButton.Enable = 'on';
obj dualEdgeCalibrationButton.Enable = 'off';
end
end
obj Busy = false;
end

function onSubplotClicked(obj, event, '')
if "obj subplotSelected
obj subplotSelected = obj clickedSubplot(event);
if "obj subplotSelected; return; end
makeEnlarged = [0 0 0 0];
makeEnlarged(obj subplotSelected) = 1;
for i = 1:4
if makeEnlarged(i)
% Enlarge the subplot
obj previousPosition{i} = get(obj hSubplots(i), 'Position');
set(obj hSubplots(i), 'Position', obj enlargedSubplotPosition);
else
% Make invisible/unClickable
set(obj hSubplots(i), 'HandleVisibility', 'off', 'Visible', 'off');
set(obj hSubplotsAxes(i), 'HandleVisibility', 'off', 'Visible', 'off');
end
end
obj ToolbarOn();
else
set(obj hSubplots(obj subplotSelected), ...
'Position', obj previousPosition{obj subplotSelected});
for i = 1:4
% Make all visible/Clickable again
set(obj hSubplots(i), 'HandleVisibility', 'on', 'Visible', 'on');
set(obj hSubplotsAxes(i), 'HandleVisibility', 'on', 'Visible', 'on');
end
obj ToolbarOff();
obj subplotSelected = 0;
end
notify(obj, 'WindowChanged');
end

function number = clickedSubplot(obj, event)
switch event.Type
    case 'axes'
        number = find(strcmp(event.Title.String, obj subplotTitles)=1);
    case 'figure'
        number = 0;
end
end

function onCalibrationPreviewButtonClick(obj, '', '')
if obj calibrationPreviewButton.Value
obj subplotSelected = 5;
for i = 1:4
% Make invisible/unClickable
set(obj hSubplots(i), 'HandleVisibility', 'off', ...
'Visible', 'off', 'Position', obj previousPosition{i});
set(obj hSubplotsAxes(i), 'HandleVisibility', 'off', ...
'Visible', 'off');
end
obj calAxisHandle = axes('Parent', obj.figureHandle, ...
'Units', 'pixels');
function onDualEdgeCalibrationButtonClicked(obj, ˜, ˜)
if obj.dualEdgeCalibrationButton.Value
makeEnlarged = [1 0 0 0];
obj.subplotSelected = 1;
for i = 1:4
if makeEnlarged(i)
% Enlarge the subplot
obj.previousPosition{obj.subplotSelected} = ...
get(obj.hSubplots(i), 'Position');
set(obj.hSubplots(i) ...,  
'Position', obj.enlargedSubplotPosition, ...  
'ButtonDownFcn', '');
set(obj.hSubplots(i), 'HandleVisibility', 'on', 'Visible', 'on');
set(obj.hSubplotsAxes(i), 'HandleVisibility', 'on', 'Visible', 'on');
else  
% Make invisible/unclickable
set(obj.hSubplots(i), 'HandleVisibility', 'off', 'Visible', 'off');
set(obj.hSubplotsAxes(i), 'HandleVisibility', 'off', 'Visible', 'off');
end
end
obj.dualEdgingTurnedOff = false;
obj.ToolbarOn();
else
obj.toolButtonsOff([1 0 1 1]);
obj.ToolbarOff();
obj.dualEdgingTurnedOff = true;
set(obj.hSubplots(1), 'Position', obj.previousPosition{1});
for i = 1:4
% Make all visible/clickable again
set(obj.hSubplots(i), 'HandleVisibility', 'on', 'Visible', 'on');
set(obj.hSubplotsAxes(i), 'HandleVisibility', 'on', 'Visible', 'on');
set(obj.hSubplots(i), 'Position', obj.previousPosition{i});
end
set(obj.hSubplots(obj.subplotSelected), 'ButtonDownFcn', @obj.onSubplotClicked);
obj.subplotSelected = 0;
if˜isEmpty(obj.leftEdgeSelection)
notify(obj, 'RequestRecalibration');
end
end
notify(obj, 'WindowChanged');
function ToolbarOn ( obj )
    zoomIconImgData = imread ( './+GUI/zoom_icon.png' );
    obj.zoomButton = uicontrol ( 'Parent', obj.figureHandle, ....
        'Style', 'togglebutton' ....
        'CData', zoomIconImgData ....
        'Units', 'pixels' ....
        'Position', [170, 365, 30, 30] ....
        'Callback', @obj.onZoomButtonClicked );

    zoomOutIconImgData = imread ( './+GUI/zoom_out_icon.png' );
    obj.zoomOutButton = uicontrol ( 'Parent', obj.figureHandle, ....
        'Style', 'pushbutton' ....
        'CData', zoomOutIconImgData ....
        'Units', 'pixels' ....
        'TooltipString', 'Zoom-out to see all data' ....
        'Position', [210, 365, 30, 30] ....
        'Callback', @obj.onZoomOutButtonClicked );

    dataTipIconImgData = imread ( './+GUI/dataTip_icon.png' );
    obj.dataTipButton = uicontrol ( 'Parent', obj.figureHandle, ....
        'Style', 'togglebutton' ....
        'CData', dataTipIconImgData ....
        'Units', 'pixels' ....
        'Position', [250, 365, 30, 30] ....
        'Callback', @obj.onDataTipButtonClick );

    elseif˜obj.calibrationPreviewButton.Value
        if obj.dualEdgeCalibrationButton.Value
            brushIconImgData = imread ( './+GUI/edge_br_icon.png' );
            obj.edgeButton = uicontrol ( 'Parent', obj.figureHandle, ....
                'Style', 'togglebutton' ....
                'CData', brushIconImgData ....
                'Units', 'pixels' ....
                'Position', [290, 365, 30, 30] ....
                'Callback', @obj.onEdgeButtonClick );
        else
            brushIconImgData = imread ( './+GUI/brush_icon.png' );
            obj.brushButton = uicontrol ( 'Parent', obj.figureHandle, ....
                'Style', 'togglebutton' ....
                'CData', brushIconImgData ....
                'Units', 'pixels' ....
                'Position', [290, 365, 30, 30] ....
                'Callback', @obj.onBrushButtonClick );
        end
    end

    dataRevertIconImgData = imread ( './+GUI/revert_icon.png' );
    obj.revertButton = uicontrol ( 'Parent', obj.figureHandle, ....
        'Style', 'pushbutton' ....
        'CData', dataRevertIconImgData ....
        'Units', 'pixels' ....
        'TooltipString', 'Revert the changes made to the reference level' ....
        'Position', [330, 365, 30, 30] ....
        'Callback', @obj.onRevertButtonClick );
end

notify ( obj, 'WindowChanged' );
end

function ToolbarOff ( obj )
    if isobject ( obj.zoomButton ); delete ( obj.zoomButton ); obj.zoomButton = []; end
    if isobject ( obj.zoomOutButton ); delete ( obj.zoomOutButton ); obj.zoomOutButton = []; end
    if isobject ( obj.brushButton ); delete ( obj.brushButton ); obj.brushButton = []; end
    if isobject ( obj.edgeButton ); delete ( obj.edgeButton ); obj.edgeButton = []; end

end
if isobject(obj.dataTipButton); delete(obj.dataTipButton); obj.dataTipButton = []; end
if isobject(obj.revertButton); delete(obj.revertButton); obj.revertButton = []; end
delete(findall(gca, 'Type', 'hgroup', 'HandleVisibility','off'));
obj.zoomObj = [];
obj.dcmObj = [];
delete(obj.brushObj);
obj.brushObj = [];
end

function onZoomButtonClicked(obj, ', ', '')
if obj.zoomButton.Value
% then turn on zoom
obj.zoomObj = zoom;
set(obj.zoomObj, 'Enable', 'on');
set(obj.hSubplots(obj subplotSelected), 'HitTest', 'on');
% turn off other tools
obj.toolButtonsOff([0 0 1 1 0]);
else
obj.toolButtonsOff([1 0 1 1 0]);
end

function onZoomOutButtonClicked(obj, ', ', '')
obj.toolButtonsOff([1 0 1 1 0]);
axis tight;
[', ', ', '] = obj.getSubPlotData (obj subplotSelected);
proxy = y ./obj.yMultiplier(obj subplotSelected);
margin1 = 0.1; margin2 = 0.1;
ylim([min(proxy) margin1 .*(max(proxy) min(proxy)) ...
max(proxy) margin2 .*(max(proxy) min(proxy))]);

function onDataTipButtonClicked(obj, ', ', '')
if obj.dataTipButton.Value
% turn off other tools buttons
obj.toolButtonsOff([1 0 1 0 1 0]);
% then turn on datacursor mode
obj.dcmObj = datacursormode;
set(obj.dcmObj, 'Enable', 'on');
set(obj.hSubplots(obj subplotSelected), 'HitTest', 'off');
end
else
obj.toolButtonsOff([1 0 1 1 0]);
set(obj.hSubplots(obj subplotSelected), 'HitTest', 'on');
end

function onBrushButtonClicked(obj, ', ', '')
if obj.brushButton.Value
% turn off other tools buttons
obj.toolButtonsOff([1 0 1 0 1 0]);
% then turn on brush mode
obj.brushObj = brush;
set(obj.brushObj, 'Enable', 'on');
set('ActionPostCallback', @obj.postBrushReferenceLevelFcn1);
else
obj.toolButtonsOff([1 0 1 1 0]);
end

function onEdgeButtonClicked(obj, ', ', '')
if obj.edgeButton.Value
% turn off other tools buttons
obj.toolButtonsOff([1 0 1 0 0 0]);
% turn on brush mode

obj.brushObj = brush;
set(obj.brushObj, 'Enable', 'on', ...
    'ActionPostCallback', @obj.postBrushReferenceLevelFcn2);
else
    obj.toolButtonsOff([1 0 1 0]);
end

function onRevertButtonClicked(obj, ".", ")
    obj.toolButtonsOff([1 0 1 0]);
    obj.referenceLevelShift = [0 0 0 0];
    axis tight;
    ["", y, "," ] = obj.getSubPlotData (obj subplotSelected);
    proxy = y ./ obj.yMultiplier (obj subplotSelected);
    margin1 = 0.1; margin2 = 0.1;
    ylim([min(proxy) margin1 *(max(proxy) - min(proxy))...
        max(proxy) + margin2 *(max(proxy) - min(proxy))]);
end

function toolButtonsOff(obj, n)
    % n = 1 Zoom button
    % n = 2 DataTip Button
    % n = 3 Selection Button
    for i = 1:length(n)
        if(n(i))
            [buttonObj, modeObj] = obj.getToolButtonObj(i);
            if ~isempty(buttonObj);
                if invalid(buttonObj)
                    buttonObj.Value = 0;
                end
            end
            if ~isempty(modeObj);
                if invalid(modeObj)
                    modeObj.Enable = 'off';
                end
            end
        end
    end
end

function [buttonObj, modeObj] = getToolButtonObj(obj, n)
    % n = 1 Zoom button
    % n = 2 DataTip Button
    % n = 3 Selection Button
    switch n
        case 1
            buttonObj = obj.zoomButton;
            modeObj = obj.zoomObj;
        case 2
            buttonObj = obj.zoomOutButton;
            modeObj = [];
        case 3
            buttonObj = obj.dataTipButton;
            modeObj = obj.dcmObj;
        case 4
            if obj.dualEdgeCalibrationButton.Value
                buttonObj = obj.edgeButton;
                modeObj = obj.brushObj;
            else
                buttonObj = obj.brushButton;
                modeObj = obj.brushObj;
            end
        case 5
            buttonObj = obj.revertButton;
            modeObj = [];
        otherwise
            buttonObj = [];
            modeObj = [];
        end
function applyReferenceLevelShift(obj, "", "")
  obj.Busy = true;
  switch obj subplotSelected
    case 1
      obj. Results. NewRefLevel. vLin = ...
      obj. Results. vLin = obj. referenceLevelShift(1);
    case 2
      obj. Results. NewRefLevel. vNonLin = ...
      obj. Results. vNonLin = obj. referenceLevelShift(2);
    case 3
      obj. Results. NewRefLevel. eLin = ...
      obj. Results. eLin = obj. referenceLevelShift(3);
    case 4
      obj. Results. NewRefLevel. eNonLin = ...
      obj. Results. eNonLin = obj. referenceLevelShift(4);
    otherwise
      return;
  end
  axes(obj. hSubplots(obj. subplotSelected));
  currentAxis = gca;
  previousXLim = currentAxis. XLim;
  previousYLim = currentAxis. YLim;
  previousYMultiplier = obj. yMultiplier(obj. subplotSelected);
  previousXMultiplier = obj. xMultiplier(obj. subplotSelected);
  [x, y, xUnitSuffix, yUnitSuffix] = obj. getSubPlotData(obj. subplotSelected);
  [obj. hSubplotsAxes(obj. subplotSelected), ...
   obj. xMultiplier(obj. subplotSelected), obj. yMultiplier(obj. subplotSelected)] = ...
   obj. plotLabeldResults(x, y, xUnitSuffix, yUnitSuffix);
  set(gca, 'XLim', previousXLim ... .*(previousXMultiplier/obj. xMultiplier(obj. subplotSelected)));
  set(gca, 'YLim', (previousYLim ... .*(previousYMultiplier/obj. yMultiplier(obj. subplotSelected)));
  set(obj. hSubplots(obj. subplotSelected), 'ButtonDownFcn', @obj. onSubplotClicked);
  title(obj. subplotTitles(obj. subplotSelected), 'FontSize', 9);
  obj.Busy = false;
end

function onsaveButtonClicked(obj, "", "")
  fileName = obj. Results. filename;
  t = obj. Results. NewRefLevel. t; %#ok<NASGU>
  v = obj. Results. NewRefLevel. v; %#ok<NASGU>
  vLin = obj. Results. NewRefLevel. vLin; %#ok<NASGU>
  vNonLin = obj. Results. NewRefLevel. vNonLin; %#ok<NASGU>
  eLin = obj. Results. NewRefLevel. eLin; %#ok<NASGU>
  eNonLin = obj. Results. NewRefLevel. eNonLin; %#ok<NASGU>
  newFolder = '.\dataFiles\';
  if exist(newFolder, 'dir'); mkdir(newFolder); end
  save([newFolder fileName dataestr(now, 'yy-mm-dd_HH-MM-SS'), '_calibrated.mat'], ....
   't', 'eLin', 'vLin', 'eNonLin', 'vNonLin');
end

function output_txt = DataTipFunction(", dataTip, event_obj)

  pos = get(event_obj, 'Position');
  output_txt = {{'t', num2str(pos(1), 4)} , ...
      {'V', num2str(pos(2), 4)}},
  dataTip. FontSize = 9;
end
function postBrushReferenceLevelFcn1(obj, ~, event_data)
    event_data.Axes.Children.BrushHandles.Children(1).FaceColorData = ...
    uint8(255.*[1;0;0;0.3]);

    if isempty(brushdata)
        obj.referenceLevelShiftIncremental(obj subplotSelected) = ...
        mean(brushdata(2,:))*obj.xMultiplier(obj subplotSelected);
        obj.referenceLevelShift(obj subplotSelected) = ...
        obj.referenceLevelShiftIncremental(obj subplotSelected);
    else
        return;
    end
end

function postBrushReferenceLevelFcn2(obj, ~, event_data)
    obj.leftEdgeSelection = ...
    event_data.Axes.Children.BrushHandles.Children(1).FaceColorData = ...
    uint8(255.*[1;0;0;0.3]);
end

function onRequestRecalibration(obj, ~, ~)
    obj.Busy = true;
    % Find selected indices
    indices(1) = find(obj.ResultsNewRefLevel.t >= ...
        obj.leftEdgeSelection(1,1).*obj.xMultiplier(1, 1));
    indices(2) = find(obj.ResultsNewRefLevel.t <= ...
        obj.leftEdgeSelection(1,end).*obj.xMultiplier(1, 1, 'last'));
    ind = indices(1):1:indices(2);

    % form a 1-vector with zeros at selected points
    rightEdgeMapping = ones(length(obj.ResultsNewRefLevel.t),1);
    rightEdgeMapping(ind) = 0;

    % Recalibration
    obj.Results = obj.Spectrum.ProcessData(...
        obj.Results.t, obj.Results.v, ...
        obj.Results.filename, obj.Sensor, rightEdgeMapping);
    obj.plotResultsWindowFlag = 1; % signal to replot the results window
    obj.Busy = false;
end

function [axisHandle, xMult, yMult] = plotLabIdResults(~, x, y, xUnitSuffix, yUnitSuffix)

    % General magnitudes setup
    yUnitPrefix = {'p', 'n', 'u', 'm', '', 'k', 'M', 'G', 'T'};
    xUnitPrefix = {'p', 'n', 'u', 'm', ''};
    ymultiplier = [1e-12 1e-9 1e-6 1e-3 1 1e3 1e6 1e9 1e12];
    xmultiplier = [1e-12 1e-9 1e-6 1e-3 1];

    % Determining which units to use
    xAvg = max(abs(x));
    xUnitIndex = find(xAvg<=le3.*xmultiplier, 1);
    xUnit = strcat(xUnitPrefix(xUnitIndex), xUnitSuffix);
    yAvg = max(abs(y));
    yUnitIndex = find(yAvg<=le3.*ymultiplier, 1);
    yMult = ymultiplier(yUnitIndex);
    yUnit = strcat(yUnitPrefix(yUnitIndex), yUnitSuffix);

    % Plotting
    axisHandle = plot(x./xMult, y./yMult);
grid on
axis tight;
proxy = y./yMult; margin1 = 0.1; margin2 = 0.1;
if (min(proxy)=max(proxy))
  ylim([min(proxy)−margin1.*(max(proxy)−min(proxy)) ... 
  max(proxy)+margin2.*(max(proxy)−min(proxy))]);
end
xlabel([('Time [ ' xUnit{1} ' ]'), 'FontSize', 9);
ylabel([('Voltage [ ' yUnit{1} ' ]'), 'FontSize', 9);
set(gca, 'FontSize', 7);
end

function [x, y, xUnitSuffix, yUnitSuffix] = getSubPlotData(obj, nSubPlot)
x = obj.ResultsNewRefLevel.t;
switch nSubPlot
  case 1
    y = obj.ResultsNewRefLevel.vLin;
    yUnitSuffix = 'V';
  case 2
    y = obj.ResultsNewRefLevel.vNonLin;
    yUnitSuffix = 'V';
  case 3
    y = obj.ResultsNewRefLevel.eLin;
    yUnitSuffix = 'V/m';
  case 4
    y = obj.ResultsNewRefLevel.eNonLin;
    yUnitSuffix = 'V/m';
  case 5
    x = obj.Calibration.tCalibration;
    y = obj.Calibration.vCalibration;
    yUnitSuffix = 'V';
end
xUnitSuffix = 's';
end

function parseInputs(obj, varargin)
  % Parses inputs
  parser = inputParser;
  parser.FunctionName = 'ColorPanel';
  addParameter(parser, 'Parent', get(groot, 'CurrentFigure'))
  addParameter(parser, 'Position', get(groot, 'DefaultUicontrolPosition'))
  addParameter(parser, 'Units', get(groot, 'DefaultUicontrolUnits'))
  addParameter(parser, 'Spectrum', [])
  parse(parser, varargin{:});
  if isempty(parser.Results.Parent)
    parent = gcf; % Use current figure for parent if none specified
  else
    parent = parser.Results.Parent;
  end
  obj.Parent = parent;
  obj.Units = parser.Results.Units;
  obj.Position = parser.Results.Position;
  obj.Spectrum = parser.Results.Spectrum;
end
classdef SensorPanel < handle
% SENSORPANEL Panel for selecting and displaying the sensor info
% UI Control for selecting a sensor

properties (Constant)
sensorFileName = 'sensors.txt';
% name of the text file where a list of sensors can be provided
% in the format name, eShiftRate on each line, for example:
% SCOS1, 50
% SCOS2, 45
% SCOS3, 350
end

properties (Transient, AbortSet)
Units % Units for this panel
Position % On parent in Units
Parent % Figure, uitab, or uipanel
end

properties (Transient, SetObservable)
Sensor % Sensor object to use
sensors % sensors list read from the file
Busy = false;
end

properties (Transient, Access = private, SetObservable)
panel
sensorNamePopup
sensorNameEdit
turtleShiftEdit
NameEditContentSizeLength = 0;
turtleShiftEditContentSizeLength = 0;
setButton
deleteButton
end

events
PanelChanged
SensorPopupChanged
NonEmptyFields
EmptyFields
end

methods
function obj = SensorPanel(varargin)
parseInputs(obj, varargin{1});
obj.readSensorsFromFile(); % read list of sensors from sensors.txt
obj.Sensor = obj.sensors(1); % set the current sensor to user custom
obj.buildPanel();
addlistener(obj.panel, 'ObjectBeingDestroyed', @(), obj.delete);
addlistener(obj, 'PanelChanged', @obj.updatePanel);
addlistener(obj, 'SensorPopupChanged', @obj.onSensorNamePopupSelectionChanged);
addlistener(obj, 'Sensor', 'PostSet', @obj.updatePanel);
addlistener(obj, 'NonEmptyFields', @obj.enableSetButton);
addlistener(obj, 'EmptyFields', @obj.disableSetButton);
notify(obj, 'PanelChanged'); % make sure everything is up to date
function set.Parent(obj, newParent)
% Validate/Set/Update
assert(isghandle(newParent), 'New Parent is expected to be a valid handle to a figure, uipanel, or uitab')
assert(ismember(newParent.Type, {'figure', 'uipanel', 'uitab'}), 'New Parent is expected to be a valid handle to a figure, uipanel, or uitab')
obj.Parent = newParent;

% Notify
notify(obj, 'PanelChanged');
end
function set.Sensor(obj, sensor)
obj.Sensor = sensor;
end

methods (Access = private)
function buildPanel(obj)
% Builds our panel

obj.panel = uipanel('Parent', obj.Parent, ...
.Title', 'Sensor', ...
'Units', obj.Units, ...
'Position', obj.Position);

names = arrayfun(@(f) f.Name, obj.sensors, 'un', 0);

obj.sensorNamePopup = uicontrol('Parent', obj.panel, ...
'Style', 'popup', ...
'String', names, ...
'Units', 'pixels', ...
'TooltipString', 'Choose a sensor', ...
'Position', [20, 50, 260, 30], ...
'Callback', @obj.onSensorNamePopupSelectionChanged);

uicontrol('Parent', obj.panel, ...
'Style', 'text', ...
'String', 'Name:', ...
'HorizontalAlignment', 'right', ...
'Units', 'pixels', ...
'Position', [20, 35, 45, 20], ...
'ButtonDownFcn', @obj.onSensorPanelDoubleClick);

obj.sensorNameEdit = uicontrol('Parent', obj.panel, ...
'Style', 'edit', ...
'Units', 'pixels', ...
'Position', [65, 35, 85, 20], ...
'KeyPressFcn', @obj.keyPressedName, ...
'TooltipString', 'Enter the name of the new sensor');

uicontrol('Parent', obj.panel, ...
'Style', 'text', ...
'String', 'E-shift:', ...
'HorizontalAlignment', 'right', ...
'Units', 'pixels', ...
'Position', [150, 35, 85, 20], ...
'ButtonDownFcn', @obj.onSensorPanelDoubleClick);

obj.sensorEShiftEdit = uicontrol('Parent', obj.panel, ...
'Style', 'edit', ...
'Units', 'pixels', ...
'HorizontalAlignment', 'left', ...
'Position', [235, 35, 45, 20], ...
'KeyPressFcn', @obj.keyPressedEShift, ...
'TooltipString', 'Enter the E-shift value in pm/(MV/m)');
obj.setButton = uicontrol('Parent', obj.panel, ....
'Style', 'pushbutton' ....
'String', 'Set' ....
'Units', 'pixels' ....
'TooltipString', 'Use the entered information as a new sensor' ....
'Position', [150, 5, 130, 30] ....
'Callback', @obj.onSetClicked);

obj.deleteButton = uicontrol('Parent', obj.panel, ....
'Style', 'pushbutton' ....
'String', 'Delete' ....
'Units', 'pixels' ....
'TooltipString', 'Delete the selected sensor' ....
'Position', [20, 5, 130, 30] ....
'Callback', @obj.onDeleteClicked);

function onSensorPanelDoubleClick(obj, "", "")
    persistent chk
    if isempty (chk)
        chk = 1;
        pause (0.5);
        % this delay distinguishes between single and double clicks
        if chk == 1
            chk = []; return;
        end
    else
        chk = [];
        obj.refreshSensorList(0);
    end
end

function onSetClicked(obj, "", "")
    name = obj.sensorNameEdit.String;
    eShift = str2double(obj.sensorEShiftEdit.String);
    if isnan(eShift)
        warndlg('Only numeric values accepted for E-shift');
        return;
    end
    sensor = interfaces.Sensor(name, eShift);
    noMatch = true;
    for i=1:length(obj.sensors)
        noMatch = noMatch&&('strcmp(name, obj.sensors(i).Name)');
    end
    if noMatch % new sensor name is not a duplicate
        obj.appendSensorToFile(sensor);
        obj.refreshSensorList(1);
    else
        warndlg('Unallowed sensor name. Either a duplicate or illegal name.');
    end
end

function onDeleteClicked(obj, "", "")
    obj.sensors(obj.sensorNamePopup.Value) = [];
    obj.sensorNamePopup.Value = 1;
    notify(obj, 'SensorPopupChanged');
    obj.overwriteSensorFileWithCurrentSensors();
    obj.refreshSensorList(0);
end

function onSensorNamePopupSelectionChanged(obj, "", "")
    obj.Sensor = obj.sensors(obj.sensorNamePopup.Value);
end

function updatePanel(obj, "")
    % Performs steps for updating the panel
obj.panel.Position = obj.Position;
obj.panel.Units = obj.Units;
obj.panel.Parent = obj.Parent;

if obj.sensorNamePopup.Value==1
  % value one is always the dummy sensor reserved for adding
  % new sensor entry
  obj.sensorNameEdit.String = '';
  obj.sensorEShiftEdit.String = '';
  obj.sensorNameEdit.Enable = 'on';
  obj.sensorEShiftEdit.Enable = 'on';
  obj.deleteButton.Enable = 'off';
  obj.disableSetButton(obj);
else
  obj.sensorNameEdit.String = obj.Sensor.Name;
  obj.sensorEShiftEdit.String = num2str(obj.Sensor.EShift);
  obj.sensorNameEdit.Enable = 'off';
  obj.sensorEShiftEdit.Enable = 'off';
  obj.deleteButton.Enable = 'on';
  obj.setButton.Enable = 'off';
end
obj.Busy = false;
end

function readSensorsFromFile(obj,˜)
  fID = fopen(obj.sensorFileName, 'rt');
  data = textscan(fID, '%s %s ', 'Delimiter', ', ');
  if ~feof(fID)
    disp('error reading sensor file');
  end
  fclose(fID);
  s(1)=interfaces.Sensor('Add New Sensor', 0);
  % first sensor is always user customizable
  for i = 1:size(data{1},1)
    s(i+1)=interfaces.Sensor(data{1,1}{i}, str2num(data{1, 2}{i}));
  end
  obj.sensors = s;
end

function appendSensorToFile(obj, sensor)
  fID = fopen(obj.sensorFileName, 'at');
  fprintf(fID, '
%s, %s', sensor.Name, num2str(sensor.EShift));
  fclose(fID);
end

function overwriteSensorFileWithCurrentSensors(obj,˜)
  fID = fopen(obj.sensorFileName, 'wt');
  for i=2:length(obj.sensors)
    fprintf(fID, '%s, %s', obj.sensors(i).Name,...
      num2str(obj.sensors(i).EShift));
    if i<length(obj.sensors)
      fprintf(fID, '
');
    end
  end
  fclose(fID);
end

function refreshSensorList(obj, selection)
  obj.readSensorsFromFile();
  names = arrayfun(@(f) f.Name, obj.sensors, 'un', 0);
  obj.sensorNamePopup.String = names;
  switch selection
    case 1
      % Set is clicked => Select the newly added sensor, last in the list
      obj.sensorNamePopup.Value = length(names);
    case 0
      if obj.sensorNamePopup.Value>length(names);
obj.sensorNamePopup.Value = 1;
end

notify(obj, 'SensorPopupChanged');
% update sensor to match popup selection
end

function keyPressedName(obj, "", E)
obj.EShiftEditContentLength = length(obj.sensorNameEdit.String);
if strcmp(E.Key, 'backspace') || strcmp(E.Key, 'delete')
obj.NameEditContentLength = obj.NameEditContentLength - 1;
else
  if isempty(E.Character) || strcmp(E.Key, 'leftarrow') || strcmp(E.Key, 'rightarrow')
    || strcmp(E.Key, 'downarrow') || strcmp(E.Key, 'uparrow');
  else
    obj.NameEditContentLength = obj.NameEditContentLength + 1;
end
if obj.NameEditContentLength <0; obj.NameEditContentLength=0; end
obj.updateSetButton();
end

function keyPressedEShift(obj, "", E)
obj.NameEditContentLength = length(obj.sensorNameEdit.String);
if strcmp(E.Key, 'backspace') || strcmp(E.Key, 'delete')
obj.EShiftEditContentLength = obj.EShiftEditContentLength - 1;
else
  if isempty(E.Character) || strcmp(E.Key, 'leftarrow') || strcmp(E.Key, 'rightarrow')
    || strcmp(E.Key, 'downarrow') || strcmp(E.Key, 'uparrow');
  else
    obj.EShiftEditContentLength = obj.EShiftEditContentLength + 1;
end
if obj.EShiftEditContentLength <0; obj.EShiftEditContentLength=0; end
obj.updateSetButton();
if obj.EShiftEditContentLength <0; obj.EShiftEditContentLength=0; end
end

function updateSetButton(obj, "", "", "")
if obj.NameEditContentLength==0||obj.EShiftEditContentLength==0 ... &&(strcmp(obj.setButton.Enable, 'on'));
notify(obj, 'EmptyFields');
return;
end
if (obj.NameEditContentLength>0&&obj.EShiftEditContentLength>0) ... &&(strcmp(obj.setButton.Enable, 'off'));
notify(obj, 'NonEmptyFields');
return;
end
end

function enableSetButton(obj, "", "")
obj.setButton.Enable = 'on';
end

function disableSetButton(obj, "", "")
obj.setButton.Enable = 'off';
end

function parseInputs(obj, varargin)
% Parses inputs

parser = inputParser;
parser.FunctionName = 'ColorPanel';

addParameter(parser, 'Parent', get(groot, 'CurrentFigure'))
addParameter(parser, 'Position', get(groot, 'DefaultUicontrolPosition'))
addParameter(parser, 'Units', get(groot, 'DefaultUicontrolUnits'))

parse(parser, varargin{:});

if isempty(parser.Results.Parent)
    parent = gcf; % Use current figure for parent if none specified
else
    parent = parser.Results.Parent;
end

obj.Parent = parent;
obj.Units = parser.Results.Units;
obj.Position = parser.Results.Position;
end
end
end

A.2.2 Functions

../+GUI/findfile.m

function [ filename ] = findfile ( filter )
%FINDFILE Finds a file with some extension

persistent lastpath;

if strcmp(lastpath,0)
    spec = [lastpath filter];
else
    spec = [];
end

if isempty(spec)
    [name, folder, ] = uigetfile(filter);
else
    [name, folder, ] = uigetfile(spec);
end

lastpath = folder;

filename = [folder name];
end

../+GUI/MainGUI.m

function MainGUI()

f = figure('Visible', 'on', 'Units', 'pixels', ....
           'MenuBar', 'none', 'IntegerHandle', 'off', ....
'Name', 'Interrogator App', 'Windowstyle', 'normal', ....
'DockControls', 'off');

WIDTH = 800;
HEIGHT = 600;

% Set window size and position
f.Resize = 'off';
screensize = get(groot, 'Screensize');
windowSize = [(screensize(3) − WIDTH) / 2, (screensize(4) − HEIGHT) / 2, WIDTH, HEIGHT];
f.Position = windowSize;

'Parent', f....
'Position', [25 360 300 130]);
dp = GUI.DAQPanel({NIdaqUSB6002.NDaqUSBFactory} ....
'Parent', f....
'Position', [25 270 300 90]);
sp = GUI.SensorPanel('Parent', f....
'Position', [25 490 300 100]);
rp = GUI.ResonancePanel('Parent', f....
'Position', [25 90 300 180]);
cp = GUI.CalibrationPanel('Parent', f....
'Position', [25 20 300 70]);
rb = GUI.BusyBar('Parent', f....
'Position', [25 5 100 15]);

h1 = subplot(3, 2, 2);
set(h1, 'Units', 'pixels'....
'Position', [400, 500, 350, 50]);
title('Rough Spectrum');

h2 = subplot(3, 2, 4);
set(h2, 'Units', 'pixels'....
'Position', [400, 70, 350, 350]);
title('Fine Spectrum');
dcm = datacursormode(f);
datacursormode on;
set(dcm, 'updatefcn'. @updateFunctionFineSpectrum);
dcm.UIContextMenu = []; 

function output_txt = updateFunctionFineSpectrum(obj, event_obj)
% Display the position of the data cursor
% obj Currently not used (empty)
% event_obj Handle to event object
% output_txt Data cursor text string (string or cell array of strings).
hFig = ancestor(event_obj.Target, 'figure');
pos = get(event_obj, 'Position');
output_txt = {{'\lambda': num2str(pos(1), 7)}, ....
{'V': num2str(pos(2), 4)}];
obj.TipHandle.Interpreter = 'tex';
obj.TipHandle.FontSize = 7;
end

function setDAQ(" . ")
oldPointer = startWatchCursor(f);
rp.DAQ = dp.DAQ;
lp.DAQ = dp.DAQ;
endWatchCursor(f, oldPointer);
end

function setLaser(" . ")
oldPointer = startWatchCursor(f);
% passes Laser object to resonance panel once it is set in laser
function setSensor(~, ~)
    oldPointer = startWatchCursor(f);
    cp.Sensor = sp.Sensor;
    rw.Sensor = sp.Sensor;
    rp.Sensor = sp.Sensor;
    endWatchCursor(f, oldPointer);
end

function setRoughSpectrum(~, ~)
    oldPointer = startWatchCursor(f);
    axes(h1);
    GUI.plotLabeldRough(rp);
    title('Rough Spectrum');
    endWatchCursor(f, oldPointer);
end

function refreshSpectrumPlot()
    oldPointer = startWatchCursor(f);
    axes(h2);
    if ~isempty(rp.Spectrum)
        GUI.plotLabeldNM(rp);
        title(['Fine Spectrum: '... ...
        GUI.num2eng1e3(rp.Spectrum.LambdaMin) 'm - '... ...
        GUI.num2eng1e3(rp.Spectrum.LambdaMax) 'm']);
    else
        cla;
        title('Fine Spectrum');
    end
    endWatchCursor(f, oldPointer);
end

function setSpectrum(~, ~)
    oldPointer = startWatchCursor(f);
    if ~isempty(lp.Laser)&&~isempty(rp.Spectrum)
    end
    cp.Spectrum = rp.Spectrum;
    rw.Spectrum = rp.Spectrum;
    refreshSpectrumPlot();
    endWatchCursor(f, oldPointer);
end

function setResults(~, ~)
    oldPointer = startWatchCursor(f);
    rw.Results = cp.Results;
    endWatchCursor(f, oldPointer);
end

function setCalibration(~, ~)
    oldPointer = startWatchCursor(f);
    rw.Calibration = cp.Calibration;
    endWatchCursor(f, oldPointer);
end

function plotResultsWindow(~, ~)
    oldPointer = startWatchCursor(f);
    if cp.plotResultsWindowFlag
        cp.plotResultsWindowFlag = 0;
        rw.plotResultsWindow;
function oldPointer = startWatchCursor(fHandle)
    oldPointer = get(fHandle, 'Pointer');
    set(fHandle, 'Pointer', 'watch');
end

function endWatchCursor(fHandle, oldPointer)
    set(fHandle, 'Pointer', oldPointer);
end

function setBusyBar(˜, ˜)
end

addlistener(dp, 'DAQ', 'PostSet', @setDAQ);
addlistener(lp, 'Laser', 'PostSet', @setLaser);
addlistener(sp, 'Sensor', 'PostSet', @setSensor);
addlistener(rp, 'RoughSpectrum', 'PostSet', @setRoughSpectrum);
addlistener(rp, 'Spectrum', 'PostSet', @setSpectrum);
addlistener(cp, 'Results', 'PostSet', @setResults);
addlistener(cp, 'Calibration', 'PostSet', @setCalibration);
addlistener(cp, 'plotResultsWindowFlag', 'PostSet', @plotResultsWindow);
addlistener(rw, 'plotResultsWindowFlag', 'PostSet', @plotResultsWindow);

% Busy state listeners
addlistener(lp, 'Busy', 'PostSet', @setBusyBar);
addlistener(dp, 'Busy', 'PostSet', @setBusyBar);
addlistener(sp, 'Busy', 'PostSet', @setBusyBar);
addlistener(rp, 'Busy', 'PostSet', @setBusyBar);
addlistener(cp, 'Busy', 'PostSet', @setBusyBar);
addlistener(rw, 'Busy', 'PostSet', @setBusyBar);

% Show the GUI
f.Visible = 'on';
end

function str=num2eng(num);
    NUM2ENG Convert numbers to engineering notation strings.
    % str = num2eng(num) converts the number N to a engineering notation string using International System of Units (SI) prefixes.
    %
    % Examples:
    % num2eng(23e8) produces the string
    % '2.3G'
    % num2eng(0.0000004) produces the string
    % '400n'
    if num == 0
        str = '0';
        return
    end
    suffix_str = 'yzafpnum kMGTPEZY';
k = 1;
    while abs(num) >= 10^((k-8)) && k<=16.
if k\textasciitilde=9, 
suff=suffix_str(k); 
else 
suff=''; 
end 
str=[num2str(num/10^{3*(k-9)}), ', ', suff];
LambdaUnitSwitch = 2;
else if LambdaAvg<1
LambdaUnitSwitch = 3;
else if LambdaAvg<1e3
LambdaUnitSwitch = 4;
else
LambdaUnitSwitch = 4;
end

plot (LambdaMultiplier(LambdaUnitSwitch).*lambda, VMultiplier(VUnitSwitch).*v);
grid on; axis tight;
proxy = VMultiplier(VUnitSwitch).*v; margin1 = 0.1; margin2 = 0.1;
ylim ([min(proxy) - margin1.*max(proxy) - min(proxy)] ... 
max(proxy) + margin2.*max(proxy) - min(proxy)));
xlabel(['Wavelength [' LambdaUnit{LambdaUnitSwitch} ']]', 'FontSize', 12);
ylabel(['Voltage [' VUnit{VUnitSwitch} ']]', 'FontSize', 12);

%Operating wavelength line
v1 = min(v); v2 = max(v);
scale1 = 2;
if (v1<0) v1 = scale1*v1; else v1 = -v1; end
if (v2<0) v2 = -v2; else v2 = scale1*v2; end
if isobject(rp.Laser)
line (LambdaMultiplier(LambdaUnitSwitch)*rp.Laser.Wavelength.*[1 1], 
VMultiplier(VUnitSwitch).*[v1 v2], 'Color', 'red');
end

../+GUI/plotLabeldCal.m

function [axisHandle, xMult, yMult] = plotLabeldCal(x, y, xUnitSuffix, yUnitSuffix, calObj)

% General magnitudes setup
yUnitPrefix = {'p', 'n', 'u', 'm', '', 'k', 'M', 'G', 'T'};
xUnitPrefix = {'p', 'n', 'u', 'm', ''};
ymultiplier = [1e-12 1e-9 1e-6 1e-3 1 1e3 1e6 1e9 1e12];
xmultiplier = [1e-12 1e-9 1e-6 1e-3 1];

% Determining which units to use
xAvg = max(abs(x));
xUnitIndex = find(xAvg<=1e3.*xmultiplier, 1);
xMult = xmultiplier(xUnitIndex);
xUnit = strcat(xUnitPrefix(xUnitIndex), xUnitSuffix);

yAvg = max(abs(y));
yUnitIndex = find(yAvg<=1e3.*ymultiplier, 1);
yMult = ymultiplier(yUnitIndex);
yUnit = strcat(yUnitPrefix(yUnitIndex), yUnitSuffix);

% Plotting
axisHandle = plot (x./xMult, y./yMult);
grid on
axis tight;
proxy = y./yMult; margin1 = 0.3; margin2 = 0.3;
ylim ([min(proxy) - margin1.*max(proxy) - min(proxy)] ... 
max(proxy) + margin2.*max(proxy) - min(proxy)));
xlabel(['Time [' xUnit{1} ']]', 'FontSize', 9);
ylabel(['Voltage [' yUnit{1} ']]', 'FontSize', 9);
set(gca, 'FontSize', 7);

% Draw Lines
line ([x(1) x(end)]./xMult, ...
(calObj.vOffset+calObj.vPP/2).*[1 1]./yMult,...

213
'Color', 'red', 'LineStyle', '—', 'LineWidth', 2);
line([x(1) x(end)]./xMult, ...
(calObj.vOffset-calObj.vPP/2).*[1 1]./yMult,...
'Color', 'red', 'LineStyle', '—', 'LineWidth', 2);
if calObj.ACDFlag
    text((x(end)-0.25*(x(end)-x(1)))./xMult,...
        (calObj.vOffset+1.9*calObj.vPP/2)./yMult,...
        'AC Coupling', 'FontSize', 10, 'Color', 'black', 'Margin', 1);
else
    line([x(1) x(end)]./xMult,...
        calObj.vOffset.*[1 1]./yMult,...
        'Color', 'black', 'LineStyle', '—', 'LineWidth', 2);
    text((x(end)-0.2*(x(end)-x(1)))./xMult,...
        (calObj.vOffset+1.75*calObj.vPP/2)./yMult,...
        'DC Coupling', 'FontSize', 10, 'Color', 'black', 'Margin', 1);
end
title('Calibration Preview', 'FontSize', 9);
end

../+GUI/plotLabeldLeftRightEdge.m

function axisHandle = plotLabeldLeftRightEdge(t, v)
lambda = t;
Vunit = {'nV', 'uV', 'mV', 'V', 'kV'}; Tunit = {'ns', 'us', 'ms', 's'};
Vmuliplier = [1e9 1e6 1e3 1]; Tmultiplier = [1e9 1e6 1e3 1];
Vavg = max(abs(v)); Tavg = max(abs(t));
if Vavg<1e-6
    VUnitSwitch = 1;
elseif Vavg<1e-3
    VUnitSwitch = 2;
elseif Vavg<1
    VUnitSwitch = 3;
elseif Vavg<1e3
    VUnitSwitch = 4;
elseif Vavg>=1e3
    VUnitSwitch = 5;
else
    VUnitSwitch = 4;
end
if Tavg<1e-6
    TUnitSwitch = 1;
elseif Tavg<1e-3
    TUnitSwitch = 2;
elseif Tavg<1
    TUnitSwitch = 3;
elseif Tavg<1e3
    TUnitSwitch = 4;
else
    TUnitSwitch = 4;
end
axisHandle = plot(Tmultiplier(TUnitSwitch).*t, Vmultiplier(VUnitSwitch).*v);
grid on
axis tight;
proxy = Vmultiplier(VUnitSwitch).*v; margin1=0.1; margin2=0.1;
ylim([min(proxy)-margin1.*(max(proxy)-min(proxy)) max(proxy)+margin2.*(max(proxy)-min(proxy))]);
xlabel('Time [' TUnit{TUnitSwitch} '']', 'FontSize', 9);
plotLabeldNM.m

function [] = plotLabeldNM(rp)
lambda = rp.Spectrum.Spectrum(1, :);
v = rp.Spectrum.Spectrum(2, :);
VUnit = {'nV', 'uV', 'mV', 'V', 'kV'};
LambdaUnit = {'nm', 'nm', 'mm', 'm'};
VMultiplier = [1e9 1e6 1e3 1e-3]; LambdaMultiplier = [1e9 1e9 1e3 1e-3];
Vavg = max(abs(v)); LambdaAvg = max(abs(lambda));
if Vavg<1e-6
    VUnitSwitch = 1;
elseif Vavg<1e-3
    VUnitSwitch = 2;
elseif Vavg<1
    VUnitSwitch = 3;
elseif Vavg<1e3
    VUnitSwitch = 4;
else
    VUnitSwitch = 4;
end
if LambdaAvg<1e-6
    LambdaUnitSwitch = 1;
elseif LambdaAvg<1e-3
    LambdaUnitSwitch = 2;
elseif LambdaAvg<1
    LambdaUnitSwitch = 3;
elseif LambdaAvg<1e3
    LambdaUnitSwitch = 4;
else
    LambdaUnitSwitch = 4;
end
plot(LambdaMultiplier(LambdaUnitSwitch).*lambda, VMultiplier(VUnitSwitch).*v);
grid on; axis tight;
proxy = VMultiplier(VUnitSwitch).*v; margin1=0.1; margin2=0.1;
ylim([min(proxy)-margin1*(max(proxy)-min(proxy)) max(proxy)+margin2*(max(proxy)-min(proxy))]);
xlabel(['Wavelength [' LambdaUnit{LambdaUnitSwitch} ']', 'FontSize', 9]);
ylabel(['Voltage [' VUnit{VUnitSwitch} ']', 'FontSize', 9]);

%Operating wavelength line
vl = min(v); v2 = max(v); dv = v2-vl;
scale1 = 2;
v1=vl-dv;
v2=v2+dv;
assert(~isempty(rp.Spectrum.OperatingLambda), 'OperatingLambda Property missing in Spectrum object');
if isobject(rp.Laser)
    clr = [1 0.2 0];
    line (LambdaMultiplier(LambdaUnitSwitch)*rp.Spectrum.OperatingLambda.*[1 1], ...
        VMultiplier(VUnitSwitch).*[vl v2], 'Color', clr);
else
    clr = [0.7 0.7 0.7];
    line (LambdaMultiplier(LambdaUnitSwitch)*rp.Spectrum.LambdaMaxSensitivity.*[1 1], ...
        VMultiplier(VUnitSwitch).*[vl v2], 'Color', clr);
end
function [axisHandle, xMultiplier, yMultiplier] = plotLabeldResultsE(t, v)

lambda = t;
Vunit = {'nV/m', 'uV/m', 'mV/m', 'V/m', 'kV/m', 'MV/m', 'GV/m'};
Tunit = {'ns', 'us', 'ms', 's'};
Vmultiplier = [1e9 1e6 1e3 1e-3 1e-6 1e-9]; Tmultiplier = [1e9 1e6 1e3 1];
Vavg = max(abs(v)); Tavg = max(abs(t));
if Vavg<1e-6
  VUnitSwitch = 1;
elseif Vavg<1e-3
  VUnitSwitch = 2;
else Vavg<1e3
  VUnitSwitch = 3;
else Vavg<1e6
  VUnitSwitch = 4;
else Vavg>=1e6
  VUnitSwitch = 5;
else Vavg>=1e9
  VUnitSwitch = 6;
else
  VUnitSwitch = 7;
end
if Tavg<1e-6
  TUnitSwitch = 1;
elseif Tavg<1e-3
  TUnitSwitch = 2;
else Tavg<1e3
  TUnitSwitch = 3;
else Tavg<1e6
  TUnitSwitch = 4;
else
  TUnitSwitch = 4;
end
axisHandle = plot (Tmultiplier(TUnitSwitch).*t, Vmultiplier(VUnitSwitch).*v);
grid on
axis tight;
proxy = Vmultiplier(VUnitSwitch).*v; margin1 = 0.1; margin2 = 0.1;
ylim([min(proxy) - margin1*(max(proxy) - min(proxy)) max(proxy) + margin2*(max(proxy) - min(proxy))]);
xlabel(['Time ', Tunit{TUnitSwitch} ''], 'FontSize', 12);
ylabel(['Voltage ', Vunit{VUnitSwitch} ''], 'FontSize', 12);
set(gca, 'FontSize', 7);
end

function [axisHandle, xMultiplier, yMultiplier] = plotLabeldResultsV(t, v)

lambda = t;
Vunit = {'nV', 'uV', 'mV', 'V', 'kV'};
Tunit = {'ns', 'us', 'ms', 's'};
Vmultiplier = [1e9 1e6 1e3 1e-3]; Tmultiplier = [1e9 1e6 1e3 1];
Vavg = max(abs(v)); Tavg = max(abs(t));
if Vavg<1e-6
VUnitSwitch = 1;
else if Vavg<1e-3
VUnitSwitch = 2;
else if Vavg<1
VUnitSwitch = 3;
else if Vavg<1e3
VUnitSwitch = 4;
else if Vavg>=1e3
VUnitSwitch = 5;
else
VUnitSwitch = 4;
end

if Tavg<1e-6
TUnitSwitch = 1;
else if Tavg<1e-3
TUnitSwitch = 2;
else if Tavg<1
TUnitSwitch = 3;
else if Tavg<1e3
TUnitSwitch = 4;
else
TUnitSwitch = 4;
end

axisHandle = plot (Tmultiplier(TUnitSwitch).*t, Vmultiplier(VUnitSwitch).*v);
grid on
axis tight;
proxy = Vmultiplier(VUnitSwitch).*v; margin1=0.1; margin2=0.1;
ylim ((min(proxy)-margin1.*max(proxy) - min(proxy)) max(proxy)+margin2.*max(proxy) - min(proxy));
xlabel(['Time ', 'Unit {TUnitSwitch} '], 'FontSize', 9);
ylabel(['Voltage ', 'Unit {VUnitSwitch} '], 'FontSize', 9);
set(gca, 'FontSize', 7);
xMultiplier = Tmultiplier(TUnitSwitch);
yMultiplier = Vmultiplier(VUnitSwitch);

../+GUI/plotLabeldRough.m

function [] = plotLabeldRough(rp)
if isempty(rp.RoughSpectrum)
cla; return
else
lambda = rp.RoughSpectrum(:, :);
v = rp.RoughSpectrum(2, :);
end
VUnit = {'nV', 'uV', 'mV', 'V', 'kV'};
LambdaUnit = {'nm', 'nm', 'mm', 'm'};
VMultiplier = [1e9 1e6 1e3 1 e-3]; LambdaMultiplier = [1e9 1e9 1e3 1];
Vavg = max(abs(v)); LambdaAvg = max(abs(lambda));
if Vavg<1e-6
VUnitSwitch = 1;
elseif Vavg<1e-3
VUnitSwitch = 2;
elseif Vavg<1
VUnitSwitch = 3;
elseif Vavg<1e3
VUnitSwitch = 4;
elseif Vavg>=1e3
VUnitSwitch = 5;
else
VUnitSwitch = 4;
if LambdaAvg<1e-6 
    LambdaUnitSwitch = 1;
elseif LambdaAvg<1e-3 
    LambdaUnitSwitch = 2;
elseif LambdaAvg<1 
    LambdaUnitSwitch = 3;
elseif LambdaAvg<1e3 
    LambdaUnitSwitch = 4;
else
    LambdaUnitSwitch = 4;
end

plot (LambdaMultiplier(LambdaUnitSwitch).*lambda, VMultiplier(VUnitSwitch).*v);
grid on; axis tight;
proxy = VMultiplier(VUnitSwitch).*v; margin1 = 0.1; margin2 = 0.1;
ylim ([min(proxy)−margin1.*(max(proxy)−min(proxy)) max(proxy)+margin2.*(max(proxy)−min(proxy)) ]);
xlabel (’Wavelength ’ [’ LambdaUnit{LambdaUnitSwitch} ’] , ’FontSize’ , 9);
ylabel (’Voltage ’ [’ VUnit{VUnitSwitch} ’] , ’FontSize’ , 9);

../+GUI/TimerGUI.m

function fh = TimerGUI(x)
% Demonstrate how to have a running clock in a GUI, and timer use.
% Creates a small little GUI which displays the correct time and is updated
% every second according to the system clock.
% Returns the object S which contains all the relevant elements

x0=x*60;
xs = mod(x0, 60);
xMin = floor(x0/60);

S.fh = figure (’units’, ’normalized’, ....
    ’position’, [0.5 0.5 0.1 0.05] ....
    ’menubar’, ’none’ ....
    ’name’, ’Low−noise timeout’ ....
    ’numbertitle’, ’off’ ....
    ’resize’, ’off’);
S.tx = uicontrol (’style’, ’text’ ....
    ’unit’, ’pix’ ....
    ’position’, [35 10 130 30] ....
    ’string’, sprintf(’%02i:%02i’, xMin, xs) ....
    ’background’, get(S.fh, ’color’) ....
    ’fontsize’, 18 ....
    ’fontweight’, ’bold’ ....
    ’foregroundcolor’, [0.9 .1 .1]);

tmr = timer (’Name’, ’Reminder’ ....
    ’Period’, 1 .... % Update the time every second.
    ’TasksToExecute’, inf .... % number of times to update
    ’ExecutionMode’, ’fixedSpacing’ ....
    ’TimerFcn’, {@updater});
start(tmr); % Start the timer object.
set(S.fh, ’deletefcn’, {@deleter}) % Kill timer if fig is closed.
fh=S.fh;

function [] = updater(varargin)
% timerfcn for the timer. If figure is deleted, so is timer.
% I use a try−catch here because timers are finicky in my
% experience.
try
    x0 = x0 − 1;
    if x0>=0

218
xSec = mod(x0, 60);
xMin = floor(x0/60);
set(S.tx, 'string', sprintf('%02d:%02d', xMin, xSec))
if ~x0
    X = load('gong'); % At the timeout, sound a gong.
sound(X.y,X.Fs*2.5);
[y,Fs] = audioread('StopLN.mp3');
sound(y,Fs);
end
clear X
else
    if mod(x0,2)
        set(S.fh, 'Color', 'red');
    else
        set(S.fh, 'Color', 'white');
    end
catch
    delete(S.fh) % Close it all down.
end
function [] = deleter(varargin)
    % If figure is deleted, so is timer.
    stop(tmr);
    delete(tmr);
end

A.3 Interfaces

A.3.1 Classes

../+interfaces/DAQState.m

classdef DAQState
% DAQState State of a daq
% Detailed explanation goes here

    enumeration
        Off, On
end

../+interfaces/IAlgorithm.m

classdef (Abstract) IAlgorithm < handle
% IAlgorithm Interface to main processing algorithm

    properties (SetAccess = private)
        Laser
        lambdaRes
        vres
        VMaxres
    end
methods (Abstract)
  resonances = GetRoughSpectrum (obj)
  % Gets a rough spectrum from the laser/daq
  % Returns interface.Resonance objects
  spectrum = GetFineSpectrum (obj, resonance)
  % Gets a fine spectrum from the laser/daq
  % Returns an instance of interface.ISpectrum
end

../+interfaces/IDaq.m
classdef (Abstract) IDaq < handle
  %DAQ Base DAQ Interface
  properties (Transient, SetObservable, SetAccess = private)
    Voltage = 0
    % Current voltage on DAQ
    State = interfaces.DAQState.Off
  end
end
methods
  function delete (obj)
    % Ensures that the daq is shut down during destruction
    try
      obj.RequestShutdown();
      catch ME
        warning ('Error shutting down daq: %s', ME.message)
    end
  end
  function [ok, V] = RequestVoltage (obj)
    % Requests a single sample from the DAQ
    obj.EnsureOn();
    V = obj.GetVoltage();
    obj.Voltage = V;
    ok = true;
  end
  function [ok, V] = RequestVoltageMean (obj, t)
    % Requests an averaged sample from the daq over time t
    obj.EnsureOn();
    V = obj.GetVoltageMean();
    obj.Voltage = V;
    ok = true;
  end
  function ok = RequestShutdown (obj)
    if obj.State == interfaces.DAQState.On
      obj.ShutDown
      obj.State = interfaces.DAQState.Off;
    end
    ok = true;
  end
end
methods (Abstract, Access = protected)
  Initialize (obj)
    % Initializes the daq
  Shutdown (obj)
    % Shuts down the daq
  V = GetVoltage (obj)
    % Gets a single sample from the DAQ
V = GetVoltageMean(obj, t)
% Gets an averaged sample from the daq over time t
end

methods (Access = private)
  function EnsureOn(obj)
    % Ensures that the daq has been initialized
    if obj.State == interfaces.DAQState.Off
      obj.Initialize()
      obj.State = interfaces.DAQState.On;
    end
  end
end
end

../+interfaces/IDaqFactory.m

classdef IDaqFactory < handle
  % IDAQFACTORY Creates DAQ objects
  % Detailed explanation goes here

  properties (Abstract, Constant)
    Name
    % Name of this daq type
  end

  methods (Abstract)
    daq = Create(obj)
    % Creates a new daq object
  end
end

../+interfaces/ILaser.m

classdef (Abstract) ILaser < handle
  % ILASER Interface to a laser
  % All lasers must implement these methods

  properties (Transient, SetAccess = protected)
    WavelengthMin = false
    % Minimum wavelength that this laser can tune to, in meters
    WavelengthMax = false
    % Maximum wavelength that this laser can tune to, in meters
    WavelengthStep = false
    % Minimum step size for the laser
    FTFlim = false;
    % Fine tuning + limits in Hz, value is on the order of GHz
    Grid
    % Grid spacing in GHz
    Powers = []
    % Collection of all powers this laser may be set to
  end

  properties (Transient, SetObservable, SetAccess = private)
    Wavelength = 0
    % Current operating wavelength of laser
    Power = 0
    % Current operating power of laser
State = interfaces.LaserState.Off
% Current state of the laser
end

methods (Abstract, Access=protected)
  Initialize(obj)
  % Initializes the laser
  ShutDown(obj)
  % Shuts down the laser
  SetWavelength(obj, wavelength)
  % Sets the wavelength of the laser
  % The implementing class should perform all actions and waiting
  % necessary to set the laser to a certain wavelength. If
  % wavelength is false, then
  GetWavelength(obj)
  % Reads the wavelength of the laser
  SetPower(obj, power)
  % Sets the power of the laser
  GetPower(obj)
  % Sets the power of the laser
  % The implementing class should perform all actions and waiting
  % necessary to set the laser to a certain power
  SetLowNoise(obj, sw)
  % Turn low noise mode ON/OFF, sw =0 for OFF, sw=1 for ON
end

methods
  function delete(obj)
    % Ensures that the laser is shut down during destruction
    try
      obj.RequestShutdown();
      catch ME
        warning( 'Error shutting down laser: %s', ME.message )
    end
  end

  function [ ok, wvRead ] = RequestWavelength(obj, wavelength)
    % Requests that the laser turn on and set itself to the
    % requested wavelength in meters
    obj.EnsureOn();
    if obj.State == interfaces.LaserState.LN
      obj.SetLowNoise( false );
      obj.State = interfaces.LaserState.On;
    end
    if wavelength >= obj.WavelengthMin && wavelength <= obj.WavelengthMax
      wvRead = obj.SetWavelength(wavelength);
    else
      warn( 'Wavelength value out of range!' );
    end
    obj.UpdateWavelength;
    obj.UpdatePower;
    ok = true;
  end

  function ok = UpdateWavelength(obj)
    % Asks the laser its wavelength
    obj.EnsureOn();
    obj.Wavelength = obj.GetWavelength;
    ok = true;
  end

  function ok = RequestPower(obj, power)
    % Requests that the laser turn on and set itself to the
    % requested power level
    obj.EnsureOn();
    if obj.State == interfaces.LaserState.LN
      obj.SetLowNoise( false );
    end
  end
obj.State = interfaces.LaserState.On;
end
if power<obj.Powers(1)
    power = obj.Powers(1);
elseif power>obj.Powers(end)
    power = obj.Powers(end);
end
obj.SetPower(power);
obj.UpdateWavelength;
obj.UpdatePower;
ok = true;
end

function ok = UpdatePower(obj)
% Asks the laser its power
obj.EnsureOn();
obj.Power = obj.GetPower;
ok = true;
end

function ok = RequestLNSwitch(obj, sw)
% Requests that low noise mode be switched on (sw=0) or off (sw=1)
if ~obj.SetLowNoise(sw);
    ok = false;
    return;
end
% Update Laser State
if (sw==true)&&(obj.State == interfaces.LaserState.On)
    obj.State = interfaces.LaserState.LN;
elseif (sw==false)&&(obj.State == interfaces.LaserState.LN)
    obj.State = interfaces.LaserState.On;
else
    disp('Unknown Laser State');
end
ok = true;
end

function ok = RequestShutdown(obj)
% Requests that the laser shut down,
% but first turn off low noise
if obj.State == interfaces.LaserState.LN
    obj.SetLowNoise(false);
    obj.State = interfaces.LaserState.On;
end
if obj.State == interfaces.LaserState.On
    obj.Shutdown();
    obj.State = interfaces.LaserState.Off;
end
ok = true;
end

methods (Access = private)

function EnsureOn(obj)
% Ensures that the laser has had its initialization method % called
if obj.State == interfaces.LaserState.Off
    obj.Initialize();
    obj.State = interfaces.LaserState.On;
end
end
classdef (Abstract) ILaserFactory < handle

%ILASERFACTORY Creates laser objects
% Detailed explanation goes here

properties (Abstract, Constant)
    Name
        % Name of this laser type
end

methods (Abstract)
    laser = Create(obj)
        % Creates a new laser object
end
end

../+interfaces/ISpectrum.m

classdef (Abstract) ISpectrum < handle

%FINESPECTRUM A calibrated laser spectrum which interacts with data

properties (SetObservable)
    LambdaMaxSensitivity
        % wavelength at the point of maximum spectrum slope
    VMaxSensitivity
        % Voltage at the wavelength with maximum spectrum slope at zero electric field
    OperatingLambda
        % Lambda at which this spectrum operates, can be changed externally
    OperatingV
        % Voltage at the operating wavelength point with no electric field
    Calibration
        % calibration object, contains E and V calibration factors
end

properties (SetAccess=protected, SetObservable)
    Spectrum
        % Contains data in the form of an array of (lambda, power)
    Results
        % Results object, contains all the data
end

properties (Dependent)
    LambdaMax
    LambdaMin
end

methods (Abstract)
    Calibrate(obj, appliedVoltageFile, appliedVoltage)
        % Calibrates this spectrum object with the passed voltage
    voltages = ProcessData(obj, powerLevels)
        % Performs a direct conversion using the calibration factor of the
    % passed data
end

methods
    function lambda = get.LambdaMax(obj)
        lambda = max(obj.Spectrum(1,:));
    end
    function lambda = get.LambdaMin(obj)
\[
\lambda = \min(\text{obj}.\text{Spectrum}(1,:));
\]

../+interfaces/LaserState.m

classdef LaserState
  %LASERSTATE Summary of this class goes here
  % LN - low-noise state
  
  enumeration
    Off, On, LN
  end
end

../+interfaces/Resonance.m

classdef Resonance
  %RESONANCE Section of a rough spectrum
  
  properties (SetAccess = private)
    Spectrum;
    % Power levels for this spectrum in the form of an array of
    % (\lambda, power)
  end

  properties
    WavelengthMin;
    WavelengthMax;
  end

  methods
    function obj = Resonance(spectrum)
      obj.Spectrum = spectrum;
      obj.WavelengthMin = spectrum(1,1);
      obj.WavelengthMax = spectrum(1,end);
    end
  
    function lambda = get.WavelengthMin(obj)
      lambda = obj.WavelengthMin;
    end
  
    function obj = set.WavelengthMin(obj, lambda)
      obj.WavelengthMin = lambda;
    end
  
    function lambda = get.WavelengthMax(obj)
      lambda = obj.WavelengthMax;
    end
  
    function obj = set.WavelengthMax(obj, lambda)
      obj.WavelengthMax = lambda;
    end
  end
end
classdef Sensor
% SENSOR object
properties (SetAccess = private)
    Name
        % Name of the sensor used in measurements
    EShift
        % E-field shift rate for the specific sensor in pm/(MV/m)
end
methods
    function obj = Sensor(name, eShift)
        obj.Name = name;
        obj.EShift = eShift;
    end
end

A.4 DAQ support files

A.4.1 Classes

classdef NIDaqUSB < interfaces.IDaq
% Daq Summary of this class goes here
% Detailed explanation goes here
properties (Access = private)
    s = false
    ch0 = false
    devs = false
end
properties (Constant)
    analogInputChannelName = 'ai0';
    sampleRate = 1000;
    averagingDuration = 0.5;
end
methods (Access = protected)
    function Initialize(obj) % public methods cause other people will call these
        disp(sprintf('Initializing NI DAQ USB 6002'));
        % Finding the DAQ Identifier on the system
        disp(sprintf('Initializing DAQ...'));
        obj.devs = daq.getDevices; % To use this method one must have
        % Data Acquisition Toolbox installed
        for kk=1:length(obj.devs)
            if strcmp(obj.devs(kk).Model, 'USB-6002');
                USBNo = kk;
            end
        end
        assert(exist('USBNo', 'var') == 1, 'No NI USB-6002 device found');
        % Opening the Analog input channel
        obj.s = daq.createSession('ni');
        obj.s;
        obj.s.Rate = obj.sampleRate;
        obj.s.DurationInSeconds = obj.averagingDuration;
end
obj.ch0 = obj.s.addAnalogInputChannel(obj.devList(USBNo).ID, obj.
analogInputChannelName, 'Voltage');
obj.ch0.TerminalConfig = 'SingleEnded';
disp(sprintf('Voltage on DAQ = %2.3f mV', 1e3.*GetVoltage(obj)));
end
function ShutDown(obj)
    release(obj.s);
end
function V = GetVoltage(obj)
    V = inputSingleScan(obj.s); %Single scan of voltage on the DAQ
end
function V = GetVoltageMean(obj, t)
    switch nargin
    case 2
        obj.s.DurationInSeconds = t;
    otherwise
        obj.s.DurationInSeconds = obj.averagingDuration;
    end
data = startForeground(obj.s);
V = mean(data); %Mean value of the voltage on the DAQ in period t
end
end
end

../+NIdaqUSB6002/NIDaqUSBFactory.m

classdef NIDaqUSBFactory < interfaces.IDaqFactory
% NIDaqUSBFactory Creates laser objects for NI-DAQ USB6002 DAQ device
% Detailed explanation goes here

    properties (Constant)
    Name = 'NI-DAQ-USB-6002';
    end
    end
methods
    function daq = Create(~)
        daq = NIdaqUSB6002.NIDaqUSB();
    end
end
end

A.5 CoBriteDX1 Laser support files

A.5.1 Classes

../+CoBriteDX1/Laser.m

classdef Laser < interfaces.ILaser
%LASER Summary of this class goes here
% Detailed explanation goes here

    properties (Access=private)
    com_port

227
```

```
```
disp('Turning the Laser OFF');
CoBriteDX1.CBMX_set_port_state(obj.ser,obj.Port,0);
CoBriteDX1.wait_tuning(obj.ser.obj.Port.obj.max_time);
fclose(obj.ser_obj); %close the laser port
clear port;
toc;
end

function wvRead = SetWavelength(obj, wavelength)
fprintf('
');
fprintf('Setting Wavelength ...
');
wavelengthNM = wavelength*1e9;
f = 1e-9*algorithms.LambdaToF(wavelength); %f [GHz]
fprintf('Target: lambda = %4.3f nm and f = %2.3f GHz
', wavelength_NM, 1e9*wavelength, f);
F = CoBriteDX1.CBMX_query_port_FTFObj.ser.obj.Port); %while length(F)˜=1; %while loops help fix the communication error with the laser, should be put into the query script
F = CoBriteDX1.CBMX_query_port_FTFObj.ser.obj.obj.Port); pause(0.1); fprintf (' Repeating query (FTF)
');
end
f0 = 1e3*CoBriteDX1.CBMX_query_port_freqoobj.ser.obj.obj.Port); %while length(f0)˜=1;
f0 = 1e3*CoBriteDX1.CBMX_query_port_freqoobj.ser.obj.obj.Port); pause(0.1); fprintf (' Repeating query (FTF)
');
deltaf = f-f0;
fprintf('PreTuning: FTF = %6.3f GHz f = %6.3f GHz
', F, f0);
fprintf('delta f = %6.3f GHz
', deltaf);
if (abs(deltaf)<obj.FTFlim)
fprintf('Fine tuning ...
');
portConf = CoBriteDX1.CBMX_query_port_config(obj.ser.obj.obj.Port); %making sure that portConf is long enough in the while loop
portConf = CoBriteDX1.CBMX_query_port_FTFObj.ser.obj.obj.Port, deltatheta);e=
fprintf('Rough Tuning ...
');
f0new = f - 12; % f0new is in GHz
f0newThz = f0new*1e-3;
portConf = CoBriteDX1.CBMX_query_port_config(obj.ser.obj.obj.Port);
fprintf('ReQuerying Laser ...
');
pause(0.1);
end
CoBriteDX1.CBMX_set_port_FTFObj.ser.obj.obj.Port = . . .
f0newThz, 12, portConf(3), portConf(4), portConf(6));
% CoBriteDX1.CBMX_set_port_freqoobj.ser.obj.obj.Port, f0newThz);
% CoBriteDX1.wait_tuning(obj.ser.obj.obj.Port.obj.max_time);
% CoBriteDX1.CBMX_set_port_FTFObj.ser.obj.obj.Port, . . .
F = CoBriteDX1.CBMX_query_port_FTFObj.ser.obj.obj.Port);
while length(F)˜=1;
F = CoBriteDX1.CBMX_query_port_FTFObj.ser.obj.obj.Port); pause(0.1); fprintf (' Repeating query (FTF)
');
end
f0_post = 1e3*CoBriteDX1.CBMX_query_port_freqoobj.ser.obj.obj.Port); %while length(f0_post)˜=1;
f0_post = 1e3*CoBriteDX1.CBMX_query_port_freqoobj.ser.obj.obj.Port); pause(0.1); fprintf (' Repeating query (freq)
');
end
wvTarget = 1e9*algorithms.FToLambda(1e9*f0+1e9*deltatheta);
wvRead = 1e9*algorithms.FToLambda(1e9*f0_post+1e9*F);
% wvRead=CoBriteDX1.CBMX_query_port_wav(obj.ser.obj.obj.Port);
fprintf('PostTuning: FTF = %6.3f GHz f = %6.3f GHz
', F, f0_post);
fprintf('
Target lambda = %4.3f nm and actual lambda = %4.3f nm
', ...
wvTarget, wvRead);
fprintf('Wave length error: %3.3f pm', 1e3*abs(wvTarget-wvRead));
while abs(wvTarget-wvRead)>obj.wvError
fprintf('Port %d-%d-%d: Error! Wavelength error exceeded', obj.Port);

obj.SetWavelength(wavelength);

f0_post = 1e3*CoBriteDX1.CBMX_query_port_frequ(obj.ser_obj.obj.Port);
while length(f0_post) != 1;
    f0_post = 1e3*CoBriteDX1.CBMX_query_port_frequ(obj.ser_obj.obj.Port); pause(0.1);
    fprintf('Repeating query (frequ)\n');
end
wvRead = 1e9*algorithms.FToLambda(1e9*f0_post+1e9*F);
end

function wavelength = GetWavelength(obj)
    f0 = 1e3*CoBriteDX1.CBMX_query_port_frequ(obj.ser_obj.obj.Port); % in THz
    while length(f0) != 1;
        f0 = 1e3*CoBriteDX1.CBMX_query_port_frequ(obj.ser_obj.obj.Port); pause(0.1);
        fprintf('Repeating query (FTF)\n');
    end
d0 = CoBriteDX1.CBMX_query_port_FTF(obj.ser_obj.obj.Port); % in GHz
    while length(d0) != 1;
        d0 = CoBriteDX1.CBMX_query_port_FTF(obj.ser_obj.obj.Port); pause(0.1); fprintf('Repeating query (FTF)\n');
    end
    f = f0+d0;
    wavelength = algorithms.FToLambda(1e9*f);
    fprintf('Current wavelength is %4.3f nm\n', 1e9*wavelength);
end

function SetPower(obj, power)
    fprintf('Setting Power ...\n');
    power_mW = 1e3*power;
    powerDBm = 10*log10(power_mW);
    lim=CoBriteDX1.CBMX_query_port_power_limit(obj.ser_obj.obj.Port);
    assert(powerDBm>lim(1)||powerDBm<lim(2), 'Power out of bounds');
    CoBriteDX1.CBMX_set_port_power(obj.ser_obj.obj.Port, powerDBm);
    CoBriteDX1.wait_tuning(obj.ser_obj.obj.Port, obj.max_time);
    pause(obj.ptime);
    pwrRead=CoBriteDX1.CBMX_query_port_power(obj.ser_obj.obj.Port);
    fprintf('Power setting error = %2.6f \n', abs(10^((powerDBm/10)-10*(pwrRead/10))));
    assert(abs(10^((powerDBm/10)-10*(pwrRead/10))<1e3*obj.pwrError || pwrRead==-100, ... 'Port %d-%d-%d: Error! Power setting/reading not working', obj.Port);
    fprintf('Power set to %2.3f mW \n', 1e3*(pwrRead));
end

function power = GetPower(obj)
    powerDBm = CoBriteDX1.CBMX_query_port_power(obj.ser_obj.obj.Port);
    power = 1e-3+10*(powerDBm/10);
    fprintf('Current power is %4.3f mW\n', 1e3*power);
end

function ok = SetLowNoise(obj, sw)
% turn dither off
if sw==true
    disp('Turning Low Noise Mode on...');
    x = inputdlg('Timer setup: set Low-noise mode timeout [mins]:', 'Timeout', 1, {'5 '});
    if isempty(x);
        obj.timer_handle = GUI.TimerGUI(str2double(x{:}));
    end
else
    disp('Turning Low Noise Mode off...');
end
curr=CoBriteDX1.CBMX_query_port_dither(obj.ser_obj.obj.Port);
if curr==1
    disp('Dither is not supported for this laser\n');
else
    if sw; ditTarget=0; else ditTarget=1; end; % 1 for OFF, 0 for ON
    CoBriteDX1.CBMX_set_port_dither(obj.ser_obj.obj.Port, ditTarget);
% CoBriteDX1.wait_tuning(obj.ser_obj, obj.Port, obj.max_time);
% pause(obj.ptime);
ditRead=CoBriteDX1.CBMX_query_port_dither(obj.ser_obj, obj.Port);
assert(ditTarget==ditRead,...
'Port %d-%d-%d: Warning! Dither status setting/reading not working', obj.Port);
ok = true;
if sw==true
   disp('ON');
else
   disp('OFF');
   if isvalid(obj.timer_handle)
      delete(obj.timer_handle);
   end
end
end
end
end
end
end
end

A.5.2 Functions

function mons=CBMX_query_monitor_values(ser_obj,Port)

% D. Stahl 11.11.11
% queries monitor values of ports
% Port address consisting of a 3 element vector, 0 is wildcard
% if wildcard is used, first 3 columns of response are Port address
% response columns: LaserDiode chip temp/LaserDiode base temp/LaserDiode
mon=CoBriteDX1.CBMX_query.prototype(ser_obj, Port, 'mon', '%f,%f,%f,%f');

../CoBriteDX1/CBMX_query_port_config.m

function config=CBMX_query_port_config(ser_obj, Port)

% D. Stahl 28.10.11
% queries Laserport setting
% Port address consisting of a 3 element vector, wildcard is NOT allowed!
% if wildcard is used, first 3 columns are Port address
% columns are: Frequency, FTF, Power, On/Off, Busy state, Dither state
if any(Port==0)
    error('No Port wildcards allowed!');
else
    config=CoBriteDX1.CBMX_query.prototype(ser_obj, Port, 'conf', '%f,%f,%f,%f,%d,%d,%d');
end

../CoBriteDX1/CBMX_query_port_dither.m

function dither=CBMX_query_port_dither(ser_obj, Port)

% D. Stahl 22.11.11
% queries dither state of ports
% Port address consisting of a 3 element vector, 0 is wildcard
% if wildcard is used, first 3 columns of response are Port address
% response 1: dither on, 0: dither off, -1: not supported
% requires FW version 2.0.0 or higher
% wildcard is used, first 3 columns of response are Port address, 4th column is
% actual value
% response is a two element vector

dither=CoBriteDX1.CBMX_query.prototype(ser_obj, Port, 'dit', '%d');

../CoBriteDX1/CBMX_query_port_freq_limit.m

function frequency_limit=CBMX_query_port_freq_limit(ser_obj, Port)

% D. Stahl 21.11.11
% queries Laserport frequency limits
% Port address consisting of a 3 element vector C–S–P, 0 is wildcard
% if wildcard is used, first 3 columns of response are used Port address, 4th column is
% actual value
% response is a two element vector

frequency_limit=CoBriteDX1.CBMX_query.prototype(ser_obj, Port, 'freq:lim', '%f,%f');

../CoBriteDX1/CBMX_query_port_freq.m

function freq=CBMX_query_port_freq(ser_obj, Port)

% D. Stahl 27.10.11
% queries Laserport freq
% Port address consisting of a 3 element vector C–S–P, 0 is wildcard
% if wildcard is used, first 3 columns of response are used Port address, 4th column is
% actual value
% response is a two element vector

freq=CoBriteDX1.CBMX_query.prototype(ser_obj, Port, 'freq', '%f');
function FTF_limit=CBMX_query_port_FTF_limit(ser_obj, Port)
% D. Stahl 11.11.11
% queries Laserport FTF offset limits
% Port address consisting of a 3 element vector C-S-P, 0 is wildcard
% if wildcard is used, first 3 columns of response are used Port address, 4th column is
% actual value
% response is a two element vector

temp=CoBriteDX1.CBMX_query prototype(ser_obj, Port, 'off:lim', '%f'); %as value is symmetrical, only
1 value is returned
FTF_limit=[-temp temp];

function FTF=CBMX_query_port_FTF(ser_obj, Port)
% D. Stahl 11.11.11
% queries FTF value of ports
% Port address consisting of a 3 element vector, 0 is wildcard
% if wildcard is used, first 3 columns of response are Port address

FTF=CoBriteDX1.CBMX_query prototype(ser_obj, Port, 'off', '%f');

function power_limit=CBMX_query_port_power_limit(ser_obj, Port)
% D. Stahl 21.11.11
% queries Laserport power limits
% Port address consisting of a 3 element vector C-S-P, 0 is wildcard
% if wildcard is used, first 3 columns of response are used Port address, 4th column is
% actual value

power_limit=CoBriteDX1.CBMX_query prototype(ser_obj, Port, 'pow:lim', '%f,%f');

function Power=CBMX_query_port_power(ser_obj, Port)
% D. Stahl 27.10.11
% queries Laserport power
% Port address consisting of a 3 element vector, 0 is wildcard
% if wildcard is used, first 3 columns are used Port address, 4th column is
% actual value

Power=CoBriteDX1.CBMX_query prototype(ser_obj, Port, 'pow', '%f');

function state=CBMX_query_port_state(ser_obj, Port)
% D. Stahl 11.11.11
% queries on/off state of ports
% Port address consisting of a 3 element vector, 0 is wildcard

233
5  
6  
7  state=CoBriteDX1.CBMX_query_prototype(ser_obj,Port,'stat','%d');

../+CoBriteDX1/CMX_query_port.wav_limit.m

1  function wav_limit=CMX_query_port.wav_limit(ser_obj,Port)
2  % D. Stahl 21.11.11
3  % queries Laserport wavelength limits
4  % Port address consisting of a 3 element vector C-S-P. 0 is wildcard
5  % if wildcard is used, first 3 columns of response are used Port address, 4th column is
6  % actual value
7  wav_limit=CoBriteDX1.CBMX_query_prototype(ser_obj,Port,'wav:lim','%f,%f');

../+CoBriteDX1/CMX_query_port.wav.m

1  function wav=CMX_query_port.wav(ser_obj,Port)
2  % D. Stahl 21.11.11
3  % queries Laserport wavelength
4  % Port address consisting of a 3 element vector C-S-P. 0 is wildcard
5  % if wildcard is used, first 3 columns of response are used Port address, 4th column is
6  % actual value
7  wav=CoBriteDX1.CBMX_query_prototype(ser_obj,Port,'wav','%f');

../+CoBriteDX1/CMX_query_prototype.m

1  function [response echo_out]=CMX_query_prototype(ser_obj,Port,command,res_format)
2  % D. Stahl 11.11.11
3  % Prototype query that can be used to execute arbitrary queries
4  % Port address consisting of a 3 element vector C-S-P. 0 is wildcard
5  % if wildcard is used, response first 3 columns are used Port address, 4th column is
6  % actual value
7  % command [String] command to be issued, i.e. 'freq'
8  % res_format [String] defines format of response, i.e.
9  % %f,%f,%f,%f,%d,%d,%d
10  % verbose=0; %outputs result to command line
11  timeout=30; %timeout for response query in seconds
12  response=[]; %default response
13  if ser_obj.BytesAvailable
14    fread(ser_obj,ser_obj.BytesAvailable); %clear receive buffer
15  end
16  sendline=sprintf([command,'? %d,%d,%d:'],Port);
17  fprintf(ser_obj,sendline);
18  tic;
19  while ser_obj.BytesAvailable==0 && toc<timeout
20    pause(0.1);
21  end
22  pause(0.05);
23  if any(Port==0) % change format if wildcard is used
24    res_format=[ '%d,%d,%d,' res_format ];
25  end
26  if ser_obj.BytesAvailable
27    data=char(fread(ser_obj,ser_obj.BytesAvailable))';
28    raw_res=regexp(data,'\n','split'); %read buffer and break it into lines
idx=1;
for i=1:length(raw_res)
    if ~isempty(strfind(raw_res{i},'ERR'))
        error(raw_res{i});
        response=[];
    else
        if ~isempty(strfind(raw_res{i},sendline(1:end-1))) % echo out on?
            echo_out=raw_res{i};
        else
            vals = sscanf(raw_res{i},res_format);
            if ~isempty(vals)
                response(idx,:)=vals;
            end
            idx=idx+1;
        end
    end
    if verbose, fprintf(strcat(command,' : ',res_format),response); end
end
if ~exist('echo_out'), echo_out=[]; end
end
else
    error('no data received');
end

function state=CBMX_query_tuning_state(ser_obj,Port)
    % D. Stahl 11.11.11
    % queries tuning state of ports
    % Port address consisting of a 3 element vector, 0 is wildcard
    % if wildcard is used, first 3 columns of response are Port address
    % response 1: tuning occurring, 0: settled
    %
    state=CoBriteDX1.CBMX_query_prototype(ser_obj,Port,'busy','%d');

function [ok_response]=CBMX_set_port_config(ser_obj,Port,Frequency,FTF,Power,OnOff,dither)
    % D. Stahl 27.10.11
    % sets Laserport state
    % Port address consisting of a 3 element vector, 0 is wildcard
    % requires FW version 2.0.0. or higher
    [ok_response]=CoBriteDX1.CBMX_set_prototype(ser_obj,Port,'conf',[Frequency FTF Power OnOff dither ],'%3.4f,%2.3f,%2.2f,%d,%d');

function [ok_response]=CBMX_set_port_dither(ser_obj,Port,dither)
    % D. Stahl 21.11.11
    % sets Laserport dither
    % response 1: dither on, 0: dither off, -1: not supported
    % Port address consisting of a 3 element vector, 0 is wildcard
    [ok_response]=CoBriteDX1.CBMX_set_prototype(ser_obj,Port,'dit',dither,'%d');
function [ok response]=CBMX_set_port_freq(ser_obj, Port, frequency)
% D. Stahl 27.10.11
% sets Laserport frequency
% Port address consisting of a 3 element vector, 0 is wildcard
[ok response]=CoBriteDX1.CBMX_set_prototype(ser_obj, Port, 'freq', frequency, '%3.4f');

function [ok response]=CBMX_set_port_FTF(ser_obj, Port, FTF)
% D. Stahl 27.10.11
% sets Laserport FTF offset
% Port address consisting of a 3 element vector, 0 is wildcard
[ok response]=CoBriteDX1.CBMX_set_prototype(ser_obj, Port, 'off', FTF, '%2.3f');

function [ok response]=CBMX_set_port_power(ser_obj, Port, Power)
% D. Stahl 12.11.11
% sets Laserport power in dBm
% Port address consisting of a 3 element vector, 0 is wildcard
[ok response]=CoBriteDX1.CBMX_set_prototype(ser_obj, Port, 'pow', Power, '%2.2f');

function [ok response]=CBMX_set_port_state(ser_obj, Port, state)
% D. Stahl 21.11.11
% sets Laserport output on/off state; 1: on 0: off
% Port address consisting of a 3 element vector, 0 is wildcard
[ok response]=CoBriteDX1.CBMX_set_prototype(ser_obj, Port, 'stat', state, '%d');

function [ok response]=CBMX_set_port_wavelength(ser_obj, Port, wavelength)
% D. Stahl 21.11.11
% sets Laserport wavelength
% Port address consisting of a 3 element vector, 0 is wildcard
[ok response]=CoBriteDX1.CBMX_set_prototype(ser_obj, Port, 'wav', wavelength, '%4.3f');

function [ok response]=CBMX_set_prototype(ser_obj, Port, command, parameter, param_format)
% D. Stahl 21.11.11
% Prototype set that can be used to execute arbitrary set commands
% Port address consisting of a 3 element vector C-S-P, 0 is wildcard
% command [String] command to be issued, i.e. 'freq'
% error [String] in case error occurred
function lCOM = getAvailableComPort() % function lCOM = getAvailableComPort() % Return a Cell Array of COM port names available on your computer
try
    s = serial('IMPOSSIBLE_NAME_ON_PORT'); fopen(s);
    lErrMsg = lasterr;
end

% Start of the COM available port
lIndex1 = findstr(lErrMsg, 'COM');
% End of COM available port
lIndex2 = findstr(lErrMsg, 'Use') - 3;

lComStr = lErrMsg(lIndex1:lIndex2);
% Parse the resulting string
lIndexDot = findstr(lComStr, '.');

% If no Port are available
if isempty(lIndex1)
    error(lComStr);
else
    ok = 1;
end
function wait_tuning (ser_obj, Port, max_time)
% D. Stahl 12.11.11
% waits until laser is settled or max_time [s] is elapsed
% Port address consisting of a 3 element vector, 0 is wildcard
timer_start = now;
fprintf('checking tuning state ... ');
while ((now-timer_start)+24*3600)<max_time
    if all('CoBriteDX1.CBMX_query_tuning_state(ser_obj,Port))
        break;
    end
pause(0.25);
fprintf('... ');
end
assert(((now-timer_start)+24*3600)<max_time, 'Max waiting time exceeded!');

A.6  OclaroTL5000 Laser support files

A.6.1 Classes

../+OclaroTL5000/Laser.m
properties (Access=private)
    com_port
end

properties (Transient, SetAccess = protected)
    timer_handle = false
end

properties (Constant)
    max_time=600; %max time to wait after tuning [s]
    ptime=5; % pause time used after setting a value for power
    %Notes on USB com ports: if there are communication errors, change
    %the com port number to a random number to avoid system conflicts,
    %connect the laser directly to the mother board, don’t connect
    %through a USB hub or front panel. Windows com port settings:
    %Baud rate: 112500,
    %data bits read and write to minimum: 64 bits,
    %latency minimum: 1 ms
    Port=[1 1 1]; % do not use wildcards here
    wvError=0.5;
    fEerror=0.001;
    pwrError=0.5e-3;
end

methods
    function obj = Laser(port)
        % laser outputs [1527.60488 1567.13256] when queried
        obj.WavelengthMin = 1528e-9;
        obj.WavelengthMax = 1565e-9;
        obj.WavelengthStep = 1e-12; %1 pm
        %in dBm, the laser gives [6 16] dBm when queried
        %FTF limits setting actual: -12.000GHz to 12.000GHz
        obj.FTFlim = 30; % in GHz
        obj.Grid = 30; % in GHz
        obj.Powers = 1e-3.*(ceil(10^((6/10))):1e-1:round(10*10^((6/10))*0.1)); % in W
        obj.com_port = port;
    end
end

methods (Access=protected)
    % these can be called only by this instance and its children(classes
    % that inherit from Laser object which inherits from ILaser)
    function Initialize(obj) % public methods cause other people will call these
        fprintf(‘\nInitializining Oclaro TL5000 ...’);
        bdRate = 9600;
        obj.ser_obj = serial(obj.com_port,’BaudRate’, bdRate);
        fopen(obj.ser_obj);

        % Setting Laser parameters
        % FTFlim
        fprintf(‘\nOclaro doesnt support fine tuning: FTF Limit = %.3f GHz’, obj.FTFlim) ;
        % Grid spacing
        obj.Grid = 1e-1*OclaroTL5000.Grid(obj); % in GHz
        fprintf(‘\nGrid spacing = %.3f GHz’, obj.Grid);
        % WavelengthMin
        f1 = OclaroTL5000.LFH1(obj);
        f2 = OclaroTL5000.LFH2(obj);
        f = f1*1e3 + f2/10;
        obj.WavelengthMin = algorithms.FToLambda(1e9*(f-2*obj.FTFlim));
        fprintf(‘\nMin Wavelength = %.4f nm’, 1e9*obj.WavelengthMin);
        % WavelengthMax
        f1 = OclaroTL5000.LFH1(obj);
        f2 = OclaroTL5000.LFH2(obj);
        f = f1*1e3 + f2/10;
obj.WavelengthMax = algorithms.FToLambda(1e9*(f+2e−3*obj.FTlim))−7e−9;
% subtracting from maximum wavelength because Oclaro laser

% gives an error for the reported max wv
fprintf(‘nMax Wavelength = %4.3f nm’, 1e9*obj.WavelengthMax);

% Powers
pLim(1)=OclaroTL5000.OPSL(obj)/100;% in dBm
pLim(2)=OclaroTL5000.OPSH(obj)/100;% in dBm
obj.Powers = ... 
1e−3.*(ceil(10ˆ( double(pLim(1))/10)):1e−1:round(10ˆ( double(pLim(2))/10)))
*0.1); %in W

% query laser until module is ready to be enabled
fprintf(‘nWaiting for module to become enable–ready ‘);
MRODY = false; CE = false;
while ‘MRODY ‘&‘CE ‘=
    OclaroTL5000.NOP(obj);
pause(0.1);
    fprintf(‘.’);
end
fprintf(‘nTurning the Laser ON’);

% setting the laser to channel one
OclaroTL5000.Channel(obj, 1);

% turning the laser ON
OclaroTL5000.ResEna(obj, 1, 1)

fprintf(‘nTuning wavelength… ‘);

% calculating frequencies
wavelengthNM = wavelength * 1e9;
f = 1e−9*algorithms.LambdaToF(wavelength); %f [GHz]
fprintf(‘nTarget: lambda = %4.3f nm and f = %2.3f GHz ‘, ...
1e9*wavelength , f);

f01 = OclaroTL5000.LF1(obj);
f02 = OclaroTL5000.LF2(obj);
f0 = f01*1e3 + f02/10;
deltaf = f−f0;
deltafGrid = round(deltaf/obj.Grid)*obj.Grid;
fprintf(‘nInitial: lambda = %4.3f nm and f = %2.3f GHz‘ , ...
algorithms.FToLambda(f0), f0);
fprintf(‘nDelta f = %6.3f GHz, matched to the grid = %6.3f GHz, error df = %6.3fGHz’ , ...
deltaf, deltafGrid, deltaf−deltafGrid);
fGrid = f0 + deltafGrid;

% setting the frequency
fprintf(‘nTuning wavelength… ‘);
fGrid2 = 1e4+mod(fGrid+1e−3.1);
fGrid1 = 1e−4*(fGrid+10−fGrid2);

OclaroTL5000.ResEna(obj, 1, 0); %laser OFF
OclaroTL5000.FCF1(obj, fGrid1);
% Verifying the frequency
f01 = OclaroTL5000.LF1(obj);
f02 = OclaroTL5000.LF2(obj);

f0_post = f01*1e3 + f02/10; % in GHz
wvRead = algorithms.FToLambda(f0_post);

fprintf('
New: lambda = %4.3f nm and f = %2.3f GHz....
', wvRead, f0_post);

while abs(wavelengthNM - wvRead) > obj.wvError
    fprintf('Port %d-%d-%d: Error! Wavelength error exceeded', obj.Port);
    obj.SetWavelength(wavelength);
end

function wavelength = GetWavelength(obj)
    f01 = OclaroTL5000.LF1(obj);
f02 = OclaroTL5000.LF2(obj);
f0 = f01*1e3 + f02/10;
df0 = OclaroTL5000.FTF(obj);

    wavelength = algorithms.FToLambda(1e9* f0);
    fprintf('
Current f: %2.3f GHz (FTF = %2.3f GHz), f0, df0);
    fprintf('
Current wavelength is %4.3f nm, 1e9*wavelength);
end

function SetPower(obj, power)
    fprintf('
Setting Power ...' );
power_mW = 1e3*power;
powerDBm = 10*log10(power_mW);

    lim = [obj.Powers(1) obj.Powers(end)];
    assert(powerDBm>lim(1)|powerDBm<lim(2), 'Requested Power out of bounds');
OclaroTL5000.ResEna(obj, 1, 0); % laser software OFF
OclaroTL5000.PWR(obj, 100*powerDBm);
OclaroTL5000.ResEna(obj, 1, 1); % laser software ON
pwrRead=1e-3*10^(powerDBm/10);

    fprintf('
Power setting error = %2.6f mW, abs(10^(powerDBm/10) - 10^(pwrRead/10))),
    assert(abs(10^(powerDBm/10) - 10^(pwrRead/10))<1e3*obj.pwrError || pwrRead==-100,...
    'Port %d-%d-%d: Error! Power setting/reading not working', obj.Port);
    fprintf('
Power set to %2.3f mW , 10**(pwrRead/10));
end

function power = GetPower(obj)
    powerResponse = OclaroTL5000.PWR(obj);
powerDBm = powerResponse/100;
power = 1e-3*10^(powerDBm/10);

    fprintf('
Current power is %4.3f mW, 1e3*power);
end

function ok = SetLowNoise(obj, sw)
    %turn dither off
if sw==true
    fprintf('
Turning Low Noise Mode on...
');
x = inputdlg('Timer setup: set Low--noise mode timeout [mins]', 'Timeout', 1, {'5 '});
if ~isempty(x);
    obj.timer_handle = GUI.TimerGUI(str2double(x{:}));
end
else
    fprintf('
Turning Low Noise Mode off...
');
end

OclaroTL5000.CleanMode(obj, 2+sw);
modeRead = OclaroTL5000.CleanMode(obj);
assert(modeRead==2+sw,...
ok = true;
if sw==true
    fprintf('ON');
else
    fprintf('OFF');
    if isvalid(obj.timer_handle)
        delete(obj.timer_handle);
    end
end
end
end
end

A.6.2 Functions

function checksum = BIP4 (data0, data1, data2, data3);
% BIP-4 checksum computed over a 32 bit word with the leading 4 bits
% prepended to the 28 bit packet and set to zero.
BIP8 = xor(hexToBinaryVector(data3, 8), ...
       hexToBinaryVector(data2, 8));
BIP8 = xor(BIP8, hexToBinaryVector(data1, 8));
BIP8 = xor(BIP8, hexToBinaryVector(data0, 8));
checksum = xor(BIP8(1:4), BIP8(5:8));
function channelResponse = Channel(obj, ch)

% N. Stan
% Channel sets the modules channel.

if nargin == 1
    WR = 0; ch = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = '30'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(ch, 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4([data0, data1, data2, data3]);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser.obj,sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdData_dec = fread(obj.ser.obj.obj.ser.obj.BytesAvailable);
    rxdData_hex = dec2hex(rxdData_dec);
    data = [rxdData_hex(3:1:2) rxdData_hex(4:1:2)]
    % convert response to binary
    rxdData_bin = hexToBinaryVector(dec2hex(rxdData_dec), 8);

    % Analyze response
    % Command specific bits
    % Channel number
    channelResponse = hex2dec(data);

    % Command independent bits
    % Communication Error
    CE = rxdData_bin(1, 5);
    if CE
        fprintf('\nChannel - Communication Error');
        LstRsp = 1;
    end
end

% Status Bits
Status = binaryVectorToDecimal(rxdData_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

%% this needs to be changed, the status will most of the time be 3, because it's queried very soon after issuing the command

%% Final wait for the command to execute
fprintf("
Setting the channel ...");
PurePhotonicsPPCL300.wait_operation(obj);
fprintf(SET!
);

../+OclaroTL5000/CleanJumpEnable.m

function fResponse = CleanJumpEnable(obj)

% N. Stan
% CleanJumpEnable activates Clean Jump

x=1;
commandRegister = 'ED'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(uint16(x), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

WR = 1;
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || Status
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxd = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxd_hex = dec2hex(rxd); % get size(rxd_hex) = [4 2]
    rxd_hex
end

    data = [rxd_hex(3:4:2) rxd_hex(4:4:2)];
    % convert response to binary
    rxd_bin = hexToBinaryVector(dec2hex(rxd_dec), 8);

    % Analyze response
    % Command specific bits
    % Channel number
function fResponse = CleanJumpGHz(obj, f)

% N. Stan
% CleanJumpGHz load the GHz part of the next frequency, with x in units of 0.1GHz
% 0.1*f – GHz part of the frequency (unsigned short)
if nargin == 1
    WR = 0; f = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = 'EB'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(uint16(f), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || Status
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4 ([data0, data1, data2, data3]);
data0(1) = binaryVectorToHex(BIP4);

sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
if size(rxdata_hex) == [4 2]
    rxdata_hex
    data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
%
% Command specific bits
% Channel number
fResponse = hex2dec(data);
%
% Command independent bits
% Communication Error
CE = rxdata_bin(1,5);
if CE
    fprintf('\nCleanJumpGHz – Communication Error');
    LstRsp = 1; continue;
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3,1:8));
% A series of eight flag bits indicating which operations ,
% if any , are still pending. Each operation that becomes pending
% is assigned one of these four bit positions .
% The module can be periodically polled ( by reading the NOP register)
% to determine which operations have completed .
% A value of 0x0 indicates that there are no currently pending operations.
%
% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1,7:8));
% 0x00 = OK flag. Normal return status
% 0x01 = XE flag. ( execution error)
% 0x02 = AEA flag. (Automatic extended addressing result being returned or ready to write)
% 0x03 = CP flag. Command not complete. pending
if Status == 1
    fprintf('\nCleanJumpGHz – Execution Error – retrying ...'); continue;
end
end

../+OclaroTL5000/CleanJumpTHz.m

function fResponse = CleanJumpTHz(obj, f)

% N. Stan
% CleanJumpTHz load the THz part of the next frequency , with x in units of THz
% f – THz part of the frequency (unsigned short)
if nargin == 1
    WR = 0; f = 0;
elseif nargin == 2
    WR = 1;
commandRegister = 'EA'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(uint16(f), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || Status
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(t) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj_ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj_ser_obj, obj_ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    if size(rxdata_hex) == [4 2]
        rxdata_hex
    end
    data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Analyze response
    % Command specific bits
    % Channel number
    fResponse = hex2dec(data);

    % Command independent bits
    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf('
CleanJumpTHz – Communication Error');
        LstRsp = 1; continue;
    end

    % Pending Operation Flags
    pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
    % A series of eight flag bits indicating which operations,
    % if any, are still pending. Each operation that becomes pending
    % is assigned one of these four bit positions.
    % The module can be periodically polled (by reading the NOP register)
    % to determine which operations have completed.
    % A value of 0x0 indicates that there are no currently pending operations.

    % Status Bits
    Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
    % 0x00 – OK flag. Normal return status
    % 0x01 – XE flag, (execution error)
function modeResponse = CleanMode(obj, mode)

% N. Stan
% Register 0x90: switch between the different low-noise modes.
% with 0 the standard dither-mode, 1 the no-dither mode and 2 the whisper-mode.
% The standard operating mode for the laser is the dither-mode.
% In this mode all the control loops are running and creating noise
% (especially in the 1–10000Hz range). By disabling these control-loops,
% either completely or partially, a lower noise behavior can be achieved.
% In the no-dither mode, the FM dither is disabled. Most other control-loops remain enabled.
% This results in the removal of the 888Hz tone (and its overtones).
% but does not address the noise below 100Hz. Note that this mode is not available
% on the most recent versions of the firmware (8.0.9 and 8.6.0) as we now recommend
% the whisper mode for all applications.
% In the whisper mode, essentially all control loops are disabled, to the extent possible.
% This significantly reduced the AM and FM noise in the below 100Hz range.
% If in any way possible, we recommend to once in a while switch back to
% the dither mode for the laser to relock. Such a switch–back could be done
% in less than 10 seconds.

if nargin == 1
    WR = 0; mode = 0;
else if nargin == 2
    WR = 1;
    assert(mode>=0&&mode<=2, 'Bad parameter passed to CleanMode')
end

commandRegister = '90'; %Register address for the command
data1 = commandRegister;
data23 = dec2hex(mode, 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);
end

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
% Command specific bits
% Mode
modeResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
CleanMode - Communication Error');
LstRsp = 1;
end

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 − OK flag, Normal return status
% 0x01 − XE flag, (execution error)
% 0x02 − AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 − CP flag, Command not complete, pending
%
% maybe add wait for pending operation to finish code

../OclaroTL5000/convertHexToASCII.m

function response = convertHexToASCII(wordHex)
% converts the hex values given
% into an array of ascii characters and displays them
response = char(hex2dec(wordHex));
if size(response, 1)>size(response, 2)
    response = response ';
end
disp(response ');

../OclaroTL5000/DitherA.m

function [ampResponse, CE, Status] = DitherA(obj, amp)
% N. Stan
% Dither Amplitude
% DitherA is an unsigned short integer encoded as
% the AM p-p amplitude deviation as 10*percentage of the optical power.
if nargin == 1
    WR = 0; amp = 0;
elseif nargin == 2
    WR = 1;
end
commandRegister = '5C'; %Register address for the command
data1 = commandRegister;
%conversion of rate into a signed hex, 16 bits long
data23 = dec2hex(amp, 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
    sendData = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendData);

    % Obtain response from the module
    pause(0.25);
    rxdDataDec = fread(obj.ser_obj, BytesAvailable);
    rxdDataHex = dec2hex(rxdDataDec);
data = [rxdDataHex(3,1:2) rxdDataHex(4,1:2)];
    % convert response to binary
    rxdDataBin = hexToBinaryVector(dec2hex(rxdDataDec), 8);

    % Analyze response
    % Command specific bits
    % Rate
    ampResponse = hex2dec(data);

    % Command independent bits
    % Communication Error
    CE = rxdDataBin(1, 5);
    if CE
        fprintf(’\nDitherA – Communication Error’);
        LstRsp = 1;
    end

    % Status Bits
    Status = binaryVectorToDecimal(rxdDataBin(1, 7:8));
    % 0x00 – OK flag, Normal return status
    % 0x01 – XE flag, (execution error)
    % 0x02 – AEA flag, (Automatic extended addressing result being returned or ready to write)
    % 0x03 – CP flag, Command not complete, pending

../../../OclaroTL5000/DitherE.m

function [WFResponse, DDEResponse, CE, Status] = DitherE(obj, WF, DDE)
% N. Stan
% Dither Enable
% The dither registers provide a way to configure dither performance of the units.
% WF – Waveform : 00 – Sinusoidal; 01 – Triangular
if nargin == 1
    WR = 0; WF = 0; DDE = 0;
else if nargin == 3
    WR = 1;
end

commandRegister = '59';
data1 = commandRegister;

data2 = '00';
data3 = binaryVectorToHex([0 0 decimalToBinaryVector(WF, 2) 0 0 DDE 0]);

LstRsp = 0;

while CE
    data0 = binaryVectorToHex(data0_b);
    data0 = binaryVectorToHex(data0);

    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    rxd = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxd = dec2hex(rxd);
    data = [rxdata.hex(3,1:2) rxdata.hex(4,1:2)]

    rxdata_bin = hexToBinaryVector(dec2hex(rxd), 8);

    rxd = hex2dec(data);
    WFResponse = binaryVectorToDecimal(rxdata_bin(3:4));
    DDEResponse = binaryVectorToDecimal(rxdata_bin(7));

    CE = rxdata_bin(1, 5);
    if CE
        fprintf ('\nDitherE - Communication Error');
    end
    LstRsp = 1;
end

Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));

0x00 - OK flag. Normal return status
0x01 - XE flag. (Execution error)
0x02 - AEA flag. (Automatic extended addressing result being returned or ready to write)
0x03 - CP flag. Command not complete, pending
function [freqResponse, CE, Status] = DitherF(obj, freq)

% N. Stan
% Dither Frequency deviation
% DitherA is an unsigned short integer encoded as
% FM p-p frequency deviation as GHz*10.

if nargin == 1
    WR = 0; freq = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = '5B'; % Register address for the command
data1 = commandRegister;
% conversion of rate into a signed hex, 16 bits long
data23 = dec2hex(freq, 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Analyze response
function [rateResponse, CE, Status] = DitherR(obj, rate)

% N. Stani
% Dither Rate
% DitherR is an unsigned integer specifying the dither rate as kHz.
% Note that DitherE is used to set the waveform for this frequency.

if nargin == 1
    WR = 0; rate = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = '5A'; % Register address for the command
data1 = commandRegister;
% Conversion of rate into a signed hex, 16 bits long
data23 = dec2hex(rate, 4);
data2 = data23(1:2); % First data byte in hex
data3 = data23(3:4); % Second data byte in hex
LstRsp = 0; % Bits set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % Set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0 LstRsp Bits26_25 WR];
data0 = binaryVectorToHex(data0_b);
% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);

% Command specific bits
freqResponse = hex2dec(data);

% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('\nDitherF - Communication Error');
    LstRsp = 1;
end

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag. Normal return status
% 0x01 - XE flag. (Execution error)
% 0x02 - AEA flag. (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag. Command not complete, pending
rxdata_hex = dec2hex(rxdata_dec);
data = [ rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
% Command specific bits
% Rate
rateResponse = hex2dec(data);
% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('\nDitherR - Communication Error');
    LstRsp = 1;
end

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

% this needs to be changed, the status will most of the time be 3, because it’s queried very soon after issuing the command

% Final wait for the command to execute
if WR
    fprintf('\nRate ...');
    PurePhotonicsPPCL300.wait_operation(obj)
    fprintf('\nSET!\n');
end

../+OclaroTL5000/FCF1.m

function fResponse = FCF1(obj, f)
% N. Stan
% The FCF1 and FCF2 registers provide a way to configure the frequency of channel 1.
% f - THz part of the frequency (unsigned short)
% This value can only be changed when the output is disabled.

if nargin == 1
    WR = 0; f = 0;
else if nargin == 2
    WR = 1;
end

commandRegister = '35'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(uint16(f), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || Status
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
end

    % Convert to hexadecimal
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdatal_dec = fread(obj.ser_obj, BytesAvailable);
    rxdatal_hex = dec2hex(rxdatal_dec);
    if size(rxdatal_hex) == [4 2]
        rxdatal_hex
    end
    data = [rxdatal_hex(3,1:2) rxdatal_hex(4,1:2)];
    % Convert response to binary
    rxdatal_bin = hexToBinaryVector(dec2hex(rxdatal_dec), 8);

    % Analyze response
    % Command specific bits
    % Channel number
    fResponse = hex2dec(data);

    % Command independent bits
    % Communication Error
    CE = rxdatal_bin(1, 5);
    if CE
        fprintf ('\nFCFI - Communication Error');
        LstRsp = 1; continue;
    end

    % Pending Operation Flags
    pendingOperation = binaryVectorToHex(rxdatal_bin(3, 1:8));
    % A series of eight flag bits indicating which operations,
    % if any, are still pending. Each operation that becomes pending
    % is assigned one of these four bit positions.
    % The module can be periodically polled (by reading the NOP register)
    % to determine which operations have completed.
    % A value of 0x0 indicates that there are no currently pending operations.

    % Status Bits
    Status = binaryVectorToDecimal(rxdatal_bin(1, 7:8));
    % 0x00 - OK flag, Normal return status
    % 0x01 - XE flag, (execution error)
    % 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
    % 0x03 - CP flag, Command not complete, pending
    if Status == 1
        fprintf ('\nFCFI - Execution Error - retrying...'); continue;
    end
function fResponse = FCF2(obj, f)

% N. Stan
% The FCF1 and FCF2 registers provide a way to configure the frequency of channel 1.
% f GHz*10 part of the frequency (unsigned short)
% This value can only be changed when the output is disabled.

if nargin == 1
    WR = 0; f = 0;
else if nargin == 2
    WR = 1;
end

commandRegister = '36'; % Register address for the command
data1 = commandRegister;

data23 = dec2hex(uint16(f), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default

CE = 1; Status = 1;

while CE || Status
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);

    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    if size(rxdata_hex)’==[4 2]
        rxdata_hex
    end
    data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];

    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Analyze response

    % Command specific bits
    % Channel number
    fResponse = hex2dec(data);

    % Command independent bits
    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf ('\nFCF2 - Communication Error');
        LstRsp = 1; continue;
    end
end
% Pending Operation Flags

```matlab
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
```

% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

```matlab
% Status Bits

Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
```

% 0x00 - OK flag. Normal return status
% 0x01 - XE flag. (execution error)
% 0x02 - AEA flag. (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag. Command not complete, pending

```matlab
if Status == 1
    fprintf('nFCF2 - Execution Error - retrying...'); continue;
end
```

```matlab
function fResponse = FTF(obj, f)
```

% The FTF register provides fine tune adjustment of the laser's wavelength
% from the set channel. The adjustment is applied to all channels uniformly.
% Returned value is in MHz, signed short.

```matlab
if nargin == 1
    f = 0;
end
```

```matlab
fResponse = 0; %Oclaro ITLA doesn’t support fine tuning
```

```matlab
function ftfR = FTFR(obj)
```

% Query the minimum/maximum fine tune frequency range capability of the
% module in MHz
% the returned value ftfR means the range is (−ftfR, ftfR) in MHz

```matlab
commandRegister = '4F'; %Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex
```

```matlab
WR = 0; %Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
```

```matlab
Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || Status
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
data0 = binaryVectorToHex(data0_b);

% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
assert(obj.ser_obj.BytesAvailable > 0, 'No response from the laser');
rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
% Command specific bits
% Channel number
ftfR = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
FTFR = Communication Error'');
    LstRsp = 1;
    end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 – OK flag. Normal return status
% 0x01 – XE flag (execution error)
% 0x02 – AEA flag (Automatic extended addressing result being returned or ready to write)
% 0x03 – CP flag (Command not complete, pending
if Status == 1
    fprintf('
FCF1 = Execution Error – retrying...'); continue;
    end

../+OclaroTL5000/getAvailableComPort.m

function lCOM_Port = getAvailableComPort()
% function lCOM_Port = getAvailableComPort()
% Return a Cell Array of COM port names available on your computer
try
    s = serial('IMPOSSIBLE_NAME_ON_PORT'); fopen(s);
catch
lErrMsg = lasterr;
end

%Start of the COM available port
lIndex1 = findstr(lErrMsg, 'COM');
%End of COM available port
lIndex2 = findstr(lErrMsg, 'Use') - 3;

lComStr = lErrMsg(lIndex1:lIndex2);

%Parse the resulting string
lIndexDot = findstr(lComStr, ', ');

% If no Port are available
if isempty(lIndex1)
lCOM{1} = '';
return;
end

% If only one Port is available
if isempty(lIndexDot)
lCOM{1} = lComStr;
return;
end

lCOM{1} = lComStr(1:lIndexDot(1)-1);

for i=1:numel(lIndexDot)+1
% First One
if (i==1)
lCOM{1,1} = lComStr(1:lIndexDot(i)-1);
% Last One
elseif (i==numel(lIndexDot)+1)
lCOM{1,1} = lComStr(lIndexDot(i-1)+2:end);
% Others
else
lCOM{1,1} = lComStr(lIndexDot(i-1)+2:lIndexDot(i)-1);
end
end

../+OclaroTL5000/Grid.m

function [gResponse, CE, pendingOperation, Status] = Grid(obj, g)

% N. Stan
% Grid sets the modules grid spacing for the channel to frequency mapping.
% g – grid spacing in GHz*10 signed short
% This value can only be changed when the output is disabled.

if nargin == 1
WR = 0; g = 0;
elseif nargin == 2
WR = 1;
end

commandRegister = '34'; %Register address for the command
data1 = commandRegister;
%conversion of g into a signed hex, 16 bits long
data23 = dec2hex(typecast(int16(g), 'uint16'), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26, 25 = [0 0]; % set to zero by default

CE = 1; Status = 1;
while CE || Status
    data0_b = [[0 0 0 0] LstRsp Bits26, 25 WR];
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser.obj, sendLine);
    pause(0.25);
    rxdata_dec = fread(obj.ser.obj, obj.ser.obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);
    % % Obtain response from the module
    % % Pending Operation Flags
    pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
    % A series of eight flag bits indicating which operations ,
    % if any, are still pending. Each operation that becomes pending
    % is assigned one of these four bit positions .
    % The module can be periodically polled (by reading the NOP register)
    % to determine which operations have completed.
    % A value of 0x0 indicates that there are no currently pending operations.
    % Status Bits
    Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
    % 0x00 – OK flag , Normal return status
    % 0x01 – XE flag , ( execution error)
    % 0x02 – AEA flag , ( Automatic extended addressing result being returned or ready to write)
    % 0x03 – CP flag , Command not complete , pending
    % % Command specific bits
    % Channel number
    gResponse = double(typecast(uint16(hex2dec(data)), 'int16'));
    % % Command independent bits
    % % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf('
Grid – Communication Error');
        LstRsp = 1;
    end
    if Status == 1
        fprintf('
Grid – Execution Error – retrying ...'); continue;
    end
end

../+OclaroTL5000/LF1.m

function freqResponse = LF1(obj)
% N. Stan
% Reads the frequency of the current channel Unisigned short [THz]
commandRegister = '40'; %Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex
WR = 0; %Command is Read–only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR];
data0 = binaryVectorToHex(data0_b);
% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPLCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);
rxdatal_hex = dec2hex(rxdatal_dec);
data = [rxdatal_hex(3,1:2) rxdatal_hex(4,1:2)];
% convert response to binary
rxdatal_bin = hexToBinaryVector(dec2hex(rxdatal_dec), 8);

% Analyze response
% Command specific bits
% Channel number
freqResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdatal_bin(1, 5);
if CE
    fprintf('
LF1 – Communication Error');
LstRsp = 1;
end
% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdatal_bin(3, 1:8));
% A series of eight flag bits indicating which operations.
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.
% Status Bits
Status = binaryVectorToDecimal(rxdatal_bin(1, 7:8));
% 0x00 – OK flag, Normal return status
% 0x01 – XE flag, (execution error)
% 0x02 – AEA flag. (Automatic extended addressing result being returned or ready to write)
% 0x03 – CP flag, Command not complete, pending

261
function freqResponse = LF2(obj)

% N. Stan
% Reads the GHz part and first decimal of the frequency of the current channel Unisigned short [ GHz]

commandRegister = '41'; %Register address for the command
data1 = commandRegister;
data2 = '00'; %first data byte in hex
data3 = '00'; %second data byte in hex

WR = 0; %Command is Read-only

LstRsp = 0; %Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; %set to zero by default
CE = 1;
while CE
    data0 = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0);

    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj,sendLine);

    pause(0.25);
    rxdataldec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdatalhex = dec2hex(rxdataldec);
    data = [rxdatal_hex(3:1:2) rxdatal_hex(4:1:2)];
    rxdatalbin = hexToBinaryVector(dec2hex(rxdataldec), 8);

    % Analyze response
    % Command specific bits
    % Channel number
    freqResponse = hex2dec(data);

    % Command independent bits
    % Communication Error
    CE = rxdatalbin(1, 5);
    if CE
        fprintf('
LF2 – Communication Error');
        LstRsp = 1;
    end

    % Pending Operation Flags
    pendingOperation = binaryVectorToHex(rxdatalbin(3, 1:8));
    % A series of eight flag bits indicating which operations.
    % If any, are still pending. Each operation that becomes pending
    % is assigned one of these four bit positions.
    % The module can be periodically polled (by reading the NOP register)
    % to determine which operations have completed.
A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 – OK flag, Normal return status
% 0x01 – XE flag, (execution error)
% 0x02 – AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 – CP flag, Command not complete, pending

../+OclaroTL5000/LFH1.m

function freqResponse = LFHI(obj)

% N. Stan
% Returns the THz part of the max frequency
commandRegister = '54'; %Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex
WR = 0; %Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Analyze response
    % Channel number
    freqResponse = hex2dec(data);

    % Command independent bits
    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf('
LFHI - Communication Error');
        LstRsp = 1;
    end
end
% Pending Operation Flags

% A series of eight flag bits indicating which operations, if any, are still pending. Each operation that becomes pending is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register) to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits

% Returns the GHz part with the first decimal of the max frequency

commandRegister = '55'; % Register address for the command

data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex

WR = 0; % Command is Read-only

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;

while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);

    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdaata_dec);
    data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
    % convert response to binary
    rxdata_b = hexToBinaryVector(dec2hex(rxdaata_dec), 8);

    Analyze response

% Command specific bits

% Channel number
freqResponse = hex2dec(data);
% Command independent bits
% Communication Error
CE = rxd_data_bin(1, 5);
if CE
    fprintf('
\nFH2 - Communication Error');
    LstRsp = 1;
end
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxd_data_bin(3, 1:8));
%A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled by reading the NOP register.
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxd_data_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

../+OclaroTL5000/LFL1.m

function freqResponse = LFL1(obj)
% N. Stan
% Returns the THz part of the min frequency

commandRegister = '52'; %Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex
WR = 0; %Command is Read–only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% BiT set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR];
data0 = binaryVectorToHex(data0_b);

% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser.obj, sendLine);

% Obtain response from the module
pause(0.25);
rxdData_dec = fread(obj.ser.obj, obj.ser.obj.BytesAvailable);
rxdData_hex = dec2hex(rxdData_dec);
data = [rxdData_hex(3:1:2) rxdData_hex(4:1:2)];
% convert response to binary
\[ r x d a t a _ { \text{bin}} = \text{hexToBinaryVector}(\text{dec2hex}(r x d a t a _ { \text{dec}}), 8); \]

\[ \% \text{ Analyze response} \]
\[ \% \text{ Command specific bits} \]
\[ \% \text{ Channel number} \]
\[ \text{freqResponse} = \text{hex2dec}(\text{data}); \]

\[ \% \text{ Command independent bits} \]
\[ \% \text{ Communication Error} \]
\[ \text{CE} = r x d a t a _ { \text{bin}}(1, 5); \]
\[ \text{if CE} \]
\[ \text{fprintf}(''nLFL1 − Communication Error''); \]
\[ \text{LstRsp} = 1; \]
\[ \text{end} \]
\[ \text{end} \]

\[ \% \text{ Pending Operation Flags} \]
\[ \text{pendingOperation} = \text{binaryVectorToHex}(r x d a t a _ { \text{bin}}(3, 1:8)); \]
\[ \% A series of eight flag bits indicating which operations, \]
\[ \% if any, are still pending. Each operation that becomes pending \]
\[ \% is assigned one of these four bit positions. \]
\[ \% The module can be periodically polled (by reading the NOP register) \]
\[ \% to determine which operations have completed. \]
\[ \% A value of 0x0 indicates that there are no currently pending operations. \]

\[ \% \text{ Status Bits} \]
\[ \text{Status} = \text{binaryVectorToDecimal}(r x d a t a _ { \text{bin}}(1, 7:8)); \]
\[ \% 0x00 − OK flag. Normal return status \]
\[ \% 0x01 − XE flag. Execution error \]
\[ \% 0x02 − AEA flag. Automatic extended addressing result being returned or ready to write \]
\[ \% 0x03 − CP flag. Command not complete, pending \]

```matlab
../+OclaroTL5000/LFL2.m
```

```matlab
function freqResponse = LFL2(obj)

\% N. Stan
\% Returns the GHz part with the first decimal of the min frequency
\ncommandRegister = '52'; \% Register address for the command
data1 = commandRegister;
data2 = '00'; \% first data byte in hex
data3 = '00'; \% second data byte in hex

\% WR = 0; \% Command is Read-only
\nLstRsp = 0; \% Bit set to logic 0 when the checksum is consistent.
\% Bit set to logic 1 forces module to resend last valid packet.
\% Used when the checksum is inconsistent.

\% Bits26_25 = [0 0]; \% set to zero by default
CE = 1;
while CE
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
    data0 = binaryVectorToHex(data0_b);

\% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
sendLine = uint8(hex2dec([data0; data1; data2; data3]));
```

266
fwrite(obj.ser_obj.sendLine);

% Obtain response from the module
pause(0.25);
rxdataldec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdatalhex = dec2hex(rxdataldec);
data = [rxdatalhex(3,1:2) rxdatalhex(4,1:2)];
% convert response to binary
rxdatalbin = hexToBinaryVector(dec2hex(rxdataldec), 8);

% Analyze response
% Command specific bits
% Channel number
freqResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdatalbin(1,5);
if CE
    fprintf('
LFL2 - Communication Error');
    LstRsp = 1;
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdatalbin(3,1:8));
% A series of eight flag bits indicating which operations are still pending. Each operation that becomes pending is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register) to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdatalbin(1,7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

../OclaroTL5000/LGrid.m

function [freqResponse, CE, pendingOperation, Status] = LGrid(obj)

% N. Stan
% Reads the GHz part and first decimal of the minimum grid spacing capability of the module
commandRegister = '56'; % Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex
WR = 0; % Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);

    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Analyze response
    % Command specific bits
    % Channel number
    freqResponse = hex2dec(data);

    % Command independent bits
    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf('
LGrid - Communication Error');
        LstRsp = 1;
    end

    % Pending Operation Flags
    pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
    % A series of eight flag bits indicating which operations
    % if any, are still pending. Each operation that becomes pending
    % is assigned one of these four bit positions.
    % The module can be periodically polled (by reading the NOP register)
    % to determine which operations have completed.
    % A value of 0x0 indicates that there are no currently pending operations.

    % Status Bits
    Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
    % 0x00 - OK flag, Normal return status
    % 0x01 - XE flag, (execution error)
    % 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
    % 0x03 - CP flag, Command not complete, pending

../+OclaroTL5000/NoDriftMode.m

function EnaDisResponse = NoDriftMode(obj, EnaDis)

    % N. Stan
    % Register 0x91: enable (1) or disable (0) the no-drift mode

    % In the standard telecom mode the laser has many control loops running to
    % ensure stability over the long run. For sensor applications we recommend the
    % whispermode, where these control-loops have been mostly disabled.
    % This does result in drift over time, and the laser needs to be put back
% into standard mode regularly for a short period of time (10 secs) for relocking.
% For critical applications, where long term stability is important
% (i.e. the laser needs to be locked), but where the drift needs to be limited.
% the no-drift mode has optimized the control loops to have a factor 10x less noise
% (some limits apply for the temperature ramp-rate etc.). The no-drift calibration
% involves the characterization of over temperature variation and compensation
% of the measured change rate based on ambient temperature.

% The unit starts up in the no-drift mode.

if nargin == 1
    WR = 0; EnaDis = 0;
else if nargin == 2
    WR = 1;
    assert (EnaDis == 0||EnaDis ==1, 'Bad parameter passed to CleanMode')
end

commandRegister = '91'; %Register address for the command
data1 = commandRegister;
data23 = dec2hex(EnaDis, 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [Bits26_25 LstRsp 0 WR];
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Analyze response
    % Command specific bits
    EnaDisResponse = hex2dec(data);
    % Mode
    if EnaDisResponse == 1
        fprintf ('\nNoDriftMode - Communication Error');
        LstRsp = 1;
    end
    % Command independent bits
    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf ('\nNoDriftMode - Communication Error');
        LstRsp = 1;
    end
    % Status Bits

269
function [MRDY, errorField, pendingOperation, CE, Status] = NOP(obj)

% Nikola Stan stan.nikola@gmail.com
% The NOP register provides a way to access the module's status, returning pending operation status, and the current value of the error field. This register may be read upon receiving an execution error for an immediately preceding command. It can also be polled to determine the status of pending operations.

commandRegister = '00'; % Register address for the NOP command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
WR = 0; % Read 0, Write 1
CE = 1;

while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);

    %Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

    % sending the formatted command to the laser
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj,sendLine);
    LstRsp = 0;
    % Obtain response from the module
    pause(0.25);
    if obj.ser_obj.BytesAvailable == 0
        disp('zero Bytes Available'); LstRsp = 1; continue;
    end
    rxdata_dec = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);

    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);
    if size(rxdata_bin)^=[4 8]
        rxdata_bin
    end

    % Analyze response

    % Command independent bits

    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf('
NOP - Communication Error');
        LstRsp = 1; continue;
    end
end
% Command specific bits
% Pending Operation Flags
% A series of eight flag bits indicating which operations, if any, are still pending. Each operation that becomes pending is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register) to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Module Ready Bit
try
    MRDY = rxdata_bin(4, 4);
catch
disp('');
end
% MRDY - Module Ready
% When 1 indicates that the module is ready for its output to be enabled.
% When 0 indicates that the module is not ready for its output to be enabled.

% Error Field Bits
errorField = binaryVectorToDecimal(rxdata_bin(4, 5:8));
% 0x00 OK - Ok, no errors
% 0x01 RNI - The addressed register is not implemented
% 0x02 RW - Register not write able; register cannot be written (read only)
% 0x03 RE - Register value range error; writing register contents causes value range error; contents unchanged
% 0x04 CIP - Command ignored due to pending operation
% 0x05 CII - Command ignored while module is initializing, warming up, or contains an invalid configuration.
% 0x06 ERE - Extended address range error (address invalid)
% 0x07 ERO - Extended address is read only
% 0x08 EFG - Execution general failure
% 0x09 CIE - Command ignored while modules optical output is enabled (carrying traffic)
% 0x0A IVC - Invalid configuration, command ignored
% 0x0B-0x0E - Reserved for future expansion
% 0x0F VSE Vendor specific error (see vendor specific documentation for more information)

end

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 OK flag, Normal return status
% 0x01 XE flag, (execution error)
% 0x02 AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 CP flag, Command not complete, pending

..+/OclaroTL5000/OOP.m

function [powerResponse, CE, pendingOperation, Status] = OOP(obj)
% N. Stan
% Reads the external optical power estimate of the module as dBm*100 signed short
    commandRegister = '42'; % Register address for the command
    data1 = commandRegister;
    data2 = '00'; % first data byte in hex
    data3 = '00'; % second data byte in hex
```matlab
% Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser, obj, sendLine);

    pause(0.25);
    rxdata_dec = fread(obj.ser, obj, BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    data = [rxdata_hex(3:1:2) rxdata_hex(4:1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj.ser, obj, BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    data = [rxdata_hex(3:1:2) rxdata_hex(4:1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Analyze response
    % Command specific bits
    powerResponse = double(typcast(uint16(hex2dec(data)), 'int16'));

    % Command independent bits
    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf('nOOP - Communication Error');
        LstRsp = 1;
    end
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending
```

```
../OclaroTL5000/OPSH.m
```

```
function powerResponse = OPSH(obj)
```

272
% N. Stahn
% Reads the maximum optical capability of the module as dBm*100 signed
% short

commandRegister = '51'; %Register address for the command
data1 = commandRegister;
data2 = '00'; %first data byte in hex
data3 = '00'; %second data byte in hex

WR = 0; %Command is Read-only
LstRsp = 0; %Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; %set to zero by default
CE = 1;

while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);
    %Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);
end

pause(0.25);
rxdataldec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdatalhex = dec2hex(rxdataldec);
data = [rxdatal_hex(3:1:2) rxdatal_hex(4:1:2)];
%convert response to binary
rxdatalbin = hexToBinaryVector(dec2hex(rxdataldec), 8);

%Analyze response
%Command specific bits
%Channel number
powerResponse = double(typecast(uint16(hex2dec(data)), 'int16'));
%Command independent bits
%Communication Error
CE = rxdatalbin(1, 5);
if CE
    fprintf('\nOPSH - Communication Error');
LstRsp = 1;
end

%Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdatalbin(3, 1:8));
%A series of eight flag bits indicating which operations,
%if any, are still pending. Each operation that becomes pending
%is assigned one of these four bit positions.
%The module can be periodically polled (by reading the NOP register)
%to determine which operations have completed.
%A value of 0x0 indicates that there are no currently pending operations.

%Status Bits
Status = binaryVectorToDecimal(rxdatalbin(1, 7:8));
%0x00 - OK flag. Normal return status
%0x01 - XE flag. (execution error)
%0x02 - AEA flag. (Automatic extended addressing result being returned or ready to write)
%0x03 - CP flag. Command not complete, pending

273
function powerResponse = OPSL(obj)

% N. Stan
% Reads the minimum optical capability of the module as dBm*100 signed
% short
commandRegister = '50'; % Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex
WR = 0; % Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPLCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdaDataDec);
data = [rxdata_hex(3:1:2) rxdata_hex(4:1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdaDataDec), 8);

    % Analyze response
    % Command specific bits
    % Channel number
    powerResponse = double(typecast(uint16(hex2dec(data)), 'int16'));

    % Command independent bits
    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf('
OPSL - Communication Error');
    LstRsp = 1;
    end
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.
function powerResponse = PWR(obj, power)

% N. Stan
% Sets or reads the power level as dBm*100
% power - the power in (dBm*100) to be set. signed short
if nargin == 1
    WR = 0; power = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = '31'; % Register address for the command
data1 = commandRegister; % Conversion of power into a signed hex, 16 bits long
data23 = dec2hex(typecast(int16(power), 'uint16'), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bits set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj,sendLine);

    % Obtain response from the module
    pause(0.25);
rxdatabin = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);
rxdatabin = dec2hex(rxdatabin);
data = [rxdatabin(3:1:2) rxdatabin(4:1:2)];
end
end

% Analyze response

% Command specific bits
% Channel number
powerResponse = double(typecast(uint16(hex2dec(data)), 'int16'));

% Command independent bits
% Communication Error
CE = rxdatabin(1, 5);
if CE
    fprintf('\n\nPWR - Communication Error');
LstRsp = 1;
end
% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));

% A series of eight flag bits indicating which operations
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

%%%% this needs to be changed, the status will most of the time be 3, because it’s queried very
% Final wait for the command to execute
if WR
    fprintf(‘\nSetting the power . . . ’);
    PurePhotonicsPPCL300.wait_operation(obj)
    fprintf(‘SET!\n’);
end

../+OclaroTL5000/ResEna.m

function pendingOperation = ResEna(obj, WR, SENA, MR, SR)
if nargin == 1
    WR=0; SENA = 0; MR = 0; SR = 0;
elseif nargin==3
    MR = 0; SR = 0;
end

commandRegister = ‘32’; %Register address for the command
data1 = commandRegister;
data2 = ‘00’; % first data byte in hex
data3 = binaryVectorToHex([0 0 0 SENA 0 SR MR]); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

% Bits26_25 = [0 0]; % set to zero by default
% CE = 1; repeat = 1;
while CE||repeat
    data0_b = ([0 0 0 0] LstRsp Bits26_25 WR);
    data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4 (data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);

    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj.sendLine);
end

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response

% Command specific bits
% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, :8));
% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
ResEna - Communication Error');
    LstRsp = 1;
end

% Status Bits
status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending
if status == 1
    fprintf('
ResEna - Execution Error, retrying...');
    repeat = 1; continue;
else
    repeat = 0;
end

% Final wait for the command to execute
fprintf('
Laser...');

PurePhotonicsPPCL300.wait_operation(obj)
switch SENA
    case 0
        fprintf('OFF!');
    case 1
        fprintf('ON!');
end

../OclaroTL5000/StatusF.m

function [flags, CE, Status] = StatusF(obj, clear)

% Nikola Stan stan.nikola@gmail.com
% The StatusF and StatusW commands return the tunable laser status
% upon a read and provide a way to clear status flags on a write.
% There are two status registers, one that primarily indicates FATAL
% conditions (0x20) and the other that primarily indicates WARNING conditions (0x21).
if nargin == 1
    clear = 0;
commandRegister = '20'; %Register address for the NOP command
data1 = commandRegister;
if clear
    dataBits = 1; WR = 1;
else
    dataBits = 0; WR = 0;
end
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex
data23 = dec2hex(uint16(dataBits), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default

CE = 1; Status = 1;
while CE || Status
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);

    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    assert(all(size(rxdata_hex)==[4 2]), 'Error! Laser response is the wrong size');
data = [rxdata_hex(3, 1:2) rxdata_hex(4, 1:2)];
    flags = rxdata_bin(3:4, :);

    SRQ = rxdata_bin(3, 1);
    ALM = rxdata_bin(3, 2);
    FATAL = rxdata_bin(3, 3);
    DIS = rxdata_bin(3, 4);

    SRQ = Service Request Bit (read only) (default 0)
    The SRQ bit is read only. When the ALM* condition is asserted, this bit is set to 1.
    The conditions which assert the ALM* condition are fully configurable
    through the alarm trigger register (0x28).

    ALM = ALARM Flag bit (read-only) (default 0)
    The ALM bit is read only. When the ALM* condition is asserted, this bit is set to 1.
    The conditions which assert the ALM* condition are fully configurable
    through the alarm trigger register (0x2A).

    FATAL = FATAL alarm bit (read-only) (default 0)
    The FATAL bit is read only. When the FATAL* condition is asserted,
    this bit is set to 1. The conditions which set the FATAL* condition
    are fully configurable through the fatal trigger register (0x29).

    DIS = Modules output is hardware disabled (read-only)
    The module's laser output disable bit is read only and represents
    the state of the hardware disable pin (DIS*).
% When set to one, the module is hardware disabled.
% When the DIS+ pin is set to zero, the SENA bit is also cleared.
% Therefore when DIS+ is set to one, the module does not re-enable
% the output until the SENA bit is also set.
% Any state change in DIS can cause SRQ+ to be asserted
% if the appropriate SRQ+ trigger is set.
% 1: Module disabled (DIS+ line is low)
% 0: DIS+ line is high

DIS = rxdata_bin(3, 4);

% FVSF, WVSF  Vendor Specific Fault (read-only) (default 0)
% The FVSF bit (0x20) is set to 1 whenever a fatal vendor specific condition
% is asserted. The WVSF bit (0x21) is set to 1 whenever
% a warning vendor specific condition is asserted.
% If either of these bits is set, the vendor will have a register
% defined which contains vendor specific fault conditions.
% This bit is also asserted when laser aging thresholds are exceeded

FVSF = rxdata_bin(3, 5);

% FFREQ & WREQ  Frequency Fatal and Warning (read-only) (default 0)
% The FFREQ bit (0x20) reports that the frequency deviation has exceeded
% the frequency fatal threshold (0x24) while WREQ bit (0x21) reports that
% the frequency deviation has exceeded the frequency warning threshold (0x25).
% When bit 10 is 1, it indicates that the frequency deviation threshold
% is being exceeded. When bit 10 is 0, the frequency deviation threshold
% is not being exceeded.

FFREQ = rxdata_bin(3, 6);

% FITHERM &WITHERM  Thermal Fatal and Warning (read-only) (default 0)
% The FITHERM bit (0x20) reports that the thermal deviation has exceeded
% the thermal fatal threshold (0x26) while WITHERM bit (0x21) reports
% that the thermal deviation has exceeded the thermal warning threshold (0x27).
% When bit 9 is 1, it indicates that the thermal deviation threshold
% is being exceeded. When bit 9 is 0, the thermal deviation threshold
% is not being exceeded.

FITHERM = rxdata_bin(3, 7);

% FPWR & WWR  Power Fatal and Warning (read-only) (default 0)
% The FPWR bit (0x20) reports that the power deviation has exceeded
% the power fatal threshold (0x22) while WWR bit (0x21) reports that
% the power deviation has exceeded the power warning threshold (0x23).
% When bit 8 is 1, it indicates that the power deviation threshold
% is being exceeded. When bit 8 is 0, the power deviation threshold
% is not being exceeded.

FPWR = rxdata_bin(3, 8);

% XEL  Flags an execution error.
% A 1 indicates an exceptional condition. Note that execution errors could be
% generated by a command just given which failed to execute as well as
% a command that was currently executing (a pending operation that just complete).
% The default RS232 configuration only sets XEL when a pending operation fails.
% The XE bit remains set until cleared.

XEL = rxdata_bin(4, 1);

% CEL  Flags a communication error.
% A 1 indicates a communication error. The CE bit remains set until cleared.

CEL = rxdata_bin(4, 2);

% MRL  Module Restarted (latched) (default 1 by definition)
% MRL can be read or set to zero. When it is 1, it indicates that
% the module has been restarted either by power up, by hardware or software reset,
% or by a firmware mandated restart. Depending upon the implementation,
% this may indicate that the laser's output signal may be invalid.
% Note that the module can be reset through the communication interface
% by writing to register 0x32. The bit remains set until cleared.

MRL = rxdata_bin(4, 3);

% CRL  Communication Reset (latched) (default 1 by definition)
% CRL can be read or set to zero. When it is set, it indicates that
% the module has undergone a communication interface reset.
% The input buffers were cleared. This can also occur after a manufacturer
% specific timeout period has elapsed in the middle of a packet transfer.
% 38 The bit remains set until cleared.
CRL = rxdata_bin(4, 4);

% FVSFL, FFREQL, FHERML, FPWRL, WVSFL, WFREQL, WHERML, WPWRL Latched fatal
% and warning indicators (RW) (default 0)
% These flags are latched versions of bits 11–8 for the fatal and warning
% threshold deviations. These bit indicators can be cleared by writing
% a 1 to these bit positions.
% When any of these bits is 1, it indicates that the corresponding deviation
% threshold has been exceeded at sometime in past (since the last clear)
% and may still be occurring.
% When any of these bits are 0, the corresponding deviation threshold
% has not occurred since the last clear.
FVSFL = rxdata_bin(4, 5);
FFREQL = rxdata_bin(4, 6);
FHERML = rxdata_bin(4, 7);
FPWRL = rxdata_bin(4, 8);

%% Command independent bits
%% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf( '\nFCF1 - Communication Error');
    LstRsp = 1; continue;
end

%% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
0x00 - OK flag, Normal return status
0x01 - XE flag, (execution error)
0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
0x03 - CP flag, Command not complete, pending
if Status == 1
    fprintf( '\nFCF1 - Execution Error - retrying...' ); continue;
end

 function wait_operation(obj)
   timer_start = now;
   completeFlag = false;
   pending0 = 0;
   while ~completeFlag && ((now-timer_start)+24*3600)<obj.max_time
       ['", pending , CE."'] = PurePhotonics:PPCL300_NOP(obj);
       if (pending0 == hex2dec(pending)) || CE
           fprintf( '\nFCF1 - ', hex2dec(pending), CE);
           pending0 = hex2dec(pending);
       end
       fprintf( '.');
       if ~hex2dec(pending)); completeFlag=true; end
       pause(0.25);
   end
   assert(((now-timer_start)+24*3600)<obj.max_time, 'Max waiting time exceeded!');
A.7 PurePhotonicsPPCL300 Laser support files

A.7.1 Classes

../+PurePhotonicsPPCL300/Laser.m

classdef Laser <interfaces.ILaser

% LASER Summary of this class goes here
% Detailed explanation goes here

properties (Access=public)

ser_obj=false

end

properties (Access=private)

com_port

end

properties (Transient, SetAccess = protected)

timer_handle = false

end

properties (Constant)

max_time=60; % max time to wait after tuning [s]
ptime=5; % pause time used after setting a value for power

% Notes on USB com ports: if there are communication errors, change
% the com port number to a random number to avoid system conflicts.
% Connect the laser directly to the mother board, don't connect
% through a USB hub or front panel. Windows com port settings:
% Baud rate: 112500,
% data bits read and write to minimum: 64 bits,
% latency minimum: 1 ms

Port=[1 1 1]: % do not use wildcards here
wvError = 0.1; % [nm]
fError = 0.001;
pwError = 0.5e-3;

end

methods

function obj = Laser(port)

% Laser outputs [1527.60488 1567.13256] when queried

obj.WavelengthMin = 1528e-9;
obj.WavelengthMax = 1565e-9;
obj.WavelengthStep = 1e-12; % 1 pm

%dBm, the laser gives [6 16] dBm when queried
% FTF limits setting actual: -12.000GHz to 12.000GHz
obj.FTFlim = 30; % in GHz

% Grid = 30; % in GHz

obj.Powers = 1e-3.*(ceil(10^6/10):1e-1:round(10*(10^16/10)))+0.1); % in W

obj.com_port = port;

end

methods (Access=protected)

% these can be called only by this instance and its children (classes
% that inherit from Laser object which inherits from ILaser)

function Initialize(obj) % public methods cause other people will call these

fprintf('
Initializing Pure Photonics PPCL300 ...
');

bdRate = 9600;

end
obj.ser.obj=serial(obj.com_port, 'BaudRate', bdRate);
frpen(obj.ser.obj);

% Setting Laser parameters
% FTFlim
obj.FTFlim = 1e-3*PurePhotonicsPPCL300.FTFR(obj); % in GHz
obj.FTFlim = 5; % Bypassed the maximum FTF limit because of
% laser power nonlinearity at higher values
fprintf('\nFTF Limit = %3.2f GHz', obj.FTFlim);

% Grid spacing
obj.Grid = 1e-1*PurePhotonicsPPCL300.Grid(obj); % in GHz
fprintf('\nGrid spacing = %3.2f GHz', obj.Grid);
if obj.Grid = obj стоим
    PurePhotonicsPPCL300.ResEna(obj, 1, 0); % make sure laser out is off
end

% WavelengthMin
f1 = PurePhotonicsPPCL300.LFH1(obj);
f2 = PurePhotonicsPPCL300.LFH2(obj);
f = f1 + f2/10;
obj.WavelengthMin = algorithms.FToLambda(1e9*(f-2*obj.FTFlim));
% f-2*obj.FTFlim is to create a margin around the extreme
% values so that fine tuning doesn’t step outside the range
fprintf('\nWavelengthMin = %4.3f nm', 1e9*obj.WavelengthMin);

% WavelengthMax
f1 = PurePhotonicsPPCL300.LFL1(obj);
f2 = PurePhotonicsPPCL300.LFL2(obj);
f = f1 + f2/10;
obj.WavelengthMax = algorithms.FToLambda(1e9*(f+2*obj.FTFlim));
% f+2*obj.FTFlim is to create a margin around the extreme
% values so that fine tuning doesn’t step outside the range
fprintf('\nWavelengthMax = %4.3f nm', 1e9*obj.WavelengthMax);

% Powers
plim(1)=PurePhotonicsPPCL300.OPS(100); % in dBm
plim(2)=PurePhotonicsPPCL300.OPSH(100); % in dBm
obj.Powers = ...
1e-3.*(ceil(10*(double(plim(1)))/10)):1e-1:round(10*10*(double(plim(2)))/10))
+0.1); % in W

% query laser until module is ready to be enabled
fprintf('\nWaiting for module to become enable-ready \');
MRDY = false; CE = false;
while "\MRDY&&CE
    [MRDY, CE, ] = ...
    PurePhotonicsPPCL300.NOP(obj);
    pause(0.1);
    fprintf(\n');
end
fprintf('\nTurning the Laser ON\');

% setting the laser to channel one
PurePhotonicsPPCL300.Channel(obj, 1);

% turning the laser ON
PurePhotonicsPPCL300.ResEna(obj, 1, 1)
fprintf('\nTurning Laser ON');

% Turning laser LowNoise state off in case it was on
CoBriteDX1.CBMX_set_port_state(obj.ser.obj.obj.Port, 1);
CoBriteDX1.wait_tuning(obj.ser.obj.obj.Port, obj.max_time);
CoBriteDX1.CBMX_set_port_dither(obj.ser.obj.obj.Port, 1);
pause(0.5);
assert (logical(CoBriteDX1.CBMX_query_port_dither(obj.ser.obj.obj.Port)) ....
function ShutDown(obj)
% switch the port of
tic;
fprintf('
Turning the Laser OFF');
P'urePhotonicsPPCL300.ResEna(obj, 1, 0);
close(obj.ser_obj); % close the laser port
delete(obj.ser_obj); % delete serial object
toc;
end

function wvRead = SetWavelength(obj, wavelength)
% calculating frequencies
% Target Frequency
wavelengthNM = wavelength * 1e9;
fTarget = 1e-9 * algorithms.LambdaToF(wavelength); % f [GHz]
rough = f - obj.FTlim;
fprintf('
Target: lambda = %4.3f nm and f = %2.3f GHz', ...
1e9*wavelength, fTarget);

% Current Frequency
f01 = P'urePhotonicsPPCL300.LF1(obj); % GHz
f02 = 1e-1 * PurePhotonicsPPCL300.LF2(obj);
f0 = f01 + 1e3 + f02;
fprintf('
Current LF: lambda = %4.3f nm and f = %2.3f GHz', ...
algorithms.FTo Lambda(f0), f0);

F = 1e-3 * PurePhotonicsPPCL300.FTF(obj); % GHz
f0fcf = PurePhotonicsPPCL300.FCF1(obj);
f02fcf = 1e-1 * PurePhotonicsPPCL300.FCF2(obj);
ofOpticalMode = f0fcf + 1e3 + f02fcf;
f0fcf = ofOpticalMode + F;
fprintf('
Current FCF: lambda = %4.3f nm and f = %2.3f GHz (FTF: %2.3f GHz)', ...
algorithms.FToLambda(f0fcf), ofOpticalMode, F);

f0fcfRounded = round(f0fcf, 1);
fprintf('
Current FCF (rounded): lambda = %4.3f nm and f = %2.3f GHz', ...
ifOpticalMode, 0fcfRounded);

f0rough = f0fcf + 1e3 + f02;

f0 = f0rough + F;

deltaFTF = f - f0;

deltaf = frough - f0rough;
deltaFGrid = ceil(deltaf/obj.Grid)*obj.Grid;

tof = fOffset = deltaf - deltaFGrid;

assert(abs(tof) <= obj.FTlim, 'Calculated FTF exceeds limits');

froughGrid = frough + deltaFGrid; % new central frequency of the optical mode

fTTF = f - froughGrid; % this is the calculated fTT to be as big as possible

fprintf('
Initial: lambda = %4.3f nm and f = %2.3f GHz (FTF = %6.3f GHz)', ...
algorithms.FToLambda(f0), f0, F);

fprintf('
Delta f = %6.3f GHz', tof);

if %CleanJump frequency tuning

f0new2 = 1e4*mod(f*1e-3, 1); % 0.1GHz part

f0new1 = 1e-4*(f*10 - f0new2); % THz part

PurePhotonicsPPCL300.CleanJumpGHz(obj, f0new2);

PurePhotonicsPPCL300.CleanJumpTHz(obj, f0new1);

PurePhotonicsPPCL300.CleanJumpEnable(obj);
deltaf = fTarget - fOpticalMode;
fprintf('\n\nndlambda = %4.3f nm and df = %2.3f GHz' , ...  
algorithm. FToLambda(fTarget) - algorithms. FToLambda(fOpticalMode) . . .

deltaf);
% setting the frequency
if deltaf == 0;
    fprintf(\nWavelength unchanged . . .') return;
elseif (abs(deltaf) < obj. FTFlim)
    fprintf(\nFine tuning . . .');
    PurePhotonicsPPCL300. FTF(obj, deltaf*1e3);
else (abs(deltaf) > obj. FTFlim)
    fprintf(\nCoarse Tuning . . .');
    [f0Target, fFTFTarget] = . . .
    PurePhotonicsPPCL300. matchToGridFTF(fTarget, . . .
    algorithms. LambdaToF(1e9*obj. WavelengthMax), . . .
    algorithms. LambdaToF(1e9*obj. WavelengthMin), . . .
    obj. Grid, obj. FTFlim, 0);

    % fTarget = f;
    % frough = fTarget - obj. FTFlim;
    deltaRough = frough - f0rough;
    deltaFTFGrid = ceil(deltaRough / obj. Grid) * obj. Grid;
    fTargetGrid = f0 + deltaFTFGrid;

    F = 1e-3 * PurePhotonicsPPCL300. FTF(obj); % FTF in GHz
    f01fcf = PurePhotonicsPPCL300. FCF1(obj);  
    f02fcf = 1e-1 * PurePhotonicsPPCL300. FCF2(obj); 
    fOpticalMode = f01fcf + f02fcf;  
    lambda0fcf = algorithms. FToLambda(f0fcf);  
    fprintf('\nCurrent CFC: lambda = %4.3f nm and f = 2.3f GHz' . . .
    algorithms. FToLambda(f0fcf), f0fcf);
    wvRead = lambda0fcf;
    f0fcfRounded = round(f0fcf, 1);
    fprintf('\nCurrent CFC (rounded): lambda = %4.3f nm and f = 2.3f GHz' . . .
    algorithms. FToLambda(f0fcfRounded), f0fcfRounded);

    % Verify the frequency
    f01 = PurePhotonicsPPCL300. LF1(obj);
    f02 = 1e-1 * PurePhotonicsPPCL300. LF2(obj);
    f0 = f01 + f02;

    fprintf('\nCurrent LF: lambda = %4.3f nm and f = 2.3f GHz' . . .
    algorithms. FToLambda(f0), f0);
    F = 1e-3 * PurePhotonicsPPCL300. FTF(obj); % FTF in GHz
    f01fcf = PurePhotonicsPPCL300. FCF1(obj); 
    f02fcf = 1e-1 * PurePhotonicsPPCL300. FCF2(obj); 
    fOpticalMode = f01fcf + f02fcf;  
    lambda0fcf = algorithms. FToLambda(f0fcf);  
    fprintf('\nCurrent CFC: lambda = %4.3f nm and f = 2.3f GHz' . . .
    algorithms. FToLambda(f0fcf), f0fcf);
    wvRead = lambda0fcf;
    f0fcfRounded = round(f0fcf, 1);
    fprintf('\nCurrent CFC (rounded) : lambda = %4.3f nm and f = 2.3f GHz' . . .
    algorithms. FToLambda(f0fcfRounded), f0fcfRounded);
    fprintf('\nWavelength error: %3.3f pm', 1e3 * abs(wavelengthNM - lambda0fcf));
    while abs(wavelengthNM - lambda0fcf) > obj. wvError
        fprintf('Port %d-%d-%d: Error! Wavelength error exceeded', obj. Port);
        obj.SetWaveLength(wavelength);
    end
end

function wavelength = GetWavelength(obj)
    f01 = PurePhotonicsPPCL300. LF1(obj);
    f02 = PurePhotonicsPPCL300. LF2(obj);
f0 = f01*1e3 + f02/10;
df0 = PurePhotonicsPPCL300.FTF(obj);

wavelength = algorithms.FToLambda(1e9*f0);
printf(‘\nCurrent f: %2.3f GHz (FTF = %2.3f GHz), f0, df0’);
printf(‘\nCurrent wavelength is %4.3f nm’, 1e9*wavelength);

f01 = PurePhotonicsPPCL300.LF1(obj);
f02 = PurePhotonicsPPCL300.LF2(obj);
f0 = f01*1e3 + f02/10;
df0 = PurePhotonicsPPCL300.FTF(obj);

wavelength = algorithms.FToLambda(1e9*f0);
printf(‘\nCurrent f: %2.3f GHz (FTF = %2.3f GHz), f0, df0’);
printf(‘\nCurrent wavelength is %4.3f nm’, 1e9*wavelength);

end

function SetPower(obj, power)

printf(‘\nSetting Power . . . ’);

power_mW = 1e3*power;

powerDBm = 10*log10(power_mW);

lim = [obj.Powers(1) obj.Powers(end)];
assert(powerDBm>lim(1) || powerDBm<lim(2), ‘Requested Power out of bounds’);

PurePhotonicsPPCL300.ResEna(obj, 1, 0); %laser software OFF
PurePhotonicsPPCL300.PWR(obj, 100*powerDBm);

PurePhotonicsPPCL300.ResEna(obj, 1, 1); %laser software ON
pwrRead=1e−2*PurePhotonicsPPCL300.PWR(obj);

printf(‘\nPower setting error = %2.6f mW’,
abs(10^6*(powerDBm/10)−10^(pwrRead/10)));
assert(abs(10^6*(powerDBm/10)−10^(pwrRead/10))<1e3*obj.pwrError || pwrRead==−100, . . .
‘Port %d-%d-%d: Error! Power setting/reading not working’,obj.Port);

printf(‘\nPower set to %2.3f mW’, 10^(pwrRead/10));
end

function power = GetPower(obj)
powerResponse = PurePhotonicsPPCL300.PWR(obj);
powerDBm = powerResponse/100;

power = 1e−3*10^(powerDBm/10);

printf(‘\nCurrent power is %4.3f mW’, 1e3*power);
end

function ok = SetLowNoise(obj, sw)

%turn dither off
if sw==true

printf(‘\nTurning Low Noise Mode on . . . ’);

x = inputdlg(‘Timer setup: set Low-noise mode timeout [mins]:’, ‘Timeout’, 1, {’5’});
if ~isempty(x);
    obj.timer_handle = GUI.TimerGUI(str2double(x{:}));
end
else

printf(‘\nTurning Low Noise Mode off . . . ’);
end

PurePhotonicsPPCL300.CleanMode(obj, 2*sw);

modeRead = PurePhotonicsPPCL300.CleanMode(obj);
assert(modeRead==2*sw . . .
‘Dither status setting/reading error’);
ok = true;
if sw==true

printf(‘ON’);
else

printf(‘OFF’);
if isvalid(obj.timer_handle)
    delete(obj.timer_handle);
end

end
classdef LaserFactory < interfaces.ILaserFactory
    % LaserFactory Creates laser objects for PurePhotonicsPPCL300 laser
    % Detailed explanation goes here
    properties (Constant)
      Name = 'PurePhotonicsPPCL300';
    end
    methods
      function laser = Create (~)
        delete (instrfind); % first clean up before doing anything
        ports = PurePhotonicsPPCL300.getAvailableComPort();
        [idx, ok] = listdlg ('PromptString', 'Select COM Port', . . .
           'SelectionMode', 'single', 'ListSize', [100 120], . . .
           'ListString', ports);
        if ~ok
          idx = 1;
          laser = [];
          return;
        end
        laser = PurePhotonicsPPCL300.Laser(ports(idx));
      end
    end
end

A.7.2 Functions

function checksum = BIP4 (data0, data1, data2, data3);
    % BIP-4 checksum computed over a 32 bit word with the leading 4 bits
    % prepended to the 28 bit packet and set to zero.
    BIP8 = xor(hexToBinaryVector(data3, 8) ....
      hexToBinaryVector(data2, 8));
    BIP8 = xor(BIP8, hexToBinaryVector(data1, 8));
    BIP8 = xor(BIP8, hexToBinaryVector(data0, 8));
    checksum = xor(BIP8(1:4), BIP8(5:8));

function channelResponse = Channel(obj, ch)
    % N. Stan
    % Channel sets the module's channel.
    if nargin == 1
        WR = 0; ch = 0;
    elseif nargin == 2
        WR = 1;
    end
commandRegister = '30'; %Register address for the command
data1 = commandRegister;
data23 = dec2hex(ch, 4); %first data byte in hex
data2 = data23(1:2); %second data byte in hex
data3 = data23(3:4); %third data byte in hex
LstRsp = 0; %Bit set to logic 0 when the checksum is consistent.
%Bit set to logic 1 forces module to resend last valid packet.
%Used when the checksum is inconsistent.
Bits26_25 = [0 0]; %set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);
    %Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);
    %Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)]
    %convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);
    %Analyze response
    %Command specific bits
    %Channel number
    channelResponse = hex2dec(data);
    %Command independent bits
    %Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf('
Channel - Communication Error');
    LstRsp = 1;
end
%Status Bytes
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
%0x00 - OK flag . Normal return status
%0x01 - XE flag . (execution error)
%0x02 - AEA flag . (Automatic extended addressing result being returned or ready to write)
%0x03 - CP flag . Command not complete, pending
%This needs to be changed . the status will most of the time be 3, because it's queried very
soon after issuing the command

%Final wait for the command to execute
fclose('SET!
');
function fResponse = CleanJumpEnable(obj)

% N. Stan
% CleanJumpEnable activates Clean Jump

x=1;

commandRegister = 'ED';
% Register address for the command

data1 = commandRegister;
data23 = dec2hex(uint16(x), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

WR = 1;

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default

CE = 1; Status = 1;
while CE || Status

data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);

% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj.sendLine);

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
if size(rxdata_hex)’=[4 2]
rxdata_hex
end
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
% Command specific bits
% Channel number
fResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
fprintf('\nCleanJumpGHz – Communication Error');
LstRsp = 1; continue;
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 − OK flag. Normal return status
% 0x01 − XE flag. (execution error)
% 0x02 − AEA flag. (Automatic extended addressing result being returned or ready to write)
% 0x03 − CP flag. Command not complete, pending
if Status == 1
    fprintf('
CleanJumpEnable − Execution Error − retrying . . .'); continue;
end

function fResponse = CleanJumpGHz(obj, f)

% N. Stan
% CleanJumpGHz load the GHz part of the next frequency, with x in units of 0.1GHz
% 0.1*f − GHz part of the frequency (unsigned short)
if nargin == 1
    WR = 0; f = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = 'EB'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(uint16(f), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || Status
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4 (data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    if size(rxdata_hex) == [4 2]
        rxdata_hex
    end

../+PurePhotonicsPPCL300/CleanJumpGHz.m
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)]; % convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
% Command specific bits
% Channel number
fResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
CleanJumpGHz - Communication Error');
    LstRsp = 1; continue;
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending
if Status == 1
    fprintf('
CleanJumpGHz - Execution Error - retrying ...'); continue;
end

function fResponse = CleanJumpTHz(obj, f)
% N. Stan
% CleanJumpTHz load the THz part of the next frequency, with x in units of THz
% f - THz part of the frequency (unsigned short)
if nargin == 1
    WR = 0; f = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = 'EA'; %Register address for the command
data1 = commandRegister;
data23 = dec2hex(uint16(f), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || Status

data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);

% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
if size(rxdata_hex) == [4 2]
    rxdata_hex
end

    data = [rxdata_hex(3:1:2) rxdata_hex(4:1:2)];
% convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
% Command specific bits
% Channel number
fResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
CleanJumpTHz - Communication Error');
    LstRsp = 1; continue;
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations, 
% if any, are still pending. Each operation that becomes pending 
% is assigned one of these four bit positions. 
% The module can be periodically polled (by reading the NOP register) 
% to determine which operations have completed. 
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 – OK flag, Normal return status
% 0x01 – XE flag, (execution error)
% 0x02 – AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 – CP flag, Command not complete, pending
if Status == 1
    fprintf('
CleanJumpTHz – Execution Error – retrying ...'); continue;
end

%..\PurePhotonicsPPCL300/CleanMode.m

function modeResponse = CleanMode(obj, mode)
% N. Stan
% Register 0x90: switch between the different low-noise modes,
% with 0 the standard dither-mode, 1 the no-dither mode and 2 the whisper-mode.

% The standard operating mode for the laser is the dither-mode.
% In this mode all the control loops are running and creating noise
% (especially in the 1-10000Hz range). By disabling these control-loops.
% either completely or partially, a lower noise behavior can be achieved.
% In the no-dither mode, the FM dither is disabled. Most other control-loops remain enabled.
% This results in the removal of the 888Hz tone (and its overtones).
% but does not address the noise below 100Hz. Note that this mode is not available
% on the most recent versions of the firmware (8.0.9 and 8.6.0) as we now recommend
% the whisper mode for all applications.
% In the whisper mode, essentially all control loops are disabled, to the extent possible.
% This significantly reduced the AM and FM noise in the below 100Hz range.
% If in any way possible, we recommend to once in a while switch back to
% the dither mode for the laser to relock. Such a switch-back could be done
% in less than 10 seconds.

if nargin == 1
    WR = 0; mode = 0;
else if nargin == 2
    WR = 1;
    assert (mode>=0&&mode<2, 'Bad parameter passed to CleanMode')
end

commandRegister = '90'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(mode, 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);
data0_b = [data0_b(1:end-1) WR]

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0_l = binaryVectorToHex(BIP4);

    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdataline = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdataline = dec2hex(rxdataline);
data = [rxdataline(3:1) rxdataline(4:2)];
% convert response to binary
    rxdataline = hexToBinaryVector(dec2hex(rxdataline), 8);

    % Analyze response
    % Command specific bits
    % Mode
    modeResponse = hex2dec(data);

    % Command independent bits
% Communication Error
CE = rxdata_bin(1,5);
if CE
fprintf('
CleanMode - Communication Error');
LstRsp = 1;
end
end

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1,7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

% maybe add wait for pending operation to finish code

../+PurePhotonicsPPCL300/convertHexToASCII.m

function response = convertHexToASCII(wordHex)
% converts the hex values given into an array of ascii characters and displays them
response = char(hex2dec(wordHex));
if size(response,1)>size(response,2)
response = response';
end
disp(response');

../+PurePhotonicsPPCL300/DitherA.m

function [ampResponse, CE, Status] = DitherA(obj, amp)
% N. Stan
% Dither Amplitude
% DitherA is an unsigned short integer encoded as
% the AM p-p amplitude deviation as 10*percentage of the optical power.

if nargin == 1
WR = 0; amp = 0;
elself nargin == 2
WR = 1;
end
commandRegister = '5C'; %Register address for the command
data1 = commandRegister;
%conversion of rate into a signed hex, 16 bits long
data23 = dec2hex(amp, 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE

% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj.obj_ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)]
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
% Command specific bits
ampResponse = hex2dec(data);
% Command independent bits
CE = rxdata_bin(1, 5);
if CE
    fprintf('nDitherA = Communication Error');
end
LstRsp = 1;

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

function [WFResponse, DDEResponse, CE, Status] = DitherE(obj, WF, DDE)
data3 = binaryVectorToHex([0 0 decimalToBinaryVector(WF, 2) 0 0 DDE 0]); % second data byte in hex
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj,sendlne);
    % % Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    data = [rxdata_hex(3:1:2) rxdata_hex(4:1:2)]
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);
    % Analyze response
    % Command specific bits
    fResponse = hex2dec(data);
    % Channel number
    WFResponse = binaryVectorToDecimal(rxdata_bin(3:4));
    DDEResponse = binaryVectorToDecimal(rxdata_bin(7));
    % Command independent bits
    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf('
DitherE - Communication Error');
        LstRsp = 1;
    end
end % this needs to be changed, the status will most of the time be 3, because it's queried very soon after issuing the command
% Final wait for the command to execute
if WR
    fprintf('
Dither ...');
    PurePhotonicsPPCL300.wait_operation(obj)
    switch DDE
        case true; fprintf('ON!
');
        case false; fprintf('OFF!
');
end
function [freqResponse, CE, Status] = DitherF(obj, freq)
% N. Stan
% Dither Frequency deviation
% DitherA is an unsigned short integer encoded as
% FM p-p frequency deviation as GHz\times 10.
if nargin == 1
    WR = 0; freq = 0;
elsif nargin == 2
    WR = 1;
end
commandRegister = '5B'; % Register address for the command
data1 = commandRegister;
% Conversion of rate into a signed hex, 16 bits long
data23 = dec2hex(freq, 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);
% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fprintf(obj.ser_obj, sendLine);
% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);
% Analyze response
% Command specific bits
% Rate
freqResponse = hex2dec(data);
% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf(’\nDitherF – Communication Error’);
    LstRsp = 1;
end
end
function [rateResponse, CE, Status] = DitherR(obj, rate)

% N. Stan
% Dither Rate
% DitherR is an unsigned integer specifying the dither rate as kHz.
% Note that DitherE is used to set the waveform for this frequency.

if nargin == 1
    WR = 0; rate = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = '5A'; % Register address for the command
data1 = commandRegister;
% Conversion of rate into a signed hex, 16 bits long
data23 = dec2hex(rate, 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);

% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

pause(0.25);
rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
% Command specific bits
rateResponse = hex2dec(data);
% Command independent bits
function fResponse = FCF1(obj, f)

% N. Stan
% The FCF1 and FCF2 registers provide a way to configure the frequency of channel 1.
% f – THz part of the frequency (unsigned short)
% This value can only be changed when the output is disabled.

if nargin == 1
    WR = 0; f = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = '35'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(uint16(f), 4);
data2 = data23(1:2); % First data byte in hex
data3 = data23(3:4); % Second data byte in hex

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % Set to zero by default

CE = 1; Status = 1;
while CE || Status
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
    data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
end

end

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 – OK flag. Normal return status
% 0x01 – XE flag. (execution error)
% 0x02 – AEA flag. (Automatic extended addressing result being returned or ready to write)
% 0x03 – CP flag. Command not complete, pending

% this needs to be changed, the status will most of the time be 3, because it’s queried very
soon after issuing the command

% Final wait for the command to execute
if WR
    fprintf('\nRate ...
');
    PurePhotonicsPPCL300.wait_operation(obj)
    fprintf('\nSET!
');
end

../+PurePhotonicsPPCL300/FCF1.m
sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

%%% Obtain response from the module
pause(0.25);
if obj.ser_obj.BytesAvailable == 0
disp('zero Bytes Available'); LstRsp = 1; continue;
end
rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
if size(rxdata_hex)"=[4 2]
rxdata_hex
end
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);
%%% Analyze response
%%% Command specific bits
% Channel number
fResponse = hex2dec(data);
%%% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
fprintf('
FCF1 - Communication Error - retrying ...'); continue;
end
%%% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.
% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending
if Status == 1
fprintf('
FCF1 - Execution Error - retrying ...'); continue;
end

../+PurePhotonicsPPCL300/FCF2.m

function fResponse = FCF2(obj, f)
% N. Stan
% The FCF1 and FCF2 registers provide a way to configure the frequency of channel 1.
% f = GHz*10 part of the frequency (unsigned short)
% This value can only be changed when the output is disabled.
if nargin == 1
WR = 0; f = 0;
elseif nargin == 2
WR = 1;
commandRegister = '36'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(uint16(f), 4);
data2 = data23(1:2); % first data byte in hex
ndata3 = data23(3:4); % second data byte in hex
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || Status
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    if obj.ser_obj.BytesAvailable == 0
        disp('zero Bytes Available'); LstRsp = 1; continue;
    end
    rxdata_dec = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    if size(rxdata_hex) == [4 2]
        rxdata_hex
    end
    data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Analyze response

% Command specific bits
% Channel number
fResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
\nFCF2 - Communication Error\n');
    LstRsp = 1; continue;
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations.
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 = OK flag, Normal return status
function fResponse = FTF(obj, f)

% N. Stan
% The FTF register provides fine tune adjustment of the laser's wavelength
% from the set channel. The adjustment is applied to all channels uniformly.
% Returned value is in MHz, signed short.

if nargin == 1
    WR = 0; f = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = '62'; % Register address for the command
data1 = commandRegister;
% Conversion of power into a signed hex, 16 bits long
data23 = dec2hex(typecast(int16(f), 'uint16'), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex

LstRsp = 0; % Bits set to logic 0 when the checksum is consistent.
% Bits set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || (Status==1)
    data0_b = [ [0 0 0 0] LstRsp Bits26_25 WR ];
data0 = binaryVectorToHex(data0_b);
% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj,sendLine);

pause(0.25);
if obj.ser_obj.BytesAvailable~=0
    rxdData_dec = fread(obj.ser_obj,obj.ser_obj.BytesAvailable);
    rxdData_hex = dec2hex(rxdData_dec);
    if size(rxdData_hex)=[4 2]
        rxdData_hex
    end
    data = [rxdData_hex(3,1:2) rxdData_hex(4,1:2)];
    % convert response to binary
    rxdData_bin = hexToBinaryVector(dec2hex(rxdData_dec), 8);
else
    fprintf('
FTF - Zero Size Response');
    CE=1; continue;
end
function ftfR = FTFR(obj)

% N. Stan
% Query the minimum/maximum fine tune frequency range capability of the
% module in MHz
% the returned value ftfR means the range is (−ftfR, ftfR) in MHz
commandRegister = '4F'; %Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex

WR = 0; %Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || Status
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);
end
% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
if obj.ser_obj.BytesAvailable == 0
disp('zero Bytes Available'); LstRsp = 1; continue;
end
assert(obj.ser_obj.BytesAvailable > 0, 'No response from the laser');
rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3:1:2) rxdata_hex(4:1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
% Command specific bits
% Channel number
ftfR = hex2dec(data);

% Command independent bits
% Communication Error
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
FTFR - Communication Error');
    LstRsp = 1;
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations.
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 = XE flag, (execution error)
% 0x02 = AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 = CP flag, Command not complete, pending
if Status == 1
    fprintf('
FCF1 - Execution Error - retrying ...'); continue;
end


../+PurePhotonicsPPCL300/getAvailableComPort.m

function lCOM_Port = getAvailableComPort()
% function lCOM_Port = getAvailableComPort()
% Return a Cell Array of COM port names available on your computer
try
function [gResponse, CE, pendingOperation, Status] = Grid(obj, g)

% N. Stan
% Grid sets the modules grid spacing for the channel to frequency mapping.
% g = grid spacing in GHz*10 signed short
% This value can only be changed when the output is disabled.

if nargin == 1
    WR = 0; g = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = '34'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(typecast(int16(g), 'uint16'), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1; Status = 1;
while CE || Status
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
    data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    if CE || Status
    end

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj, BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3:4) rxdata_hex(4:1)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 – OK flag, Normal return status
% 0x01 – XE flag, (execution error)
% 0x02 – AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 – CP flag, Command not complete, pending

% Command specific bits
% Channel number
gResponse = double(typecast(uint16(hex2dec(data)), 'int16'));

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
Grid – Communication Error
');
    LstRsp = 1;
end
if Status == 1
    fprintf('
Grid – Execution Error – retrying ...
'); continue;
end
end

function freqResponse = LF1(obj)
% N. Stan
% Reads the frequency of the current channel Unisgned short [THz]

commandRegister = '40'; % Register address for the command
data1 = commandRegister;
data2 = '00'; % First data byte in hex
data3 = '00'; % Second data byte in hex

WR = 0; % Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % Set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);
% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
end
sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
if obj.ser_obj.BytesAvailable == 0
    disp('zero Bytes Available'); LstRsp = 1; continue;
end
rxdata_dec = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);
rxdump = dec2hex(rxdump);
data = [rxdump(3, 1:2) rxdump(4, 1:2)];
% Convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdump), 8);

% Analyze response
% Command specific bits
% Channel number
freqResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
LF1 - Communication Error');
    LstRsp = 1;
end
% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.
% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
function freqResponse = LF2(obj)

% N. Stan
% Reads the GHz part and first decimal of the frequency of the current channel Unisigned short [GHz]

commandRegister = '41'; %Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex

WR = 0; %Command is Read-only

LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default

CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    pause(0.25);

    if obj.ser_obj.BytesAvailable == 0
        disp('zero Bytes Available'); LstRsp = 1; continue;
    end
    rxdata_dec = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    data = [rxdata_hex(3:1:2) rxdata_hex(4:1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec) , 8);

    % Obtain response from the module

    % Command specific bits
    % Channel number
    freqResponse = hex2dec(data);

    % Command independent bits
    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf('
LF2 - Communication Error');
        LstRsp = 1;
    end
end
% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));

% A series of eight flag bits indicating which operations, if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.
%
% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));

% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

../+PurePhotonicsPPCL300/LFH1.m

function freqResponse = LFH1(obj)

% N. Stan
% Returns the THz part of the max frequency

commandRegister = '54'; %Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex
WR = 0; %Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);
    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
    data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);
    % Obtain response from the module
    pause(0.25);
    freqResponse = hex2dec(data);
function freqResponse = LFH2(obj)

%% Command independent bits

%% Communication Error
CE = rxdataln(1,5);
if CE
    fprintf(’nLFH1 – Communication Error’);
LstRsp = 1;
end
end

%% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdataln(3,1:8));
% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

%% Status Bits
Status = binaryVectorToDecimal(rxdataln(1,7:8));
% 0x00 – OK flag, Normal return status
% 0x01 – XE flag, (execution error)
% 0x02 – AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 – CP flag, Command not complete, pending

../+PurePhotonicsPPCL300/LFH2.m

commandRegister = ’55’; %Register address for the command
data1 = commandRegister;
data2 = ’00’; % first data byte in hex
data3 = ’00’; % second data byte in hex

WR = 0; %Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj,sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdatalc = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdatalc_hex = dec2hex(rxdatalc);
data = [rxdatalc_hex(3,1:2) rxdatalc_hex(4,1:2)];
    % convert response to binary
    rxdatalc_bin = hexToBinaryVector(dec2hex(rxdatalc), 8);
% Analyze response
% Command specific bits
% Channel number
freqResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('\nLFH2 - Communication Error');
end
LstRsp = 1;
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations.
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.
% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

../+PurePhotonicsPPCL300/LFL1.m

function freqResponse = LFL1(obj)
% N. Stan
% Returns the THz part of the min frequency
commandRegister = '52'; % Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex
WR = 0; % Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
    sendLine = uint8(hex2dec([data0 ; data1 ; data2 ; data3]));
    fwrite(obj_ser_obj, sendLine);
end
function freqResponse = LFL2(obj)
% N. Stan
% Returns the GHz part with the first decimal of the min frequency

commandRegister = '52'; %Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex

WR = 0; %Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response

% Command specific bits
% Channel number
freqResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
fprintf('
LFL1 - Communication Error');
LstRsp = 1;
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations. If any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register) to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

../+PurePhotonicsPPCL300/LFL2.m
data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);

% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
rxdatal_dec = fread(obj.ser_obj.obj.ser_obj.BytesAvailable);
rxdatal_hex = dec2hex(rxdatal_dec);
data = [rxdatal_hex(3:1:2) rxdatal_hex(4:1:2)];
% convert response to binary
rxdatal_bin = hexToBinaryVector(dec2hex(rxdatal_dec), 8);

% Analyze response
% Command specific bits
% Channel number
freqResponse = hex2dec(data);

% Command independent bits
% Communication Error
CE = rxdatal_bin(1, 5);
if CE
fprintf('
LFL2 - Communication Error');
LstRsp = 1;
end
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdatal_bin(3, 1:8));
% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdatal_bin(1, 7:8));
% 0x00 - OK flag, Normal return status
% 0x01 - XE flag, (execution error)
% 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag, Command not complete, pending

../+PurePhotonicsPPCL300/LGrid.m

function [IGridResponse, CE, pendingOperation, Status] = LGrid(obj)
% N. Stan
% Reads the GHz part and first decimal of the minimum grid spacing capability of the module
commandRegister = '56'; % Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex
WR = 0; % Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
  data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
  data0 = binaryVectorToHex(data0_b);
  % Calculating BIP4 checksum
  BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
  data0(1) = binaryVectorToHex(BIP4);
  sendLine = uint8(hex2dec([data0; data1; data2; data3]));
  fwrite(obj.ser.obj, sendLine);
  % Obtain response from the module
  pause(0.25);
  rxdatal_dec = fread(obj.ser.obj.obj.ser.obj.BytesAvailable);
  rxdatal_hex = dec2hex(rxdatal_dec);
  data = [rxdatal_hex(3:1:2) rxdatal_hex(4:1:2)];
  % convert response to binary
  rxdatal_bin = hexToBinaryVector(dec2hex(rxdatal_dec), 8);
  % Analyze response
  % Command specific bits
  % Channel number
  lGridResponse = hex2dec(data);
  % Command independent bits
  % Communication Error
  CE = rxdatal_bin(1, 5);
  if CE
    fprintf('
LGrid - Communication Error')
    LstRsp = 1;
  end
end
% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdatal_bin(3, 1:8));
% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.
% Status Bits
Status = binaryVectorToDecimal(rxdatal_bin(1, 7:8));
% 0x00 – OK flag, Normal return status
% 0x01 – XE flag, (execution error)
% 0x02 – AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 – CP flag, Command not complete, pending
% matches the target frequency \( f \) to the frequency grid of the laser (\( f_{\text{Min}}: d f_{\text{Grid}}: f_{\text{Max}} \))
% making the FTF minimum if frequencies increment (INCDEC=1) or
% making the FTF maximum if frequencies decrement (INCDEC=0)

if \( f = f_{\text{Min}} \& \& f = f_{\text{Max}} \):
    fGrid = \( f_{\text{Min}}: d f_{\text{Grid}}: f_{\text{Max}} \);
    if INCDEC
        ind = find (fGrid <= f + fFTFLim, 1, 'last');
    else
        ind = find (fGrid >= f - fFTFLim, 1);
    end
    f0 = fGrid(ind);
    fFTF = f - f0;
else
    f0 = f; fFTF = 0;
end

../+PurePhotonicsPPCL300/NoDriftMode.m

function EnaDisResponse = NoDriftMode(obj, EnaDis)

% N. Stan
% Register 0x91: enable (1) or disable (0) the no-drift mode
% In the standard telecom mode the laser has many control loops running to
% ensure stability over the long run. For sensor applications we recommend the
% whispermode, where these control−loops have been mostly disabled.
% This does result in drift over time, and the laser needs to be put back
% into standard mode regularly for a short period of time (10secs) for relocking.
% For critical applications, where long term stability is important
% (i.e. the laser needs to be locked), but where the drift needs to be limited,
% the no−drift mode has optimized the control loops to have a factor 10x less noise
% (some limits apply for the temperature ramp−rate etc.). The no−drift calibration
% involves the characterization of over temperature variation and compensation
% of the measured change rate based on ambient temperature.
% The unit starts up in the no−drift mode.

if nargin == 1
    WR = 0; EnaDis = 0;
else if nargin == 2
    WR = 1;
    assert (EnaDis==0|EnaDis==1, 'Bad parameter passed to CleanMode')
end

commandRegister = '91'; %Register address for the command
data1 = commandRegister;
data23 = dec2hex(EnaDis, 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26.25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [[0 0 0 0] LstRsp Bits26.25 WR];
    data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
    data0(1) = binaryVectorToHex(BIP4);
sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj_ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj_ser_obj, obj_ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response

% Command specific bits

% Mode
EnaDisResponse = hex2dec(data);

% Command independent bits

% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('\nNoDriftMode - Communication Error');
    LstRsp = 1;
end

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 – OK flag, Normal return status
% 0x01 – XE flag, ( execution error )
% 0x02 – AEA flag, ( Automatic extended addressing result being returned or ready to write )
% 0x03 – CP flag, Command not complete, pending

../+PurePhotonicsPPCL300/NOP.m
% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4 (data0, data1, data2, data3);
data0(1) = binaryVectorToDecr(BIP4);

% sending the formatted command to the laser
sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj_ser_obj, sendLine);
LstRsp = 0;

% Obtain response from the module
pause(0.25);
if obj_ser_obj.BytesAvailable == 0
    disp('zero Bytes Available'); LstRsp = 1; continue;
end
rxdata_dec = fread(obj_ser_obj, obj_ser_obj.BytesAvailable);

% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

if size(rxdata_bin) == [4:8]
    rxdata_bin
end

% Analyze response

% Command independent bits

% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
NOP - Communication Error');
    LstRsp = 1; continue;
end

% Command specific bits

% Pending Operation Flags
pendingOperation = binaryVectorToDecr(rxdata_bin(3, 1:8));

% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Module Ready Bit
try
    MRDY = rxdata_bin(4, 4);
catch
    disp('');
end

% MRDY - Module Ready
% When 1 indicates that the module is ready for its output
% to be enabled
% When 0 indicates that the module is not ready for its output
% to be enabled.

% Error Field Bits
errorField = binaryVectorToDecr(rxdata_bin(4, 5:8));
% 0x00 OK - Ok, no errors
% 0x01 RNI - The addressed register is not implemented
% 0x02 RNW - Register not write-able; register cannot be written (read only)
% 0x03 RVE - Register value range error; writing register contents causes value range error;
%    contents unchanged
% 0x04 CIP - Command ignored due to pending operation
% 0x05 CII - Command ignored while module is initializing, warming up, or contains an invalid
%    configuration.
% 0x06 ERE – Extended address range error (address invalid)
% 0x07 ERO – Extended address is read only
% 0x08 EXF – Execution general failure
% 0x09 CIE – Command ignored while module’s optical output is enabled (carrying traffic)
% 0x0A IVC – Invalid configuration, command ignored
% 0x0B−0x0E——Reserved for future expansion
% 0x0F VSE Vendor specific error (see vendor specific documentation for more information)

end

% Status Bits

Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));

% 0x00 – OK flag. Normal return status
% 0x01 – XE flag. (execution error)
% 0x02 – AEA flag. (Automatic extended addressing result being returned or ready to write)
% 0x03 – CP flag. Command not complete, pending

../+PurePhotonicsPPCL300/OOP.m

function [powerResponse, CE, pendingOperation, Status] = OOP(obj)

% N. Stan
% Reads the external optical power estimate of the module as dBm+100 signed
% short

commandRegister = '42'; % Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex

WR = 0; % Command is Read–only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR;
data0 = binaryVectorToHex(data0_b);

    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

    sendLine = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];

    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Analyze response

% Command specific bits
% Channel number
powerResponse = double(typecast(uint16(hex2dec(data)), 'int16'));
function powerResponse = OPSH(obj)

% N. Stan
% Reads the maximum optical capability of the module as dBm*100 signed
% short

commandRegister = '51'; %Register address for the command
data1 = commandRegister;
data2 = '00'; % first data byte in hex
data3 = '00'; % second data byte in hex

WR = 0; %Command is Read-only
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.

Bits26_25 = [0 0]; % set to zero by default

CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR];
data0 = binaryVectorToHex(data0_b);

% Calculating BI6 checksum
BI6 = PurePhotonicsPPCL300.BI6(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BI6);

sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
rxdataldec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdatalhex = dec2hex(rxdataldec);
data = [rxdatalhex(3,1:2) rxdatalhex(4,1:2)];
% convert response to binary
rxdatalbin = hexToBinaryVector(dec2hex(rxdataldec), 8);
% Analyze response
% Command specific bits
% Channel number
powerResponse = double(typecast(uint16(hex2dec(data)), 'int16'));
%
% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
OPSH - Communication Error
');
    LstRsp = 1;
end
%
% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations are still pending. Each operation that becomes pending is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register) to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.
%
% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 - OK flag. Normal return status
% 0x01 - XE flag. (execution error)
% 0x02 - AEA flag. (Automatic extended addressing result being returned or ready to write)
% 0x03 - CP flag. Command not complete, pending
function powerResponse = PWR(obj, power)

% N. Stan
% Sets or reads the power level as dBm+100
% power - the power in (dBm+100) to be set, signed short
if nargin == 1
    WR = 0; power = 0;
elseif nargin == 2
    WR = 1;
end

commandRegister = '31'; % Register address for the command
data1 = commandRegister;
data23 = dec2hex(typecast(int16(power), 'uint16'), 4);
data2 = data23(1:2); % first data byte in hex
data3 = data23(3:4); % second data byte in hex
LstRsp = 0; % Bit set to logic 0 when the checksum is consistent.
% Bit set to logic 1 forces module to resend last valid packet.
% Used when the checksum is inconsistent.
Bits26_25 = [0 0]; % set to zero by default
CE = 1;
while CE
    data0_b = [[0 0 0 0] LstRsp Bits26_25 WR];
data0 = binaryVectorToHex(data0_b);

% Calculating BIP4 checksum
BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);

sendLine = uint8(hex2dec([data0; data1; data2; data3]));
fwrite(obj.ser_obj, sendLine);

% Obtain response from the module
pause(0.25);
rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
rxdata_hex = dec2hex(rxdata_dec);
data = [rxdata_hex(3,1:2) rxdata_hex(4,1:2)];
% convert response to binary
rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

% Analyze response
% Command specific bits
% Channel number
powerResponse = double(typecast(uint16(hex2dec(data)), 'int16'));

% Command independent bits
% Communication Error
CE = rxdata_bin(1, 5);
if CE
    fprintf('
PWR − Communication Error');
end
LstRsp = 1;
end

% Pending Operation Flags
pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
% A series of eight flag bits indicating which operations,
% if any, are still pending. Each operation that becomes pending
% is assigned one of these four bit positions.
% The module can be periodically polled (by reading the NOP register)
% to determine which operations have completed.
% A value of 0x0 indicates that there are no currently pending operations.

% Status Bits
Status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
% 0x00 − OK flag, Normal return status
% 0x01 − XE flag, (execution error)
% 0x02 − AEA flag, (Automatic extended addressing result being returned or ready to write)
% 0x03 − CP flag, Command not complete, pending
% this needs to be changed. the status will most of the time be 3, because it's queried very
% soon after issuing the command

% Final wait for the command to execute
if WR
    fprintf('Setting the power ...');
    PurePhotonicsPPCL300.wait_operation(obj)
    fprintf('SET!
');
end

../+PurePhotonicsPPCL300/ResEna.m

function pendingOperation = ResEna(obj, WR, SENA, MR, SR)
if nargin == 1
    WR=0; SENA = 0; MR = 0; SR = 0;
elseif nargin==3
    MR = 0; SR = 0;
end

commandRegister = '32';  %Register address for the command
data1 = commandRegister;
data2 = '00';  % first data byte in hex
data3 = binaryVectorToHex([0 0 0 0 SENA 0 SR MR]);  % second data byte in hex
LstRsp = 0;  % Bit set to logic 0 when the checksum is consistent.
            % Bit set to logic 1 forces module to resend last valid packet.
            % Used when the checksum is inconsistent.
Bits26_25 = [0 0];  % set to zero by default
CE = 1;
while CE
    data0_b = [0 0 0 0] LstRsp Bits26_25 WR];
data0 = binaryVectorToHex(data0_b);
    % Calculating BIP4 checksum
    BIP4 = PurePhotonicsPPCL300.BIP4(data0, data1, data2, data3);
data0(1) = binaryVectorToHex(BIP4);
sendDate = uint8(hex2dec([data0; data1; data2; data3]));
    fwrite(obj.ser_obj, sendLine);

    % Obtain response from the module
    pause(0.25);
    rxdata_dec = fread(obj.ser_obj, obj.ser_obj.BytesAvailable);
    % convert response to binary
    rxdata_bin = hexToBinaryVector(dec2hex(rxdata_dec), 8);

    % Analyze response
    % Command specific bits
    % Pending Operation Flags
    pendingOperation = binaryVectorToHex(rxdata_bin(3, 1:8));
    % A series of eight flag bits indicating which operations,
    % if any, are still pending. Each operation that becomes pending
    % is assigned one of these four bit positions.
    % The module can be periodically polled (by reading the NOP register)
    % to determine which operations have completed.
    % A value of 0x0 indicates that there are no currently pending operations.

    % Command independent bits
    % Communication Error
    CE = rxdata_bin(1, 5);
    if CE
        fprintf ('\nResEna - Communication Error');
        LstRsp = 1;
    end

    % Status Bits
    status = binaryVectorToDecimal(rxdata_bin(1, 7:8));
    % 0x00 - OK flag, Normal return status
    % 0x01 - XE flag, (execution error)
    % 0x02 - AEA flag, (Automatic extended addressing result being returned or ready to write)
    % 0x03 - CP flag, Command not complete, pending
    if status == 1
        fprintf ('\nResEna - Execution Error, retrying ...');
    
322
Lstrsp = 0; continue;
end
end

% Final wait for the command to execute
fprintf( '\nLaser...');
PurePhotonicsPPCL300.wait_operation(obj)
switch SENA
  case 0
    fprintf('OFF!');
  case 1
    fprintf('ON!');
end

../+PurePhotonicsPPCL300/wait_operation.m

function wait_operation(obj)
  % D. Stahl 12.11.11
  % waits until laser is settled or max_time [s] is elapsed
  timer_start=now;
  completeFlag = false;
  pending0 = 0;
  while ~completeFlag && ((now-timer_start)*24*3600)<=obj.max_time
    ['.', '.', pending, CE, '='] = ...
    PurePhotonicsPPCL300.NOP(obj);
    if (pending0 == hex2dec(pending))||CE:
      fprintf('.');
      pending0 = hex2dec(pending);
    end
    fprintf('.');
    if (hex2dec(pending)); completeFlag=true; end
    pause(0.25);
  end
  fprintf(\n  \nTime elapsed: %3.2f s out of %3.2f s', (now-timer_start)*24*3600), obj.max_time);
assert((now-timer_start)*24*3600)<=obj.max_time ...
  || ((now-timer_start)*24*3600)>100*obj.max_time, ... 
  (\nMax waiting time exceeded!');

A.8 Auxiliary files

../DataTipFunction.m

function output_txt = DataTipFunction(obj,event_obj,rwObj)
  % Display the position of the data cursor
  % obj          Currently not used (empty)
  % event_obj    Handle to event object
  % output_txt   Data cursor text string (string or cell array of strings).
  pos = get(event_obj,'Position');
  output_txt = {{'t=',num2str(pos(1),4)},...
    {'V=',num2str(pos(2),4)}
  }
  obj.TipHandle.Interpreter = 'tex';
  obj.TipHandle.FontSize = 7;
If there is a Z-coordinate in the position, display it as well

if length(pos) > 2
    output_txt{end+1} = ['Z: ', num2str(pos(3),4)];
end

../postBrushEdgeLeftRightFcn.m

function postBrushEdgeLeftRightFcn(~, event_data, rwObj)
    nlines = length(event_data.Axes.Children);
    brushdata = cell(nlines, 1);
    for ii = 1:nlines
        fprintf('Line %i
', ii)
        fprintf('X: %f Y: %f Z: %f
', brushdata{ii})
    end
end

../postBrushFcn.m

function postBrushFcn(~, event_data, rwObj)
    refLevel = mean(event_data.Axes.Children.YData);
    if rwObj.referenceLevelButton.Value
        if ~isempty(strfind(event_data.Axes.YLabel.String, 'GV')); multiplier=le9; end
        if ~isempty(strfind(event_data.Axes.YLabel.String, 'MV')); multiplier=le6; end
        if ~isempty(strfind(event_data.Axes.YLabel.String, 'kV')); multiplier=le3; end
        if ~isempty(strfind(event_data.Axes.YLabel.String, 'mV')); multiplier=le−3; end
        if ~isempty(strfind(event_data.Axes.YLabel.String, 'uV')); multiplier=le−6; end
        if ~isempty(strfind(event_data.Axes.YLabel.String, 'nV')); multiplier=le−9; end
        if strcmp(event_data.Axes.Title.String, rwObj.subplotTitles{1})
            disp(sprintf(' vLin with multiplier %2.3f', multiplier));
            rwObj.referenceLevelShift(1) = ...
            rwObj.referenceLevelShift(1) + multiplier+refLevel; end
        if strcmp(event_data.Axes.Title.String, rwObj.subplotTitles{2})
            disp(sprintf(' vNonLin with multiplier %2.3f', multiplier));
            rwObj.referenceLevelShift(2) = ...
            rwObj.referenceLevelShift(2) + multiplier+refLevel; end
        if strcmp(event_data.Axes.Title.String, rwObj.subplotTitles{3})
            disp(sprintf(' eLin with multiplier %2.3f', multiplier));
            rwObj.referenceLevelShift(3) = ...
            rwObj.referenceLevelShift(3) + multiplier+refLevel; end
        if strcmp(event_data.Axes.Title.String, rwObj.subplotTitles{4})
            disp(sprintf(' eNonLin with multiplier %2.3f', multiplier));
            rwObj.referenceLevelShift(4) = ...
            rwObj.referenceLevelShift(4) + multiplier+refLevel; end
        end
    end