



# FiSpec FBG Sensor Systems

Programmer's manual

Firmware version covered in this handbook:  
FiSpec firmware ver. 10.4

# Contents

1	Data transfer protocol of FiSpec devices.....	1
1.1	Overview of available instructions .....	1
1.2	Communication between FiSpec and host computer .....	3
1.2.1	Basic communication parameters .....	3
1.2.2	Wireless LAN and Ethernet connected FiSpec devices .....	3
1.2.3	Multiplex operation of stacked FiSpec devices .....	4
1.2.4	Transmission of pixel intensities (spectrum).....	5
1.2.5	Transmission of wavelength+intensity or Strain+Temperature.....	6
1.2.6	Transmission of wavelength list .....	7
1.2.7	Handling of floating point values .....	8
1.3	General command set .....	8
1.4	Configuration of Peak detection .....	10
1.5	Readout of present onboard settings .....	12
1.6	Onboard global drift correction .....	14
1.6.1	Calculating wavelengths of pixels.....	14
1.6.2	Calculation of drift corrected wavelengths .....	14
1.6.3	Drift correction commands .....	15
1.7	On board strain/temperature calculation .....	16
1.8	Setting mode and constants for onboard strain/temperature calculations .....	17
1.8.1	Constant TEK mode (mode 0).....	18
1.8.2	Global cubic temperature interpolation (mode 1).....	18
1.8.3	Channel specific error correction (mode 2) .....	19
1.9	Onboard Edge-FBG measurements.....	21
1.9.1	Principles of Edge-FBG measurements .....	21
1.9.2	Variables and channel sorting conventions .....	22
1.9.3	Setting up Edge FBG measurements .....	23
1.9.4	Onboard Edge measurement commands.....	24
1.10	Streaming modes, optional External UART .....	27
1.10.1	Basic UART communication parameters .....	27
1.10.2	Streaming mode .....	28
1.11	Autostart mode, and Saving settings to Flash memory .....	28
2	Program code examples .....	30
2.1	Exemplary commands sequences for setting up onboard calculation.....	30
2.2	LabView program code.....	31

2.3	C program code .....	34
2.4	Python program code.....	37
2.4.1	Python code overview and libraries .....	37
2.4.2	Python file "main.py" .....	39
2.4.3	Python file "FiSpec_GUI.py" .....	46
3	Firmware changelog .....	49

# 1 Data transfer protocol of FiSpec devices

The communication with the spectrometer system bases on a serial protocol whose details are described in the following chapter.

## 1.1 Overview of available instructions

General commands	
?>	Get system name
p?>	Get several parameters
LED, x>	Switches internal light source
WLL>	Get wavelength of pixels list
m, x>	Set averaging
ADCC, x>	Switch adaptive dark frame compensation
ADBA, x, y, z>	Automatic periodic dark frame update
iz, x>	Set integration time
a>	Globally start measurements
o>	Globally stop measurements
s>	Get latest spectrum
AO, x>	Start auto-optimization of integration time / averages
Configuration of peak detection channels	
KA, x>	Set number of active channels
Ke, x, y, z>	Set peak detection channel details
Pv, x>	Switch follow peak function
PNg, x>	Set follow peak limit
PeM, x>	Set peak detection mode (obsolete)
P>	Get latest peak positions/amplitudes
Readout of present onboard setting	
KAa>	Transmit channel number
PAa>	Transmit number of pixels
KLa>	Channel list: start wavelength
KLe>	Channel list: end wavelength
KLWL0>	Channel list: zero wavelength
KLT0>	Channel list: zero temperature
KLTK>	Channel list: temperature compensation
KLTyp>	Channel list: FBG type
aRWL?>	actual reference wavelengths
Max?>	maximum amplitude
e?>	Error code; get signal quality of every peak detection channel
Onboard global drift correction	
RK, x>	Switch reference correction
RTsv, x>	Switch the use of internal temperature sensor
Rta, x>	Set actual reference temperature
RSP, x, y>	Set reference sensor parameters
RS2P, x, y>	Set reference sensor No. 2 parameters

KS1R, x>	Corr. slope single reference FBG
RSA, x>	Number of reference sensors
AS_RK, x>	Sets reference compensation state at device power up
<b>On board strain/temperature calculation</b>	
OBB, x>	Switch on board calculation
OBsaT0, x>	Set same T <sub>0</sub> value for all channels
OBseT0, x, y>	Set T <sub>0</sub> for a specific channel
OBsWL0, x, y>	Set channel's zero wavelength
OBsTyp, x, y>	Set channel type (i.e. temperature/strain)
OBsTK, x>	Set channel's temperature compensation state
OBn, x>	Set actual values as zero values
OBNe, f>	Set actual values as zero values (single fiber)
<b>Setting mode and variables for onboard strain/temperature calculations</b>	
OeK, x>	Set optoelastic constant for strain measurements
TEKI, x>	Set interpolation mode for temperature measurements
TeK, x>	Set TEK for constant value case
KPTWLvs, x, y, z>	Set global "T->V" temperature interpolation parameter
KPWLVTs, x, y, z>	Set global "V->T" temperature interpolation parameter
KPTFTs, f, x>	Set active fiber type for global temperature interpolation
KubParam?>	Get parameters of global temperature interpolation
EKFps, f, x, y, z>	Set channel specific error compensation parameter
EKFParam?>	Get parameters of channel specific error compensation
KalID, w, x, y, z>	Set user selectable calibration ID
KalID?>	Get user selectable calibration ID
<b>Setting mode and variables for onboard Edge-FBG calculations</b>	
EBs, x>	Enable/Disable Edge-FBG calculation
EFPs, x, y, z, f>	Set fiber parameters for Edge-calculations
EaKs, X1, X2, Y1, Y2>	Set same Edge-FBG coordinates for all sensor planes
EXKs, f, x, X1, X2>	Set single sensor plane Edge-FBG coordinates (X pair)
EYKs, f, x, Y1, Y2>	Set single sensor plane Edge-FBG coordinates (Y pair)
EOBN, f>	Zero edge sensors (straight sensor)
EOBRN, f>	Zero edge sensors direction (coiled sensor)
EaPhi0N>	Zero all $\phi_0$
EKBs, x>	Set half channel width for amplitude calculation
EES, f, x, y>	Set ellipsoid angle $\theta$ for a single sensor plane
EaEs, x, f>	Set same ellipsoid angle $\theta$ for all sensor planes in one fiber
EUm>	Measure background spectrum
EK?>	Get all Edge-FBG coordinates
EE?>	Get all ellipsoid angles $\theta$
EFP?>	Get fiber parameters
E>	Get latest edge-FBG measurement results (angle/radius)

<b>streaming modes, optional ext. UART</b>	
<code>extBdR, x&gt;</code>	Set baudrate of optional UART connector
<code>DauSe, x&gt;</code>	Switch/set streaming mode
<code>UARTse, x&gt;</code>	Set UART transmission mode (Firmware <10.0)
<code>MpxNr, x&gt;</code>	Set multiplex number for non stackable devices
<b>Autostart, saving settings to Flash memory</b>	
<code>AuSt, x&gt;</code>	Switch Autostart
<code>speichern&gt;</code>	Save settings to Flash (refer to chapter 1.11!)

*Table 1 Table of available instructions.*

Table 1 lists the serial commands available for the control of the spectrometers. These are described in detail in the following sub chapters.

## 1.2 Communication between FiSpec and host computer

### 1.2.1 Basic communication parameters

The serial communication between spectrometers and host computer is accomplished via a FTDI USB 2.0 HS chip (VID:0403, PID:6015) with a baudrate of 3MBd. The FTDI chip itself supports many different baudrates; however, the microcontroller UART uses a fixed one. To communicate, this parameters shall be used:

Baudrate: 3,000,000 Bit/s

8 data bits, 1 stop bit, no termination character, no handshake

All numbers are sent and received as integers, with certain conversion factors determining the number of decimal places. In general, the factor for wavelengths is 10,000, and the temperature conversion factor is 100. The details are explained below.

The instructions are sent as ASCII strings, using a ">" as only termination character. Beware that this is important, as any additional termination characters like CR/LF will confuse the microcontroller. The standard settings in some serial communication libraries do not work out of the box due to such behaviour; for example, a customer reported communication with his FiSpec worked when using jssc library only after switching to a native interface in his Java environment.

When manually testing commands via a terminal program, it is recommended not to use programs that add termination characters as a default (for example, "PuTTY" does not work). For example, the freeware terminal program "HTerm" (Tobias Hammer, <http://www.der-hammer.info/pages/terminal.html>) works well in this regard.

### 1.2.2 Wireless LAN and Ethernet connected FiSpec devices

There are extension boxes available that can be plugged into the external UART of FiSpec devices and offer network access by providing an Ethernet port or wireless LAN (WLAN).

The extension boxes are using TCP/IP protocol, with either DHCP or fixed IP addresses and port number 8888.

#### Single wireless LAN/Ethernet devices

If a single device, for example an X100, is equipped by a wireless LAN or Ethernet extension, this extension box is essentially transparent for both transmitted and received data. So all commands described in this handbook are to be used as described, with the exception of the ?> command: the

extension box will automatically append the string " WLAN" or " Ethernet" to the FiSpec's answer. So, for example, instead of "FiSpec FBG X100\r\n" the string "FiSpec FBG X100 WLAN\r\n" will be returned.

### Stacked devices with converter box and ID-pins

Stacked devices with ID pins use an Ethernet interface box to connect to the network. This box is also transparent towards transmitted and received data. However, as there can be several devices connected at the same bus at the same time, two things have to be kept in mind:

- the unmodified ?> command will result in the interface box to deliver the string "FBG Interface Converter", regardless of further devices being plugged in or not.
- in order to address a certain device in the stack, the multiplex syntax described in chapter 1.2.3 has to be used.

Furthermore, when sending a command with which no answer of the microcontroller is expected (for example, a>), the interface box sends an automatic answer "ack" after 1ms.

### 1.2.3 Multiplex operation of stacked FiSpec devices

In multiplexed devices, the external UART is used for communication via a serial bus system (i.e. RS485) in which all device's transmit lines (and receive lines, respectively) are connected in parallel. So every FiSpec is listening to the same whole data stream of the host computer, and itself and all of its fellow FiSpec devices will send on the same line. To avoid collisions, it is crucial to use the correct commands.

There is no need to activate multiplex mode in FiSpecs. In stackable multiplex systems, each FiSpec regularly (f=5Hz) scans its ID pins and therefore already knows its unique multiplex ID according to the physical arrangement of the devices.

To address a command to one certain FiSpec device, the associated multiplex ID followed by a vertical bar has to be prefixed to it:

**x | command [ , number1 , number2 , . . . ] >**

x=Multiplex ID of FiSpec to address                      x=1, 2, ..., number of available FiSpec / x=999

command[,variable 1, variable 2, ...]=command with associated variables

If all FiSpec should execute the same command, x=999 has to be used. Beware that this case should be used only with commands that don't generate answers (i.e. a>/ m, x>/...) as then data corruption will occur due to all devices sending at the same time with all of them switching to low impedance state. Same is for the streaming mode; if this mode is enabled with stacked devices, all of them will transmit at the same time corrupting each other's signals.

For example, to request the spectrum from a specific FiSpec with ID=5:

5 | s>

If there is no prefix at all, the FiSpec generally assumes the command is meant for itself and will start working on it. So using "normal" commands in multiplex systems will result in the same results as using a multiplex ID of x=999, including the above mentioned danger of data corruption when using data generating commands.

If no ID pins are available in a device (e.g., FiSpec X100), it is possible to set a fixed Multiplex number. It is assigned by the following command:

**MpxNr , x>**

x=Multiplex number of non-stackable FiSpec                      x=1, 2, ..., 65534

So when using unique Multiplex numbers, it is possible to connect several device's UART interfaces in parallel, even without further electrical buffers, as the devices themselves set their I/O lines to



high impedance state shortly after transmitting. However, this feature is entirely experimental and not officially supported.

#### 1.2.4 Transmission of pixel intensities (spectrum)

When sending the `s>` instruction, the microcontroller answers by sending the intensities of the spectrometer's pixels. The spectral information consists of an array of unsigned 16bit-integers, one for every pixel. The numbers are transmitted as a String of hexadecimal numbers, with each `uint16_t` as a high byte followed by its low byte. The Spectral array (array size: number of pixels) is terminated by a four byte string: "Ende" resp. 0x45 0x6E 0x64 0x65.

The sequence is illustrated in Table 2 (left): In the first three 16 bit numbers, the pixel intensity information were exchanged by some real time information before transmission:

- First `int16`: on board sensor temperature(°C)\*100. In this number, high/low byte are arranged as described above. However, in this case they have to be treated as a signed 16 bit number.
- Second `int16`: currently calculated Refslope=slope\*1,000,000 of the on board drift compensation.
- Third `int16`: currently calculated RefOffset=pixelshift\*10,000 of the on board drift compensation.
- after that many `uint16` numbers (unsigned!): intensity of the pixels.

If the system is a multi-fiber one, data is now transmitted for the other fibers as well (grey area in Table 2). The system temperature in this case is always the same, as there is only one sensor per system, but reference slope and offset are unique for the different fibers.

After all data is transmitted, the "Ende" string is transmitted.

transmitted spectrum			transmitted wavelength list				
Fiber 0	Temp	Bit 0...15		WL_00	Bit 16...31	WL_00	Bit 0...15
	RefSlope_0	Bit 0...15		WL_01	Bit 16...31	WL_01	Bit 0...15
	RefOffset_0	Bit 0...15		WL_02	Bit 16...31	WL_02	Bit 0...15
	Int_03	Bit 0...15		WL_03	Bit 16...31	WL_03	Bit 0...15
	Int_04	Bit 0...15		WL_04	Bit 16...32	WL_04	Bit 0...15
	⋮			⋮			
	Int_0m	Bit 0...15		WL_0m	Bit 16...31	WL_0m	Bit 0...15
Fiber 1	Temp	Bit 0...15		WL_10	Bit 16...31	WL_10	Bit 0...15
	RefSlope_1	Bit 0...15		WL_11	Bit 16...31	WL_11	Bit 0...15
	RefOffset_1	Bit 0...15		WL_12	Bit 16...31	WL_12	Bit 0...15
	Int_13	Bit 0...15		WL_13	Bit 16...32	WL_13	Bit 0...15
	Int_14	Bit 0...15		WL_14	Bit 16...31	WL_14	Bit 0...15
	⋮			⋮			
	Int_1n	Bit 0...15		WL_1n	Bit 16...31	WL_1n	Bit 16...31
⋮	⋮		⋮				

Fiber 3	Temp	Bit 0...15	WL_30	Bit 16...31	WL_30	Bit 0...15
	RefSlope_3	Bit 0...15	WL_31	Bit 16...31	WL_31	Bit 0...15
	RefOffset_3	Bit 0...15	WL_32	Bit 16...31	WL_32	Bit 0...15
	Int_33	Bit 0...15	WL_33	Bit 16...32	WL_33	Bit 0...15
	Int_34	Bit 0...15	WL_34	Bit 16...31	WL_34	Bit 0...15
	⋮				⋮	
	Int_3p	Bit 0...15	WL_3p	Bit 16...31	WL_3p	Bit 0...15
	"En"		"En"		"de"	
	"de"					

Table 2 Data format for spectral data (s>, left) and wavelength list (WLL>, right).

Beware that although the pixel information is exchanged by other information, the wavelength scale (WLL> instruction) starts at the first transmitted item (in this case, the temperature value), like shown in Table 2 (right).

In single-fiber systems, no further information about the pixel numbers is needed. To assign the values correctly in multi-fiber systems, however, one has to gain information about how many pixels are related to which fiber. This can be done by sending the PAa> command once, like described in chapter 1.5.

### 1.2.5 Transmission of wavelength+intensity or Strain+Temperature

When using the P> instruction, depending on the on board calculating mode either the intensities and wavelength positions of the FBG peaks or the on board calculated strain and temperature values are sent to the host computer.

Fiber 0	WL_00	Bit 16...31	WL_00	Bit 0...15
	Int_00	Bit 16...31	Int_00	Bit 0...15
	WL_01	Bit 16...31	WL_01	Bit 0...15
	Int_01	Bit 16...31	Int_01	Bit 0...15
	⋮			
	WL_0m	Bit 16...31	WL_0m	Bit 0...15
	Int_0m	Bit 16...31	Int_0m	Bit 0...15
	Temp	Bit 0...15	0 (Empty)	
	RefSlope_0	Bit 0...15	RefOffset_0	Bit 0...15
Fiber 1	WL_10	Bit 16...31	WL_10	Bit 0...15
	Int_10	Bit 16...31	Int_10	Bit 0...15
	WL_11	Bit 16...31	WL_11	Bit 0...15
	Int_11	Bit 16...31	Int_11	Bit 0...15
	⋮			
	WL_1n	Bit 16...31	WL_1n	Bit 0...15
	Int_1n	Bit 16...31	Int_1n	Bit 0...15

	<b>Temp</b>	Bit 0...15	0 (Empty)	
	<b>RefSlope_1</b>	Bit 0...15	<b>RefOffset_1</b>	Bit 0...15
...	⋮			
<b>Fiber 3</b>	<b>WL_30</b>	Bit 16...31	<b>WL_30</b>	Bit 0...15
	<b>Int_30</b>	Bit 16...31	<b>Int_30</b>	Bit 0...15
	<b>WL_31</b>	Bit 16...31	<b>WL_31</b>	Bit 0...15
	<b>Int_31</b>	Bit 16...31	<b>Int_31</b>	Bit 0...15
	⋮			
	<b>WL_3p</b>	Bit 16...31	<b>WL_3p</b>	Bit 0...15
	<b>Int_3p</b>	Bit 16...31	<b>Int_3p</b>	Bit 0...15
	<b>Temp</b>	Bit 0...15	0 (Empty)	
	<b>RefSlope_3</b>	Bit 0...15	<b>RefOffset_3</b>	Bit 0...15
	"En"		"de"	

Table 3 Data format in case of onboard peak detection.

The data is also transmitted as a hexadecimal string array. Starting with channel 0, the wavelength(nm)\*10,000 is transmitted as an int32\_t number, followed by the amplitude\*10,000 (also a int32\_t number). The other channels are transmitted likewise successively. The 32 bit numbers all start with the highest byte down to the lowest.

When the onboard calculation case is enabled (see chapter 1.7), instead of wavelength and amplitude, strain( $\mu\text{m}/\text{m}$ )\*10,000 and temperature( $^{\circ}\text{C}$ )\*100 are transmitted in the same way.

After the peak wavelength/amplitude or strain/temperature information, there is another "channel" transmitted consisting of two 32 bit numbers featuring the same format (high byte down to low byte).

- first int16: temperature( $^{\circ}\text{C}$ )\*100 of the on board temperature sensor
- second int16: empty (zero)
- third int16: current reference compensation value; RefSlope= slope\*1,000,000
- fourth int16: current reference compensation value; RefOffset= offset(nm)\*10,000

In case of a multi-fiber system, the measured data of the other fibers is transmitted likewise (see Table 3, grey section). Please use the `KAa>` command (see chapter 1.5) once before to know how many channels there are per fiber.

The complete array size therefore is (number\_of\_channels+1)\*number\_of\_fibers and like in the spectral case it is terminated by a four byte string: "Ende" resp. "0x45 6E 64 65".

### 1.2.6 Transmission of wavelength list

When answering the `WLL>` instruction, the microcontroller sends wavelength(nm)\*10,000 for each pixel successively. This is done like in the "peak detection" case above; each number is a signed 32 bit integer starting at the highest byte. Termination string is also "Ende". For details, see Table 2 (right).

In multi-fiber systems, the wavelengths of all fibers are transmitted successively. It is therefore needed to send the `PAa>` command once, like described in chapter 1.5, to be able to relate the wavelength information correctly.

Alternatively, the pixel's wavelengths can be calculated as mentioned in chapter 1.6.1.

### 1.2.7 Handling of floating point values

Most commands used in FiSpec devices operate with integer values, using certain factors to achieve sufficient decimal places. However, this method reaches its limitations when using numbers who may differ in several orders of magnitude, like is the case with the cubic polynomial temperature interpolation (see chapter 1.8). So in order to transfer the interpolation parameters TV\_0...TV\_3, VT\_0...TV\_3 and TC\_0...TC\_3 mentioned in that chapter, their 32 bit single precision floating point values have to be type casted to 32 bit integers and vice versa. The transfer in general follows the same rules like the integer values; so the same byte/word-swapping due to the little-endianess of the microcontroller has to be done. The floating point values follow the IEEE 754 standard.

As the commands are sent in plain text format, before sending a floating point value it just has to be casted to an `int32_t` value. In Fig. 9, an example for LabView is shown.

When receiving values, similar to how in chapter 1.2.4 described the data has to be casted to `uint16_t` numbers first, byte and word swapped due to little endianess, combined to 32bit values and then casted to single precision floating point values. As an example, Fig. 10 shows how to get the TV and VT parameters via the `KubParam?>` command.

### 1.3 General command set

This chapter contains the basic commands for operating the spectrometer. For programming it is recommended to first ask for system information (Pixel numbers, wavelength information and the like) and set all important settings (integration time, averaging, ...). Sending the `a>` command starts measurements. The system continuously measures spectra and calculates positions and intensities in the pre-set peak channels. Each time when the `s>` or `P>` command is sent, the microcontroller sends the desired information of the latest elapsed measurement. If there is no new data since the last request, the answer is delayed accordingly until new data is available.

**?>** *"?"*

Asks for the system name.

Answer: String with Terminator `\r\n`, i.e. „FiSpec FBG X100 `\r\n`“

**p?>** *"Parameter? / parameters?"*

Requests the system's parameters.

Answer: String with terminator `\r\n`. The string contains variable names (between `"#"` and `"_"`) and the corresponding integer values (between `"_"` and the next `"#"`). The following variables are transmitted:

<code>#Version_x</code>	<code>x=Firmware version*10</code>
<code>#Pixel_x</code>	<code>x=Number of measured pixels</code>
<code>#Mindestintegrationszeit_x</code>	<code>x=Minimum integration time (µs)</code>
<code>#Seriennummer_x</code>	<code>x=Serial number</code>
<code>#A1/A2/A3_x</code>	<code>internal pixel parameters (see chapter 1.6)</code>
<code>#B1/B2/B3_x</code>	<code>internal wavelength parameters (see chapter 1.6)</code>
<code>#Kalibrierungstemperatur_x</code>	<code>x=100*system temperature during factory wavelength calibration</code>
<code>#Kanalanzahl_x</code>	<code>number of currently active peak detection channels</code>
<code>#IntReferenz_x</code>	<code>x=0/1/2; none, one or two internal reference FBG available</code>
<code>#WL0Ref_x</code>	<code>x=10,000*wavelength(nm) of internal reference FBG sensor @ T<sub>0</sub></code>
<code>#KanalbreiteRef_x</code>	<code>x=10,000*width(nm) of reference FBG sensor peak detection channel</code>
<code>#T0Ref_x</code>	<code>x=100*T<sub>0</sub> calibration temperature of reference FBG sensor</code>
<code>#tInt_x</code>	<code>x=integration time (µs) at startup</code>
<code>#Mittelungen_x</code>	<code>x=spectrum averages at startup</code>

#Dauersenden_x	x=0/1/2; state of streaming mode at startup
#Autostart_x	x=0/1; Autostart at startup dis-/enabled
#OBBerechnen_x	x=0/1; onboard strain/temp. calculation dis-/enabled at startup
#UARTModus_999	obsolete; for compatibility reasons with older firmware
#extBaudrate_x	x=Baudrate of optional external UART connector at startup
#Schreibzugriffe_x	x=number of write processes to the flash memory. Standard value: 1
#Faseranzahl_x	x=number of fibers in multi-fiber systems
#MultiplexNr_x	x=Multiplex ID in stacks of multiplexed systems
# Intern_x	x=internally readout pixels
#St_x	x= wavelength compensation slope
#TEK_x	x=thermoelastic constant in the constant value interpolation case
#TEKRef_x	x=value used for internal reference FBG wavelength compensation
#OEK_x	x=optoelastic constant used for onboard strain measurements
#TEKinterpol_x	x=0/1/2; temperature interpolation mode (refer to <code>TEKI, x&gt;</code> command in chapter 1.8)
#RK_x	x=0/1; reference compensation dis-/enabled
#AStartRK_x	x=0/1; reference compensation dis-/enabled state at startup
#PeakErkM_x	x=0/1; peak detection mode (center of gravity, derivative)
#EdgeKB_x	x=0...8; half channel width for Edge-FBG calculations

*Table 4 Transmitted parameter string variables.*

In case of a multi-fiber system (#Faseranzahl>1) some variables (A, B, Kanalanzahl, WLORef, ...) are modified to indicate their respective fiber. Fiber numbers range from 0...3, and the variable names are extended by adding "\_" and the fiber number. For example, instead of "#A1\_901" it is then "#A1\_0\_901".

**LED, x>** *"LED schalten / switch LED"*

Switches the internal light source of the interrogator systems on (x=1) or off (x=0).

**WLL>** *"Wellenlängenliste / wave length list"*

Requests a list of wavelengths of each Pixel.

Answer: number\_of\_pixels\*32 bit, ends with „Ende“ (see chapter 1.2.6 for details).

Alternatively, the pixel's wavelengths can be calculated as described in chapter 1.6.1.

**m, x>** *"Mittelungen / averaging"*

x=1...1000

Sets on board averaging (x=1=>no averaging). In this mode, the full internal speed of the system can be used to get better measurement quality, even if data transmission is slower.

**ADCC, x>** *"Adaptiver Dunkelbildabzug / Adaptive Dark Current Compensation"*

x=1/0

Enables (x=1) or disables (x=0) the adaptive onboard dark frame compensation. This function uses either the darkframe already stored in factory, or, if activated by `ADBA, 1, y, z>` command, a dark-frame automatically updated in between measurements.

**ADBA, x, y, z>** *"Automatische Dunkelbildaktualisierung / Automatic periodic dark frame update"*

x=1/0

y=update time(s)\*100 between dark frame measurements      z=2...4000

z=darkframe low pass filter

z=1...100

Enables (x=1) or disables (x=0) the automatical periodic darkframe update. In case of x=1, the time between updates can be chosen by y. Beware that dark frame compensation (command **ADCC, x>**) has to be active for this command to show any effect.

**iz, x>** *"Integrationszeit / integration time"*

x=integration time(µs) x=30...65,000,000

Stops measurement if already running, sets integration time and returns to preliminary state.

**a>** *"St\_a\_rten / start measurements"*

Starts internal measurements. Has to be called once before requesting spectra or peaks. It is recommended to first set all other parameters before starting measurements.

**o>** *"St\_o\_ppen / stop measurements"*

Stops internal measurements. May be called after finishing measurements, but is not mandatory (spectrometer doesn't enter idle mode or the like).

**s>** *"Spektrum ausgeben / get spectrum"*

Requests last measured spectrum.

Answer: pixel amplitudes; output format is described in chapter 1.2.

**AO, x>** *"Auto-Optimierung/ Auto-optimization"*

x=Target frequency (Hz) x=1...5000

Requests an auto-optimization of integration time and averages to achieve a certain target frequency according to the currently connected sensors's signal amplitude. The algorithm tries to optimize the maximum signal amplitude while still regaining some headroom, and further optimizes the averages to reach the desired frequency to get the best signal quality possible.

Optimization bases on the transfer rate of the USB connection and the number of channels to be transferred; the transfer of whole spectra will be slower.

Answer: array of three 32bit numbers, followed by "Ende". The three numbers contain:

1. new integration time (µs) / 2. new averages / 3. error code.

Error code:

0 = target frequency achieved (as good as technically feasible; limitations due to USB bandwidth possible)

1 = target frequency achieved (as good as technically possible), but amplitude lower than the amplitude optimization target range (although still as high as possible)

2 = error! Algorithm is not converging (=> integration time/averages remain like before)

3 = target frequency out of range (x<1 or x>5000Hz)

## 1.4 Configuration of Peak detection

While the spectrometer is active, all active peak search channels are evaluated each time a spectrum has been measured. Therefore it is possible to ask for peak information instead of a whole spectrum at any time after having set channel number and channel boundaries. Beware that setting channel boundaries does not automatically activate this channels; only the first x channels (set by **KA, x>**) are calculated and transmitted.

**KA, x [ , f ]>** *"Kanal-Anzahl / number of channels"*

x=1...32 (normal peak detection channels)

f=fiber number (optional, for multi-fiber system) f=0...3

From now on, the set number of channels will be measured and transferred when requested. Pre-defined channels are already present on board, so choosing more channels than having defined before does no harm but may result in unexpected Peak amplitudes/wavelengths.

**Ke, x, y, z [, f] >** *"Kanal einstellen / set channel"*

x=peak detection channel number x=0...31(23<sup>2</sup>)

y=  $\lambda_s$ (nm) \*10,000 x=999/1000: internal reference channel(s)

z=  $\lambda_e$ (nm)\*10,000

f=fiber number (optional, for multi-fiber system) f=0...3

Defining the range of a single peak detection channel at a time, starting at a wavelength  $\lambda_s$  until  $\lambda_e$ . Beware: if the desired peak detection channel is wider than 200 pixels, this command will be rejected.

With this command it is also possible to redefine the reference sensor channel to use an external sensor. In this case, channel numbers of 999 (first sensor) and 1000 (second sensor, if any) have to be used. Beware that in case of reference channel widths of <0.1nm no reference compensation will be applied.

**Pv, x >** *"Peaks verfolgen / follow peaks"*

x= 1/0

Switches peak following on (x=1) or off (x=0) in case of onboard peak detection. The microcontroller itself shifts the peak detection channels according to the detected wavelength shift of the FBGs. The original channel positions are saved onboard, so when disabling peak following, the original state is being recovered.

**PNg, x >** *"Peaknachführgrenze / follow peak limit"*

x= (0.01...0.49)\*100

In case of activated onboard peak following, this value determines the maximum peak correction allowed per measurement, where x=1...49 equals values of 1...49% of the actual channel width. If the calculated peak channel correction is greater than this range, the measured value is decided to be an error and therefore the peak detection channel is not changed in this case.

**PeM, x >** *"Peakerkennungsmodus / peak detection mode"* (obsolete since v10.3)

x= 1/0

Switches between the modes of onboard peak detection. By default, peaks are detected by determining the center of gravity of each peak with a detection limit of 20% (x=0). The mode can be changed to an extreme value determination using the first derivative (x=1) for firmware versions <10.3.

**P >** *"Peaks ausgeben / get peaks"*

Requests last measured peaks (position/amplitude or strain/temperature); equivalent to the *s >*-command.

Answer: detected position and amplitude (or strain/temperature with enabled on board calculation, see chapter 1.7) of the active peak channels. One additional "channel" consists of on-board sensor temperature and WL compensation; see chapter 1.2.5 for details.

## 1.5 Readout of present onboard settings

If peak detection channels different to the delivery standard values were saved in the EEPROM, it can be useful to know details. In this chapter, such commands are described. In general, the answer to this commands is an array of numbers. These are transmitted in the same way as described in chapter 1.2.6: they consist of `int32_t` numbers (exception with `uint16_t` numbers: `KAa>` and `PAa>` commands). In case of a multi-fiber system, the information for fiber 0, 1, 2, ... follows successively. After all values are sent, they are followed by "Ende" as terminator.

For transmission over UART, UART mode 4 is necessary.

Please bear in mind that the internal reference FBG channels are not covered by this commands if not explicitly stated so. If you need information about these, you may get it by interpreting the information string following the `p?>` command (see chapter 1.3).

**KAa>** *"Kanalanzahl ausgeben" / "transmit number of channels"*

Answer: One (or, in multi-fiber systems, as many as fibers exist) number representing the number of active channels (set by the `KA, x[, f]>` command before). Numbers are in `uint16_t` format.

**PAa>** *"Pixelanzahl ausgeben" / "transmit number of pixels"*

Answer: One (or, in multi-fiber systems, as many as fibers exist) number representing the number of transmitted pixels when using the `s>` command. This command is mandatory in multi-fiber systems, as it is needed to assign the answer of the `s>` and `WLL>` command to the single fibers. Numbers are in `uint16_t` format.

**KL a>** *"Kanalliste: Anfangswellenlänge / channel list: start wavelength"*

Answer: Formatted like described in the `WLL>` command (see chapter 1.2.6), the transmitted data consists of an array of wavelengths\*10,000, representing the start wavelengths of the presently active peak detection channels. There are as many of them as stated in the `KAa>` command.

**KL e>** *"Kanalliste: Endwellenlänge / channel list: end wavelength"*

Answer: Formatted like described in the `WLL>` command (see chapter 1.2.6), the transmitted data consists of an array of wavelengths\*10,000, representing the end wavelengths of the presently active peak detection channels. There are as many of them as stated in the `KAa>` command.

**KLWL0>** *"Kanalliste: Nullwellenlänge / channel list: zero wavelength"*

Answer: Formatted like described in the `WLL>` command (see chapter 1.2.6), the transmitted data consists of an array of wavelengths\*10,000, representing the zero wavelengths of the presently active peak detection channels. There are as many of them as stated in the `KAa>` command.

**KL T0>** *"Kanalliste: Nulltemperatur / channel list: zero temperature"*

Answer: Formatted like described in the `WLL>` command (see chapter 1.2.6; also `int32_t` numbers instead of the usual temperature format `int16_t` for the sake of similar handling of all `KL...>` commands), the transmitted data consists of an array of wavelengths\*100, representing the zero temperature of the presently active peak detection channels. There are as many of them as stated in the `KAa>` command.

**KLTK>** *"Kanalliste: Temperaturkompensation / channel list: temperature compensation"*

Answer: Formatted like described in the `WLL>` command (see chapter 1.2.6), the transmitted data consists of an array of numbers, representing if the associated FBG is temperature compensated (number=1) by the temperature value of the FBG listed one position before (see explanations for



commands `OBsTyp, x, y, [f]>` and `OBsTK, x, y[, f]>` in chapter 1.7) or not (number=0). There are as many `i32_t` numbers as stated in the `KAa>` command.

**KLTyp>** *"Kanalliste: FBG-Typ / channel list: FBG type"*

Answer: Formatted like described in the `WLL>` command (see chapter 1.2.6), the transmitted data consists of an array of numbers, representing if the associated FBG is declared to be a mechanically decoupled temperature FBG (number=1) or a strain FBG (number=0). There are as many `i32_t` numbers as stated in the `KAa>` command.

**aRWL?>** *"aktuelle Referenzwellenlängen?" / "actual reference wavelengths?"*

For informational purposes; it may be useful to know the presently internally measured reference FBG wavelengths.

Answer: Array of Reference WL 1 and Reference WL 2 (regardless of if a system may have only one Reference FBG per fiber), as many times as fibers are in the system, followed by "Ende" in the very end. All numbers are `int32_t`.

**Max?>** *"maximale Intensität?" / "maximum amplitude?"*

If the FiSpec is used in peak detection mode only without transmitting whole spectra, this function may be useful to check how much signal headroom is left to avoid overexposure. The microcontroller answers with an array of as much `uint16_t` values as fibers (in a multi-fiber system; else one single value) exist, followed by "Ende", as such being of the same format as the answer to the `s>` command. This values represent the hardware ADC value of the most intensely illuminated pixel in each fiber's respective CMOS sensor sector (maximum value: 65535), irrespective of any background subtraction.

**e?>** *"Fehlercode?" / "Error code?"*

In order to get an easily interpretable information about insufficient signal quality or fiber breakage, this command delivers the status of every possible FBG peak detection channel (i.e. four fibers, each with a maximum of 32 peak detection channels), regardless if this full number of channels is set or not or less than four fiber ports physically exist. The threshold used to differentiate between "good" and "bad" signal qualities is the same like used in the "follow peak" mode. So if the signal to background ratio is too weak (varying also according to the number of averages set) the error bits of the according peak detection channels will be set and the channel borders will not follow the measured peak wavelength.

Additionally, if a measured peak amplitude is >63,000, the error bit of this FBG channel is also set, as in this case signal clipping is highly likely.

The answer consists of six 32 bit numbers, followed by "Ende".

The respective 32 bits of the first four 32 bit numbers represent the FBG channels, with 1=bad signal, and 0=good signal. Channels not in use are set as "0".

In the fifth 32 bit number, each byte represents one of the four fiber ports, while the first four bits of each byte indicate four different error codes:

- Bit 0: S/N ratio
- Bit 1: over exposure
- Bit 2: peak following
- Bit 3: reference FBG error

These bits will be set as soon as at least one of the peak detection channels of this fiber port exhibits the respective error.

The sixth 32 bit number is for future use.

Beware that there may be a signal level warning even if the FBG peaks are still visible. In this case, the measured value may be still meaningful, but will be noisier and more unreliably than normal. Fig. 8 shows a simple LabView implementation of this command.

## 1.6 Onboard global drift correction

### 1.6.1 Calculating wavelengths of pixels

The easiest way to derive the (uncompensated) wavelength to each pixel is to use the `WLL>` command (see chapter 1.2.6). Alternatively it is also possible to calculate the wavelengths out of the A and B values derived by the `p?>` command :

Keep in mind that the first transmitted pixel of a fiber port does not necessarily start at the physical pixel number 1 ; it can be located anywhere on the sensor. A certain fiber port's start pixel equals the related physical pixel number  $A_2$  ; all in all the fiber port features a number of  $A_1$  pixels.

The wavelength of each pixel is calculated like follows:

$$\lambda(P) = 0,0001 * (B_1 * 0.007^2 * P^2 + B_2 * 0.007 * P + B_3)$$

with  $P = A_2 + i$

and  $i = 0, 1, 2, \dots, (A_1 - 1)$

$\lambda(P)$  = wavelength of a certain physical pixel, calculated from A-values

$P$  = physical Pixel number

$A_1/A_2/A_3$  = internal pixel parameters

$B_1/B_2/B_3$  = internal wavelength parameters

### 1.6.2 Calculation of drift corrected wavelengths

An automatic global correction of the spectrometer wavelength according to one or two dedicated FBGs ("Drift compensation") is possible. When transmitting complete spectra (`s>` instruction) with activated on board drift correction, these spectra contain the measured wavelength compensation information (slope and offset; "second/third int16\_t" as described in chapter 1.2.5).

It is possible to ignore the reference FBG that has been mounted inside the FiSpec devices and to use two external temperature stabilized FBG sensors instead to further optimize the system stability. In this case, the peak detection channels are established by the `Ke, x, y, z [, f]>` command (see chapter 1.4), the internal temperature sensor has to be disabled (command `RTsv, x>`), and the associated external temperature has to be set (command `RTa, x>`).

If spectra are measured by the `s>` command, the wavelength of the spectrum has to be shifted globally on the host computer by calculating the compensated x-axis values out of the actual slope/offset values and the uncompensated wavelength values once derived by the `WLL>` command (or calculated according to chapter 1.6.1) every time a new whole spectrum arrives.

In case of onboard peak detection, the transmitted peak wavelength information (`P>` instruction) is already corrected. The peak array also includes the drift compensation values for informational purposes, although in this case they are not needed for further calculations.

The wavelength of each pixel is calculated like follows:

$$\lambda_{corr} = \lambda + slope * (\lambda - \lambda_{ref\_0}) + offset$$

$\lambda_{corr}$  = compensation corrected wavelength of the pixel

$\lambda$  = wavelength of the pixel according to the `WLL>` command array

$\lambda_{ref\_0}$  = zero wavelength of the reference FBG

slope/offset: compensation values derived from microcontroller

### 1.6.3 Drift correction commands

For using onboard global drift correction, the following commands are used:

**RK, x>** *"Referenzkorrektur / enable reference correction"*

x=1/0

Enables (x=1) or disables (x=0) the onboard drift correction

**RTsv, x>** *"Referenztemperatursensor verwenden / use reference temperature sensor"*

x=1/0

If an internal temperature sensor is present and the reference FBG is thermally tightly coupled to it, the temperature information may be used for a separate temperature compensation of the reference FBG to further enhance the compensation. This command enables (x=1) or disables (x=0) the use of the internal temperature sensor for this purpose.

**RTa, x>** *"Referenztemperatur aktuell / set actual reference temperature"*

x=T(°C)\*100                      x=-5000...20000 (i.e. -50...200°C)

If an internal temperature sensor is not used, this command tells the system about the actual temperature T of the reference FBG sensor.

**RSP, x, y[, f]>** *"Referenzsensorparameter / reference sensor parameters"*

x=Wavelength  $\lambda$ (nm)\*10,000

x=3,500,000...20,000,000

y=corresponding temperature T(°C)\*100

y=-5000...20,000

f=fiber number (optional, for multi-fiber system)

f=0...3

Sets the calibration parameters of the reference FBG sensor, i.e. the known wavelength  $\lambda$  at a known temperature T.

**RS2P, x, y[, f]>** *"Referenzsensor 2-Parameter / reference sensor No. 2 parameters"*

Like RSP, x, y[, f]>; sets values of an optional second temperature reference sensor.

**KS1R, x[, f]>** *"Korrektursteigung einzelnes Referenzgitter" / "corr. slope single reference FBG"*

x=slope\*1,000,000

f=fiber number (optional, for multi-fiber system)                      f=0...3

In case of only one reference FBG, the wavelength offset is determined automatically by the micro-controller, whereas the slope value can be set by the user with this command.

**RSA, x>** *"Referenzsensoranzahl" / "number of reference sensors"*

x=number of (external) reference FBG sensors                      x=1/2

In case of external reference sensors this command determines if one or two of them should be used.

**AS\_RK, x>** *"Autostart-Referenzkompensation" / "autostart reference compensation"*

x=1/0

This variable determines if the reference compensation is switched on at device power up (x=1) or not (x=0). It can be saved to flash memory by the `speichern>` command.

## 1.7 On board strain/temperature calculation

It is possible to let the microcontroller do the calculations for strain and temperature measurements, thus eliminating the need for calculating these values externally. Beware, however, that these calculations base on the on board calculated peak values, so that the results are a bit noisier than the ones derived by mathematically more complex Gauß fits like e.g. the ones used in the BraggSens software. Each FBG and temperature compensation FBG to be detected needs its own detection channel defined by its left and right border like already described in the `Ke, x, y, z` command. Apart from these, in onboard calculation mode each channel gets a series of additional parameters by commands described below.

In general, if an FBG channel (channel number  $i$ ) is set as being a temperature compensated strain FBG, the FBG in the channel directly underneath (channel number  $i-1$ ) will be used by the algorithm as the associated temperature compensation FBG. So the FBG channels have to be arranged with this in mind.

**OBb, x** > *"OnBoard-Berechnung / Switch On board calculation"*

x=1/0

Globally enables (x=1) or disables (x=0) the onboard strain and temperature calculation. When enabled, the `P` command will return an array of Strain/Temperature values instead of Peak Wavelength/Amplitude values. As a standard setting at startup on board calculation is off, as long as not saved to flash memory by the `speichern` command otherwise.

**OBsaT0, x, f** > *"onboard: setze alle  $T_0$  / set all  $T_0$  values at once"*

x=zero Temperature  $T_0(^{\circ}\text{C})*100$

f=fiber number (optional, for multi-fiber system) f=0...3

Set same  $T_0$  value for all channels. This command can be used when assuming that all FBGs involved are at the same (room-) temperature when zeroing.

**OBseT0, x, y, f** > *"onboard: setze einzelnes  $T_0$  / set  $T_0$  for a specific channel"*

x=peak detection channel number x=0...31

y=zero temperature  $T_0(^{\circ}\text{C})*100$

f=fiber number (optional, for multi-fiber system) f=0...3

If FBGs (and their respective optional temperature compensation FBGs) are mounted at places with different Temperatures at zeroing time, their zero temperatures can be set individually with this command.

**OBswl0, x, y, f** > *"onboard: setze Nullwellenlänge / set zero wavelength for a specific channel"*

x=peak detection channel number x=0...31

y=zero wavelength(nm)\*10,000

f=fiber number (optional, for multi-fiber system) f=0...3

Set zero wavelength ( $\lambda_{s,0}$  or  $\lambda_{\theta,0}$ ) of a specific channel when sending the `OBn` command is not desired, i.e. when setting zero wavelengths according to measurements in the past regardless of actual value.

**OBsTyp, x, y, f** > *"onboard: setze Gittertyp / set FBG type"*

x=peak detection channel number x=0...31

y=FBG type y=0/1

f=fiber number (optional, for multi-fiber system) f=0...3

Set FBG type in channel x as temperature (y=1) or strain (y=0) FBG.

**OBsTK, x, y [, f] >** *"onboard: setze Temperaturkompensation / set temperature compensation "*

x=peak detection channel number x=0...31

y=temperature compensation state y=1/0

f=fiber number (optional, for multi-fiber system) f=0...3

Assigns if the strain FBG in channel x shall be temperature compensated (y=1) by another one (per definition the FBG in the channel with number x-1) or not (y=0). Temperature FBGs naturally have no further temperature compensation FBGs themselves.

When all settings are set, the onboard calculation can be switched on with the **OBB, x >** command. Furthermore, to get sensible values it is necessary to "zero" the system one time in strainless state like pressing the "zero Temp/strain" button in the BraggSens software. This is accomplished by the **OBN >** command:

**OBN >** *"onboard-Nullen / Set actual values as zero values"*

Sets all current wavelengths of all fibers (in a multi-fiber system) as zero wavelengths.

**OBNe, f >** *"onboard-Nullen einzeln/ Set actual values as zero values (single fiber)"*

f=fiber number (for multi-fiber system) f=0...3

Sets all current wavelengths of a single fiber (in multi-fiber systems) as zero wavelengths.

## 1.8 Setting mode and constants for onboard strain/temperature calculations

When being used out of the box, predefined constant values for both optoelastic constants ("OEK", strain measurements) and thermoelastic constants ("TEK", temperature measurements) are used. In general, these constants deliver good results.

For strain measurements, the OEK can be changed:

**OeK, x >** *"optoelastische Konstante" / "optoelastic constant"*

x=OEK\*1000 x=0...50,000 (i.e. OEK=0...50.000)

Sets the optoelastic constant for onboard strain measurements.

As the material of the glass fibers changes its optical characteristics over a broader temperature range and furthermore different glass fibers differ slightly from each other, for temperature measurements sometimes it is advisable to exchange the thermoelastic constant (TEK) by polynomial approximations to optimize accuracy. Starting from firmware v. 9.5 on, there are three different temperature calculations possible:

- mode "0": constant TEK (one single constant for all measurements)
- mode "1": global third order polynomial temperature interpolation with one set (up to four different sets, e.g. for different fiber types, can be saved in flash memory) of coefficients
- mode "2": peak detection channel specific third order polynomial temperature error correction. The temperatures calculated out of the chosen global polynomial parameter set are peak detection channel specifically corrected each by a specific cubic polynomial error function. This can be used for pairing specific sensor fibers to a specific device by calibrating both of them together, enabling the best possible accuracy.

This three different methods are described in the following sections.

The interpolation mode can be chosen by using the **TEKI, x >** command:

**TEKI, x>** "TEK-Interpolation" / "thermoelastic constant interpolation"

x=0/1/2

Sets the above mentioned interpolation mode. The interpolation state will be saved to EEPROM when using the `speichern>` command.

### 1.8.1 Constant TEK mode (mode 0)

For temperature measurements in interpolation mode 0, the TEK can be set:

**TeK, x>** "optoelastische Konstante" / "optoelastic constant"

x=TEK\*1E9 x=0...50,000 (i.e. TEK=0...50.000\*10<sup>6</sup>)

Sets the thermoelastic constant for onboard temperature measurements.

All further temperature measurements of all FBG channels will then be based on this value.

### 1.8.2 Global cubic temperature interpolation (mode 1)

The cubic polynomial interpolation bases on calibration measurements of wavelength shifts relative to the sensor temperature. For determining the coefficients experimentally, the following steps have to be carried out:

- increase the sensor temperature T in appropriate steps and afterwards measure wavelengths of all FBGs.
- calculate the change in wavelength value for each temperature step and FBG channel:

$$V=(\lambda/\lambda_0)-1$$

- for each FBG channel: plot a T- V diagram and the inverse V-T-diagram (with [T]=°C).
- fit polynoms (parameters VT\_0...VT\_3 and TV\_0...TV\_3) to this diagrams:

$$T = VT_0 + VT_1 * V + VT_2 * V^2 + VT_3 * V^3$$

$$V = TV_0 + TV_1 * T + TV_2 * T^2 + TV_3 * T^3$$

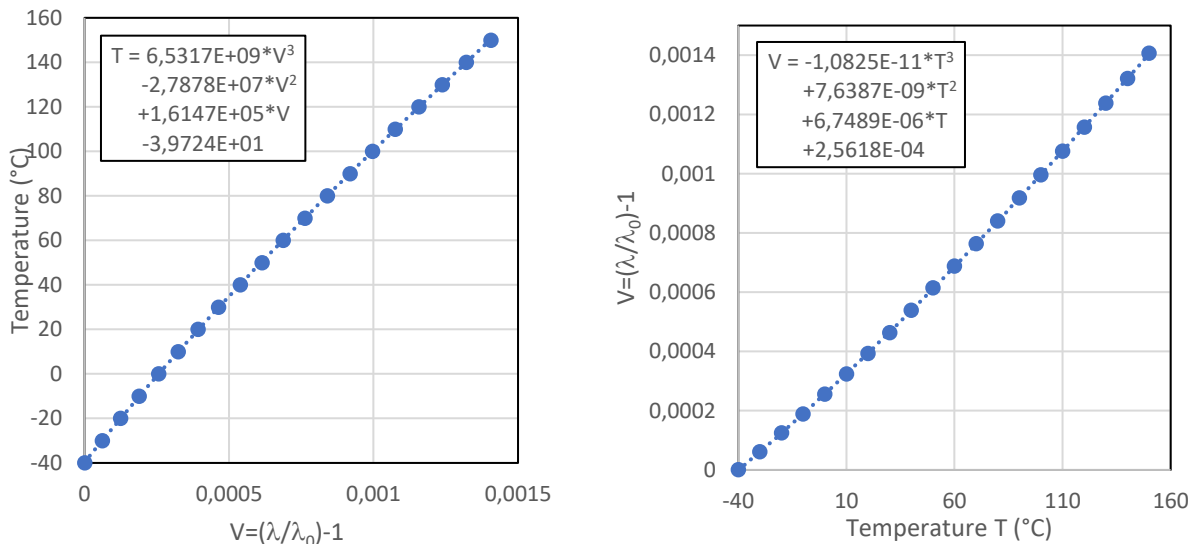


Fig. 1 Exemplary experimentally derived T-V and its inverse V-T diagram for 800nm single mode fiber with fitted polynoms.

In Fig. 1, such measurement data with fitted cubic functions is shown. As can be seen, in reality the functions are not purely linear. The inverse function V-T allows for arbitrary zero temperatures during later measurements.

It is possible to store four complete sets (one set consisting of four T-V coefficients  $K_0...K_3$  and four V-T coefficients  $C_0...C_3$ ) of cubic coefficients in the microcontroller, thus enabling the user to freely



- switch the spectrometer to mode 1 and set the appropriate fiber type
- set different sensors temperatures  $T_{\text{set}}$  and measure the related temperature  $T_{\text{meas}}$  with the spectrometer
- calculate the correction values  $C$  for each FBG channel:

$$C = T_{\text{set}} - T_{\text{meas}}$$

- for each FBG channel: plot a  $C - T_{\text{meas}}$  diagram (with  $[T_{\text{meas}}]=^{\circ}\text{C}$  and  $[C]=\text{K}$ ).
- fit cubic polynoms (parameters TC\_0...TC\_3) to this diagrams:

$$C = TC\_0 + TC\_1 * T_{meas} + TC\_2 * T_{meas}^2 + TC\_3 * T_{meas}^3$$

This channel specific polynomial parameters can now be transmitted to the spectrometers and used in mode 2 for temperature correction.

To set the parameters, use the following command:

**EKFps, f, x, y, z>** "Einzelkanalfehlerpolynomparameter setzen" /  
"Set channel specific error correction polynomial parameter"

f=fiber port number	f=0...3
x=peak detection channel	x=0...31
y=parameter number	y=0...3 (e.g., 3 for parameter TV_3)
z=parameter value TC_y	floating point value
Sets the channel specific $T_{meas} \rightarrow C$ temperature correction parameter TC_y.	

To readout the currently used correction parameter values set in the device, use the following command:

**EKFParam?>** *"Einzelkanalfehlerparameter?" / "parameters of channel specific correction?"*

Get polynomial parameters of the channel specific error correction functions. Returns floating point numbers in a 32 bit array (like in `WLL>` command, for example). The parameter sets of the currently connected fiber ports (always four, regardless of how many fiber ports are physically available) are transmitted successively, each one containing the four parameters `TC_y` (all numbers in 32 bit floating point format) for 32 FBG channels in a row. The exact order can also be seen in the LabView source code shown in Fig. 11.

Along with the single channel calibration a user specifiable calibration identification code may also be set and saved to Flash memory. This ID consists of four 32bit numbers that can be utilized to taste.

<b>KalID,w,x,y,z&gt;</b>	<i>"Kalibrierungs-ID setzen" / "Set calibration ID"</i>
--------------------------	---

w, x, y, z = ID numbers                      w,x,y,z = int32 bit numbers  
Sets four user selectable ID numbers.

**KalID?>** *"Kalibrierungs-ID?" / "Calibration ID?"*

Get user selectable calibration ID numbers.

Answer: array of four 32 bit numbers followed by "Ende".



## 1.9 Onboard Edge-FBG measurements

### 1.9.1 Principles of Edge-FBG measurements

Incorporating three or four FBGs at the same fiber position ("sensor plane") enables one to measure the spatial curvature of the glass fiber at that point. By inscribing and measuring several of these sensor planes at different positions along a glass fiber, it is possible to reconstruct the spatial position of this fiber relative to its surrounding environment.

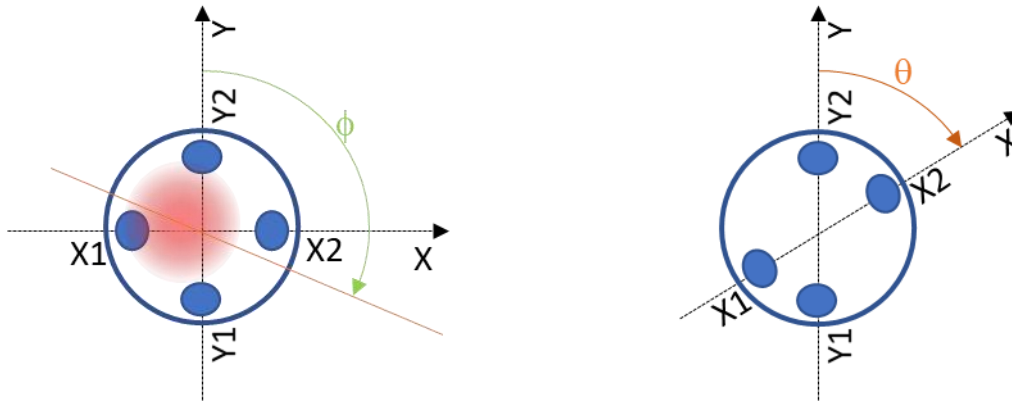


Fig. 2 (left): Cross section of a single mode fiber core with four FBGs (blue ellipses) near the borders (coordinates X1, X2 and Y1, Y2) and a slightly off-axis mode field (red) due to fiber curvature in  $\phi$ -direction.

Fig. 3 (right): Cross section of a sensor plane of ellipsoidally arranged FBGs with an ellipsoid angle  $\theta$  different to  $\pi$ .

In a straight fiber, the mode field of the broadband light guided through it is centered. However, if the fiber is curved at a certain point, the mode field there is shifted slightly off-axis, along the normal vector of the bending curve (the light is "trying to leave the fiber centrifugally" instead of following the bending). Fig. 2 shows this scenario, with four FBGs written near the fiber core's borders (thus the designation "Edge-FBG"), and the bending in  $\phi$ -direction resulting in a mode field shift to the upper left corner. In a straight fiber, both FBGs located at X1 and X2 coordinates would be irradiated by the same amount of light intensity; same goes for Y1 and Y2. However, in the shown case with the mode field being deflected, X1 receives more light than X2, and Y2 receives more light than Y1.

So by detecting the intensity ratios of  $I_{X1}/I_{X2}$  and  $I_{Y1}/I_{Y2}$ , it is possible to measure both bending radius R and bending direction angle  $\phi$ .

The calculation principle is more exhaustively described in the paper of Waltermann et al.<sup>1</sup> In that paper a three FBG approach is discussed, which in principle is already sufficient for 3D measurements. However, by using four FBGs like shown in Fig. 2, polarization effects and light source intensity fluctuations can be further eliminated.

During FBG production it might happen that the two FBG pairs X1/X2 and Y1/Y2 are not arranged perfectly rectangular to each other, like shown in Fig. 3. In this case, the angle  $\theta$  between x- and y-axis is not exactly rectangular ( $\theta=\pi/2$ ) anymore, as would be the case with perfect Edge-FBGs. To overcome this issue, the onboard calculation can use experimentally derived angle values for compensation. This angle can for example be determined for each sensor plane by measuring the phase shift between both x and y portion of the measured R and  $\phi$  signal while rotating the fiber that is bent by a constant radius (see, for example, also Fig. 11 and 12 at Waltermann et al.<sup>1</sup>).

<sup>1</sup> C.Waltermann et al., Multiple off-axis fiber Bragg gratings for 3D shape sensing, Applied Optics Vol. 57, Issue 28, pp. 8125-8133 (2018).

## 1.9.2 Variables and channel sorting conventions

The variables used in this chapter are the following:

### Coordinates and angles:

$X1/X2/Y1/Y2[\mu\text{m}]$	Coordinates of the four FBGs in one sensor plane (see Fig. 2)
$\theta$ [rad]	ellipsoidal angle between x- and y-axis of a sensor plane (see Fig. 3)
$R[\text{mm}]$	measurement result: measured bending radius at the sensor plane
$\phi$ [rad]	measurement result: bending direction relative to the X/Y coordinate system (see Fig. 2)
$\phi_0$ [rad]	angle $\phi$ measured and internally saved during "define direction" process

### Fiber parameters (one set of values per fiber port):

NA	numerical aperture of the glass fiber
waveguide radius $r$ [ $\mu\text{m}$ ]	mode field diameter of the glass fiber at about 850nm wavelength
Offset-to-Radius const. C	Constant used to convert the measured mode field displacement into the fiber's bending radius

### Sensor plane arrangement

A fixed number of four FBGs form a sensor plane. It does not matter if the FBG channels are defined as temperature or strain, and as zeroing wavelengths also have no impact on the edge measurements as well.

Per definition, this FBG channels and sensor planes are to be arranged in the FiSpec in a certain order:

FBG channel nr.	0	1	2	3	4	5	7	6	8	...
FBG at coordinate	$X1_0$	$X2_0$	$Y1_0$	$Y2_0$	$X1_0$	$X2_1$	$Y1_1$	$Y2_1$	$X1_3$	...
	Sensor plane 0				Sensor plane 1				...	

FiSens produced Edge-FBG sensor fibers already have the wavelengths fitting this scheme; so in this case a simple automatic peak search will result in the correct order. However, when using other layouts it is possible to change the wavelengths at any time, as long as the above mentioned channel order is followed.

As the maximum number of FBG detection channels in a FiSpec device is 32 (channel 0...31), a maximum of eight sensor planes (0...7) is possible.

### 1.9.3 Setting up Edge FBG measurements

To measure the three-dimensional shape of a glass fiber, the following sequence can be used:

On a new device, firstly the basic numbers have to be set:

- Set same edge-FBG coordinates for all sensor planes (command `EaKs, X1, X2, Y1, Y2>`). To start, use the following numbers:
  - $X1=Y1=+2\mu\text{m}$
  - $X2=Y2=-2\mu\text{m}$
- Set fiber parameters for Edge-calculations (command `EFPs, x, y, z, f>`). These are to be set for every single fiber port. For example:
  - Fiber NA=0.1,
  - waveguide radius  $r=2.9$
  - offset-to-radius-constant  $C=1700$ .
- If the device is newly updated from an older firmware, all  $\phi_0$  have to be zeroed (command `EaPhi0N>`); else the E> command will not transmit any R and  $\phi$ -numbers.
- For better results, the channel width can be set to  $\pm 3$  pixels (command `EKBs, x>`)
- If no details about the fibers are known, the ellipsoid angles at all fiber ports should be set to  $\theta=\pi/2$  (command `EaEs, x, f>`, each fiber port separately).

After this preparations, which only have to be done once, measurements can start:

- Enable onboard Edge-FBG calculations (command `EBs, x>`)
- Observe the spectra of the fiber ports and choose sensible integration time and averages. This settings should be used in future, although slight differences in background caused by different integration time are corrected for by the microcontroller.
- Disconnect all fibers from all fiber ports
- Get background (command `EUm>`); this command works for all fiber ports at the same time
- connect fibers again

Now, edge measurements can be started by constantly sending the E> command and reading out the data. Zeroing is necessary before measurements and has to be done for every fiber separately:

- Lay the sensor fiber straight on an even surface and after some time send the "straight sensor" command (command `EOBN, f>`) for each fiber. This saves the zero intensities of every FBG.
- Coil the sensor fiber and send the "define direction" command (`EOBRN, f>`) for each fiber. In this way, the zero angle  $\phi_0$  is saved for every sensor plane and all sensor planes are aligned radially towards each other.
- After everything is set up, save the settings to EEPROM (command `speichern>`).

Now the edge measurements should work, even after a power cycle. So after powering the device send the `EBs, 1>` command once, and then start measuring by sending the E> command repeatedly (occasionally zeroing may be needed depending on the circumstances).

#### 1.9.4 Onboard Edge measurement commands

For using onboard Edge FBG measurements, the following commands are used:

**EBs , x>** *"Edgeberechnungen schalten" / "Enable/Disable Edge-FBG calculations"*

x=0/1

Enables (x=1) or disables (x=0) onboard edge calculations. In general, enabled Edge mode does not affect normal measurements. However, for very fast measurement rates enabled Edge-mode may result in delays, even if no E> commands are sent. If a high measurement rate is of concern while doing Edge-FBG measurements, one may consider lowering the Edge-channel width (command EKBS , x>).

**EFPs , x , y , z , f>** *"Edge: Faserparameter setzen" / "Set Set fiber parameters for edge-calculations"*

x=fiber NA\*1000 x>0

y=waveguide radius[μm]\*1000 x>0

z=Offset-Radius-Parameter z>0

f=fiber port number f=0...3

Sets the fiber port specific fiber parameters.

**EaKs , X1 , X2 , Y1 , Y2>** *"Edge: alle Koordinaten setzen" / "Set same edge-FBG coordinates for all sensor planes"*

X1=X1[μm]\*1000

X2= X2[μm]\*1000

Y1= Y1[μm]\*1000

Y2= Y2[μm]\*1000

Sets the same set of coordinates (see Fig. 2) for all sensor planes of all fiber ports.

**EXKs , f , x , X1 , X2>** *"Edge: x- Koordinaten setzen " / " Set pair of x coordinates"*

f=fiber port number f=0...3

x=sensor plane number x=0...7

X1=X1[μm]\*1000

X2=X2[μm]\*1000

Set Edge-FBG coordinates (X pair, see Fig. 2) for a single sensor plane of a specific fiber port.

**EYKs , f , x , X1 , X2>** *"Edge: y- Koordinaten setzen" / "Set pair of y coordinates"*

f=fiber port number f=0...3

x=sensor plane number x=0...7

Y1=Y1[μm]\*1000

Y2=Y2[μm]\*1000

Set Edge-FBG coordinates (Y pair, see Fig. 2) for a single sensor plane of a specific fiber port.

**EOBN , f>** *"Edge: onboard-Nullen" / "Zero edge sensors (straight sensor)"*

f=fiber port number f=0...3

Set the actual FBG intensities of a specific fiber port as zero intensities for Edge calculations (to be done with a straight sensor fiber).

**EOBRN, f>** *"Edge: onboard-Richtung nullen" / "Zero edge sensors direction (coiled sensor)"*

f=fiber port number f=0...3

Set the actual sensor plane angles  $\phi$  of a specific fiber port as zero angles  $\phi_0$  for Edge calculations (to be done with a coiled sensor fiber).

**EaPhi0N>** *"Edge: alle  $\phi_0$  nullsetzen" / "Zero all  $\phi_0$ "*

Set all zero angles  $\phi_0$  to "0". Needed once after a firmware update from older versions with undefined values in EEPROM.

**EKBs, x>** *"Edge: Kanalbreite setzen" / "Set channel width"*

x=half channel width x=1...8

Set channel width (x points to the left and right, starting from the absolute maximum in the respective peak detection channel) for Edge amplitude calculation. The intensity maximum that is used for the edge calculation then is calculated out of a curve fit comprising these data points. Beware that setting this channel width to high values may result in delays for very fast measurement rates.

**EES, f, x, y>** *"Edge: Ellipsenwinkel  $\theta$  setzen" / "Set ellipsoid angle  $\theta$ "*

f=fiber port number f=0...3

x=sensor plane number x=0...7

y= $\theta \cdot 10,000$  y= $-2\pi \cdot 10,000 \dots + 2\pi \cdot 10,000$

Set ellipsoid angle  $\theta$  for a single sensor plane of a specific fiber port.

**EaEs, x, f>** *"Edge: alle Ellipsenwinkel setzen" / "Set all ellipsoid angles"*

x= $\theta \cdot 10,000$  x= $-2\pi \cdot 10,000 \dots + 2\pi \cdot 10,000$

f=fiber port number f=0...3

Set the same ellipsoid angle  $\theta$  for all sensor planes of a specific fiber port.

**EUm>** *"Edge: Untergrund messen" / "Measure background spectrum"*

Measure background spectra of all fiber ports. Bear in mind to disconnect all fibers from the device and set the desired integration time before. Curve fits of spectra derived in this way are subtracted from the measured amplitudes before they are used in the onboard edge calculations.

**EK?>** *"Edge: Koordinaten?" / "Get all Edge-FBG coordinates"*

Answer: Formatted like described in the WLL> command (see chapter 1.2.5). The transmitted data consists of an array of int32\_t numbers, representing the coordinate[ $\mu\text{m}$ ]\*1000 of the sensor plane FBGs.

The order is as following: X1<sub>0</sub>/X2<sub>0</sub>/Y1<sub>0</sub>/Y2<sub>0</sub>/ X1<sub>1</sub>/X2<sub>1</sub>/Y1<sub>1</sub>/Y2<sub>1</sub>/... X1<sub>7</sub>/X2<sub>7</sub>/Y1<sub>7</sub>/Y2<sub>7</sub>

For four-fiber port devices, coordinates of the FBGs at Fiber ports 1...3 are appended accordingly. At last, the array is followed by "Ende".

**EE?>** "Edge: Ellipsenwinkel  $\theta?$ " / "Get all ellipsoid angles  $\theta$ "

Answer: Formatted like described in the WLL> command (see chapter 1.2.5). The transmitted data consists of an array of int32\_t numbers, representing the angle  $\theta[\text{rad}] \cdot 10,000$  between the sensor plane's x-and y-axis (see Fig. 3).

The order is as following:  $\theta_0 / \theta_1 / \dots \theta_7$

For four-fiber port devices, the angles of the sensor planes at the other Fiber ports 1...3 are appended accordingly. At last, the array is followed by "Ende".

**EFP?>** "Edge: Faserparameter?" / "Get fiber parameters"

Answer: Formatted like described in the WLL> command (see chapter 1.2.5). The transmitted data consists of an array of int32\_t numbers, representing the numerical aperture  $NA \cdot 1000$ , waveguide radius  $r[\mu\text{m}] \cdot 1000$ , and Offset-to-Radius-constant  $C \cdot 1000$ .

The order is as following:  $NA/r/C$

For four-fiber port devices, the fiber parameters of the fibers connected to Fiber ports 1...3 are appended accordingly. At last, the array is followed by "Ende".

**E>** "Edge: Ergebnisse ausgeben" / "Get edge-FBG results"

Requests last measured Edge results (angle/radius); equivalent to the P>- command.

Answer: number of sensor planes N per fiber port (16bit integer numbers), followed by detected edge-FBG measurement results (angle  $\phi[\text{rad}] \cdot 10,000$  and radius  $R[\text{mm}] \cdot 10,000$ , both being 32bit numbers), error codes and "Ende". In Fig. 6, this is shown for the LabView programming language.

Order:  $N_0 / N_1 / N_2 / N_3 / R_0 / \phi_0 / R_1 / \phi_1 / \dots R_{NO} / \phi_{NO} \dots 6 \times 32\text{bit} \text{ "Ende"}$

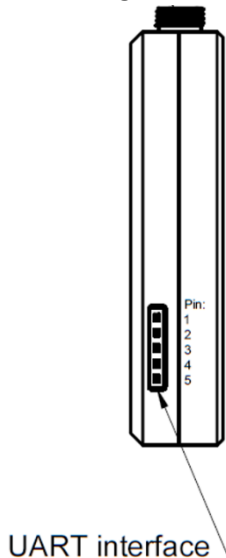
Number of Sensor planes	results $R/\phi$ fiber port 0	$R/\phi$ fiber ports 1...3 (if applicable)	Error code
----------------------------	----------------------------------	--	------------

The error code is the same like derived by the e?> command (see chapter 1.5).

## 1.10 Streaming modes, optional External UART

### 1.10.1 Basic UART communication parameters

As an option, the FiSpec interrogators may be equipped with an external UART connector. In principle, the microcontroller listens for the whole command set on both USB and external UART. Also, the answers will be sent simultaneously to both ones with exactly the same protocol, with the external UART being able to use different Baudrates, whereas the USB FTDI chip always stays fixed at 3 MBd.



Pin No.	Signal
1	+5V
2	+3.3V (output)
3	GND
4	RX
5	TX

Fig. 4 UART interface: physical layout.

Table 5 UART interface: electrical layout.

The details of the UART connector can be seen in Fig. 4 and Table 5. Pin 1 is internally connected to the 5V-Pin of the USB connector; so if no USB is connected, a power source has to be connected to this pin. An internal voltage regulator converts this 5V to 3.3V that powers the microcontroller. This 3.3V rail is present at Pin 2 (output only); so external devices connected by the user may also be powered by this pin (up to 200mA).

The serial communication uses 3.3V levels. Beware that the pins (except of the 5V power pin) are **not 5V tolerant** and no further ESD protection is applied!

**Connecting 5V levels at any pin except of Pin 1 will destroy the microcontroller!**

To set the external Baudrate, use the following command:

**extBdR, x>** "externe Baudrate / set Baud rate of external UART connector"  
x= Baudrate (Bit/s) (e.g. 2400, 9600, 28800, 57600, 115200, 460800, 921600, 3000000)

Sets the Baudrate of the optional external UART connector. The other parameters are: 8 data bits, 1 stop bit, no termination character, no handshake. The Baudrate changes immediately after execution of this command.

When using significantly different Baudrates on both interfaces, there may be major differences on how long the data transmission of whole spectra or peak arrays lasts. In this case, the correct settings

have to be well considered. For example, the transmission of a whole spectrum may take about 2 seconds when using UART transfer rates of 9600 Bd.

From firmware 10.0 on, the answer to every command is sent only to the interface this command came from (USB or UART). In this way, it is for example possible to send single requests by USB to a device mounted in a UART stack together with several others without obstructing the ongoing UART communication.

### 1.10.2 Streaming mode

In the conventional **sequential case** as described in the previous sections (every spectrum or peak array is individually requested for by the remote computer) it is in the responsibility of the remote control device sending the next request not until it received the whole requested data set (i.e. until the „Ende“ sign). As soon as the microcontroller receives a new *s>* or *P>* command, it interrupts the transmission and starts a new one. **So when using *s>* and *P>* commands, the requesting side (regardless if connected on external UART or USB) has to make sure it received the whole data before sending a new request.**

In **streaming mode**, however, it is not necessary to send single requests; the microcontroller automatically sends **new data as soon as the transmission is fully accomplished** and new data is available. Streaming mode is available both with USB and UART operation.

To switch between streaming and sequential mode, the following command may be used:

**DauSe, x>**                      *"Dauersenden / set Streaming mode"*

x=0: Sequential mode; *s>* and *P>* command may be used. (Standard setting)

x=1: Peak arrays (like the ones resulting from *P>* command) are transmitted endlessly until another *DauSe, x>* or *o>* command is sent to the FiSpec device

x=2: Complete spectra (as following *s, 1>* commands) are transmitted endlessly until another *DauSe, x>* or *o>* command is sent

The data stream will be sent to the interface that sent the command initiating the stream (USB or UART). This target interface will be saved into EEPROM when using the *speichern>* command, so in case streaming mode is activated and saved, after repowering the data stream will go to the same interface.

### 1.11 Autostart mode, and Saving settings to Flash memory

Like already described in chapter 1.3, after connecting the spectrometer to a power source, it is necessary to start the internal measurement process (command *a>*) and switch the internal light source on (command *LED, x>*) prior to measurements. This can also be done automatically, i.e. if a instantly starting warm-up of the system (with the LED being a significant heat source) is desired. This naturally comes at the cost of higher power consumption from the very beginning on, however.

**AuSt, x>**                      *"Autostart schalten / Switch Autostart"*

x=0: no Autostart; measurements have to be initiated by *a>* and *LED, 1>* command

x=1: Autostart (has to be saved into Flash by the *speichern>* command to be effective)



**speichern>** *"Einstellungen speichern / Save settings to flash memory"*

Saves the current state (Autostart, streaming mode, peak channels, integration time, averages, zero values for onboard calculation, UART parameters, dark spectrum) to flash memory. **BEWARE:** no wear leveling is applied when saving, so this command is not to be used routinely, but only when really necessary. Each time the command is executed, an internal counter is incremented (value shown in "#Schreibzugriffe\_x" after p?> command).

**FiSens recommends not to exceed 50 saving processes in total during the lifetime of the device to avoid data corruption. Any "#Schreibzugriffe\_x" values higher than 51 will void warranty.**

When, besides of Autostart, streaming mode is also activated and both settings are saved into the Flash memory, it is possible to use the FiSpec as a completely self-sufficient system starting autonomously and sending a constant serial data stream without any external commands. As the zero channel values of the onboard calculations are also being saved, a certain combination of FiSens interrogator and FBG sensor chain can then be used without further action (like setting of channels or zeroing) necessary as long as the system remains unchanged.

Beware, however, that automatically starting streaming at high data rates may flood your remote serial interface (depending on the programming techniques and hardware used) with data up to the point of unaddressability. So before saving the combination of Autostart and streaming mode to flash memory, it is wise to test streaming mode beforehand in the planned manner. In case of problems, it can be useful to just send `DauSe, 0>` commands several times while purging the i/o buffers to regain control.

## 2 Program code examples

### 2.1 Exemplary commands sequences for setting up onboard calculation

In this subchapter, a possible command sequence is explained to set up the spectrometers in order to measure wavelength or strain/temperature values with FBG sensors attached to a FiSpec device.

As initial zero values, the ones that are already known of the FBGs will be used, e.g. the wavelengths  $\lambda_0$  can be derived from the data sheet and the room temperature can be set as zero temperature  $T_0$ . If you plan on applying the OBN> "zero" command (i.e. all sensors are of known temperature while zeroing), the exact zero values are not important and should only be roughly fitting (i.e. inside the channel borders); with OBN> they will later be automatically set by the Spectrometer itself.

Now say, for example, you want to measure two FBGs:

- one FBG wavelength at 825nm, channel borders at +/-2nm, i.e. 823 and 827nm
- one FBG wavelength at 830nm, channel borders at +/-2nm, i.e. 828 and 832nm

Then you have to send the following commands:

Basic settings of the spectrometer:

```
LED,1>           //switch on the light source
iz,639>          //some integration time, for example 639µs
m,1>             //set averaging to "1"
a>              //start measurement
```

Now, the channels to be measured will be set:

```
Ke,0,8230000,8270000> //channel number 0, 825nm FBG
Ke,1,8280000,8320000> //channel number 1, 830nm FBG
KA,2>                //set the number of channels to be calculated
                        //and transmitted (here: two)
```

From now on, everything is set and the system is running. The results of the P> command now consist of wavelength (about 800...900nm) and intensity (about 2000...65000) values.

If you want to use the onboard calculation mode, however, you have to provide further information about the attached sensor:

```
OBsaT0,2100>      //set zero temperature T_0=21°C for all
                  //channels at once (if you plan to zero your
                  //FBGs at other temperatures than 21°C, this
                  //temperature should be set here; setting every
                  //FBG to its own different zero temperature can
                  //be done by the OBseT0,x,y> command)
OBsWL0,0,8250000>  //set zero wavelength 825nm for channel 0
OBsWL0,1,8300000>  //set zero wavelength 830nm for channel 1
OBsTyp,0,1>        //set channel 0 type as a temperature FBG
OBsTyp,1,1>        //set channel 1 type as a temperature FBG
OBB,1>            //activate on board calculation
```

Now the P>-command's answer will consist of the calculated strain (in µm/m) and temperature (in °C) values.

After ensuring that all FBG physically are at the zero temperature previously defined by the OBsaT0,x> command, the system has to be zeroed. After setting the channels there should be some

delay applied, so that a few measurements can be done whose measurement results can be used for zeroing purposes! Now zero:

```
OBN>                                     //the currently measured peak wavelengths of
                                     //the channels are from now on used as zero
                                     //wavelengths
```

Now, the peak information (values depending on the previously set on board calculation mode) can be derived by repeatedly sending the P> command:

```
P>                                     //ask for the latest calculated FBG
                                     //temperature/strain
```

Immediately after zeroing, while the FBG temperature and mechanical state has not changed yet, the output strain/temperature values should be near  $\pm 0 \mu\text{m}/\text{m}$  and  $T_0$ , respectively.

## 2.2 LabView program code

The following section shows some basic LabView code that can be used as a basis for own programs. This code is also included in the example program `FiSpec Data processing_LV2014.vi` that is delivered separately.

Fig. 5 shows how to process the data string the microcontroller transmits after the P> command. In case of the s> command, data has to be processed like shown in Fig. 7.

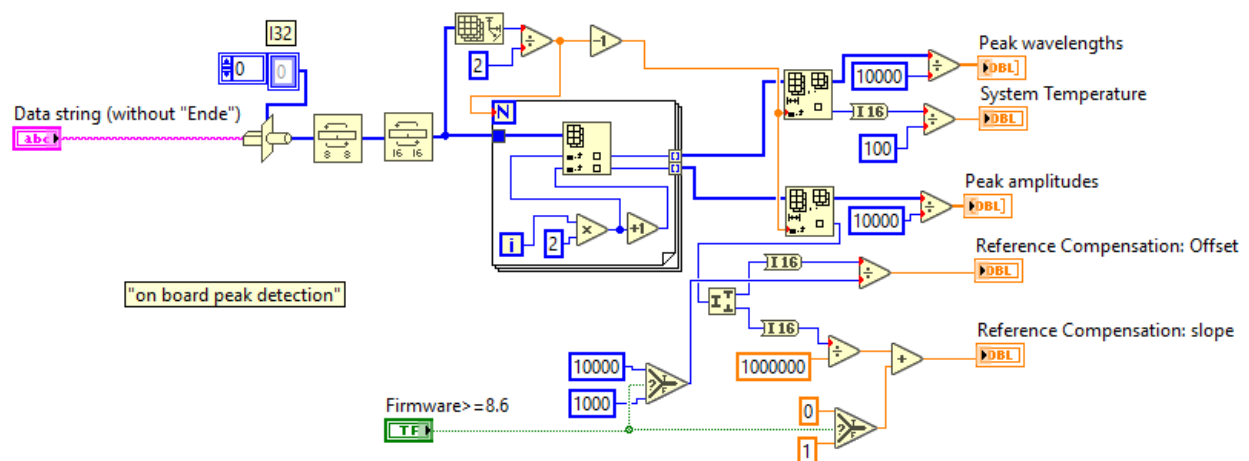


Fig. 5 LabView code: data processing for "on board peak detection" case.

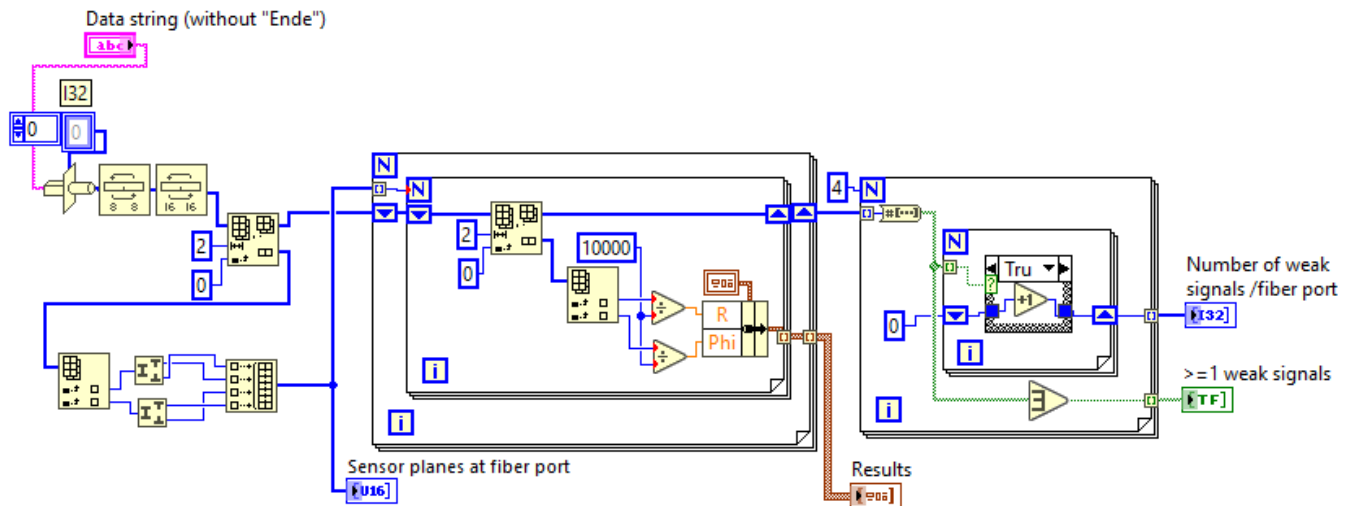


Fig. 6 LabView code: data processing for "on board edge measurements" case ( $E>$  command).

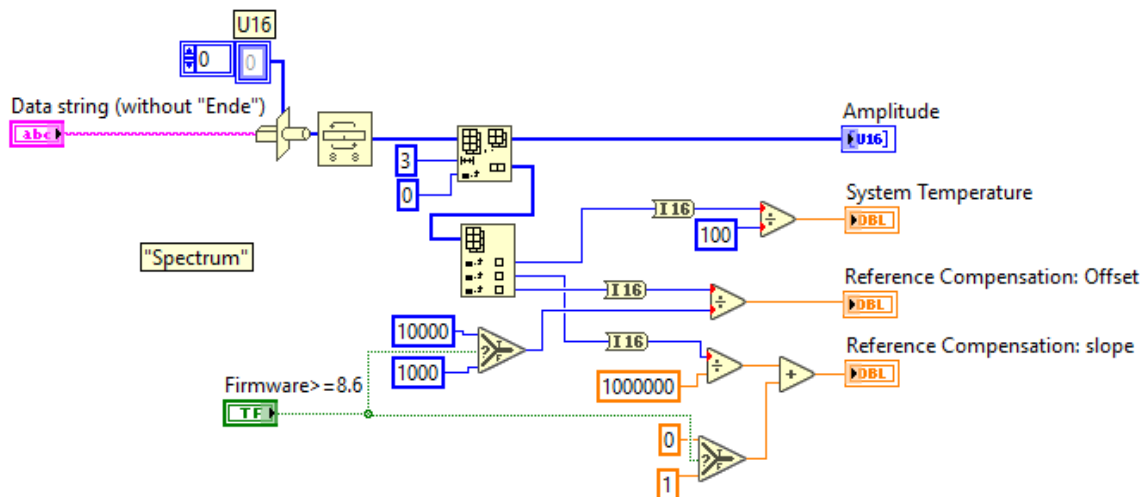


Fig. 7 LabView code: data processing for transmitted spectrum.

In Fig. 8 is shown how to get the signal quality information for each FG channel: four loop iterations covering four possible fiber ports (32bit for 32 peak detection channels). In the fifth 32bit number (lower loop), each byte represents one of the four fiber ports, while the first four bits of each byte indicate the four different error codes.

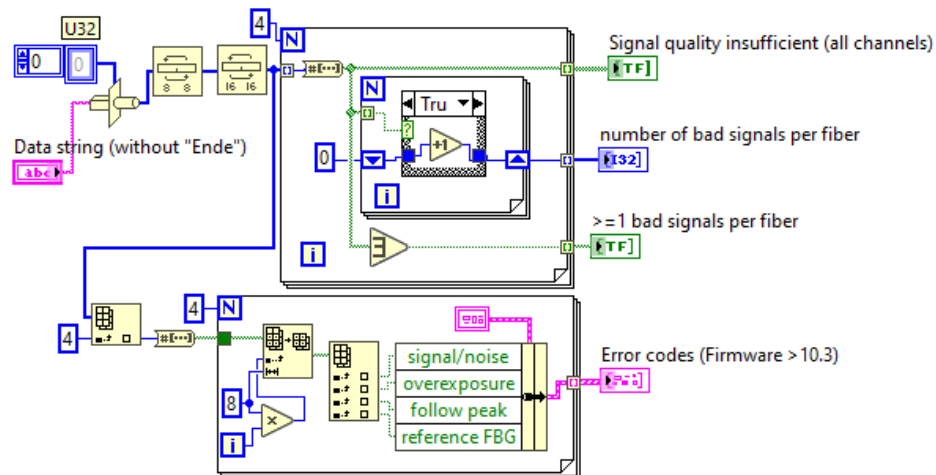


Fig. 8 LabView code: data processing for error code answers.

Fig. 9 and Fig. 10 show how to read or send single precision (i.e. 32bit) floating point numbers used in the cubic polynomial Temperature fit commands:

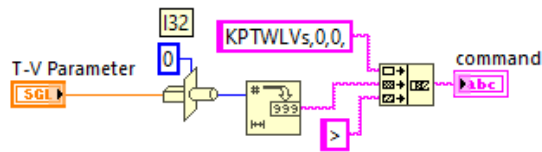


Fig. 9 LabView code: sending floating point numbers to the microcontroller (here: example for setting a cubic parameter).

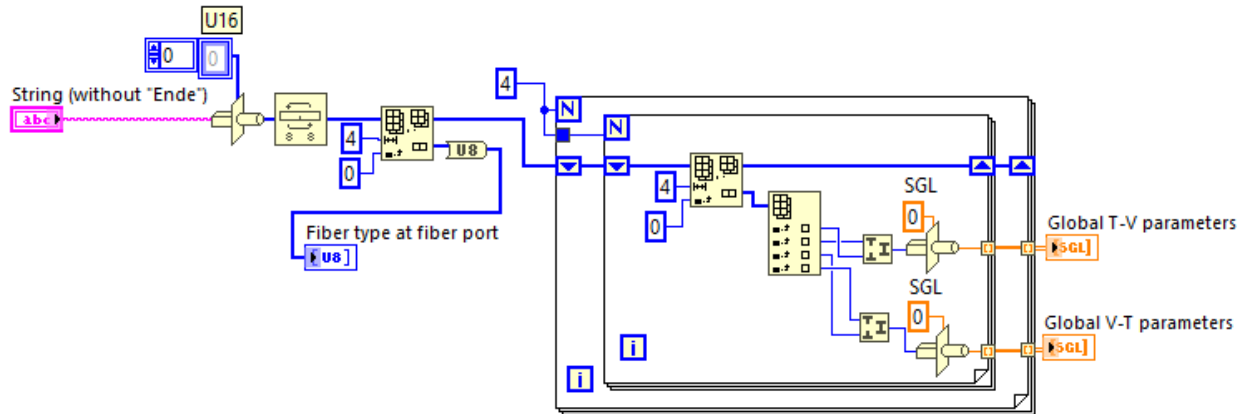


Fig. 10 LabView code: extracting floating point numbers from the microcontroller's answer string following the *KubParam?>* command (inner loop: four parameters, outer loop: four fiber types).

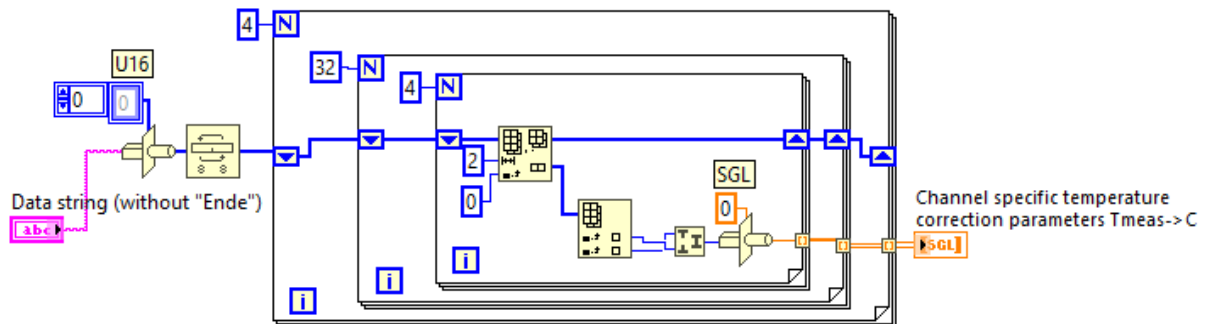


Fig. 11 LabView code: extracting floating point numbers following the *EKFParm?>* command. Inner loop: four parameters *TC<sub>y</sub>*; middle loop: 32 peak detection channels; outer loop: four fiber ports.

## 2.3 C program code

On the following pages you find an example C program that shows some simple communication via windows. It was tested to run with CodeBlocks and MinGW compiler.

```
/**
 * Copyright (c) 2020 FiSens GmbH
 * Permission is hereby granted, free of charge, to any person obtaining a copy of this software and asso-
 * ciated documentation files (the "Software"), to deal in the Software without restriction, including without
 * limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
 * the Software, and to permit persons to whom the Software is furnished to do so, subject to the following
 * conditions:
 * The above copyright notice and this permission notice shall be included in all copies or substantial
 * portions of the Software.
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
 * LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
 * EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
 * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
 * USE OR OTHER DEALINGS IN THE SOFTWARE.
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>

int main()
{
    /* Windows COM-Port */
    HANDLE comPort;
    DWORD read_write_length;
    int portNo = 23;
    char portName[32];
    int n;
    /* Windows COM-Port */

    /* Buffer for send/receive */
    char send[32];
    char receive[1024] = {0};
    char int_buf[12];
    /* Buffer for send/receive */

    /* Open COM-Port */
    sprintf(portName, "\\\\.\\COM%d", portNo);
    comPort = CreateFile(portName, GENERIC_READ|GENERIC_WRITE, 0, 0, OPEN_EXISTING, 0, 0);

    if(comPort == INVALID_HANDLE_VALUE)
    {
        printf("\terror: COM%d is not available. \n", portNo);
        return -1;
    }
    /* Open COM-Port */

    /* COM-Port configuration */
    DCB dcbSerialParameters = {0};
    dcbSerialParameters.DCBlength = sizeof(dcbSerialParameters);

    if(!GetCommState(comPort, &dcbSerialParameters))
    {
        printf("Error");
    }

    dcbSerialParameters.BaudRate = 3000000;
    dcbSerialParameters.ByteSize = 8;
    dcbSerialParameters.StopBits = ONESTOPBIT;
    dcbSerialParameters.Parity = NOPARITY;

    if(!SetCommState(comPort, &dcbSerialParameters))
    {
        printf("Unable to set serial port settings\n");
    }
    /* COM-Port configuration */

    /* Timeout configuration */
    COMMTIMEOUTS timeouts = {0};

    timeouts.ReadIntervalTimeout = 50;
    timeouts.ReadTotalTimeoutConstant = 50;
    timeouts.ReadTotalTimeoutMultiplier = 10;
    timeouts.WriteTotalTimeoutConstant = 50;
    timeouts.WriteTotalTimeoutMultiplier = 10;

    if(!SetCommTimeouts(comPort, &timeouts))
    {
        printf("Error setting timeouts\n");
    }
    /* Timeout configuration */
}
```

```

/* COM-Port is open */
printf("COM%d opened successfully\n", portNo);
/* COM-Port is open */

/* Connect to device */
_Bool device_available = 0;
while(!device_available)
{
    printf("Connecting...\n");
    /* Send command ">" */
    strcpy(send, ">");
    n = strlen(send);
    WriteFile(comPort, send, n, &read_write_length, 0);
    /* Send command ">" */

    /* Receive data */
    strset(receive, 0);
    Sleep(100);
    ReadFile(comPort, receive, sizeof(receive), &read_write_length, 0);
    /* Receive data */

    if(read_write_length > 0)
    {
        if(strncmp(receive, "FiSpec FBG X100", 15) == 0) {device_available = 1;}
        else if(strncmp(receive, "FiSpec FBG X150", 15) == 0) {device_available = 1;}
        else {device_available = 0;}
    }
}
printf("Connected to: %s", receive);
/* Connect to device */

/* Configuring the device */
/* Set active channels */
int FBG_count = 4;
int FBG_wavelength[4] = {8200000, 8300000, 8400000, 8500000};
int FBG_halfwidth = 15000;
for(int i=0; i<FBG_count; i++)
{
    strcpy(send, "Ke,");
    sprintf(int_buf, "%d", i);
    strcat(send, int_buf);
    strcat(send, ",");
    sprintf(int_buf, "%d", FBG_wavelength[i]-FBG_halfwidth);
    strcat(send, int_buf);
    strcat(send, ",");
    sprintf(int_buf, "%d", FBG_wavelength[i]+FBG_halfwidth);
    strcat(send, int_buf);
    strcat(send, ">");
    n = strlen(send);
    WriteFile(comPort, send, n, &read_write_length, 0);
    Sleep(10);
}
/* Set active channels */

/* Set quantity of active FBG */
strcpy(send, "KA,");
sprintf(int_buf, "%d", FBG_count);
strcat(send, int_buf);
strcat(send, ">");
n = strlen(send);
WriteFile(comPort, send, n, &read_write_length, 0);
Sleep(10);
/* Set quantity of active FBG */

/* Set integration time */
int integration_time = 50000;
strcpy(send, "iz,");
sprintf(int_buf, "%d", integration_time);
strcat(send, int_buf);
strcat(send, ">");
n = strlen(send);
WriteFile(comPort, send, n, &read_write_length, 0);
Sleep(10);
/* Set integration time */

/* LED on */
strcpy(send, "LED,1>");
n = strlen(send);
WriteFile(comPort, send, n, &read_write_length, 0);
Sleep(10);
/* LED on */

/* Starting internal measurements */
strcpy(send, "a>");
n = strlen(send);
WriteFile(comPort, send, n, &read_write_length, 0);
Sleep(10);
/* Starting internal measurements */
/* Configuring the device */

```

```

/* Repeat measurements */
int FBG_peak[4] = {8200000, 8300000, 8400000, 8500000};
int FBG_ampl[4] = {0, 0, 0, 0};
strcpy(send, "P>");
n = strlen(send);
while(1)
{
    /* Send command "P>" */
    WriteFile(comPort, send, n, &read_write_length, 0);
    Sleep(10);
    /* Send command "P>" */

    /* Receive data */
    strset(receive, 0);
    ReadFile(comPort, receive, sizeof(receive), &read_write_length, 0);
    /* Receive data */

    /* If end marker is recognized, display the data with timestamps */
    if(receive[read_write_length-4] == 'E' &&
        receive[read_write_length-3] == 'n' &&
        receive[read_write_length-2] == 'd' &&
        receive[read_write_length-1] == 'e')
    {
        printf("\n%d", (int)time(NULL));
        for(int i=0; i<FBG_count; i++)
        {
            FBG_peak[i] = (unsigned char)receive[8*i] + 256*(unsigned char)receive[8*i + 1] +
65536*(unsigned char)receive[8*i + 2] + 16777216*(unsigned char)receive[8*i + 3];
            FBG_ampl[i] = (unsigned char)receive[8*i + 4] + 256*(unsigned char)receive[8*i + 5] +
65536*(unsigned char)receive[8*i + 6] + 16777216*(unsigned char)receive[8*i + 7];
            printf(",%d,%d", FBG_peak[i], FBG_ampl[i]);
        }
        /* If end marker is recognized, display the data with timestamps */

        /* Wait twice the integration time (converted to ms) */
        Sleep(integration_time/500);
        /* Wait twice the integration time (converted to ms) */
    }
}
/* Repeat measurements */

/* Stop internal measurements */
strcpy(send, "O>");
n = strlen(send);
WriteFile(comPort, send, n, &read_write_length, 0);
Sleep(10);
/* Stop internal measurements */

/* LED off */
strcpy(send, "LED,O>");
n = strlen(send);
WriteFile(comPort, send, n, &read_write_length, 0);
Sleep(10);
/* LED off */

/* Close COM-Port */
CloseHandle(comPort);
/* Close COM-Port */

return 0;
}

```



## 2.4 Python program code

### 2.4.1 Python code overview and libraries

The following two sub-chapters contain the source code. These are also part of the FiSpec SDK. The code below was tested on both a desktop PC (Windows 11, Visual Studio Code) and with a Raspberry Pi (Raspbian, Thonny) and should be platform independent.

All libraries used by the developers are stored inside the 'requirements.txt' file which will be provided together with the python code.

The most important libraries for running the python code example are

- tkinter: generating the graphic user interface (GUI)
- matplotlib: creating graphs from data in list format
- pySerial: establishing a serial connection to a given COM-Port
- threading: running multiple tasks simultaneously

The user needs to install named libraries using the following commands in Windows command prompt.

```
python -m pip install tkinter
```

```
python -m pip install matplotlib
```

```
python -m pip install pySerial
```

The python code example consists of two files:

- main.py – sending commands, receiving measurement data, mathematical operations
- FiSpec\_GUI.py – Design of example GUI

To run the code first execute FiSpec\_GUI.py followed by main.py. If successful, the graphic user interface as shown below will pop up. Modification regarding the GUI can be achieved by editing FiSpecGUI.py.

#### **Side note – virtual environment:**

If different versions of the named libraries are used for other projects on target device the user can do so by creating virtual environments for those projects. After that it will be possible to run code with specific, project-dependent requirements.

A virtual environment can be created and controlled with the command prompt by using the following commands:

- 1) Navigate to the directory in which 'main.py' and 'FiSpec\_GUI.py' are located by using

```
cd [path to directory]
```

- 2) Create a virtual environment by using

```
python -m venv venv
```

which will create a virtual environment called 'venv' in the given directory. An additional folder called 'venv' should now be visible inside the directory.

- 3) Activate the virtual environment by running

```
venv\scripts\activate.bat
```

in case of success the activation is shown by '(venv)' at the beginning of a new line in windows command prompt.

- 4) Install the necessary libraries by running the command lines from above or install all at once by running

```
python -m pip install -r requirements.txt
```

- 5) If you want to add new libraries to your project you can do so by installing them and afterwards save them in requirements.txt by running

```
python -m pip freeze > requirements.txt
```

If you're using source control (for example GitHub) it will now be easier for other project participants to reconstruct the code to a running version by passing the 'requirements.txt' file together with the code.

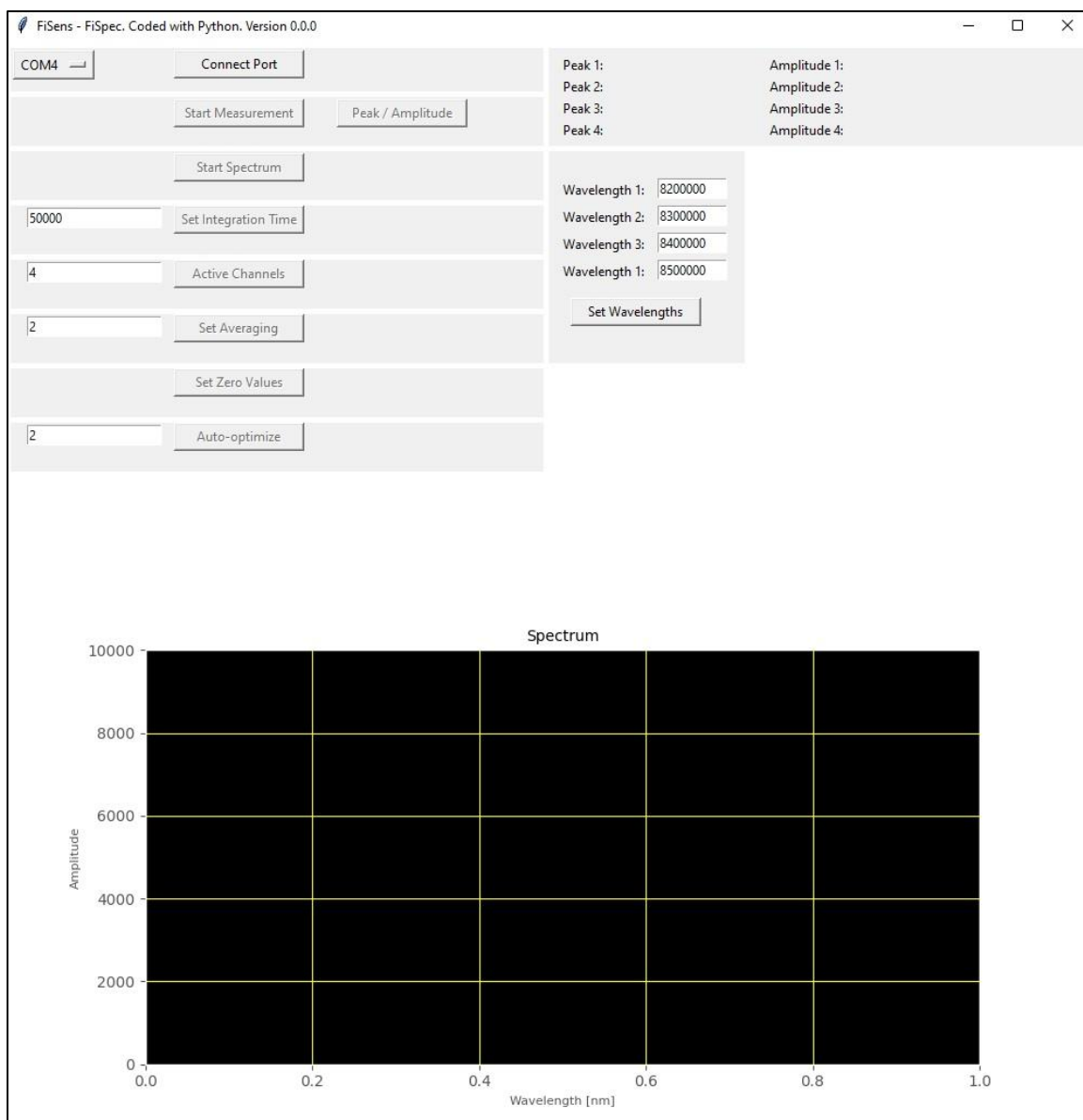


Fig. 12 GUI of the python example program.

After starting the program, the user has to pick a COM-Port by selection in the optionsmenu. If the “Connect Port”-button is pressed and if a FiSens device can be found under the given COM-Port the program connects to the device via the pySerial package. Upon start an object of the Serial class will be created. By initializing the attributes of this object the serial port can be configured, for example setting values for timeouts, stopbits, baudrate etc. If the program is successfully connected to a device the other buttons of the graphic user interface become enabled.

By using the Buttons and entry fields the user can try and test some fundamental functions from the FiSpec program. The table below shows all implemented commands in the python example code.

Command	Description	Button in GUI
Ke,x,y,z>	Peak detection details	Connect Port
LED,1>	Switches internal light source	Connect Port
WLL>	Get wavelength of pixel list	Connect Port
OBB,x>	Switch on board calculation	Peak / Amplitude – Temperature/Strain
a>	Globally start measurement	
iz,x>	Set integration time	Set Integration Time
KA,x>	Transmit channel number	Active Channels
m,x>	Set averaging	Set Averaging
OBN,x	Set zero values	Set Zero Values
AO,x>	Auto-optimization	Auto-Optimize
P>	Get measurement data	Start/Stop Measurement
s>	Get Spectrum	Start/Stop Spectrum

Table 6 Commands used in the Python example program.

## 2.4.2 Python file "main.py"

```
# FiSens FiSpec - main
# Version 0.3
# November, 2022
from tkinter import BOTH
import FiSpec_GUI as fsG
import serial.tools.list_ports
import threading
import matplotlib as plt
plt.use ('TkAgg')
from matplotlib.backends.backend_tkagg import (FigureCanvasTkAgg)
from matplotlib.figure import Figure
from matplotlib import style
import time

style.use("ggplot")

measIsOn=False # Flag which indicates if measurement is running.
specIsOn=False # Flag which indicates if spectrum is running.
pausedSpec=False # Flag which indicates if the spectrum is paused. While configuring the Spectrum should
not run.
internalMode0 = False # Flag which indicates which kind of measurement data will be provided
oldPort=""
xWll = [] # stores wavelength list (x-axis) of the plot
ySpec = [] # stores amplitude liste (y-axis) of the plot

fbg_count=4
fbg_wavelength=[8200000, 8300000, 8400000, 8500000]
fbg_halfwidth= 15000
bufferSize = 4096

def buildSpectrum():
    global fig, ax, canvas
    plotSize = (5, 3)
    fig = Figure(plotSize) # creating a figure which will contain the spectrum
    fig.tight_layout()
```

```

ax = fig.add_subplot(111) # adding the plot to fig
ax.set_title('Spectrum', fontsize=10)
ax.set_facecolor('black')
# ax.set_xlim(left=770, right=910)
# ax.set_ylim(bottom=0, top=10000)
ax.set_xlabel('Wavelength [nm]', fontsize=8)
ax.set_ylabel('Amplitude', fontsize=8)
ax.grid(color='yellow')

canvas = FigureCanvasTkAgg(fig, fsGl.frame_Plot) # creating a canvas object and embedd fig which will
contain the spectrum plot
# canvas.get_tk_widget().pack(fill=tk.BOTH, expand=1) #
canvas.get_tk_widget().pack(fill=BOTH, expand=1)
canvas.draw()

def sendrecv(command):
    try:
        ser.flushInput()
        ser.write(command.encode())
    except:
        print("Failed to send a message!")
        return ""
    try:
        response=ser.read(size=bufferSize)
    except:
        print("No response received on COM-Port!")
        return ""
    return response

def checkCOMs():
    global oldPort, measIsOn, specIsOn
    # check for available COM-Ports
    fsGl.checkCOMs()
    if fsGl.portObj.get()!=oldPort:
        try:
            ser.close()
        except:
            pass
        measIsOn=False
        specIsOn=False
        fsGl.disableButtons()
    elif ser.is_open:
        fsGl.enableButtons()
    else:
        measIsOn=False
        specIsOn=False
        fsGl.disableButtons()
    oldPort=fsGl.portObj.get()

def connectCOM():
    # Try to connect to selected COM Port and execute configurations
    global ser, oldPort
    dev_available=0
    ser.port = fsGl.portObj.get() # Get selection from Optionsmenu
    ser.baudrate = 3000000 # Configure COM-Port
    ser.bytesize = 8
    ser.parity = "N"
    ser.stopbits = 1
    ser.timeout = 0.5
    try:
        ser.open()
    except:
        print("Failed to open COM-Port!")
        return
    if ser.is_open:
        print("Successfully connected to device on COM-Port: " + ser.port)
        dev_dec = sendrecv(">")
        fsGl.enableButtons()
    try:
        print(dev_dec)
        if dev_dec == "FiSpec FBGX100":
            dev_available=1
            print("Connected to device: " + dev_dec)

        elif dev_dec == "FiSpec FBGX150":
            dev_available=1
            print("Connected to device: " + dev_dec)

        elif dev_dec == "FiSpec FBGX152":
            dev_available=1
            print("Connected to device: " + dev_dec)
        else:

```

```

        dev_available=0
        print("Error: No FiSens-device found!")
        # fsG1.disableButtons()
        # ser.close()
        # return
except:
    dev_available=0
    print("Error: No device found!")
    return

# Configure device after establishing a connection
# For further information on commands check FiSens FiSpec Programmer's manual
wlconfig() # "Ke,x,y,z>"
ledON() # "LED,x>"
checkWL() # "WLL>"
integration() # "iz,x>"
setAveraging() # "m,x>"
startInternal() # "a>"
setInternalMode() # "OBB,x>"

def wlconfig():
    # Set peak detection channel details. Use default values first
    # Can be configured by typing user specific wavelength numbers in entry fields
    global fsG1, specIsOn, pausedSpec

    if specIsOn == True: # while configuring the device, the spectrum should not be running
        specIsOn == False # pause the spectrum ...
        pausedSpec=True

    fbg_wavelength[0] = fsG1.setwl_1.get()
    fbg_wavelength[1] = fsG1.setwl_2.get()
    fbg_wavelength[2] = fsG1.setwl_3.get()
    fbg_wavelength[3] = fsG1.setwl_4.get()

    for x in range(len(fbg_wavelength)):
        ma1=int(fbg_wavelength[x])-fbg_halfwidth
        ma2=int(fbg_wavelength[x])+fbg_halfwidth
        conf_msg="Ke," + str(x) + "," + str(ma1) + "," + str(ma2) + ">"
        ser.flushInput()
        ser.write(conf_msg.encode())
        print("Device configured: " + conf_msg)
        time.sleep(0.5)

    if pausedSpec == True:
        specIsOn == True # ... restart the spectrum
        pausedSpec=False # pause ends here

def ledON():
    # Switches internal light source (0=off, 1= on)
    global specIsOn, pausedSpec
    if specIsOn == True:
        specIsOn == False
        pausedSpec=True
    try:
        ser.flushInput()
        ser.write("LED,1>".encode())
        print("Turn on internal LED")
    except:
        print("Error: LED")
        return
    time.sleep(0.5)

    if pausedSpec == True:
        specIsOn == True
        pausedSpec=False

def checkWL():
    # Get wavelength of pixel list - x-axis of spectrum
    global xWll
    try:
        wll_data=sendrecv("WLL>")
    except:
        print("Error: WLL>")
        return

    wll_data_len = len(wll_data)
    wll_data_len4=int(wll_data_len/4)

    for j in range(wll_data_len4-1):
        try:
            xWll.append((wll_data[4*j] + 256*wll_data[4*j+1] + 65536*wll_data[4*j+2] +
16777216*wll_data[4*j+3])/10000)

```

```

        except:
            pass

def setInternalMode():
    global internalMode0, specIsOn, pausedSpec

    if specIsOn == True:
        specIsOn == False
        pausedSpec=True

    if internalMode0 == False:
        try:
            ser.write("OBB,0>".encode())
            internalMode0 = True
            fsG1.toggleMeasMode_Bt.configure(text="Temperature / Strain")
        except:
            print("Error: Set internal Measurement Mode")
            return
        time.sleep(0.5)
    else:
        try:
            ser.write("OBB,1>".encode())
            internalMode0 = False
            fsG1.toggleMeasMode_Bt.configure(text="Peak / Amplitude")
        except:
            print("Error: Set internal Measurement Mode")
            pass
        time.sleep(0.5)

    if pausedSpec == True:
        specIsOn == True
        pausedSpec=False

def startInternal():
    # Globally start measurement
    global specIsOn, pausedSpec
    if specIsOn == True:
        specIsOn == False
        pausedSpec=True

    try:
        ser.write("a>".encode())
    except:
        print("Error: a> - Globally start measurement")
        return
    time.sleep(0.5)

    if pausedSpec == True:
        specIsOn == True # then start it again
        pausedSpec=False

def integration():
    # Set integration time
    global specIsOn, pausedSpec
    if specIsOn == True:
        specIsOn == False
        pausedSpec=True

    integration_time=fsG1.intT_In.get()
    try:
        intT_Value=int(integration_time)
    except:
        print("Error: Not a number!")
        return
    if type(intT_Value) == int:
        setIt_msg="iz," + str(integration_time) + ">"
        try:
            ser.write(setIt_msg.encode())
        except:
            print("Error: Set integration time")
            pass

    if pausedSpec == True:
        specIsOn == True # then start it again
        pausedSpec=False

def setChannel():
    # Set number of active Channels
    global specIsOn, pausedSpec
    if specIsOn == True:
        specIsOn == False
        pausedSpec=True

```

```

fbg_count=fsG1.setCh_In.get()
try:
    fbg_count_value=int(fbg_count)
except:
    print("Not a number!")
    return
if type(fbg_count_value)==int:
    setCh_msg="KA," + str(fbg_count_value) + ">"
    try:
        ser.write(setCh_msg.encode())
    except:
        print("Error: KA,x> - Set number of active channels")
        pass

if pausedSpec == True:
    specIsOn == True # then start it again
    pausedSpec=False

def setAveraging():
    #Set averaging
    global specIsOn, pausedSpec
    if specIsOn == True:
        specIsOn == False
        pausedSpec=True

    aver=fsG1.setAv_In.get()
    try:
        aver_value=int(aver)
    except:
        print("Not a number!")
        return
    if type(aver)==int:
        setAv_msg="m," + str(aver_value) + ">"
        try:
            ser.write(setAv_msg.encode())
        except:
            print("Error: Averaging")
            pass

    if pausedSpec == True:
        specIsOn == True # then start it again
        pausedSpec=False

def setZero():
    # Set actual values as zero values
    global specIsOn, pausedSpec
    if specIsOn == True:
        specIsOn == False
        pausedSpec=True

    setZe_msg="OBN,x>"
    try:
        ser.write(setZe_msg.encode())
    except:
        print("Error: Set Zero Values")
        pass

    if pausedSpec == True:
        specIsOn == True # then start it again
        pausedSpec=False

def setAutoOpt():
    # Start auto-optimization of integration time / averages
    # If successful, values in entry fields will be overwritten
    global specIsOn, pausedSpec
    if specIsOn == True:
        specIsOn == False
        pausedSpec=True

    vAO=fsG1.setAO_In.get()
    try:
        vAO_=int(vAO)
    except:
        print("Error: Not a number!")
        return
    if type(vAO_)==int:
        vAO_msg="AO," + str(vAO_) + ">"
        rcv_data=sendrecv(vAO_msg)
    try:
        intT_ao = rcv_data[0]
        aver_ao = rcv_data[1]

```

```

        intT_ao_str = str(intT_ao)
        aver_ao_str = str(aver_ao)

        print("Integration time optimized" + intT_ao_str)
        print("Average optimized" + aver_ao_str)
    except:
        print("Error: Auto-optimization")
        return
    val_intT_ao = "".join([i for i in intT_ao_str if i.isdigit()])
    val_aver_ao = "".join([i for i in aver_ao_str if i.isdigit()])
    intT_ao = int(val_intT_ao)*1000
    print(str(val_intT_ao))
    print(str(val_aver_ao))
    fsG1.intT_In.delete(0, 6)
    fsG1.intT_In.insert(0, str(intT_ao))
    fsG1.setAv_In.delete(0, 6)
    fsG1.setAv_In.insert(0, val_aver_ao)

    if pausedSpec == True:
        specIsOn == True # then start it again
        pausedSpec=False

def measurement():
    # Requests data from device and displays the numeric values in form of labels - "P>"-Command
    global ser
    fbg_peak=[8200000, 8300000, 8400000, 8500000]
    fbg_ampl=[0, 0, 0, 0]
    fbg_strain=[-5000000, -50, -50, -50]
    fbg_temp=[-5000000, -50, -50, -50]
    while 1:
        if measIsOn == True:
            if ser.is_open:
                rcv_data=sendrecv("P>")
            else:
                print("COM-Port is closed. Try again!")

            if internalMode0 == True:
                # Received data represents Wavelength and Amplitude
                for i in range(fbg_count):
                    try:
                        fbg_peak[i] = (rcv_data[8*i] + 256*rcv_data[8*i+1] + 65536*rcv_data[8*i+2] +
16777216*rcv_data[8*i+3])/10000
                        fbg_ampl[i] = (rcv_data[8*i+4] + 256*rcv_data[8*i+5] + 65536*rcv_data[8*i+6] +
16777216*rcv_data[8*i+7])/10000
                    except:
                        print("Error: Measurement - No Data")
                        pass

                        fsG1.label_Peak1.configure(text="Peak 1: " + str(fbg_peak[0]))
                        fsG1.label_Peak2.configure(text="Peak 2: " + str(fbg_peak[1]))
                        fsG1.label_Peak3.configure(text="Peak 3: " + str(fbg_peak[2]))
                        fsG1.label_Peak4.configure(text="Peak 4: " + str(fbg_peak[3]))
                        fsG1.label_Ampl1.configure(text="Amplitude 1: " + str(fbg_ampl[0]))
                        fsG1.label_Ampl2.configure(text="Amplitude 2: " + str(fbg_ampl[1]))
                        fsG1.label_Ampl3.configure(text="Amplitude 3: " + str(fbg_ampl[2]))
                        fsG1.label_Ampl4.configure(text="Amplitude 4: " + str(fbg_ampl[3]))
                else:
                    # Received data represents Strain and Temperature
                    for i in range(fbg_count):
                        try:
                            strain_4bytes=[rcv_data[8*i], rcv_data[8*i+1], rcv_data[8*i+2], rcv_data[8*i+3]]
                            fbg_strain[i]=int.from_bytes(strain_4bytes, byteorder="little", signed=True) /
10000

                            temp_4bytes=[rcv_data[8*i+4], rcv_data[8*i+5], rcv_data[8*i+6], rcv_data[8*i+7]]
                            fbg_temp[i] = int.from_bytes(temp_4bytes, byteorder="little", signed=True) / 100
                        except:
                            print("Error: No Data")
                            pass

                            fsG1.label_Peak1.configure(text="Strain 1: " + str(fbg_strain[0]))
                            fsG1.label_Peak2.configure(text="Strain 2: " + str(fbg_strain[1]))
                            fsG1.label_Peak3.configure(text="Strain 3: " + str(fbg_strain[2]))
                            fsG1.label_Peak4.configure(text="Strain 4: " + str(fbg_strain[3]))
                            fsG1.label_Ampl1.configure(text="Temperature 1: " + str(fbg_temp[0]))
                            fsG1.label_Ampl2.configure(text="Temperature 2: " + str(fbg_temp[1]))
                            fsG1.label_Ampl3.configure(text="Temperature 3: " + str(fbg_temp[2]))
                            fsG1.label_Ampl4.configure(text="Temperature 4: " + str(fbg_temp[3]))

                    time.sleep(1)

def ctrlMeas():
    # Toggles Button Start Measurement and sets Variable measIsOn which enables/disables the function
    global measIsOn

```



```

if measIsOn:
    fsG1.meas_Bt.config(text='Start Measurement')
    measIsOn=False
else:
    fsG1.meas_Bt.config(text='Stop Measurement')
    measIsOn=True

def updateSpectrum():
    # Requests data from device and displays it as a spectrum - "s>"-Command
    global fig, ax, canvas, ySpec, xWll
    while 1:
        if specIsOn == True:
            if ser.is_open:
                try:
                    ySpec.clear()
                    spec_data=sendrecv("s>")
                    spec_data_len=len(spec_data)
                    spec_data_len2=int(spec_data_len/2)
                    for i in range(spec_data_len2):
                        try:
                            ySpec.append(spec_data[2*i]+256*spec_data[2*i+1])
                        except:
                            pass
                    fig.clear()
                    ax.clear()
                    ax = fig.add_subplot(111)
                    ax.set_title('Spectrum', fontsize=10)
                    ax.set_facecolor('black')
                    ax.set_xlabel('Wavelength [nm]', fontsize=8)
                    ax.set_ylabel('Amplitude', fontsize=8)
                    ax.grid(color='yellow')
                    try:
                        ax.set_ylim(bottom=0, top=30000)
                        # if len(xWll) < 2000:
                        #     checkWL()
                        if len(xWll) > len(ySpec):
                            # y and x List have to have same dimension
                            while(len(xWll) > len(ySpec)):
                                ySpec.append(0)
                                ax.plot(xWll, ySpec, c='green')
                        elif len(xWll) < len(ySpec):
                            while(len(xWll) < len(ySpec)):
                                ySpec.pop()
                                ax.plot(xWll, ySpec, c='green')
                        else:
                            ax.plot(xWll, ySpec, c='green')
                    except:
                        pass
                    canvas.get_tk_widget().pack(fill=BOTH, expand=1)
                    canvas.draw()
                except:
                    print("Error: Spectrum Data Request")
                    pass
            else:
                print("COM-Port is closed. Try again!")
                time.sleep(1)

def ctrlSpec():
    # Toggles Button Start Specrum and sets Variable specIsOn which enables/disables the function
    global specIsOn
    if specIsOn:
        fsG1.spec_Bt.config(text='Start Spectrum')
        specIsOn=False
    else:
        fsG1.spec_Bt.config(text='Stop Spectrum')
        specIsOn=True

def assignButtons():
    fsG1.refreshCOM_Bt.configure(command=checkCOMs) # assign the function to the button
    fsG1.connect_COM_Bt.configure(command=connectCOM)
    fsG1.setwl_Bt.configure(command=wlconfig)
    fsG1.toggleMeasMode_Bt.configure(command=setInternalMode)
    fsG1.intT_Bt.configure(command=integration)
    fsG1.setCh_Bt.configure(command=setChannel)
    fsG1.setAv_Bt.configure(command=setAveraging)
    fsG1.setZe_Bt.configure(command=setZero)
    fsG1.setAO_Bt.configure(command=setAutoOpt)
    fsG1.meas_Bt.configure(command=ctrlMeas)
    fsG1.spec_Bt.configure(command=ctrlSpec)

if __name__ == '__main__':
    ser=serial.Serial(timeout=30)

```

```

fsG1 = fsG.FiSpec_GUI() # creating an object of the FiSpec_GUI class
assignButtons()
fsG1.disableButtons() # Disable buttons on first launch
buildSpectrum() # prepare Spectrum
checkCOMs() # checking for available COM-Ports
# Create 2 threads which will do work simultaneously
t_measurement = threading.Thread(target=measurement, daemon=True)
t_measurement.start()
t_createSpectrum = threading.Thread(target=updateSpectrum, daemon=True)
t_createSpectrum.start()
fsG1.master.mainloop()

```

## 2.4.3 Python file "FiSpec\_GUI.py"

```

# FiSens FiSpec, GUI Class, Version 0.3 - Coded with Python. November, 2022
import tkinter as tk
from tkinter import Frame, StringVar, Label, Entry, Button, OptionMenu
from tkinter import DISABLED
from tkinter import NORMAL
import sys
import glob
import serial
import serial.tools.list_ports
# The GUI class. All tkinter widgets (Frames, Buttons, Labels, etc.) are defined as attributes.
# Also, positioning of those widgets is specified in this class
class FiSpec_GUI:
    def __init__(self):

        self.master=tk.Tk()
        self.master.configure(background="white")
        self.master.title("FiSens - FiSpec. Coded with Python. Version 0.3 - October 2022")
        self.master.geometry("1000x1000")

        # -----
        self.frame_COM_select = Frame(master=self.master)
        self.frame_COM_select.place(x=5,y=5, width=490, height=40)

        self.portObj = StringVar(master=self.master)
        self.portList = ["COM-Ports"]
        self.portObj.set(self.portList[0])
        self.drop_COM = OptionMenu(self.frame_COM_select, self.portObj, *self.portList)
        self.drop_COM.grid(row=0, column=0)

        self.refreshCOM_Bt = Button(self.frame_COM_select, text="Refresh List", font=('Helvetica
bold',10))
        self.refreshCOM_Bt.place(x=150, y=2, width=120)

        self.connect_COM_Bt = Button(self.frame_COM_select, text="Connect Port", font=('Helvetica
bold',10))
        self.connect_COM_Bt.place(x=300, y=2, width=120)
        # -----
        self.frame_start_meas = Frame(master=self.master)
        self.frame_start_meas.place(x=5,y=50, width=490, height=45)

        self.toggleMeasMode_Bt = Button(self.frame_start_meas, text="Peak / Amplitude", font=('Helvetica
bold',10))
        self.toggleMeasMode_Bt.place(x=300, y=2, width=120)

        self.meas_Bt = Button(self.frame_start_meas, text="Start Measurement", font=('Helvetica bold',10))
        self.meas_Bt.place(x=150, y=2, width=120)
        # -----
        self.frame_start_spec = Frame(master=self.master)
        self.frame_start_spec.place(x=5,y=100, width=490, height=45)

        self.spec_Bt = Button(self.frame_start_spec, text="Start Spectrum", font=('Helvetica bold',10))
        self.spec_Bt.place(x=150, y=2, width=120)

        # -----
        self.frame_intgt = Frame(master=self.master)
        self.frame_intgt.place(x=5,y=150, width=490, height=45)

        self.intT_In = Entry(self.frame_intgt, width=20, font=('Helvetica bold',10))
        self.intT_In.place(x=15, y=2)
        self.intT_In.insert(0, "50000")

        self.intT_Bt = Button(self.frame_intgt, text="Set Integration Time", font=('Helvetica bold',10))
        self.intT_Bt.place(x=150, y=0, width=120)
        # -----
        self.frame_setCh = Frame(master=self.master)

```

```

self.frame_setCh.place(x=5,y=200, width=490, height=45)

self.setCh_In = Entry(self.frame_setCh, width=20, font=('Helvetica bold',10))
self.setCh_In.place(x=15, y=2)
self.setCh_In.insert(0, "4")

self.setCh_Bt = Button(self.frame_setCh, text="Active Channels", font=('Helvetica bold',10))
self.setCh_Bt.place(x=150, y=0, width=120)
# -----
self.frame_setAv = Frame(master=self.master)
self.frame_setAv.place(x=5,y=250, width=490, height=45)

self.setAv_In = Entry(self.frame_setAv, width=20, font=('Helvetica bold',10))
self.setAv_In.place(x=15, y=2)
self.setAv_In.insert(0, "2")

self.setAv_Bt = Button(self.frame_setAv, text="Set Averaging", font=('Helvetica bold',10))
self.setAv_Bt.place(x=150, y=0, width=120)
# -----
self.frame_setZe = Frame(master=self.master)
self.frame_setZe.place(x=5,y=300, width=490, height=45)

self.setZe_Bt = Button(self.frame_setZe, text="Set Zero Values", font=('Helvetica bold',10))
self.setZe_Bt.place(x=150, y=0, width=120)
# -----
self.frame_autoO = Frame(master=self.master)
self.frame_autoO.place(x=5,y=350, width=490, height=45)

self.setAO_In = Entry(self.frame_autoO, width=20, font=('Helvetica bold',10))
self.setAO_In.place(x=15, y=2)
self.setAO_In.insert(0, "2")

self.setAO_Bt = Button(self.frame_autoO, text="Auto-optimize", font=('Helvetica bold',10))
self.setAO_Bt.place(x=150, y=0, width=120)
# -----
self.frame_Plot = Frame(master=self.master)
self.frame_Plot.place(x=5,y=500, width=990, height=495)

self.frame_Display_Data = Frame(master=self.master)
self.frame_Display_Data.place(x=500,y=5, width=495, height=90)

self.label_Peak1 = Label(self.frame_Display_Data, text="Peak 1: ", font=('Helvetica bold',10))
self.label_Peak1.place(x=10, y=5)
self.label_Peak2 = Label(self.frame_Display_Data, text="Peak 2: ", font=('Helvetica bold',10))
self.label_Peak2.place(x=10, y=25)
self.label_Peak3 = Label(self.frame_Display_Data, text="Peak 3: ", font=('Helvetica bold',10))
self.label_Peak3.place(x=10, y=45)
self.label_Peak4 = Label(self.frame_Display_Data, text="Peak 4: ", font=('Helvetica bold',10))
self.label_Peak4.place(x=10, y=65)

self.label_Ampl1 = Label(self.frame_Display_Data, text="Amplitude 1: ", font=('Helvetica
bold',10))
self.label_Ampl1.place(x=200, y=5)
self.label_Ampl2 = Label(self.frame_Display_Data, text="Amplitude 2: ", font=('Helvetica
bold',10))
self.label_Ampl2.place(x=200, y=25)
self.label_Ampl3 = Label(self.frame_Display_Data, text="Amplitude 3: ", font=('Helvetica
bold',10))
self.label_Ampl3.place(x=200, y=45)
self.label_Ampl4 = Label(self.frame_Display_Data, text="Amplitude 4: ", font=('Helvetica
bold',10))
self.label_Ampl4.place(x=200, y=65)
# -----
self.frame_wl = Frame(master=self.master)
self.frame_wl.place(x=500, y=100, width=180, height=195)

self.label_wl_1 = Label(self.frame_wl, text="Wavelength 1: ", font=('Helvetica bold',10))
self.label_wl_1.place(x=10, y=25)

self.setwl_1 = Entry(self.frame_wl, width=10, font=('Helvetica bold',10))
self.setwl_1.place(x=100, y=25)
self.setwl_1.insert(0, "8200000")

self.label_wl_2 = Label(self.frame_wl, text="Wavelength 2: ", font=('Helvetica bold',10))
self.label_wl_2.place(x=10, y=50)

self.setwl_2 = Entry(self.frame_wl, width=10, font=('Helvetica bold',10))
self.setwl_2.place(x=100, y=50)
self.setwl_2.insert(0, "8300000")

self.label_wl_3 = Label(self.frame_wl, text="Wavelength 3: ", font=('Helvetica bold',10))
self.label_wl_3.place(x=10, y=75)

```

```

self.setwl_3 = Entry(self.frame_wl, width=10, font=('Helvetica bold',10))
self.setwl_3.place(x=100, y=75)
self.setwl_3.insert(0, "8400000")

self.label_wl_4 = Label(self.frame_wl, text="Wavelength 4: ", font=('Helvetica bold',10))
self.label_wl_4.place(x=10, y=100)

self.setwl_4 = Entry(self.frame_wl, width=10, font=('Helvetica bold',10))
self.setwl_4.place(x=100, y=100)
self.setwl_4.insert(0, "8500000")

self.setwl_Bt = Button(self.frame_wl, text="Set Wavelengths", font=('Helvetica bold',10))
self.setwl_Bt.place(x=20, y=135, width=120)

def checkCOMs(self):
    # Updates the list of COM-Ports shown in the optionsmenu of the GUI
    # first destroy old widget and clear old list
    self.drop_COM.destroy()
    self.portList.clear()

    if sys.platform.startswith('win'):
        ports=serial.tools.list_ports.comports(include_links=True) #get available COM-Ports
    elif sys.platform.startswith('linux') or sys.platform.startswith('cygwin'):
        ports = glob.glob('/dev/tty[A-Za-z]*')
    elif sys.platform.startswith('darwin'):
        ports = glob.glob('/dev/tty.*')
    else:
        raise EnvironmentError('Unsupported platform')

    # and add them to the list
    for onePort in ports:
        if sys.platform.startswith('win'):
            self.portList.append(onePort.device)
        else:
            self.portList.append(onePort)
    # self.portObj.set(self.portList[0])
    # if no devices are found
    if len(self.portList) == 0:
        self.portList.append("No devices found!")
        self.portObj.set(self.portList[0])
        print(self.portList)
    # create new optionsmenu and place it in the GUI
    self.drop_COM = OptionMenu(self.frame_COM_select, self.portObj, *self.portList)
    self.drop_COM.grid(row=0, column=0)

def enableButtons(self):
    self.toggleMeasMode_Bt.configure(state=NORMAL)
    self.spec_Bt.configure(state=NORMAL)
    self.meas_Bt.configure(state=NORMAL)
    self.intT_Bt.configure(state=NORMAL)
    self.setCh_Bt.configure(state=NORMAL)
    self.setAv_Bt.configure(state=NORMAL)
    self.setAO_Bt.configure(state=NORMAL)
    self.setZe_Bt.configure(state=NORMAL)

def disableButtons(self):
    self.toggleMeasMode_Bt.configure(state=DISABLED)
    self.spec_Bt.configure(state=DISABLED, text='Start Spectrum', font=('Helvetica bold',10))
    self.meas_Bt.configure(state=DISABLED, text='Start Measurement', font=('Helvetica bold',10))
    self.intT_Bt.configure(state=DISABLED)
    self.setCh_Bt.configure(state=DISABLED)
    self.setAv_Bt.configure(state=DISABLED)
    self.setAO_Bt.configure(state=DISABLED)
    self.setZe_Bt.configure(state=DISABLED)

```

### 3 Firmware changelog

#### **v. 6.4**

- onboard "follow peak" function added.
- fixed corrupt spectra while changing integration time

#### **v. 6.5**

- added internal reference FBG measurement
- added reference compensation

#### **v. 6.6**

- changed "LED" command; LED current is fixed now. See chapter 1.3 for details.
- optimized reference compensation

#### **v. 6.7**

- optimized hardware parameters

#### **v. 6.8**

- changed internal reference compensation algorithm
- exchanged current wavelength compensation number by measured reference FBG wavelength (see chapter 1.2 for details) in transmitted data. As both versions use the same 32bit number, there is no need to change data readout unless this specific values was used before.
- changed thermoelastic constant from  $8.2E-6$  to  $8.65E-6$

#### **v 6.9**

- increased number of onboard channels from 24 to 32
- optimized channel border resolution
- added onboard FBG calculations (strain, temperature, zeroing)
- `P>` command now initiates transmission of either wavelength/amplitude like before (standard mode) or strain/temperature (in onboard calculation mode)
- optional external UART: Baudrate can be set
- optional external UART: amount of transmitted data can be chosen (nothing, only basic information, peak information, complete spectra). USB is still sending in parallel to the UART.
- streaming mode added: it is now possible to endlessly send values as soon as they are measured (and previous data transmission is completed) without the need of initiating this by `s>` and `P>` commands of the host computer
- in streaming mode, the microcontroller will wait until transmission of both UART and USB is finished
- eliminated the transmission of one supernumerous channel (containing meaningless numbers) between the results of the last peak detection channel and the temperature/reference FBG wavelength information in the `P>` command answer.
- added autostart switch, either immediately starting data streaming or just switching light source (for warm up) and internal measurements on after connecting to power source
- enabled to save current state (onboard calculation mode, onboard calculation channel details, zero values, UART settings, streaming mode, autostart mode) to microcontroller's flash memory. Beware of the maximum number of write processes described in chapter 1.11!

#### **v 7.0**

- added center-of-gravity onboard peak detection

- increased internal temperature sensor update frequency

#### **v 7.1**

- supports multi-fiber FiSpec systems
- pixel binning removed
- added periodic darkframe updates
- changed standard value of maximum follow peak value (before:  $0.4 \times$  channel width; now:  $0.25 \times$ )
- supports multiplexed systems
- readout of set channel border values possible
- restricted channel width to 200 pixels
- supports second reference FBG

#### **v 7.2**

- internal temperature sensor value averaging optimized
- compensation: introduced slope value transmission.  $\text{transmission} = (\text{value} - 1) \times 10^6$

#### **v 8.0**

- optimized periodic darkframe updates
- future firmware updates via USB possible

#### **v 8.1**

- optimized reference wavelength smoothing
- new experimental peak detection mode: spectral FIR-filtering
- fixed: from version 7.1 to 8.0 maximum follow peak channel shift value was not used

#### **v 8.2**

- in active reference compensation mode/deactivated follow peak mode: channel borders now shift according to reference FBG compensation offset
- fixed: microcontroller was no longer accessible after receiving more than 30 characters without termination character ">"

#### **v 8.3**

- thermoelastic constants (separate values for reference FBGs and sensor FBGs possible) and optoelastic constant can now be changed via serial commands.
- further smoothing of reference FBG signal

#### **v 8.4**

- in follow peak mode: channel width internally reduced to 60%, to prevent neighboring peaks interfering during significant temperature/strain changes. After switching follow peak mode off, the original channel width is being restored.
- reference FBG correction equation changed
- fixed: follow peak function (in ver. 8.3: not permanently active when desired so)
- some stability fixes
- automatically detecting channels in reference FBG channels range. Since now: when zeroing the channels,  $T_0$  is set to the device temperature.
- follow peak function optimized: the channel borders will follow the peak only when the signal quality is sufficient (preventing erroneous channel changes e.g. when no sensor fiber is attached).

#### **v8.5**

- changed internal reference compensation calculation; now also two reference FBG supported
- TEK and OEK now saved in flash memory.
- Protocol changed: setting TEK/OEK by sending value/1000 instead of value/10000
- added: sectionwise defined TEK values (up to 10 supporting points) for onboard calculations possible
- now negative values for  $T_0$  possible (-273.15°C...+327.0°C)

#### **v8.6**

- fixed: clipping of spectra (max. value: 51000) when using averaging > 1
- fixed: time constant very high when using averaging > 1
- protocol change in transferred peak / spectra arrays: offset\*10000 instead of offset\*1000

#### **v 8.7**

- code optimizations

#### **v8.8**

- added: setting TEK constant for every single fiber in multi-fiber-systems now possible

#### **v8.9**

- added: error code (fiber breakage / signal quality of each FBG channel) now calculated on board
- code optimizations

#### **v9.0**

- added: approximation of on board TEK value by cubic polynomial possible
- added: calibration ID can be saved in the device's flash memory

#### **v9.1**

- code optimizations

#### **v9.2**

- fixed (only applies to X1-devices): errors in `TeKPs>` command (cubic polynomial parameters)

#### **v9.3**

- added: onboard auto-optimize function
- fixed: corrupt spectra in UART mode 4 (used e.g. in multi-fiber-devices, since firmware v8.9)
- added: multiplex number can now be set and saved in flash memory for devices without physical ID pins
- optimized startup behaviour of internal temperature sensor's signal

#### **v9.4**

- fixed: commands OBseT0/OBsWL0/OBsTK/OBsTyp were ignored if respective channel number < overall channel number of first fiber
- optimized rounding of channel borders to nearest pixel
- added: new setting that defines if reference compensation shall be activated at device startup

#### **v9.5**

- replaced sectionwise TEK interpolation mode (mode 1) by global cubic polynomial TEK calculation
- global TEK interpolation parameters for up to four different fiber types can be stored in EEPROM

#### **v9.6**

- fixed: Auto-optimization algorithm sometimes did not converge for intense signals (i.e., X150), and answers contained invalid numbers for integration time / averages
- Auto-optimization further optimized
- signal quality error bits (`e?>` command) now also set at intense, clipping signals (intensity > 63000)
- in peak following mode, when error bit is set: channel borders now remain unaltered at their last valid position (before: reset them immediately to their original position like in "not-follow-peak case")

#### **v9.7**

- optimized: slight temperature errors occurring after zeroing if "TV" polynomial is not exactly the reverse function of the "VT" polynomial.

#### **v9.8**

- fixed: follow peak function sometimes ceased following when using narrow channels or small follow peak limits values (set by "PNg, x>" command)

#### **v9.9**

- general code optimizations
- fixed: `KalID>` command returned error code array instead of calibration number (only UART transmission affected)
- center of gravity peak detection mode: optimized accuracy if peaks are near the channel borders
- fixed: in center of gravity peak detection mode, peak amplitudes were slightly off since ver. 9.8. (peak position accuracy was not affected, though)
- fixed: with periodic darkframe update enabled and dark frame subtraction disabled, the darkframes were subtracted nonetheless (optimization of `ADCC, x>` and `ADBA, x, y, z>` commands)

#### **v10.0**

- fixed: since v9.3 devices do not set the ID pins correctly after saving EEPROM in stacked state
- increased internal measurement frequency from 220Hz to 300Hz
- saving streaming interface selection to EEPROM
- fixed: instabilities when channel borders nearing pixel number boundaries with "follow peak" active
- protocol changed: removed different UART transmission modes; instead now answers are sent to the interface that the respective command came from (USB or UART). Therefore:
  - `UARTse, x>` command obsolete
  - `#UARTModus_999` instead of `#UARTModus_1...4` in `p?>` command answer string
- fixed: in four-fiber port systems quickly setting many FBG channels at once resulted in corrupted channel borders
- general code optimizations

#### **v10.1**

- added: onboard Edge-FBG measurement functionality

#### **v10.2**

- For speed reasons, Edge calculations are now only executed when activated before (`EBs, x>` command added)



**v10.3**

- optimized linearization of peak detection (before: slight wavelength nonlinearities appeared when peaks crossed pixel borders)
- removed first derivative peak detection mode (command `PeM, x` > obsolete)
- supports devices with 1, 2, 3 and 4 fiber ports (before: only 1 and 4 fiber ports possible)

**v10.4**

- error bits are now also set in peak following mode if peak following is temporarily disabled (e.g. due to bad signal quality or extreme wavelength jumps)
- `e?` > command: fifth 32 bit number (until now reserved for future use) now contains more specific real time information about what kind of error (S/N ratio, over exposure, peak following, or reference FBG error) occurs at each fiber port in at least one peak detection channel.
- automatic periodic dark frame update (`ADBA, x, y, z` > command): for averages > 1 the measured dark spectra are now averaged like normal spectra (until now: only single, not averaged frames, regardless of active averaging setting) before being low pass filtered.
- internal threshold for error detection adjusted for reference FBG channels