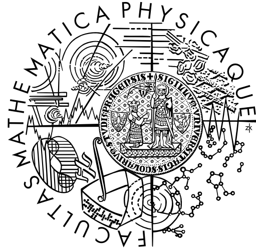

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Ivona Oboňová

Firmware for CzechLight optical measurement and calibration device

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Miroslav Kratochvíl
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, date 6.1.2020 signature of the author

I would like to thank my supervisors Miroslav Kratochvíl and Jan Kundrát, for their expertise, fast replies and their patience. I would also like to thank Jakub Vondráček, for his support and for proofreading this text.

Title: Firmware for CzechLight optical measurement and calibration device

Author: Ivona Oboňová

Department: Department of Software Engineering

Supervisor: RNDr. Miroslav Kratochvíl, Department of Software Engineering

Abstract: The goal of this thesis is to implement firmware for the Optical Measurement and Calibration Device, which was designed and constructed in CESNET. The purpose of the device is to simplify the calibration of various fibre-optical networking devices, used in CESNET infrastructure. The thesis includes an overview of the internal structure and communication interfaces within the device, which is than used for designing and implementing the firmware. The results are demonstrated on realistic hardware, by running the measurement on an existing optical component. The produced firmware will serve as a basis for the development of more advanced devices in CESNET.

Keywords: firmware low-level device drivers fibre optic communications networking

Contents

Introduction **3**

1 OMCD firmware structure and interfaces **7**
1.1 Serial communication interface 9
1.1.1 Serial port on Unix systems 10
1.2 Network Configuration Protocol 13
1.2.1 YANG 13

2 Implementation of the OMCD firmware **17**
2.1 Implemented drivers 17
2.1.1 Property Interface and Serial Communication 19
2.1.2 Driver Interface 20
2.1.3 Integrable Tunable Laser ITLA 20
2.1.4 Coaxial Fiber Optic Switch SERCALO 26
2.1.5 Optical Spectrum Analyzer AXSUN 27
2.2 YANG interface 31
2.3 High-level OMCD driver 31
2.4 Testing and demo versions 33

3 Results **35**
3.1 Self-calibration 35
3.2 Use case: Measuring ROADM 37

Conclusion **41**

List of Figures **47**

List of Tables **49**

Introduction

In optical fiber communication, the carrier of information through the optical system is optical signal. This optical signal is a light beam emitted from a transmitter and modulated with specific intensity. As the signal travels through the optical system, its energy dissipates along the way. This phenomenon is called *attenuation*.

Attenuation determines the loss of strength of signal within optical components. Attenuation is mainly caused by phenomena like scattering, absorption and leakage [Fib]. It is measured by comparing the input and output optical power. To measure the attenuation of some component, a light source or a transmitter (with a built-in light source) and power meter are needed. To measure attenuation of a component, there are two steps which must be followed. First the amount of power which is emitted must be determined. This can be known from the emitter (if the power is configurable), or obtained by connecting the light source directly to a power meter. Second, the component being tested is connected between the transmitter and the power monitor, and the power is measured again (see Fig. 1 for illustration). The attenuation is measured as the difference between these two values [Hec02]. Expectably, the second value should be lower than the first value and thus the difference should be positive, since attenuation is a loss of power. (Otherwise, we would be dealing with *amplification*.)

Generally, this process is called *calibration*. Calibration is the comparison of measured values obtained by a device under some tests. This is typically done to determine an error or verify the accuracy of the tested device [Aut]. The calibration process may continue with a corrective adjustment if the difference is large enough to cause concern. This is done for precision measurements in optical systems.

Goals Czech Education and Scientific NETwork (CESNET) designed and constructed Optical Measurement and Calibration Device (OMCD), with the purpose of simplifying the calibration of optical components and other optical devices. The OMCD consists of transmitter, Optical Spectrum Analyzer (OSA) and two

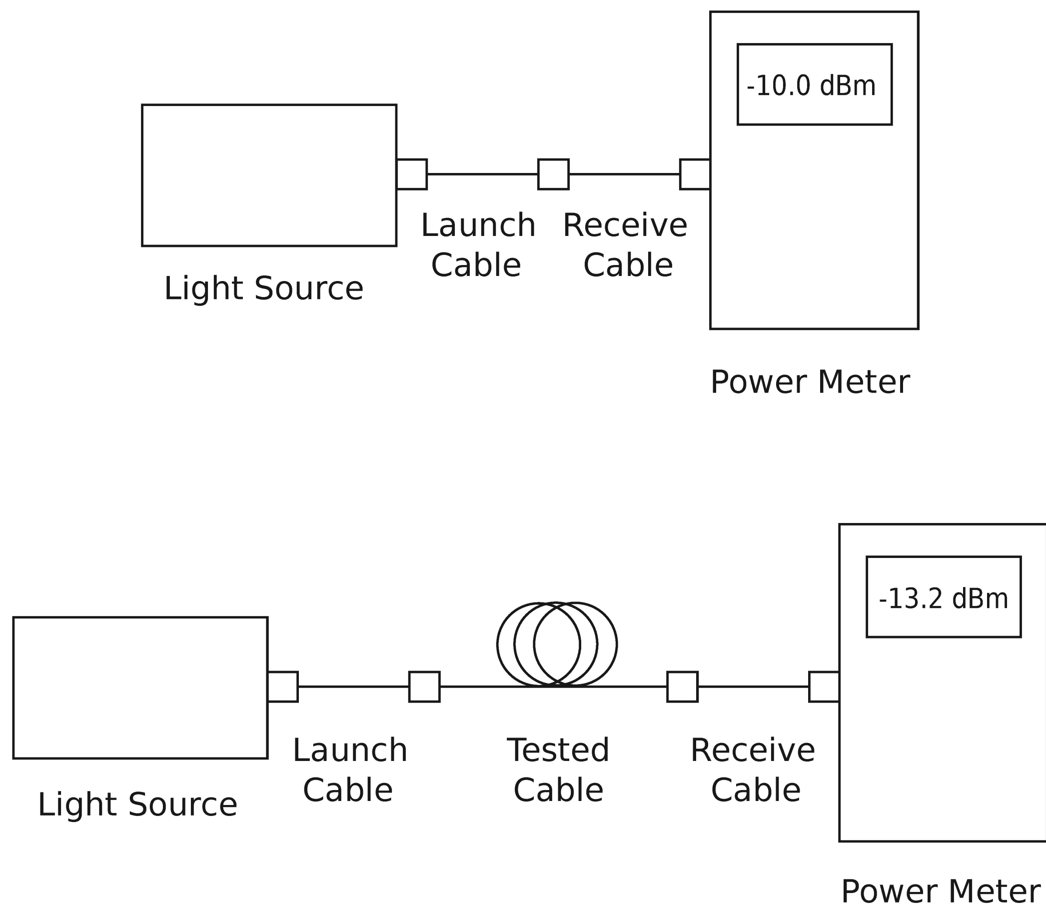

Figure 1: The calibration process on cable. (Image by Hecht [Hec02])

optical switches. The transmitter and OSA are required, as the calibration is done with these devices. The two switches are optional, but were included to obtain measurements from more than one port and to avoid the need for frequently reconnecting the tested components. The device contains the Beaglebone single-board computer[1] that run a Linux operating system, to which the user may connect to control the device.

The aim of this thesis is to create a working firmware for the OMCD. This firmware should manage the communication between inner devices (transmitter, OSA and switches) and implement self-configuration based on the YANG data model [MB10] to achieve high-level configuration and control of the device over Network communication protocol (NETCONF) [RE+11]. The firmware allows to use the device for the described self-calibration and measurement processes.

Contents of the thesis The Chapter 1 describes the OMCD firmware structure and interfaces. It describes serial-port-based communication in Unix systems that is needed for work with the transmitter and OSA, the NETCONF protocol, and the usage of the YANG data model for sending data over NETCONF. Chapter 2 describes the implementation of the firmware, including the precise description of communication with devices within OMCD, use data and communication packet formats, mapping of this data to the YANG data modeling language, connection of this functionality within the whole system. Chapter 3 demonstrates the functionality of the resulting firmware. In particular, it shows how the firmware is used to run device self-calibration and then the calibration of a Reconfigurable Optical Add-Drop Multiplexer (ROADM) device [Kun+19] that was provided by CESNET.

[1] See `https://beagleboard.org/bone`.

Chapter 1

OMCD firmware structure and interfaces

As mentioned in the Introduction, the OMCD consists of these devices: transmitter, OSA and switches. It is necessary to have two switches, one for the transmitter and one for the OSA. As shown in Fig. 1.1, the transmitter emits a signal to Switch 1 and the OSA measure the signal from Switch 2. The tested optical component or optical device is connected between these switches. Then these switches must be set to the correct ports to which the tested optical component or optical device is connected to provide the correct path for signal transmission.

After the implementation of inner functionality of the device, device management was needed. The NETCONF is used for device management. In Fig. 1.2, a user connects as a NETCONF Client to edit the configuration data or retrieve the current configuration. NETCONF Client provides a session with the server and sends requests from the user to the server. NETCONF Server executes all transactions, runs the device and reports device statistics and operational data. YANG modules define the schema of the configuration data, state data and RPC calls for the NETCONF Server. The Application represents the network device [Bah+16].

The first goal of the firmware for the OMCD was to provide the serial communication between all inner devices of OMCD as all devices of OMCD provide the RS-232 communication [Wikc] to make the calibration possible. The second goal of the firmware was to provide device management over NETCONF by implementing the YANG data model.

This chapter is divided into two sections: The first section is about the serial communication and how to obtain serial communication through serial port in Unix systems to create the inner functionality of the device. The second section is about NETCONF and YANG data modelling language, which are used to configure and control the device.

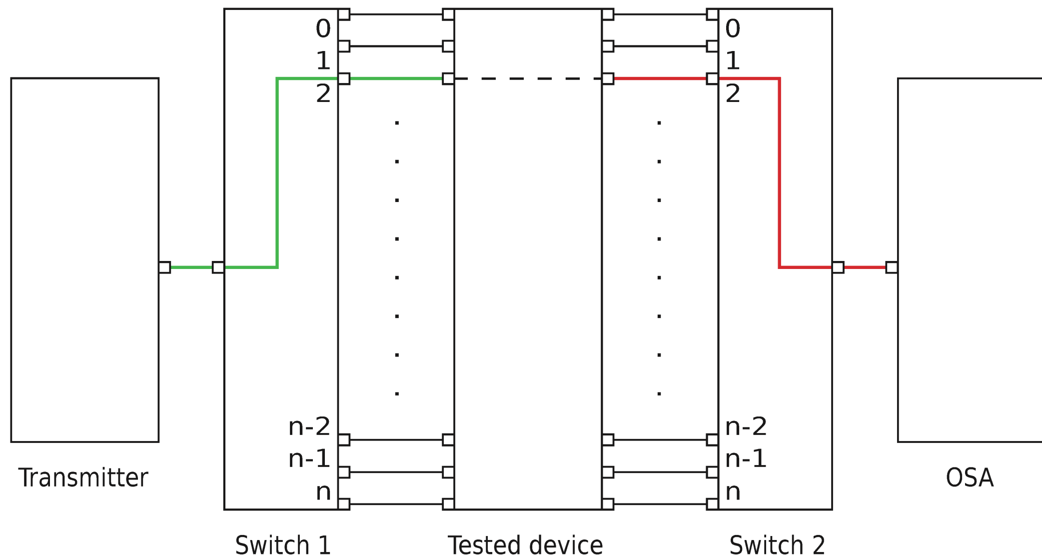

Figure 1.1: The structure of Optical Measurement and Calibration Device. The green path from the Transmitter through Switch 1 to Tested device indicates the emitted signal. The red path from the Tested device through Switch 2 to the OSA indicates the measured signal.

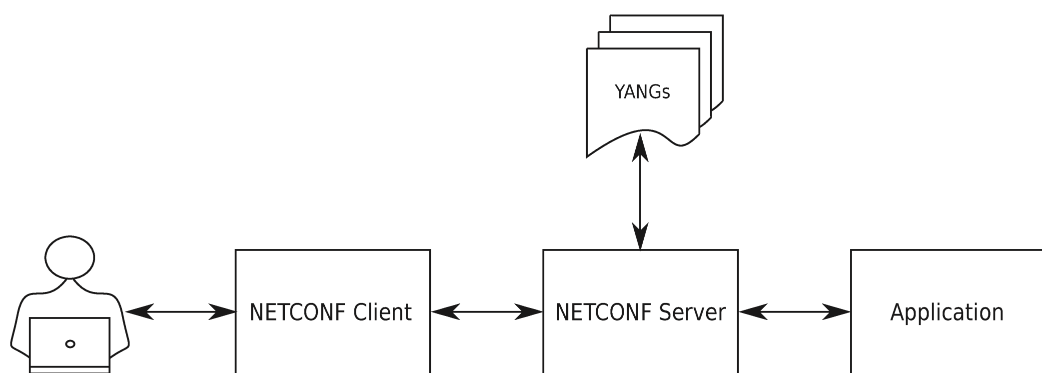

Figure 1.2: The schema of device management with NETCONF and YANG.

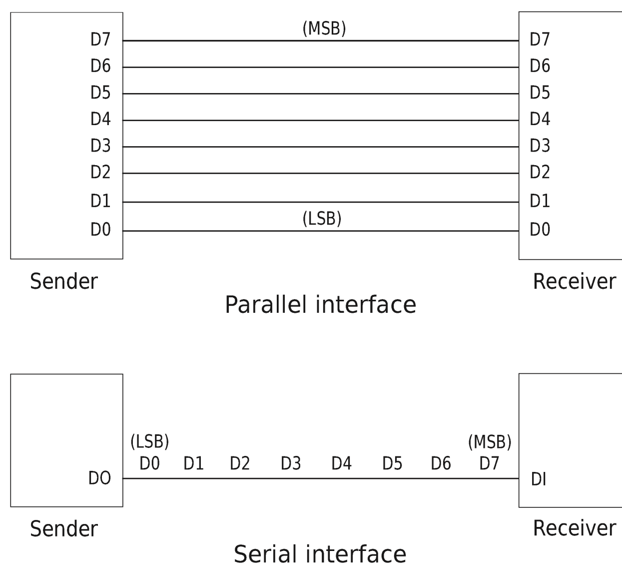

Figure 1.3: Parallel vs. serial communication.

1.1 Serial communication interface

Serial communication is a method where data are sent sequentially one by one bit over a communication channel or computer bus. It is the opposite of parallel communication, where bits are sent at once through many channels. This can be seen in Fig. 1.3. There are two types of serial communication: `synchronous` and `asynchronous` serial communication.

Synchronous serial communication is a method where data are sent at constant rate with clocks on both sides of the transmission: in both the transmitter and the receiver. These clocks must be synchronized and must run at the same rate. During the transmission, these clocks tend to drift apart. That means the resynchronization is required to continue the transmission. In this type of communication, there is no need to have start, stop and parity signals included in the transmission packet. This is required for asynchronous serial communication.

Asynchronous serial communication is a method where data are sent without synchronization, the transmitter and the receiver must provide their own clock timing. This is done by start and stop signals framed into the transmitted packet. The start bit synchronizes the clock of the receiver and the stop signal resets the state of the receiver and allows receiving a new packet. [Axe99]. This thesis focuses on asynchronous serial communication and asynchronous serial ports.

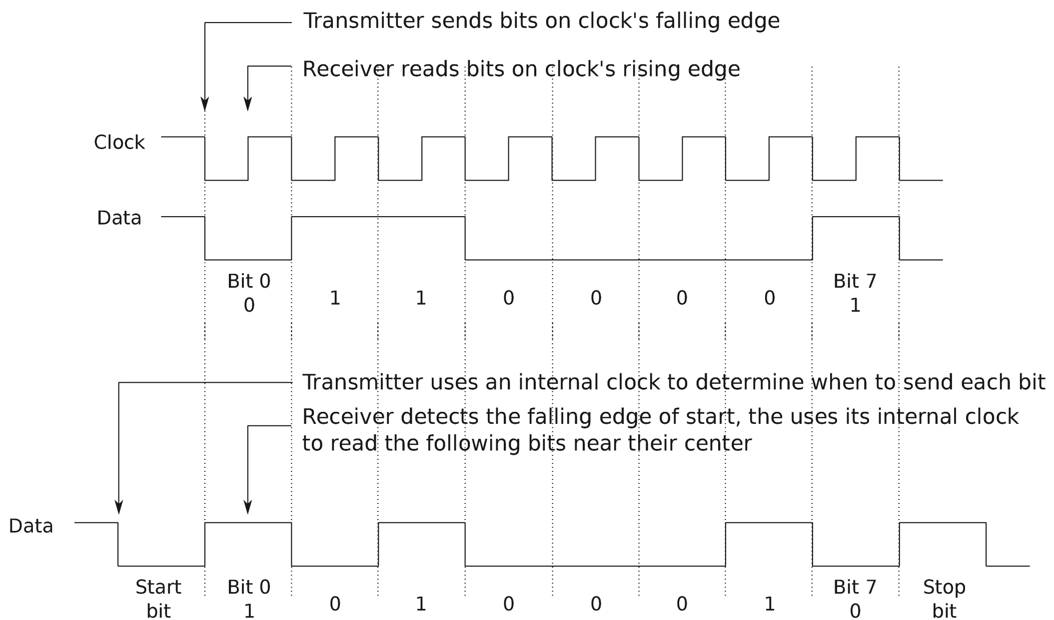

Figure 1.4: Synchronous vs. asynchronous serial communication. (Image by [Axe99])

1.1.1 Serial port on Unix systems

A serial port is an interface that provides the serial communication. In Unix systems, some of the device files present in the `/dev` directory are special files that can be used to access the serial ports. Otherwise, these can be opened, read from, written to and closed like ordinary files. This allows us to communicate with the device, which is attached to the serial port; the files serve as an interface to the actual driver, which is part of the kernel.

On Unix systems, the serial ports are typically named as `/dev/tty*`, in Table 1.1 the variety under Unix based systems can be seen.

Unix system	Serial port 1	Serial port 2
Linux®	/dev/ttyS0	/dev/ttyS1
Solaris®/SunOS®	/dev/ttya	/dev/ttyb
Digital UNIX®	/dev/tty01	/dev/tty02
IRIX®	/dev/ttyf1	/dev/ttyf2
HP-UX	/dev/tty1p0	/dev/tty2p0

Table 1.1: Serial port files on Unix systems. (Table by Strupp [Str].)

Listing 1 The termios structure defined in POSIX terminal interface.

```
struct termios {
  tcflag_t c_iflag;     /* input specific flags */
  tcflag_t c_oflag;     /* output specific flags */
  tcflag_t c_cflag;     /* control flags */
  tcflag_t c_lflag;     /* local flags */
  cc_t     c_cc[NCCS];  /* special characters */
};
```

Configuring the serial port

Computers and devices that provide asynchronous serial ports contain an Universal Asynchronous Transmitter-Receiver (UART). UART is a hardware device that provides low-level details of communication (e.g. the transmission speeds, data format) and conversion between USB, RS-232 or even parallel interface [NP16].

To start the communication with a device through the serial port, setting these low-level details on the side of the computer is necessary. In Unix systems, this can be done by POSIX terminal interface (dubbed *termios*), which is the Unix API for serial input and output [Ter]. The serial control structure and POSIX control functions are defined in `termios.h` header file from C POSIX library [Wika].

The structure `termios` in Listing 1, is defined in the `termios.h` and contains mostly flags. Importantly, the `c_cflag` describes basic communication parameters: baud rate, the length of data packet, parity bit, stop bit, etc. [Str] Some constants for the `c_cflag` are shown in Table 1.2. The functions which get and set the attributes for serial communication are `tcgetattr(3)` and `tcsetattr(3)`. The `tcgetattr(3)` function fills the pre-allocated `termios` structure with current serial port configuration. After the new configuration is initialized by setting flags in `termios` structure, the `tcsetattr(3)` sets the new configuration [Tcg].

Accessing the serial port

As said earlier, the serial port is represented by a file in the `/dev` directory. To provide communication with the device through the serial port, we use the standard Unix system calls on this file:

- `open(2)` [Ope], for opening the communication with serial device
- `close(2)` [Clo] for terminating the communication

	Constant	Description
Bit mask for baud rate	B9600	9600 baud
	B57600	57 600 baud
	B115200	115 200 baud
Bit mask for data bits	CS7	7 data bits
	CS8	8 data bits
Other	PARENB	Enable parity bit
	PARODD	Use odd parity instead of even
	CSTOPB	2 stop bits (default 1)
	CREAD	Enable receiver
	CLOCAL	Local line

Table 1.2: Defined constants for the `c_cflag` (Table by [Str].)

- `write(2)` and `read(2)` [IBMa; IBMb] to write to and read from the serial device
- `fcntl(2)` for manipulating several relevant settings of the file descriptor.

The serial port is typically opened with following flags:

- `O_RDWR`, to get bi-directional communication.
- `O_NOCTTY`, to not let the serial port become the controlling terminal of the process. If `O_NOCTTY` is not used, any input (e.g. mouse, keyboard abort signal) might affect (abort) the process.
- `O_NDELAY`, to set the file descriptor to no-delay mode. If `O_NDELAY` is not used, the process might be put to sleep e.g. because of a DCD signal[1]

By default, the `write(2)` function may block if the serial port is not ready to accept data, and the `read(2)` function may block if there are no available characters. We may alter this behavior by setting two flags using the function `fcntl(2)`:

- `O_NDELAY`, which causes the `write(2)` and `read(2)` return 0 in case of blocking.
- `O_NONBLOCK`, which causes `write(2)` and `read(2)` to not block the process and write or read what is possible, or sets the error code to EAGAIN and return -1.

[1]The DCD signal is used to indicate that a computer or device on the other end of a serial cable is connected to another remote.

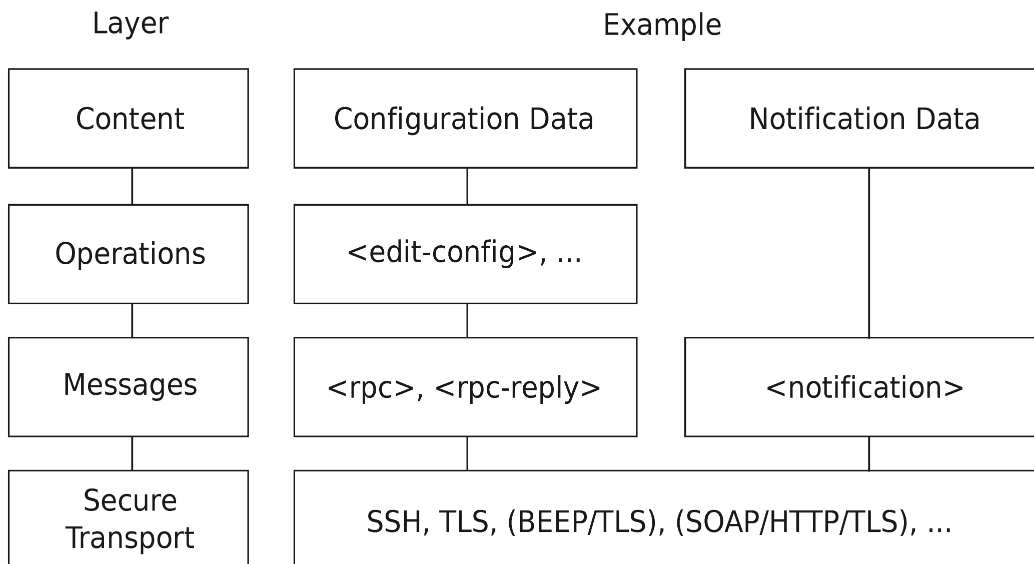

Figure 1.5: NETCONF layers. (Image by [RE+11])

1.2 Network Configuration Protocol

NETCONF is a session-based network management protocol which defines a mechanism to retrieve and upload configuration data and manage network devices. It was published as RFC 4741 [RE06] and then updated as RFC 6241 [RE+11]. NETCONF uses Remote Procedure Call (RPC) [Wikb] paradigm to realise operations and Extensible Markup Language (XML) [Wikd] to encode configuration data and protocol messages.

NETCONF can be partitioned into layers as shown in Fig. 1.5. The Secure Transport layer provides communication between server and client. It can be layered on any transport protocol, which provides the required functionality. The Messages layer provides a framing and encoding mechanism for notifications and RPC requests and responses. The Operations layer provides a set of operations invoked from RPC. The Content layer consists of configuration and notification data. The YANG data modelling language was implemented and covers the Operations and Content layers. [RE+11]

1.2.1 YANG

YANG is a data modelling language used to model configuration and state data sent over NETCONF or other network management protocol. It was published as RFC 6020 [MB10] and then updated as RFC 7950 [MB16].

The YANG data models are structured into modules and submodules. The data of modules can be imported into other modules. YANG models are structured as a tree. Each node in the tree has a name and has a value or a set of child nodes. YANG represents the operations and data in XML [Wikd] format and relies on XML Path Language (XPath) [Wike] notation to specify inter-node references and dependencies.

The following text introduces important constructs and statements used in YANG that are used to build the YANG model for our OMCD.

module

Statement `module` represents the root of a tree and defines the name of the module and groups the operations and data. (See Line 1 in Listing 2.)

type and typedef

YANG has a set of built-in types (statement `type`) like boolean, string, int8, uint8, int16, uint16 etc. These types are called "based types". The statement `typedef` defines new types. These types are called "derived types". To define a new type, the identifier as an argument followed by a block of substatements is required.

The `type` substatement in `typedef` is mandatory. The `type` defines the base type from which this type is derived. The `type` has optional substatements, for example `range`, which defines the range of values. In some cases, some substatements are required, for example `fraction-digits` is mandatory if the `decimal64` type is present. The `fraction-digits` rounds the type to a defined number of decimal places.

Other substatements for `typedef` are optional, for example `units`. The `units` substatement takes an argument, which is the definition of the units associated with the type. (See Line 4 in Listing 2.)

leaf

The `leaf` defines a leaf node in the tree schema. To define a leaf node, the identifier as an argument followed by a block of substatements is required. A `leaf` instance contains data. It has exactly one value of a particular `type` and has no child nodes.

The `type` substatement is mandatory and takes a name of a based or derived type as an argument. `Leaf` can have optional substatements like `mandatory`. This substatement takes argument `true` or `false`. If `mandatory` substatement is not present, the default value is false. If mandatory is set to true, the leaf must exist. (See Line 14 in Listing 2.)

container

The `container` defines the inner data in the tree schema. To define a `container`, the identifier as an argument followed by a block of substatements is required. A `container` does not have any value, it contains child nodes which have data values.

There are two types of containers. The first type of `container` does not have any meaning on its own. It only exists to contain the child nodes, to organize a structure in the tree. This is the default type. The second type of `container` has some significance. The existence of this container in the data tree is important for configuration of data. These containers are explicitly created and deleted. This second type of `container` is called “presence container” and it is established by substatement `presence` in the container. (See Line 11 in Listing 2.)

rpc

The `rpc` statement defines the RPC operation for NETCONF. To define `rpc`, the identifier as an argument followed by a block of substatements is required. This argument is the name of the RPC and is used as the name of the element directly under the `<rpc>` element.

The main substatements of `rpc` are `input` and `output`, these substatements are optional. Logically, the `input` defines input parameters for the RPC operation and `output` defines output parameters for the RPC operation. These substatemens do not take any arguments and define nodes under the nodes of RPC. (See Line 24 in Listing 2.)

Listing 2 Example of YANG data model.

```
module access-validator {
  namespace "http://bachelorthesis.example.com/access";

  typedef age {
    type uint32 {
      range "0..150";
    }
    units "years";
  }

  container register-person {
    presence "Register person with name and age";

    leaf person-name {
      type string;
      mandatory true;
    }
    leaf person-age {
      type age;
      mandatory true;
    }
  }

  rpc access {
    input {
      leaf your-age {
        type age;
        mandatory true;
      }
    }
    output {
      leaf allowed {
        type boolean;
        mandatory true;
      }
    }
  }
}
```

Chapter 2

Implementation of the OMCD firmware

This chapter describes the implementation of OMCD firmware. The main functionality provided is the communication between its components, which are: transmitter, switches and OSA as already mentioned in the introduction. For every component a driver[1] was written. Then a driver for communication between all devices was created. After the inner functionality of OMCD was written, the control interface of the device was implemented by creating a YANG data model interface to make it available for higher-level configuration over NETCONF protocol.

The implementation of drivers was based on an abstract interface provided by the internal software of CESNET. The code is structured in directories as shown in Table 2.1.

2.1 Implemented drivers

In this section, we will discuss the implementation of drivers which can be found in `src/drivers/` directory for devices which will comprise the OMCD. We created three drivers for three devices:

- Transmitter: Integrable Tunable Laser
- Switch: Coaxial Fiber Optic 1xN Switch
- Power monitor: Optical Spectrum Analyzer

[1]Software driver which provides a programming interface to control a specific lower level interface. [Dri]

Component	Driver file
ITLA driver (see Section 2.1.3)	src/drivers/itla/Itla.h src/drivers/itla/Itla.cpp src/drivers/itla/Packet.h src/drivers/itla/Packet.cpp src/drivers/itla/Property.h src/drivers/itla/Property.cpp
SERCALO driver (Section 2.1.4)	src/drivers/sercalo/MemsSwitch.h src/drivers/sercalo/MemsSwitch.cpp src/drivers/sercalo/Property.h src/drivers/sercalo/Property.cpp
AXSUN driver (Section 2.1.5)	src/drivers/axsun/Ocm.h src/drivers/axsun/Ocm.cpp src/drivers/axsun/Packet.h src/drivers/axsun/Packet.cpp src/drivers/axsun/Property.h src/drivers/axsun/Property.cpp
YANG model (Section 2.2) High-level interface (Section 2.3)	yang/czechlight-calibration-device.yang src/appliance/CalibrationBox.h src/appliance/CalibrationBox.cpp
Unit tests (Section 2.4)	tests/itla-properties.cpp tests/sercalo-mems-switch.h tests/axsun-packet-parsing.cpp tests/axsun-ocm.cpp tests/calibration-box.cpp
Demonstrations (Section 2.4)	src/demo/itla-demo.cpp src/demo/sercalo-demo.cpp src/demo/axsun-demo.cpp

Table 2.1: Directory layout of the implementation source code.

Listing 3 The abstract interface for generic properties where read method returns just a single PropertyValue.

```
struct AbstractProperty {
  virtual ~AbstractProperty() {}

  virtual PropertyValue read(io::StreamPtr& conn) = 0;
  virtual void write(io::StreamPtr& conn,
                     const PropertyValue& value) = 0;
};
```

As mentioned in the introduction, two switches were used. However, the exact same model was used for both of these switches, so the implemented driver works for both of them without any modifications.

2.1.1 Property Interface and Serial Communication

The devices have many properties to offer, eg. the tunable laser (transmitter) can transmit signal with defined frequency, power etc. To reach these properties and change the setting, like setting the frequency and power, we need to put these commands into logical units.

The main meaning of abstract classes in Listings 3 and 4 is that they provide an interface for serial transactions. The implemented properties inherit from one of these abstract classes and implement their own communication. Each property is forced to communicate with the device with its own configuration by its own methods. The advantage of this approach is that different types of properties unify the interface and references can be stored into one collection.

Methods read and write in Listing 3 get an input parameter `io::StreamPtr`, which is an object that provides the interface of serial communication. Then the class `io::Tty` implements the serial communication. These classes are provided by internal software of CESNET. The write method has another input parameter PropertyValue, which is the value that will be written to the device through the serial port.

In Listing 4 the read transaction is different. From some devices, a lot of data is obtained during a single read transaction and our goal is to keep all these data in a clear manner. That is the purpose of the `PropertyTree` input parameter: it holds all obtained data. For clarity, a prefix string is used, which is inserted at the beginning of string names of the data kept in the container.

Listing 4 The abstract interface for generic properties where read modifies PropertyTree to return more data.

```
struct AbstractProperty {
  virtual ~AbstractProperty() {}

  virtual void read(io::StreamPtr& conn,
                    const std::string& prefix,
                    PropertyTree& tree) = 0;
  virtual void write(io::StreamPtr& conn,
                    const PropertyValue& value) = 0;
};
```

2.1.2 Driver Interface

Every driver class inherits from the class `Driver`, which is the driver interface provided by CESNET. Every driver is initialized with `io::StreamPtr`, which is as we said the serial communication interface and `io::Tty` is the key of serial communication. The instance of `io::Tty` is an input parameter in constructors of drivers. Every driver initializes from the constructor properties which the device offers, checks the communication by reading noop register or id specifics from the device and provides this specifics to a logger `cla::Log`. We used spdlog [Mel] which is a logging framework. The mandatory methods to implement are:

- `std::vector<std::string> propertyNames();` to return the list of properties which device offers. This properties are collected into `std::map<std::string, std::unique_ptr<AbstractProperty> >`.
- `std::vector<std::string> propertyNames();` to provide the read transaction on property.
- `std::vector<std::string> propertyNames();` to provide the write transaction on property.

2.1.3 Integrable Tunable Laser ITLA

The first implemented device was a transmitter. The specific model (Pure Photonics ITLA PPCL200) the Integrable Tunable Laser made by the Pure Photonics company was used. The Pure Photonics ITLA is a Continuous Wave Tunable Laser with integrated electronics [Phoa].

ITLA has many registers that provide many properties. For example, setting optical power, getting factory information about the device, setting frequency

with a lot of parameters like: laser channel, grid spacing, first channel frequency and fine tune frequency. The product is compliant to the OIF (Optical Internetworking Forum) MSA (Multi-Source Agreement). OIF-ITLA-MSA-1.2 is publicly available [Phob]. We will describe the communication interface with the device, which communication protocol is used, how bytes are supposed to be packed into a packet and how we implemented the commands to adjust properties of the device.

For brevity, when talking about a packet which is transmitted from the host to the module we will use “Request packet” instead and when talking about a packet which is transmitted from the module to the host we will use “Response packet”.

Communication protocol

The communication with the product is through a standard RS-232 interface. To facilitate communication from the host to the module, the Request packet must be configured as in Table 2.2. This packet consists of four bytes. The third and fourth bytes are reserved for data, which the host is trying to send to the module. The second byte carries the number of the register, from which the host wants to obtain information, or to which the host wants to set data. The first byte is more complicated and carries more information. The first byte consists of the following bits:

- Bit 24: This bit indicates, whether the request is a read or write transmission.
- Bits 25–26: These bits are not used and should be set to 0.
- Bit 27: LstRsp bit for communication errors. Setting this bit to 1 makes the module ignore the rest of the packet. Then the last response is returned.
- Bits 28–31: Checksum.

After successful transmission from the host to the module, a response packet from the module should be received by the host. The response packet from the module should look as shown in Table 2.3. This packet also consists of four bytes. The third and fourth bytes are reserved for data. The second byte carries the number of the register from which the host wants to obtain information or to which the host wants to set data. Again, the first byte is more complicated and carries more information. The first byte consists of the following bits:

- Bits 24–25: Status error. The meanings of status errors can be found in Table 2.4.

31	30	29	28	27	26	25	24
Read/Write	0		LstRsp	Checksum			
23	22	21	20	19	18	17	16
Register number							
15	14	13	12	11	10	9	8
Data 15:8							
7	6	5	4	3	2	1	0
Data 7:0							

Table 2.2: Structure of the request packet for ITLA.

31	30	29	28	27	26	25	24
Status error		1	Consistency	Checksum			
23	22	21	20	19	18	17	16
Register number							
15	14	13	12	11	10	9	8
Data 15:8							
7	6	5	4	3	2	1	0
Data 7:0							

Table 2.3: Structure of the response packet from ITLA.

- Bit 26: This bit is not used and should be set to 1.
- Bit 27: This bit indicates whether the received packet was damaged, i.e. it is set to 1 if the Request packet (from host) had incorrect checksum.
- Bits 28–31: Checksum.

In case of a read transaction, the register number is sent back from the module to the host as an acknowledgement. In case of a write transaction, both the register number and data are sent.

The checksum `BIP-4` (Bits interleaved parity four bits wide) is calculated by using XOR on all bytes and then using XOR on the high and low half of the resulting byte. Listing 5 shows the algorithm.

Communication packet

It is convenient to create an interface for creating and parsing packets appropriately. That is why we created structures `RequestPacket` and `ResponsePacket`.

`RequestPacket` is a structure, whose constructor takes number of register, data and `ReadWriteRequest` value as input parameters. `ReadWriteRequest` is

Register	Flag name	Status
0x00	OK	`Normal return status.` Indicates that everything worked as expected.
0x01	XE	`Execution error.` Indicates an execution error of the last command.
0x02	AEA	`Automatic extended addresing.` Indicates that the response is longer than the regular length of data in the packet. If the response is long enough to trigger the AEA flag, then the 2nd and 3rd bytes contain the number, which indicates how many times the AEA register has to be read. If the AEA register is read more times than indicated in the 2nd and 3rd bytes, the next response throws an execution error.
0x03	CP	`Command not complete, pending.` Indicates that the command will take longer than expected. The host can read from the nop register (0x00) and see from the response packet, if the command is still being executed. When it's done, the OK flag is returned.

Table 2.4: ITLA: Packet status table.

an enum with only two possible values: 0 and 1. These values indicate whether the transaction is a read (0) or write (1) request. The `RequestPacket` constructor creates a packet according to its configuration, which is then stored in string `rawPacket`.

`ResponsePacket` is a structure whose constructor takes string `rawPacket`, which is the whole packet received from the module. The `ResponsePacket` constructor parses the packet, checks the checksum, checks a few bits and if everything is correct, it then stores packet status and data into its variables.

Implemented Device Properties

ITLA provides many properties and we didn't need most of them, so we implemented only the vital ones. Many properties have similar characteristics and that is why we decided to group them into three main classes. Each class inherits from the abstract class and each class implements its own read and write communication.

- `AEAProperty` is a class that provides mainly the ability to read long strings from AEA register. As we have shown, transferred data are only 16-bits

Listing 5 Calculating the checksum.

```
uint8_t calcBIP4(uint8_t firstByte, uint8_t secondByte,
                 uint8_t thirdByte, uint8_t fourthByte)
{
    uint8_t bip8 = (firstByte & 0x0F)
                 ^ secondByte
                 ^ thirdByte
                 ^ fourthByte;
    uint8_t bip4 = ((bip8 & 0xF0) >> 4)
                 ^ (bip8 & 0x0F);
    return bip4;
}
```

Listing 6 The Request packet structure.

```
    struct RequestPacket {
      RequestPacket(const uint8_t reg,
                    const int16_t data,
                    const ReadWriteRequest rw);

      std::string rawPacket;
    };
```

long, so if we want to read a large amount of data from the device, we need to read it through the AEA register.

- `OneRegisterProperty` is a class that provides basic communication with one register.
- `TwoRegisterProperty` is a class that provides basic communication with two registers.

Every property object created through these main classes contains its own one or two registers, and every property object provides its own communication by the communication protocol.

However, our focus was to implement basic operations regarding frequency and optical power and also operations for enabling and disabling the optical output and soft and hard resetting the device. We created another three property classes:

- `FrequencyProperty` for setting or getting the frequency.

Listing 7 The Response packet structure.

```
struct ResponsePacket {
  ResponsePacket(const std::string& rawPacket);

  uint8_t status;
  uint16_t data;
};
```

- `PowerProperty` for setting or getting the optical power.
- `OnOffProperty` for enabling/disabling the optical output or resetting the device.

These properties were not necessary, it is possible to communicate with the device just through the first three classes, but we wanted to operate the frequency and optical power while simultaneously bound-checking the hardware capabilities.

Through `FrequencyProperty` we can obtain and set the frequency by adjusting parameters in the following equation from the Integrable Tunable Laser Assembly Multi Source Agreement [For15]:

$$\begin{aligned}\text{Freq} =& (\text{LaserChannel} - 1) * \text{GridSpacing} \\ &+ \text{FirstChannelFrequency} \\ &+ \text{FineTune}.\end{aligned}$$

To get the frequency, we have read the fine tune frequency and laser channel and provide the equation with grid spacing and first channel frequency. When setting the frequency, we have to check that the frequency we want to set is not above or below the hardware limit. Then we provide the calculation with laser channel and fine tune frequency which correspond to the new frequency together with grid spacing and first channel frequency and set the laser channel and fine tune frequency to the right registers.

`PowerProperty` is used to set or read the optical power. The process of setting the optical power includes checking, whether the new optical power is not above or below the hardware limit before the write transaction is sent. Read transactions behave the same as during regular communication.

`OnOffProperty` implements communication with just one register to enable enabling/disabling optical output and soft/hard resetting the device. This register offers all these commands by setting the right bit into the right position of this 8-bit register.

2.1.4 Coaxial Fiber Optic Switch SERCALO

Coaxial fiber optic switch made by the Sercalo company is a MEMS switch where mirror redirects light from a common fiber to one of N ports [Ltd17]. Sercalo's switch provides properties, like id of the product, resetting the device, reading the temperature of the device etc. But the most important property is the port property.

Communication protocol

The switch can communicate over UART or I^2C. We chose the communication over UART, thus only the UART packets will be described.

If the host wants to send a packet to the module, it must have the following configuration:

```
CMD <parameter1> <...> [optional_parameter1] [...] EOL
```

where

- `CMD` is the command, it is a sequence of chars
- <> are mandantory parametes
- `[]` are optional parameters
- `EOL` is end of line. The device recognizes these forms of end of line: CR [2], LF [3], or the combination CR+LF. Device always replies with a message with the CR+LF end.

After successful transmission from the host to the module, a reply from the module should be received by the host. The reply is a command-dependent acknowledgment or an error message. The error message starts with a sequence of chars ERR, followed by a space and additional data.

In this case the communication is pretty simple, that is why we did not need to create any packet structure. We created the method:

```
std::string talkToHw(io::StreamPtr& conn,
                     const std::string& command,
                     const std::string& request);
```

which creates the proper packet, sends it to the device and checks the response packet.

[2] Carriage return: \r

[3] Line feed: \n,

Implemented Device Properties

In this implementation, we chose to implement every property in a seperate class. The reason is that every property of the device is unique, unlike with ITLA, where some properties had the same communication structure with the same data types. We created the following properties:

- `IdProperty`, which gives us information about productModel, serial number or firmware version. This property works with the command ID.
- `ResetProperty`, which resets the device. This property works with the command RST.
- `PortProperty`, which sets or returns the optical switch network. It is also possible to disable the common port. Then the common port is not routed to any port and the optical path is open. This property works with command POS and SET.
- `TemperatureProperty`, which gets the temperature of the microcontroller. This property works with the command TMP.
- `OpticalBandProperty`, which sets or returns the default optical band or the optical band in use. The default optical band is the optical band that is automatically set after reset or power-on. This property works with the command DBAND for the default optical band and BAND for the optical band in use.

2.1.5 Optical Spectrum Analyzer AXSUN

Optical Spectrum Analyzer made by the AXSUN company is a device which performs an optical power and frequency measurement and calculates the optical signal [Tec13].

Communication protocol

The communication with the product is through a standard RS-232 interface. To facilitate communication from the host to the module, the Request packet must be configured as in Table 2.5. After successful transmission from the host to the module, a response packet from the module should be received by the host. The response packet from the module should look like in Table 2.6. The length of the packet can vary because of the data. Both response and request packets consist of three main parts: Header, Payload and Footer. The `Header` also consists of several parts:

	31:24	23:16	15:8	7:0
Header	Message Identifier			
	Message Length			
	Unused			
	Unused			
Data	Data			
	Data Checksum			
Checksum	Unused			
	Message Checksum			

Table 2.5: Structure of the request packet for AXSUN.

	31:24	23:16	15:8	7:0
Header	Message Identifier			
	Message Length			
	Device Status			
	Device Temperature			
Data	Data			
	Data Checksum			
Checksum	Error Code			
	Message Checksum			

Table 2.6: Structure of the request packet from AXSUN.

- Message Identifier
- Message Length
- Unused in request packet, Device Status in response packet
- Unused in request packet, Device Temperature in response packet

`Payload` are data which are sent. If the host wants to read from the device and does not want to send any data, it must send one unsigned 32-bit integer with value `0`. The `Footer` also consists of several parts:

- Data checksum
- Unused in request packet, Error Code in response packet
- Message checksum

In this case, there are two types of checksum. The `Data checksum` is calculated from the payload of the packet. The `Message checksum` is calculated from the

Listing 8 The calculated checksum for AXSUN.

```
uint32_t checksum(const std::string& data)
{
  uint32_t sum = 0;

  auto it = data.cbegin();
  while (it != data.cend())
    sum +=
      cla::utils::parseBigEndianBuffer<uint8_t>(it);

  return ~sum;
}
```

Listing 9 The Request packet structure for AXSUN.

```
struct RequestPacket {
  RequestPacket(uint32_t messageIdentifier,
                const std::vector<uint32_t>& payload);
  std::string rawPacket;
};
```

whole packet excluding the Message checksum. Checksum is calculated by cumulative sum of all bytes and after the last byte is summed, the data should be converted into one's complement. This algorithm can be seen in Listing 8.

Communication packet

It is convenient to create an interface for creating and parsing packets appropriately. That is why we created structures `RequestPacket` and ResponsePacket.

`RequestPacket` is a structure, whose constructor takes the input `messageIdentifier` as an unsigned 32-bit integer and `payload` as a reference to a vector of unsigned 32-bit values. The data are sent in vector form, so parsing the data into a vector is required. The `RequestPacket` constructor creates a packet according to its configuration, which is then stored in string `rawPacket`.

To store the data in a clear way, the response packet needs a more demanding structure. We created structure `Header`, which stores four variables, which are sent or received in the header part of the packet. Also we created structure `Footer`, which stores three variables, which are sent or received in the footer part of the packet. Finally, structure `ResponsePacket` was created. This structure stores the `Header` and `Footer` structures. Data are stored in the `payload`

Listing 10 The Response packet structure for AXSUN.

```
struct Header {
  uint32_t messageIndetifier;
  uint32_t messageLength;
  uint32_t deviceStatus;
  uint32_t deviceTemperature;
};

struct Footer {
  uint32_t dataChecksum;
  uint32_t errorCode;
  uint32_t messageChecksum;
};

struct ResponsePacket {
  ResponsePacket(const std::string& rawPacket);

  Header header;
  std::string payload;
  Footer footer;
};
```

string. The constructor of `ResponsePacket` parses the packet into variables according to the configuration above.

Implemented Device Properties

We implemented three properties:

- `SampleScan`, which provides information about the measured signal. With a specific subcommand, the device makes a specific optical measurement and reports the data back.

- `DiagnosticData`, which provides information about the device, like Firmware version, PC Board ID, Optical Module ID and Kernel stored in the flash memory. Also provides information like raw cooler voltage, current-on-board temperature, optical module bench temperature etc.

- `WarmReset`, which executes a warm reset on the device. After warm reset, the device shall transmit a diagnostic packet, but that is not necessary for us.

Property `SampleScan` provides many subcommands, but we implemented only three of them:

- Subcommand `Peaks` will request the device to make an optical measurement and report back the Peak channels only.
- Subcommand `PeaksPowerFrequency` will request the device to make an optical measurement and report back the peaks of channels and all calibrated powers followed by all calibrated frequencies. The device will report back both the number of peaks and number of calibrated power.
- Subcommand `HighResolutionPeak` will request the device to make an optical measurement and report back the peaks of channels and provides more information about the frequency.

2.2 YANG interface

Firstly, defining the data boundaries for NETCONF was necessary. The following boundaries were the best in terms of range and hardware capabilities. We created three derived types: `port-number`, `frequency-mhz` and `power-dBm`. The `port-number` is derived from the base type `uint8` and the range of values is 1–36. The `frequency-mhz` is derived from the base type `uint32` and the range of values is 191500000–196250000 MHz. The `power-dBm` is derived from the base type `decimal64` and the range of values is -100.00..10.00 dBm.

We wanted the final OMCD to perform two operations: transmit signal to a specific port and measure signal from a specific port. For the transmission, we created a YANG container `signal-source`. This container is a presence container, that means that if the container exists, the device is enabled. The container has three leaves, one for each type created: one for port, one for frequency and one for power to transmit signal with specified frequency and power to a specified port. These leaves are mandatory, so the values must be set.

For the measurement we used statement rpc `measure`. The `measure` consists of input and output substatements. Both input and output have leaves inside. The input leaf contains a port value. To measure, we need to specify from which port we want to measure. The output leaf contains the power value. The output of the measurement process is only the power of the signal.

2.3 High-level OMCD driver

After we implemented YANG data model, we needed to provide API for configuration operation `signal-source` and rpc operation `measure` to manage the

Listing 11 The abstract interface for generic properties where read method returns just a single PropertyValue.

```
void Box::writeMultiProperties(const PropertyTree& values);
PropertyTree Box::rpc(const std::string& name,
                      const PropertyTree& input);
```

ITLA, two SERCALO switches and AXSUN. The interface consists of two methods show in 11.

`Box::writeMultiProperties` in Listing 11 is called when the container `signal-source` is created to configure the OMCD to transmit the signal with specific frequency and power to a specific port. The method gets an input parameter `PropertyTree& input`, which is a container that collects data for device configuration. It parses the element expressions as an XPath in a for loop (the YANG data modelling language expresses tree schema with XPath notation). It checks the name of the container and the names of the leaves, and if everything is correct, it saves the data to its private variables. If the values of data are of type `DeletedProperty`, they represent that the container was deleted and we are disabling the device; the method consequently disables the ITLA and switch of ITLA. Otherwise, we are configuring the device with the data, i.e. setting the ITLA to emit the signal, and setting the ITLA switch to the port, to which the signal will be transmitted. The emitted signal is automatically adjusted according to the calibration data stored in the device.

`Box::rpc` in Listing 11 is called when the RPC operation `measure` is called to measure from a specific port. The method gets an input parameter `std::string name` which is the full XPath to RPC operation defined in YANG data model, and `PropertyTree& input` that contains the input data for the device. It expects that `PropertyTree& input` contains only one entry: `"port", number`. The method checks this and then sets the switch of AXSUN to a given port (if the port number is correct) and sets AXSUN to measure the signal from that port. If AXSUN does not return any peak, that means it measured no signal and the method returns the minimal defined power, which is -100dBm. If AXSUN returned one peak, then the method returns this peak plus the calibration data, to obtain the original optical power. If AXSUN measures more than one peak, it throws an error, as that is not supposed to happen.

All implemented functionality is finally registered in Sysrepo [Vas], which manages the communication with the user and forwarding of the NETCONF request.

2.4 Testing and demo versions

During the implementation of drivers, we ensured the correctness by writing unit tests for ITLA, SERCALO, AXSUN and CalibrationBox. The unit tests use MOCK objects mainly for simulating the serial port communication. We used Doctest [**doctestand**] as a C++ unit test framework and trompeloeil [Fah] for MOCK objects in C++. We created demo versions for devices and used them to prove that the driver works as expected on the hardware.

Chapter 3

Results

After we implemented the firmware, we have conducted two simple experiments to verify the functionality. First, we have cross-compiled the software for ARM platform and loaded it into the Beaglebone board in the OMCD. In the first experiment, we used OMCD for self-calibration, i.e. for measurement of the actual optical output power of the ITLA and subsequent measurement of the attenuation of the rest of the internal components. In the second experiment we used the self-calibrated OMCD to measure the actual production hardware, in this case the ROADM multiplexer used and provided by CESNET.

In this chapter we use the results of the experiments as a demonstration of the functionality of the implemented firmware.

3.1 Self-calibration

The self-calibration is conducted as follows: Before we connect the fibres into TX[1] and RX[2] ports, the fibres must be clean. If grime is present, the fibres must be cleaned, because this also may also cause losses in the transmitted signal. After cleaning the fibres, we need to assure that both devices (OMCD, power meter) are set to equal wavelength range. The typical optical band is C-band, which is around 1530 to 1565 nm. Then we can connect them and measure the attenuation as described in the introduction (see Fig. 1). We configure the emitted signal and turn on the optical output of ITLA. The configuration is done by using the netconf-cli tool [Kub] in a similar way as in Listing 12. Then we can measure the attenuation and calibrate the system. We first measure the ports on the switch connected to ITLA by a power meter as shown in Fig. 3.2. The results are written into `/cfg/calibration/switch-itla.txt` file. Second, we measure the ports

[1] Transmitted
[2] Received

Figure 3.1: OMCD device with the final firmware installed. In the top left part the AXSUN device can be seen, ITLA is located in the middle under the GIGABYTE heatsink, and to the right of ITLA there is the Beaglebone single-board computer. The two SERCALO switches are located right of Beaglebone.

Listing 12 The communication with firmware using the NETCONF Client console.

```
# netconf-cli
Welcome to netconf-cli
/> set czechlight-calibration-device:
czechlight-calibration-device:measure
czechlight-calibration-device:signal-source/
frequency   port    power
/> set czechlight-calibration-device:signal-source/frequency 193000000
/> set czechlight-calibration-device:signal-source/power -10
/> set czechlight-calibration-device:signal-source/port 1
/> commit
```

on the switch connected to AXSUN using AXSUN. The results are written into `/cfg/calibration/switch-axsun.txt` file. These result (shown in Table 3.1) are used to improve the precision of the calibration process.

3.2 Use case: Measuring ROADM

After the OMCD wa calibrated, we were able to measure other devices to test their attenuation. For the experiment we used the Reconfigurable Optical Add-Drop Multiplexer (ROADM) [Kun+19] that is routinaly used within CESNET. We connected the 6 ports of ROADM to OMCD; the connection schema is shown in Table 3.2.

The final measurement setup is shown in Fig. 3.3. [3] We transmitted the signal at -10dBm optical power. As we can see from measuring the add path in Table 3.3, we obtained amplification on port E1, attenuation on port E3 and E2 and E4 were measured dark. That means we not only found out that the system has attenuation and amplification, but also has ports, which do not work correctly. This might have been caused by broken connectors. From the measurements of the drop path in Table 3.4 we can see significant amplification.

[3] In ROADM, the "add" inserts one or more new wavelength channels to an existing WDM (Wavelength-Division Multiplexing) signal. The "drop" extracts one or more channels, passing these signals to another network path. We measured both add and drop paths.

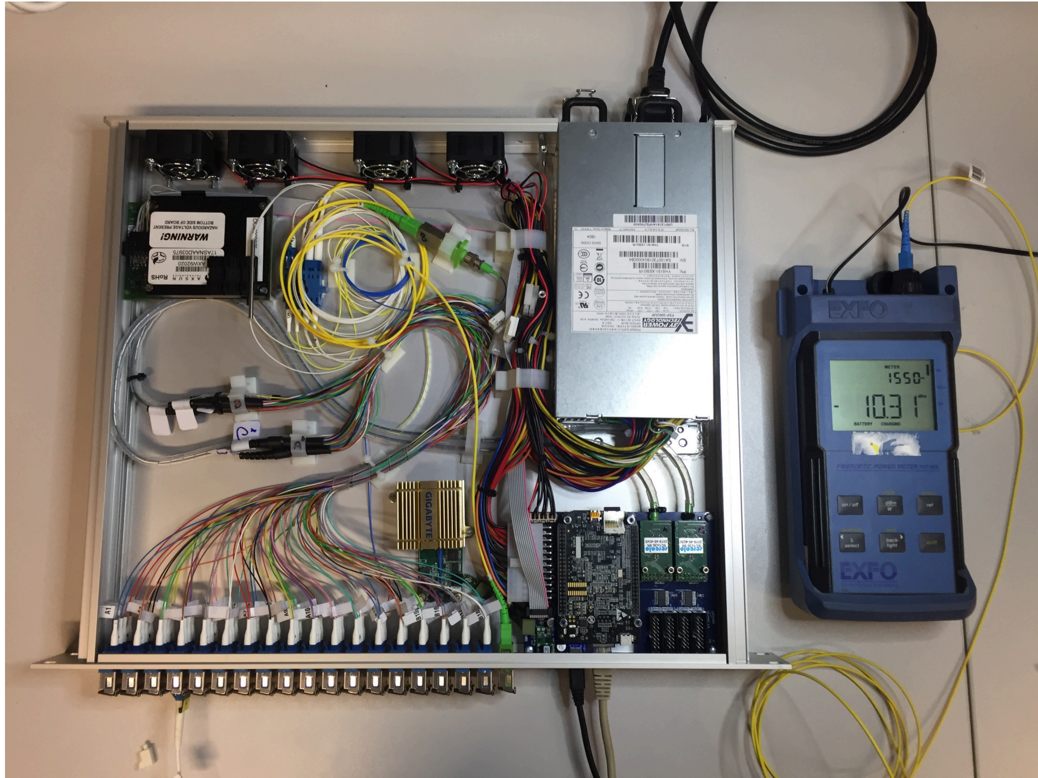

Figure 3.2: OMCD device during the process of self-calibration. The power meter on the right is used as a reference for the self-calibration of ITLA.

Port	Switch on ITLA	Switch on AXSUN
1	0.45	0.30
2	0.87	0.40
3	0.75	0.80
4	0.36	0.40
5	0.60	0.40
6	1.00	0.60

Table 3.1: Results of measuring the attenuation on ports of ITLA and AXSUN. (Only the first 6 ports were measured.)

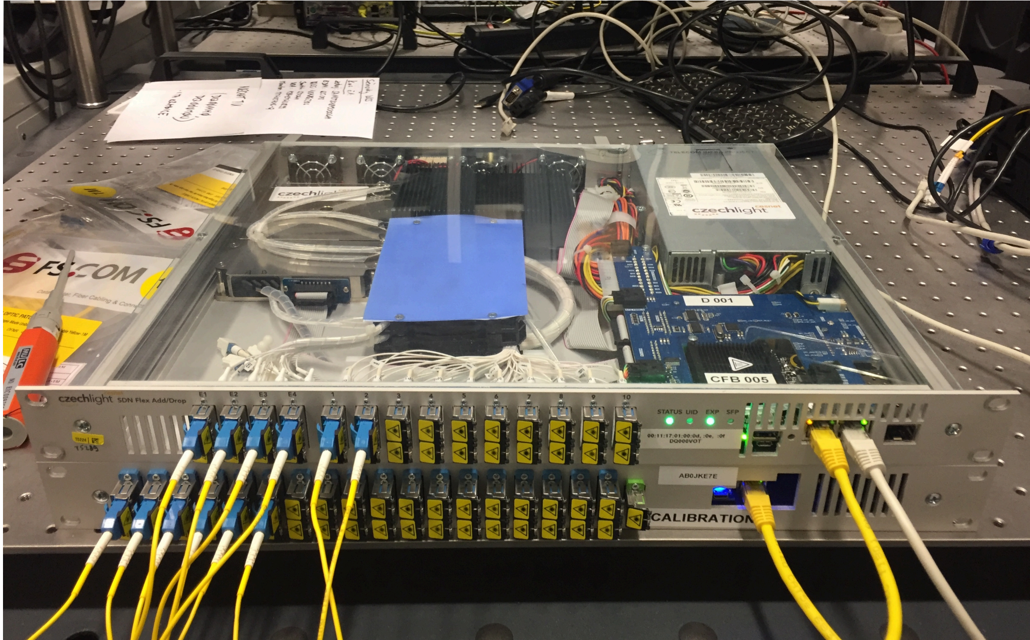

Figure 3.3: Measuring ROADM with OMCD. The device stacked on top is the ROADM, OMCD is below.

ROADM port	**Calibration box port**
E1	1
E2	2
E3	3
E4	4
1	5
2	6

Table 3.2: Connected ports of ROADM to OMCD.

Measured port	Measured signal added from port 1	Measured signal added from port 2
E1 (1)	-8.90	-8.20
E2 (2)	-100.00	-100.00
E3 (3)	-14.20	-13.40
E4 (4)	-100.00	-100.00

Table 3.3: Measurements of the ports E1–E4 on ROADM (on OMCD ports 1–4) obtained by sending the signal to port 1 (in OMCD port 5) and to port 2 (in OMCD port 6).

Measured dropped port	Measured signal sent from port E1	Measured signal sent from port E2
1	-1.5	-1.6
2	-2.3	-2.1

Table 3.4: Measurements of the ports 1 and 2 on ROADM (on OMCD ports 5 and 6) obtained by sending the signal to port E1 (in OMCD 1) and to port E1 (in OMCD port 2).

Conclusion

In this thesis, we have described the implementation and testing of the OMCD firmware.

In Chapter 1, we described the asynchronous serial communication required for the communication with OMCD device components, and the UART serial ports on Unix systems. We detailed the NETCONF protocol as used as a main communication format with the OMCD firmware, and the YANG data modeling language, which is used to generically describe devices and their properties.

In Chapter 2, we have described the implementation of the firmware. The implementation included drivers for transmitter (ITLA), switches (SERCALOs) and Optical Spectrum Analyzer (AXSUN). Further, the chapter focuses on the precise YANG data model interface for the device, higher-level configuration interface over NETCONF, integration of the individual components into a firmware package, demonstration versions, and testing.

In Chapter 3 we demonstrated the practical use of the device: using the newly constructed firmware, we measured the attenuation of the inner device systems, and used the calibrated device to measure the attenuation of a ROADM multiplexer.

As the main result of this thesis, CESNET has obtained a customized in-house solution for measuring the optical network components, which serves as a viable alternative of commercial products. It is expected than the device and the firmware will become a base for creating more complicated and specialized measurement devices.

Bibliography

[Aut] International Society of Automation. *Calibration principles.* URL: https://www.isa.org/pdfs/calibration-principles-chapter1/.

[Axe99] Jan Louise Axelson. *Serial Port Complete: Programming and Circuits for RS-232 and RS-485 Links and Networks with Disk.* Lakeview Research, 1999. ISBN: 0965081923.

[Bah+16] Arshdeep Bahga et al. "Software Defined Things in Manufacturing Networks". In: *Journal of Software Engineering and Applications* 09 (Jan. 2016), pp. 425–438. DOI: 10.4236/jsea.2016.99028.

[Clo] *close(2).* Linux Manual Pages. URL: https://linux.die.net/man/2/close.

[Dri] *Driver (software).* URL: https://en.wikipedia.org/wiki/Driver_(software).

[Fah] Björn Fahller. *trompeloeil.* URL: https://github.com/rollbear/trompeloeil.

[Fib] "Optical Fibers". In: *Fiber-Optic Communication Systems.* John Wiley and Sons, Ltd, 2011. Chap. 2, pp. 24–78. ISBN: 9780470918524.

[For15] Optical Internetworking Forum. *Integrable Tunable Laser Assembly Multi Source Agreement.* 2015.

[Hec02] Jeff. Hecht. *Understanding fiber optics.* Upper Saddle River, NJ: Prentice Hall, 2002. ISBN: 0130278289 9780130278289 013122803X 9780131228030.

[IBMa] IBM. *read(2).* IBM Manual Pages. URL: https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.bpxbd00/rtrea.htm#rtrea.

[IBMb] IBM. *write(2).* IBM Manual Pages. URL: https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.bpxbd00/rtwri.htm.

[Kub] Václav Kubernát. "Tool for configuration and monitoring". URL: https://dspace.cvut.cz/bitstream/handle/10467/83195/F8-BP-2019-Kubernat-Vaclav-thesis.pdf?sequence=-1&isAllowed=y.

[Kun+19] J. Kundrát et al. "Opening up ROADMs: Let Us Build a Disaggregated Open Optical Line System". In: *Journal of Lightwave Technology* 37.16 (2019), pp. 4041–4051. ISSN: 1558-2213. DOI: 10.1109/JLT.2019.2906620.

[Ltd17] Sercalo Microtechnology Ltd. *SC Coaxial Fiber Optic 1xN Switch with Interface.* 2017.

[MB10] Ed. M. Bjorklund. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF).* RFC 6020. RFC Editor, Oct. 2010. URL: https://tools.ietf.org/html/rfc6020.

[MB16] Ed. M. Bjorklund. *The YANG 1.1 Data Modeling Language.* RFC 7950. RFC Editor, Aug. 2016. URL: https://tools.ietf.org/html/rfc7950.

[Mel] Gabi Melman. *spdlog.* URL: https://github.com/gabime/spdlog.

[NP16] U. Nanda and S. K. Pattnaik. "Universal Asynchronous Receiver and Transmitter (UART)". In: *2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS).* Vol. 01. 2016, pp. 1–5. DOI: 10.1109/ICACCS.2016.7586376.

[Ope] *open(2).* Linux Manual Pages. URL: https://linux.die.net/man/2/open.

[Phoa] Pure Photonics. *Low Noise Tunable Laser.*

[Phob] Pure Photonics. *Operating guide for Pure Photonics ITLA PPCL200.*

[RE06] Ed. R. Enns. *NETCONF Configuration Protocol.* RFC 4741. RFC Editor, Dec. 2006. URL: https://tools.ietf.org/html/rfc4741.

[RE+11] Ed. R. Enns et al. *Network Configuration Protocol (NETCONF).* RFC 6241. RFC Editor, June 2011. URL: https://tools.ietf.org/html/rfc6241.

[Str] John Paul Strupp. *What Are Serial Communications?* URL: https://www.cmrr.umn.edu/~strupp/serial.html#2_1.

[Tcg] *tcgetattr(3).* Linux Manual Pages. URL: https://linux.die.net/man/3/tcgetattr.

[Tec13] AXSUN Technologies. *AXSUN Host Interface Communications Specification.* 2013.

[Ter] *Serial programming – termios.* URL: https://en.wikibooks.org/wiki/Serial_Programming/termios.

[Vas] Michal Vasko. *Sysrepo.* URL: https://github.com/sysrepo/sysrepo/.

[Wika] Wikipedia. *C POSIX Library.* URL: https://en.wikipedia.org/wiki/C_POSIX_library.

[Wikb] Wikipedia. *Remote Procedure Call.* URL: https://en.wikipedia.org/wiki/Remote_procedure_call.

[Wikc] Wikipedia. *RS-232.* URL: https://en.wikipedia.org/wiki/RS-232.

[Wikd] Wikipedia. *XML.* URL: https://en.wikipedia.org/wiki/XML.

[Wike] Wikipedia. *XPath.* URL: https://en.wikipedia.org/wiki/XPath.

List of Figures

1 The calibration process on cable. (Image by Hecht [Hec02]) . . . 4

1.1 The structure of Optical Measurement and Calibration Device. The green path from the Transmitter through Switch 1 to Tested device indicates the emitted signal. The red path from the Tested device through Switch 2 to the OSA indicates the measured signal. 8
1.2 The schema of device management with NETCONF and YANG. 8
1.3 Parallel vs. serial communication. 9
1.4 Synchronous vs. asynchronous serial communication. (Image by [Axe99]) 10
1.5 NETCONF layers. (Image by [RE+11]) 13

3.1 OMCD device with the final firmware installed. In the top left part the AXSUN device can be seen, ITLA is located in the middle under the GIGABYTE heatsink, and to the right of ITLA there is the Beaglebone single-board computer. The two SERCALO switches are located right of Beaglebone. 36
3.2 OMCD device during the process of self-calibration. The power meter on the right is used as a reference for the self-calibration of ITLA. 38
3.3 Measuring ROADM with OMCD. The device stacked on top is the ROADM, OMCD is below. 39

List of Tables

1.1 Serial port files on Unix systems. (Table by Strupp [Str].) 10
1.2 Defined constants for the `c_cflag` (Table by [Str].) 12

2.1 Directory layout of the implementation source code. 18
2.2 Structure of the request packet for ITLA. 22
2.3 Structure of the response packet from ITLA. 22
2.4 ITLA: Packet status table. 23
2.5 Structure of the request packet for AXSUN. 28
2.6 Structure of the request packet from AXSUN. 28

3.1 Results of measuring the attenuation on ports of ITLA and AXSUN. (Only the first 6 ports were measured.) 38
3.2 Connected ports of ROADM to OMCD. 39
3.3 Measurements of the ports E1–E4 on ROADM (on OMCD ports 1–4) obtained by sending the signal to port 1 (in OMCD port 5) and to port 2 (in OMCD port 6). 40
3.4 Measurements of the ports 1 and 2 on ROADM (on OMCD ports 5 and 6) obtained by sending the signal to port E1 (in OMCD 1) and to port E1 (in OMCD port 2). 40