YRKESHÖGSKOLAN
NOVIA

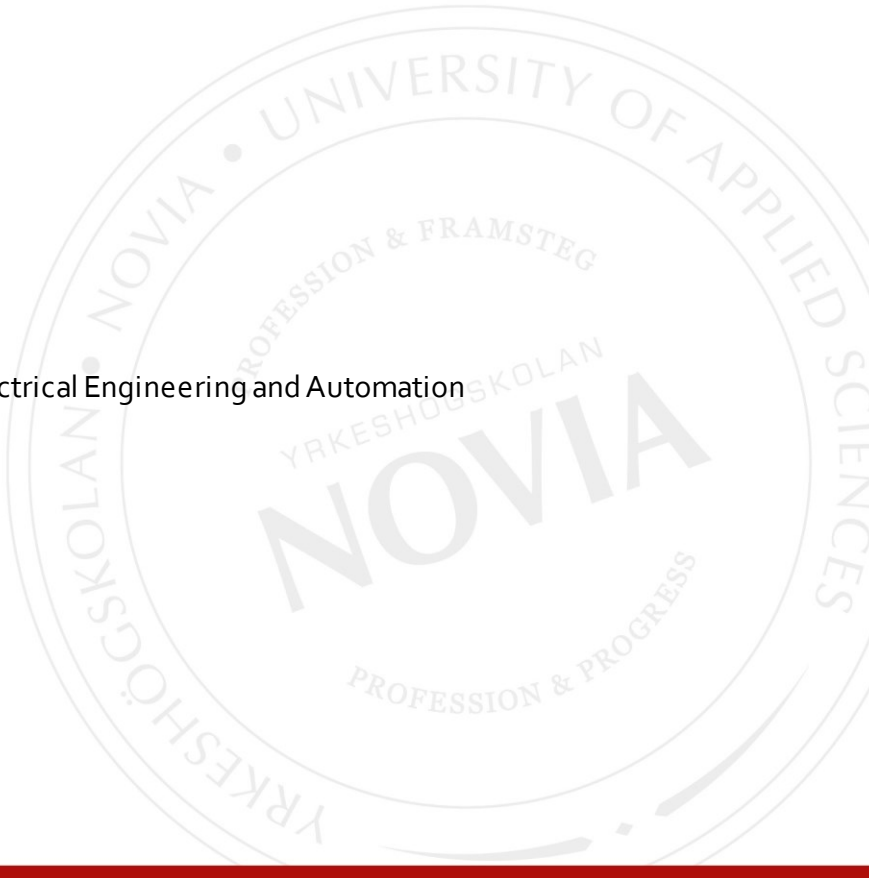
Development of Calibrator Driver Package

William Haavisto

Bachelor's thesis

Degree Programme in Electrical Engineering and Automation

Vasa 2021



EXAMENSARBETE

Författare:	William Haavisto
Utbildning och ort:	El-och automationsteknik, Vasa
Inriktningsalternativ:	Informationsteknik
Handledare:	Kaj Wikman

Titel: Utveckling av kalibratordrivrutinpaket

Datum: 18.3.2021 Sidantal: 35

Abstrakt

Examensarbetet omfattar utvecklingen av ett nytt kalibratordrivrutinpaket. Verktyget som används för att utveckla drivrutinen var 2020-versionen av LabVIEW, där några av dess nya funktioner kommer att undersökas och implementeras om det visar sig förbättra projektet.

Det nuvarande systemet är produktorienterat, vilket innebär att det bara stöder ett fåtal enheter. Detta skapar ett beroende på gamla system som kräver konstant underhåll, vilket på lång sikt kan leda till dyra kostnader och svårare att underhålla. För att undvika detta problem måste drivrutinen vara modulärt, konfigurerbart, möjligt att utvidga och vara både bakåt- och framåtkompatibel.

Den teoretiska delen av arbetet beskriver enhetsstandarder och teknologier som används i detta projekt, en kort introduktion till LabVIEWs nya gränssnittfunktion samt en förklaring till varför de verktyg som användes blev valda. I den praktiska delen beskrivs planeringen av projektet, uppbyggnaden av arbetsmiljön och hur projektkraven implementerades och testades. Exempel på hur objektorienterad programmering fungerar i LabVIEW kommer även presenteras.

Resultatet av detta arbete är en flexibel drivrutin utvecklad i LabVIEW som kan stöda och implementera gamla såväl som nya enheter utan att vara tvungen att större förändringar bör göras i systemet.

Språk: engelska Nyckelord: drivrutin, OOP, LabVIEW, kalibrator

---

OPINNÄYTETYÖ

Tekijä:	William Haavisto
Koulutus ja paikkakunta:	Sähkö- ja automaatiotekniikka, Vaasa
Suuntautumisvaihtoehto:	Tietotekniikka
Ohjaajat:	Kaj Wikman

Nimike: Kalibraattorin ajuripaketin kehitys

Päivämäärä: 18.3.2021 Sivumäärä: 35

Tiivistelmä

Tämä opinnäytetyön tavoitteena oli kehittää kalibraattorin uutta ajuripakettia. Ajuri kehitettiin LabVIEW 2020-versiossa, jossa joitain sen uusia ominaisuuksia tutkitaan ja otetaan käyttöön, jos ominaisuudet osoittautuvat hyödylliseksi projektille.

Nykyinen järjestelmä on tuotekeskeinen, eli tukee ainoastaan muutamia laitteita. Tämä luo riippuvuuden vanhoista järjestelmistä, jotka vaativat jatkuvaa ylläpitoa, mikä voi pitkällä tähtäimellä olla kallista ja vaikea ylläpitää. Tämän ongelman välttämiseksi ajurin on oltava modulaarinen, laajennettava, konfiguroitava, sekä taakse että eteenpäin yhteensopiva.

Opinnäytetyön teoriaosassa kuvaillaan projektissa käytettyjä yksikköstandardeja ja tekniikoita, esitellään lyhyesti yhtä LabVIEW:n uutta käyttöliittymäominaisuutta sekä selitetään, miksi käytetyt työkalut valittiin. Käytännönosassa kuvaillaan projektin suunnittelua, työympäristöä ja kuinka vaatimukset toteutettiin ja testattiin. Käytännönosassa esitetään myös esimerkkejä siitä, miten olio-ohjelmointi toimii LabVIEW:ssä.

Tämän työn tuloksena on LabVIEW:ssä kehitetty joustava laiteohjain, johon on mahdollista lisätä tukea vanhoille sekä uusille laitteille tekemättä järjestelmään suuria muutoksia

---

Kieli: englanti Avainsanat: ajuri, olio-ohjelmointi, LabVIEW, kalibraattori

---

BACHELOR'S THESIS

Author:	William Haavisto
Degree Programme:	Electrical Engineering, Vasa
Specialization:	Information Technology
Supervisors:	Kaj Wikman

Title: Development of Calibrator Driver Package

Date: 18.3.2021 Number of pages: 35

Abstract

This bachelor's thesis covers the process of developing a new calibrator driver package. The software used to create the driver was the 2020 version of LabVIEW, where some of its new features will be researched and implemented if proven to be beneficial for the project.

The current system is product oriented, meaning it only supports a few devices. This creates a dependency on old systems that requires constant maintenance, which in the long term can get more expensive and difficult to maintain. To avoid this problem, the driver must be modular, expandable, configurable, and both backward and forward compatible.

The theoretical part of the thesis describes the unit standards and technologies used in this project, a brief introduction to one of LabVIEW's new interface feature as well as an explanation to why the tools that were used were chosen. The practical part describes the planning of the project, the setup of the work environment, and how the requirements were implemented and tested. Examples of how object-oriented programming works in LabVIEW will also be presented.

The result of this thesis is a loosely coupled device driver developed in LabVIEW that can add support for old obsolete devices as well as brand new devices without having to make any major changes to the system.

Language: english Key words: driver, OOP, LabVIEW, calibrator

---

Table of Contents

Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
ARM	Advanced RISC Machine
BIPM	International Bureau of Weights and Measures
COTS	Commercial off-the-shelf
DD	Dynamic Dispatch
DLL	Dynamic-Link Library
EOL	End of Life
FDS	Functional Design Specification
GPL	General Public License
PA	Product Automation
PPL	Packed Project Library
RDP	Remote Desktop Protocol
RSA	Rivest–Shamir–Adleman
RTD	Resistance Temperature Detector
SD	Static Dispatch
SI	International System of Units
SVN	Subversion
URS	User Requirement Specification
VI	Virtual Instrument
VPN	Virtual Private Network

1 Introduction

When hardware becomes obsolete the old system might meet its EOL *(End of Life)* and be replaced. However, depending on how the system is built, this might not be a simple task. For example, if the old system is product oriented, meaning that the system is bound to a specific device or devices, it creates a tightly coupled dependency.

Adding support for new and older devices to the same system becomes difficult, requires constant maintenance, and only gets more expensive the older the system is because of this dependency.

In this thesis, the development and planning of a new device driver package for several types of calibrators was carried out. The purpose was to replace the older driver with a more up-to-date version with additional features. As the older drivers were developed on an older version of LabVIEW, a migration to the newest version had to be done. Another important aspect of the project was to investigate if any of the new features could be implemented and/or replace older code.

1.1 Purpose

The purpose of this project is to reduce having to maintain older systems and to add support for both old and new products to the current system, hence the need of a modular, configurable, and expandable calibrator driver. For example, dependencies would be more loosely coupled, meaning that the program does not need to know what protocol or device it must load until called upon. This would allow every calibrator to use the same base APIs *(Application Programming Interface)* with specific functions for each calibrator model and firmware version. Adding support for new devices would be much easier as the developer would only have to override the common functions and add specific implementations for the new device.

Currently the older drivers are using C header files from the calibrator firmware, which are parsed during runtime and information about commands are either read from files or hard coded in the drivers. This causes problems for both the PA *(Product Automation)* and embedded teams as this creates dependencies between drivers, embedded software, firmware, and driver versions. For example, some commands might have the same id, but different functionality or entirely different ids depending on the calibrator firmware.

1.2 Scope

The current device drivers support a variety of devices, but this project will only focus on the drivers for the calibrators. The implementation to the new system is also not included in this project scope, as the implementation would be done by another developer and the new system that would implement the new drivers is currently not used in production.

1.3 Goals

The desired outcome of this project is to implement a feature that ask the user whether they want to load configurations based on the current calibrator firmware or not. If users choose to use it, the firmware control will load the commands and settings based on the calibrator's firmware and if not, it will simply load default commands and settings that are commonly used by every calibrator.

Implementation of pressure calibration and electrical calibration, adjustment and definition were also planned to be implemented into the new drivers. The new drivers should be backward and forward compatible, meaning that adding support to a new device would be easy and old devices would still be supported.

2 Theory

The theory section will go more in-depth about the components used in this project and give the reader a general understanding of the project.

2.1 Device driver

According to Microsoft's official documentation, "a device driver is a software component that lets the operating system and device communicate with each other". [1]. In other words, a device driver can be thought of as a broker between software and hardware.

This allows the software to explain what it wants to achieve by providing the device driver with information that it is programmed to understand, which it then forwards to the hardware where the command is then executed.

The advantage of the device driver is that the software does not need to know how to directly work with the hardware and no user interface is needed for the driver. Instead, only the

device driver and software must know how to interfere with each other, which in turn allows much less dependencies between the components.

Without device drivers the process of developing new software and hardware would be nearly impossible since manufacturers would have to know exactly how to communicate with other manufacturers hardware and/or software. [2].

According to MacAfee's Consumer Security Evangelist Gary Davis, the number of connected devices was predicted to reach 50 billion in 2020. This number include all the devices, machines and sensors that currently exists. [3].

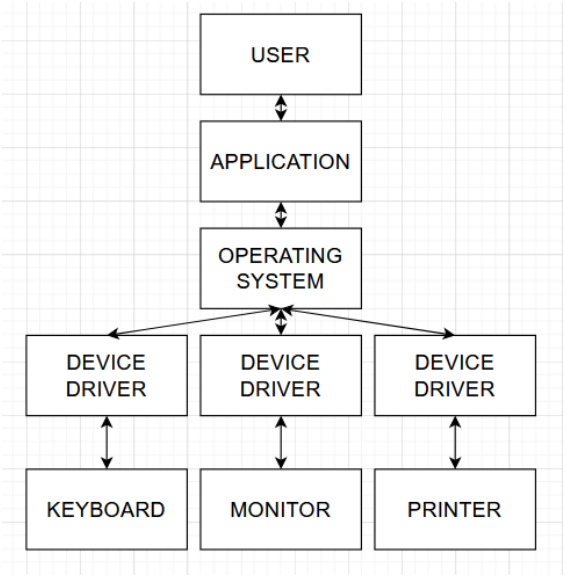


Figure 1: The purpose of a device driver.

When creating a driver, developers must consider which operating system the driver is intended to work on since a driver intended for Windows OS might not work on MacOS or Linux and vice versa.

Not all devices require the manufacturer to create own device drivers. Microsoft provides built-in drivers and automatically downloads the newest version of the driver for an easier setup. A good example of this is when a USB flash drive is connected to the computer, Windows will use the generic USB device driver. Thanks to this, manufacturers do not need to develop drivers for USB devices such as keyboards, computer mice or other types of peripherals.

Sometimes, however, there might be a need to manually install the device drivers. The core drivers are developed by the device manufacturer and the drivers Windows provide are usually stripped-down versions of these drivers. The device might work with the drivers automatically provided but if the device has special functions, the appropriate drivers are required for the device to function properly. [4].

2.2 Calibrator

Since the beginning of recorded history, people have been using tools similar to calibrators even if rather primitive compared to modern equipment. In ancient Egypt, standard units of measurement were used. One of the standards of length was the length of an Egyptian pharaoh's forearm between his elbow and the tip of his middle finger. Bars of this length were made and used as tools called cubit rods. [5]

BIPM (*International Bureau of Weights and Measures*), which coordinates the worldwide measurement system and aims to unify measurements worldwide, formally defines "calibrator" as "a measurement standard used in calibration". [6].

Calibrators are instruments used as references, where the measurement is taken from the calibrator and compared against another less accurate instruments measurements to determine how accurate the instrument is compared to the calibrator.

A calibrator can also be used as a source rather than a measuring device. The process is similar but instead the instrument that is to be tested is connected to the calibrator and a unit with a known value, for example 50 °C, is sent from the calibrator to the instrument.

The result from the instrument can then be calculated to determine whether it is within range of its specification. Measurements taken with a calibrator are usually traceable to one or more of the SI (*International System of Units*) base units.

The general steps to perform a calibration is as follows. [6].

1. Take a measurement with the calibrator.
2. Take a measurement with the instrument that is to be tested.
3. Evaluate the uncertainty of the measurements.
4. Calculate the difference between the measurements to get the error between the calibrator and the instrument.
5. Document the taken measurements and results.

6. Repeat these steps for as many times as required by the calibration procedure.
7. After enough measurements have been taken and compared, verify that the instrument fulfills the product specification. If it does not, determine whether to repair or adjust the instrument and then re-calibrate it to confirm that it performs within the range of the specification.

2.3 Calibration

BIPM formally defines "calibration" as "the documented comparison of the measurement device to be calibrated against a traceable reference device". [7].

A calibration is performed when a calibrator and another instrument is compared. In some cases, the reference may not always be a calibrator but instead a mechanical part, physical reference, liquid, or gas for example. After a calibration has been performed, the result can indicate that the instrument might be inaccurate. The process of adjusting the accuracy is called adjustment or trimming. Contrary to popular belief, adjustment is not formally part of the calibration process but instead a separate process.

2.3.1 Measurement Standard

The measurement standards of the known values used when calibrating an instrument follow the earlier mentioned SI system which is maintained by the BIPM. The SI consists of 7 base units known as ampere, candela, kelvin, kilogram, meter, mole and second. These base units derive from constants of nature. On 16 November 2018, representatives from 60 countries voted to revise the SI, changing the definition of its base units. The decision that was made meant that on 20 May 2019 and from that point onward, all SI units were to be defined in terms of constants that describe the natural world. [8].

Defining constant	Symbol	Numerical value	SI Unit
hyperfine transition frequency of Cs	$\Delta\nu_{Cs}$	9 192 631 770	Hz
speed of light in vacuum	c	299 792 458	$m\ s^{-1}$
Planck constant	h	$6.626\ 070\ 15 \times 10^{-34}$	J s
elementary charge	e	$1.602\ 176\ 634 \times 10^{-19}$	C
Boltzmann constant	k	$1.380\ 649 \times 10^{-23}$	$J\ K^{-1}$
Avogadro constant	N_A	$6.022\ 140\ 76 \times 10^{23}$	mol^{-1}
luminous efficacy	K_{cd}	683	$lm\ W^{-1}$

Figure 2: SI Constants and Base Units. [9].

The advantage of using the SI system is interoperability. If every company were to use the same measurements and definitions when manufacturing products, it would not matter which part of the world the products originate from as the measurements would follow the same standard. To have interoperability, all measurements most be traceable to the same reference.

2.3.2 Traceability

The reference standard must be traceable. This means that the reference standard used must have been calibrated using a higher-level standard, where the high-level calibration has been done in a national calibration center or equivalent. Traceability can be thought of as an unbroken chain of calibrations. If the chain is broken at any point, measurements below that point will be rendered unreliable. Every step in the traceability must be documented and included in a calibration certificate.

2.3.3 Calibration Certificate

As earlier mentioned in BIPMs formal definition of calibration, the word “documentation” is included. This implies that for every calibration performed, the calibration must be recorded. This document is commonly referred to as a calibration certificate. This certificate includes the results of the calibration and information such as equipment, environment, uncertainty of the calibration, date of calibration and signatures. A calibration certificate gives reassurance that an instrument has been calibrated correctly and show proof that a calibration has been performed. [7].

Calibration certificates may vary since calibration laboratories might not follow the same standards and the contents of the certificate can also vary depending on where the calibration fits in the traceability pyramid.

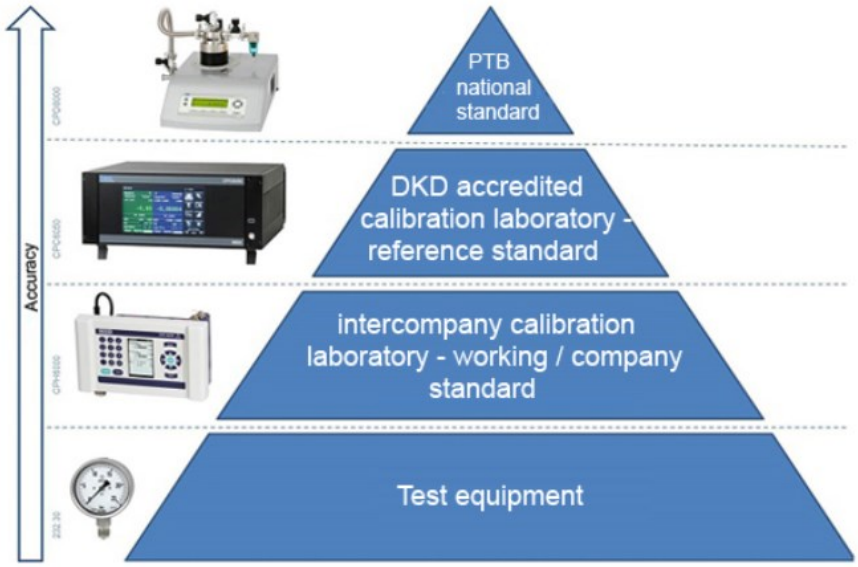


Figure 3: Traceability Pyramid. [9]

For example, a certificate of a calibration of a cooler done at a convenience store contains much less data than a certificate of a precision adjustment calibration done at a national calibration laboratory.

2.3.4 Uncertainty

When an instrument is calibrated with a higher-level device, there will always be some uncertainty which can be thought of as an amount of doubt in the calibration process to indicate how well the calibration process went. Uncertainty can be caused by reason such as the instrument which is under test, the reference standard, environmental conditions, and calibration methods used.

In case the uncertainty of the calibration process is larger than that of the calibrated instruments tolerance level, the calibration will not make much sense. The aim is to keep the uncertainty small enough compared to the tolerance limits of the calibrated instrument.

Error and uncertainty should not be mixed. In calibration, the error is the difference between the calibration results of comparing a reference standard to an instrument. The error has no meaning unless the uncertainty of the measurement is known. [7].

2.4 LabVIEW

First launched in 1986, LabVIEW, which is short for Laboratory Virtual Instrument Engineering Workbench, is a software development environment commonly used for data acquisition and automation control developed by National Instruments. The programming language used in LabVIEW is known as G. What differentiates G from other programming languages such as C and Java, is that G is a graphical programming language. Instead of writing text-based code, the applications are created with the help of block diagrams.

LabVIEW applications are called VIs *(Virtual Instrument).* The reason for this is that their appearance and functionality is comparable to physical laboratory instruments, such as oscilloscopes and meters. A VI is split into two parts, a front panel, and a block diagram. The front panel contains all the buttons, graphs and controls the user can interact with while the block diagram contains all the functionality of the virtual instrument.

2.4.1 Area of application

Listed below are four common scenarios where LabVIEW is applied. [10] .

- **Automated manufacturing tests of components and systems.**

Test systems are used to verify that the product is within the specifications given by the manufacturer. The primary reason for these types of tests is test consistency, error reduction, improvements, and increased reliability.

- **Automated product design validations of components and systems.**

A product validation test is used to validate that the product design is working as intended during the design process before the manufacturer starts producing the product. The quantities that must be tested, such as temperature and voltage, can be rather vast and might have to be repeated several times. Hence, design validation tests save a lot of time in data collection and analyzation.

- **Control and/or monitoring of machines, industrial equipment, and processes.**

Using LabVIEW to control and/or monitor embedded industrial applications in rapid prototyping and development using COTS (*Commercial off-the-shelf*) hardware, tolerance timing and acquisition of high-speed signals.

- **Condition monitoring of machines and industrial equipment**

Condition monitoring is generally used to either improve a machines reliability or reducing maintenance costs.

2.4.2 Creating a project

When creating a new project in LabVIEW, the user will be presented with an empty front panel and block diagram. Right clicking on the block diagram will bring up a palette which contains functions, constants, and structures.

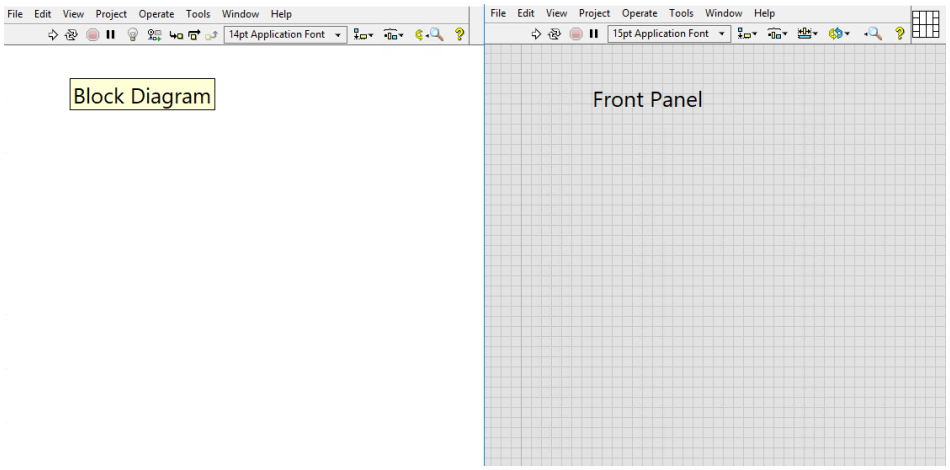


Figure 4: Empty front panel and block diagram

To place a block in the block diagram, the user simply drags and drops the blocks. The different blocks are then connected with wires to create a VI with the desired functionality. Inputs are usually placed on the left and outputs on the right. When adding an input or output to a function or VI, the user can choose between a controller, constant or indicator.

A controller lets the user select and change values and can be accessed from the front panel, while a constant can only be accessed from the block diagram and has a value that stays the same until the developer changes it, meaning that a user does not have access to it. An indicator presents the results of the VI after the process is finished.

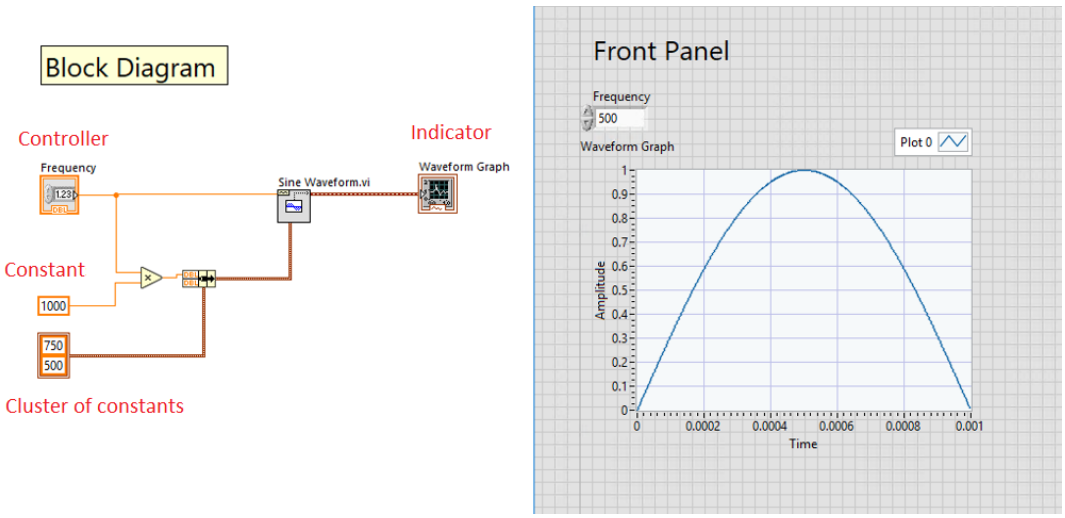


Figure 5: Block diagram and front panel of the Single Sine Wave.VI

As projects get more complicated and the number of structures and wires increase, code review and debugging can become difficult as complex LabVIEW project tend to be wide and the LabVIEW window can easily take up all the space on the computer monitor. To keep code more organised and easier to read, creating a subVI might be a good idea. A subVI is a VI within a VI and can be compared to a function in text-based programming languages.

SubVIs can be used to split up parts of a large VI into several smaller VIs, which is easier to keep track of and troubleshoot.

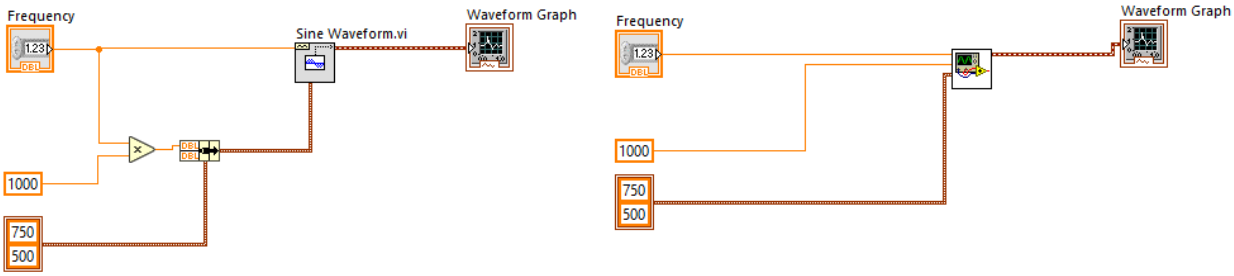


Figure 6: Before and after creating a subVI

A SubVI is created by selecting a section of the block diagram. The selection is then turned into a SubVI with the correct inputs and outputs wired to the connector pane. In case the user requires additional inputs and/or outputs, to change the layout of the connector pane or edit wire properties, it can be done by changing the properties of the VI in the upper right corner.

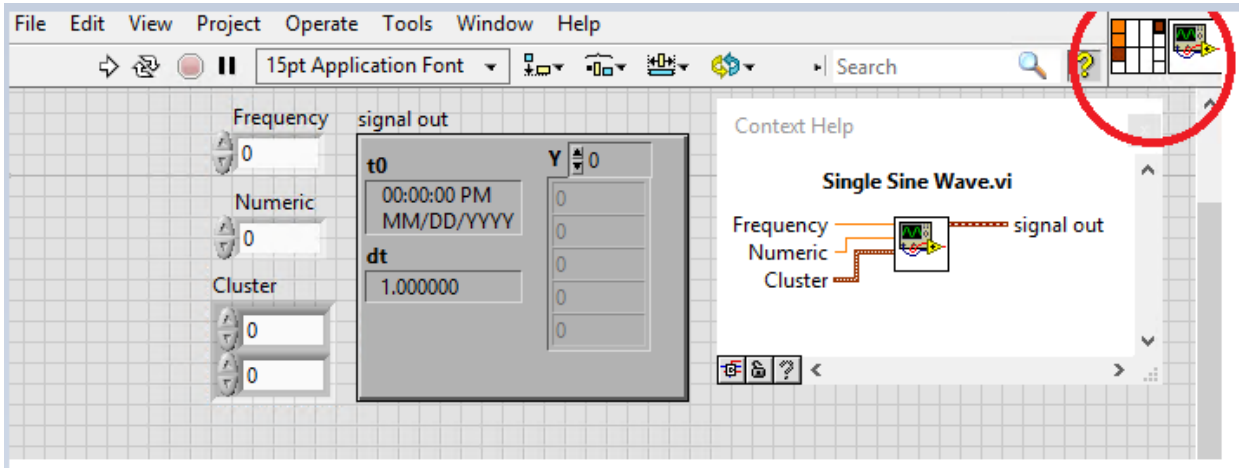


Figure 7: The connector pane of a VI can be accessed from the Front Panel.

When a VI is created, a default icon is automatically added. It is good practice to change the icon and give the VI a description of the functionality so other developers has an easier time understanding the purpose of the VI without having to spend unnecessary time figuring it out on their own.

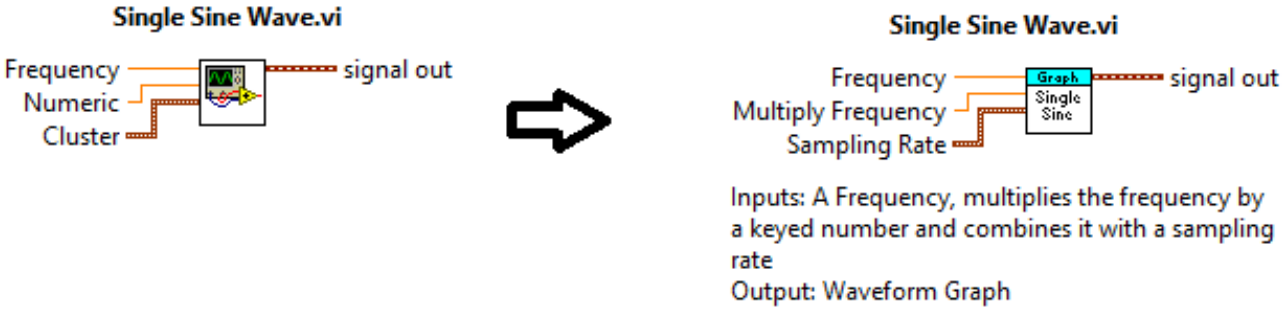


Figure 8:Before and after adding icons and descriptions.

2.4.3 Object-Oriented LabVIEW

OOP *(Object-Oriented Programming)* in LabVIEW follows the same concepts as other OOP languages such as C#. This includes inheritance, encapsulation, polymorphism, and classes. OOP shifts the programmers focus from functionality to data. The key difference between functional programming and OOP is that in functional programming data cannot be stored in objects while OOP does the opposite. The benefits of OOP are modular and reusable code which is easier to maintain and modify.

OOP is a good choice for larger projects, but for smaller projects functional programming might be the better choice in case there is not much to maintain, as projects are created faster and easier. [11].

Classes

In LabVIEW, a class is a specialization of the project library type. The only difference being that while libraries can be within libraries, the same is not possible with classes. A LabVIEW class contains an object and a set of methods. The object is a private data cluster, and the methods are member VIs of the class with access to the data. As the data is always private, it can only be accessed through the member VIs of the class.

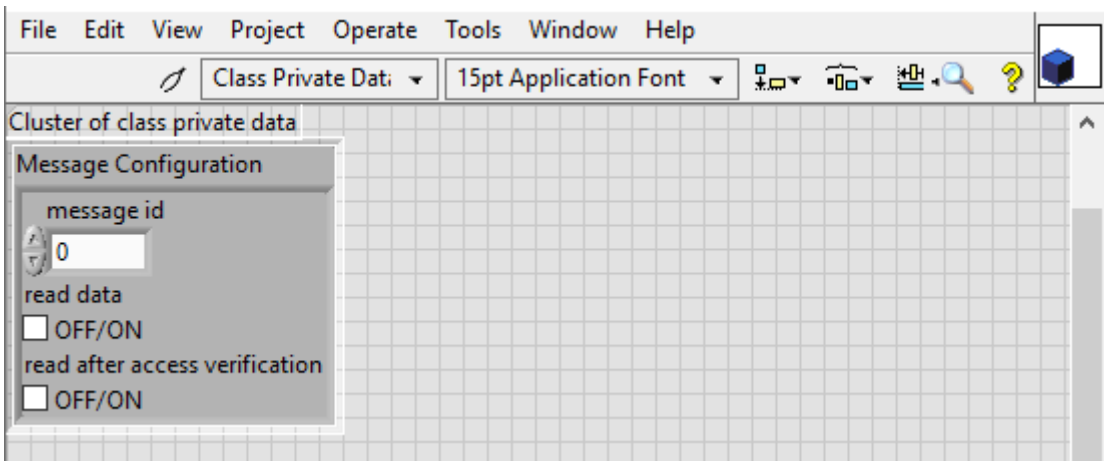


Figure 9: Example of data a class can contain.

Inheritance

An existing class is the starting point for a new class, meaning that a child class inherits a parent classes functionality and overrides and/or reuses said functionality. The benefit with inheritance is that code can often be reused, is more stable since child classes are built on an already working parent class and is easier to maintain as common functionality does not have to be coded again and code duplication is avoided.

Encapsulation

Encapsulation can be thought of as consolidation of data and methods into an object, with restricted access. As earlier mentioned, a class is a cluster of private data. To be able to access the data, a member VI is created.

The class member VIs accessibility can then be selected. With encapsulation, modular code can be created that is easy to update or change without affecting other sections of code in the project.

The class data is always private but the accessibility of the member VIs that expose it can be changed to either public, private, protected or community. For example, scoping a member VI to public allows any VI to call on the member VI as a subVI while scoping to protected only allows VIs within the same class or inheriting classes to call on the member VIs. Access scoping is used to define which VIs are supposed to be called on as subVIs and those which are not supposed to be called on directly. [11].

Polymorphism

Polymorphism is the ability to adapt and accept input data of different data types automatically. A class can also adapt to the input data while the program is running.

A run-time polymorphic method in LabVIEW is known as a DD *(dynamic dispatch),* while a non-polymorphic method is known as a SD *(static dispatch).* The difference being that a DD allows for child classes to override inherited VIs functionality if the inputs and outputs are the same as the parent classes, while SD does not allow child classes to override.

The application does not need to know which class to load before being executed as it can dynamically load the classes during runtime. For example, a parent class that is being loaded can have VIs 1 and 2 loaded while the inheriting child class also loads the same VIs, but also its own member VIs 4 and 5. This again allows code to be reusable and flexible, as both old and new products are easier to implement. [12].

3 Tools and Methods

This section goes into detail about the equipment and tools used during the project and work methods used to keep track of the work progress.

3.1 Test Environment

The plan was to be able to work remotely, thus two computers were needed. One for the developer and one in the laboratory.

To be able to connect to the local computer remotely, a VPN *(Virtual Private Network)* connection was needed. A webcam was also set up, because the developer had to be able to see the screens of the calibrators when a command was tested to be able to confirm that it was working properly.

In the beginning of the project, TeamViewer was considered as an option for the developer to be able to remotely connect to the local computer. However, Microsoft's RDP (*Remote Desktop Protocol*) was opted to be used instead. While RDP requires the user to get the IP address of the computer to be remotely connected and change firewall settings, TeamViewer only has to be installed on both the host and remote computer. TeamViewer works on all operating systems, while RDP only works on Windows.

A disadvantage of RDP is that it presents a static point of attack for hackers and bots since RDP connections are usually protected by a username and password. The hacker might be able to access the remote computer with guessing the correct username and password, since there is no limit of tries before getting the right combination.

TeamViewer does not have this problem since security was taken into account during development. Compared to RDP, TeamViewer does not require to be opened, and uses RSA (*Rivest–Shamir–Adleman*) public-key cryptosystem and AES (*Advanced Encryption Standard*) session encryption. The private key never leaves the client computer, which ensures that other computers or users cannot decipher the data stream between the host and remote computer. [13].

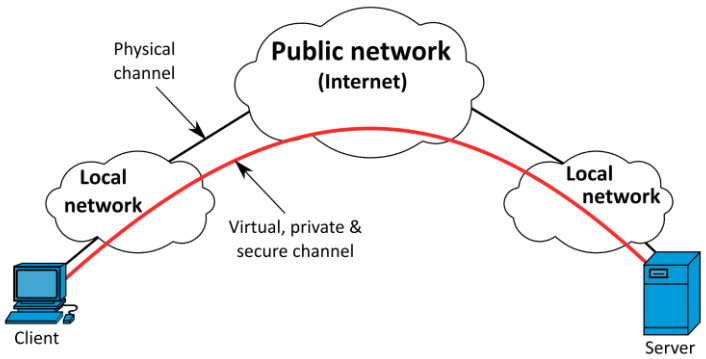


Figure 10: VPN connectivity. [14].

However, if the computer using RDP is behind a VPN, the vulnerability significantly decreases as the computer is no longer exposed directly to the internet and can no longer be accessed by hackers or bots without being on the same VPN.

The reason this project used RDP instead of TeamViewer is because a VPN connection was already in use. RDP is also already installed on all Windows computers and is completely free, while TeamViewer is only free for personal use.

This project used TortoiseSVN for revision control. In short, TortoiseSVN is an Apache Subversion client which has been implemented as a Windows shell extension. It does not require a subversion command line client to run and provides a simple user interface, making it easy to use. [15].

As it is developed under GNU GPL (*General Public License*), it is completely free to use without restrictions. What makes TortoiseSVN unique is that it can be used in any development environment with any tools and type of files. [16].

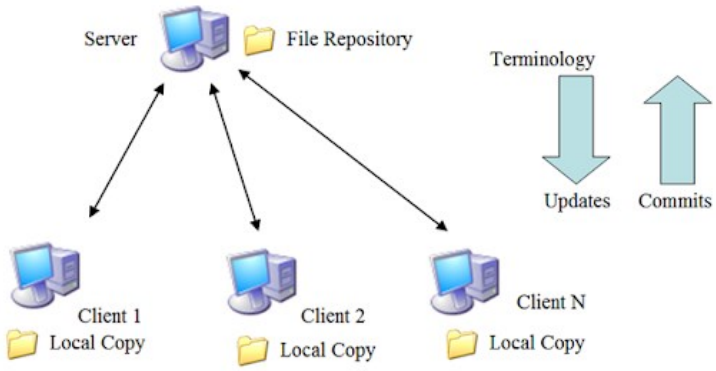


Figure 11: SVN (*Subversion*) Architecture. [17].

3.2 LabVIEW

The existing project was made in an older version of LabVIEW and the goal was to import all the needed projects to the newest version, LabVIEW 2020. A concern was that some VIs might get broken when importing them to the newest version, since some functionality might have changed between the versions.

Currently, when a project is built, a PPL *(Packed Project Library)* is created. A PPL is a compiled version of a LabVIEW project library with all its content. A project library allows for more modular code, scoping code such as making it public or private, and provides a way to give VIs the same names in different libraries even if they are all in the same application. A PPL can be compared to a DLL (*Dynamic-Link Library*).

The projects had to be opened in a specific order, since some projects were dependant on other PPL. If a project is opened without the correct PPL, it will result in LabVIEW not being able to compile the project due to missing dependencies. When opening a project with missing dependencies, LabVIEW will ask for the location of the dependency, or in this case the PPL, and after mapping it to the correct location the project can be compiled. After all projects had been opened following the correct order and mapped, the project had been successfully moved to the latest version of LabVIEW.

A part of this project was to investigate if the newly introduced interface feature could be implemented in this project. Interfaces were introduced in LabVIEW 2020, hence being a reason why LabVIEW was upgraded to the newest version.

Interfaces work similarly in LabVIEW. A new data type defines a set of tasks than an object can do without specifying how those tasks must be done. Interfaces are essentially classes without data, private data control and the method which is used to override classes in LabVIEW. A class that inherits from an interface must implement all methods in the interface. A class can only have one parent class but can inherit as many interfaces as it wants. Interfaces can also inherit other interfaces.

Interfaces work similarly in LabVIEW. A new data type defines a set of tasks than an object can do without specifying how those tasks must be done. Interfaces are essentially classes without data, private data control and the Call Parent Class Method node which is used to override classes in LabVIEW.

There are places where interfaces might come in handy. For example, interfaces fix the so called “fragile base class problem”, where developers are forced to place everything in the base class since otherwise the subset will be incorrect.

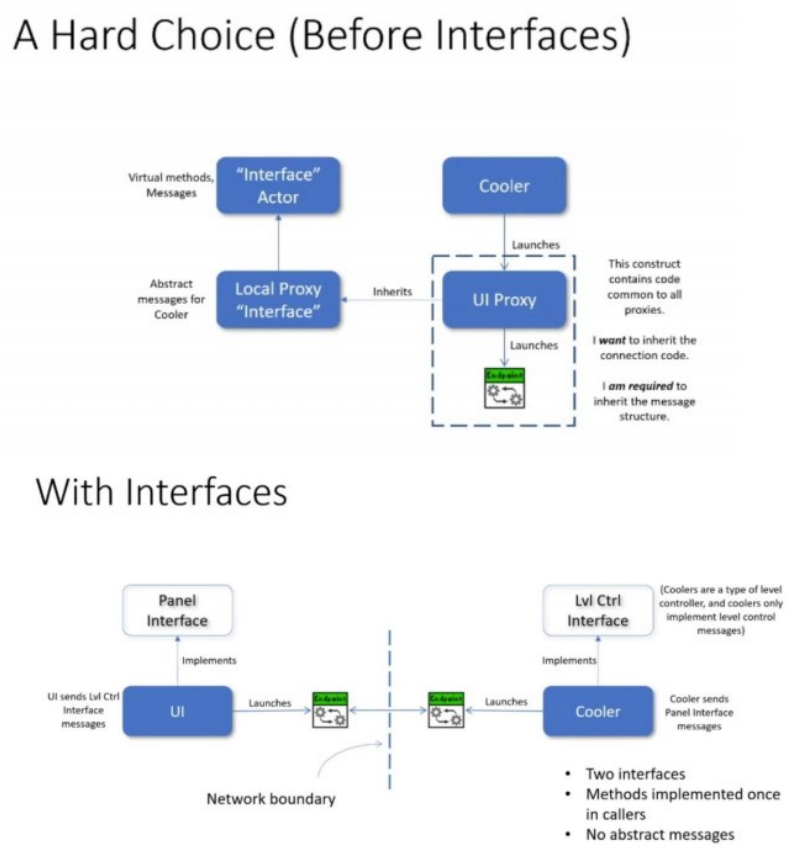


Figure 12: Implementation of interfaces in a cooler project. [18].

In this project a potential place where interfaces could have been implemented was the factory project, which has the task of combining communication and device drivers.

However, interfaces were opted not to be implemented as they were not currently necessary. The reason being that the benefit of implementing interfaces in the current project would have been insignificant and would have required code to be refactored.

4 Assignment

This section goes into detail about how the project was planned. This includes from where the project started, what the requirements were and how the requirements were implemented.

4.1 Planning

The new driver was not to be implemented from scratch but instead it was to be implemented in already existing project and be compatible with the communication, configuration, logger, and factory projects.

A first draft URS (*User Requirement Specification*) had been made and most of the new features and requirements were implemented according to it. With these requirements, a FDS *(Feature Design Specification)* was made to have a clear understanding of which features to implement, in what order and how to implement them. In short, a URS gives an overall picture of the project and an FDS gives an understanding of how the requirements should be fulfilled. It was decided that the project would be split into four different releases. Before a release was pushed to the repository, a risk assessment and a code review had to be made.

4.2 Current Project

The current architecture of the device driver is divided in two projects, one base class and one concrete class. This way the driver becomes configurable, easier to distribute and is "lazy loaded", in other words, only initialized when the object is needed.

The base classes are found in the base class project and are compiled into PPLs. As the base classes do not provide much functionality, it is possible to load them at the same time as the main application is loaded.

The device drivers for the calibrators are implemented in a different project. The classes in the device driver are:

- **Common class** - contains functionality commonly used by all calibrators.
- **Family classes** - inherits from the common class and implements functionality that differs between some calibrator types but are similar between others.
- **Specific calibrator classes** - inherit from the family class and implements functionality specific to a specific calibrator type.

4.3 Releases

First Release

The first release would include a firmware control feature and command needed for pressure calibration.

A configuration file replaced the C header files used in the previous system. Instead of parsing a C header file and load everything that it contained, the configuration file is used to load only the necessary data, such as commands and settings. It would also be possible to

easily add sections for different calibrator types and models and load different settings and commands based on the firmware.

A typical LabVIEW configuration file is separated into three parts:

- **Section** – contains all the keys and key values.
- **Key** – namespace for the key value.
- **Key Value** – contains the value of the key, usually a numeric or a string value.

A configuration file had already been created to replace the C header files but lacked the data and structure for this project, as it was still under development.

A firmware control feature was needed, since depending on the firmware in the calibrator, commands could have different functionality or different ids.

This had caused some inconveniences in the PA and embedded teams when new firmware was developed and command functionality or properties were modified, as the current driver used a configuration file that did not allow multiple firmware versions.

Second Release

The second release included commands used for electrical calibration as well as access verification. Access verification is used when a restricted command is called upon, meaning that the command will only execute if the access verification is correct.

In this case, when the PC uses a restricted command, the calibrator will send a message back to the PC. The PC must then parse the message and return the correct response back to the calibrator before the command can be executed.

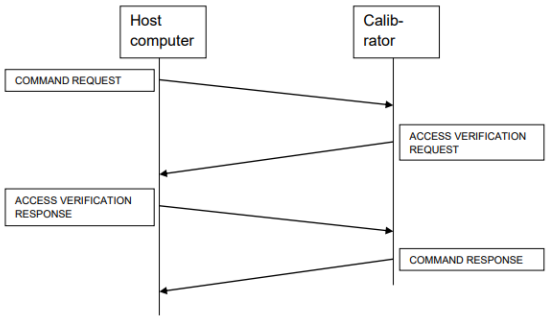


Figure 13: Illustration of Access Verification

Third Release

The third release would include commands used for electrical adjustment.

Fourth Release

The fourth release include commands used for electrical definition.

5 Implementation

This section goes into detail how the requirements mentioned in the planning phase were implemented in the new device driver.

5.1 First release

Firmware Control

In this project, the section was the calibrator, the key was the firmware, and the key value told the application which settings and commands to load. The problem was that the key had to be unique and adding more firmware keys for example “firmware section 1”, “firmware section2” would not have solved the problem, since the application might not have known which firmware to load during run-time.

```
; version control JSON string with settings and commands
version settings = [{"min":0,"max":20,"settings":"settings section1","command":"command section1"},
{"min":20,"max":40,"settings":"settings section2","command":"command section2"}]
```

Figure 14: Example of firmware settings in a JSON array

The solution on how to implement the firmware control was to modify the existing configuration file to include a JSON string which had an array of firmware settings and commands.

The firmware control has been in the Dataloader. The Dataloaders purpose is to parse the configuration file for settings and commands and store them in memory during run-time. A rundown of the firmware control is as follows:

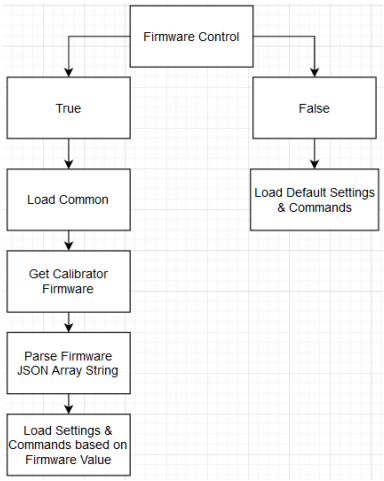


Figure 15: Flowchart of Firmware Control

The firmware control parses the configuration file and searches for the version control key section. The value can be either true or false.

If the value is false, default settings and command will be loaded, and the calibrator firmware ignored i.e., the settings and commands will be loaded in the same way the old configuration file was used to load settings and commands.

If the value is true, a common section in the configuration file will be loaded. This common section contains commands that works the same way in every calibrator model and will never be modified.

After the common section has been loaded, a command which reads calibrator information is used. With this command, the major and minor firmware versions are obtained from the calibrator. The firmware versions are then sent to a VI which will load the settings and commands based on the firmware version.

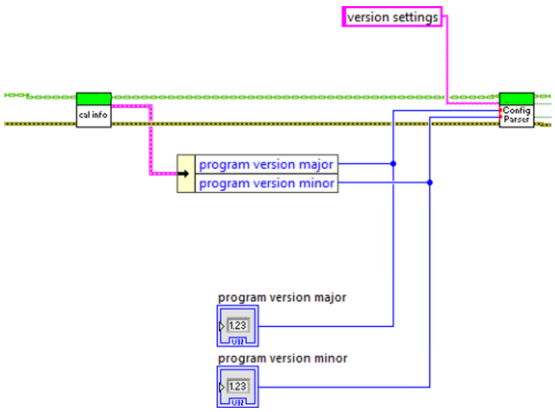


Figure 16: Example of how the firmware version can be obtained.

The configuration parser has three inputs. A section key, which it will search for in the configuration file and two inputs for the major and minor firmware version. The major and minor firmware versions are then combined to a single value.

For example, if the major version is 2 and the minor version is 105 the combined firmware version will be 2.105.

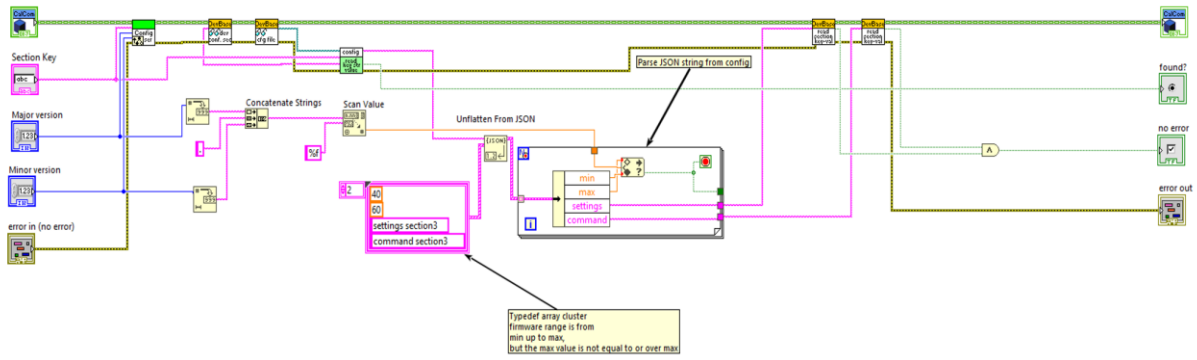


Figure 17: Configuration Parser block diagram.

The section key that this VI searches for contains a JSON array. This array contains different setting and command sections as well as two values, min and max, which the firmware ranges between.

This JSON array is then parsed by a VI called Unflatten from JSON. The JSON array is then converted into a cluster which contains an array which matches the elements found in the JSON array.

For example, the firmware version is 2.105. A for loop which contains the combined firmware version, and an unbundled cluster of the data is looped until a matching value is found. In this case if the firmware is 2.105 the lower limit must be 2. The obtained section names are then wired to a VI that opens the configuration file and searches for the sections. These settings and commands are then loaded during run-time.

Pressure Calibration

The first release implements the commands needed to perform a pressure calibration: These commands are used when calibrating pressure modules and are not restricted by access verification. Commands implemented are:

- **Set Barometric Value**
- **Zero Pressure Module**
- **Select Pressure Type**
- **Select Pressure Measurement**

The commands were first implemented in the base class with a logging function. After that, the family common classes inherited it from the base class with their own implementations.

A new command must be added to the configuration file so it can be used. The commands are also in the form of an JSON array which includes the unique command id.

Every command has a unique id, and the command will not work if the id does not match with the data when communicating with the calibrator.

The inputs of the Select Pressure Measurment.vi is the calibrator measurement channel, the module type, and the pressure type.

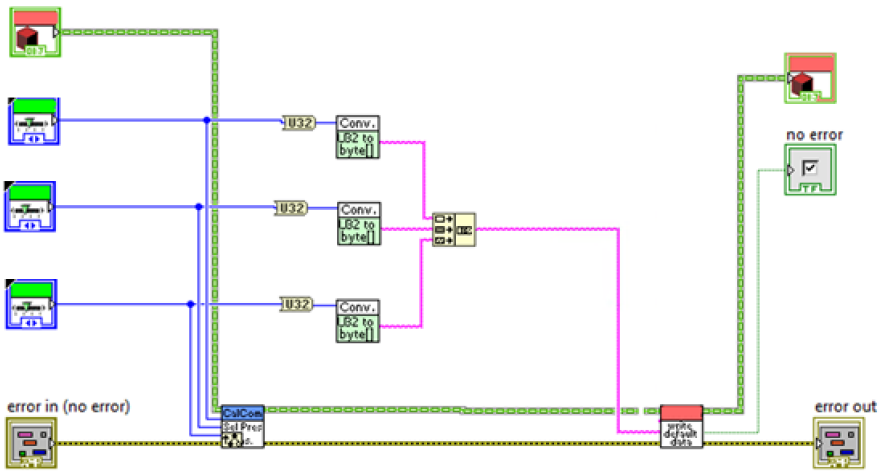


Figure 18: Select Pressure Measurement block diagram.

A VI that takes the data as an input then sends it to the calibrator. When the command is initialized, the configuration file will be opened, parsed and the id of the commands obtained.

When this id is sent to the calibrator, it knows which command to execute and what type of data to expect.

Select Pressure Type only changes the pressure type on an active channel. No module type is needed.

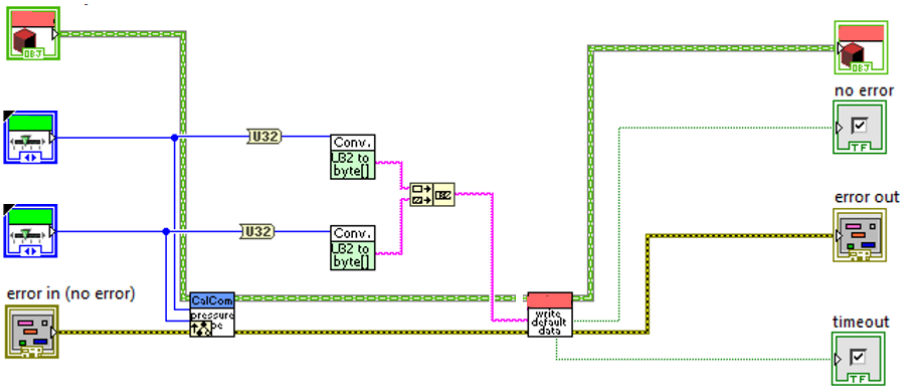


Figure 19: Select Pressure Type block diagram.

The implementation of Zero Pressure Module only needs the channel, as this command will zero all values on the selected channel.

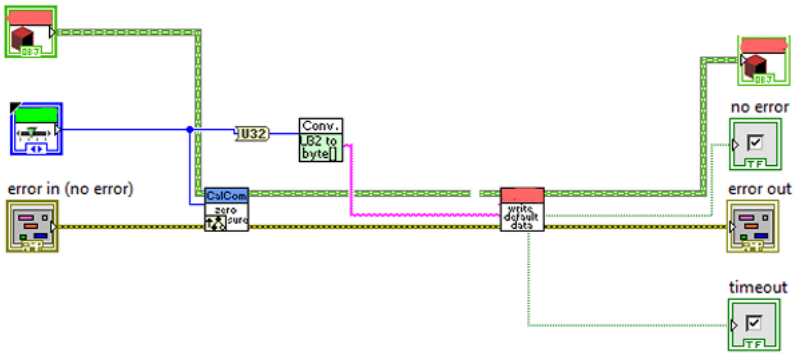


Figure 20: Select Pressure Type block diagram.

The last implementation in the first release was the Set Barometric Pressure Value. This command is used to manually change the barometric pressure value. As such, the inputs of this VI is the channel where the value is to be set as well as the value itself.

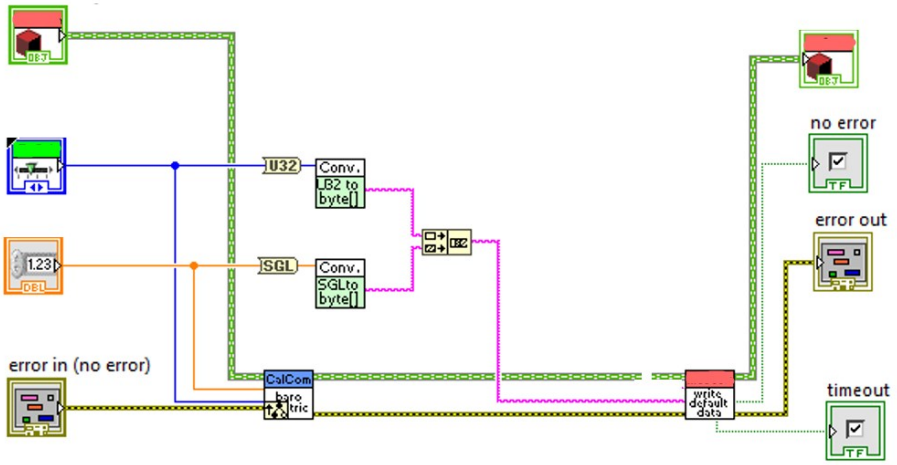


Figure 21: Set Barometric Pressure Value block diagram.

5.2 Second release

Electrical Calibration

The second release implements commands used for electrical calibration. This data is saved in the calibrator and are used for setting a time period in which it can be guaranteed that the value does not creep. This can be compared to when a sensor is manually adjusted to display a certain value. The value is expected to stay within an acceptable range for a certain amount of time before it must be adjusted again.

- **Set Calibration Period**
- **Save Module Calibration Data**

The block diagram can be seen in the figure below for the Set Calibration Period. The module index starts from 0 and represents the first module in the calibrator. The number of modules varies in every calibrator.

A period of days is then given as a signed 16-bit integer. An integer is also sent to the calibrator, which currently has no function but might change in the future.

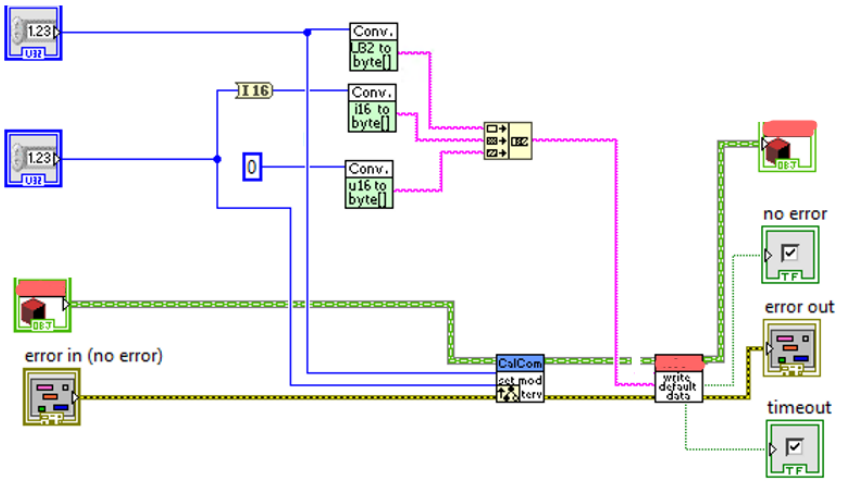


Figure 22: Set Calibration Period block diagram.

The inputs of the Save Module Calibration Data are the index of the module, the name of the laboratory to determine where to module was saved and a timestamp to determine when the module was saved.

The max size for pressure modules is 8 bytes and the name of the laboratory is limited to 28 characters.

Access Verification

Access verification had already been implemented, but it was working incorrectly. The problem was an access verification message was sent to the PC, it sent back the incorrect response.

This was fixed by not using LabVIEW's own Byte Array to String.vi, but instead using a custom byte array to string converter to modify the length and offset of the message, which was then used to obtain the correct message to send back to the calibrator.

Another problem with the current access verification was that it was sending the incorrect id to the calibrator. While the data was correct, the id was not. Instead of sending the id for access verification, the PC was sending the id of the restricted command. Because of this, the calibrator did not authorize use of the command.

To fix this, a new VI to write access verification data to the calibrator was implemented. The VI worked in the same manner as the normal write VI, but instead of using the id that was stored when a command was initialized, it used the id that was obtained when an access verification was initialized.

5.3 Third release

The third release included the commands needed to perform electrical adjustment. Electrical adjustment is used to adjust values of modules to fit the specifications according to the requirements. Commands implemented are:

- **Adjust Zero Point**
- **Adjust Span Point**
- **Save Adjustment Data**

Adjust Span Point and Adjust Zero Point are implemented in the exact same way, the only difference being the name of the command. The calibrator uses linear interpolation to calculate the adjustment values that are in range of but not equal to span or zero points i.e., the calculated values cannot be the same as any of the given points.

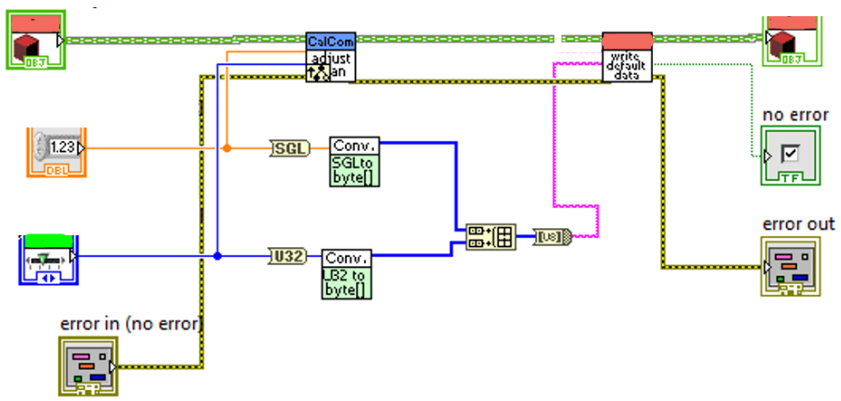


Figure 23: Adjust Span Point block diagram.

The Save Adjustment Data is implemented in the same way as the earlier implemented Save Module Calibration Data.vi, only difference between the two VIs is the name of the command.

5.4 Fourth release

The fourth release implements the commands needed to perform electrical definitions. The module uses the definition data to compensate for ambient temperatures and heat produced by components. Readjustment (trim) of a module at room temperature cancels out the offset and gain errors. These errors accumulate over time but are typically less temperature dependent. Commands implemented are:

- **Set Adjust Coefficients**
- **Set Definition Temperatures**
- **Clear Def Data**
- **Clear Trim Data**
- **Save Def Data**
- **Autotrim**
- **Set Resistance Sim Offset**

The Set Adjust Coefficient is used to set zero and span adjustment setpoint tolerance coefficient.

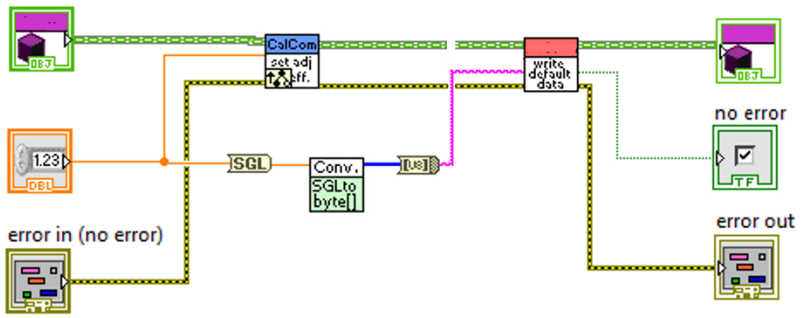


Figure 24: Set Adjust Coefficient block diagram.

The module type is represented as a 32-bit long and the temperatures as an array of floats. The for loop will loop as many times as there are elements in the array, resulting in a byte array.

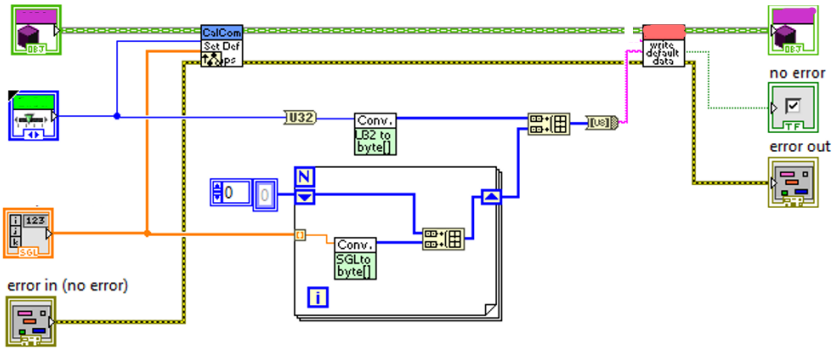


Figure 25: Set Definition Temperatures block diagram.

The Clear Definition Data clears definition data and has a total of 5 inputs:

1. Module type
2. Clear level
3. Range, which has a range index of 0 to 3.
4. Range Sign Type, which is the sign type of the range.

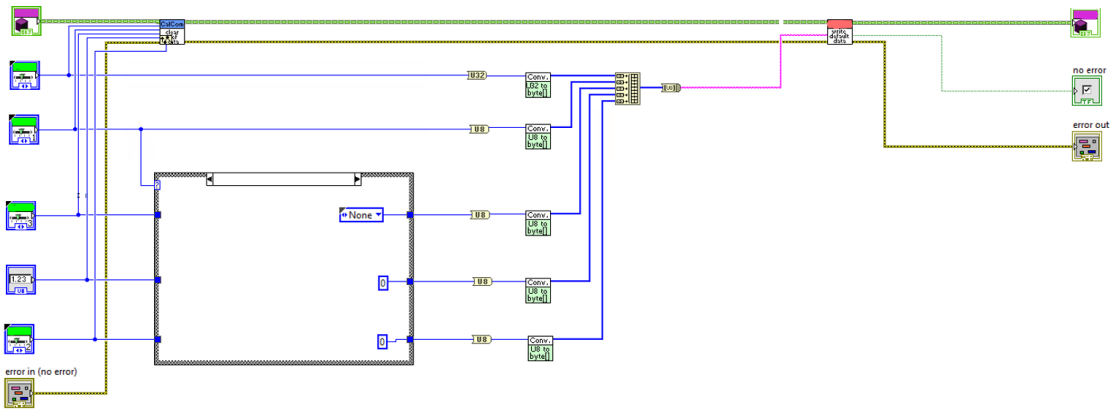


Figure 26: Clear Definition Data block diagram.

Clear Trim Data has the same inputs and works in a similar way to Clear Definition Data but clears adjustment data instead.

Save Definition Data is used to save definition data to a module. The inputs are the module that the data is to be saved to and a date when the data was saved.

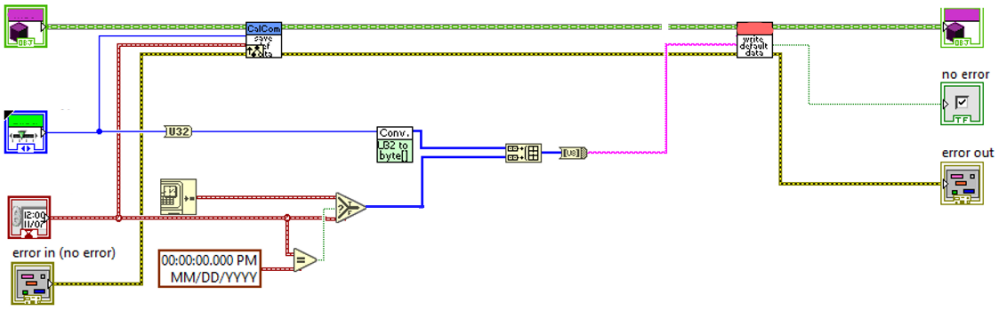


Figure 27: Save Definition Data block diagram.

Autotrim is used to automatically adjust a measurement channel, and as such only the measurement channel is needed as an input.

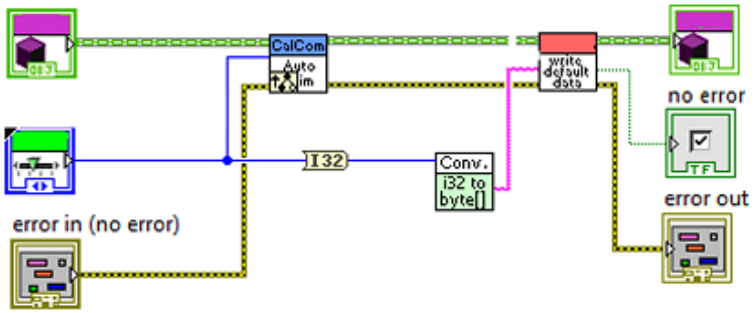


Figure 28: Autotrim block diagram.

Set Resistance Sim Offset is used to set the RTD *(Resistance Temperature Detector)* resistance and offset voltage and are represented as float values.

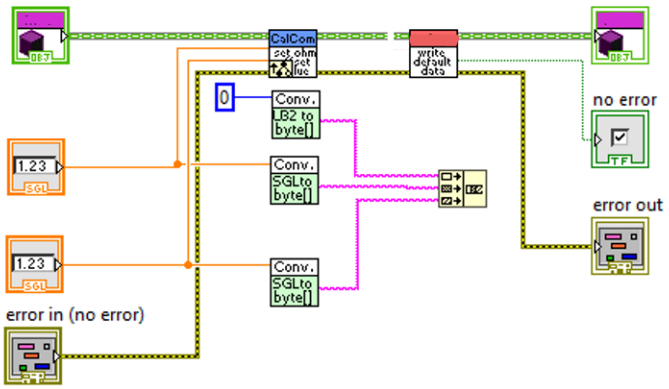


Figure 29: Set Resistance Sim Offset block diagram.

6 Testing the releases

All implementations in the releases had to tested to confirm that they were working correctly. The tests were made in the factory project, which combines the communication and device driver and initializes a session. There were two alternatives to test the implementations. The first one was to use LabVIEW's unit test framework, which is essentially an automated test consisting of three parts:

- **Setup** – Initialize, open connection, and declare inputs of the test case.
- **VI under test** – The VI that is to be tested. Receives data from setup and sends results to teardown.
- **Teardown** – Close connection, declare outputs and excepted results of the test case.
-

Unit test ends with a ". lvtest" extension. A unit test can be started by either right clicking the test in the project folder and selecting "Run" from the dropdown menu or opening the project and pressing the "Run" button. After a test has finished, the test will either pass or fail. If a test fails, errors can be found in the "Test Errors" tab.

Unit tests are useful when a VI must be tested several times. This saves time as the test is fully automated and after the unit test has finished, a test result will show if the test was successful and if errors occurred.

The other alternative was to make a test application for every release where the commands are implemented and tested manually. While it takes longer to make, troubleshooting and finding bugs is faster as it is easier to follow along the wires.

For example, using the probe tool which can be used to check value changes on a wire when the VI is running and step-debugging which pauses on a block in a running VI until resumed and continues running until the next block is encountered. These tools are useful when confirming that values are correct, and that the application performs as expected.

The method that was chosen was to create a test project for each release, since it is easier to find errors and is more flexible than following a framework.

The test project was made as using event-driven programming. An event structure is placed within a while loop. When the VI is initialized, it will keep on looping until a stop button is pressed. When a button that is wired to the event structure is pressed, an event will occur. All VIs inside that event will then be initialized.

To keep the obtained data in memory, shift registers are used. What a shift register essentially does is store data while the loop is running and can be thought of as a local variable.

7 Result

This project was part of a larger project. The goal of this project was to create a driver that was flexible, without dependencies and be back- and forward compatible. With the results from the tests and from the calibrator behaviour, the project can be considered a success.

The new device driver is also not bound to any calibrator type, so new devices are easy to add as the VIs can be overridden and implemented in any way that the device requires.

One of the requirements which was to get rid of the header files used to read the commands was also achieved thanks to the new configuration file and with the implementation of the firmware control feature, commands with different functionality depending on the calibrator's firmware could easily be implemented.

8 Conclusions and discussions

I had no prior experience with LabVIEW before this project. However, as my specialization was in information technology, I was familiar with different text-based programming languages and quickly noticed similarities in LabVIEW. What helped me get started with the LabVIEW environment was National Instruments.

This project has given me a deeper understanding of how projects are planned and the progress of software development. The weekly pulse meetings were very useful. Pulse meetings are when supervisors and employees meet and discuss the current situation of the project and exchange ideas and improvements frequently.

The device driver was not built from scratch but continued from where it was left and with more features added. An understanding of how OOP in LabVIEW differentiates from traditional LabVIEW programming was also a valuable lesson.

In its current state, the device driver can be implemented in the new system. The device driver has been tested and proven to work. As the device driver works in a very similar fashion to the driver in the system currently used, the implementation should be straightforward. It is also possible to make changes and expand the device driver, as classes

can be easily added with own specific implementations. But as the new system is still under development, some changes might have to be made to the new device driver in the future.

The biggest problem was in the first release, where a bug was causing a command to fail. I would probably not have found it as fast as I did if I had used the unit test framework instead of creating test applications.

The problem was in the inheritance, as it had overridden the VI incorrectly and was using an empty implementation instead of the one it was supposed to use. The remaining releases did not take as long to implement as the first one, as by this time I sufficient knowledge on how to implement the rest of the commands.

Lastly, I want to thank Kaj Wikman who acted as my mentor from Novia.

9 References

[1] Microsoft, "What is a driver?," 20 April 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-. [Accessed 4 January 2021].

[2] T. Fisher, "What Is a Device Driver?," 08 May 2020. [Online]. Available: https://www.lifewire.com/what-is-a-device-driver-2625796. [Accessed 5 January 2021].

[3] G. Nick, "How Many IoT Devices Are There in 2020?," 13 October 2020. [Online]. Available: https://techjury.net/blog/how-many-iot-devices-are-there/. [Accessed 4 January 2021].

[4] C. Hoffman, "Should You Use the Hardware Drivers Windows Provides, or Download Your Manufacturer's Drivers?," 26 July 2017. [Online]. Available: https://www.howtogeek.com/191405/should-you-use-the-hardware-drivers-windows-provides-or-download-your-manufacturers-drivers/. [Accessed 5 January 2021].

[5] Brüel & Kjær, "The Birth of Calibration," [Online]. Available: https://www.bksv.com/en/knowledge/blog/perspectives/egyptian-cubit. [Accessed 10 January 2021].

[6] Fluke, "Calibrator: A Comprehensive Introduction," [Online]. Available: https://us.flukecal.com/calibrator. [Accessed 6 January 2021].

[8] Bureau International des Poids et Mesures, "The International System of Units," [Online]. Available: https://www.bipm.org/en/measurement-units/. [Accessed 16 January 2021].

[9] M. Bundschuh, "Traceability," 5 March 2017. [Online]. [Accessed 6 January 2021].

[10] Viewpoint Systems, "What is LabVIEW used for?," [Online]. Available: https://www.viewpointusa.com/labview/what-is-labview-used-for/. [Accessed 28 January 2021].

[11] National Instruments, "Creating LabVIEW Classes," [Online]. Available: https://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/creating_classes/. [Accessed 5 February 2021].

[12] E. Stern, "Object-Oriented LabVIEW: Polymorphism," 7 September 2016. [Online]. Available: https://www.bloomy.com/support/blog/object-oriented-labview-polymorphism-part-3-3-part-series. [Accessed 5 February 2021].

[13] TeamViewer, "Security Overview," [Online]. Available: https://www.teamviewer.com/en/trust-center/security/. [Accessed 1 February 2021].

[14] Wikipedia, "Virtual private network," [Online]. Available: https://en.wikipedia.org/wiki/Virtual_private_network. [Accessed 1 February 2021].

[15] TortoiseSVN, "About TortoiseSVN," [Online]. Available: https://tortoisesvn.net/about.html. [Accessed 15 March 2021].

[16] FinancesOnline, "TortoiseSVN Review," [Online]. Available: https://reviews.financesonline.com/p/tortoisesvn/. [Accessed 1 February 2021].

[17] K. Vikas, "Version control with TortoiseSVN," 8 September 2008. [Online]. Available: http://designpatternschash.blogspot.com/2008/09/what-is-tortoise-svn.html. [Accessed 1 February 2021].

[18] S. Loftus-Mercer and A. Smith, "Introduction to G Interfaces in LabVIEW 2020," [Online]. Available: https://www.youtube.com/watch?v=gQU3eM0yLMk&feature=youtu.be. [Accessed 20 February 2021].

[19] American Innovations, "End of Life Policy," [Online]. Available: https://www.aiworldwide.com/about/legal/end-of-life-policy/. [Accessed 6 January 2021].

[20] Omega, "Introduction to process and temperature instrument calibration," [Online]. Available: https://www.omega.co.uk/prodinfo/calibrators.html. [Accessed 6 January 2021].