Auto-Detection Over Various Protocols In C++

Simplifying the use of RadiMation®

An assignment from

DARE!!

20 MARCH 2020

COMPANY: DARE!! B.V.
Author: Ralph Heij

Version: v3

Auto-Detection Over Various Protocols in C++

Simplifying the Use of RadiMation®

Thesis Graduation Internship

General:	
Date	20-03-2020
Author	Ralph Heij
Version	vA04

Institute:	
Name	De Haagse Hogeschool
Location	Delft
Course	Electrical Engineering
Student Number	15028453
Email	heijralph@outlook.com
1st School Counsellor	Jesse op den Brouw
2nd School Counsellor	Patrick Morley

Company:	
Name	DARE!! B.V.
Location	Woerden
Project Supervisor	John de Rooij

Table of Contents

Table of figures

1 | Foreword

This thesis is written to describe the assignment, solution design and solution implementation during my internship at DARE!! At DARE! I have been given three assignments to simplify and rationalize the user interface of a comprehensive software product called RadiMation®. It also describes how I solved the three assignments they have given me. This internship is an integral part of my study Electrical Engineering and is required to receive my Bachelor's degree from the Haagse Hogeschool. I performed this internship from 18 November 2019 to 20 March 2020 and, as stated, all my activities and experience during this period is brought together in this report.

In addition, this report can be used as a reference book for any follow-up activities to further simplify the user interface of RadiMation® as there are still various simplifications possible to improve the easiness and efficiency of using RadiMation® by the end user.

Although this report is public, it is to be noted that all the information I have gathered during this internship at DARE! is and will remain the intellectual property rights of DARE!!.

Furthermore, this report is also intended to be an official description of my internship for the Haagse Hogeschool.

I would like to thank DARE!! and, in particular, Ing. John de Rooij, for providing me these assignments as well as guidance and useful tips when I was unable to find the way forward. I enjoyed the working atmosphere at DARE! and appreciated feeling very welcome.

I would also like to thank Jesse op den Brouw, my 1st School Counsellor, for giving his guidance and constructive remarks for this thesis.

I trust that this final report provides an adequate insight of the various phases of my internship and about the knowledge and competencies I have built up.

Delft, 20 March 2020
Ralph Heij
Student Electrical Engineering – study year 2019-2020
De Haagse Hogeschool, location Delft.

2 | Summary

2.1 English

This thesis is about my assignment to simplify a complex software product called RadiMation®. RadiMation® is an application, written in C++, and is used to automate a range of EMC testing.

This document also has the purpose of providing information about how the interfaces and device drivers within the application are selected and how, through the changes applied to the source code, the configuration of RadiMation® has been simplified. In essence, the assignment is split into three parts, and is completely focussed on C++ programming and changing the source code to achieve the desired result.

RadiMation® is a software application developed and marketed by DARE!! BV. DARE!! is a company that produces hardware and software focused on the automation of EMC measurements. EMC in short is Electromagnetic Compatibility, basically making sure that no noise or as little noise as possible is generated inside a device that might disturb other devices. Hardware wise, DARE!! has created various devices such as power meters and a RadiCentre, a large hub for different interface cards and devices that support further automation of EMC testing. This RadiCentre is connected to RadiMation® (see page 1).

These assignments are related to the auto-detection of various protocols such as GPIB, RS-232 and USB, as well as the removal of duplicate options to make the configuration for the end user an easier process. The three assignments are the following:

1. Improvement of the auto-detection with USB.

Before the modifications were applied, the detect button would only work when a single device was connected. It would give the user a message to disconnect any other devices and then press the button again. My first assignment was targeted to detect the devices that are connected and give the user the possibility to select the device from a pop-up list.

This first assignment, the improvement of auto-detection with USB, is about finding the name of the currently loaded device driver, fetches a list of all currently connected devices through USB. Then for each device identifies their names and compares it with the name of the loaded device driver. This scan is also possible in case a RadiCentre®, a box with interface cards that is connected to RadiMation®, is used. Depending on the number of matches, there is could be no match, which results in an error message, one match, which results in all the data being filled in automatically, and more than one match, which results in a GUI popping up showing a list of all matches, in which the end user has to make a unique selection from.

2. Removal of duplicate options that confuse the end user.

It used to be confusing for the end user to select the correct protocol as, for example, the GPIB protocol could be chosen through the standard listed protocols, or via the GPIB options through the VISA protocol. My second assignment has the purpose to simplify the selection of possible protocols the software uses to communicate with the device.

The device driver has various protocols that are similar in the type of settings they have, such as GPIB and CompliantGPIB, that both have the same list of settings. This is quite confusing for the end user.

To solve this, the code is modified and now runs a priority system. There is HTTP, TCPIP, RSIB and UDP, all taking one address. There are two variants of USB as well. There are a few protocols that have no alternatives, so they are left as is, such as RS-232. On top of this, there is VISA, a library that can go through all of these communication methods in a better way. Therefore, based on what the end user chose as communication method (HTTP, USB, RS-232 or GPIB), RadiMation® attempts communication through VISA first. If this fails, it will attempt to use the next protocol down the line, until it finds a protocol that works or until it is out of options and results in an error message.

3. Introduction of auto-detection with GPIB and RS-232.

The goal of the third assignment is the introduction of auto-detection with the RS-232 and GPIB protocols, as this was not available. The auto-detection for each of these protocols work in a similar way. Both will create a list of all connected devices and then run a function called CheckDevice() on each connected device, returning an error in case the connected device is not the one of the currently loaded driver. Now, the devices that return no error are added to a list and from there it is decided what has to happen, similar to that described in the USB solution above. No matches return an error message. One match, data will be automatically filled in and more than one match results in a GUI window popping up requesting the end user to make a unique selection.

Although all assignments have been completed and they deliver the desired result, this internship has been quite challenging for its many new and advanced concepts of C++. In combination of the complexity of the application. The complexity of the general structure of the code and the large number of classes (>4500) took a few weeks to comprehend and even then, it wasn't completely clear to me. However, I have been able to significantly improve my C++ capabilities as well as other competences, like Analysis, Research and Solution Design.

Even after the simplification that has been realized by changing sections of the source code, there is still more simplification possible and required as RadiMation® continues to be a quite complex software product.

This report may help others within DARE!! to get quicker access to information required to perform further simplification.

2.2 Nederlands

Dit verslag behandelt mijn afstudeeropdracht en deze opdracht heeft als doel het vereenvoudigen van een complex softwareproduct, genaamd RadiMation®. RadiMation® is een toepassing, ontwikkeld en geschreven in C++, en wordt gebruikt voor het automatiseren van een scala of EMC tests.

Dit document heeft ook tot doel informatie te verschaffen omtrent de wijze waarop de interfaces en device drivers binnen de toepassing worden geselecteerd en hoe, nadat de C++ source code is aangepast, het configureren van RadiMation® is vereenvoudigd. In essentie, de opdracht is gesplitst in drie delen, en is volledig toegespitst op het programmeren in C++ en op het aanpassen van de source code teneinde het beoogde resultaat te verkrijgen.

RadiMation® is een softwaretoepassing die ontwikkeld is door DARE!! B.V. DARE!! is een internationaal georiënteerd bedrijf dat apparatuur en programmatuur ontwikkeld voor EMC-testapplicaties. EMC staat voor Elektromagnetische Compatibiliteit. De definitie van EMC is hoe apparatuur zich verdraagt ten opzichte van haar elektromagnetische omgeving. In andere woorden; in welke mate beïnvloedt apparatuur andere apparatuur of in welke mate wordt apparatuur door andere apparatuur beïnvloed? Vanuit een hardware perspectief, DARE!! heeft een aantal apparaten ontwikkeld zoals Power Meters, Electric Field Probes, Electric Field Generators en een RadiCentre®, een hub voor verschillende interfaces en apparatuur en die aangesloten wordt op RadiMation®. Dit RadiCentre® ondersteunt het verder automatiseren van de EMC tests.

De (sub) opdrachten zijn gericht op het auto-detecteren van verschillende protocollen zoals GPIB, RS-232 en USB, alsmede op het verwijderen van ogenschijnlijk overtollige opties om het configuratie proces voor de eindgebruiker eenvoudiger en minder verwarrend te maken. De drie opdrachten luiden als volgt:

1. Verbetering van de auto-detectie via USB.

Voordat de aanpassingen in de code waren aangebracht, werkte de detect button alleen als er maar een enkel apparaat was aangesloten. Indien meer apparaten weren aangesloten, kreeg de gebruiker een bericht om de overige apparaten te verwijderen en opnieuw op de knop te drukken. Mijn eerste opdracht was gericht op het detecteren van alle aangesloten apparaten en daarna de gebruiker de mogelijkheid te geven uit een pop-up lijst de gewenste apparaten te kiezen.

Deze eerste opdracht, het verbeteren van de auto-detectie via USB, omvat het vinden van de naam van de geladen device driver en het ophalen van een lijst met alle apparaten die via USB verbonden zijn. Daarna, voor elk van de apparaten, het identificeren van de naam en deze te vergelijken met de naam van de geladen device driver. Deze uitgewerkte routine is zelfs ook mogelijk wanneer een RadiCentre®, een apparaat met interface kaarten die met RadiMation® verbonden is, wordt gebruikt. Afhankelijk van het aantal “matches” zijn er verschillende mogelijkheden. In geval van geen match volgt een error bericht, bij één match worden de gegevens automatisch ingevuld en bij meerdere matches verschijnt een GUI pop-up met een lijst van alle matches waaruit de gebruiker een keuze kan maken.

2. Verwijderen van gelijksoortige options die de gebruiker verwarren.

Het selecteren van het juiste protocol is verwarrend voor de eindgebruiker omdat, bij voorbeeld, het GPIB-protocol gekozen kan worden via de standaard lijst van beschikbare protocollen, maar ook via de GPIB-opties via het VISA-protocol. Mijn tweede opdracht heeft als doel dit selectieproces met

betrekking tot de protocolkeuze waarmee met de apparatuur wordt gecommuniceerd, te vereenvoudigen.

De device driver heeft verschillende protocols die vergelijkbaar zijn qua type van de settings, zoals GPIB en CompliantGPIB, die beide dezelfde settings kennen. Dit is erg verwarrend voor de eindgebruiker. Om dit op te lossen is de source code aangepast met een soort van priority routine. We hebben de protocollen HTTP, TCPIP, RSIB en UDP, die alle maar één adres vragen. Er zijn ook twee USB-varianten. Daarnaast zijn er enkele die uniek zijn en geen alternatieven kennen zoals RS-232. Die zijn as-is gelaten. Bovendien is er nog het VISA-protocol, een alternatief die al deze communicatiemethoden optimeert. Daarom, onafhankelijk van de keuze van de communicatiemethode van de gebruiker (HTTP, USB, RS-232 or GPIB), RadiMation® probeert allereerst de communicatie via VISA. Als dit niet lukt, zal worden geprobeerd met het volgende protocol in de lijst te communiceren totdat een werkend protocol wordt gevonden. Zodra het einde van de lijst is bereikt en geen succesvolle communicatie tot stand is gebracht, dan volgt een error message.

3. Implementatie van auto-detectie via GPIB en RS-232.

Het doel van de derde opdracht is de implementatie van auto-detectie voor de protocollen GPIB en RS-232 omdat deze mogelijkheid tot op heden niet ter beschikking stond. De auto-detectie voor elk van deze protocollen werkt op een gelijksoortige manier. Voor beide wordt wederom een lijst van alle aangesloten apparaten gecreëerd en dan wordt voor ieder verbonden apparaat een functie genaamd CheckDevice() aangeroepen. Deze functie retourneert een error melding als het verbonden apparaat niet overeenkomt met de geladen device driver. Vervolgens, de apparaten die geen error retourneren worden opgenomen in een lijst. Vanaf dit punt is het proces zoals bij de auto-detectie via USB. In geval van geen match volgt een error message, bij één match worden de gegevens automatisch ingevuld en bij meerdere matches verschijnt een GUI pop-up met een lijst van alle matches waaruit de gebruiker een keuze kan maken.

Alhoewel alle opdrachten succesvol zijn geïmplementeerd en zij het resultaat leveren dat ook werd verwacht, was deze afstudeeropdracht een echte uitdaging wegens de complexiteit van het softwareproduct waarvan ik de source code moest gaan aanpassen. Ook omdat bij dit programma veel van de nieuwe en geavanceerde mogelijkheden van C++ zijn gebruikt. Wegens de complexiteit van de programmastructuur en het grote aantal Classes (>4500) heeft het enkele weken geduurd voordat ik enigszins begreep hoe een en ander functioneerde. En zelfs dan, maar met enige mate.

Wel kan ik bevestigen dat deze uitdagende opdracht sterk heeft bijgedragen aan het verder ontwikkelen van mij C++ programmeervaardigheden alsmede het uitbouwen van de noodzakelijke competenties zoals Analyse, Research en Solution Design.

Zelfs na de vereenvoudigingen die het gevolg waren van mijn opdrachten, blijft er nog voldoende potentiaal over om RadiMation® verder te vereenvoudigen.

Dit verslag kan daarom ook anderen binnen DARE!! helpen een sneller beeld te krijgen van enkele functionaliteiten en kan wellicht bijdragen tot een snellere toegang tot informatie in geval tot verdere vereenvoudiging wordt besloten.

3 | Introduction

"Electromagnetic compatibility, EMC is the concept of enabling different electronics devices to operate without mutual interference - Electromagnetic Interference, EMI - when they are operated in close proximity to each other. All electronics circuits have the possibility of radiating of picking up unwanted electrical interference which can compromise the operation of one or other of the circuits." [1]

DARE!! B.V. has developed hardware and software, regarding the measurement of EMC. These products detect interferences in any electrical product. DARE!! has made products such as electric field probes, electric field generators, power meters etc. To control these products, a software called RadiMation® is utilized.

This software application has been written and updated over the years. The end user interface is complex and, in some case, even confusing. That is why, within DARE!! BV, it became desirable to simplify certain areas within the application. This graduation assignment addresses that need and starts with simplifying the way how RadiMation® configures the drivers and the protocols with which it communicates with the connected devices.

Within this thesis, first the background of the assignment will be described by a.o. explaining about the company that has given me this assignment and the products and services they provide to the market.

In Chapter 5 the assignment(s) will be described in detail and in the Chapters 6, 7 and 8 one will find the details how the solutions have been designed and built executed. In addition, certain concepts that are not necessarily related to the assignment but will help the reader in understanding it are also included where necessary.

Finally, a detailed description of the competencies that have been improved can be found in Chapter 9 followed by a final conclusion and recommendation in Chapter 10.

4 | The company DARE!!

DARE!! Instruments is specialized in creating measurement equipment for performing automated EMC measurements. Those products are sold worldwide and are high-tech and innovative products using RF and Laser technology.

DARE!! Instruments is a separate branch of DARE!! B.V., which is separated into two entities, DARE!! Services and DARE!! Products. DARE!! Instruments is part of DARE!! Products.

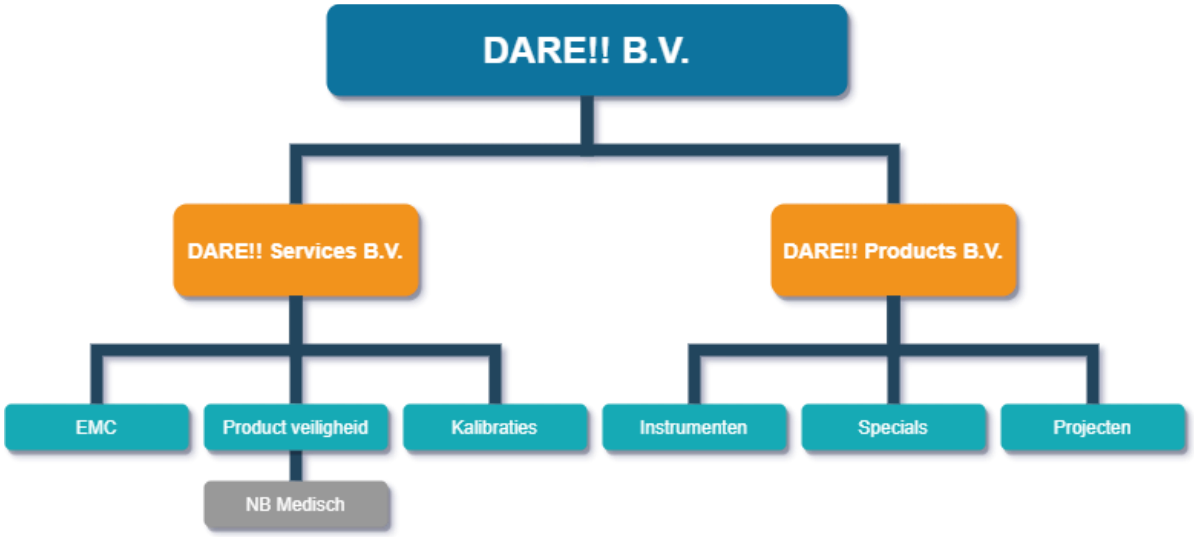


Figure 1 Company structure of DARE!! B.V.

DARE!! Products is an internationally orientated company with a focus on creating EMC testing applications. Examples of such devices are Electric field probes, Electric field Generators and Power Meters. Other than making products, DARE!! Products is also involved in special projects related to EMC and RF.

An important piece of software developed by DARE!! is RadiMation®, the software that controls all products made by DARE!! and can perform measurements with them.

Figure 2 logo of DARE!!

5 | The assignment

5.1 RadiMation®

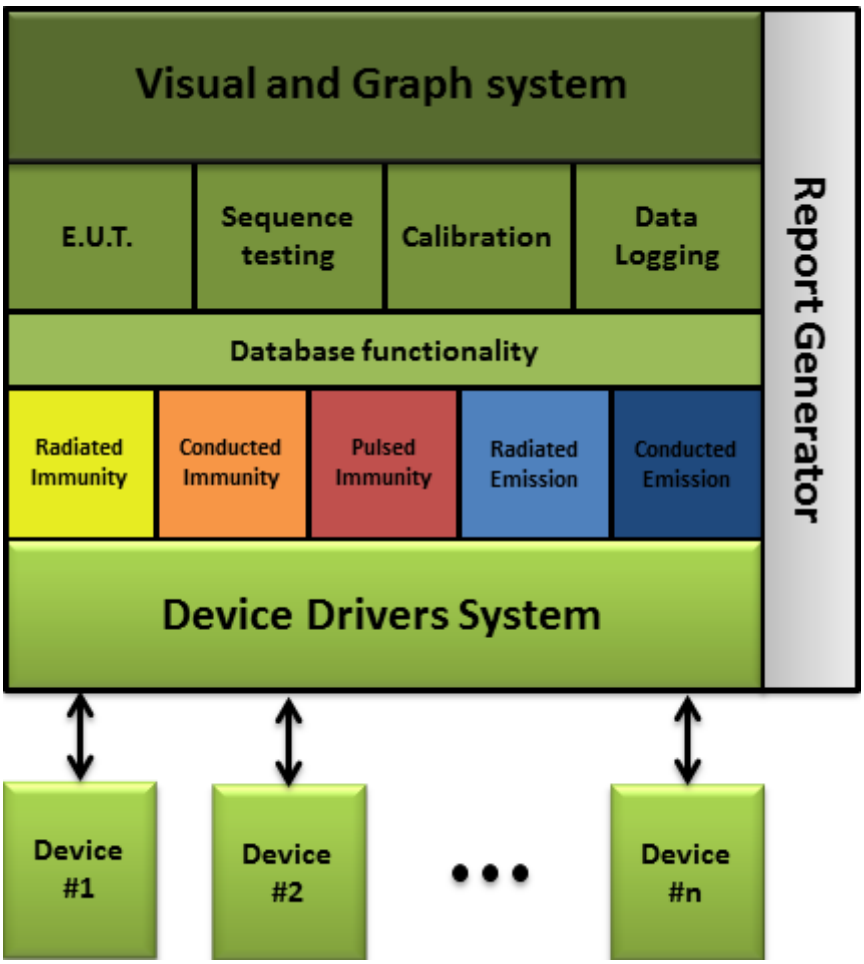


Figure 3 Modular overview of RadiMation®, from the wiki

RadiMation® is a software product that runs on a computer and is able to perform a completely automated EMC measurement controlling all the required test and measurement equipment. This includes signal generator, high power RF amplifiers, power meters, field strength sensors, turntables, antenna towers, spectrum analysers and network analysers and other supporting equipment. Those instruments are controlled by several protocols like RS-232, USB, GPIB, Ethernet and VISA.

Measurement equipment can normally be controlled by a combination of one or more of these protocols. To control these test and measurement equipment's, RadiMation® is using device drivers that control the hardware products by setting and changing the correct settings to achieve the desired operation. RadiMation® has more than 4500 different of those device drivers that are created using object-oriented C++ code.

These device drivers are all bundled .dll files. This means that the main program loads those .dll files to acquire data on how to display a device specific GUI or how to control that specific device. For example, a device by DARE!! will show you various communication methods such as USB, GPIB or RS-232 and more. A device from the Marconi series (specifically the Marconi 2022A) can only connect through GPIB, so it will only show that option and even its own unique window when trying to configure that device.

Figure 4 Splash screen of RadiMation®

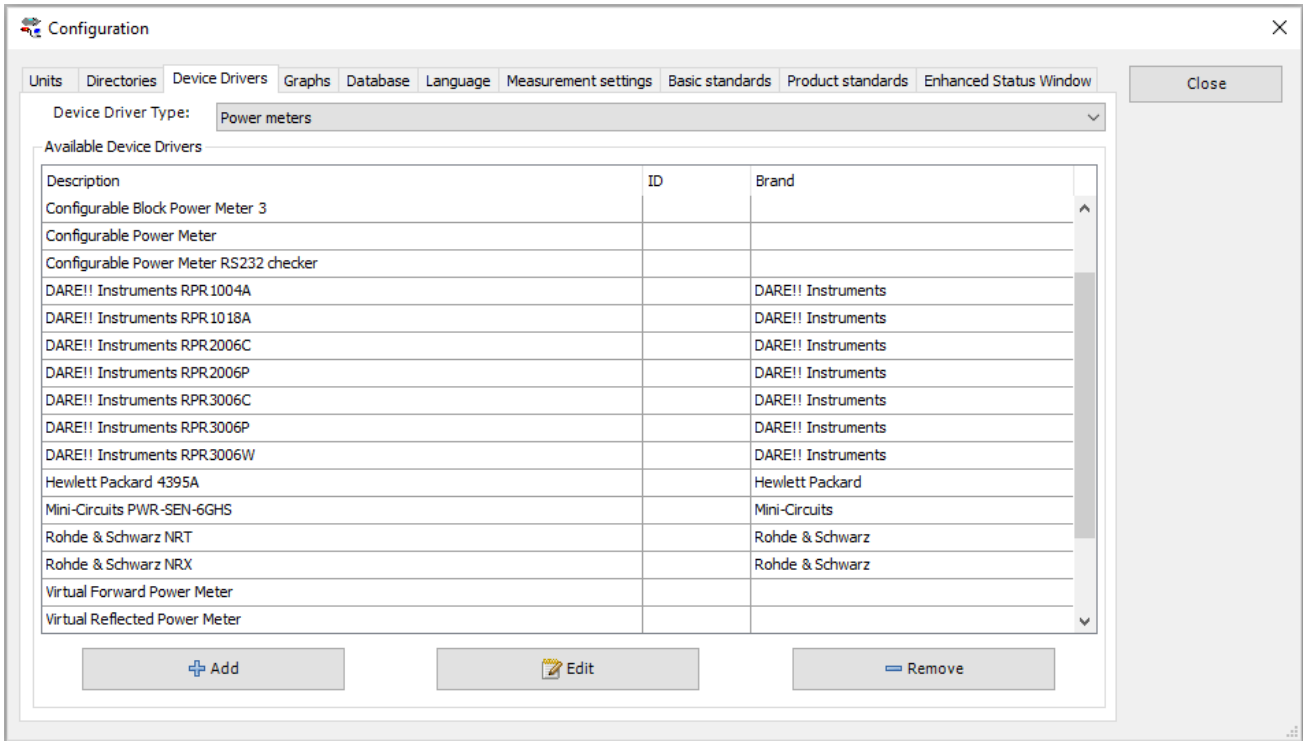


Figure 5 Selection of power meters when configuring the device drivers

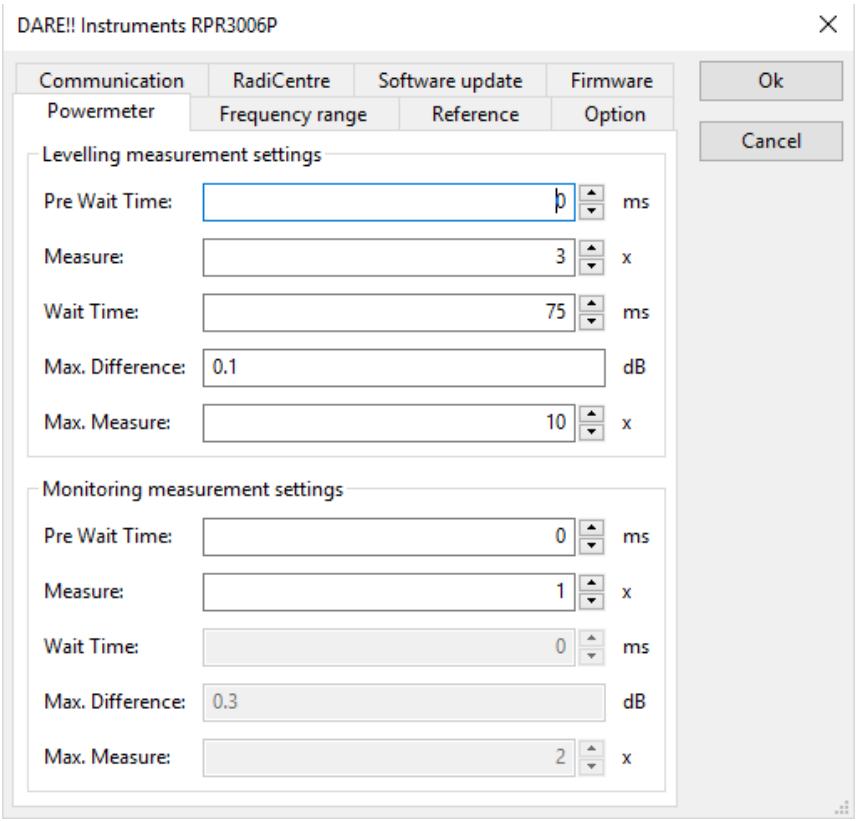


Figure 6 Configuration window of the RPR3006P, a Powermeter developed by DARE!!

5.2 The details of the graduation assignment

The graduation assignment is to simplify the configuration of the communication settings of these device drivers for the end-user. This can be separated in three different parts:

1. Reduce the duplicated selection of communication protocols: At this moment the VISA connection settings allow to select the GPIB protocol, while also a specific GPIB connection setting is present:

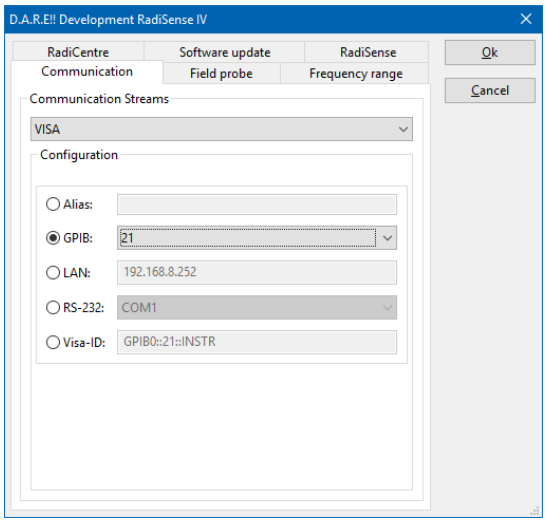


Figure 7 Configuring GPIB through VISA

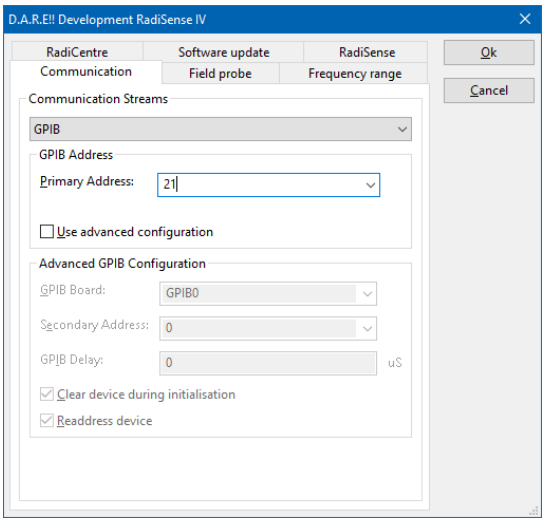


Figure 8 Configuring GPIB normally

Also, for RS-232 communication, two possible configurations are available:

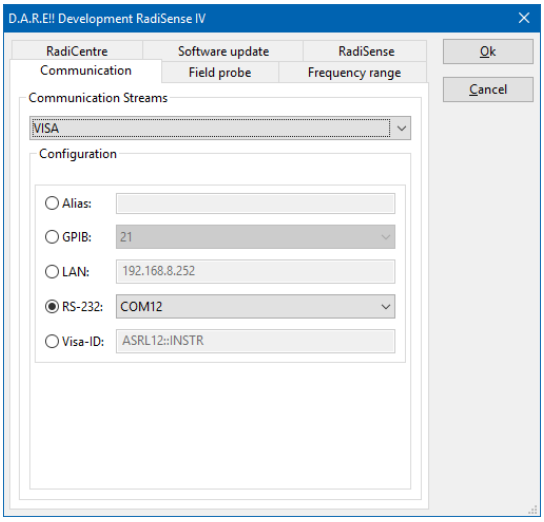


Figure 9 Configuring RS-232 through VISA

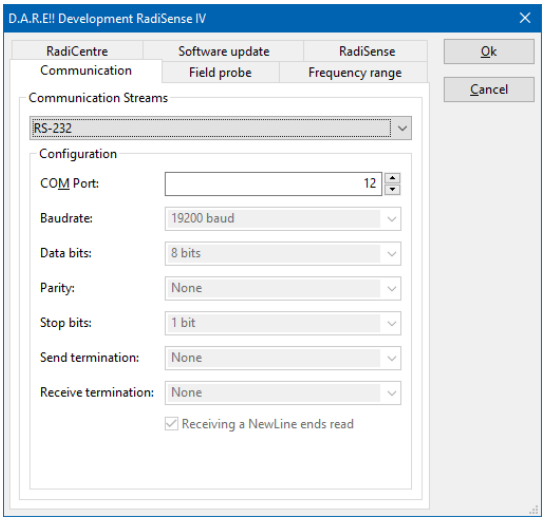


Figure 10 Configuring RS-232 normally

And also, for TCP/IP connections multiple configuration windows are possible:

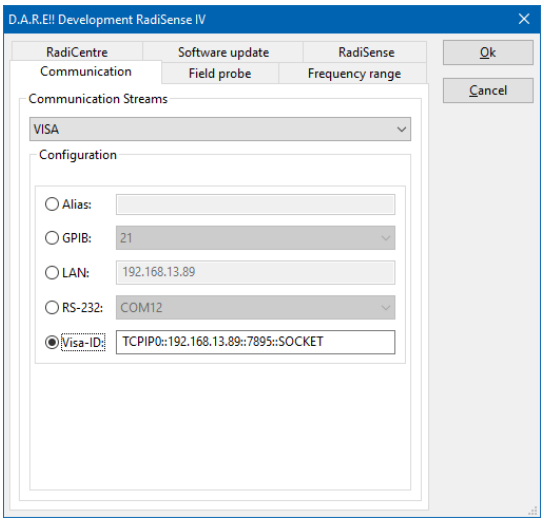


Figure 11 Configuring TCPIP through VISA

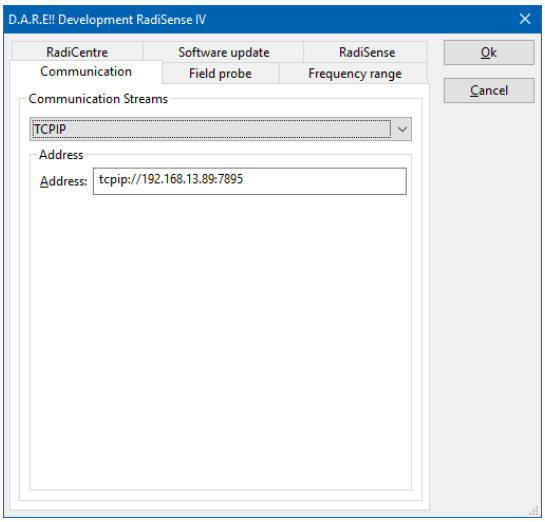


Figure 12 Configuring TCPIP normally

These duplicated windows and settings are internally using different communication settings and software libraries, but it is very confusing for the end-user. The improvement should simplify the configuration where the device drivers internally automatically determine the best software library to communicate with the instrument.

2. Simplify the auto detection of DARE!! Instrument products when multiple of those products are connected to the PC at the same time. At this moment the auto detection of an USB connected DARE!! Instrument product requires that only a single product is connected. If multiple products are connected the auto detection is not able to detect the correct product. This should be improved in such a way that the end-user for example can select the correct instrument from a list of one of the detected/connected instruments.

3. Auto detection of the available test and measurement equipment that is connected to the PC being able to automatically determine which measurement equipment is connected to a PC, will provide a much easier installation to our end-users. This can be achieved by scanning the USB, GPIB and Ethernet bus to detect if one or more of our products is connected.

The assignment is thus related to providing a simpler interface for the end-users regarding the available communication protocols. To achieve this, improved algorithms should be implemented that interact with the available communication buses to perform auto detection of the available equipment.

The implementation of the improvements should also comply to the DARE!! internal structured development process, which requires that first a functional description and an implementation design is created. Once the actual implementation is finished, also a final test report which includes functional tests, should be delivered.

5.3 FNS

FNS stands for “Functionale Specificatie” (FuNctional Specification). It is a document that DARE!! Products uses to describe future products, which can be software or hardware, or parts of software or hardware. Inside an FNS one can find the following:

- Changelog
- Introduction
- Product name, goal, description, finances
- Power requirements, measurements, communication interfaces

Many of these are not relevant when developing software. In such a case, an FNS will entail more about the requirements of the software, the preconditions, an analysis and a design based on that analysis. The person who makes an FNS does not necessarily be the one who implements the feature. However, this was the case with this assignment.

There are three separately written FNS’s, one for each part of this assignment. All can be found in the appendix.

According to DARE!!, an FNS is all the documentation they need to continue working with the software that has been written. Because the FNS specifies all the visible and behavioural changes that are notable for the end-user.

5.4 RadiCentre

“EMC test systems can vary from a simple EMC test system with one or two instruments, to complex installations with many EMC measurement instruments connected, while in many cases even a turntable and an antenna mast should be controlled. In order to enable full automated EMC testing, these devices and measuring instruments, should be controlled in an automated manner. Where the EMC test software RadiMation® acts as the software centre of the EMC system; the EMC test system RadiCentre is the core of the hardware. With the introduction of the RadiCentre by DARE!! Instruments, cost effective, full automated testing finally becomes reality!” [2]

At a very basic level, a RadiCentre is practically a large USB hub which can run its own programs, for specific hardware. It can hold up to seven interface cards, depending on the model. These cards are specifically made for one purpose. For example, a RadiPower is a card that has 4 USB slots, in which power meters from DARE!! can be connected to. There is a total of eight different cards currently available, but only the RadiPower card was used for testing the code. When using the auto-detect functionality in the USB section, the code has to communicate with the RadiCentre to tell it what devices are connected to it.

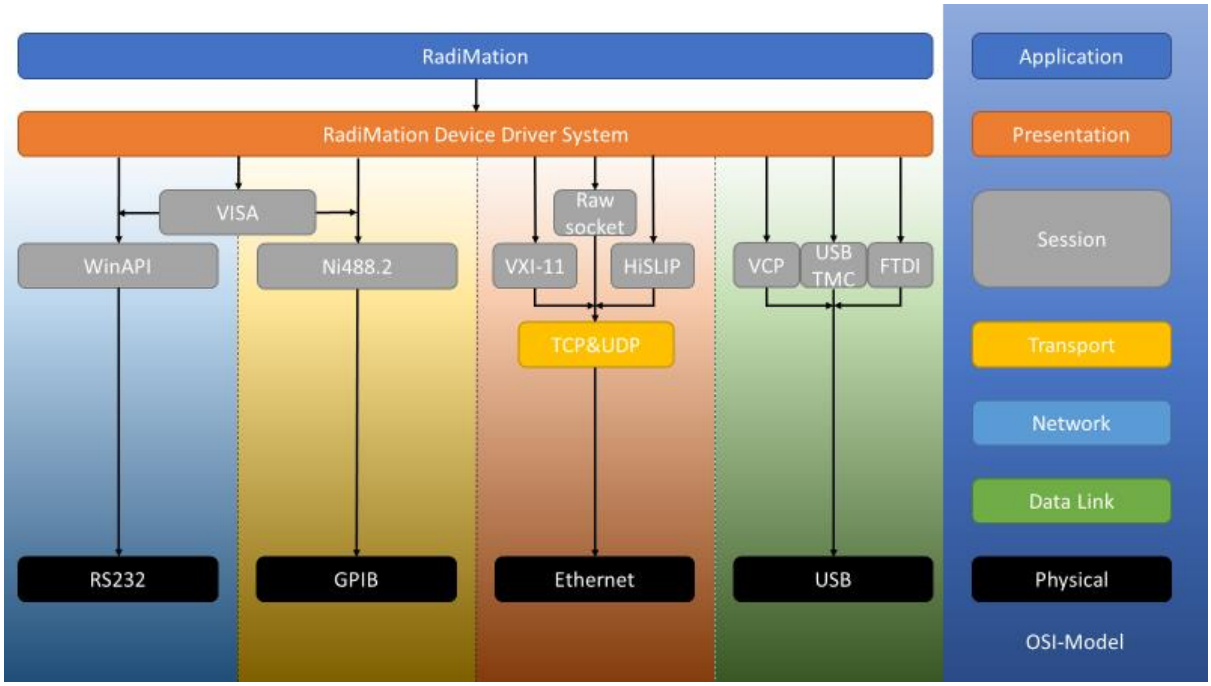


Figure 13 OSI model of RadiMation®

5.5 wxWidgets

wxWidgets is a library for C++ that allows programs to make a GUI relatively easily. wxWidgets is used throughout RadiMation® using .xrc files. These are files that hold data of a window. Every window in RadiMation® has its own class, which holds the references to buttons or other settings. This way programming behaviour is made easy. Code can be executed when certain buttons are pressed, data can be retrieved from fields where the user can type in and the window itself can be set to behave in a certain manner, such as customisations of the borders around textboxes. To edit these .xrc files, a program called “xrced” is used, a free program that allows for the creation and editing of .xrc files.

Using xrced It is possible to add the buttons to the GUI of the RS-232 communication settings window and that of the GPIB communication settings window. In the code another window was taken as an example to figure out how RadiMation® interacts with the buttons and textboxes.

Every button or textbox that is shown to the user has an ID tied to it in the .xrc file. Using this ID RadiMation® can locate the component and create a pointer to that component. This way code can be executed on a button press, or data can be extracted from a textbox.

5.6 Programming terms

In the following solutions there will be explanations about a lot of code and there will be mention of certain concepts that exist within C++ programming or are relevant to RadiMation®. Here is a list of terms that have been used with explanations of them:

String	A set of characters that can resemble a word, or sentence. “Hello” Is an example of a string.
Array	A ‘container’ of a data type with a set size. An array of integers can have several numbers within it, but the size must be specified when it is initialised.
Vector	An array of a variable size. A vector is initialised with a data type, such as an integer or a string, or even another vector. Now, data can be added and removed at will.
Class	A ‘container’ of multiple data types, however specified. Can contain anything and can have data protected (only accessible by itself unless shared), public (accessible by anyone), and private (data only accessible by itself). Classes can also inherit these data types from other classes, so it is not needed to re-declare these data types.
Pointer	The address of a location containing data. Sometimes it’s not possible or not efficient to carry (or copy into another variable) all the data at that location to somewhere else. The address is used instead, so the data at that location of memory can be accessed.
Object	An instance of a class. A class only defines the data types. An object actually holds the data.
*IDN?	A command that is send to devices to ask them to return their name. This does not always work.
ID_NUMBER?	A command specific to DARE!! devices, it returns their identification number.
VISA	A library for easier communication over USB, GPIB, RS-232 or HTTP.
COM port	A communication port, such as USB or RS-232. Does not have to be physical as virtual comports exist, such as Bluetooth.
Baud rate	Definition of the speed of a data transfer. Some devices have a set speed, any baud rate lower or higher than the set baud rate will fail communication, as it is not what the device expects
Map	Takes a pair of any type of data and any type of data. A map makes sure that you cannot add a pairing with the same entry twice. For example, one can assign the sentence “Hello” to an integer value of 1 and save it to the map but cannot then assign a pairing of 1 and the sentence “World” to the same map.

6 | Solution I, USB auto-detection of multiple devices

6.1 The general approach

As an example, let's take a power meter from DARE!!: The RPR3006P, the configuration window can be seen in figure 6.

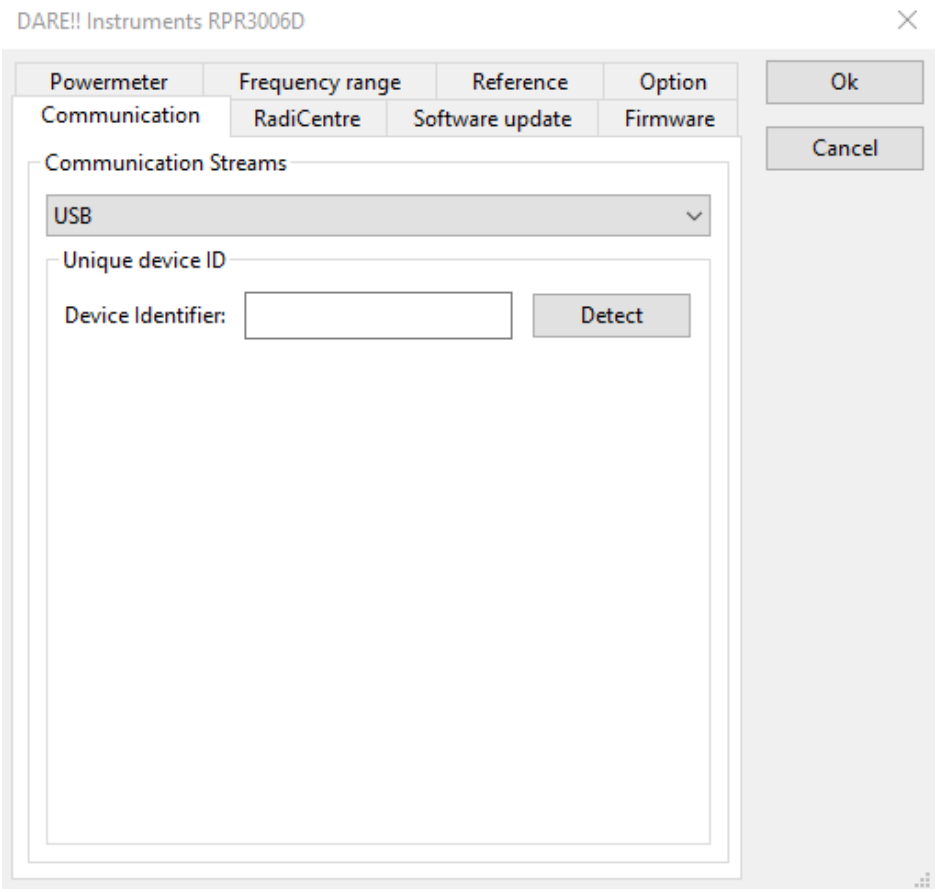


Figure 14 When configuring USB communication, showing the Detect button

The "Detect" Button starts the code that has been expanded.

In short, the code works like this:

1. Acquire the name of the currently loaded device driver
2. Fetch a list of currently connected devices (even through a RadiCentre)
3. Compare the name of the loaded driver with the names of all devices
4. Determine result on the number of matches: on 0, error message. On 1, automatically fill in the address. On 2, show a menu where the user has to select the correct one.

The assignment specifically says that the auto-detection should work for DARE!! devices through USB. This means that the names of the devices that have to be fetched (for comparison purposes) are conveniently in one location. This location, however, is not easily accessible.

6.2 Acquiring the name of the currently loaded driver

In figure 15 is a map of classes that either inherit or contain a reference to each other. This is to help visualise the process from where the names come from and where they need to go to.

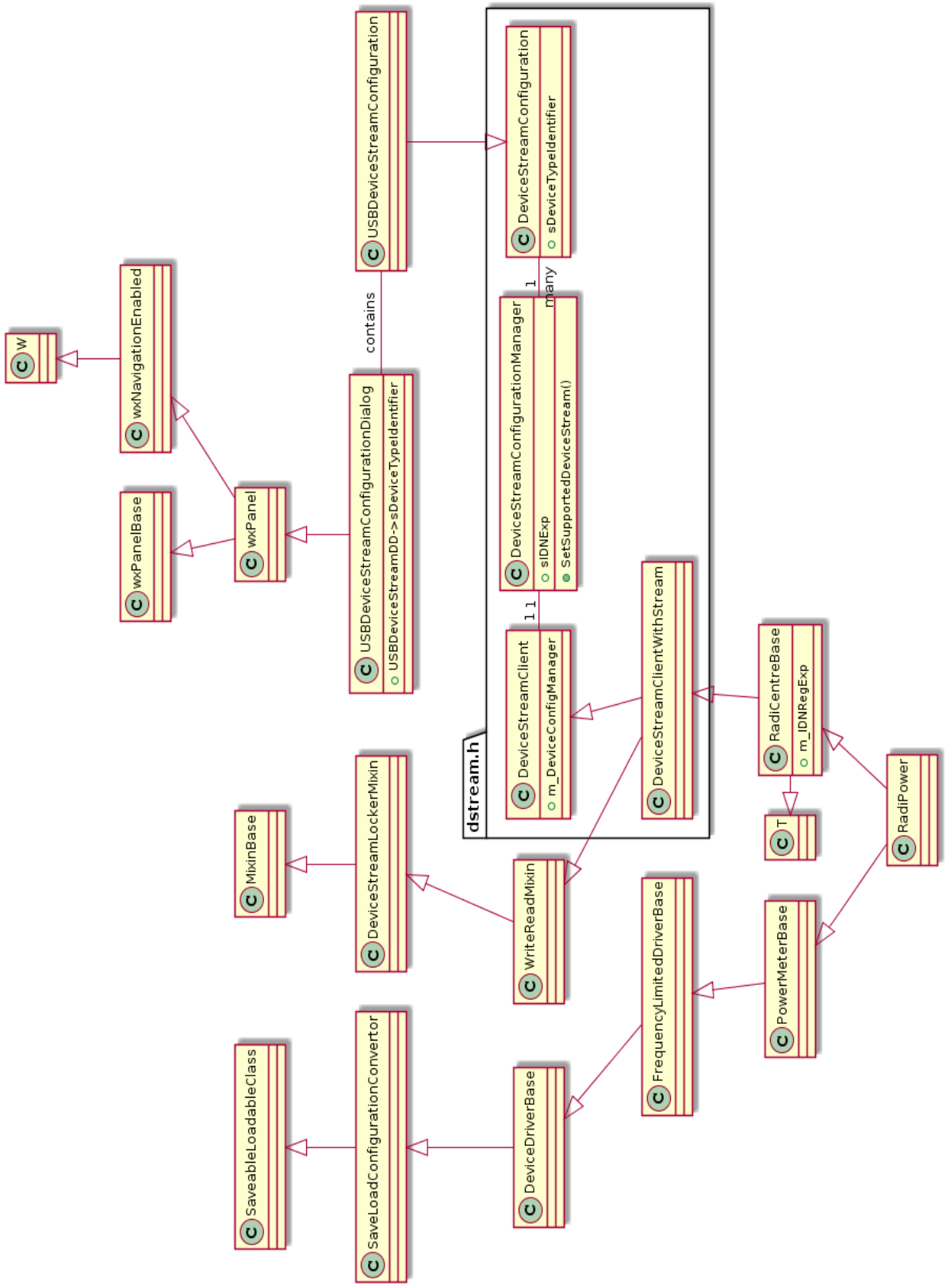


Figure 15 Inheritance tree of a section of RadiMation®, showing a potential path from the RadiPower class to the USBDeviceStreamConfigurationDialog class

For the following explanation, try to follow along with figure 15. The names of the devices are listed in RadiPower, at the bottom. RadiPower inherits all functions up to DeviceStreamClient, which has a subclass called DeviceStreamConfigurationManager. USBDeviceStreamConfigurationDialog is where the name needs to go to. This class contains a pointer to USBDeviceStreamConfiguration, which is inherited from DeviceStreamConfiguration. The problem is that there is only one device stream config manager, and there are many device stream configurations. This means it is impossible to fetch the name from the configuration to the manager, but the manager can give the name to each configuration.

A string variable has been put inside DeviceStreamConfigurationManager called sIDNExp and a string variable in DeviceStreamConfiguration called sDeviceTypeIdentifier. Considering they are both in dstream.h, there is a way to have the data in sIDNExp be copied over to sDeviceTypeIdentifier. m_DeviceConfigManager is an object made from the class DeviceStreamClient. In the constructer of RadiCentreBase (Basically when an object of this class is created) the m_IDNRegExp variable is passed to sIDNExp inside m_DeviceConfigManager. Then, when a DeviceStreamConfiguration is created the data is passed in sIDNExp to sDeviceTypeIdentifier, and now the code is able to access the name in USBDeviceStreamConfigurationDialog for comparison purposes.

6.3 Fetch a list of currently connected devices

This one is a bit easier, as this was (mostly) already present. Before the new additions, there already was an auto-detect button. However, it only worked when a singular device was connected, else it would throw an error message. Neither did it check to see if it was the correct device, it just wanted a single device. During this detection, a command was send, which is “ID_NUMBER?\r\n”. the ID_NUMBER returns an address, or better known as a string of 8 different numbers, from 0 to 255. (for example, 0.6.213.42.32.5.74.87). Every address comes from a chip from another company that guaranteed that every address is unique.

There is no worry of devices not responding to this command, as every DARE!! device is programmed to respond to this. This is not the case when it comes to GPIB or RS-232 communication, which will be discussed later.

Of course, only the address is not enough. The name is required as well, which is retrieved using the “*IDN?\r\n” Command. What is happening here is first the *IDN? command is send to ask for the name, wait a moment and then read what is send back. This is repeated for the ID_NUMBER? command.

However, one of these devices might return the name “RadiCentre”. The largest RadiCentre configuration currently has 8 card slots, and each card has 4 USB ports. These numbers may also increase in the future. In the event that a RadiCentre is found, another scan has to be made.

Communication with the RadiCentre is slightly different. the RadiCentre can be asked how many slots it has, but it can’t tell which slot has a card. A scan will have to be made on each port on each card, which is done like this: “1:A*IDN?” with the 1 being the card number and A being the first slot. This way the names of every device connected through a RadiCentre can be stored in a vector of strings, but not the identification. That would just return the ID of the RadiCentre. This is OK, as the user is then told to then configure the RadiCentre tab, while the ID of the RadiCentre is filled in. Scanning through each port works by using a for loop with characters as the counter, ranging from 1 to 8, and from A to (A + the number of slots in a card). This way a command can be constructed to send to the

RadiCentre. In the event that the first scan of a card returns a specific error, it will skip the rest of the slots in that card to save time, as it can be assumed that the card simply does not exist. This also means that the vector of strings that stores the names is filled with four (or how many slots the card that is being scanned has) blank spots. Should a scan succeed and return a device, its name is then stored inside the vector of strings in the object that is used to store all data that will be collected from the communication. This data consists of the name of the device, potentially names of devices is it is a RadiCentre and the ID number.

The data received is now checked if it is empty or not. In the event that it isn't, the identification number and the *IDN? response are stored in the object.

6.4 Compare the name of the loaded driver with the names of all devices

There is now a vector loaded with all connected devices via USB, which contains data on the name, the identifier and devices connected to it if it is a RadiCentre. In the RadiPower file all names that are needed are listed, but they are not listed the same. Certain device drivers have a name such as RPR1014A, which is good, others have a name such as (RPR3006P|7002-006), which has to be cut down back into RPR3006P. This is done with grabbing substrings, from the first letter to one before the '|'.

Now the size of the vector is checked. At 0 entries, the code can conclude that no device is connected at all.

At 1 entry, there is only 1 device connected. The name of the loaded driver is now compared with the name of the device. If this is a match, the address is filled in and the program is done. If not, a check must be made to see if this is a RadiCentre.

In the event that it is a RadiCentre, all the names logged when the communication with the RadiCentre took place need to be checked. Should one match, the address is filled in of the RadiCentre and RadiMation® will tell the user to configure the RadiCentre tab. Should more be found behind the RadiCentre a menu shows up which will list all the devices. This menu is discussed later. In the event that a RadiCentre is found but the device is also not connected behind a RadiCentre, an error message will appear mentioning that it could find a RadiCentre but not the device.

It is also possible that there is a device connected, it does not match the name and is not a RadiCentre. In this case a message will be shown asking the user if they want to fill in the address anyway, even though it's the wrong device.

At 2 or more entries, the process is slightly more complicated. A separate counter is needed to keep track of name matches in the extra number of devices. The name of every device is checked, and the counter is increased, should a match occur. If one of these devices is a RadiCentre, a check must be made for every device behind the RadiCentre and increase the counter if another match is made. Based on this count, a decision is made similar to that if only one device was found in the vector. No matches, error message. One match check if it is a RadiCentre otherwise fill in the address directly. If there are still duplicate devices, a GUI will be shown listing all devices.

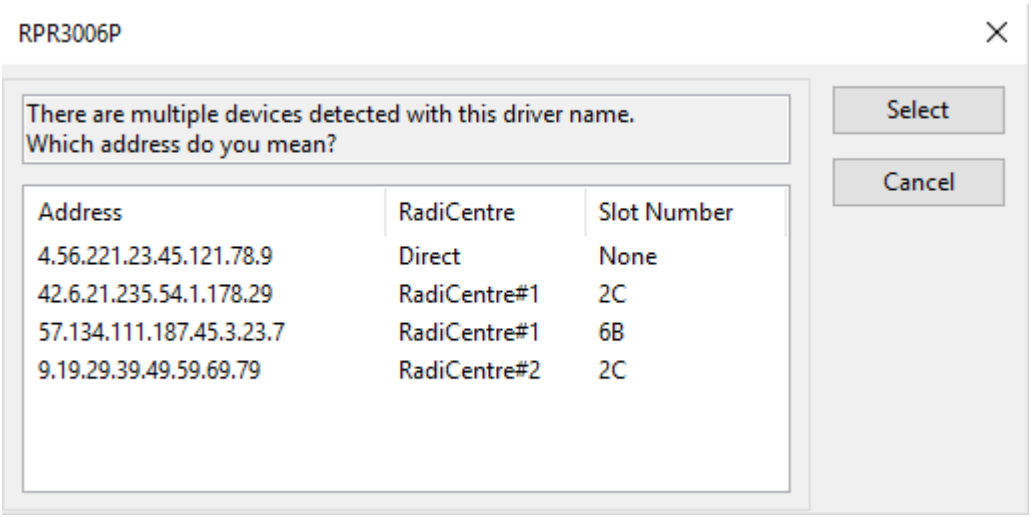


Figure 16 GUI that shows when multiple amounts of the same device is connected to a RadiCentre or the computer

In the above image the GUI is visible that shows all device with a duplicate name. A device is either directly connected to a PC, or through a RadiCentre. There can be more than one RadiCentre too. Only one address can be selected at the same time, and it is impossible to press the "Select" button without having clicked on an address, else the program will show a warning message. With an address selected, it is then filled into the Device Identifier field. With that the program is finished.

To fill in this list, first the columns have to be created. Then, the data must be filled in per row. Another name comparison is made here, just like before. If this is a match, the address of the device is added to the list here, with the word "Direct" since it is not a RadiCentre. Should a scan be a RadiCentre, all devices behind it will be scanned as well. Then, based on its position in the vector, a slot number is calculated and shown on the list, alongside what RadiCentre it is connected to.

7 | Solution II, removal of duplicate options

7.1 The List

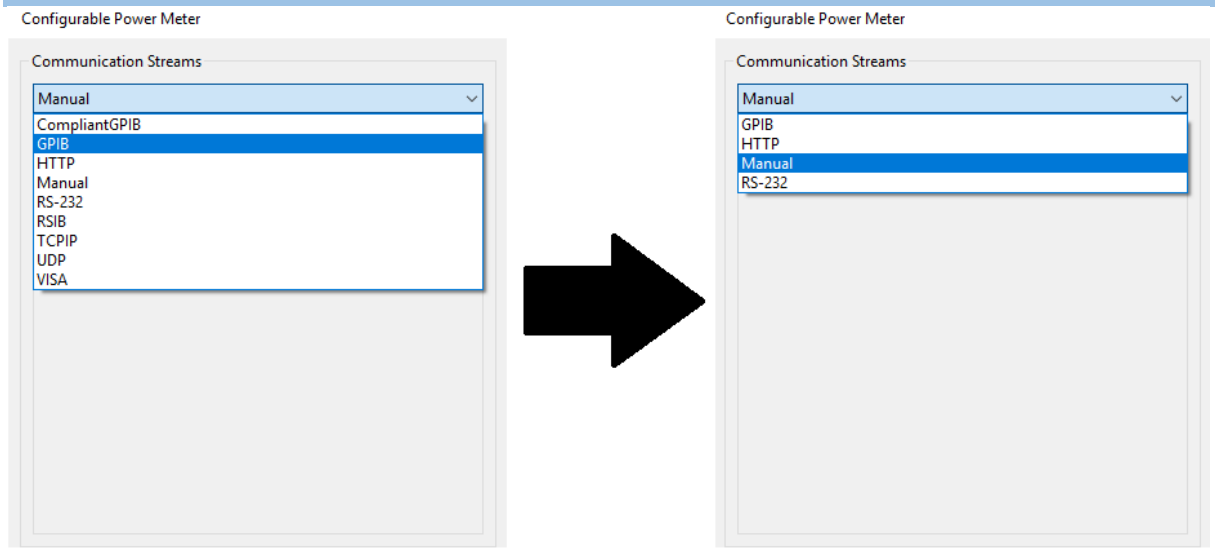


Figure 17 How the list was, to how the list should look like. This list specifically is missing the USB communication method

RadiMation® allows communication over a total of twelve different protocols. However, there are a lot of similarities. HTTP, TCPIP (Ethernet), RSIB and UDP all take a single address as argument to communicate with. GPIB and CompliantGPIB only have a difference internally in RadiMation® and is not important to the end user. These two take the exact same settings. USB also has a similar version, namely USB_D2xx. There is also RS-232, manual, and USBHID. RS-232 has no similar counterpart in RadiMation®, Manual has no configurable settings and USBHID is very rarely seen in RadiMation® so it is set aside and not included with the other USB communication protocols. USBHID also takes two settings (PID and VID) instead of a singular address that USB and USB_D2xx uses.

As seen in figure 9, VISA also has similar options. However, VISA is not always installed on the PC alongside RadiMation®. Therefore, a separate check has to be made in the code to see if VISA is installed. If it is, the options through VISA take priority, and will be attempted first.

7.2 The general approach

In general, the following needs to happen:

1. The list must be shortened to HTTP, GPIB, RS-232, Manual, USB, if these or similar communication protocols are present.
2. Once a communication protocol is selected, the end user can press a button to check if the device is connected. This button already exists and is not part of the assignment.
3. RadiMation® will automatically fill in the other communication protocols (of those available) with the same data. As an example, let's take HTTP as the selected communication protocol.
4. How this should work is a priority system. RadiMation® will take a communication protocol, attempt to connect with it, and fail or succeed. In the event of success this protocol is then chosen to continue with. In case of failure the next protocol down the list is used.
5. Due to HTTP being chosen, the list is as follows of high to low priority: VISA->HTTP->TCPIP->UDP->RSIB.
6. The address filled in by the end user when the driver was being configured is saved and applied to all other communication protocols.

7. VISA is checked if it is installed. If so, use that. The LAN sub-section of VISA is automatically selected and filled in. Communication using this protocol is attempted with this address. This will almost always work. If not, or VISA is not installed, immediately proceed to the HTTP protocol.
8. RadiMation® will now attempt to work with HTTP. If it succeeds if keeps this protocol and if it fails it picks the next one in the list, being TCPIP. This continues on until RSIB is attempted.
9. Should it fail at RSIB the end user is told that communication has failed.

7.3 Shortening the list

Most of the code to be edited already exists in a class that has been addressed before, which is the DeviceStreamConfigurationManager in dstream.h. This class holds the data of the configurations of all devices and so it is possible to manipulate the data here to be carried over when a different communication protocol is about to be used. One of the functions called when populating the list of device drivers is SetSupportedDeviceStream(). This function reads which communication protocols are supported by the driver and fills a map. This map takes a string (the name of the communication protocol) and a pointer to the configuration of this particular device. This map is then used later to receive the correct configuration based on the communication protocol that the end user selected.

This code is now edited to show a smaller list. Normally a vector would be filled with the names of all communication protocols and that would be used. Now, a second vector is created. In the event that either GPIB protocol exists, only the normal GPIB protocol is added. When any of the internet protocols exist (HTTP, TCPIP, RSIB, UDP), only HTTP will be added and so forth. VISA is no longer necessary to be shown here, therefore it is excluded. This second vector is now used to populate the list instead. The first vector is still saved for later use.

7.4 Selecting the correct communication method

Before an attempt to communicate with the device is made, RadiMation® will have to check which communication protocol has been selected. This is a function called GetSelectedDeviceStream().

There are three types of communication protocols that need to be cycled through. These are GPIB, HTTP and USB. In the previous paragraph it was explained how a second vector was populated and used as the list that is shown to the end user. Now, after the end user no longer sees this list, it is now populated with all supported options. This is necessary because this way the other device stream configurations are created, which will have to be slightly manipulated later. The several options for GPIB, HTTP and USB are stored inside an array. This step is skipped if the connection is RS-232, USBHID or Manual. Depending on which option is selected, an array is passed through a function that will cycle through all communication protocols inside the array.

First, the function (called OptionCycler()) checks for each entry in the array if it is a supported communication protocol. There is no need to try TCPIP communication, for example, if it is not supported. If it is an option, it is now selected as the current communication protocol. Using this protocol, a small attempt to connect to the device is made. Should it succeed, the function is done and will return. If not, the next protocol is selected, and the program is now back at the beginning of this function. Before it does this, the data saved is also carried over to the next valid communication protocol. This is explained later. If all communication protocols fail the selection will revert to its first viable selection. Now the function GetSelectedDeviceStream() has the correct device stream selected, for which it will load its slightly modified configuration and finish normally.

7.5 Carrying over the data

Before a communication attempt is made, the data that RadiMation® will use has to be intercepted with that which was filled in by the end user. The end user can no longer configure the TCPIP protocol so RadiMation® must make sure that the address filled in at the HTTP section is also applied in the TCPIP protocol. Every communication protocol has a connect function, but some protocols share the same connect function. Inside every connect function, another function related to the device stream configuration is called: the GetUniqueAddressString() function. This function simply returns what was filled in by the end user when he or she configurated the device driver and is present in every connect function. A string has been created as a property in the device stream configuration object.

Once the GetUniqueAddressString() function is called, the same as before is returned but along the way its output is also saved to the string. All the HTTP communication protocols take a single address such as http://192.168.1.0:80 and the USB take a device identification of 8 numbers, as talked about in Solution 1. GPIB is a bit different. With GPIB, inside the string the board number is stored, alongside its primary and secondary address as well as its delay, whether the device should be cleared during initialisation and whether or not the device should be readdressed. These six variables are stored inside a single string, separated by commas. Converting this to a string is easy. Getting this data out of the string is a different story, as each piece of data has their own datatype, such as bool, int, DWORD and more. The string has to be cut up in between the comma's and then applied to their respective variables separately, before a connection is made. Now RadiMation® will apply this data and attempt a communication.

8 | Solution III, automatically finding addresses connected via GPIB or RS-232

8.1 GPIB

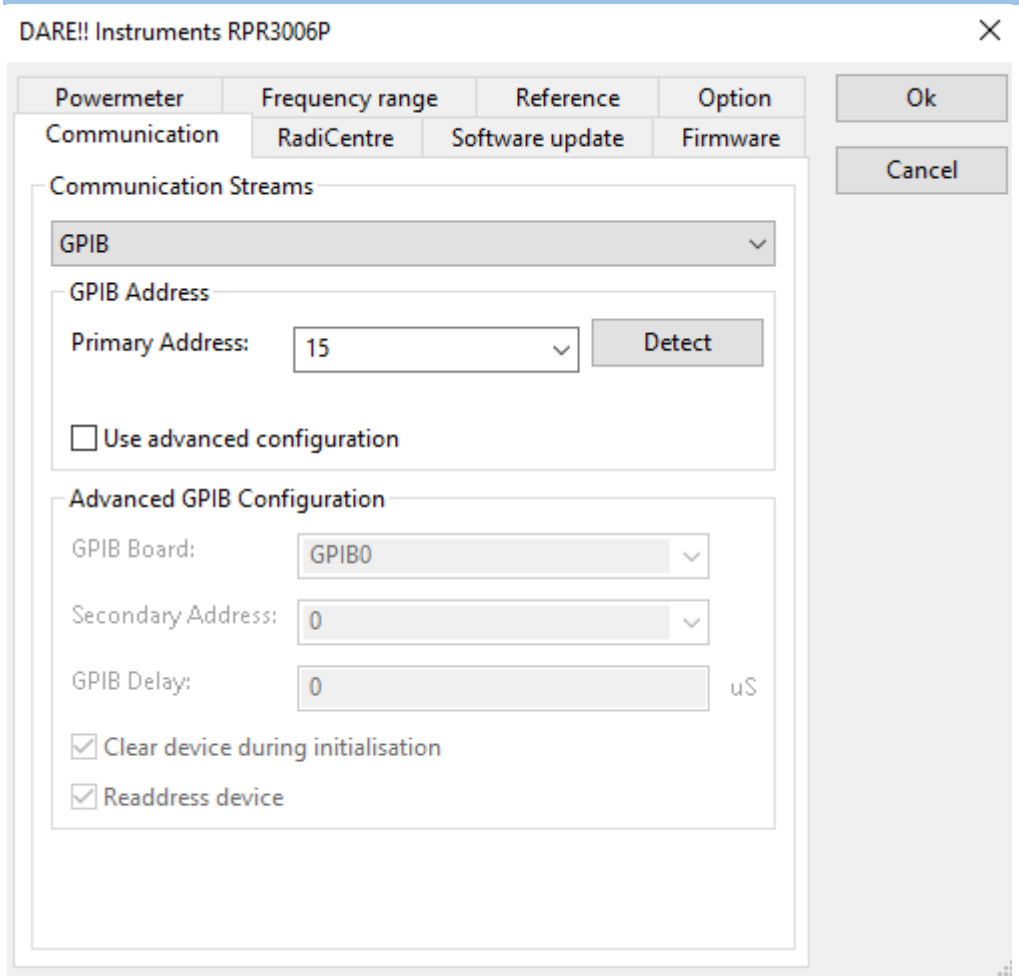


Figure 18 Configuration window of GPIB

8.1.1 What is GPIB

GPIB stands for General Purpose Interface Bus, though it can also be called the IEEE 488 bus. Every device connected has its own unique address. This is a value from 0 to 31, which includes the GPIB controller Board as well.

“In the original HPIB protocol, transfers utilise three wire handshaking system. Using this the maximum data rate achievable is around 1 Mbyte per second, but this is always governed by the speed of the slowest device. A later enhancement often referred to as HS-488 relaxes the handshaking conditions and enables data rates up to about 8 Mbytes / second.

The connector used for the IEEE 488 bus is standardised as a 24-way Amphenol 57 series type. This provides an ideal physical interface for the standard. The IEEE 488 or GPIB connector is very similar in format to those that were used for parallel printer ports on PCs although the type used for the GPIB has the advantage it has been changed so that several connectors can be piggy-backed. This helps the physical setting up of the bus and prevents complications with special connection boxes or star points.“ [5]

In GPIB, there are three methods for how devices can act: Controllers, talkers and listeners. Controllers make sure there are no conflicts on the bus, such as two talkers talking. This would corrupt the data. Talkers put information on the bus and listeners receive that information. Many devices are able to switch from being a listener to a talker and vice versa.

Figure 19 Physical GPIB connector, image taken from https://www.electronics-notes.com/images/GPIB-connector-01.jpg

8.1.2 The general approach

The plan here is similar to the solution of the USB auto-detection. Find all devices (in the case of GPIB I use a special function, FindListeners()) and ask each device if they are the device we are looking for. This is done through another function inside RadiMation® called CheckDevice(), which will be explained later. This function returns either an error or not. Once done, the number of devices that did not return an error is counted and an appropriate result will be displayed.

8.1.3 GPIB Boards

There can be many GPIB boards connected to a computer. However, RadiMation® only allows up to a maximum of four boards. However, usually only one board is available, but then we still don't know if that's board #0, 1, 2 or 3. Therefore, a function is used to communicate with the GPIB boards one by one to see if there is a response, and if there is, they are added to a vector. This vector is later used when scanning the individual boards and making sure that there are no scans on boards that don't exist.

At this point, a "Please Wait" message will appear, since this process may take up some time.

8.1.4 The search with GPIB

The data that needs to store is the GPIB board number, and any devices connected to that specific GPIB board. The thing is, there can be multiple GPIB boards. To save this data efficiently, a two-dimensional vector has been applied. This vector stores a GPIB Board as its first number, and then any GPIB address on that board as its second. For example, a vector called GPIBData could have data in position GPIBData[0][3], which would be the on board #0 and on that board, the fourth address.

> *NI-488.2 Library*
>
> To find all the connected devices with GPIB, a function of the NI 488.2 library is used. The National Instruments 488.2 library holds functions related to GPIB communication.
>
> "NI-488.2 provides support for customers using NI GPIB controllers and NI embedded controllers with GPIB ports.
>
> NI-488.2 is an NI instrument driver with several utilities that help in developing and debugging an application program. NI-488.2 includes high-level commands that automatically handle all bus management, so you do not have to learn the programming details of the GPIB hardware product or the IEEE 488.2 protocol. Low-level commands are also available for maximum flexibility and performance. "[3]

From this library a function called FindListeners() is utilised. This function requires an array as input and a different array as output. It will proceed to fill the output with the addresses of the devices, based on the addresses of the input. So, if the list is filled with data ranging from 0 to 31, it should scan each and every GPIB address. A little extra is required to make this function functional in the code.
an empty input array is created with a size of 32, which will then be filled with the values of 0 to 31. FindListeners() requires that the last entry in the input array has a value of NOADDR, so It is declared that in position 31 of that list is equal to NOADDR. Then, an output array is created of data type "short". This is because FindListeners() requires it. Every entry is filled in this array with a value of -1, which is later explained. Finally, another function is used to determine the address of the GPIB board itself. This way it is eventually excluded in the resulting vector. Now the function FindListeners() is called to fill the output array with addresses. Any address that does not respond will remain at a value of -1. Now with the use of a for loop the address of the GPIB board is filtered out and those with a value of -1 are filtered out, so that the resulting vector will be filled with correct addresses.

8.1.5 The CheckDevice() function

Each driver in RadiMation® includes a function, CheckDevice() that checks if that specific device is connected. Whenever CheckDevice() is called, and depending on which device driver is currently loaded, that function of checking if the device is connected, will be performed. It will return an error in the case that it cannot find the device, and no error if it can. Using this function, the code does not need to know the name of the device for comparison, it already knows if it is there or not.

Now for every entry inside the vector that holds the addresses, CheckDevice() is called. Every address that succeeds is copied over to another vector to hold the address data for the successfully connected device, and a counter is increased for counting the number of matches.

At the point the "Please Wait" window can close, as the part that takes up most of the time is now finished.

8.1.6 Determining an action

Now, three things can be concluded:

1. There are no matches. A message will be displayed notifying the user that the auto-detection was a failure.
2. At one match the correct device is found and the GPIB board and address is filled in.
3. At more than one match, similar to solution 1, a new GUI window is shown that contains all duplicate addresses, allowing the user to select one of them.

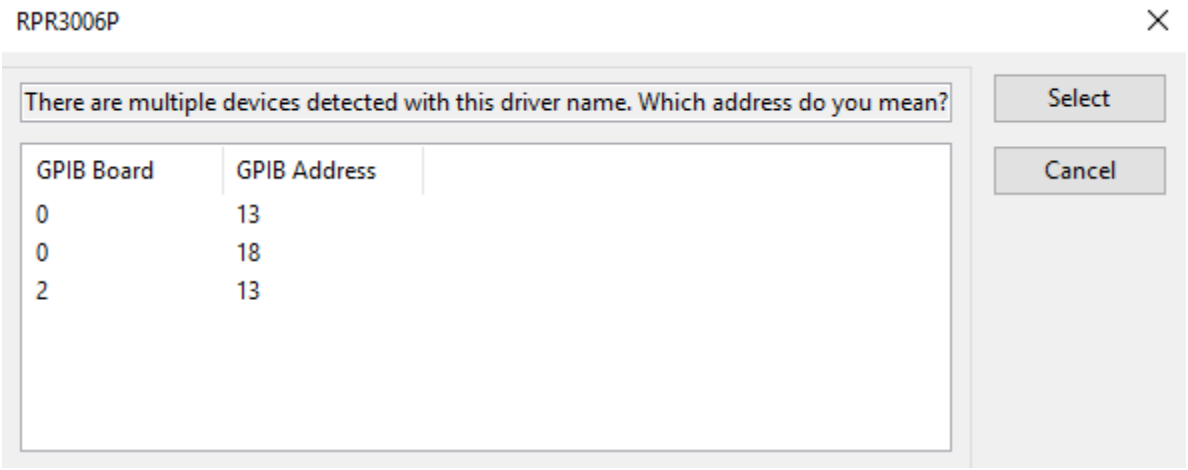


Figure 20 The GUI window shown when multiple devices are connected through GPIB

8.2 RS-232

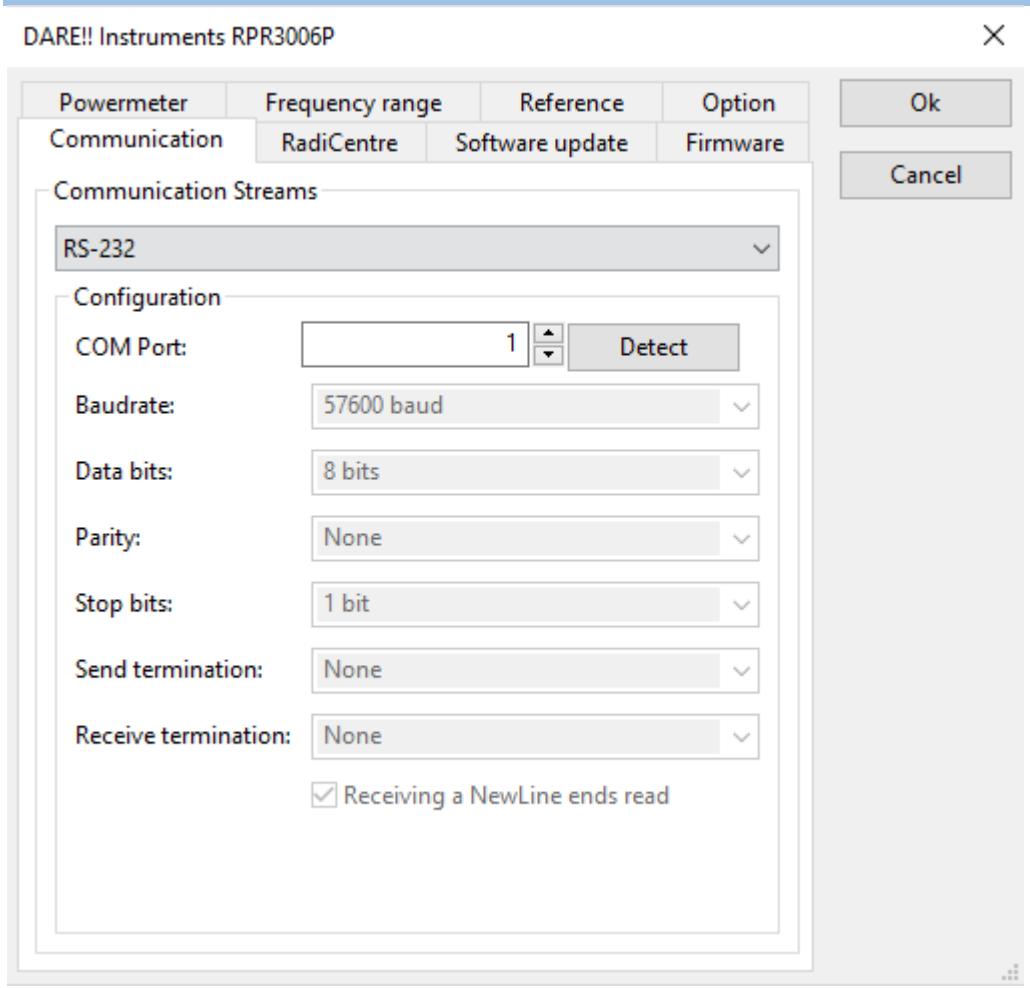


Figure 21 The window shown when configuring RS-232

8.2.1 What is RS-232

"RS-232 defines the signals connecting between DTE and DCE. Here, DTE stands for Data Terminal Equipment and an example for DTE is a computer. DCE stands for Data Communication Equipment or Data Circuit Terminating Equipment and an example for DCE is a modem. RS-232 uses serial communication, where one bit of data is sent at a time along a single data line. This is contrast to parallel communication, where multiple bits of data are sent at a time using multiple data lines."[4]

RS-232 is, to nobody's surprise, a communication protocol. This protocol Includes variables such as baud rates, start bits, stop bits and parity. Fortunately, it is rare that the start- and stop bits and parity changes settings other than default so these variables have not been considered when coming up with the design for this solution.

8.2.2 The general approach

The plan is basically the same as that of the GPIB auto-detection. Find all connected devices and for each of them run the CheckDevice() function on them.

8.2.3 VISA

At the very start of the code, the "Please Wait" window is displayed, as the searching with RS-232 needs three seconds per potentially found device. If VISA is installed, it is a better and more refined

way of communicating via RS-232 and is preferably used. The downside to this, is that certain COM ports are reserved, such as COM1 and COM2. To solve this, National Instruments created aliases to COM ports, called “ASRL”. VISA will attempt to have COM ports be the same as ASRL ports, but this does not always happen. Furthermore, with how currently the communication over RS-232 is built in RadiMation®, it can go like this: there is a device on COM9, which has an alias of ASRL11. In the code a request for communication is made with port 9. It will attempt to communicate over COM first, which is COM9. This will succeed. The problem is, when it moves on to 11, it will first communicate with COM11, which will fail. So now the code will search for ASRL11, which will succeed because it has an alias for COM9. The device is now found twice. This has to be prevented else duplicate data will be stored. In the event that VISA is not installed, only the COM ports will be searched.

To prevent this from happening, the code must acquire the list of aliases. Fortunately, there is a text file that can simply read and take the data from, which is visaconf.ini.

The first step, after showing the “Please Wait” window, is to check is VISA is installed. No need to fetch a file from VISA if it doesn’t exist. Of course, the file does not contain just the aliases, it contains a bunch of other data as well that is irrelevant to the code. So, RadiMation® begin to scan each line and then compare that line to the string “[ALIASES], which is the start of the relevant section. From this point on until the end of the section, marked with the word “NumAliases”, every line will be added to a vector of strings, so that later the code can scan through the relevant lines individually. Now the file is no longer needed, so the program closes the file. A correct line looks like this: “Alias11="'COM8','ASRL10::INSTR’” From this line, the numbers 8 and 10 are important. These numbers are paired and added to a map of numbers. This map will later serve as the base of an exclusion list, so that a device is not scanned twice. Another property that is being made of use is that, if the COM port and the ASRL value are different, the ASRL value is always higher.

8.2.4 Scanning

RadiMation® has fourteen set baud rates. Sometimes a baud rate is already set by a device driver inside RadiMation®, because when developing drivers for the devices, RadiMation® already has the correct baud rate saved for that device. This is not always the case, as certain it is simply unknown which baud rate has to be used. Therefore, in the scenario where it is not clear which baud rate has to be used, the code will have to scan each baud rate.

Even though a large amount of ports can be connected via RS-232, it seems unlikely the number of connected devices will pass further than 256 ports, so the code will start scanning from port 1 to port 255. The first thing to do is to check the exclusion list (empty at the start). Should the number the program is currently scanning be in the list, the number is skipped and start with the next number. This is to prevent the above situation with devices being scanned twice. Continuing, the port is now set using a function. This port will be referenced when trying to open a connection. Afterwards, a check has to be made to see if the settings of the current RS-232 configuration can be edited or not. When attempting to configure a device driver in RadiMation®, sometimes the settings are already locked, because DARE!! is sure that the device uses these settings. In this case, there is no need to scan each baud rate, just the one that is configured. Then, a connection is opened, by using the baud rate, parity, data bits and stop bits. If this succeeds, a CheckDevice() function is called for this device. If this does not return an error, the baud rate is added to a vector of numbers and so is the com port, as well as a counter that is raised for the number of successful matches.

Now, the alias of the COM port is added to the exclusion list, so that it is skipped, and it is not found again.

8.2.5 Determining the result

Same procedure as all the other solutions. Count the amount of matches from CheckDevice(). At no matches, error message, at one match, fill it in, at two or more matches show the GUI as shown in figure 22 and the end user has to select the device.

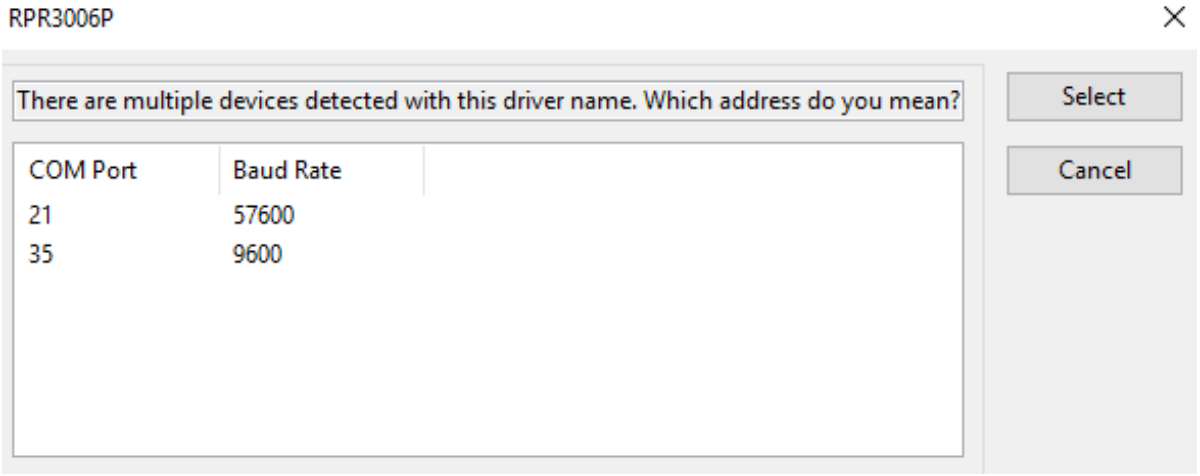


Figure 22 The GUI shown when multiple devices are connected through RS-232

9 | Competences

An internship should also contribute to the further development of the competencies that are required to adequately perform my activities after my education is completed and I am ready for “the real work”. Below, I have addressed how this internship has contributed to this requirement.

9.1 Analysing

It was necessary in my assignments, to extend the USB auto-detection function, to create a button for auto-detection with GPIB and RS-232 and to remove options in a list of communication protocols. For this I had to fully understand how the original code is structured and how it works, how roughly RadiMation® is build (such as its class structure) up, the level of complexity within RadiMation® and develop ideas as to how to build upon this foundation. Questions such as “how should the code work exactly?”, or “where I should edit the code?” must be asked. RadiMation® is an incredibly large and complex program and it is easy to get lost in it. For each one of these assignments an FNS was made, but these also required knowledge of the current state of the code. In these FNS’s I wrote about how the code should work and made a few terms to test the code on, once it is functional. Alongside this I also broke up the code in pieces and gave each piece a time estimate for my overall planning to make sure I finish my assignments on time. All this required a significant effort in the area of Analysis.

9.2 Design

Using this foundation, being the result of my analysis, a solution for each sub-assignment had to be designed. I have demonstrated this in each of the FNS’s to make a design based on what I had analysed. Such as how the new auto-detection button for USB should work, by adding in additional commands to receive more data from the devices, so I can sort them. With GPIB and RS-232 I thought about a similar solution of scanning all devices and then sending a command to them. With the removal of options, I needed a new method for making sure it switched communication protocols should one fail. Considering I was unable to find enough physical devices to test the communication with, I had to input dummy data (fake data) to see if my code worked, alongside plenty of debug messages that would show up in the log. To help me understand what needed to be done, I have created flowcharts that showed me inheritance trees of the classes and helped me visualise and draw a mental path to retrieving certain values. The FNS’s themselves are counted as documentation, as far as DARE!! is concerned. These FNS’s are attached to this report.

9.3 Realisation

While creating the code, there are guidelines set by DARE!! when it comes to programming. There is a large checklist that has to be filled in to make sure the code is stable and can be shipped with the next release of RadiMation®. An example of such is when variables are named, their first letter dictates their type, such as a string will start with an ‘s’ and an integer will start with an ‘I’, leading to variable names such as ‘sWord’ and ‘iNumber’. This practice has been committed/implemented to throughout the code. Due to the amount of code I had to create/modify, it was very unlikely that it would be correct the first time around. Therefore, much testing had to be performed to find and identify any bugs in the code, and to make sure the code is created properly and without error. The process for this is also roughly documented inside the FNS’s.

9.4 Manage

At the start of the internship I made a rough planning estimate of the time I needed for the various solutions I was targeting and when, taking into account that I needed some time to invest into other school activities for classes for not yet completed tests. In each of the FNS’s I had to break up certain aspects of the assignment and give an estimate of how long it would take to program the solutions. This was are hard to do due to being unable to estimate how much extra analysis had to be performed

with RadiMation's code and being unable to foresee any bugs that may arise. Every four weeks I had a conversation with my project supervisor about the current status of the assignment. Here we discussed any errors I was making and how to improve them. Every day at 12:00 the project supervisor was asking my colleagues and me if we had any questions about the code. This helped me with no longer hitting a wall most of the time and saving up time to use for the assignment instead of just thinking and not reaching a conclusion.

9.5 Research

First of all, RadiMation® is an incredibly complex program, written in C++ and it uses all of the special functions C++ was created for. The fact that it included more than 4500 classes is an indication of the complexity I was dealing with. It took me weeks of research and analysis to somewhat comprehend the structure and functioning of this application.

Before being able to create auto-detection code for RS-232 and GPIB, I needed to know how each of these protocols worked on a basic level. This would influence certain solution design decisions that I had to make. Although the functioning of these protocols is well described in the Internet, the RadiMation® code also includes functions that demonstrate how GPIB and RS-232 worked when connecting with a device. In addition, I had to consult a book on WxWidgets since I needed to use a property of a GUI that was not used before in RadiMation® and it was not well explained on the internet. Since RadiMation® is a highly complex C++ program, there are plenty of concepts used that I had no experience with, nor did I even hear about them. For example, virtual functions, the memmove function (eventually not used), vectors and much more.

10 | Conclusion

The three assignments that were given to me by DARE!! have been successfully completed. My activities have contributed to the simplification of the configuration process for the end user.

By introducing the auto-detection of the devices that are connected either directly to RadiMation® or indirectly via a RadiCentre®, the end user will not be confused about which protocol to select. Also, the removal of similar/redundant selection options when selecting protocols contributes significantly to the efficiency and user experience.

All these modifications were done through changing the source code of the program which is written in C++.

This internship at DARE!! was a challenge. The complexity of the RadiMation® program, being an application that has been growing over time, requires a thorough understanding of it structure and applied programming techniques. As a student and hence a “beginning C++ programmer” instead of an experienced Software Engineer, the complexity was sometimes overwhelming.

In the past, I thought I had learned many high-level concepts of C++, and I also thought knowing how to potentially find my way around a large project similar to RadiMation®. This project however caused me to be often confused and stuck and not exactly knowing how to continue. Thankfully, with the guidance of the DARE!! colleagues and my project supervisor I was able to overcome the challenges I encountered on route to completion of my assignment.

My colleagues were friendly and helpful, in the time that my project supervisor was unavailable I could rely on my colleagues to help me with certain issues. I have devoted all my energy to writing, as far as I know, bug free code (after testing of course) and I trust that, with my additions to and modification of, the source code in C++, DARE!! will be happy to include it in the next version of their product.

With these additions, the end user is now more easily able to find connected devices through RadiMation® instead of needing to look up any ID for each device. Not only that, there is now clarity in the options available and so RadiMation® can no longer confuse the user when it comes to making choices during the configuration process.

11 | References

1. https://www.electronics-notes.com/articles/analogue_circuits/emc-emi-electromagnetic-interference-compatibility/what-is-emc-basics-tutorial.php
2. https://www.dare.eu/emc-systems
3. https://www.ni.com/nl-nl/support/downloads/drivers/download.ni-488-2.html#329025
4. https://www.electronicshub.org/RS-232-protocol-basics/
5. https://www.electronics-notes.com/articles/test-methods/gpib-ieee-488-bus/what-is-gpib-ieee488.php
6. Cross-Platform GUI Programming with wxWidgets, by Julian Smart and Kevin Hock, second edition

Document 1248

--- FUNCTIONELE SPECIFICATIE [FNS] ---

Multiple DARE!! devices address auto-detect

Written by	:	RAHE
Filename	:	FNS - Support for connecting multiple devices.docx
Date	:	7-11-2019

Change log:

Version	Date	Changes	By
1.0	7-11-2019	Initial Version	RAHE
1.1	12-11-2019	General changes	RAHE
1.2	2019-11-19	Review and a few additional questions	JORO
1.3	20-11-19	Answer questions	RAHE
1.4	2019-12-10	Review and a few additional questions	JORO
1.5	2019-12-10	Answer additional questions	RAHE

Approved by (*):

Date:

Senior Engineer / Customer

(*) The design review document must be approved by a senior engineer before proceeding to the next development stage. In the case a senior engineer carried out the design review, another senior engineer must approve this document.

In the case of an external customer, this customer also must approve and sign this document.

Every version of the document must be signed by a digital signature. The signed document must be printed as PDF and stored in the relevant project directory.

Omschrijving product:

Product naam:

Multiple DARE!! devices address auto-detect

Doel:

Allow end users with multiple connected DARE!! devices to have all addresses configured by pressing a single button.

Financieel

What is the planned selling price?

- This feature will be part of the standard product for all customers with a valid support contract.

What are the planned production costs?

- No production costs.

What are the expected production numbers?

- Not applicable

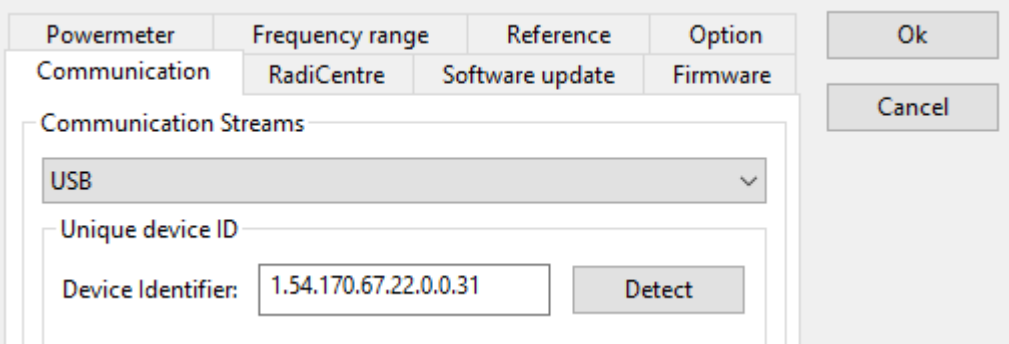
Specifications:

A simple button should be added to the GUI that when clicked, finds all connected DARE!! Devices and fills in the correct address given which device driver is currently open

Requirements:

A button in the GUI with “Detect” on it, in the communication tab, with the USB option.

No visual changes have to be made, the functionality of the existing button will be changed.



Preconditions:

One or more DARE!! Devices have to be connected. Should none be found, an error is displayed with the text “No DARE!! device found”.

The function should look at what device driver is currently open, fetch all connected devices, compare the names of the device driver and all connected devices, and if one matches, grab the address and auto-fill it.

Two problems may arise with this function.

- Problem 1: If multiple devices of the same type are connected
- Problem 2: If devices are connected to a RadiCentre, which is connected to the PC

Problem 1 can be solved by showing a list of addresses where the end user has to select the correct one for the device.

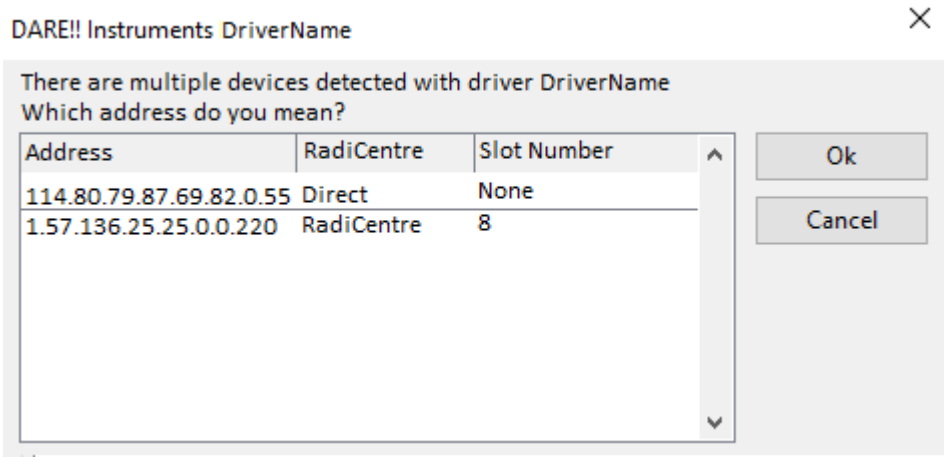
Problem 2 can be solved by scanning again from the viewpoint of the RadiCentre

Design:

According to the analysis, a suggested design has been created:

1. The existing button in the USB settings of the communication tab in DARE!! Device drivers’s functionality will be slightly changed
2. The name of the currently opened device driver will be stored in a string.
3. A list will be created of the names of all currently connected devices
4. Every name will be compared to the one stored

5. Should multiple matches occur, a second list will be generated for the end-user to interact with and select the correct address. The list should look like this:



The name "DriverName" used here is merely an example. It will be replaced with the name of the currently loaded device driver. The Ok button will always be selectable, but if no selection is made, an error message will show up with the message "No selection is made! Please select an address".

6. Should a RadiCentre be found, then the RadiCentre's address should be ignored and instead the search must continue through the RadiCentre. This can be done by asking the RadiCentre what devices are currently connected to it, and adding that list to the original one. Once completed, the search continues as normal. At the end of the scan all devices with the same name will be listed, even those found through a RadiCentre.
7. Should no device be found, a number of error messages can be shown to the user depending on the situation, such as:
    a. No connected DARE!! Devices are detected at all
    b. Another DARE!! device (RPR3006W) is found but that is another model device
    c. Multiple other DARE!! devices are found, but none of them is of the correct model.

Estimate:

1. Researching methods to find connected devices (10 hours)
2. Implement a search function (8 hours)
3. Have the function search after a RadiCentre is detected (15 hours)
4. Extra GUI screen for selecting addresses (10 hours)

Testen van het prototype:

- Connect no device, press the button. It should show one of the error messages
- Connect two different DARE!! devices, if it is the correct driver, only the correct address should be inserted into the box
- Connect two of the same DARE!! devices, if it is the correct driver, the GUI should pop up
- Repeat the tests above with a device connected through a RadiCentre, the results should be the same as the previous tests
- Repeat the tests above with two of the same DARE!! Devices through a RadiCentre

Document 1248

--- FUNCTIONELE SPECIFICATIE [FNS] ---

Reduction of options within device driver communications

Written by	:	RAHE
Filename	:	FNS - Reduction of options within device driver communications
Date	:	7-11-2019

Change log:

Version	Date	Changes	By
1.0	7-11-2019	Initial Version	RAHE
1.1	12-11-2019	General Changes	RAHE
1.2	2019-11-19	Review and additional questions	JORO
1.3	20-11-19	Update on questions	RAHE
1.4	2019-12-10	Review and additional questions. Added backwards compatibility	JORO
1.5	2019-12-11	Answer additional questions	RAHE

Approved by (*):

Date:

Senior Engineer / Customer

(*) The design review document must be approved by a senior engineer before proceeding to the next development stage. In the case a senior engineer carried out the design review, another senior engineer must approve this document.

In the case of an external customer, this customer also must approve and sign this document.

Every version of the document must be signed by a digital signature. The signed document must be printed as PDF and stored in the relevant project directory.

Omschrijving product:

Product naam:

Reduction of options within device driver communications

Doel:

Reduce the confusion for the end-user when configuring a device

Financieel:

What is the planned selling price?

- This feature will be part of the standard product for all customers with a valid support contract.

What are the planned production costs?

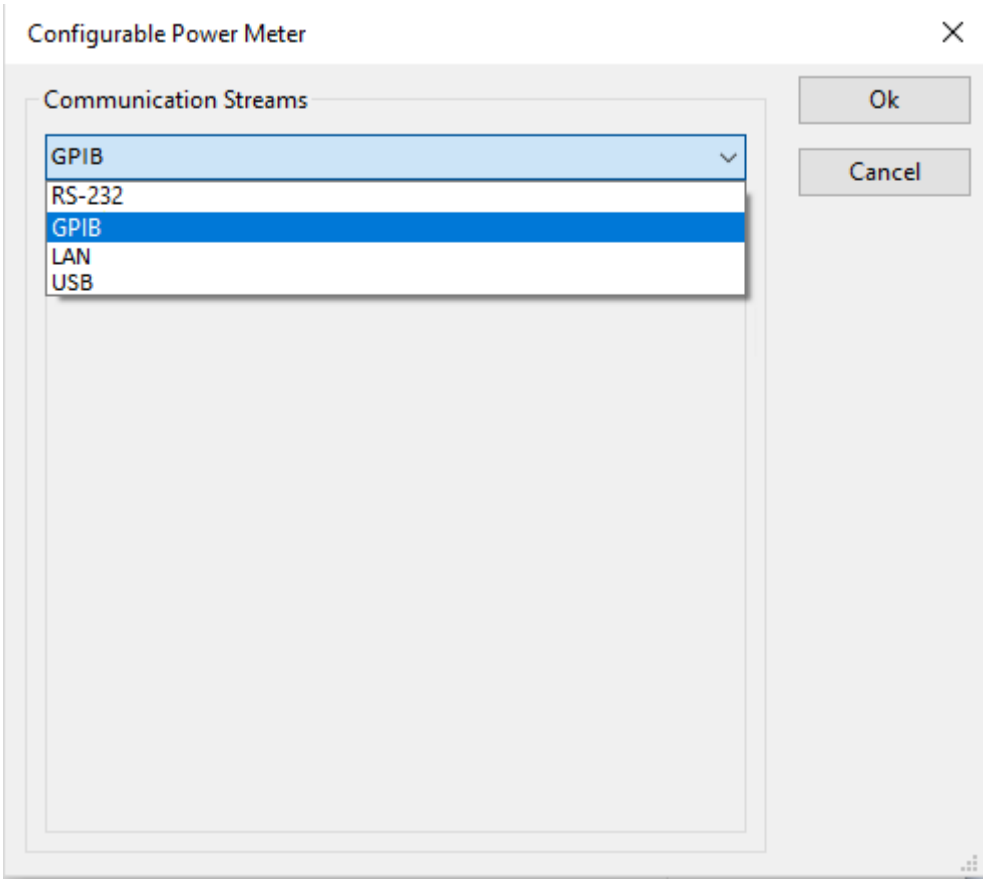
- No production costs.

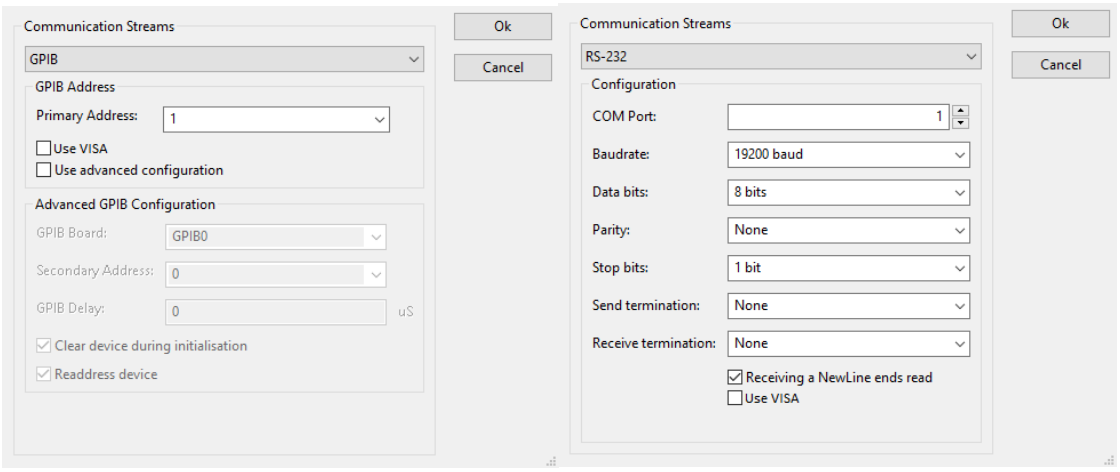
What are the expected production numbers?

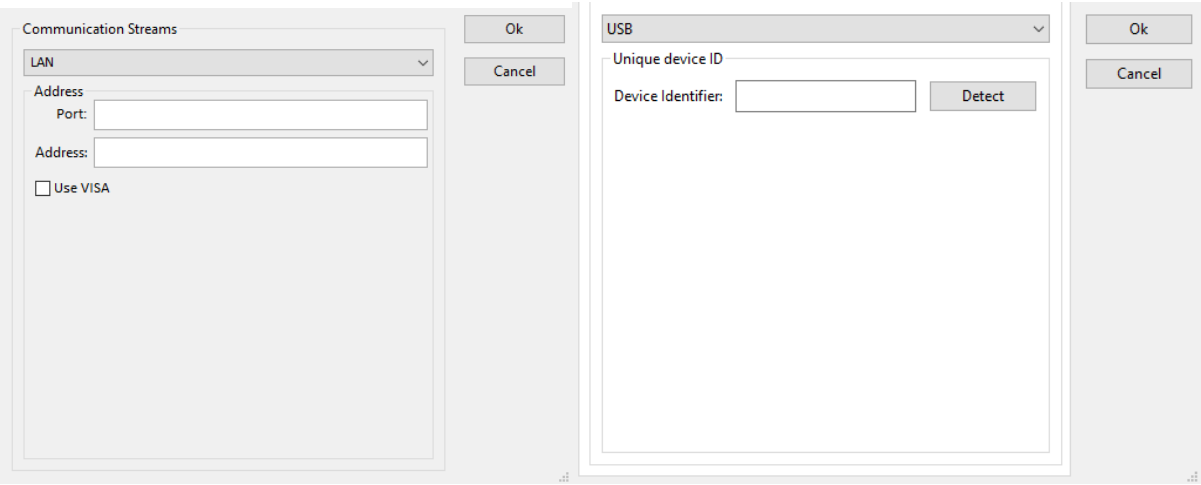
- Not applicable

Specifications:

Certain options in the communications tab are confusing for the end user, as there are three ways to open a GPIB connection (GPIB, CompliantGPIB and through VISA), two ways for RS-232 (through WindowsAPI and VISA) etc. The end result requires removal of duplicate options, and only the connection methods remain, which is USB, LAN, GPIB or RS-232.







Options such as VISA-Alias and the port selection of LAN based instruments will still be available, under the correct tab. If VISA is not installed, the VISA-Alias option is greyed out. The VISA option should be enabled by default.

All the device drivers that are currently using a communication method that is not present anymore in the new situation, should automatically be converted to the corresponding new available communication stream, with the correct settings.

An existing working device driver should remain working after this new implementation. (Backwards compatibility!)

Requirements:

Removal of the options in the communications menu

Preconditions:

None

Analysis:

To reduce the amount of options a potential solution is a priority system. In the background a series of checks should be made to choose the correct method of communication.

All options except for the protocols themselves should be removed, resulting in less confusion for the end-user. The options window still exists for tweaking these communication methods. The button used to connect to the device shall be used for the auto-detection. It should scan for the device, using the settings set in the communication menu.

Design:

According to the analysis, a suggested design has been created:

8. Edit the function that is called when a connection to the device is made
9. Depending on how the communication is configured, an amount of checks will happen
    a. When GPIB is selected, VISA>GPIB
    b. When LAN is selected, VISA>HTTP>TCPIP>UDP>RSIB

c. When RS-232 is selected, VISA>WindowsAPI
d. When USB is selected, USBTMC>DARE USB>VCP

10. These checks are based on what is possible within the communication settings. Not all devices can do communication via USB, if it's not possible, it should be skipped.
11. Checks will happen based on what is installed (VISA for example), and just trial and error for each option. If VISA is not installed the first attempt can be skipped.
12. Check if the first configuration is possible. If it is, select it and stop there. If it isn't, attempt to do the same with the next one down the line
13. When all options fail, the device driver only has to return the corresponding ERRORCODE. The error message such as "Failed to find device! Is it connected properly and is the correct port chosen?".
14. Removal of the options VISA, UDP, TCPIP, HTTP, RSIB, CompliantGPIB

Estimate:

5. Separate check for each protocol (30 hours)
6. Function to go by each method (3 hours)
7. Removal other options (5 hours)
8. Ensure that existing device driver will be converted to one of the new possibilities (20 Hours)

Testen van het proto type:

- Test if a device can connect with and without VISA on. It should connect in both situations
- Have separate devices attempt to connect through TCPIP, UDP and HTTP with only the LAN option selected
- Connect a device through GPIB
- Connect a device through LAN
- Connect a device through USB
- Connect a device through RS-232
- For each of these, see if the first possible method went through
- Connect a device with deliberate wrong settings, attempt to get every warning message to pop up

Document 1248

--- FUNCTIONELE SPECIFICATIE [FNS] ---

Automatically finding the address of devices connected through RS-232 or GPIB

Written by	:	RAHE
Filename	:	Automatically finding the address of devices connected through RS-232 or GPIB
Date	:	7-11-2019

Change log:

Version	Date	Changes	By
1.0	7-11-2019	Initial Version	RAHE
1.1	13-11-2019	General Changes	RAHE
1.2	2019-11-19	Review and additional questions	JORO
1.3	2019-11-20	Answer questions	RAHE
1.4	2019-12-10	Review and small additional questions	JORO
1.5	2019-12-10	Answer small questions	RAHE

Approved by (*):

Date:

Senior Engineer / Customer

(*) The design review document must be approved by a senior engineer before proceeding to the next development stage. In the case a senior engineer carried out the design review, another senior engineer must approve this document.

In the case of an external customer, this customer also must approve and sign this document.

Every version of the document must be signed by a digital signature. The signed document must be printed as PDF and stored in the relevant project directory.

Omschrijving product:

Product naam:

Automatically finding the addresses of connected devices through RS-232 or GPIB.

Doel:

Allow end users to press a button and RadiMation® automatically finds the correct address.

Financieel:

What is the planned selling price?

- This feature will be part of the standard product for all customers with a valid support contract.

What are the planned production costs?

- No production costs.

What are the expected production numbers?

- Not applicable

Specifications:

A simple button should be added to the GUI that carries out the function of detecting at which address the device is connected.

Requirements:

A button in the GUI with “Detect” on it, in the tab of RS-232 and GPIB.

Preconditions:

A device has to be connected through RS-232 or GPIB.

Analysis:

GPIB

After scanning all GPIB boards in the advanced settings, every possible GPIB port (except the address of the GPIB board itself) has to be scanned. A device will eventually be found (if connected).

RS-232

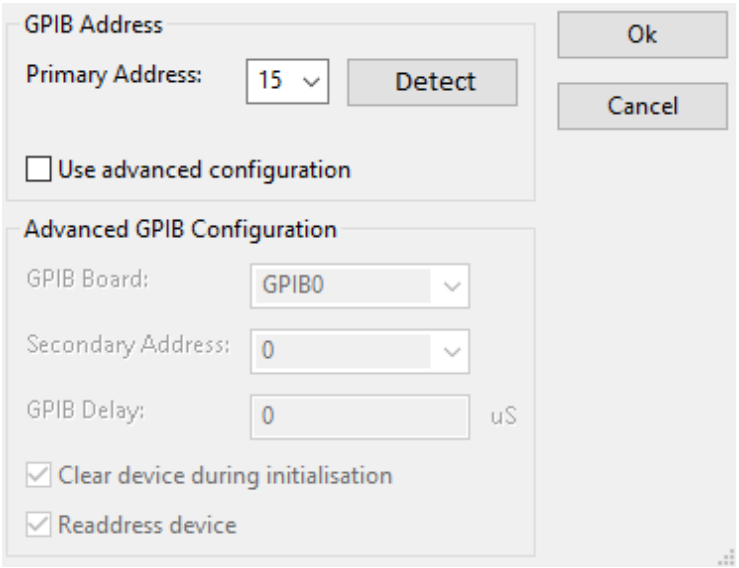
The problem with RS-232 is that it can span across different baud rates. To solve this, first a scan should be made on the default set baud rate. Should this fail, a scan should be made on all available standard baud rates.

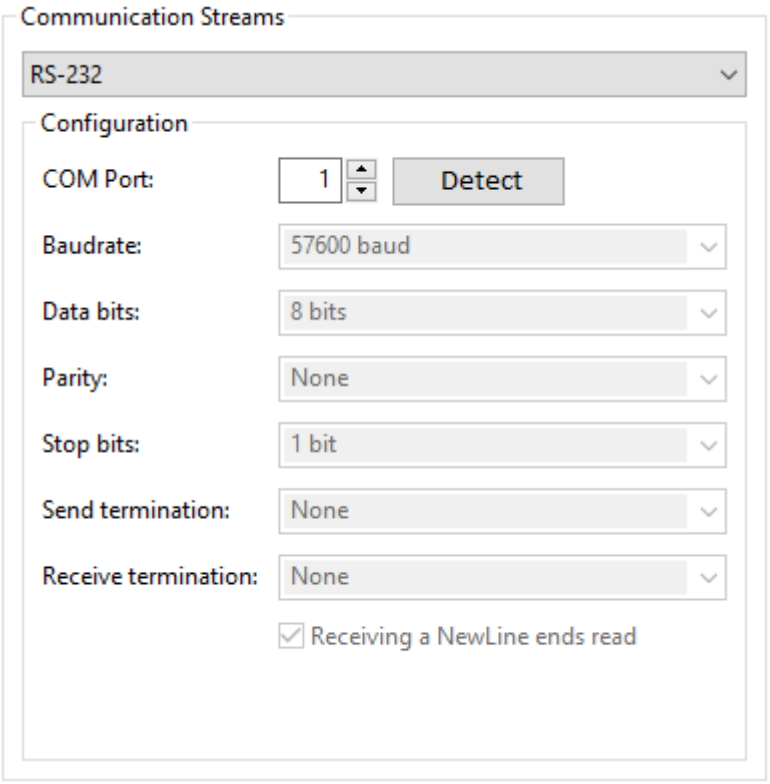
In both cases, if multiple devices are found a list should pop up showing all the ports with connected devices. The data contained in this list can be checked by using the existing CheckDevice function.

Design:

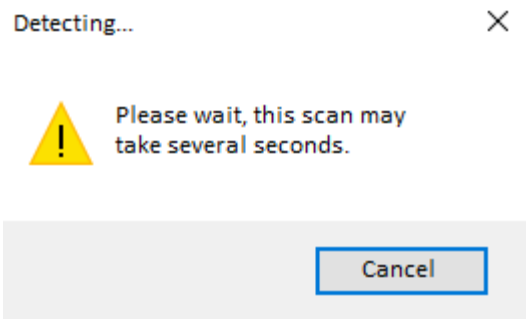
According to the analysis, a suggested design has been created:

A button will be placed in the GPIB and RS-232 tab, looking like this:



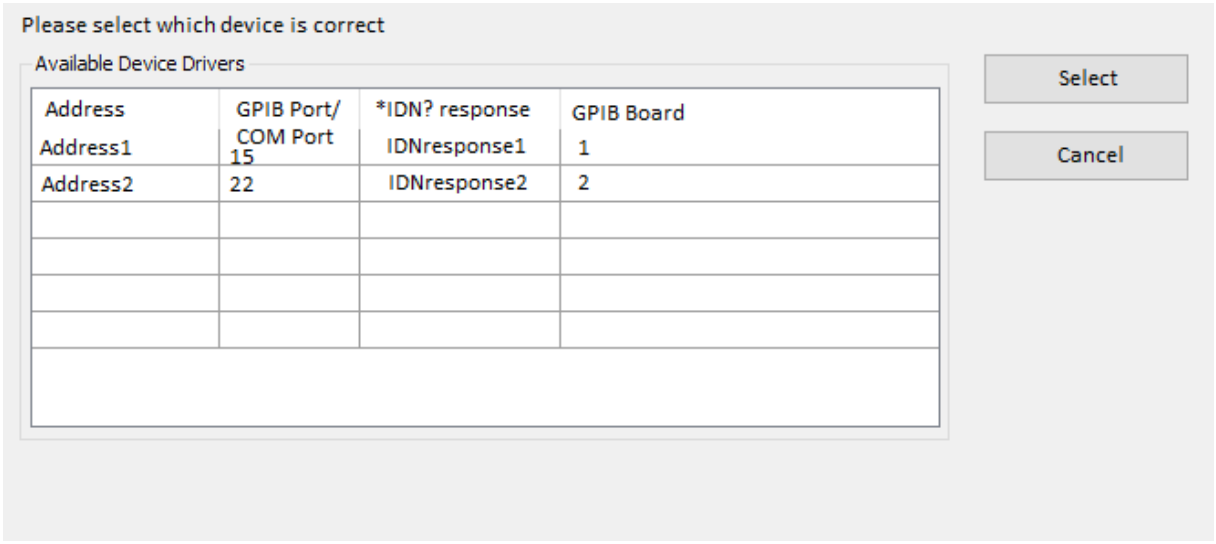


1. Once the 'Detect' button is pressed, it will scan each GPIB port (if in the GPIB tab) or each RS-232 port (if in the RS-232 tab). A warning message will pop up, preventing anything from happening until this scan each completed. The only other option other than waiting, is a cancel button to stop the search early. The message should display the text "Please wait, this detection may take several seconds".



2. If no device is a found, an error message is given to the user, which will be: "No device found, is everything connected properly?".
3. If only one device is found, the address of that device will automatically be selected in the 'Primary Address' or the 'COM port' setting. However in the event that the loaded device driver does not match the name of the connected device, a warning message should pop up saying "A device has been detected but is not the same as the device currently being configured. Do you want to continue?" followed by a yes or no button.
4. Should multiple devices be discovered, they will all be compared to the currently loaded device driver. If only one matches, it picks that one. Should multiple devices pop up that match the name of the loaded driver, a GUI should pop-up allowing the user to find the

names of the devices.



5. In the event that that the detection is RS-232, a COM Port is shown in the table above. If it is GPIB, GPIB Port is shown instead. Two similar GUI's will have to be made for this.

Estimate:

9. Add a button to the GUI (4 hours)
10. Create the GUI to display all devices (10 hours)
11. Functionality to go past each address (20 hours)
12. Show the 'Busy' dialog with the ability to cancel the running search (5 hours)

Testen van het proto type:

- Connect no device, press the button, it should show the user an error message, such as "No device found on xxx bus"
- Connect a device through RS-232 at 57600 baud
- Connect a device through RS-232 at a different baud rate
- Connect a device through GPIB
- Connect multiple devices through GPIB
- Connect multiple devices through RS-232
- Possible to find the correct GPIB device when it is connected to the second GPIB adapter of the PC?