Joona Matikainen

Controlling RS-232 equipped devices using Raspberry Pi and C++

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme in Information Technology

Thesis

29 November 2018

Metropolia

Author Title Number of Pages Date	Joona Matikainen Controlling RS-232 equipped devices using Raspberry Pi and C++ 32 pages 29 November 2018
Degree	Bachelor of Engineering
Degree Programme	Degree Programme in Information Technology
Professional Major	Software engineering
Instructors	Olli Hämäläinen, Senior Lecturer

The objective of this thesis was to develop a prototype solution on how to control RS-232 equipped devices with inexpensive hardware using a general programming language, for a company working in the audio-visual (AV) industry.

Different hardware and programming languages were explored and weighed against the specifications of the customer. The hardware selected was the Raspberry Pi 3 Model B and the programming language was C++. The working solution was approved by the customer upon demonstrating its functionality. The price of the hardware and the versatility and universality of the programming language were to the customer's satisfaction.

The customer was given full access to the developed solution, upon which further development by the customer may be made. Future development ideas, which can be based on this thesis, were also presented to the customer.

Keywords	RS-232, RS-232 control, Raspberry Pi, C++

Metropolia

Contents

List of Abbreviations

AV	Audio-visual.
CLI	Command-line interface.
CR	Carriage return. Derived from typewriters, a carriage return returns the paper carriage back to the initial side of the paper.
CPU	Central processing unit or in simple terms the computer's processor.
GPIO	General-purpose input/output.
GUI	Graphical user interface.
LF	Line feed. Derived from typewriters, a line feed moves the paper on a typewriter up by a single line.
OS	Operating system.
PID	Process ID. A unique identification number given by the operating system to an active process.
RAM	Random access memory. The computer's temporary memory used for real time processing purposes.

1 Introduction

This thesis explores, discusses and demonstrates how to implement RS-232 communication between a computer program created using the C++ programming language on a Raspberry Pi 3 Model B and any other RS-232 enabled device. The thesis is done for a company in the audio-visual (AV) industry. The name of the company is not published in this thesis and will be referred to as the “Company”.

The Company is interested in controlling RS-232 enabled AV devices, such as displays, projectors and HDMI switchers, in a more cost-effective and creative manner, than what the AV industry currently offers. The AV industry has its own set of established manufacturers, such as Crestron, Extron, AMX and Kramer to name a few, all of which offer computers, known as control processors, for controlling various AV devices. These control processors are either programmed or configured by an AV implementor, such as the Company, to control the power states, volume levels or selected picture source inputs of all the AV devices in for example, a meeting room.

However, the challenge with such devices is that they are rather expensive, due to their proprietary nature, and do not use standardized programming languages to create the control programs with, nor do they offer access to the actual operating system (OS) of the control processor. This lack of standard programming languages and access to the operating system limit certain creative possibilities, as the control processor’s manufacturer’s software is between the AV implementor’s program and the operating system.

The given task was to find hardware and software that meet the computing and physical dimension specifications of the Company’s installation environments and create a prototype program demonstrating its functionality. RS-232 was selected as the communication standard with which external device control was to be established, as all the displays and projectors the Company uses in its projects have RS-232 installed and almost all their projects require some picture output device.

The following specifications were defined by the Company. The solution needs to:

- be able to process multiple programs simultaneously.
- be able to connect to a local area network with a network cable. Wireless network connectivity is a bonus feature, but not required.

- support generally used object-oriented programming languages.
- fit between a wall mounted display and the wall the display is mounted to.

The Company provided the funding for any hardware purchases needed during the process and gave access to its office to be used as testing space. The Company has several projectors and displays in their office with which the RS-232 communication could be tested. The thesis is approached using the top-down design.[1] It comes as a natural approach as the task given is an abstract need without specifications on the exact details of what needs to be used to create the final solution.

2 Platform selection process

The first part of the platform selection process was getting familiar with a physical installation environment. The environment was provided by the Company in their office, as they have both displays and projectors readily installed in their demo rooms. The smallest gap the device needs to fit to is the gap between the back of a wall mounted display and the wall. The projector was of no real concern as a small computer can be hidden on top of a ceiling mounted projector.

By observing such an installation in the Company's office and by referring to the technical drawings of display wall mounts the Company frequently uses, the device should fit the approximately 40 mm gap between the display and the wall. Almost any modern microcontroller, tablet computer or laptop computer can fit between this 40 mm gap. This allows for a wide selection of hardware to suit the need and allows the selection process to focus on finding the proper computing platform without needing much attention to the physical size of the hardware.

The computing specifications require the support for a well-known object-oriented programming language. The solution is to choose an operating system which supports the use of widely used programming languages. The most commonly found operating systems are Windows, macOS and Linux.

macOS is the operating system developed by Apple Inc. to run only on their own Mac computers.[2] Of the currently available models MacBook, MacBook Air, MacBook Pro and Mac mini would fit the physical requirements of fitting between 40 mm. The downside of the Apple Mac computers is their price range. The physical, operating system and other technical specifications, such as networking, meet the requirements, but do not quite fit in with the requirement of cost-effectiveness. The cheapest of the Macs is the Mac mini with a starting price of approximately €570. This is around the same price range as the devices currently used by the Company.

2.1 Raspberry Pi 3 Model B selection

Of the other two popular and widely supported operating systems Windows and Linux can be installed in a much wider range of hardware, as they are not restricted to certain

devices such as the macOS by Apple. A simple search in the small sized or “mini” computer products sold by Finnish electronics marketers, such as Verkkokauppa.com and Partco, gives the cheapest computer, with support for widely used operating systems, to be a Raspberry Pi Model A+. The Raspberry Pi Model A+ is an older model of the Raspberry Pi which has only one CPU core, with a price of approximately €30. Next in the search results is the Raspberry Pi 2 model B with a price of approximately €40. Going down the search further the latest model, the Raspberry Pi 3 Model B, comes up with a price of approximately €45.

With such small absolute price differences, it makes more sense to choose the latest model (Raspberry Pi 3 Model B) with the best connectivity options and most computing power. Additional costs to the computer itself are the (2.5A 5V) power supply and separate casing. The separate casing is needed, because the Raspberry Pi itself consists of only the motherboard and all its attached electronics and connectors. A full Raspberry Pi 3 Model B kit, including the Raspberry Pi itself, the power supply, memory card and casing, costs approximately €80.

The Raspberry Pi 3 Model B meets the requirements of the Company as it has enough processing power to run several applications at the same time. It also has the option of installing a lot of storage memory in the form a Micro SD card and by expanding the storage memory to external storage devices accessed by USB, such as USB HDDs. It has integrated network connectivity using Bluetooth, wireless or wired LAN. The four onboard USB 2.0 ports allow the connection of several different types of devices. A very special hardware feature is the 40-pin general-purpose input/output (GPIO) which allows the connection of a vast amount of lower-level communication devices, such as sensors and relays.[3]

2.2 Operating system selection

The Raspberry Pi 3 Model B supports several operating systems. Both Windows and Linux operating systems can be installed on the Raspberry Pi 3 Model B. The Windows version in question is the Windows 10 IoT Core.[4] There are several Linux distributions that can be installed, of which Raspbian is the most popular as it is the only operating system supported by the Raspberry Pi Foundation, the manufacturer of the Raspberry

Pi computers. Other operating systems, including the Windows 10 IoT Core, are compatible with Raspberry Pi 3 Model B, but not officially supported by the Raspberry Pi Foundation.[5]

The Company had not given any specifics on the operating system other than the support for well-known object-oriented programming languages. Raspbian was selected as the operating system for the Raspberry Pi, because of its official support status by the Raspberry Pi Foundation. The Raspberry Pi Foundation has extensive tutorials, manuals and documentation on how to use the Raspberry Pi with Raspbian, making it the most sensible option. The official support also means, that the Company is not restricted to the know-how of a single IT professional, as it has the option of getting support from the community built around Raspberry Pi and possibly even directly from the Raspberry Pi Foundation. Raspbian can be downloaded as two versions from the official Raspberry Pi website: Raspbian with preinstalled development tools and a full graphical user interface (GUI) desktop or Raspbian Lite, a “headless” minimalistic release with essential programs and the Bourne Again Shell (bash) command-line interface (CLI) for user interaction.[5] Raspbian Lite was chosen, as there is no real need for a graphical user interface in the wanted solution of RS-232 communication.

2.3 Programming language selection

The current programming language used by the Company in their projects is not object-oriented, but a procedural sandbox version of C/C++. This has posed certain limits to how reusable the coding can be made compared to object-oriented programming languages. In a procedural programming language all data manipulation is done to variables using operators or separate functions. These separate functions are tedious to use as they do not “travel” together with certain datatypes, but instead must be remembered separately to be used for a specific task, even if that function was specifically created for a very particular purpose.

With classes, in object-oriented programming, it is possible to create objects that have built-in attributes and methods. Variables are also referred to as attributes and functions as methods in the programming community and are used interchangeably with variables and functions when discussing classes. Classes can be used as datatypes. An instance of a class is called an object. The object, being an instance of a class, has the class’s attributes and methods always available, making the object a powerful and almost self-

sustaining entity in the program. The object can perform initializations to itself when created and manipulate the values of its attributes using its methods without having to call separate functions from the outside to perform the object specific data manipulations. This makes the maintaining of the code easier and makes the code more reusable. Reusability is increased if a class is inherited by a sub or child class. Further reusability may be introduced by creating interfaces, which can be described as abstract classes.[6]

Two popular object-oriented programming languages were screened and compared with the aim of finding a long-term programming basis for new implementations. The two programming languages in question are Python and C++. No other object-oriented languages were screened, because these two are amongst the most popular languages and are very well known in programmer communities, making it easier for the Company to hire new programmers in the future.

The first examined programming language was Python. The official Python website, maintained by the Python Software Foundation, says the following about Python: “Python is an interpreted, interactive, object-oriented programming language.”[7] The interpreted and interactive parts of Python come from the Python shell environment in which Python programs or scripts are interpreted on-the-go. Some call Python a script-like language because of this feature. The Python scripts can be loaded to the Python shell environment where they are read one line at a time. Python scripts are not compiled to suit the computer’s processor architecture but are instead interpreted in the shell during runtime. This makes Python a good choice when cross-platform operation of the scripts is a key feature in development.

The Python shell is interactive. One may explore and test small snippets of programming in the shell environment without having to create a separate source code file every time. This makes quick tests easier and more agile. It is interactive also in the aspect, that as a Python source code file is interpreted in the shell environment and the shell encounters an error, it informs the user of what type of error it encountered and on which line of the source code file the error occurred. This makes the need of a separate program debugger unnecessary as the shell handles debugging features extensively on its own.

As an object-oriented programming language Python meets all the requirements of the initial specifications, see section 1. It has a very clear syntax which is easy to understand and approach for beginners in programming and is used as an entry-level programming language in many circumstances. The Python standard library is very extensive and has

very detailed documentation on all its classes and methods. As an object-oriented programming language Python does not have any shortcomings. However, it does have its shortcomings in certain lower-level applications and proprietary uses, especially in embedded devices.

Python requires its shell environment in one way or another for the language to work, because the shell is what does the interpreting of the source code. In practice this requires a compatible operating system on which the shell must be installed and run. This is not necessarily a huge problem technically, as Python's source code is open and fully downloadable from the official website.[8, 9] If a ready installable Python shell does not exist for a certain operating system it is possible to create one using the source code. However, this does require more effort from the developer and in some cases, it might be technically impossible, especially for small embedded devices which do not have a higher-level operating system.

Another shortcoming, from the perspective of a proprietary device developer, is that of unhidden source code. Even though it does not require the source code to be hidden for it to be proprietary, as it is something defined in the license of the program, it something worth paying attention to, because the world is not made of only law-abiding citizens. Hiding the source code is a means of defending against piracy and copyright infringements. By hiding the source code into byte format, which is not human readable, an additional layer of protection is added against illegal copying.

Although this may not stop the most talented and enthusiastic crackers, it does make the task of cracking the executable program into source code format much harder. Hard enough to stop small time copycats, such as competitors looking for a shortcut in product development, from copying the source code and implementing it to their own projects. This added level of security is something worth considering when dealing in a highly competitive market where it is essential, that the product is not immediately reproduced by competitors.

A Python script, equivalent of source code, is not naturally hidden. The script files, with the extension of .py, are intended to run on the Python shell as if a user were interacting with the shell directly. It is possible to compile Python scripts into bytecode, which is not humanly readable, but it has its shortcomings too. The generated .pyc files are not portable between different versions of the Python shell and they can be easily decompiled with certain programs such as decompyle.[10, 11] This limitation may be overcome using

programs such as PyInstaller, which create an executable containing both the Python interpreter and the user created scripts.[10, 12] The drawback of this is that software such as these do not necessarily support all new releases of Python libraries, which may or may not be essential in a product's development.[13]

C++, like Python, is an object-oriented general-purpose programming language.[14] But unlike Python, C++ does not need a separate shell environment for it work and run. Instead it runs on a lower-level of abstraction in the computer system. C++ is a complied programming language, meaning its human readable source code is translated into machine code.[14]

The main advantage of a compiled programming language is its speed of execution after compilation, because the executable generated by the compiler is run as a native program in the operating system.[14] The limitation of a compiled programming language is, that it needs to be compiled separately for each platform it is implemented in, making it harder to deploy across different platforms.[14]

Having the program run on several platforms is not a concern for the Company in this thesis, as they intend to use a single device to solve all the RS-232 limitations they currently face. The advantage of having only the compiled executable on the device compared to the implementation options of Python is the reason the Company chose C++ as the programming language for the implementation. C++ fulfills all the programming language requirements presented by the Company, including source code protection.

3 RS-232 and AV devices

RS-232 was originally introduced in 1960 as Recommended Standard 232, by the Electronic Industries Association (EIA). The standard defines the electrical signal characteristics, interface mechanical characteristics and circuitry intended for communication between data terminal equipment (DTE) and data circuit-terminating equipment (DCE). The standard does not define character encoding, the order the bits travel or other logical protocols. It is a standard for the physical medium for data transfer. Currently the standard is maintained by the Telecommunications Industry Association (TIA) with the current standard's revision being TIA-232-F. The common reference to the standard is the original name of RS-232, even though the industry standard has changed the standard's name.[15]

On AV devices, the most common physical interface of an RS-232 port is the DE-9 connector. This is also the most common interface found on USB – RS-232 adapters. Serial ports using RS-232 are still commonly found on almost all professional projectors, displays and digital sound processors. The advantage of serial ports, including the RS-232, is that they are inexpensive to manufacture, they can transmit almost any type of information and can use very long cabling.[16]

The more modern universal serial bus (USB) can have a single cable length of 5m or a maximum of 98ft (30m) using hubs in between.[16, 17] A single RS-232 link without any additional hubs or repeaters in between can be as long as 130ft (39.6m) or more.[16] This comparison is crucial in the AV industry, because cable lengths may be very long from a control processor to a projector in the ceiling.

Even though longer cabling (100m) can be achieved with CAT5e and newer Ethernet cabling standards, and TCP/IP or UDP/IP control is enabled in many new AV devices, there have been issues with the AV device manufacturer getting the Ethernet protocols or ports to function properly.[18] For this reason many AV integrators still rely on the RS-232 control ports for device control, because they have generally had less problems. This is also the reason why manufacturers such as Crestron provide RS-232 control ports on their newest devices. Crestron's DM-NVX-350 series, a 1000BASE-T based 4K image and audio encoder and decoder, has a RS-232 port for external device control, such as display or a projector.[19]

The most common RS-232 variant used in the AV industry is the most minimal physical configuration of 3-wires for bi-directional communication. The 3-wire RS-232 consists of the transmit data, receive data and ground lines.[15]

4 Development environment and tools

The development of the solution was done using the Linux distribution Ubuntu, installed on an HP EliteBook 850 G1 laptop.[20, 21] Ubuntu is based on the Debian Linux distribution, which is also the same distribution on which Raspbian is based on.[22, 23] Because both operating systems are based on Debian, they use very similar, if not identical, file systems and bash shell commands. This allows most of the development to be done in an almost identical environment, but with a wider range of programming tools and more powerful hardware.

Atom was used as the code editor during the project. It is an open-source editor with syntax support to the most widely used programming languages, including C++.[24] C++ could be programmed using any text editor. Atom was chosen for comfortability of programming, not necessity. Other needed development tools were the Crestron QM-RMC, the g++ compiler and Google Drive.

4.1 Crestron QM-RMC

The Crestron QM-RMC is a compact control system processor designed for working in AV environments.[25] The device was provided by the customer to work as a testing subject in RS-232 control. Crestron Electronics, Crestron for short reference, produces both hardware and software designed for home automation, building and enterprise management and presentation systems. The Company uses Crestron’s products in many of their AV installations and was therefore a natural testing tool for the thesis. The Company understands the results quicker when using a known reference point easily understandable to them.

The initial program installed in the Crestron QM-RMC was a simple echo program. It sends back the exact string of characters it has received upon finding a carriage return (CR) character. The Crestron QM-RMC is programmed using Crestron SIMPL Windows software.[26] Crestron’s software is only available to their partners and is highly proprietary. The use of the necessary Crestron software was enabled by the Company for the duration of the project.

4.2 g++

g++ is a C++ compiler that comes with the GNU Compiler Collection (GCC).[27] It is preinstalled in both Raspbian and Ubuntu. g++ is widely used as a C++ compiler on Linux, macOS and Windows operating systems. Given its wide use it is a sensible compiler choice in this development, where the Company is just getting into the world of well-known programming languages and operating systems.

The programs created and tested on the Ubuntu distribution, used as the primary development environment, work on the Raspbian if compiled on the Raspberry Pi. Computer programs are compiled to work on the processor architecture they are to be run on. The Intel i5 processor on the HP EliteBook 850 G1, to which the Ubuntu distribution is installed, belongs to the processor instruction set architectural family known as “x86-64”.[28] All executable programs compiled using the g++ on the Ubuntu are therefore compatible with x86-64 processors and generally to be assumed incompatible with processors implementing a different instruction set. The instruction set architecture on the processor of the Raspberry Pi 3 Model B belongs to the “ARM” architecture.[29] This requires the programs to be compiled separately for the tests on the Raspberry Pi.

4.3 Documentation and backups

Documentation and backups on the project are saved to a Google Drive folder, which was shared with the Company.[30] This enabled the important documentation on how to implement the solution with the necessary device configuration steps and the necessary source code to be readily available to the customer upon the completion of the project. It also functioned as a backup platform during the project with more than one person having access to the folder, bringing critical safety to the development of the solution.

This method also provided a readily expandable and continuable collaboration environment for future development. Other popular collaboration solutions, such as GitHub, were also explored, but these would not have been easily adapted by the Company at this point. Google Drive was selected because it is a platform the Company is accustomed to.[31, 30]

5 Implementation

The core implementation of the solution was divided into three distinct phases, with one leading to the other.

1. Create or find a controllable serial communication response device to test the serial communication with.
2. Establish raw serial communication between the response device and the Raspberry Pi.
3. Create the real-time RS-232 communication program for the Raspberry Pi using C++.

The first part had already been decided upon during the development environment and tools part of the project, the Crestron QM-RMC.

5.1 Controllable response device, Crestron QM-RMC

The controllable response device for serial communication is needed to verify the functionality of the serial communication in a guaranteed way. Crestron provides tools with which to maintain and monitor their devices. The primary monitoring and maintenance tool for Crestron control processors, such as the QM-RMC, is Crestron Toolbox.[32]

Crestron Toolbox comes with its own debugger, SIMPL Debugger, for debugging programs created with Crestron SIMPL Windows, the program used for programming Crestron control systems.[32, 26] SIMPL Debugger allows for the monitoring and debugging of a user-created program running on a Crestron control processor. SIMPL Debugger shows all the program's internal data joins and their current states, including serial communication.

Crestron uses a concept called "joins" in its programming environment. Joins can be thought of as data buses or variables. These joins are manipulated and used to transfer data between modules and logical symbols within a Crestron SIMPL Windows program. The external serial communication goes through a symbol called "Serial Driver" in Crestron SIMPL Windows and can be accessed using "serial joins". Serial join activity can be monitored in Crestron Toolbox's SIMPL Debugger in either ASCII, hexadecimal or mixed representation. Joins are connected to each other using unique join names, such as "to_raspberry" and "from_raspberry", as seen in Figure 1.

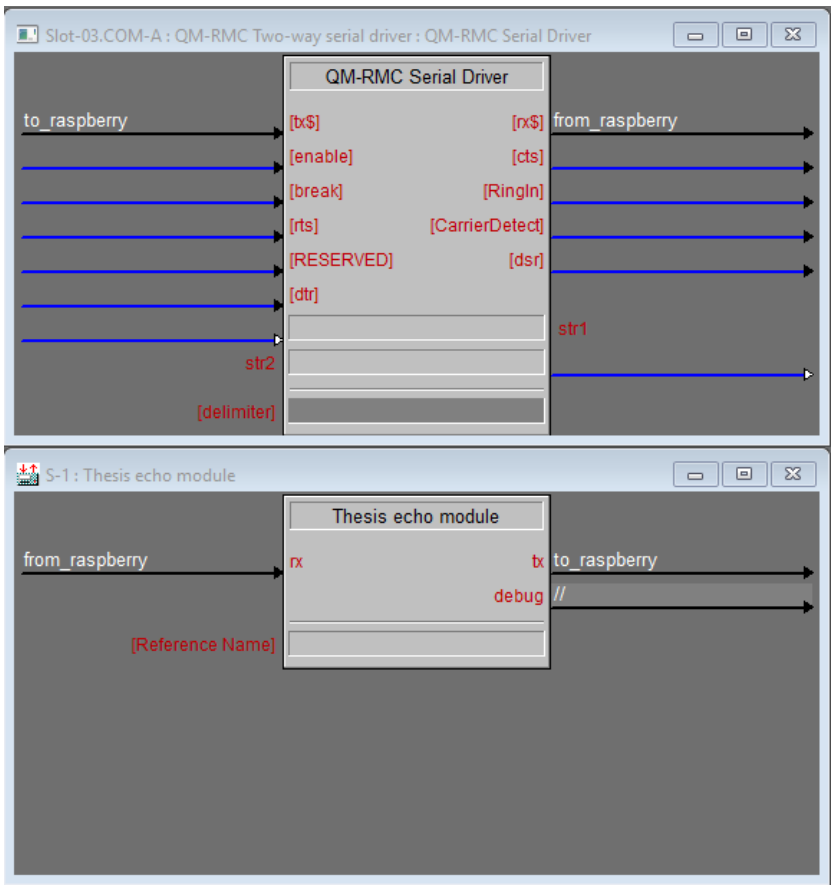

Figure 1. Crestron QM-RMC serial data echo.

Figure 1 shows the contents of the initial SIMPL Windows echo program used to verify serial communication with the Raspberry Pi. The black colored lines in Figure 1 represent “serial joins” and blue lines are “digital joins”. Digital joins have only two possible states, on or off, 1 or 0; hence the term digital join. Digital joins are not used in the initial echo program.

The echo program, created with Crestron SIMPL Windows, is installed to the Crestron QM-RMC using Crestron Toolbox. The functionality of the echo program is tested with a serial communication tool, such as PuTTY on Windows.[33] This test is done on Windows for the simplicity of testing. Crestron’s software is only available on Windows, and the monitoring of both PuTTY and Crestron Toolbox SIMPL Debugger need to be done at the same time.

The data stream received on the RS-232 port of the Crestron QM-RMC is directed to the input of the echo processing module “Thesis echo module” using the join named “from_raspberry”. The information is processed for CR or LF + CR characters, which delimit lines in text files and terminal applications, and converts a CR to a LF + CR combination for compatibility with the Windows operating system. All other characters are

passed through unmodified. The processed data stream is then sent back out from the Crestron QM-RMC's RS-232 port using the join "to_raspberry".

A USB – RS-232 adapter is used between the computer and Crestron QM-RMC to establish serial communication between the two devices. The RS-232 communication specifications set on the Crestron QM-RMC are as follows:

- Baud rate: 115200
- Data bits: 8
- Stop bits: 1
- Parity: None
- Handshaking (HW): None
- Handshaking (SW): None

The USB – RS-232 adapter connected to the Windows computer is also configured to use the same settings. These settings can be set from Windows Device Manager and partially from PuTTY. Once the settings are set, the communication with Crestron QM-RMC's echo program is verified. Figure 2 shows the results of the completed echo program using PuTTY.

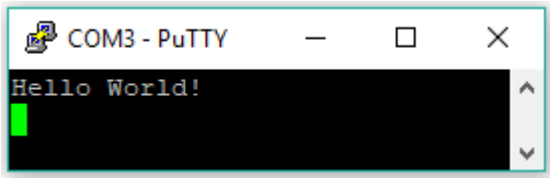

Figure 2. Crestron QM-RMC echo program test with PuTTY.

Figure 3 shows the processing of the same text instance inside Crestron QM-RMC. The figure shows the incoming text from PuTTY in the "from_raspberry" signal and the text sent back to PuTTY is in the signal "to_raspberry", which in return is displayed on the terminal window in PuTTY. The moment a character is typed in PuTTY, it is sent to the Crestron QM-RMC, which returns the sent character back to PuTTY and is then printed on the terminal screen of PuTTY. PuTTY sends the "Enter key" as CR, which the QM-RMC returns as CR + LF. If the Crestron QM-RMC's echo program did not work, no text would be visible in the terminal window of PuTTY, Figure 2.

	Signal	Value	Time
ABC	from_raspberry	H	0 ms
ABC	to_raspberry	H	16 ms
ABC	from_raspberry	e	297 ms
ABC	to_raspberry	e	313 ms
ABC	from_raspberry	l	407 ms
ABC	to_raspberry	l	422 ms
ABC	from_raspberry	l	532 ms
ABC	to_raspberry	l	547 ms
ABC	from_raspberry	o	797 ms
ABC	to_raspberry	o	829 ms
ABC	from_raspberry		00:00:01.579
ABC	to_raspberry		00:00:01.594
ABC	from_raspberry	W	00:00:01.969
ABC	to_raspberry	W	00:00:01.985
ABC	from_raspberry	o	00:00:02.125
ABC	to_raspberry	o	00:00:02.157
ABC	from_raspberry	r	00:00:02.219
ABC	to_raspberry	r	00:00:02.250
ABC	from_raspberry	l	00:00:02.344
ABC	to_raspberry	l	00:00:02.375
ABC	from_raspberry	d	00:00:02.438
ABC	to_raspberry	d	00:00:02.469
ABC	from_raspberry	!	00:00:03.110
ABC	to_raspberry	!	00:00:03.141
ABC	from_raspberry	\x0D	00:00:04.016
ABC	to_raspberry	\x0A\x0D	00:00:04.079

Figure 3. Creston QM-RMC echo program inside the Crestron QM-RMC using Crestron Toolbox's SIMPL Debugger.

5.2 Establishing communication between Raspberry Pi and Crestron QM-RMC

The key to using serial communication, with the Raspbian operating system on the Raspberry Pi, is understanding the concept of "device files". The so called device files are found in the /dev directory of the file system.[34, 35] The introductory paragraphs to device files in the Opensource.com article "Managing devices in Linux" summarizes the concept well:

> Device files are also known as device special files. Device files are employed to provide the operating system and users an interface to the devices that they represent. All Linux device files are located in the /dev directory, which is an integral part of the root (/) filesystem because these device files must be available to the operating system during the boot process.

> One of the most important things to remember about these device files is that they are most definitely not device drivers. They are more accurately described as portals to the device drivers. Data is passed from an application or the operating system to the device file which then passes it to the device driver which then sends it to the physical device. The reverse data path is also used, from the physical device through the device driver, the device file, and then to an application or another device.[35]

The device file feature in Linux makes the implementation of the C++ program communicating with the serial port as easy as reading to and from a text file. Before the implementation of the C++ program can be done, a thorough understanding of how the device file of the USB – RS-232 adapter behaves and is found in the /dev directory is needed.

The easiest way of finding the USB – RS-232 adapter's device file is by looking for it in the kernel ring buffer, which has a feature of showing messages related to hardware.[36] This feature allows any change in the hardware configuration to show up when reading the kernel ring buffer, which is done using the command line tool named "dmesg".[37]

Finding the device file using dmesg starts by attaching the USB – RS-232 adapter to one of the Raspberry Pi's USB ports after the Raspberry Pi has booted up properly and no other hardware changes occur. The dmesg only shows the latest messages and therefore it is important to not do any other hardware changes after connecting the USB – RS-232 adapter.[38] Figure 4 shows the results of dmesg right after connecting the USB – RS-232 adapter.

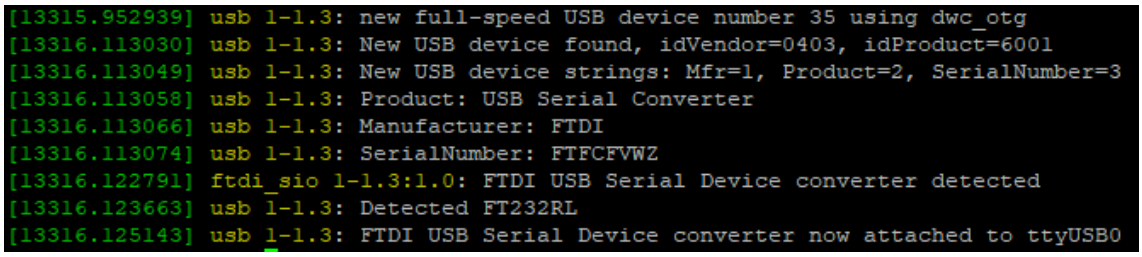

Figure 4. Last lines of the dmesg command showing the detection of the USB – RS-232 adapter.

The last line in the output in Figure 4 gives the name of the device file associated with the device: ttyUSB0. The existence of this device file can be confirmed by listing the device files found in the /dev directory using the command line tool "ls".

If for some reason the attaching of the USB – RS-232 adapter cannot be attached to the Raspberry Pi after fully powering the Raspberry Pi on, it is possible to find it using log files which the operating system creates during boot up. If the USB – RS-232 adapter was connected to the Raspberry Pi before powering it on it is possible to search for the

adapter's device file from the /var/log/messages file, which contains the same information and more, as the kernel ring buffer read using dmesg.[38]

For actual communication to occur between the Raspberry Pi and the Crestron QM-RMC, the USB – RS-232 adapter attached to the Raspberry Pi needs to be configured to communicate with the correct serial settings, as listed in section 5.1. This can be done using the preinstalled command line tool "stty", which is found on the Raspbian and Ubuntu Linux distributions.

stty is part of the GNU Core Utilities software package installed on many of the Linux distributions.[39] The command "stty -F /dev/ttyUSB0 -a" lists all the current settings of the ttyUSB0 device, the USB – RS-232 adapter. Figure 5 shows the output of the command with ttyUSB0 having system default settings. There is a total of 74 settings which can be modified. Understanding them is important in order to fully utilize the serial device as intended.

```
pi@raspberrypi:~ $ stty -F /dev/ttyUSB0 -a
speed 9600 baud; rows 0; columns 0; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>;
swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V;
discard = ^O; min = 1; time = 0;
-parenb -parodd -cmspar cs8 hupcl -cstopb cread clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc -ixany
-imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt echoctl echoke
-flusho -extproc
```

Figure 5. Default settings of ttyUSB0, printed using the stty utility.

The full description of the 74 settings is found from the utility's manual or "man" page, which is displayed by the terminal program by typing the command "man stty". The commands needed to get ttyUSB0 set for proper communications with the Crestron QM-RMC are shown in Figure 6, along with a printout of the changed settings.

```
pi@raspberrypi:~ $ stty -F /dev/ttyUSB0 raw
pi@raspberrypi:~ $ stty -F /dev/ttyUSB0 -echo
pi@raspberrypi:~ $ stty -F /dev/ttyUSB0 115200
pi@raspberrypi:~ $ stty -F /dev/ttyUSB0 -a
speed 115200 baud; rows 0; columns 0; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>; swtch = <undef>; start =
susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; discard = ^O; min = 1; time = 0;
-parenb -parodd -cmspar cs8 hupcl -cstopb cread clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr -icrnl -ixon -ixoff -iuclc -ixany -imaxbel -iutf8
-opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
-isig -icanon iexten -echo echoe echok -echonl -noflsh -xcase -tostop -echoprt echoctl echoke -flusho -extproc
```

Figure 6. Crestron QM-RMC echo program's compatible serial settings on the Raspberry Pi.

Establishing basic serial communication between the Raspberry Pi and Crestron QM-RMC can be done using simple utilities like "echo" and "cat" from the terminal emulator

program on the Raspberry Pi. echo is a simple program for displaying text on the standard output of the operating system.[40] The echo program in the establishing of raw communication is used to send strings to the Crestron QM-RMC. echo in itself is not enough to send the string to the Crestron QM-RMC as echo prints the string to standard output, the terminal screen in this case.[40]

The Bourne Again Shell (bash), used by the terminal emulator program to communicate with the operating system on the Rapsberry Pi, has a feature called redirection.[41] Redirection allows the input or output of a program to be redirected to, e.g. a file.[41] Adding this feature when calling the echo program allows the output of the echo program to be redirected and written to the file specified. In this case the output of echo is redirected to the device file ttyUSB0, the device file for the USB – RS-232 adapter. Figure 7 shows how this is done using the append operator “>>”. The write operator “>” would also work in this case, but out of safe habits of not overwriting a file’s contents the append operator is used in the example.

```
pi@raspberrypi:~ $ echo Hello World! >> /dev/ttyUSB0
```

Figure 7. Sending strings to the Crestron QM-RMC from the Raspberry Pi using echo and redirection.

With the devices connected to each other and the serial ports configured correctly, the Crestron QM-RMC receives the string “Hello World!” from the Raspberry Pi and echoes it back, as shown in Figure 8. The echoed response can be seen on the Raspberry Pi by having a second terminal session open at the same time, with the “/dev/ttyUSB0” device file opened using the cat utility, as shown in Figure 9.

	Signal	Value	Time
ABC	from_raspberry	Hello World!\x0A	0 ms
ABC	to_raspberry	Hello World!\x0A	31 ms

Figure 8. Crestron QM-RMC’s serial communication during the raw Raspberry Pi communication test.

```
pi@raspberrypi:~ $ cat /dev/ttyUSB0
Hello World!
```

Figure 9. Raspberry Pi receiving Crestron QM-RMC’s echo response using the cat utility

5.3 The C++ serial communication program on Raspberry Pi

Since a fully functional and working application for configuring the serial devices on the Raspbian operating system already existed, it was decided to utilize that for configuring the serial device from the new C++ program. The first task in the actual programming of the prototype C++ program was to establish a way with which to communicate with the stty utility from the C++ program. This feature, of communicating with the bash from the newly created C++ program, played a crucial role in other following parts of the implementation as well.

The answer for the question of how to communicate with the stty utility was found from a technical blogpost on www.jeremymorgan.com.[42] The code snippet from the website can be seen below:

```
string GetStdoutFromCommand(string cmd) {

string data;
FILE * stream;
const int max_buffer = 256;
char buffer[max_buffer];
cmd.append(" 2>&1");

stream = popen(cmd.c_str(), "r");
if (stream) {
while (!feof(stream))
if (fgets(buffer, max_buffer, stream) != NULL) data.append(buffer);
pclose(stream);
}
return data;
}
```

Listing 1. Code posted by Jeremy Morgan on how to capture the output of a linux command in C.[42]

The key feature in the solution posted by Jeremy Morgan was use of the popen() and plcose() functions, which are part of the GNU C Library (glibc) that come with GCC and the g++ complier.[43] The introduction sentences of using popen() and pclose() describe the functionality of the function well:

> A common use of pipes is to send data to or receive data from a program being run as a subprocess. One way of doing this is by using a combination of pipe (to create the pipe), fork (to create the subprocess), dup2 (to force the subprocess to use the pipe as its standard input or output channel), and exec (to execute the new program). Or, you can use popen and pclose.
>
> The advantage of using popen and pclose is that the interface is much simpler and easier to use. But it doesn't offer as much flexibility as using the low-level functions directly.[43]

The same manual page, from which the above quote is from, confirms the example of Jeremy Morgan's code snippet with its own similar example of how to utilize the two functions.

An implementation, based on Jeremy Morgan's blogpost and the GNU C Library Manual, was created for the purpose of communicating with bash. A set of useful functions using strings and vectors were stored in a library called "bashcmd.h". The storing of the functions to library is highly useful and recommended, as these are very fundamental functions that allow for the utilization of pre-existing utilities in bash. The bashcmd.h library was used in every part of the implementation to do tasks such as checking the /dev directory's files, checking the running processes on the operating system and communicating with the stty utility. The base function created for communicating with bash was called bashCommandString().

The examples on using stty in Figures 5 and 6 show that the utility needs one important parameter in its invocation, the serial device's device file's location in the file system. The results of Figure 4 gave the device name ttyUSB0. Knowing that such device files are found in the /dev directory and confirming the existence of ttyUSB0 it with the ls utility, the full path or location of the device file is therefore /dev/ttyUSB0. This full path needs to be sent to the stty utility for it to find the device file and check its serial communication settings.

The first step the C++ program should therefore do is check the existence of the desired device file, in this ttyUSB0, before sending it as a parameter to the stty utility. No harm is done by invoking stty with a device file that does not exist in the file system. It is in poor programming taste to send erroneous information if it can be checked before doing so. Therefore, a file existence check function was created to the bashcmd.h library which utilizes the bash communication function created earlier. The file existence check function can be seen below.

```
ssize_t bashDoesFileExist(std::string const *file_name)
{
     ssize_t result = -1;
     std::string command = "[ -e " + *file_name + "\
 ] && echo \"yes\" || echo \"no\"";
     std::string response;

     if(bashCommandString(&command, &response) > -1)
     {
          if(response == "yes\n")
          {
               result = true;
```

```
            }
            else if(response == "no\n")
            {
                  result = false;
            }
      }

      return result;
}
```

Listing 2. File existence checking function for the Raspberry Pi running Raspbian Lite.

The function uses the self-written bash communication function bashCommandString() to send commands and receive responses. The bash command sent in the function's if-statement when searching for the existence of the /dev/ttyUSB0 device file is "[-e /dev/ttyUSB0] && echo yes || echo no". This tells bash to check if the given file exists, using the -e option, and makes bash respond to the file check command with either "yes" or "no". Because bash always adds a new line by default to every response it gives the answer check is done with the "\n" character appended to the expected responses. The function then returns "true" if the file exists, "false" if it does not exist and -1 if the bash-CommandString() function failed for some unknown reason.

The second step in using the stty utility, after first confirming the existence of the device file, is to configure the serial device's device file to the correct communication parameters shown in Figure 6. The procedure to configure the ttyUSB0 device file for appropriate communication with the Crestron QM-RMC using the serial settings listed in section 5.1 is the following:

1. Set the communication to "raw"
2. Disable "echo"
3. Set the baud rate to 115200
4. Check that the settings are as they should be

The event of modifying the settings and checking them can be seen in Figure 6, which was used as the point of reference for creating the configuration function for C++. The first command sent to bash in regards to the stty utility is "stty -F /dev/ttyUSB0 raw". The man page of stty describes raw as a combination setting, which sets the following settings in stty:

> same as -ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr -icrnl -ixon -ixoff -iuclc -ixany -imaxbel -opost -isig -icanon -xcase min 1 time 0[44]

Every dash character (-) means a setting is turned off and a setting name without the dash character prepended means that setting is turned on. The next setting to send the stty utility is to turn off the automatic character echo setting. If the setting is left on, the serial device on the computer will echo each received byte back to the sender. This setting is not useful when controlling AV devices, because the AV devices usually don't expect the device controlling them to send any responses back to them. The full command in bash is "stty -F /dev/ttyUSB0 -echo". In this command the dash character tells the stty utility to turn off input character echoing.[44]

The third setting that needs to be set is the baud rate. This is done by simply sending the new wanted baud rate at the end of the command in the following way: "stty -F /dev/ttyUSB0 115200". The command sets the baud rate to the wanted 115200. After all required settings are set, they need to be checked and verified. There is no point in starting communication with the Crestron QM-RMC if the settings are not correct on the Raspberry Pi. All the four steps of using the stty utility were programmed to their own functions with all of them utilizing the previously created bashCommandString() function.

5.3.1 Using the device file with a C++ program

With the serial device fully configured for communication with the Crestron QM-RMC, the next step was to experiment on how to use the device files to send and receive data. Earlier research showed, that it should be as simple as reading and writing text to a text file. This feature was tested with two separate programs, one for sending data and one for receiving. The two actions were first tested separately in order to gain proper understanding on the nature of the two different actions.

C++ has a standard library header called fstream. fstream contains a class of the same name, which is used for operating on files. As the device file can be opened as any other file on the file system, fstream works suitably for sending data out of the USB – RS-232 adapter to the Crestron QM-RMC. The code snippet below shows how simple it is to send text strings out of the Raspberry Pi using C++.

```
#include <fstream>

int main()
{
      std::fstream file;
      file.open("/dev/ttyUSB0", std::fstream::out | std::fstream::trunc);

      if(file.is_open())
```

```
        {
                file << "Hello World!";
                file.close();
        }

        return 0;
}
```

Listing 3. C++ code to send data out the Raspberry Pi using the device file associated with the connected USB – RS-232 adapter.

The device file is first opened, just like any other file by opening a file stream to the file. Then the "<<" operator is used to send data to the file stream, which in this case is the device file, which is translated by the operating system and its drivers to the USB – RS-232 adapter and eventually the text "Hello World!" is received by the Crestron QM-RMC. After "Hello World!" has been sent to the file stream, the file stream is closed.

Receiving data from the Crestron QM-RMC proved to be more challenging than anticipated after the ease of sending data. It is easier to send information out of a system, because all it needs is to be in the correct format and no further processing is needed. Much more processing is needed when receiving data. An important question was raised before testing how to receive data with a C++ program on Raspberry Pi: how to manage asynchronous data?

Synchronous communication is the easiest communication method to implement, when controlling external devices. The command protocols of most projectors and displays send a response, indicating if a command was received properly or not, immediately after receiving a valid command. In practice. the easiest way to implement this is to first send the command, wait for the expected time it takes for the external device to send its automatic response and finally check if the device responded within the expected time.

However, it is not always possible to communicate in a synchronous fashion with external devices. The external devices may send information on their own without being asked for, and this information may be important. It is also possible that the latency of the externally controlled device's response varies. Such a case would also need the ability to handle the reception of data outside the expected response time.

The abstract solution to the question above is to design the reception of the data from the external device asynchronously, even if the communication method of some devices may be synchronous by nature. Synchronous communication behavior can be taken into account using asynchronous data capture, but not the opposite way around.

A key feature in receiving asynchronous data is, that it needs to be active all the time. If this is done carelessly, by having the program wait for something to come in, the listening to possible incoming data would block the execution of the whole program until something is received. This means that nothing else, such as user interaction with the program, can be processed until something is received. The code snippet below shows and example of receiving asynchronous data while blocking the execution of the rest of the program.

```
#include <fstream>
#include <string>
#include <iostream>

int main()
{
      std::fstream::file;
      file.open("/dev/ttyUSB0", std::fstream::in);

      if(file.is_open())
      {
            std::string from_device;
            file >> from_device;
            file.close();
      }

      std::cout << "Hello user!\n";
      return 0;
}
```

Listing 4. Simple C++ code for receiving data on the Raspberry Pi from the device file associated with the connected USB – RS-232 adapter.

In the example above the text “Hello user!” would not be printed on the screen until something was sent to the device associated with the device file /dev/ttyUSB0. The data sent from the external device is received, but all other execution is blocked while doing so. This is not a working solution for software that needs to do user interaction or some other function while controlling external devices at the same time. The way to create a non-blocking asynchronous data receiver is to implement multitasking. Two multitasking techniques were investigated for this purpose, of which one was chosen in the implementation.

The opening sentences on multithreading in the Wikipedia page on threads in computing summarizes multithreading well:

> Multithreading is mainly found in multitasking operating systems. Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of one process. These threads share the process's resources, but are able to execute independently. The threaded programming model

> provides developers with a useful abstraction of concurrent execution. Multithreading can also be applied to one process to enable parallel execution on a multiprocessing system. [45]

Multithreading is often referred to as being quick, resource light and easy to implement and is therefore recommended by many programmers in programming related forums. The limitation of multithreading programming can be seen in the quotation above in the end of the second sentence, “that allows multiple threads to exist within the context of one process.” This part means that the seemingly simultaneous actions the separate threads do, all exist within one process, thus limiting its portability. One task can not be put on a separate processor or device from the other, because they all exist within the same program and its shared memory space on one computer. The alternative multitasking approach to multithreading is multiprocessing.

> Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system.[46]

The quote above, taken from the Wikipedia page on multiprocessing, summarizes the term in an abstract way and can be interpreted from many standpoints. From a hardware standpoint this would mean that multiple physical processing units exist within a single processing die, also known as multicore processors, or multiple processors within a larger computer system. From a software engineering standpoint this can be thought of using multiple processes within the operating system to handle separate tasks or processes, as they are commonly called.

The advantage of multiprocessing software in comparison to multithreading is, that the separate processes are run independently from different executables and can be utilized more dynamically. If they run on the same computer system and that computer system has a multicore processor, both processes may end up running on separate processor cores, giving them more speed to run efficiently.

One may design a support processes to do certain tasks in the background, that other processes utilize in different manners. Depending on the implementation the processes may even be on different computer systems and communicate with each other through means of e.g. networking protocols. Multiprocessing gives more freedom of implementation techniques, even if it were slower to start and more complex to implement.

The possibilities of multiprocessing, especially the possibility of reusing the same program on separate computers, was something that the Company saw as a valuable feature. The receiving of data from the external serial device was therefore chosen to be done in the multiprocessing method.

A separate program was created with the task of reading data from /dev/ttyUSB0 and storing the received data in a buffer file. The main program, which handles user interaction and processing the sent and received data, fetches the received data from the buffer file. This way anything sent by the external device is always caught and may be processed by the main program as need be. The main program is not blocked for lengthy times that would be detectable to a user, because fetching readily available data from a file and closing it happens extremely fast. The program in charge of receiving data from the external serial device is shown below.

```
#include <fstream>

int main()
{
    std::fstream device;
    device.open(/dev/ttyUSB0, std::fstream::in);
    int const buf_size = 256;

    if(device.is_open())
    {
        while(true)
        {
            char c = device.get();
            std::fstream buffer;
            buffer.open(/home/pi/ttyUSB0.buf, std::fstream::out |
                        std::fstream::app);

            if(buffer.is_open)
            {
                buffer << c;
                buffer.close();
            }
        }

        device.close();
    }

    return 0;
}
```

Listing 5. The C++ code for the program in charge of receiving data from an external device attached to the USB – RS-232 adapter on the Raspberry Pi.

The program opens the device file and reads the incoming data stream one character or byte at a time. The method get() is a blocking function. When using it with device files, it waits until there is something to read and then returns the data. The device file is completely empty when it is opened, as it is not a standard file containing some form of data,

but is in fact a data stream. The file does not raise the end-of-file (EOF) flag either, because there is no real file and therefore no real end. This allows for the get() method to wait until something has come into the data stream and immediately extracts it, when it finds the first character.

The received character is appended to the buffer file /home/pi/ttyUSB0.buf, from which the main program reads the received data. There is no fear of losing new data when the data is stored to the buffer file, because the opened input stream contains within itself a file stream buffer while it is active. New incoming data is stored in the random access memory (RAM) until they are recovered by get() to the buffer file, or the device file is closed.[47, 48]

With both sending and receiving accomplished separately, the next task was to combine the two separate actions into one user interactive program. The abstract outline for the user interactive program is as follows:

1. Prompt the user for a text string to be sent to the Crestron QM-RMC.
2. Wait a short while to allow the Crestron QM-RMC to process the data it receives and echo it back.
3. Print the data received from the Crestron QM-RMC.
4. Repeat steps 1 to 3 until the user types the word “exit”.

The aim of the user interactive program was to demonstrate to the Company how serial communication works with the Raspberry Pi 3 Model B and what are the base requirements upon which further applications may be built. The echo program on the Crestron QM-RMC, introduced in section 5.1, remains unchanged for this purpose. The short user interactive program to communicate with the Crestron QM-RMC is shown below.

```
#include <iostream>
#include <string>
#include <fstream>
#include <thread>
#include <chrono>

int main()
{
    std::cout << "RS-232 communication with Crestron QM-RMC\n";

    while(true)
    {
        std::string user_input;
        std::cout << "Enter somthing to send to Crestron QM-RMC: ";
        std::getline(std::cin, user_input);

        if(user_input == "exit")
```

```
        {
            std::cout << "Bye bye!\n";
            break;
        }
        else
        {
            std::fstream qm_rmc;
            qm_rmc.open("/dev/ttyUSB0", std::fstream::out |
                   std::fstream::trunc);

            if(qm_rmc.is_open())
            {
                qm_rmc << user_input;
                qm_rmc.close();
            }

            std::this_thread::sleep_for(std::chrono::milliseconds(300));
            std::fstream buffer;
            std::string from_qm_rmc;
            buffer.open("/home/jonah/Thesis/Code/ttyUSB0_listener.buf",
                   std::fstream::in);

            if(buffer.is_open())
            {
                std::getline(buffer, from_qm_rmc);
                buffer.close();
            }

            buffer.open("/home/jonah/Thesis/Code/ttyUSB0_listener.buf",
                   std::fstream::out | std::fstream::trunc);

            if(buffer.is_open())
            {
                buffer << "";
                buffer.close();
            }

            std::cout << "Crestron QM-RMC responded: " << from_qm_rmc <<
                 std::endl;
        }
    }

    return 0;
}
```

Listing 6. The user interactive program written in C++.

For the above program to work, two other things need to be done prior to executing the program. The serial device on the Raspberry Pi needs to be configured to work with the correct parameters, as shown in Figure 6, and the program in charge of listening to the incoming data on the serial device on the Raspberry Pi needs to be running. After doing the above-mentioned prerequisites the user interaction program can be run successfully. Figure 10 shows the output of the user interactive program communicating with the Crestron QM-RMC. Figure 11 shows the same instance on the Crestron QM-RMC.

```
RS-232 communication with Crestron QM-RMC
Enter something to send to Crestron QM-RMC: hi qm-rmc. how are you doing?
Crestron QM-RMC responded: hi qm-rmc. how are you doing?
Enter something to send to Crestron QM-RMC: exit
Bye bye!
```

Figure 10. Output of the user interactive program on the Raspberry Pi 3 Model B.

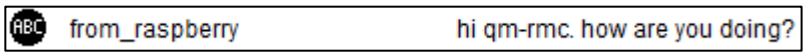

Figure 11. RS-232 communication coming in to the Crestron QM-RMC and sent immediately back out.

6 Results and future development ideas

After the completion and demonstration of the user interactive program, the program was modified to control a Panasonic video projector. Instead of sending raw user input, the program sent commands to power on and power off the projector. The command protocol for controlling the projector can be found from Panasonic's official website documentation. Successful projector control received approval from the Company and the task was accepted as completed. The Company was satisfied with the overall price of a single Raspberry Pi 3 Model B kit and the fact that standard C++ libraries were enough to complete the task.

The next step in development could be the creation of a class for controlling the serial device communications. The class would allow for the creating of the serial port or device as an object, making it easy to reference and control in a larger application. The class should contain the necessary device file checking functions and stty configuration functions to initialize the object for real communication. It should also contain some means of controlling the data receiving program so that it would not have to running before the application in need of it.

One way of implementing the control of the data receiving program would be to create the data receiving program with forking. This is done with the fork() function found in unistd.h in Linux and other POSIX systems.[49, 50] The fork() function creates an exact copy of the program executable on runtime, but with a new process ID (PID). The process which is started with fork() is called a child process and the process calling fork() is called a parent process.[50]

By using the return values of fork(), the program can be coded so that it knows when it is a child process and when it is not. The program should be programmed to only call fork() if it is a parent process and to skip the calling of fork() and do the actual other code, such as the data receiving in this case, when it is a child process. The PID of the child process can be received by the parent process upon calling fork() and this value can be stored in a separate file. This file can then be used to find the PID of the child process running in the background. The PID can then later be used to stop the child process by doing a bash "kill" command with the PID.

The class for controlling a serial device on a Raspberry Pi, combined with the control of the data receiving program from within that class, are highly recommended updates to

the functionality and reusability of the base code produced by this thesis. The updates would allow for any program using that class to become a multiprocessing software application. The software in the future could be extended to divide parts of the application across different computers on a network, simply by modifying parts of the class and creating new helper programs like the data receiving program.

References

1 Cosma Rohilla Shalizi. Top-Down Design (Introduction to Statistical Computing). [Online]. Available from: http://bactra.org/weblog/950.html [Accessed 6 July 2018]

2 Apple Inc. macOS. [Online]. Available from: https://www.apple.com/macos/what-is/ [Accessed 29 March 2018]

3 Raspberry Pi Foundation. Raspberry Pi 3 Model B. [Online]. Available from: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ [Accessed 29 March 2018]

4 Microsoft Corporation. Windows IoT. [Online]. Available from: https://developer.microsoft.com/en-us/windows/iot/getstarted [Accessed 29 March 2018]

5 Raspberry Pi Foundation. Raspbian. [Online]. Available from: https://www.raspberrypi.org/downloads/raspbian/ [Accessed 29 March 2018]

6 Tutorials Point India Private Limited. Interfaces in C++ (Abstract Classes). [Online]. Available from: https://www.tutorialspoint.com/cplusplus/cpp_interfaces.htm [Accessed 29 November 2018]

7 Python Software Foundation. General Python FAQ. [Online]. Available from: https://docs.python.org/3/faq/general.html [Accessed 31 March 2018]

8 Python Software Foundation. Terms and conditions for accessing or otherwise using Python. [Online]. Available from: https://docs.python.org/3/license.html#terms-and-conditions-for-accessing-or-otherwise-using-python [Accessed 31 March 2018]

9 Python Software Foundation. Python Source Releases. [Online] Available from: https://www.python.org/downloads/source/ [Accessed 31 March 2018]

10 Python Software Foundation. Concealing (“Protecting”) Source Code. [Online] Available from: https://wiki.python.org/moin/Asking%20for%20Help/How%20do%20you%20protect%20Python%20source%20code%3F [Accessed 3 April 2018]

11 SourceForge.net. decompyle – Python Decompiler. [Online]. Available from: https://sourceforge.net/projects/decompyle/ [Accessed 3 April 2018]

12 PyInstaller Development Team. Welcome to PyInstaller official website. [Online]. Available from: http://www.pyinstaller.org/ [Accessed 3 April 2018]

13 GitHub, Inc. PyInstaller Supported Packages. [Online]. Available from: https://github.com/pyinstaller/pyinstaller/wiki/Supported-Packages [Accessed 3 April 2018]

14 cplusplus.com. A Brief Description. [Online]. Available from: http://www.cplusplus.com/info/description/ [Accessed 3 April 2018]

15 Wikipedia. RS-232. [Online]. Available from: https://en.wikipedia.org/wiki/RS-232 [Accessed 22 November 2018]

16 Axelson J. Serial Port Complete. 2nd edition. Madison, WI, USA: Lakeview Research LLC; 2007. p. 2

17 Wikipedia. USB. [Online]. Available from: https://en.wikipedia.org/wiki/USB [Accessed 22 November 2018]

18 Wikipedia. Ethernet over twisted pair. [Online]. Available from: https://en.wikipedia.org/wiki/Ethernet_over_twisted_pair [Accessed 22 November 2018]

19 Crestron Electronics. DM-NVX-350. [Online]. Available from: https://www.crestron.com/en-US/Products/Video/DigitalMedia-Streaming-Solutions/Encoder-Decoders/DM-NVX-350 [Accessed 22 November 2018]

20 Canonical Ltd. ubuntu. [Online]. Available from: https://www.ubuntu.com/ [Accessed 18 November 2018]

21 HP Development Company, L.P. HP EliteBook 850 G1 -kannettava. [Online]. Available from: https://support.hp.com/fi-fi/drivers/selfservice/hp-elitebook-850-g1-notebook-pc/5405368 [Accessed 18 November 2018]

22 Canonical Ltd. Debian is the rock on which Ubuntu is built. [Online]. Available from: https://www.ubuntu.com/community/debian [Accessed 18 November 2018]

23 Raspbian. Welcome to Raspbian. [Online]. Available from: http://raspbian.org/RaspbianAbout [Accessed 18 November 2018]

24 GitHub, Inc. atom.io. [Online]. Available from: https://atom.io/ [Accessed 18 November 2018]

25 Crestron Electronics. QM-RMC. [Online]. Available from: https://www.crestron.com/en-US/Products/Control-Hardware-Software/Hardware/Control-Modules/QM-RMC [Accessed 3 July 2018]

26 Crestron Electronics. SW-SIMPL. [Online]. Available from: https://www.crestron.com/en-US/Products/Control-Hardware-Software/Software/Control-System-Software/SW-SIMPL [Accessed 6 July 2018]

27 Free Software Foundation. GCC. [Online]. Available from: https://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/G_002b_002b-and-GCC.html [Accessed 18 November 2018]

28 Wikipedia. x86-64. [Online]. Available from: https://en.wikipedia.org/wiki/X86-64 [Accessed 18 November 2018]

29 Raspberry Pi Foundation. BCM2837. [Online]. Available from: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md [Accessed 18 November 2018]

30 Google LLC. Google Drive. [Online]. Available from: https://www.google.com/drive/ [Accessed 6 July 2018]

31 GitHub, Inc. GitHub. [Online]. Available from: https://github.com/ [Accessed 6 July 2018]

32 Crestron Electronics. SW-TB. [Online]. Available from: https://www.crestron.com/en-US/Products/Control-Hardware-Software/Software/Development-Software/SW-TB [Accessed 6 July 2018]

33 PuTTY. [Online]. Available from: https://www.putty.org/ [Accessed 6 July 2018]

34 How-To Geek, LLC. What Does “Everything Is a File” Mean in Linux?. [Online]. Available from: https://www.howtogeek.com/117939/htg-explains-what-everything-is-a-file-means-on-linux/ [Accessed 8 July 2018]

35 Opensource.com. Managing devices in Linux. [Online]. Available from: https://opensource.com/article/16/11/managing-devices-linux [Accessed 8 July 2018]

36 LinuxQuestions.org. What Kernel Ring Buffer Contains? [Online]. Available from: https://www.linuxquestions.org/questions/linux-kernel-70/what-kernel-ring-buffer-contains-478645/ [Accessed 8 July 2018]

37 Linux man-pages project. DMESG(1). [Online]. Available from: http://man7.org/linux/man-pages/man1/dmesg.1.html [Accessed 8 July 2018]

38 Stack Exchange Inc. Unix & Linux. What’s the difference of dmesg output and /var/log/messages?. [Online] Available from: https://unix.stackexchange.com/questions/35851/whats-the-difference-of-dmesg-output-and-var-log-messages [Accessed 8 July 2018]

39 Linux man-pages project. STTY(1P). [Online]. Available from: http://man7.org/linux/man-pages/man1/stty.1p.html [Accessed 12 July 2018]

40 Free Software Foundation, Inc. GNU Coreutils. 15.1 echo: Print a line of text. [Online]. Available from: https://www.gnu.org/software/coreutils/manual/html_node/echo-invocation.html#echo-invocation [Accessed 21 July 2018]

41 Free Software Foundation, Inc. GNU. Bash Reference Manual. 3.6 Redirections. [Online]. Available from: https://www.gnu.org/software/bash/manual/html_node/Redirections.html [Accessed 21 July 2018]

42 Jeremy Morgan. How to Capture the Output of a Linux Command in C++. [Online]. Available from: https://www.jeremymorgan.com/tutorials/c-programming/how-to-capture-the-output-of-a-linux-command-in-c/ [Accessed 27 August 2018]

43 Free Software Foundation. GNU. The GNU C Library. 15.2 Pipe to a Subprocess. [Online]. Available from: https://www.gnu.org/software/libc/manual/html_node/Pipe-to-a-Subprocess.html [Accessed 18 November 2018]

44 David MacKenzie. stty(1) – Linux man page. [Online]. Available from: https://linux.die.net/man/1/stty [Accessed 18 November 2018]

45 Wikipedia. Thread (computing). [Online]. Available from: https://en.wikipedia.org/wiki/Thread_(computing) [Accessed 20 November 2018]

46 Wikipedia. Multiprocessing. [Online]. Available from: https://en.wikipedia.org/wiki/Multiprocessing [Accessed 20 November 2018]

47 cplusplus.com. std::fstream. [Online]. Available from: http://www.cplusplus.com/reference/fstream/fstream/ [Accessed 22 November 2018]

48 cplusplus.com. std::filebuf. [Online]. Available from: http://www.cplusplus.com/reference/fstream/filebuf/ [Accessed 22 November 2018]

49 The Open Group. unistd.h – standard symbolic constants and types. [Online]. Available from: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/unistd.h.html [Accessed 22 November 2018]

50 The Open Group. fork – create a new process. [Online]. Available from: http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html# [Accessed 22 November 2018]