Teaching electronics in time of disease - virtual laboratory for Linux in embedded systems

Wojciech M. Zabołotny[a], Michał Kruszewski[a], and Paweł Linczuk[a]

[a]Institute of Electronic Systems, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland

ABSTRACT

The COVID-19 pandemic crisis has significantly hindered the functioning of universities. The necessity to work remotely has disabled access to the hardware laboratories. In the teaching of embedded systems, the possibility of working with real hardware is really important. Fortunately, the embedded computer system itself may be efficiently emulated with QEMU. However, the problem may arise with the emulation of its environment, the I/O devices, especially those providing the user interface. This paper describes the methods to modify and use QEMU to emulate the real hardware used in the laboratories. It also discusses the advantages and limitations of such an approach. The elaborated methodology may be useful also after the pandemic, to support and improve teaching in normal conditions, e.g., by allowing students to perform at home the realistic experiments on emulated hardware.

Keywords: Teaching electronics, Virtual hardware, QEMU, Embedded systems

1. INTRODUCTION

The course “Linux for Embedded Systems” has been taught in the Faculty of Mathematics and Information Science at Warsaw University of Technology since 2014. The course aims to teach the students how the GNU/Linux operating system may be efficiently used in embedded systems. The students may also gain practical skills in building the dedicated Linux system for specific platforms and applications. The course consists of eight 2-hour lectures and 30 hours of practical work in the laboratory, where the students prepare, run, and debug the Linux-based systems with gradually increasing complexity. As the laboratory’s hardware platform was prepared in 2014, the Raspberry Pi version B boards were selected due to their availability, good performance/price ratio, and broad community support. One of the goals was that the students should be able to continue the work with the embedded Linux based designs after the course. The high number of Raspberry Pi based projects available on the Internet support that goal.

Another advantage of Raspberry Pi is that the operating system image is loaded from the SD card. In many embedded systems, the operating system is stored in the soldered FLASH memories. The system’s frequent modifications or incorrect configuration may result in rapid wear and necessity to replace that memory chips. The SD cards are easier to replace and eliminate the risk of "soft-bricking" of the board. In the worst case, they may be reformatted using the SD card reader connected to the PC.

The correct choice of the hardware platform is confirmed by the fact that after six years of teaching the course, all boards are still working correctly.

To imitate the typical embedded system with serial console and limited user interface, the Raspberry Pi has been supplemented with the USB-UART adapter and a simple I/O board with three switches and 4 LEDs connected to GPIO pins. For more sophisticated I/O operations, additional devices may be connected via I^2C or SPI interfaces. More complex devices may also be connected via the USB interface. The architecture of the setup used in the laboratory is shown in Figure 1.

Further author information: (Send correspondence to W.M.Z.)
W.M.Z.: E-mail: wzab@ise.pw.edu.pl, Telephone: +48 22 234 6693

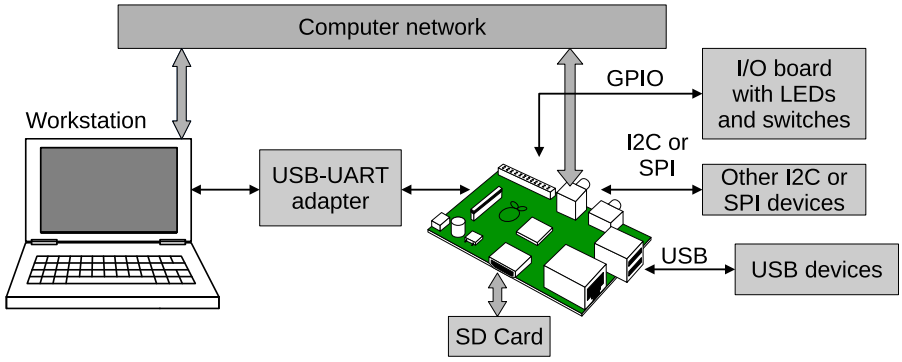

Figure 1: Setup in the standard laboratory.

1.1 Software organization

After the first semester of teaching the course, it was discovered that the standard SD card socket in Raspberry Pi B tends to bend the SD card, and finally destroys them after multiple insertions and removals. Therefore the initial concept that the students should write the generated operating system image after removing the SD card from the Raspberry Pi and placing it into the SD card reader had to be modified.

The standard Raspberry Pi firmware was supplemented with the additional U-Boot bootloader. The boot procedure may be interrupted via the serial console, and the student may enter the command that loads the special "rescue" version of the Linux prepared by tutors. The "rescue" version uses initial ramdisk (initramfs) as the root filesystem. Therefore after booting, the user may arbitrarily modify the SD card, changing the partitions, formatting them, downloading via network and installing the operating system kernel and filesystem. In case if the student destroys the files needed to boot the "rescue" image, he may boot his Raspberry Pi from the "master" SD card, load the "rescue" image, replace the "master" SD card with his card, and restore the original files.

That procedure worked perfectly in later semesters of the course.

1.2 Environments used to generate the Linux image

During the course, the students should be able to create the system optimized for their particular application. The most popular frameworks for that purpose are:

- Yocto[1]
- Buildroot[2]
- OpenWRT[3]

Because the laboratory classes are organized as 3-hour sessions, it was important that the system's building is as fast as possible. The Yocto framework is well supported by hardware vendors like Intel, Arm Holding or Xilinx, and widely used in the industry. Unfortunately, the time of the initial building of the Linux system image for Raspberry Pi in Yocto is unacceptably long - ca. 90 minutes*. The rebuilding of the image, after the modifications, may be significantly faster, but there is always a risk that a student may need to recreate his design from scratch. Therefore Yocto was considered to be not well suited for our laboratory. In the case of Buildroot, the compilation time using the precompiled toolchain is much smaller - ca. 16 minutes. Usage of "ccache" enabled further reduction of compilation time to ca. 8 minutes.

To allow students to work with a distribution suited for embedded systems, except Buildroot, the OpenWRT has been used. In that case, however, the system is not built from scratch. The students use the precompiled version and build their own necessary extensions with the OpenWRT SDK.[4]

*The tests were performed on a machine with Intel Core i7-4790 CPU working at 3.80GHz

1.3 Problems arising from the coronavirus pandemic

Due to the restrictions aimed at limiting the spreading of the COVID-19, it was required to suspend regular lectures and laboratory sessions and perform all the classes remotely. In the case of lectures, the problem was solved by using videoconferencing applications. First tests were done using Jitsi,[5] but finally, Microsoft Teams[6] was used. In the case of laboratories, however, the problem was more complicated. Unfortunately, it was not possible to configure laboratory setups and provide students with remote access. Certain operations (e.g., power-cycling the Raspberry Pi, or modification and verification of electrical connections) required physical access to the hardware. It was also impossible to borrow the laboratory sets to the students because the number of students was much higher than the number of available sets (the laboratories were performed in 5 groups sharing the same equipment). Therefore the only solution was to perform the laboratory using the virtual hardware which the students could simulate in their computers.

2. PREPARATION OF THE VIRTUAL LABORATORY

The essential component of the virtual laboratory is the simulator that emulates the embedded system. The free and open-source QEMU[7] simulator was selected for that purpose. According to the documentation,[8] it is able to emulate 22 hardware platforms. It includes models of many peripheral devices and may be extended by the user by adding the new models of devices. In particular, the QEMU offers emulation of the I/O devices essential for the embedded systems:

- The UART serial console,
- The mass storage - hard disk, SD card or FLASH memory,
- Different methods to connect the emulated machine to the network.[9]

It is also possible to connect the user-provided devices, but that will be discussed in section 2.2

2.1 Selection of the emulated platform

The initial estimations suggested that the regular laboratories will be resumed at the end of the semester. Therefore, it was planned to perform the laboratory exercises first using the virtual platform in the first part of the semester, and then repeat them on the real hardware in the second part of the semester[†]. To simplify porting the designs from the virtual platform to the real one, the emulated platform should be similar to the Raspberry Pi. The Raspberry Pi 2B is emulated in QEMU as the "raspi2" machine, but unfortunately, the simulation is incomplete, and especially the booting process is not fully modeled.[10] Therefore it was excluded.

Review of the QEMU-based virtual platforms, directly supported by Buildroot, gives the following choices:

- arm-versatile
- arm-vexpress
- arm-virt

The "arm-versatile" board is based on a very old arm926t CPU, and therefore it was rejected. The "arm-vexpress" may be emulated with cortex A9 or cortex A15 processor and is suitable for experiments with Buildroot. However, its disadvantage is the lack of USB and PCI support, which limits the possibilities to connect external devices. Only the "arm-virt" platform is supported by Buildroot and by precompiled versions of OpenWRT. The Buildroot offers only a configuration for the 64-bit version of "arm-virt", but this configuration may be easily modified to support the 32-bit version.

Therefore, finally, the "arm-vexpress" and "arm-virt" platforms have been chosen as the virtual laboratory basis.

[†]Unfortunately, as the pandemic has not finished before the end of the semester, the laboratory had to be performed only in the virtual version.

2.2 Support for additional hardware

The QEMU implements models of various I/O devices and allows the use of certain devices (e.g., the soundcard) from the host machine. However, the important part of the laboratory is the possibility of working with additional devices, in particular the simple GPIO-connected I/O board with LEDs and switches. There are two main possibilities to support the emulation of such hardware. One is the forwarding of devices connected to the host. The another is adding the dedicated device model.

2.3 Forwarding of host devices

QEMU allows forwarding of the USB devices connected to the host, to the emulated machine. It is required that the user running the QEMU has read/write access to the device (which can be easily obtained by modifying the udev rules on the host) and that the forwarding command is added when starting the QEMU. The following commands switch on USB emulation in the “arm-virt” machine and forward the device with a given vendor and product identifiers to the emulated USB 2.0 bus:

```
-usb \
-device qemu-xhci \
-device usb-ehci,id=ehci \
-device piix4-usb-uhci \
-device usb-host,vendorid=0x0c45,productid=0x6340,bus=ehci.0 \
```

Unfortunately, the USB forwarding cannot be used for the "arm-vexpress" platform, as it is not equipped with any working model of a USB controller. The USB device forwarding may also be used to give the emulated machine access to the I2C devices connected to the host. It is possible by the usage of the USB-I2C adapter. The cheap open hardware and open source solutions[11,12] are available and are serviced by the “i2c-tiny-usb” driver available in the mainline Linux kernel. A similar approach may be used to forward the SPI devices, but the only freely available USB to SPI adapter, based on the CH341A chip,[13] supports only a limited set of SPI modes. There are commercial USB to SPI adapters supported by the mainline kernel, manufactured by Diolan,[14] but they are rather expensive for a typical student.

There is a similar problem with the availability of the USB to GPIO adapters that could be used for connecting the I/O user interface board. Additionally, there is no reason to force the students to use the real hardware user interface board, as its functionality may be emulated using the GUI on the host machine. Therefore, for that purpose, another approach was used.

2.4 Existing host-accessible GPIO emulations in QEMU

In that approach, the GPIO chip model in QEMU should be able to communicate with an application running on the host. Particularly, that application may implement the GUI with simulated LEDs, buttons, and switches. It is necessary to provide bidirectional, asynchronous notifications. The host application must be informed about the GPIO state’s changes caused by the emulated machine, and the emulated machine must be informed about the changes of the state of the connected buttons and switches in the GUI.

The search for similar solutions delivered two designs. The first one, “virtual_gpio_basic”,[15] is based on the “ivshmem” device, a PCI device model in QEMU, and enables communication between multiple virtual machines or between a virtual machine and its host. In the new QEMU, the “ivshmem” device is replaced with two separate implementations - “ivshmem-plain” and “ivshmem-doorbell”, and therefore that solution requires an update. However, the bigger problem is that it is not compatible with the “arm-vexpress” platform because it does not support PCI. The second design[16,17] adds the model of the Raspberry Pi GPIO to the “arm-versatile” platform. It uses the shared memory for communication between the host and the emulated machine but does not implement any asynchronous notifications. For example, the LED widget must periodically read the shared memory region to update the LED’s state.

To overcome the limitations of the existing designs, the new solution has been proposed and developed.

2.5 The new host-accessible GPIO emulation in QEMU

In the new approach, the POSIX message queues are used to provide bidirectional communication between the GPIO chip model in QEMU and the application running on the host. Short messages containing just three fields are transmitted, for input pins from host to QEMU and for output pins from QEMU to the host (currently, each pin's direction must be defined in advance):

- GPIO magic value: 0x6910 (16 bits)
- GPIO pin number (8 bits)
- GPIO pin new state: 0 or 1 (8 bits)

To be usable as the teaching aid in the laboratory, the emulated GPIO should be compatible with the old, sysfs-based method to control GPIOs,[18] and the new libgpiod-based control.[19] An existing model of the GPIO chip supported by a mainline kernel driver was chosen as a start point to ensure that. After the review of available GPIO models, the MPC8XXX model was selected. As that chip is not available in the "arm-vexpress" and "arm-virt" platforms, the reference to the model was added in the arrays describing the address space, and the interrupt vectors. In the case of the "arm-vexpress" platform, the node describing the new GPIO chip has been added to the device tree. In the case of the "arm-virt" platform, the device tree is created dynamically when the emulation is started. Therefore, for that platform the function adding the device tree node and its properties has been added to the procedure that initializes the virtual platform.

The messages transmitted from the host to QEMU are received in a dedicated thread, which decodes the message, updates the state of the particular input pin in the model, and triggers the interrupt if that pin is configured as an interrupt source.

2.5.1 Implementation of the host-GUI based I/O board.

The emulated MPC8XXX chip delivers 32 I/O lines. The I/O board assumes that the first twelve lines are the inputs connected to the stable switches, the next twelve bits are the inputs connected to unstable buttons, and the last 8 bits are the output connected to LEDs. The I/O board has been implemented in Python, using the GTK toolkit in ca. 200 lines of code (excluding comments). Figure 2 shows the I/O board GUI.

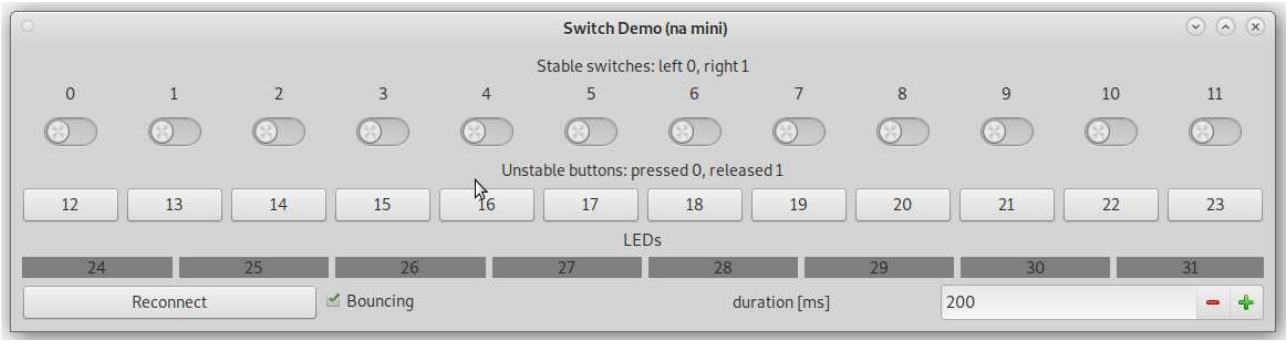

Figure 2: Graphical user interface imitating the I/O board.

The emulated I/O board may also simulate the problem typical for mechanical switches and buttons – so-called bounce effect. If the "bouncing" is switched on, the board emulates the random number (1, 2, or 3) of output state pulses whenever the switch or button changes its state. The user defines in milliseconds the maximum duration of simulated bouncing.

2.6 Availability of the sources

The virtual laboratory setup was published for the students as a github repository.[20] The "master" branch delivers an example implementation of the Internet radio that may be run either on a QEMU-emulated machine or on a real Raspberry Pi or Orange Pi Zero board. The branch "gpio" adds the I/O board's emulation for "arm-vexpress" in Linux. A special

"gpio_uboot" branch adds support for the emulated I/O board also in U-Boot, with a possibility to create the bootable SD card image, hence enabling students to prepare designs with button-selectable "safe" or "user" modes. The branches "gpio_simple_USB" and "gpio_simple_USB_USBIP" add support for USB in the "arm-virt" platform (without support for U-Boot and booting from emulated SD card).

3. RESULTS AND DISCUSSION

The proposed virtual laboratory setup was successfully used during the second semester of the academic year 2019/2020. The majority of students successfully completed the laboratory assignments, developing and debugging their designs at home using the provided repository as a template for their designs. The block diagram of the typical configuration is shown in Figure 3.

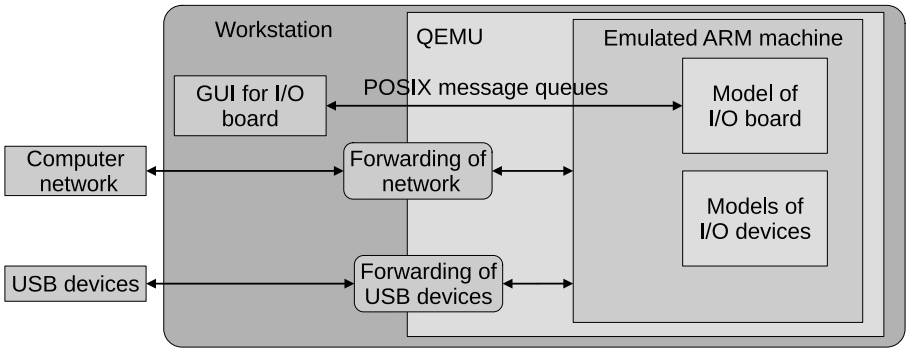

Figure 3: Configuration of the virtual laboratory running on the user's machine.

In a few cases the students could not compile the Buildroot or run QEMU efficiently on their home machines (e.g., when they had the Windows-only computers and could run Linux only in a virtual machine). For them, a possibility to remotely use the Linux machines at the University was created. The compilation of the Linux image, running the QEMU, and the I/O board GUI was done on the remote machine. However, in such configuration, the additional USB devices may be forwarded only from the remote machine, which is physically inaccessible for the student. That limitation has been successfully overcome by using the USB/IP.[21] The resulting configuration of the virtual laboratory in those cases is shown in Figure 4.

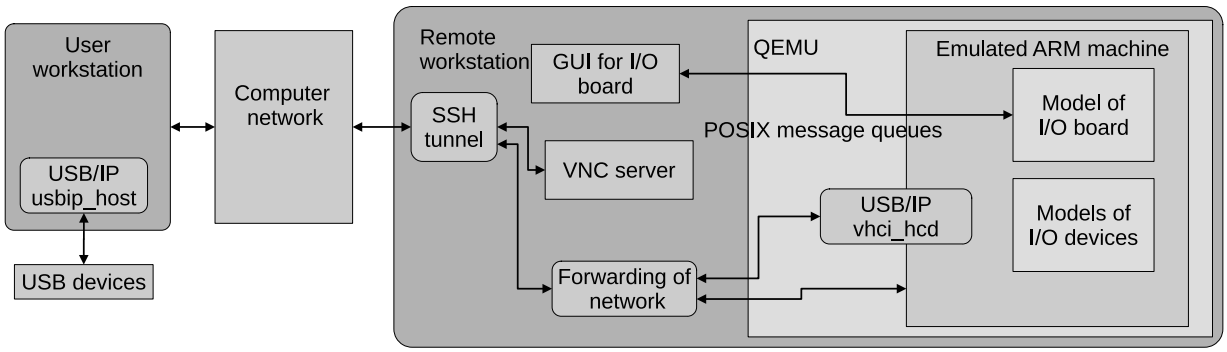

Figure 4: Configuration of the virtual laboratory running on the remote machine.

The quality of USB devices' forwarding via USB/IP highly depends on the forwarded device's bandwidth usage and the quality of the student's Internet connection. In most cases, the problems occurred in the USB soundcard's forwarding, as the sound was interrupted and distorted.

The emulated GUI-connected I/O board worked well. However, there is a problem with synchronizing the initial state of the GPIO chip's QEMU model and the GUI. A dedicated "reconnect" routine has been added that should be run after both QEMU and GUI is started and connected.

The patched QEMU generated with the Buildroot was also successfully used to run the precompiled OpenWRT image with students' additions.

Further research on possibilities to connect QEMU-implemented device models with host-running applications is needed. It would especially be interesting to either forward (otherwise than via USB) or externally model the I^2C and SPI devices.

Except for the technical issues, it must be stated that the virtual laboratory also has certain deficiencies. In particular, the contact between the teacher and the student is worse. It is much more difficult to give the students the subtle suggestions, helping them to find the right solution by themselves.

Also, the students do not feel working in a group, which is important, even if they work on their individual tasks. In the regular laboratory, the teacher's hints to a single student are heard by others and may stimulate further questions or discussion, that is beneficial for the group as the whole. That positive effect is significantly reduced even when the virtual laboratory classes are performed via the videoconferencing application.

The virtual laboratory appeared to be more time-consuming and tiring for the teacher. The designs provided by the students had to be compiled and verified by the teacher. In the standard laboratory, that task could be significantly parallelized. The teacher could ask the student to demonstrate the essential functionalities of the design. He could also ask the student to explain certain details or to modify the solution slightly. That enabled verification of the authorship. In the remote laboratory, the verification if the student prepared his or her solution independently is more difficult.

An additional disadvantage was that the students had no opportunity to work with the real hardware. Therefore they could not be practically trained regarding the right practices to work with electronic devices (e.g., the safe handling of hardware including the ESD protection).

It must be emphasized that the solutions prepared for the virtual laboratory may be useful also after the pandemic period.

A possibility of verifying the design using the virtual hardware may help the students prepare better for the standard laboratory. Such a possibility may be fruitful not only in education but also in developing the software for the embedded systems. The extended QEMU enables verification of the prepared operating system in various platforms and in various hardware configurations, improving the development and testing.

4. CONCLUSIONS

Thanks to the open QEMU emulator and its extensibility, it was possible to achieve the aims of the "Linux for embedded systems" course in pandemic conditions.

It was possible to create the virtual equivalent of the setup used in the normal laboratory, enabling the students to perform the laboratory assignments at home with the teacher's remote assistance.

The prepared virtual laboratory setup may also be useful in normal teaching after the pandemic. It may help the students better prepare for the laboratory or perform their own experiments extending the laboratory assignments.

The proposed solutions may also be used to improve the embedded Linux development and testing in practical applications.

REFERENCES

[1] "Yocto Project – It's not an embedded Linux distribution – it creates a custom one for you." https://www.yoctoproject.org/ [Online; accessed 18-July-2020].

[2] "Buildroot - Making Embedded Linux Easy." https://buildroot.org/ [Online; accessed 18-July-2020].

[3] "OpenWrt Project: Welcome to the OpenWrt Project." https://openwrt.org/ [Online; accessed 18-July-2020].

[4] "Using the SDK." https://openwrt.org/docs/guide-developer/using_the_sdk [Online; accessed 18-July-2020].

[5] "Jitsi.org - develop and deploy full-featured video conferencing." https://jitsi.org/ [Online; accessed 18-July-2020].
[6] "Czat, spotkania, połączenia, współpraca – Microsoft Teams." https://www.microsoft.com/pl-pl/microsoft-365/microsoft-teams/group-chat-software [Online; accessed 18-July-2020].
[7] "QEMU - the FAST! processor emulator." https://www.qemu.org/ [Online; accessed 18-July-2020].
[8] "QEMU - Documentation/Platforms." https://wiki.qemu.org/Documentation/Platforms [Online; accessed 18-July-2020].
[9] "QEMU/Networking." https://en.wikibooks.org/wiki/QEMU/Networking [Online; accessed 18-July-2020].
[10] "Bare metal Raspberry Pi 2: Generating an SD card image for QEMU emulation." https://stackoverflow.com/questions/49025416/bare-metal-raspberry-pi-2-generating-an-sd-card-image-for-qemu-emulation [Online; accessed 18-July-2020].
[11] Harbaum, T., "i2c-tiny-usb." https://github.com/harbaum/I2C-Tiny-USB [Online; accessed 18-July-2020].
[12] Thompson, D., "i2c-star - A STM32 based clone of the i2c-tiny-usb." https://github.com/daniel-thompson/i2c-star [Online; accessed 18-July-2020].
[13] Schorcht, G., "CH341A USB to SPI and GPIO Linux kernel driver." https://github.com/gschorcht/spi-ch341-usb [Online; accessed 18-July-2020].
[14] "USB-SPI Interface Adapters Comparison Table." https://diolan.com/usb-spi-adapters-comparison [Online; accessed 18-July-2020].
[15] Shubin, N., "Virtual gpio module for using together with ivshmem qemu." https://github.com/maquefel/virtual_gpio_basic [Online; accessed 18-July-2020].
[16] Platt, E. R., "Virtual Peripheral Interfaces in Emulated Embedded Computer Systems." https://repositories.lib.utexas.edu/bitstream/handle/2152/46169/PLATT-MASTERSREPORT-2016.pdf [Online; accessed 18-July-2020].
[17] Platt, E. R., "MSc Thesis – source code." https://github.com/evplatt/mse_report [Online; accessed 18-July-2020].
[18] "GPIO Sysfs Interface for Userspace." https://www.kernel.org/doc/Documentation/gpio/sysfs.txt [Online; accessed 18-July-2020].
[19] Gołaszewski, B., "New GPIO interface for linux user space." https://ostconf.com/system/attachments/files/000/001/532/original/Linux_Piter_2018_-_New_GPIO_interface_for_linux_userspace.pdf [Online; accessed 18-July-2020].
[20] Zabołotny, W. M., "Simple Internet radio built using mpd/mpc and Flask with Buildroot." https://github.com/wzab/BR_Internet_Radio [Online; accessed 18-July-2020].
[21] Brown, P., "Tutorial – USB/IP." https://www.linux-magazine.com/Issues/2018/208/Tutorial-USB-IP [Online; accessed 18-July-2020].