Approaches to teaching embedded development

Radu Mantu
University Politehnica of Bucharest
radu.mantu@upb.ro

Florin Stancu
University Politehnica of Bucharest
florin.stancu@upb.ro

Mihai Chiroiu
University Politehnica of Bucharest
mihai.chiroiu@upb.ro

Nicolae Țăpuș
University Politehnica of Bucharest
nicolae.tapus@upb.ro

2023 22nd RoEduNet Conference: Networking in Education and Research (RoEduNet) | 979-8-3503-9366-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/RoEduNet60162.2023.10274941

***Abstract*—There is an ever-increasing demand for the embedded systems skillset in both research and industry projects. Aside from requiring a high level of understanding in Operating Systems and Computer Architectures, one also needs to master low-level programming, compilers, build systems and debugging skills. In this paper, we present our take on organizing an ARM development workshop indended for university students pursuing Bachelor's and Master's degrees. Our work aims to instill fundamental knowledge of embedded systems architecture and low-level software development on ARM Cortex-A platforms.**

Our curriculum encompasses lectures and practical lab sessions focusing on building the firmware, bootloaders and Linux distributions from scratch for real-life hardware. This paper asserts three key contributions: first, it proposes a methodology for teaching systems engineering, with specific emphasis on the ARMv8 architecture. Second, it delves into our challenges encountered during the workshop, exploring potential solutions and alternatives. We also analyze the feedback from the participants, which gives insights on what went well, what didn't, where the main difficulties lie and suggestions for overcoming them in future iterations. Lastly, the paper highlights our commitment to education by sharing workshop materials on the university's Open Courseware platform.

***Keywords*—ARM development, Education, Open-Source**

I. Introduction

In order to address the rising demand for embedded systems engineers, we have organized an ARM development [1] workshop, sponsored by Google and in collaboration with NXP. Having selected 20 students from both Bachelor's (Bs.) and Master's (Msc.) programmes, our trainers organized lectures and practical lab sessions with the goal of familiarizing them with the industry standards [2], [3] and providing them with the skill set necessary to create a Linux distribution from scratch and set up the necessary bootloaders.

In our lectures, we covered the fundamentals of Operating Systems [4] design and presented a comparative analysis of the ARMv8-A boot process and its x86-64 counterpart, giving due consideration to the differences between BIOS or UEFI [5] and the more notable Open Source Software (OSS) ARM bootloaders. During the lab sessions we encouraged collaboration between the younger and the more experienced participants, establishing this as a precept from the onset. As a result, we observed significant improvements day over day, especially in students that were still in their first three years of Bs.

We claim the following contributions:

- We propose a methodology for teaching ARMv8 [6] development on Cortex-A processors, with a focus on the i.MX8M family.
- We discuss the challenges we have encountered while organizing the workshop, as well as the benefits of alternative solutions, or lack thereof.
- We have made our teaching materials publicly available on the Open Courseware of our university [1].

The remainder of this paper is structured as follows. Section II presents the particularities of ARMv8-A simulators and emulators when contrasted to real hardware. Section III elaborates on the Trusted Firmware-A boot process and how we structured our sessions in order to cover each boot stage. In Section IV we discuss the feedback we received. Section V concludes this paper.

II. A choice of hardware and simulators

One of the earliest choices that impact the teaching strategy consists of selecting a platform for students to work with. Although utilizing development boards (i.e.: actual hardware) would be ideal, one must factor in their cost and market availability. Consequently, we take into consideration both system emulators and simulators as alternatives. The alternatives we identified are as follow:

Fixed Virtual Platform (FVP): These complete system simulators offered by ARM are the best choice for bare-metal development. Normally, System on Chips (SOC) come with preinstalled bootloaders (BL_1) burned into eFuses by the Original Equipment Manufacturer (OEM). The FVP on the other hand allows the student to easily develop and prototype the initial bootloader. Among the different choices of FVP flavors, we recommend the Architecture Envelope Models (AEM). These are fixed platform configurations for ARMv8-A and the newer ARMv9-A architectures. Utilizing these configurations can ensure consistency between the work environment of every student, thus allowing for easy replication of encountered problems for the trainer. We note that performing an accurate simulation is a computationally costly endeavor. As a result, certain compromises should be made in order to ensure a tolerable work experience. For example, the default behaviour of the FVP is to simulate

[1] https://ocw.cs.pub.ro/courses/ass

979-8-3503-9366-8/23/$31.00 ©2023 European Union

an authentic representation of the CPU cache. In order to accomplish this feat, the CPU frequency is diminished by approximately an order of magnitude. Disabling this feature can increase responsiveness with no significant detriment. One aspect that needs to be taken into account however, is correctly configuring BL_1. On ARM Cortex-A processors, the System Counter frequency is provided to any software that requires it by means of the *CNTFRQ_EL0* register. This register is set once during initialization by a BL_1 bootloader that was written with knowledge of the underlying hardware. Unless properly customized, the user may experience a skewed representation of time inside the FVP.

QEMU: As one of the most prevalent system emulators in the Linux ecosystem, QEMU [7] leverages a JIT called the Tiny Code Generator (TCG) to emulate a wide range of CPU architectures, including 32 and 64-bit ARM. Unlike the FVP, QEMU focuses on performance rather than accurate hardware representation. This is achieved by integrating offloading support for a number of accelerators (i.e.: hypervisors), most notable being the Kernel Virtual Machine (KVM) [8], [9] but also by implementing support for VirtIO [10] devices that are specifically designed for efficient operation in virtualized environments. Additionally, QEMU exposes a rich configuration interface via its QEMU Monitor Protocol (QMP) that enables even an inexperienced user to perform simple configurations through interface tools such as the Virt Manager. It is important to note that due to the concessions made in favor of execution speed, QEMU does not emulate most micro-architectural features such as CoreSight [11]. In cases where components such as the Program Flow Trace (PFT) or Embedded Trace Macrocell (ETM) are required (e.g.: fuzzing, control flow enforcement, etc.) QEMU fares worse than its counterparts. Despite the lack of CoreSight and by extension, of the Debug Interface (ADI) and Serial Wire Debug (SWD), QEMU presents significantly superior debugging capabilities by means of its integrated GNU Debugger (`gdb`) [12] server. In contrast to this, the FVP only allows debugging via ARM's proprietary Iris Debug and Trace interface that can be interacted with from the ARM Development Studio. Although the FVP has GDB integration in the form of a plugin, it is important to note that it has a reduced feature set and is not part of the Base FVP, but instead of the Fast Model deliverable.

Development boards: Chip manufacturers such as NXP or Microchip usually embed a number of different systems on an integrated circuit called System on a Chip (SoC). These systems include the Central Processing Unit (CPU) with an optional Visual Processing Unit (VPU) but also memory interfaces, I/O devices (e.g.: UART) and I/O interfaces (e.g.: PCIe). Before designing the final product and sending the PCB design for manufacturing, everything is usually prototyped on a development board. This board consists of a System on a Module (SoM), a board-level circuit that integrates the SoC and can be further connected to an extension boards that contains a multitude of peripheral devices (e.g.: HDMI output, Ethernet connector, etc.) One aspect that makes development boards preferable to the FVP and QEMU for teaching purposes is the addition of IPs by the SoC designer. For the i.MX8M family of processors for instance, one must load proprietary firmware offered by NXP during the booting sequence in order to enable access to the DRAM. Another example would be the power regulator devices embedded in the SoC that are responsible for voltage clamping, and device activation, as well as myriad other features exposed via the *syscon* driver on Linux. Both previously mentioned elements are redundant to simulated or emulated platforms. Furthermore, the implementation of the power regulator can cause issues that one would not encounter unless working on actual hardware (although admittedly, most are related to bare-metal virtualization). This holds even more true since multiple SoC versions from the same family can have different power regulators. One aspect to consider is that hardware debugging is more prohibitive than on QEMU or even FVP. While most development boards expose a JTAG or SWD interface, a hardware debugger is required in order to interact with these systems. Unfortunately, hardware debuggers can be several times as expensive as the development boards themselves. Morover, support for certain SoCs is limited (e.g.: multiple Segger debuggers lacking support for the i.MX8M Cortex-A processor but being able to interact with the Cortex0-M MCU).

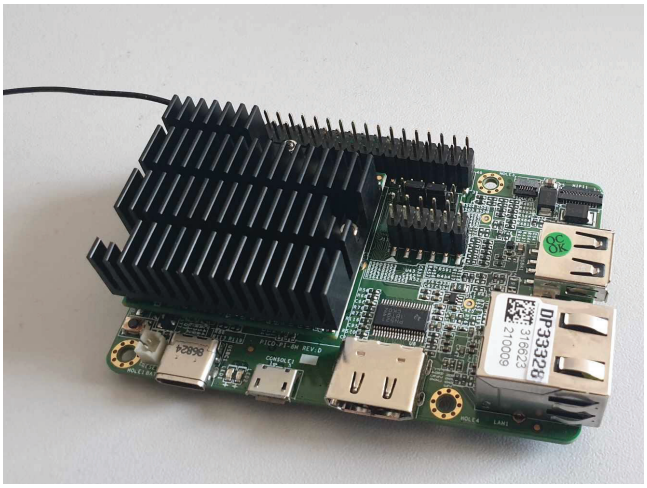

Fig. 1. TechNexion Pico-Pi development board based on i.MX8M Quad.

Thus, we decided to base our hands-on lab sessions on a somewhat cheap hardware development device, the TechNexion PICO-PI carrier board (Fig. 1) with a NXP i.MX8 Quad (ARM Cortex A-53) CPU.

III. The Trusted Firmware-A boot process

While true that, aside from the interaction with the bootROM, the developer is free to design his boot process as he pleases, we decided to adhere to the Trusted Firmware-A (TF-A) firmware design since it is widely adopted in the industry. As a result, the ARMv8-A boot is split into the boot stages that are depicted in Fig. 2 and are part of the Firmware Image Package (FIP). Although Trusted Firmware

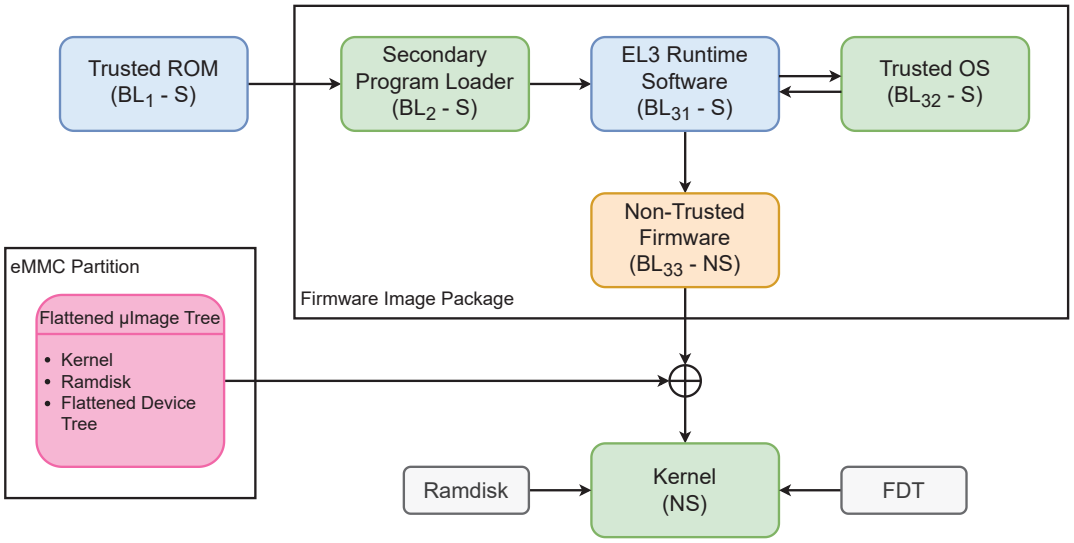

Fig. 2. ARMv8 booting process. Each bootloader runs either in Secure (S) mode or Non-Secure (NS) mode. The exception level at which each bootloader runs is color coded: Blue - EL3, Yellow - EL2, Green - EL1.

does provide reference implementations for each of these components, we decided to use the most widely recognized OSS projects that implement them.

- BL_1: Trusted ROM - provided by the OEM.
- BL_2: Secondary Program Loader - U-Boot SPL.
- BL_{31}: EL3 Runtime Software - TF-A.
- BL_{32}: Trusted OS - OP-TEE.
- BL_{33}: Non-Trusted Firmware - U-Boot.

Although it is possible that Android's Trusty is encountered more often, we argue that trying to create a Linux distribution from scratch instead of porting Android to the Pico-Pi i.MX8MQ board renders the point mute. Furthermore, OP-TEE [13] is already well standardized, having in-kernel support and still being actively developed.

In the materials that we have published on the Open Courseware of our university, one may notice that, more often than not, we have used forks of the previously listed projects, their mainline counterparts being plagued by regression bugs due to lack of testing on real platforms which, unfortunately, is still the case with such low-level embedded firmware projects.

For example, in the case of BL_{31}, the mainline TF-A build process currently has a linker script error for our platform so we decided to use the NXP fork instead. The same applies for BL_2 but for a different reason. Since U-Boot is continuously being updated, new features are regularly being added to certain drivers. Naturally, this causes the drivers to increase in size. Since our platform does not have an SD card slot, the only solution available (at least initially) for us to provide BL_1 with the FIP image containing the remaining bootloaders was to use the Serial Download Protocol (SDP). As a result the USB driver had to be added to BL_2 in order for it to take over the data reception from BL_1. However, the addition of the USB driver from mainline U-Boot caused the size of BL_2 to exceed 64KB, the maximum amount of available SRAM (keeping in mind that DRAM is inaccessible at this point in the boot process). As a result, we decided to use an older U-Boot fork that still had a working USB driver implementation but with fewer features.

In the following sub-sections we describe how we partitioned the study of the TF-A boot proces across the four sessions and what our intentions were for each of them. We note that we allocated an extra session at the midway point in order to ensure that all students can catch up and have a functioning system for moving forward.

A. Session 1: Firmware & Bootloaders

In this initial session, the goal was to create a Firmware Image Package containing the essential bootloaders (i.e.: BL_2, BL_{31}, BL_{33}) as well as the proprietary NXP firmware needed to enable access to the 2GB of Low-Power DDR4 DRAM.

Creating this FIP image required the students to install an appropriate toolchain for cross-compilation and familiarize themselves with the Kbuild system, used primarily by the Linux kernel but adopted by other notable OSS projects such as U-Boot, Unikraft and Buildroot.

Due to the USB driver size issue forcing us to use an older U-Boot fork, we were unable to utilize the recommended FIP image packer called `binman`. Instead, we used `imx-mkimage`, a wrapper over *u-boot*'s `mkimage` that provides object layout information to the backend so that the resulting image is compliant with the SDP implementation from BL_1.

Succeeding in constructing the FIP image and booting it via SDP would bring the students up to BL_{33}, the first interactive component of the booting process. By interacting with the U-Boot shell, they would be able to extract board information statically configured in BL_1, obtain the relocation information of BL_{33}, perform memory tests on the DRAM banks and

query the MMC subsystem for available persistent storage devices.

B. Session 2: Bootstrapping Linux

Having reached BL_{33} in the first session, a Linux kernel, a Flattened Device Tree (FDT) [14] and a root filesystem would next be required in order to obtain a fully functioning system. The goal for this session was to create a Flattened uImage Tree (FIT) [15] encapsulating the aforementioned elements. The FIT format is in fact a Device Tree Blob (DTB) organized in such a way that U-Boot can interact with it, extract each individual component at pre-established memory locations and cede control to the newly loaded kernel.

In order to build the root filesystem, we had two choices: Buildroot [16] or the Yocto project [17]. Based on our experience with Yocto we deemed it too complex to understand in only a few hours. The difficulty of learning Yocto stems mostly from its `bitbake` build system and the overlay-based recipe extension system. Moreover, it is known that Yocto has a high resource consumption, requiring at least 50GB of storage for a single build. Buildroot on the other hand uses the already familiar Kbuild system and has a more accessible documentation available.

Although the FIT image could be placed in the Firmware Image Package as a data blob to be loaded at a pre-established address, we decided to place it in a FAT32 partition on the eMMC. To this end, we had the students create a loopback block device, initialize it with a Master Boot Record (MBR) partition table, add a partition and format it with a FAT32 filesystem. In order to overwrite the eMMC memory with this block image we used a TechNexion fork of the Universal Update Utility (`uuu`) offered by NXP. The reason behind this is that TechNexion have a `uuu` script and bootloader already implemented that perform the eMMC overwrite operation with data received over SDP. Although there is a better alternative to this solution, we wanted to introduce the concept of loopback devices since they are vital in understanding modern application packaging systems such as Snap and Flatpack.

C. Session 3: Trusted Execution

After ensuring that everyone had a working Linux build, we tasked the students with installing a Secure World (ARM TrustZone) OS, OP-TEE, within the boot FIP Image (as BL_{32}). By convention, OP-TEE should reserve space (usually 36MB) at the end of the DRAM bank, or the 32-bit address space. Because OP-TEE did not have a configuration for the Pico-Pi, these values needed to be calculated and set manually. Moreover, this physical address range needed to be communicated to the Linux kernel by means of a `/reserved-memory/` node in the device tree. Although unlikely, unknowingly accessing this address range from a Non-Secure CPU state (e.g.: kernel buffer allocation) would cause an access error.

Once OP-TEE was loaded, initialized and recognized by the kernel, the students moved on to writing a Trusted Application to run at S-EL0 (Secure Userspace). In order to easily copy over the user space OP-TEE library, service and test applications, we instructed the students to use the mainline BL_{33} to expose the eMMC device to their host computer via USB Mass Storage (UMS) [18]. This effectively gave them direct access to the eMMC device to freely partition, format and mount, without relying on the TechNexion `uuu` script and the SDP eMMC flashing process.

D. Session 4: Kernel development

In the final session we broached the subject of kernel development, starting with the fundamentals of writing kernel modules. To further simplify the process of transferring the cross compiled kernel modules to the board (i.e.: not restart the board and use UMS) we connected the two via an Ethernet cable and asked the students to assign static IP address to the respective network interfaces. As a means of transferring the files we had first rebuilt the root filesystem, outputting it directly on an ext4 partition (in order to have symlinks), and including the `OpenSSH` server package.

Following this initial setup stage, the first driver the students were tasked to write consisted of a simple character device that ensured communication between user space and kernel space. Using this as a starting point, they would then identify the physical address of the serial device used as backend for the teletype (tty). This was done by parsing *sysfs* via `udevadm`. Having obtained this information, they remapped the physical address range of the memory mapped UART registers to kernel virtual memory, only to implement their own `earlycon` UART driver. This type of driver is usually employed early on, before the Generic Interrupt Controller (GIC) is set up and uses a polling-based approach to output early debug messages.

IV. Evaluation and student feedback

For our workshop, we accepted 20 students from over 50 applicants based on a brief technical questionnaire and their CV. Out of the 20 students, 14 have completed the workshop and 12 have provided feedback. The feedback consisted of a form proposed by the mentors, containing mostly free-answer inquiries, and a questionnaire hosted on a Moodle instance.

Based on the feedback, we quantified the student satisfaction in Fig. 3. The recurrent aspects that influenced these ratings were as follow:

+ Novelty and low-level nature of the broached subject. ARM development in general is not a common topic of study in most curricula.
+ The opportunity to work on real hardware, rather than simulators.
+ Technical knowledge of the trainers and ability to quickly resolve issues.
+ Tangential information that clarifies concepts encountered in regular courses.
- Difficulties in progressing the laboratory when an issue arises.
- Lack of a FVP / QEMU setup for independent study.

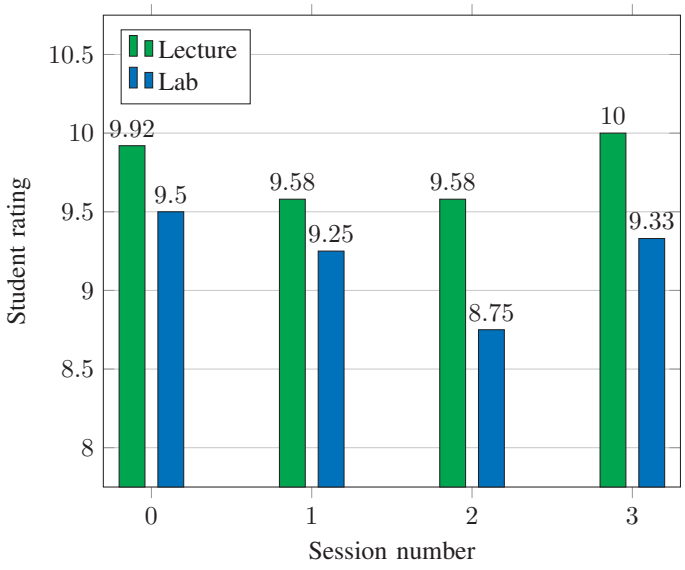

Fig. 3. Student satisfaction (N=12) with regard to the Lectures and Labs comprising each Session, reported on a scale of [1, 10].

From our interaction with the students we identified three factors that resulted in difficulties comprehending the materials or completing the proposed tasks:

- Lack of knowledge regarding Operating System theory. Mostly present for first year Bs. students.
- Large quantity of new information relative to the 4 hour interval allocated for each session.
- Unfamiliarity with Kbuild system and the architecture of the five major OSS projects that needed to be integrated to obtain a working system (i.e.: U-Boot, Trusted Firmware-A, OP-TEE, Buildroot, Linux).

Out of the 12 respondents, 10 considered that the technical abilities gained from these sessions will prove beneficial for their career, 8 have been inspired to apply their new skill set to practical individual projects, and 9 are very likely to participate in future workshops.

V. Conclusion

In this paper we presented our approach to organizing an ARM development workshop. We have discussed the benefits and deficiencies of selecting either a simulator, an emulator or actual hardware as a practical teaching instrument. Furthermore, we have made our teaching materials publicly available. We commit to add support for FVP / QEMU as a means to encourage independent study.

We hope that our work will become a helpful tool for introducing students for the embedded development world, with the aim to sufficiently motivate them to become involved into specific research projects.

Moving forward, we resolve to provide better resources for first year Bachelor's students in order for them to quickly become accustomed with the necessary Operating Systems design concepts. This decision was made in lieu of setting age requirements after observing the disproportionate gains in technical ability. However, we have decided to set minimal requirements with respect to their programming environment since the greatest difficulty for the trainers was solving Virtual Machine specific problems (e.g.: Oracle VirtualBox not being able to emulate USB2 devices by default).

References

[1] M. Levy and C. Promotions, "The history of the arm architecture: From inception to ipo," *ARM IQ*, vol. 4, no. 1, 2005.
[2] M. Ghobakhloo, "Industry 4.0, digitization, and opportunities for sustainability," *Journal of cleaner production*, vol. 252, p. 119869, 2020.
[3] L. S. Dalenogare, G. B. Benitez, N. F. Ayala, and A. G. Frank, "The expected contribution of industry 4.0 technologies for industrial performance," *International Journal of production economics*, vol. 204, pp. 383–394, 2018.
[4] A. S. Tanenbaum and A. S. Woodhull, *Operating systems: design and implementation*. Prentice Hall Englewood Cliffs, 1997, vol. 68.
[5] R. Wilkins and B. Richardson, "Uefi secure boot in modern computer security solutions," in *UEFI forum*, 2013, pp. 1–10.
[6] R. Grisenthwaite, "Armv8 technology preview," in *IEEE Conference*, 2011, p. 60.
[7] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41. Califor-nia, USA, 2005, p. 46.
[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, no. 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230.
[9] I. Habib, "Virtualization with kvm," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.
[10] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.
[11] Z. Ning and F. Zhang, "Understanding the security of arm debugging features," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 602–619.
[12] R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with gdb," *Free Software Foundation*, vol. 675, 1988.
[13] C. Göttel, P. Felber, and V. Schiavoni, "Developing secure services for iot with op-tee: a first look at performance and usability," in *Distributed Applications and Interoperable Systems: 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings 19*. Springer, 2019, pp. 170–178.
[14] D. Gibson and B. Herrenschmidt, "Device trees everywhere," *OzLabs, IBM Linux Technology Center*, 2006.
[15] J. A. Fernandes, "Flattened image trees: A powerful kernel uimage format," in *Embedded Linux Conference*, 2013.
[16] T. Petazzoni and F. Electrons, "Buildroot: a nice, simple and efficient embedded linux build system," in *Embedded Linux System Conference*, vol. 2012, 2012.
[17] A. Vaduva, *Learning Embedded Linux Using the Yocto Project*. Packt Publishing Ltd, 2015.
[18] J. Axelson, *USB mass storage: designing and programming devices and embedded hosts*. lakeview research llc, 2006.