System for Remote Collaborative Embedded Development

Martin Domajnko
University of Maribor,
Faculty of Electrical
Engineering and Computer Science,
Koroška cesta 46, 2000 Maribor, Slovenia
martin.domajnko@student.um.si

Nikola Glavina
University of Maribor,
Faculty of Electrical
Engineering and Computer Science,
Koroška cesta 46, 2000 Maribor, Slovenia
nikola.glavina@student.um.si

Aljaž Žel
University of Maribor,
Faculty of Electrical
Engineering and Computer Science,
Koroška cesta 46, 2000 Maribor, Slovenia
aljaz.zel@student.um.si

ABSTRACT

This paper explores the challenges and devised solutions for embedded development which arose during the COVID-19 pandemic. While software development, nowadays with modern tools and services such as git, virtual machines and communication suits, is relatively unaffected by resource location. That is not the case for firmware and embedded systems, which relies on physical hardware for design, development, and testing. To overcome the limitations of remote work and obstructed access to actual hardware, two ideas were implemented and tested. First, based on integrated circuit emulation using QEMU to emulate an ARM core and custom software to facilitate communication with the embedded system. Second, remote programming and debugging over the internet with a dedicated computer system acting as a middle man between a development environment and physical hardware using OpenOCD debugger.

Keywords embedded development, remote development, OpenOCD, QEMU, ARM, ARM semihosting

1 Introduction

During the design, development, and testing phases of our Passive Floating Probe project [7], we heavily relied on physical access to the hardware. The restrictions that were put in place during the COVID-19 pandemic didn't stop the development process of our project, but they heavily limited it. Lack of working hardware for every member of the team was our major issue, so we tried to think up solutions to overcome that problem. The first idea, which would allow our work on the project to continue, was based on the QEMU [8] machine emulator. We emulated our integrated circuit with an ARM core processor and added a custom software layer, which served as an emulation of the real-world communication pipeline with the system. The solution proved useful, albeit with the restriction that it allowed only local development, prompting us to develop our second idea. Remote programming and debugging of our hardware over the internet was the main part of that solution. This was achieved with a dedicated computer system acting as a middle man between our development environment and the targeted hardware, enabling programming and debugging.

Finally, we present the structure of this paper. In the second section, we are going to explain in detail the implementations of both solutions. Following in section three, where we are going to talk about the limitations of the implemented solutions and the lessons we learned. The paper will conclude in section four with possible improvements.

2 Implementation of remote embedded development

For the work on the project to continue under COVID-19 restrictions, a solution had to be devised to solve the problems that came with remote work on a project dependent on specific hardware. This included hardware development, maintenance, and the ability to develop and test software on the hardware remotely. During the academic year, two solutions were devised and implemented.

2.1 Solution using emulator

The first solution was based on microcontroller emulation, as seen in figure 1, using a specialized version of QEMU emulation software called xPack [14] version 2.8.0-9. This enabled us to emulate a variant of STM32

*Listed in alphabetical order

StuCoSReC University of Maribor Press
DOI https://doi.org/10.18690/978-961-286-516-0.4
ISBN 978-961-286-516-0

family of microcontrollers. The variant chosen was STM32F407-Discovery development board, since it was closest to our target hardware, and we had access to a matching development board on which to test the differences. Using the emulator method proved to be a great benefit, since not all members had access to real hardware at that time, but everyone could set up the emulator on their computer. Since we didn't have previous experience working with STM family microcontrollers, the emulator also allowed us to focus on platform-specific software issues without the complexity of hardware issues in novice programmers. However, this also came with a cost. First one, the emulator didn't work out of the box. Second one, no sensors can be connected to the board directly since interfaces in the emulator are virtual.

Before the emulator could be used with the desired family, a minor bug inside the emulator source code, that was causing ROM memory overlap, needed to be patched manually. After initial setup, we established a basic form of IO system using a script that allowed us to run the emulator and automatically redirect the standard output using UNIX pipes to a file where we could monitor the output. The emulator approach enabled us to test out the version of real time operating system FreeRTOS [2] for STM32 microcontrollers. Some modification to compiler flags were required, especially the soft floating-point unit because the emulator was unable to emulate real FPU. While unable to connect sensors to the virtual environment, we were mostly focused on creating task manager, which took care of the correct operation and communication between individual tasks or sensors. Tasks, that were supposed to be related to the sensors, were temporarily replaced with empty functions that returning predefined values for testing.

To enable outside communication with our embedded software, semihosting feature of ARM architecture was used [1]. A custom communication service was implemented in which standard input and output were redirected to netcat running locally, creating a network interface. After successful compilation of the program, the emulator could be run with the compiled elf binary and redirect semihosting IO to stdio. An example in our case: *< /dev/null nc -q -1 -l 5000 | qemu-system-gnuarmeclipse –verbose –verbose –board STM32F4-Discovery –mcu STM32F407VG -d unimp,guest_errors –image STM32F407-Discovery-blinky.elf –semihosting-config enable=on,target=native | nc -l 6000 > /dev/null.* This allowed us to emulate a serial connection to and from the emulated microcontroller over a network connection.

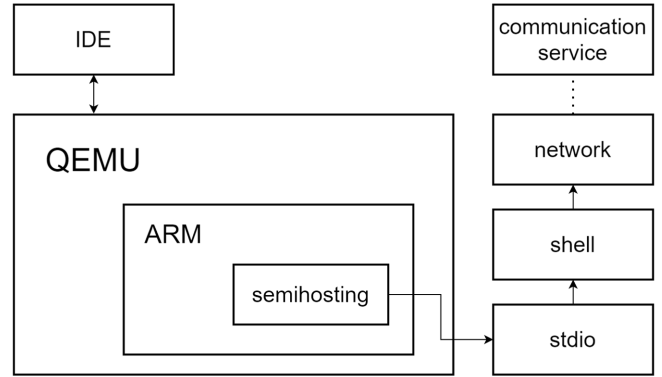

Figure 1: Schematic of the development system using emulation.

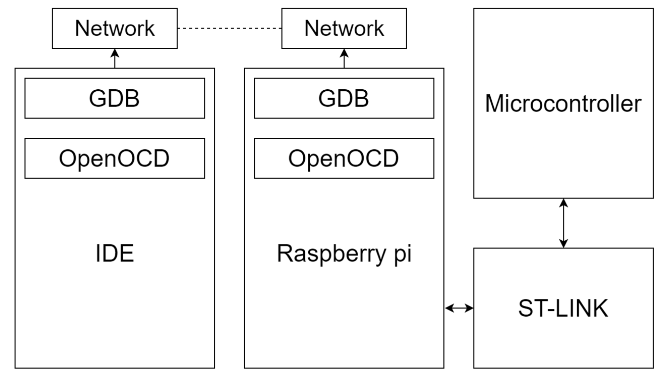

Figure 2: Schematic of the remote development system.

2.2 Solution using remote development

To amend the shortcomings of the development on the emulator, a remote development solution was devised and implemented. The solution, as seen on figure 2, is based on a dedicated computer system leveraging remote connection functionality of GDB or "GNU Project Debugger" [4] for remote programming and real-time debugging.

The system was built using Raspberry Pi 3 Model B [9], running ARM version of Ubuntu version 20.04 LTS [13]. The selected single board computer handled network communications, attached peripherals and services needed to facilitate remote programming and real-time debugging. For embedded development, an ST-LINK in-circuit debugger and programmer [11], in our case STM32F407G-DISC1 development board [10] providing ST-LINK/V2-A, was connected via USB connection to the Raspberry Pi. To connect GDB debugging functionality with the ST-LINK programming functionality with the target integrated circuit, the OpenOCD software [6] version 0.10.0 was used.

To prepare the setup, two configuration files in OpenOCD format needed to be created. First one defining "hla_serial" value, describing the serial number of the connected ST-LINK device. The second one defining "-event gdb-detach" behavior as "resume". Defining this permitted the embedded program to run even after the debug session disconnected, allowing to test the functionality over longer periods of time without constant connection to the host machine.

With the prepared configuration files, a script was created to start the OpenOCD session with the correct parameters for the ST-LINK device, target device, network settings and created configuration files. An example in our case: *openocd -c "bindto $HOSTNAME"*

-c "gdb_port 3333" -c "tcl_port disabled" -c "telnet_port disabled" -f /usr/share/openocd/scripts/interface/stlink-v2.cfg -c "adapter_khz 480" -c "transport select hla_swd" -f /usr/share/openocd/scripts/target/stm32l4x.cfg -f ./gdb_resume.cfg -f ./serial.cfg » log.txt

The script was started inside a tmux [12] instance. This allowed for the OpenOCD session to run without an active user connection to the shell executing the script, or alternately for multiple users to be connected to the same shell instance to observe debug in print messages.

Once the system was set up, multiple ST-LINK devices could be connected and used simultaneously by adding additional configuration files with serial numbers and starting OpenOCD sessions on different network ports.

3 Usage and lessons learned

The development process was arranged in the form of the required hardware development and maintenance to be in the domain of our mentor and be kept at the university. Team members could make request for hardware modifications and then develop and debug the project software remotely. While concurrent remote software collaboration was achieved through distributed revision control system Gitea [3] as source code repository and management tool.

3.1 Emulator

The emulator provided a good start into getting acquainted with embedded development. This allowed us to learn and develop embedded software without physical hardware. The downside was the inability to work with real sensors, which later made us switch to a remote system. Another problem was that the emulator was not capable of running functions that required precise timing, such as real-time code or interrupt execution.

3.2 Remote

The advantage of remote system made it possible for us to connect to the targeted hardware from anywhere as if it was accessible locally. This included real-time debugging with the ability to see microcontroller processor states and memory values. The problem with this solution was the restriction of a single connection to the OpenOCD instance, which limited the work on the hardware to a single developer at a time. This was resolved with communication and access scheduling.

4 Conclusions

The system was sufficient to allow our work on the project to continue. During development, we were fortunate enough to not have major problems with security. The system, as it was used, would allow anyone to access the system if they identified the used network ports and protocols. This was not a major concern, as it only allowed to program our particular microcontroller with a dedicated firmware. As such, this problem was not addressed during the production. Possible additional security was tested, with the implementation of username and password authentication using NGINX reverse proxy server [5] and httpd access restrictions on the system URL.

Acknowledgment

The authors acknowledge the financial support from the Institute of Computer Science of the Faculty of Electrical Engineering and Computer Science and would like to thank mag. Jernej Kranjec for his guidance and assistance. The authors would also like to acknowledge the remaining members of the project group, namely Tilen Koren, Anna Sidorova and Viktorija Stevanoska for their work on the project.

References

[1] ARM. Arm target input/output facilities. https://developer.arm.com/documentation/dui0471/g/Bgbjjgij, 2021. Accessed: 2021-07-30.

[2] FreeRTOS. Real time operating system for microcontrollers. https://www.freertos.org/, 2021. Accessed: 2021-07-30.

[3] Gitea. Lightweight code hosting solution. https://gitea.io, 2021. Accessed: 2021-07-30.

[4] GNU. The gnu project debugger. https://www.gnu.org/software/gdb/, 2021. Accessed: 2021-07-30.

[5] NGINX. Reverse proxy. https://www.nginx.com/, 2021. Accessed: 2021-07-30.

[6] OpenOCD. Open on-chip debugger. http://openocd.org/, 2021. Accessed: 2021-07-30.

[7] Perrone, M., Knupleš, U., Žalik, M., Keršič, V., and Šinko, T. Pasive floating probe. In *StuCoSReC: Proceedings of the 2019 6th Student Computer Science Research Conference* (2019), pp. 13–17.

[8] QEMU. the fast! processor emulator. https://www.qemu.org/, 2020. Accessed: 2020-04-30.

[9] Raspberry Pi. Raspberry pi 3 model b. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, 2021. Accessed: 2021-07-30.

[10] STM. Discovery kit with stm32f407vg mcu. https://www.st.com/en/evaluation-tools/stm32f4discovery.html, 2021. Accessed: 2021-07-30.

[11] STM. Stm st-link/v2 in-circuit debugger/programmer. https://www.st.com/en/development-tools/st-link-v2.html, 2021. Accessed: 2021-07-30.

[12] tmux. Terminal multiplexer. https://github.com/tmux/tmux/wiki, 2021. Accessed: 2021-07-30.

[13] Ubuntu. Ubuntu os. https://ubuntu.com/download/raspberry-pi, 2021. Accessed: 2021-07-30.

[14] xPack. The xpack qemu arm. https://xpack.github.io/qemu-arm/, 2020. Accessed: 2020-04-30.