Emulating a Raspberry Pi Zero Platform to Analyze Insulin-Pump Stack Safety in a CI/CD Pipeline

Martin Ubl
Dept. of Comp. Science and Eng.
University of West Bohemia
Pilsen, Czechia
ublm@kiv.zcu.cz

Jakub Silhavy
Dept. of Comp. Science and Eng.
University of West Bohemia
Pilsen, Czechia
silhavyj@students.zcu.cz

Tomas Koutny
New Technologies for Inf. Society
University of West Bohemia
Pilsen, Czechia
txkoutny@kiv.zcu.cz

***Abstract*—Our research focuses on technologies in diabetes treatment, primarily considering the software architecture and stack used for treatment. We have already demonstrated that our software architecture, SmartCGMS, is capable of hosting complex systems, from laboratory scenarios to intricate treatment setups. As we intend to expand our testing to real hardware, we are compelled to perform thorough testing of our architecture under target conditions. We chose the Raspberry Pi Zero as a consumer-available, cost-effective solution, powered by an ARMv6 processor, which is the gold standard in today's embedded electronics. We introduce our Raspberry Pi Zero emulator, which enables us to conduct exhaustive functional testing and validation through emulation. We present the biggest challenges of emulator design and development and validate its function using a custom real-time operating system.**

***Index Terms*—raspberry, insulin pump, armv6, smartcgms**

I. Introduction

Embedded software plays a pivotal role in technological advancements, powering a vast array of devices ranging from industrial machinery to medical devices. As these embedded systems become increasingly complex and interconnected, ensuring the reliability and robustness of their software components becomes a critical requirement during their deployment [1]. Testing, particularly automated testing, emerges as a critical practice in embedded software development, offering a systematic approach to validating functionality, identifying defects, and ensuring overall system integrity. This paper explores the importance of automated testing in the context of embedded software development for medical devices, clarifying its significance in enhancing product quality, accelerating development cycles, and mitigating risks associated with software failures in mission-critical applications.

Our research focuses on software architecture and stack for Type 1 Diabetes Mellitus treatment along with its deployment to an embedded wearable device. To be able to verify the software stack suitability, safety and reliability, testing on the target environment, i.e., the embedded wearable device, is required. To perform exhaustive testing, we decided to deploy the software stack to an emulator first. This also allows us to integrate the emulator to a Continuous Integration, Continuous Deployment (CI/CD) pipeline, minimizing the occurrence of integration-related errors.

This publication was funded by a university-specific research project SGS-2022-015 of University of West Bohemia.

A. Diabetes mellitus

Diabetes mellitus (DM) is a widespread civilisation disease [2]. It is a heterogeneous group of diseases, all of which manifest with elevated blood glucose (BG) levels. The most common types of diabetes are: Type 1 (T1DM), Type 2 (T2DM), gestational, secondary, etc. T1DM, which is in a primary focus of our research, is caused by absolute insulin insufficiency. As the result, the patient is required to dose the insulin externally. The patient typically does so using either an insulin pen or an insulin pump [3]. Additionally, the patient needs to utilize a BG meter to estimate his/her current glucose levels prior to delivering the insulin dosage. This is important for calculating the correct safe amount of insulin to be dosed.

Modern setups use a semi-automatic insulin pump and a continuous glucose monitor (CGM) [3], [4]. A CGM measures the glucose in short intervals (typically every 5 minutes). This allows for a control loop to be formed through the patient. Current treatments setups use a hybrid-closed loop treatment, which maintains the BG automatically if it is safe to do so, but falls back to a human-validation or manual mode if there are no (software-perceived) safe options [3]. Both the insulin pump and the CGM classify as wearable medical devices.

B. SmartCGMS

SmartCGMS is a software framework and architecture for signal analysis [5]. It decomposes the whole system into a set of software components, which operate independently. The top-level entity is called *a filter*. It may encapsulate other entities, such as *models* (mathematical, ...), *signals*, *approximators*, *solvers* and *metrics*.

Filters are connected to form a linear chain. They communicate by passing a small message called a *device event*. This message is a container of an event that occurred. For example, when a new BG is measured, the device event holding that information is passed from the first filter to the next one, until it reaches the last one in the chain. Such a device event holds the logical and wall-clock timestamp of the event occurrence, signal level (the BG value), and a signal identifier. For every measurement, a new device event is created.

Messages are passed synchronously – one filter passes the message by invoking a method call on the following filter in chain.

A SmartCGMS filter chain is configured with a single *ini* file, that lists all filters in order. Each filter may be configured with its own parameter set. As each filter represents a single functional block, it is possible to use an identical configuration for a simulated run and the target environment, with a single change – we replace the simulated components with real ones.

SmartCGMS is designed to run on all sorts of devices, including desktop computers, servers, embedded devices and wearables [6], [7].

C. Wearable medical devices

A wearable medical device is an embedded device operating in low-power mode, that has been certified for medical purposes [8]. Such purposes may include monitoring of the patient's health, disease treatment, and more. The requirements posed to such a device are high, mainly in terms of safety and reliability, but also regarding the whole development process.

ARM architecture is deemed suitable for the purpose of wearable medical device development [9]. This conclusion is based on many aspects, such as power consumption, operational performance, cost, and support for integrated peripherals. Moreover, it has a long tradition in the embedded device market.

D. Embedded device emulation

Emulation is a process of imitating the behaviour of one (guest) computer architecture on a different (host) computer architecture using a software providing the emulation service (emulator) [10]. Emulation usually imitates functional properties of the guest architecture at the cost of performance [11].

Embedded systems comprise a number of components depending on their purpose. Apart from the System on Chip (SoC), that usually lies in heart of an embedded system, there are a number of peripherals attached to it through some sort of a bus or other interface. The emulation of such a system must support the incorporated SoC, as well, as the connection of specialized peripherals. To model the emulated system credibly and reliably, the emulator has to serve as a digital twin of the real hardware [12]. Thus, it should model the interface of all essential peripherals to reflect the real scenario [13]. For example, a digital twin of an insulin pump must, in addition to the SoC, emulate an insulin pump reservoir, infusion mechanism, communication module (e.g., Bluetooth), and possibly even a signaling device (such as a buzzer or an LED indicator).

Furthermore, an emulated environment may benefit from the simulation aspect. We can, for example, inject errors and faults into the system [14], e.g., simulate the infusion mechanism being stuck, communication being interrupted or jammed, etc. This may greatly improve the safety of the overall system, as we can emulate a wide variety of faulty situations, that the software must adequately compensate for or at least fall back to a safe mode in order to prevent harming the patient [15].

II. Related works

As the scope of the paper spans multiple topics, we split this chapter to two subsections: ARM architecture emulation and CI/CD integration.

A. ARM emulation

In the field of the ARM architecture emulation, there are several solutions available [16]. First of all, *QEMU* provides a strong and universal base for emulation and virtualization using its dynamic binary translation method [17]. Its support for the ARM architecture is sufficient to run most modern operating systems. However, its support for the Raspberry Pi Zero platform is incomplete and contains incompatible implementations. Furthermore, the use of *QEMU* to emulate a feature-complete target system that includes peripherals is problematic and there is currently no support for it.

OVPSim is a complex and powerful emulation suite that supports a variety of architectures and boards, much like *QEMU* [18], [19]. Unlike other tools from the same category, the emulator core is distributed under a closed-source distribution, free for use only for personal and non-commercial use. The so-called *OVP models*, which model the core and peripheral parts of emulated system and platform, are open-source and freely available. The main drawback is the licensing and closed-source distribution of the emulation core.

The *gem5* software is also a complex emulation suite that supports the ARM architecture, as well, as RISC-V, x86, MIPS, and more [20]. It greatly benefits from its modular design and high degree of decomposition. This allows for an universal approach and for modeling of an arbitrary system. Nonetheless, *gem5* does not support any peripherals, nor is it ready for automated testing.

The *CPUlator* is a web browser-based software for ARM architecture emulation that places emphasis on educational aspects, namely on the assembly language and the ARM Instruction Set Architecture (ISA) [21]. The main drawback of this software is its lack of support for peripherals and some of the advanced memory-related features, such as paging and memory-mapped input-outputs (MMIO).

Similar to *CPUlator*, *ARMSim#* is an educational software for ARM architecture emulation [22]. The software focuses on modularity and educational aspects. It is possible to extend the emulator with peripherals. Nonetheless, they are mapped through a non-standard interface. It also lacks support for advanced memory-related features, such as paging and MMIO.

In conclusion, all the available solutions suffers from either lack of support for peripherals and/or advanced memory-related features (e.g., paging, MMIO, etc.), or unsuitable licensing and distribution options.

B. CI/CD integration

As far as the integration of an emulator in CI/CD pipelines is concerned, studies often suggest to use Hardware In Loop (HIL)-based pipelines [23]. This explicitly requires to have the target hardware platform included in the build and testing

pipeline. Nonetheless, such an approach introduces some inherent flaws. For instance, anybody extending the SmartCGMS software stack, who also requires the modifications to be validates, must own the target hardware.

Other studies suggest the use of software-based models or meta-models, that substitute the real hardware with a model [24], [25].

Recently, a digital twin prototype-based testing was adopted to substitute the HIL-based pipelines [26], [27]. Its main advantage is that the system is modelled as a software component and tested using the target configuration. The main difference from the model-based testing is, that it runs on the same codebase, thus mitigating the portability-related bugs and errors.

III. Approach

The main goal is to emulate a selected System on Chip (SoC) with a feature set fulfilling the requirements of the SmartCGMS software stack. The selection of the ARM architecture and Raspberry Pi Zero platform for emulation within our research was strategically reasoned. ARM architecture offers several advantages, including low power consumption, compact size, and cost-effectiveness, aligning well with the resource constraints often encountered in embedded systems and software stacks like SmartCGMS. Leveraging Raspberry Pi Zero enhances our platform's accessibility and affordability, making it more attainable for researchers and enthusiasts. Moreover, the Raspberry Pi Zero's compatibility with a wide range of peripherals and its robust community support facilitate seamless integration and rapid prototyping, which is crucial for the iterative development process inherent in research endeavors. By opting for the ARM architecture and the Raspberry Pi Zero platform, we ensure that SmartCGMS remains scalable, efficient, and accessible, thereby fostering innovation and collaboration in the field of CGM and insulin pump systems.

The decision to develop our own emulator for Raspberry Pi Zero instead of utilizing existing solutions like QEMU or OVPSim was primarily motivated by our focus on CI/CD within the SmartCGMS framework. Moreover, it is essential for us to have full control over the emulation core and the measurement of safety metrics obtained through the emulation. By developing our emulator, we could seamlessly integrate it into our CI/CD pipeline, streamlining the testing and deployment processes for our software platform. This custom solution allows for greater flexibility and control over the emulation environment, ensuring compatibility with our specific development and deployment workflows. Additionally, developing our emulator enables us to optimize performance and efficiency tailored to the requirements of SmartCGMS, thereby enhancing the reliability and scalability of our CI/CD infrastructure. Overall, the decision to develop our emulator aligns with our commitment to maximizing automation and efficiency in the continuous integration and deployment of SmartCGMS, ultimately facilitating more rapid and reliable software releases.

A. Emulator design

The emulator, called *ZeroMate*, adopts a modular architecture design that allows the decomposition of the emulation core and optional components, such as the peripherals. Rather than supporting a wide variety of SoC's, our emulator focuses solely on the Raspberry Pi Zero platform (or, in fact, any platform with a similar design and base components). This allows us to design the emulator to fit the specific platform and its requirements. The single-platform focus allows for rapid development, effective testing, and deployment. Furthermore, a custom code base facilitates the integration of testing scenario definition, as well, as the consequent integration. As we intend to use the emulator in a CI/CD pipeline, it is crucial that it supports a headless execution with a clear interface and effective validation of testing scenarios.

On the architectural level, *ZeroMate* is decomposed to the following modules:

- Core – `ARM1176JZF_S` processor core, ALU, coprocessors, MMU and buses
- Peripherals – a set of memory-mapped peripherals, including e.g.,:
  - RAM
  - integrated peripheral components like GPIO controller, USART controller, I2C bus, and system timer
  - external peripherals chained with the integrated peripherals, e.g., a button, LED, 7-segment display, switch, OLED display, etc.
  - custom virtual peripherals – logic analyzer, debug monitor, etc.
- GUI – the graphical interface is an optional component that controls the emulation through a minimalistic (and also modular) user interface

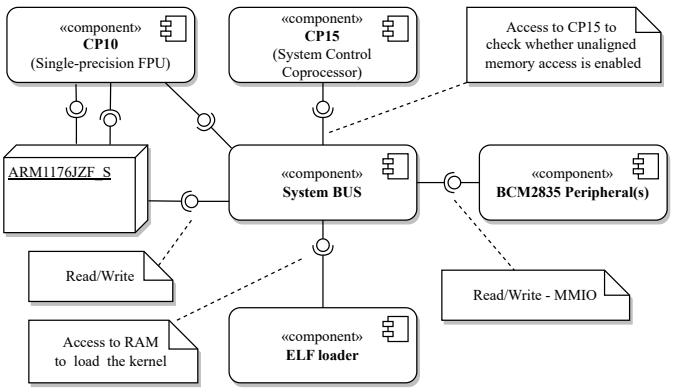

Fig. 1. ZeroMate component diagram

Figure 1 depicts the emulator architecture. All components are implemented as separate modules that are also interchangeable.

The testing environment must support scenario definition and behavioral validation. SmartCGMS is configured using a single `.ini` file, and thus, scenario definition is done using SmartCGMS chain configurations. SmartCGMS produces logs

of its execution. The emulator collects these logs and we validate the behavior by their close examination. The logging feature is switched to a monitor mode during emulation, unlike production mode, in which the device logs to its internal memory. The monitor mode transfers log outputs through a debug monitor directly to the host to make validation possible by the test environment.

B. CI/CD pipeline

With the emulator designed as such, we can integrate it into a CI/CD pipeline of SmartCGMS. We currently use Jenkins-based CI/CD internally to build and deploy SmartCGMS on all target platforms. We currently support the following platforms:

- x86, x86_64
  - MS Windows, Debian GNU/Linux, macOS
- ARM6, ARM7, ARM8
  - Raspberry Pi OS (incl. Zero)
  - Android

All supported platforms, excluding the Android build, pass through the testing task in the CI/CD pipeline. We use a custom tool to test the framework on a module level (e.g., if a filter fulfills its purpose) and also on the integration/regression level (e.g., if a treatment setup still performs within an acceptable error range).

We are currently working on a support for the ARM distribution of FreeRTOS-based SmartCGMS. As a part of the work, we consider including the emulator into the build pipeline.

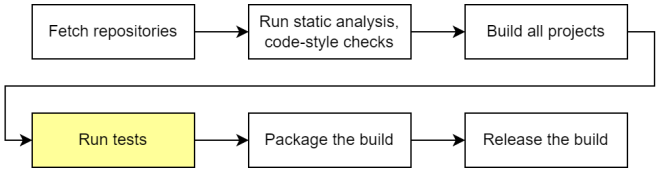

Fig. 2. CI/CD pipeline used to build and deploy SmartCGMS. This paper proposes an enhancement for the testing phase marked as yellow.

Figure 2 depicts the current CI/CD pipeline used to build and deploy SmartCGMS for all platforms. The step marked yellow represents a testing step (currently excluded for Android deployment). In this step, the ZeroMate emulator will be employed for the FreeRTOS-based build of SmartCGMS.

IV. ZeroMate Emulator

We implemented a modular emulator capable of emulating Raspberry Pi Zero basic feature set. The core part is able to emulate most of the ARM1176JZF_S processor features and a basic set of BCM2835 peripherals. It also features support for CP10 and CP15 coprocessors (single-precision floating point and system control coprocessors). The emulator implements basic MMU features, including paging and memory protection. It is also capable of emulating memory-mapped peripherals including BSC (I2C protocol), miniUART, etc.

The main implementation language is modern C++20. The decision to utilize the C++20 language for implementing the emulator stems from several compelling factors. First and foremost, C++20 is a modern and robust language with enhanced support for memory safety and focus on performance. Additionally, C++ ecosystem of libraries offers a vast amount of excellent libraries for e.g., creating a graphical user interface, logging, running tests, etc.

During development, we utilized libraries distributed exclusively under some form of open-source software licence to avoid licensing issues. Moreover, all of the chosen libraries have cross-platform support.

In terms of the emulator's booting process, the emulation skips a few steps that the real device performs. A real device powers on the GPU first, leaving the ARM core dormant [28]. The GPU performs first two stages of booting, eventually ending up with reading the kernel image from the SD card and loading it into the main memory. We skip these steps and let the emulation process start with the kernel image already loaded into the emulated main memory.

The emulator supports loading of ELF files. We decided for ELF file support for its convenience – most modern compilers have a default ELF output, and it can also contain debug information. Once instructed, the emulator unpacks and loads the ELF file into the main memory and starts the ARM core emulation. Moreover, the emulator is capable of loading additional ELF files for disassembly and debug information extraction, which allows for debugging of user-space processes with their symbols resolved.

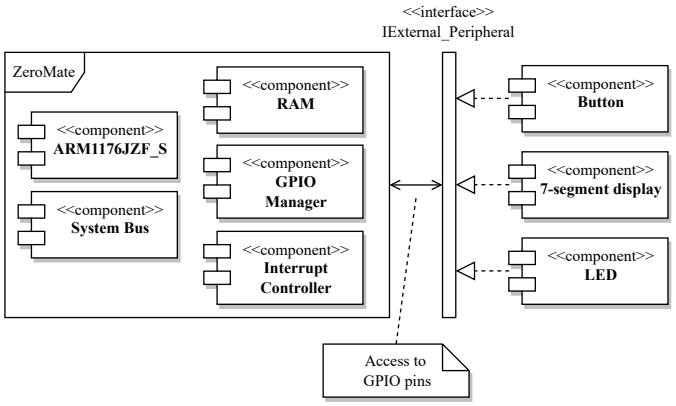

Fig. 3. Component diagram of peripherals and peripheral interface of ZeroMate

Figure 3 depicts the peripheral component diagram. ZeroMate differentiates between internal (BCM2835) and external peripherals. Internal peripherals are tightly bound to the emulator core – these include e.g., RAM, GPIO controller and interrupt controller. External peripherals are always chained through internal peripherals. For instance, a button component always connects to a specific pin of the GPIO controller. Every external peripheral is required to implement a mandatory interface `IExternal_Peripheral`, that is designed to be interoperable via the shared object interface (e.g., it does not use C++-specific data types and structures). As a result, potential developers can develop their own external peripherals

in any object-oriented compiled language capable of producing shared objects.

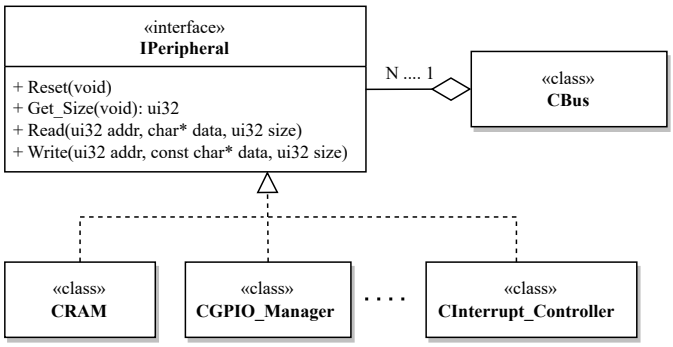

Fig. 4. Interface of an internal peripheral

Figure 4 depicts the internal peripheral interface. Each such internal peripheral is mapped to the memory space via the system bus. The system bus then manages a set of peripherals, i.e., their starting address and address space sizes in order to route memory access requests.

Since there is no implicit global clock source in a real device, we decided to synchronize all peripherals using CPU as a virtual clock source. As the main emulator thread steps through the execution instruction by instruction, we estimate the number of clock cycles it took to execute each instruction. After execution of each single instruction, all listeners are notified about the number of cycles that passed. Every peripheral (internal or external) can register as a clock source listener. This approach is used e.g., to accurately synchronize the miniUART or BSC (I2C) peripherals.

The GUI is implemented using an immediate mode GUI library *imgui*. Figure 5 depicts the base emulator window. Each peripheral (internal or external) has its own rendering method. Thus, it can render its user interface as required. For example, the RAM peripheral draws the memory dump, the miniUART peripheral renders a terminal interface with an input text field and an output area, and the button peripheral renders a single button. Every peripheral handles its own rendering cycle. Therefore, there is no need for an additional means of specifying the user interface. It is also important to note, that an external peripheral do not have to render an user interface, if it is not required to do so.

The emulator uses a custom logging system accessible from both the emulator core and all peripherals. Note that this logging system serves as means of communicating the emulation internal state rather than the emulated program state.

We also implemented two virtual peripherals to allow for easier debugging and examining the guest system state. First of all, we implemented a simple debug monitor peripheral. This is a virtual peripheral that does not exist in a real device and serves solely as a simple means of communicating string-based information to the user. It consists of a matrix of 80x25 characters. If mapped to a specific memory region, the guest system can simply write characters to the matrix in order to display them in the emulator GUI. This is somewhat similar

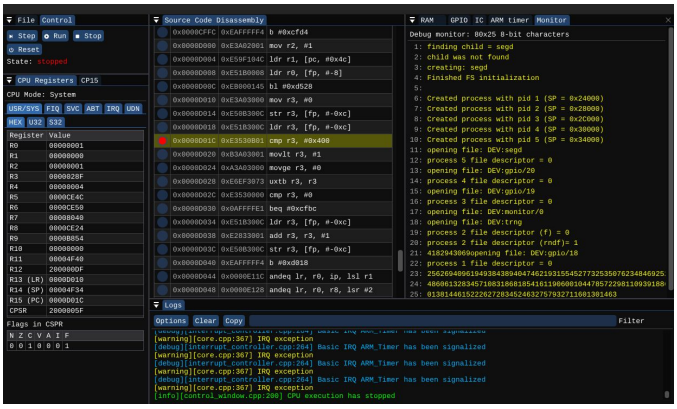

Fig. 5. Base emulator window

to standard framebuffer based character outputs (except that, e.g., there is no information about color).

The second virtual peripheral is the logic analyzer. Figure 6 depicts the component interface. We can connect the logical analyzer component to any GPIO pins and observe their state in time. The logical analyzer is synchronized to the CPU virtual clock (cycle) source. Thus, it is able to maintain synchronization during debugging.

Fig. 6. Logical analyzer virtual peripheral of the ZeroMate emulator

Figure 7 depicts the user interface of selected components – the OLED display, LED, toggle switch, button, 7-segment display (chained through an 8-bit shift register), and serial terminal (miniUART). They represent a basic set of components and also serve as an example for implementing more, possibly more sophisticated peripherals.

The function of each instruction is verified using a set of unit tests. These tests follow the functional description in the ARM1176JZF-S technical reference manual [29].

V. Experimental Setup

Since the FreeRTOS-based SmartCGMS distribution is still under development, we had to use another project to test the emulator core and implemented peripherals. For preliminary

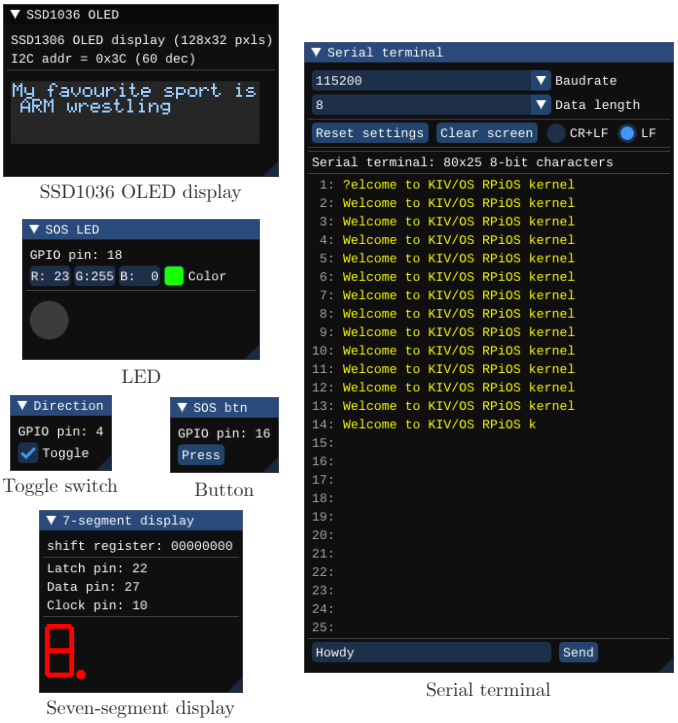

Fig. 7. A selection of external peripherals implemented in the ZeroMate emulator

testing, we used an educational real time operating system KIV-RTOS developed by us [30]. We have verified its function using real hardware. Therefore, we can validate its behaviour in an emulated environment.

KIV-RTOS is an educational operating system kernel and ecosystem developed solely for educational purposes. Additionally, it stresses out various features important for real-time systems. It supports user-space process isolation, paging, virtual filesystem, context switching with an EDF scheduler, and more. An experimental setup used to test the emulator included 5 user-space tasks, each of them having its own distinct purpose. The first task controls the 7-segment display (timer-based events with a firm deadline) – it increments or decrements the numbers based on switch state. The second task waits for changes on a tilt sensor (emulated by a switch) and writes out a log message to the logging pipe (fifth task). The third task waits for an SOS button press. After the button press has been detected, it sends a log message to the logging pipe (fifth task) and notifies the fourth task – the SOS task, which blinks the LED with SOS signal sequence (three short, three long and three short blinks, all based on timer events). The fifth task listens on a shared logging pipe and upon receiving a message, it forwards it to the miniUART to be displayed in the serial terminal.

First, we validated the behavior of the emulator against a real device. Figure 8 depicts a snapshot of KIV-RTOS running in an emulator. Apart from a noticeable drop in performance, which was expected, we did not observe any behavioral changes between the emulated and real device.

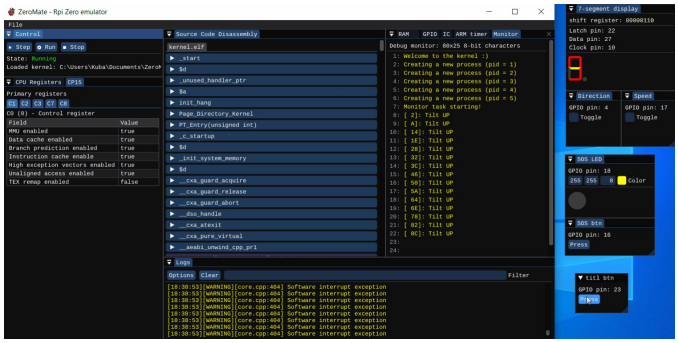

Fig. 8. KIV-RTOS kernel running in ZeroMate emulator

As the next step, we measured the emulation performance in terms of instructions emulated per second (IPS). We followed the instruction types set by the ARM architecture [31]. Table I depicts the measurement results for KIV-RTOS execution.

Instruction type	Relative occurrence [%]	Emulation speed [Minst/s]
Software interrupt	0.014	0.100
Block data transfer	0.436	0.833
Store return state	0.033	1.282
Return from exception	0.033	2.041
Co-processor register transfer	0.001	2.500
Single data transfer	**41.705**	**3.846**
Halfword data transfer	0.020	3.846
Multiply long	< 0.001	3.846
Change processor state	0.033	4.000
Data processing	**41.061**	**5.556**
Program status register transfer	0.007	5.882
Extend	5.388	8.333
Count leading zeros	< 0.001	8.333
Multiply	0.003	9.091
Branch	**9.940**	**10.000**
Signed multiply	0.002	10.000
Branch and exchange	1.214	16.949
NOP	0.100	31.250

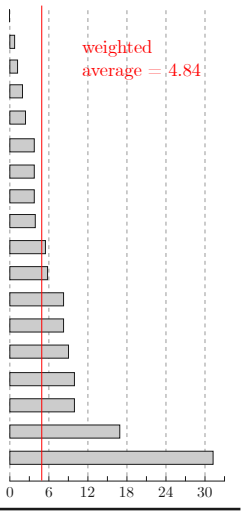

TABLE I
ZEROMATE EMULATION PERFORMANCE FOR VARIOUS INSTRUCTION TYPES

The experiment was conducted on the Lenovo ThinkPad P50 laptop, running the Windows 10 operating system. The laptop is equipped with an Intel(R) Core(TM) i7-6820HQ CPU running at 2.70GHz and 16GB of RAM. The measured and calculated emulation speed of KIV-RTOS kernel with a set of real-time tasks emulated by the ZeroMate emulator was estimated to 4.84 Minst/s.

VI. Conclusion and Future Work

We developed a robust and modular emulator of Raspberry Pi Zero environment. The emulator implements a basic feature set required to run SmartCGMS within a FreeRTOS distribution. We proposed an integration of our emulator into an existing CI/CD pipeline to mitigate possible portability-related. We also proposed the use of the emulator to simulate hardware

faults to eliminate the need for a real faulty hardware. The emulator thus serves as a digital twin of a real device.

The emulator itself is versatile enough it has already been used for educational purposes during the Operating Systems course at the University of West Bohemia, namely for emulation of an educational operating system – KIV-RTOS. Nonetheless, its main purpose lies in software testing within a CI/CD pipeline.

Future work includes the final integration with the FreeRTOS-based SmartCGMS distribution. For our future research, we must define a set of scenarios, that reflects real-world situations. For example, apart from the non-faulty situations, we must also define a scenario set including faulty peripherals. This may include malfunctioning insulin reservoir dosing servo, noisy feedback measurements, etc.

The emulator source code and binary releases are available at https://github.com/SmartCGMS/ZeroMate.

Acknowledgement

References

[1] J. Zander, I. Schieferdecker, and P. Mosterman, *Model-Based Testing for Embedded Systems*, ser. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, 2017. [Online]. Available: https://books.google.cz/books?id=6lDRBQAAQBAJ

[2] J. E. Hall and M. E. Hall, *Guyton and Hall Textbook of Medical Physiology E-Book: Guyton and Hall Textbook of Medical Physiology E-Book*. Elsevier Health Sciences, 2020.

[3] R. Zhao, Z. Lu, J. Yang, L. Zhang, Y. Li, and X. Zhang, "Drug delivery system in the treatment of diabetes mellitus," *Frontiers in bioengineering and biotechnology*, vol. 8, p. 880, 2020.

[4] B. Bode, A. King, D. Russell-Jones, and L. K. Billings, "Leveraging advances in diabetes technologies in primary care: a narrative review," *Annals of medicine*, vol. 53, no. 1, pp. 805–816, 2021.

[5] T. Koutny and M. Ubl, "SmartCGMS as a Testbed for a Blood-Glucose Level Prediction and/or Control Challenge with (an FDA-Accepted) Diabetic Patient Simulation," *Procedia Computer Science*, vol. 177, pp. 354–362, 2020, the 10th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2020). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920323164

[6] T. Koutny and D. Siroky, "Analyzing energy requirements of meta-differential evolution for future wearable medical devices," in *World Congress on Medical Physics and Biomedical Engineering 2018: June 3-8, 2018, Prague, Czech Republic (Vol. 3)*. Springer, 2019, pp. 249–252.

[7] M. Otta, "Towards a health software supporting platform for wearable devices," *Procedia Computer Science*, vol. 210, pp. 112–115, 2022, the 12th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2022). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050922015836

[8] D. Hemapriya, P. Viswanath, V. M. Mithra, S. Nagalakshmi, and G. Umarani, "Wearable medical devices — design challenges and issues," in *2017 International Conference on Innovations in Green Energy and Healthcare Technologies (IGEHT)*, 2017, pp. 1–6.

[9] Z. Wu, M. Liu, L. Qin, S. Ye, and H. Chen, "Wearable Medical Devices' MCU Selection Analysis Based on the ARM Cortex-MO+ Architecture," *Zhongguo yi Liao qi xie za zhi= Chinese Journal of Medical Instrumentation*, vol. 39, no. 3, pp. 192–196, 2015.

[10] M. Howard and R. Bruce Irvin, "ESP32: QEMU Emulation Within a Docker Container," in *Proceedings of the Future Technologies Conference*. Springer, 2023, pp. 63–80.

[11] T. Buchert, L. Nussbaum, and J. Gustedt, "Accurate emulation of CPU performance," in *Euro-Par 2010 Parallel Processing Workshops: HeteroPar, HPCC, HiBB, CoreGrid, UCHPC, HPCF, PROPER, CCPI, VHPC, Ischia, Italy, August 31–September 3, 2010, Revised Selected Papers 16*. Springer, 2011, pp. 5–12.

[12] X. Dai, S. Zhao, B. Lesage, and I. Bate, "Using digital twins in the development of complex dependable real-time embedded systems," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2022, pp. 37–53.

[13] T. Sun, X. He, and Z. Li, "Digital twin in healthcare: Recent updates and challenges," *Digital Health*, vol. 9, p. 20552076221149651, 2023.

[14] T. Markwirth, R. Jancke, and C. Sohrmann, "Dynamic fault injection into digital twins of safety-critical systems," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 446–450.

[15] Y. Zhang, R. Jetley, P. L. Jones, and A. Ray, "Generic safety requirements for developing safe insulin pump software," *Journal of diabetes science and technology*, vol. 5, no. 6, pp. 1403–1419, 2011.

[16] K. Lee, W. Han, J. Lee, H. S. Chwa, and I. Shin, "Fast and accurate cycle estimation through hybrid instruction set simulation for embedded systems," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 370–370.

[17] F. Bellard, "QEMU, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555.

[18] Imperas, "OVPsim Simulator," 2023, accessed on 6. 5. 2024. [Online]. Available: https://www.ovpworld.org/

[19] A. Liu, "A Benchmark and Evaluation of Imperas OVPSim Virtual Platform Tool Using RISC-V Processors," 2022, accessed on 6. 5. 2024.

[20] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[21] "CPUlator Computer System Simulator," accessed on 8. 5. 2024. [Online]. Available: https://cpulator.01xz.net/

[22] "ARMSim# version 2.1," accessed on 9. 5. 2024. [Online]. Available: https://webhome.cs.uvic.ca/~nigelh/ARMSim-V2.1/index.html

[23] M. Talekar and V. K. Harpale, "CI-CD Workflow For Embedded System Design," in *2023 7th International Conference On Computing, Communication, Control And Automation (ICCUBEA)*, 2023, pp. 1–3.

[24] N. A. Visnevski, "A Novel, Model-Based, Specification-Driven Embedded Software Integration Platform," in *2021 IEEE Aerospace Conference (50100)*, 2021, pp. 1–18.

[25] B. El Khalyly, A. Belangour, M. Banane, and A. Erraissi, "A new metamodel approach of CI/CD applied to Internet of Things Ecosystem," in *2020 IEEE 2nd International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS)*, 2020, pp. 1–6.

[26] A. Barbie, W. Hasselbring, N. Pech, S. Sommer, S. Flögel, and F. Wenzhöfer, "Prototyping Autonomous Robotic Networks on Different Layers of RAMI 4.0 with Digital Twins," in *2020 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, 2020, pp. 1–6.

[27] E. Glaessgen and D. Stargel, "The digital twin paradigm for future NASA and US Air Force vehicles," in *53rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference 20th AIAA/ASME/AHS adaptive structures conference 14th AIAA*, 2012, p. 1818.

[28] H. D. Ghael, L. Solanki, and G. Sahu, "A review paper on Raspberry Pi and its applications," *International Journal of Advances in Engineering and Management (IJAEM)*, vol. 2, no. 12, p. 4, 2020.

[29] "ARM1176JZF-S Technical Reference Manual," accessed on 10. 5. 2024. [Online]. Available: https://developer.arm.com/documentation/ddi0301/h/?lang=en

[30] "KIV-RTOS," accessed on 7. 5. 2024. [Online]. Available: https://github.com/MartinUbl/KIV-RTOS

[31] J. Yiu, *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Newnes, 2013.