# Cortex-M Simulator

Tomáš Jakubík
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies
Technical University of Liberec
tomas.jakubik@tul.cz

MCD Application Team
STMicroelectronics Sophia Antipolis
Valbonne
France
st.com

***Abstract*—This paper describes a project of simulator of Cortex-M microprocessors. This project builds on top of Unicorn Engine, which is used to simulate the ARM core. Benefit of this project is capability to load a production firmware and to replace microprocessor peripherals. The same firmware can be executed on a physical board and the same firmware can be simulated. This allows quick continuous integration and testing for development of embedded firmware.**

***Keywords*—cortex-m; simulation; embedded electronics; continuous integration**

## I. Introduction

Modern software development uses principles known as Continuous Integration (CI) [1]. The software being developed is built under controlled environment and tested with unit and integration tests. The results are immediately used by the developers to prevent regression and to continue the development. More continuous operations usually follow, but they are out of scope of this article.

These CI techniques are not common in Micro Controller Unit (MCU) embedded firmware development [2]. Some unit tests can be done, but many integration tests require the hardware board where the MCU is placed. This simulator can provide new options in this area.

There is the *Unicorn Engine* project based on QEMU [3] which, apart from other things, can simulate the ARM core. This *CortexM_Simulator* project builds on top of Unicorn and adds the option to emulate peripherals and the rest of the real Cortex-M MCU. The same binary firmware which is simulated can later be sent to the customer.

The simulation can be advantageous in several situations. It may be used for fuzzing and simulated many times with random inputs to cover as many potential bugs as possible.

Another use might be an IOT device which sleeps for hours between each activity. A full integration test of this device might take days, but days of simulation can take only moments of real time.

Some applications require testing many devices communicating together. That would be costly, because of a lot of hardware. In simulation only a bigger CI server is needed and it can be shared or rented.

And as a last point, it can also be more practical for an engineer to stay in his office and use a simulator instead of going into the lab and setting up proper test conditions.

ISBN 978-80-261-0892-4, ©University of West Bohemia, 2020

### A. State of the Art

There are Open Virtual Platforms (OVP) [4] established by Imperas. It is more mature solution, but it follows a slightly different philosophy. There are also QEMU and SystemC based emulators [5]. The emphasis of these projects is to provide simulation of the entire hardware. From the point of firmware developer, it can be difficult to recreate all the hardware virtually and it can be difficult to connect the simulation to other components on the host PC [6]. Simulator described in this article is focused on replacing Hardware Abstraction Layer (HAL) functions to simplify the process and allow reusing the simulation for more MCUs.

There are also proprietary instruction set simulators from Keil or IAR, yet these don't handle peripherals and cannot simulate production firmware.

There is also another class of cycle accurate simulators [7]. They are usable to prevent timing atacks, but less comfortable in validation of the entire system.

## II. Principles of the Simulator

Basic idea of this simulator is either to emulate MCU hardware as is or to replace parts of the firmware which are using it.

As an example, firmware sends a byte of data to a serial communication line via *UART* peripheral. This is done by writing the byte into the *TDR* register. We can either emulate the *TDR* register and log the value each time it is written or we can replace the *UART* writing function from the firmware.

Emulating peripheral hardware registers might be closer to reality, but replacing HAL of the firmware is much easier. It would be very difficult or impossible to emulate the peripheral exactly which implies that the HAL has to be validated separately on a real hardware. For modern MCUs, the manufacturer already provides extensive HAL which is validated by its many customers as well as the manufacturer itself [8]. It is not necessary to repeat the validation process again and it is not necessary to include the HAL code into the simulation. Replacing the HAL interface has also advantages of portability, because different processor families have different registers, but the same HAL interface.

This simulator provides tools for both methods.

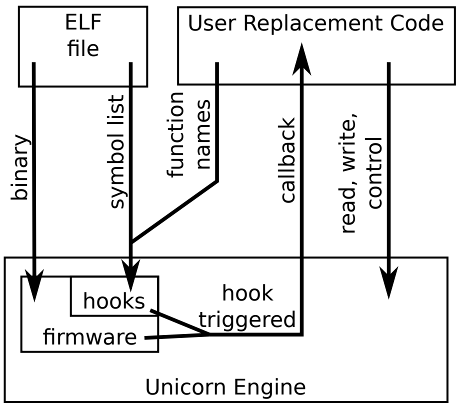

Fig. 1. Replacing Function

### A. Replacing a Hardware Register

Unicorn Emulator has the ability to add hooks for write or read of a particular memory. It will run the firmware instructions and when there is an access to the register address, the simulation is stopped. Simulator user will then have a callback to input/output data, manage interrupts or change the simulation state. When done, the simulation resumes.

In case for the *TDR* register, it is very simple, but common peripherals have tens of registers and it gets complicated quickly.

### B. Replacing a Firmware Function

The firmware being simulated is the same binary which runs on the MCU. Replacing a function doesn't mean changing the firmware. The simulator is able to intercept the function when the firmware is running, emulate the behavior instead of the firmware and skip the original firmware function.

There are several steps to replace a function as depicted on Fig.1. Similar process can be used to replace interrupt handlers or global variables.

*1) Matching the Function Address:* As a first step and before running the simulation, the replaced function needs to be matched in the simulated firmware. Simulator obtains the firmware as an *ELF* file. It uses *arm-none-eabi* tools to extract a table of symbols and addresses. These symbols are compared to a name of the function which needs to be replaced. This way the function can move in the firmware during development and it is still found. At least as long as the function interface stays the same.

When the function address is known, it can be added as a hook to the Unicorn Engine. In this case, the simulation is stopped as soon as Program Counter (PC) loads the function address when the firmware tries to branch into the replaced function.

*2) Getting Function Parameters:* Now we have the simulator stopped at the beginning of the replaced function. The next step is to get function parameters.

ARM has standardized the way of passing function parameters [9]. In case of extreme link-time optimization, the linker might break the standard, but in most cases, linker will keep the compiled interfaces.

The replacement callback needs to know the order and types of parameters and provide them to the simulator. The parameters are parsed from the *Rn* registers or MCU stack. When the callback finishes processing, it can return a value back to the firmware.

Regular parameters are passed as 4 byte items in core registers, difficulties begin with passing and returning structures. Extra care needs to be put into the structure format and alignment. The alignment is usually size of the biggest element. When the alignment fits, it is possible to pass whole structures between the arm compiled firmware and the *x86_64* compiled simulator callback.

*3) Skipping the Original Function:* At the end of the callback, the Link Register (LR) can be copied to PC register and the simulation is resumed. This is to skip the firmware function and continue from the point where the function returns.

### C. Interrupts

The interrupt controller *NVIC*, used in the Cortex-M processors, is emulated in the simulator library.

When interrupt is set to pending, the simulator checks priorities and stops the simulation. The firmware code can be stopped at any moment as on a real MCU. This allows to catch potential deadlocks, but the differences in timing might reveal different problems than which appear on a real MCU.

The simulator pushes context on stack, looks for the interrupt vector at the beginning of the firmware and switches PC register to the interrupt routine. There is also a special *EXC_RETURN* value in the LR register as in a real MCU [10]. It is used at the end of the interrupt routine to stop the simulation again and return PC register to the main code or resume lower priority interrupt.

### D. Replaced Interrupts

Another option is to use an interrupt replacement, skip the HAL code and call the application handler immediately. Continuing the previous example, since we skipped the HAL code for *UART* on the way down to the hardware to transmit, we can also skip the HAL code on the way up when receiving.

When the interrupt is triggered, the simulator will check that there is a replacement interrupt and will not use the interrupt vector. Instead, user replacement will get a callback to prepare function parameters. Then an address of some higher level firmware function is loaded into PC register instead of the interrupt vector. Another user callback is called when the firmware function returns to get the return value.

This mechanism was implemented as a complement to replacement functions because of two reasons. The HAL code which processes the interrupt wasn't initialized and could fail instead of calling the higher level function. Second reason is that interrupt handlers usually immediately start writing into hardware registers to clear interrupt flags. All these registers would have to be emulated just to pass program into a higher level function.

The function address and utilities to work with parameters are similar to replacing a function.

### *E. Advanced Features*

*1) Global Interrupts Disable:* The replacements work only for functions which are listed in the *ELF* file. Inline functions are directly copied into other code and cannot be replaced. Unfortunately, some hardware oriented functions are made inline to optimize the code.

Standard ARM *CMSIS* functions are usually put into *init* functions and can be replaced together with the *init* function. One exception are functions *__enable_irq()*, its disable counterpart and *PRIMASK* control. These functions are usually used in a macro to disable and enable interrupts around critical sections. It is used instead of mutexes in a bare metal firmware.

To solve this common issue, the simulator has an option to scan the disassembly of the firmware and look for all instructions which work with *PRIMASK*. Bit *PRIMASK*, stored in *CONTROL* core register, disables interrupts globally. It can be controlled only by 4 different instructions. All uses of these instructions are found and a hook is added for each one. The simulator then automatically enables and disables interrupts.

*2) Bootloaders:* One element which is usually not tested properly and which takes a long time to test is a bootloader. In case of Over The Air (OTA) update, the firmware has to be completely functional, including the communication stack. Any mistake can prevent you from fixing that mistake.

The replacements in the simulator are separated to modules. Each module can match the names from a different *ELF* file. One for bootloader and another for application. The modules can deactivate RAM based replacements when the MCU switches between application and bootloader.

*3) C++ Firmware:* Some developers are starting to use C++ for embedded firmware development. Using C++ linker mangles the function names to allow for overloading and all the C++ goodies. The *arm-none-eabi* tools, used to parse the *ELF* file, can demangle C++ names back into useful names. It is possible, given correct function parameters, to match the demangled name and replace C++ function or method.

Methods can be replaced easily with object pointer as a first parameter, but the object pointer can be useless. Compiler can put data into C++ object in any way and it might be impossible to decipher the correct data structure.

## III. Using the Library to Create a Simulator

The simulator library needs to be compiled with user code into a C++ simulator. Lot of code specific to hardware and firmware needs to be added, but with some precautions much of it can be reusable for other projects.

### *A. Recommended Layout*

A good practice is to design only the replacements and keep the testing to an external program.

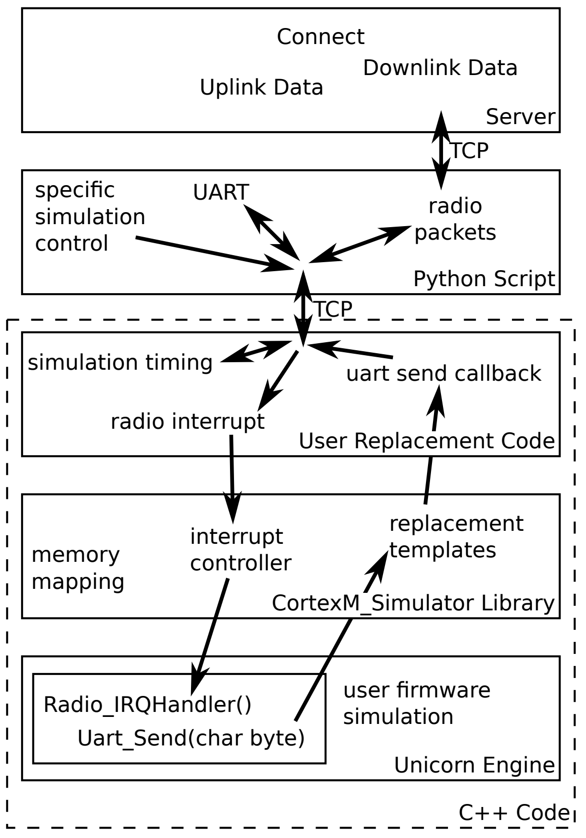

Fig. 2. Recommended Layout

Let's imagine we want to test a brand new IOT device. The recommended architecture is depicted on Fig.2.

The Unicorn Engine is the lowest layer of the simulator application. It simulates the firmware binary. Above it, there is the simulator library. It provides all utilities needed to replace the peripherals and control the simulation easily. On top of the C++ application, there are the replacements specific to the hardware and firmware being simulated.

The replacements report all necessary information via TCP connection to an external testing suite. In this case, there is a Python script which logs *UART* trace, controls the simulation and forwards the radio packets to an IOT server. This script is also responsible for setup of the testing scenario and for evaluating the test results.

For a simple device, the test logic could be included in the C++ simulator, but separating the test outside has advantages. Perhaps it would be easier to write the test in *Go* and perhaps we need to test two devices communicating with each other instead of server. All could be done without changing the simulator itself.

### *B. Writing Replacements*

It is not easy to create all replacements necessary to simulate a new device. The way to start is to find which peripherals are used by the device and which are important to test the device. Peripherals which are used by the firmware, but are not necessary for the simulation can be replaced with dummy functions.

A very important part is usually the timing. The timing should be kept in mind from the beginning, since it flows through most parts of the simulation and the test. Currently, time is not handled by the simulator and needs to be managed when replacing Real Time Clock (RTC) peripheral or other source of time for the firmware.

The replacements are usually created by inheriting C++ classes. For simple replacements, there are method templates which automatically deduce the parameters of simple functions, but inheriting manually has more capabilities. User code needs to complete the inherited class by providing virtual callbacks. These callbacks are used when the replaced variable, register or function are accessed. Part of these classes are useful methods to read or write the replaced variable, to work with function parameters or object which can control the simulation.

During the development, it is common that the simulated firmware will run for a while and then it tries to access a piece of hardware which is not yet emulated. The simulator will end with an error, giving user the information about program counter and address that the firmware tried to access. Next step is to look into the firmware disassembly and MCU user manual and create new replacement for this hardware.

## IV. Conclusion

We were able to successfully simulate an IOT device and do a simple test of its functionality. This test was done in a few seconds which is much quicker than the same binary firmware running on a real hardware. Real device spends several minutes waiting to comply with duty cycle limitation.

We didn't compare the speed nor accuracy with state of the art simulators as neither of these parameters are important to the purpose of this project. The accuracy is defined by QEMU and by the HAL replacements written by the user. The purpose is to provide easy to use tool for firmware developers and add another layer of validation between unit tests and a real hardware.

We cannot yet recommend fully relying on this simulator in production. In the end, the results should be validated on a physical hardware, but it could serve as a quick integration test for firmware developers. It is a promising platform to start adding CI features to embedded development.

### A. Future Development

The simulator is now usable to test a simple device, but each device needs a lot of custom code. There are possibilities to simplify the custom code and add common functionality to the library. Some improvements are simple, such as emulating *BASEPRI* register, but some are more difficult.

*1) Operating Systems and Floating Point:* The simulator cannot run operating systems or firmware using floating point unit. The limitations come from the interrupt controller and its stack operations. Features necessary for the correct behavior will come in next Unicorn release, but were not available in time of writing this article.

*2) Simplify the Interface:* There are C++ templates used to ease writing of the replacements, but they are quite experimental. There is a lot of space to improve the C++ template magic.

*3) Universal Communication Module:* Since most applications will use some kind of connection to an outside test suite, it might be beneficial to add an optional universal communication module. The module would set up a TCP connection and have common communication protocol.

*4) Universal Timing Module:* A lot of simulation time can be saved, when the low-power device is sleeping. The simulator can have a virtual time and advance it forward each time the device tries to sleep. Problems are hidden in the events that might happen during the advanced period. A lot of development is needed to solve problems with interfacing the generalized test suite outside the simulator to synchronize the events.

*5) GDB connection:* The firmware being simulated can misbehave when the replacements are not developed properly. Connection to GDB would help to debug the simulator as well as the simulated firmware. At least one project for Unicorn-GDB connection already exists [11].

## Acknowledgment

This project was started during an internship at STMicroelectronics which supported the project through the beginnings.

Initial impulse for the development of this simulator was a BasicMAC project [12]. It uses the Unicorn Engine to simulate firmware in a simpler but less reusable way.

## References

[1] M. SHAHIN, M. A. BABAR, and L. ZHU, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access*, 2017.

[2] L. E. Lwakatare, T. Karvonen, T. Sauvola, P. Kuvaja, H. H. Olsson, J. Bosch, and M. Oivo, "Towards DevOps in the Embedded Systems Domain: Why is It So Hard?" in *49th Hawaii International Conference on System Sciences (HICSS)*, 2016.

[3] N. A. Quynh. Unicorn Engine. [Online]. Available: https://www.unicorn-engine.org/

[4] Open Virtual Platforms. Imperas. [Online]. Available: http://www.ovpworld.org/

[5] G. Delbergue, M. Burton, F. Konrad, B. L. Gal, and C. Jego, "QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[6] *OVP Peripheral Modeling Guide*, 1.8.6 ed., Imperas, 2019.

[7] J. Bauer and F. Freiling, "Towards cycle-accurate emulation of cortex-m code to detect timing side channels," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*, 2016.

[8] *UM1884 Description of STM32L4/L4+ HAL and low-layer drivers*, 7th ed., STMicroelectronics, 2017.

[9] *Procedure Call Standard for the Arm Architecture*, Release 2019Q1.1 ed., Arm Ltd, 2019.

[10] *Cortex-M4 Devices Generic User Guide*, B ed., Arm Ltd, 2011.

[11] I. Hallé. unicorn-gdbserver. [Online]. Available: https://github.com/isra17/unicorn-gdbserver

[12] Basic MAC Repository. Semtech. [Online]. Available: https://github.com/lorabasics/basicmac