Rick Ramirez

A BRIEF EXPLORATION INTO TOCK OS'S PERFORMANCE

1. Introduction

This project focuses in on Tock. Tock is an embedded operating system designed for running multiple concurrent, mutually distrustful applications on low-memory and low-power microcontrollers[5]. Tock is different from other embedded operating systems in the fact that it is implemented in Rust while supporting userspace applications written in multiple languages (Mainly C and Rust). With his unique implementation it would be useful to see how Tock stacks up in performance to other more traditional embedded operating systems.

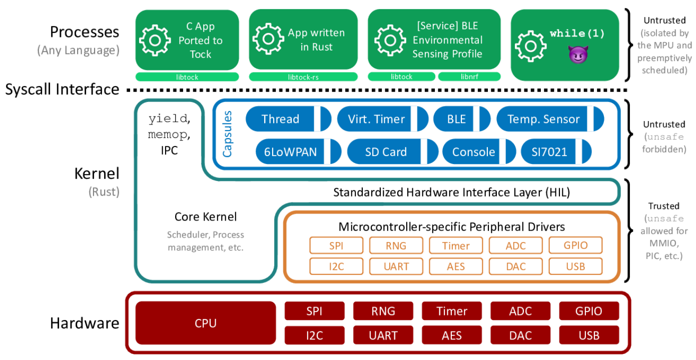

Figure 1. Tock's architecture [7]

2. Tock Components

The Tock Architecture is seen in Figure 1. Tock code is split into 3 distinct areas: core kernel, capsules, and processes. Rust is a type-safe language and is used to write both the kernel and capsules. A capsule is simply an instance of a Rust struct and associated functions. Capsules can directly interact with each other, accessing fields and calling functions within other capsules (assuming said capsule has exposed certain fields and functions to other capsules). Capsules are initialized by trusted configuration code and run inside the kernel in privileged hardware mode. This is not problematic because of Rust! A buggy capsule can only access resources explicitly given to it and only in ways permitted by the interfaces that said resource exposes. The one downside of capsules is that they are scheduled in the same thread event loop as the kernel and must be trusted for liveness. If a capsule somehow does panic or does not yield, the system will have to be rebooted to recover. A benefit of using Rust code for these two portions of Tock is that Rust code cannot use memory differently than intended (preventing bad things from happening). However there are certain tasks/actions a kernel must take that would violate Rust's safety features and Tock compensates for this by allowing the core kernel to use unsafe Rust (which is what you would think it is: Rust that doesn't have Rust's safety properties). This of course defeats the purpose of using Rust in the first place so unsafe Rust code is kept to a minimum and is only used in the core kernel. Capsules cannot use unsafe Rust (thus do not have to be trusted)and to top it all off processes are in userland and can be written in any language. This gives incentive to developers to hop on the platform since they can directly port their existing C apps onto Tock.

3. The Question Investigated

C has been the defacto language of choice when it comes to embedded operating systems. Historically writing operating systems in C comes with both benefits and drawbacks. C is fast and memory efficient (”closer to the hardware” than other language) which is a must when dealing with devices that have limited resources such as microcontrollers. C is also prone to bugs and undefined behavior. C puts a lot of trust on the developer, it will do what it is told to do even if that thing is bad/will cause problems. On the flip side Rust is a type-safe language that can promise just as much performance as C with some baked in protections and fewer to no bugs/undefined behaviors. Rust will quite literally not allow developers to write bad code as it will get caught and errors will be thrown. With all these benefits it only makes sense to explore writing embedded operating systems in Rust. The question then becomes how does Tock compare to other embedded operating systems in terms of performance?

4. Practical Significance

Embedded devices are (if not already) taking over the world. We see these devices everywhere. In our planes, cars, toys, kitchen appliances, watches, etc. In the every advancing world of technology we creep closer and closer to having embedded devices be in everything and bowing down to our robot overlords. Before that time comes it would be useful to test out different languages to create these embedded operating systems and see what kind of benefit we gain from them.

5. Methodology

The following two pieces of equipment were used in conducting the experiments: A Saleae Logic Analyzer (Logic Pro 8) and an Arduino Nano 33 BLE Sense.

The basic idea behind the experiments is as follows: Connect the logic analyzer to one of the GPIO pins on the board (whichever one you want to use as your “timing pin”). Turn on said GPIO pin, *Run some code to time*, Turn off said GPIO pin after the last line of your code and use a logic analyzer to get precise timings of when the pin went high then low so you’re able to detect how long something in between took.

Three different experiments were ran on three different platforms. The experiments were testing: A GPIO system call (set high and set low), a delay function call (100ms delay), and a toy application that used both GPIO system calls and delay calls (a 4-bit binary counter with LEDs). The three platforms tested on were Tock OS (version 1.6), Mbed (version 5.15), and Riot OS (version 2021.01). All tests were ran on an Arduino Nano 33 BLE Sense(from this point forward refered to as just the Nano). In order to load Tock and Riot onto the Nano a virtual machine (Virtual Box 6.1) running Ubuntu 20.04.2 LTS was used. Mbed was loaded onto the Nano from the Arduino IDE in Windows.

A basic workflow to recreate these experiments is as follows: Connect the logic analyzer to one of the GPIO pins on the Nano and designate it as your "timing pin", load your desired OS onto the Nano (Mbed/Tock/Riot/Other OS you want to test), load your application that runs some piece of code you want to test in between setting your timing pin high then low, finally collect the data/waveforms from the logic analyzer software for later analysis. Repeat for each experiment you want to run and each OS you want to test.

6. Results

The following section details the results from the experiments ran as described above. A quick side note is that all the numbers here are presented with my timing mechanism overhead subtracted. When A GPIO pin is set high and then low to time some piece of code you are also getting the time it takes to turn the pin high and low added in. To account for this a baseline reading of just turning a GPIO pin high then low was taken and subtracted from all the subsequent experiments to remove that overhead.

Table 1
Timing for setting a GPIO pin high/low in microseconds

OS	Set High	Set Low
Mbed	1.08 μs	1.142 μs
Tock	26.764 μs	26.814 μs
Riot	0.22 μs	0.22 μs

Table 2
Timing for a 100 MS Delay in milliseconds

Mbed	100.003 ms
Tock	97.867 ms
Riot	99.872 ms

Table 3

Timing for 4 bit binary counter with Delay in milliseconds

Mbed	799.384 ms
Tock	783.546 ms
Riot	799.294 ms

Table 4

Timing for 4 bit binary counter without Delay in microseconds

Mbed	36.768 μs
Tock	755.536 μs
Riot	7.846 μs

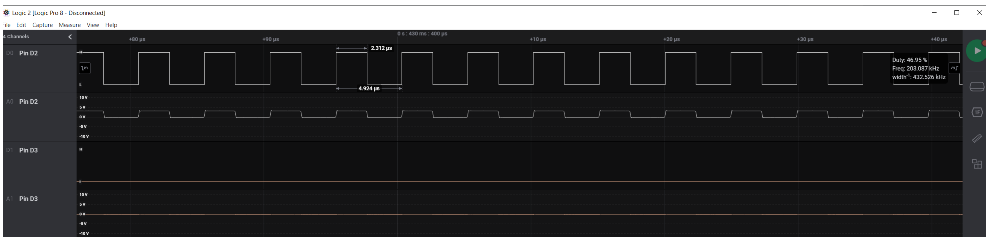

Figure 2. Mbed's waveform for the set GPIO pin high

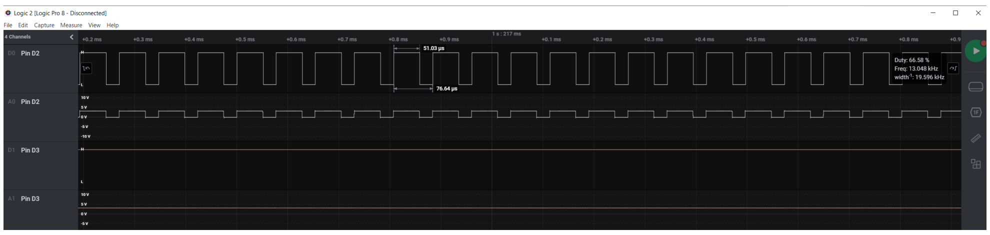

Figure 3. Tock's waveform for the set GPIO pin high

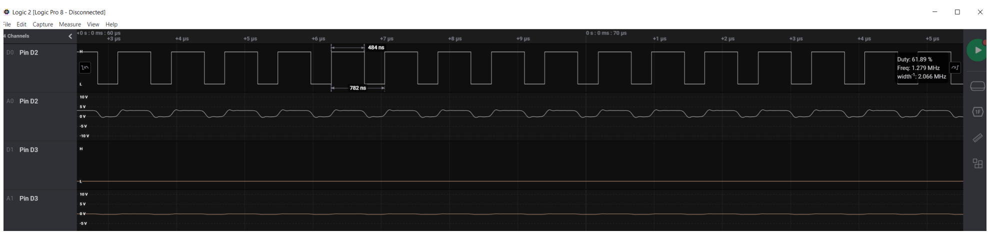

Figure 4. Riot's waveform for the set GPIO pin high

Waveforms for the other experiments are located in the appendix but can also be found at the following link: `https://github.com/rickr04/Comp730_Tock/tree/main/WaveFormPictures`

7. Discussion

As the results show Tock is slower by quite a big margin when compared to other embedded operating systems. In the GPIO set high/low test Tock is slower by about a factor of 24 when compared to Mbed and slower by about a factor of 121 when compared to Riot. What is more interesting is the fact that in the Delay test Tock finished faster than the other two OSs. This issue is further exacerbated when we look at the 4 bit counter with delay (which uses mutiple delay calls) as Tock finishes about 13 ms faster than the other two OSs which is cause for further research as the number is uncomfortably high. The hope was that Tock would perform just as if not better than other conventional embedded OSs thus giving more validity to Rust as a language for embedded devices but these results show the opposite.

Despite the loss of performance, Tock being written in Rust should not be taken as the sole reason for lacking behind. There are a lot of factors at play here and it is my opinion that what we see is due in part by differing philosophies and design choices. Performance is obviously not Tock's top priority, rather (as stated in the beginning) Tock is an embedded operating system designed for running multiple concurrent, mutually distrustful applications. This goal is different from what Mbed seeks to achieve which is different from what Riot wants to get done which is going to be different from what some other embedded operating system is trying to do and etc etc. Regardless of the purpose of these embedded operating systems they are all embedded operating systems and seeing the differences in performance compared to what their design was meant to do is interesting.

Checkout: `https://github.com/rickr04/Comp730_Tock` for more step by step instructions on how to run tests and get different OSs onto the Nano. This repo is also host to all data/code/images used throughout this report. The timing files produced by the logic analyzer are also located here and are free for you to download and view yourself.

8. Future And Related Work

Future work on Tock should be conducted in order to see the progression of this embedded operating system. Tock is currently undergoing an overhaul with its syscall interface as it transitions into version 2.0. Running these test again plus a few more would be beneficial for both end users, the developers of Tock, and a curious college graduate who will one day come back to this project.

Perhaps also testing an embedded operating system in Rust which is more tailored to performance would also be a good idea for future work because as of now it seems Rust is one of the limiting factors when it comes to speed/performance of this OS (at least in my very small sample size).

As far as related work is concerned there is a lot out there about various embedded operating systems such as: SOS[2], TinyOS [4], TOSThreads[3], and Contiki[1]. There is also a nice Master thesis [6] dealing with Tock and power efficiency.

Acknowledgements

The research presented in this report was made possible with the help of the Tock slack group, Don Porter, Stefan Jeglinski, and a lot of coffee.

References

[1] Dunkels A., Grönvall B., Voigt T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, 2004.

[2] Han C., Kumar R., Shea R., Kohler E., Srivastava M.: A dynamic operating system for sensor nodes. In: *MobiSys '05*. 2005.

[3] Klues K., Liang C., Paek J., Musaloiu-Elefteri R., Levis P., Terzis A., Govindan R.: TOSThreads: thread-safe and non-invasive preemption in TinyOS. In: *SenSys '09*. 2009.

[4] Levis P.: TinyOS: An Open Operating System for Wireless Sensor Networks (Invited Seminar). In: *7th International Conference on Mobile Data Management (MDM06)*, 2006. URL `http://dx.doi.org/10.1109/mdm.2006.151`.

[5] Levy A., Campbell B., Ghena B., Giffin D.B., Pannuto P., Dutta P., Levis P.: Multiprogramming a 64kB Computer Safely and Efficiently. In: *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[6] Nilsson F., Lund S.: Abstraction Layers and Energy Efficiency in TockOS, a Rust-based Runtime for the Internet of Things. 2017.

[7] Tock: *https://github.com/tock/tock/blob/master/doc/Overview.md.*

9. Appendix: Waveforms

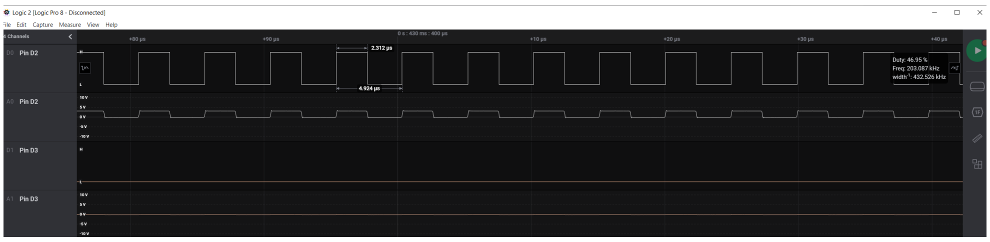

Figure 5. Mbed's waveform for the set GPIO pin low

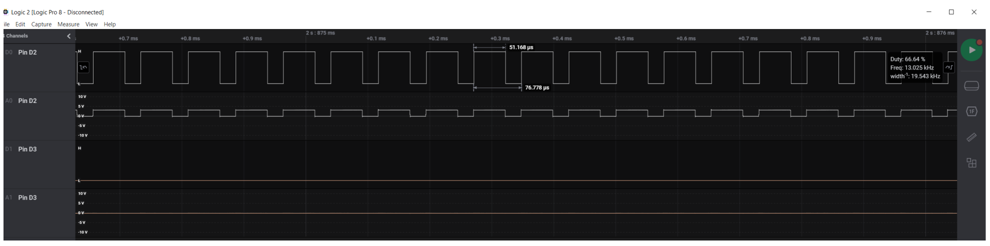

Figure 6. Tock's waveform for the set GPIO pin low

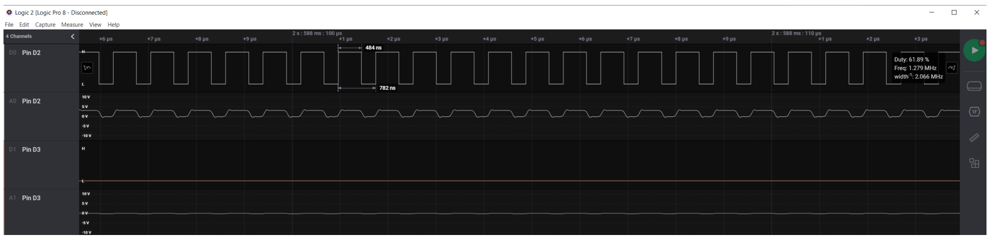

Figure 7. Riot's waveform for the set GPIO pin low

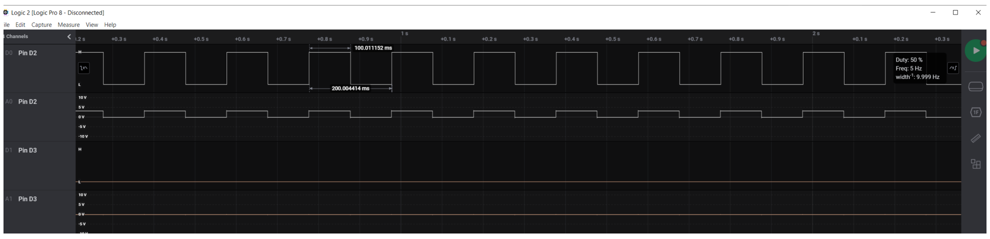

Figure 8. Mbed's waveform for the 100MS Delay

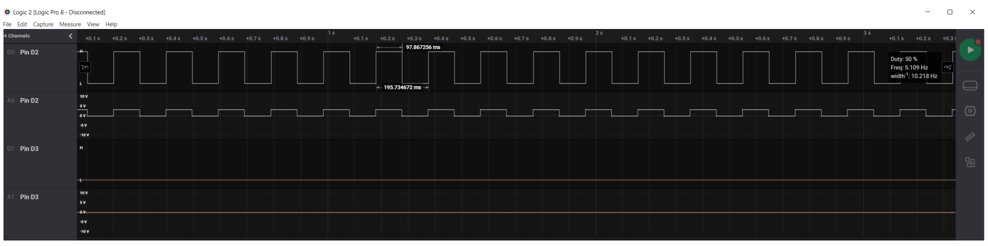

Figure 9. Tock's waveform for the 100MS Delay

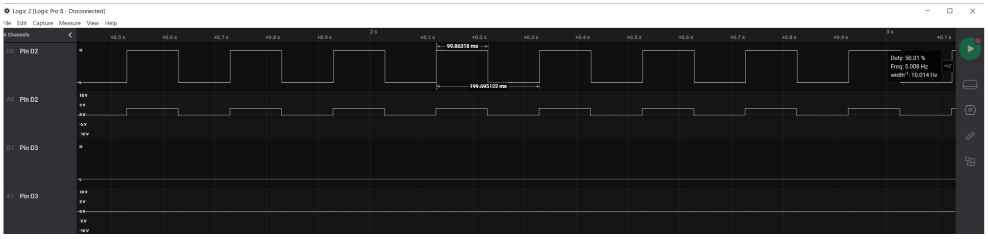

Figure 10. Riot's waveform for the 100MS Delay

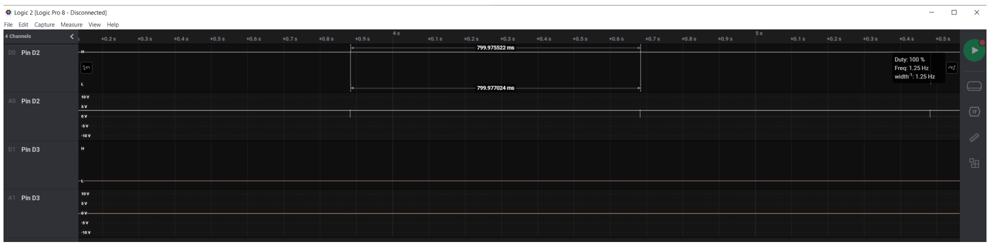

Figure 11. Mbed's waveform for 4 bit counter with Delay

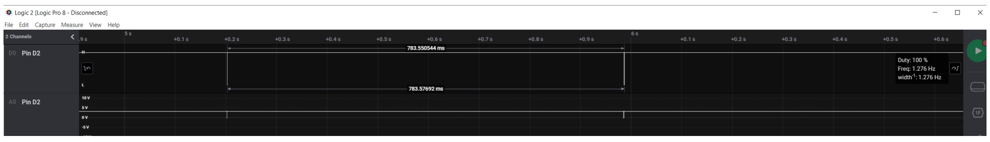

Figure 12. Tock's waveform for 4 bit counter with Delay

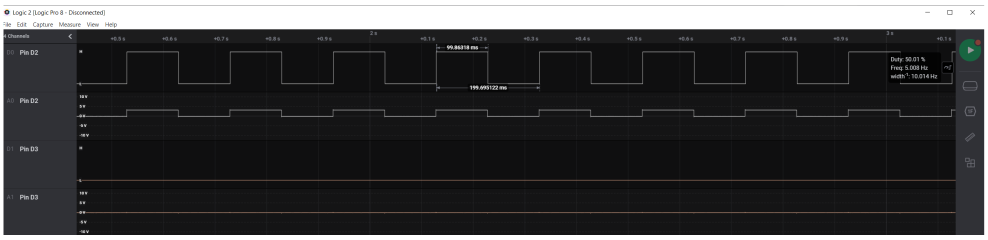

Figure 13. Riot's waveform for 4 bit counter with Delay

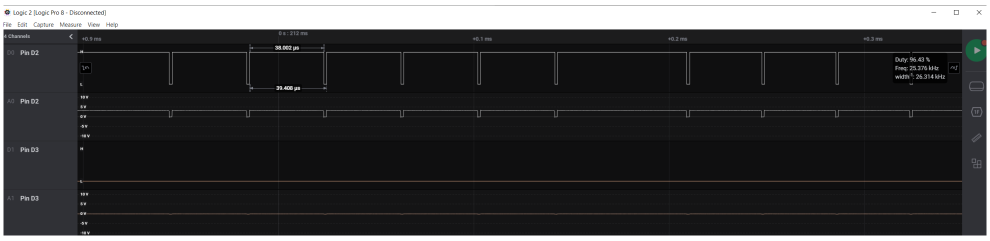

Figure 14. Mbed's waveform for 4 bit counter without Delay

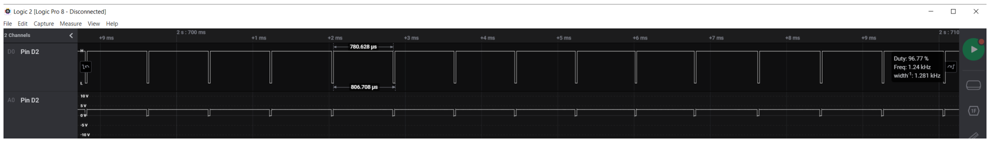

Figure 15. Tock's waveform for 4 bit counter without Delay

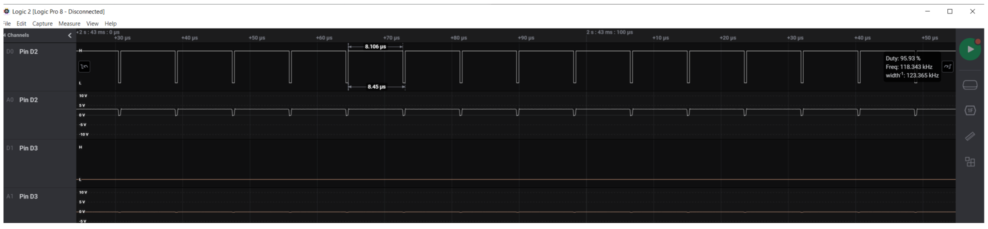

Figure 16. Riot's waveform for 4 bit counter without Delay

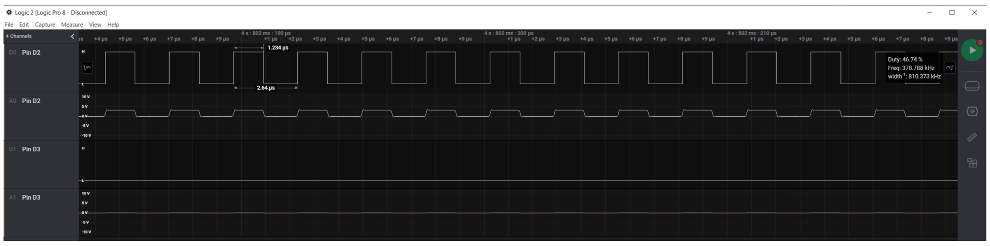

Figure 17. Mbed's waveform for baseline GPIO On then off

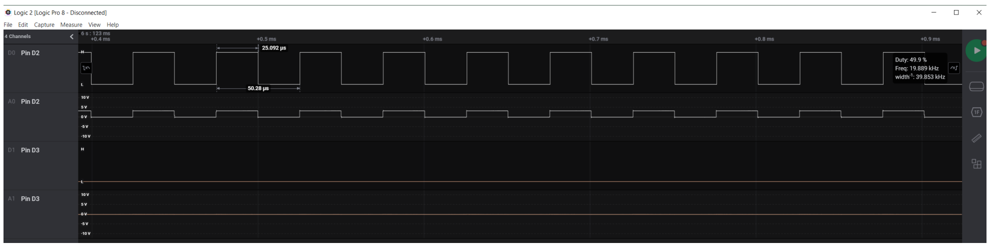

Figure 18. Tock's waveform for baseline GPIO On then off

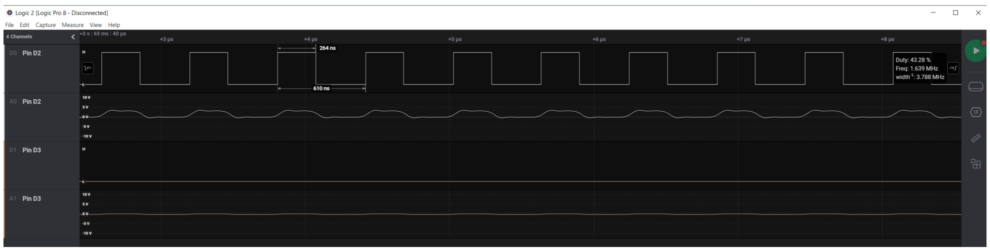

Figure 19. Riot's waveform for baseline GPIO On then off