Embedded Group Project

Project 4 - Kernel Space Encoder

Steinarr Hrafn Höskuldsson
Arnþór Gíslason
Andrew Madden

Reykjavik University

October 2022

1 Part 1

The header pins were soldered onto the Raspberry Pi Zero W2 and a voltmeter used to check that the pin orientation was correct.

2 Part 2

The four different methods for interfacing with the GPIO pins were tested.

2.1 Test Setup

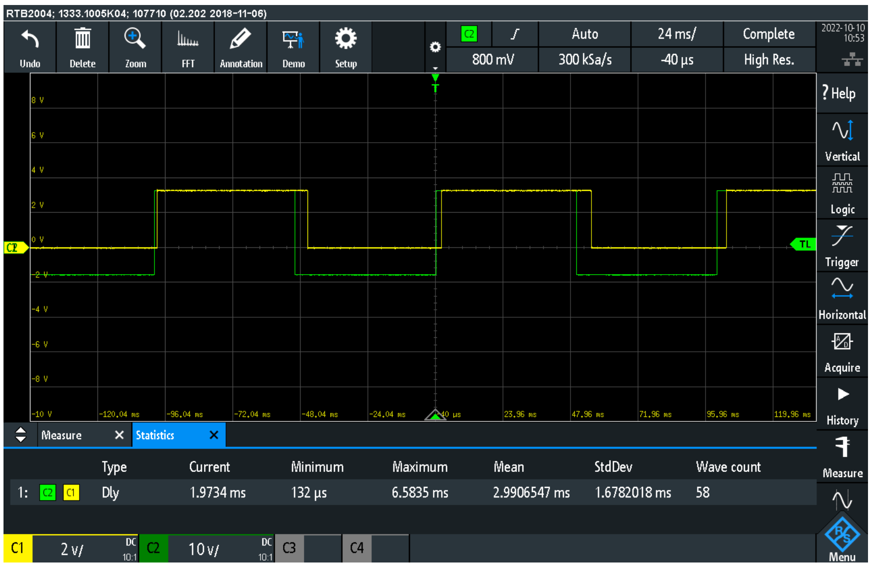
A RIGOL DG1022 function generator was hooked up to the input pin and set to output a square wave with amplitude 3 volts and frequency 1kHz. A Rhode&Schwartz RBT2004 oscilloscope was used to probe the input and output pins on different channels. The oscilloscope was then set to measure the time difference between rising edges.

With each method a screenshot of the oscilloscope was taken and the CPU load was read by running top in another terminal window.

2.2 Results

	Mean delay	StdDev	CPU usage
Shell script	3000 μs	1680 μs	19%
Read loop	8.5 μs	2.2 μs	100%
Polling	147 μs	24 μs	0.3%
Kernel Module	3.9 μs	0.48 μs	Not detectable

Table 1:



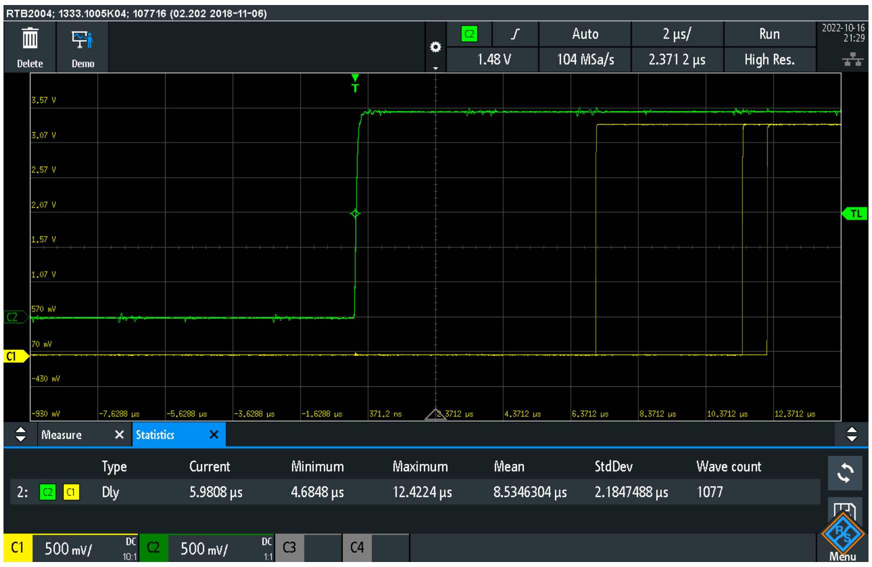
RTB2004; 1333.1005K04; 107710 (02.202 2018-11-06)

C2 · Auto · 24 ms/ · Complete
800 mV · 300 kSa/s · -40 µs · High Res.
2022-10-10 10:53

	Type	Current	Minimum	Maximum	Mean	StdDev	Wave count
1: C2 C1	Dly	1.9734 ms	132 µs	6.5835 ms	2.9906547 ms	1.6782018 ms	58

C1 2 V/ DC 10:1 · C2 10 V/ DC 10:1 · C3 · C4

Figure 1: Response time of a shell script utilizing sysfs to mirror a pin. CPU usage:19%

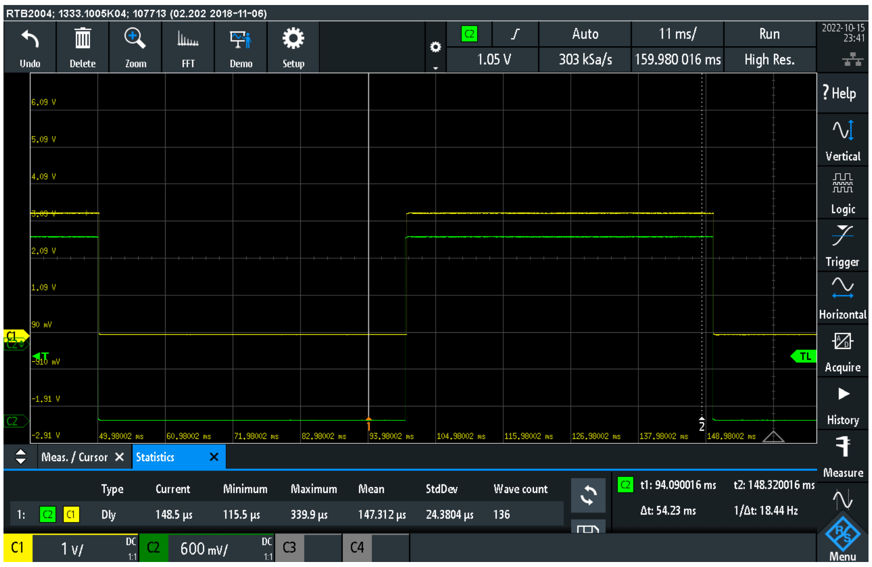

RTB2004; 1333.1005K04; 107716 (02.202 2018-11-06)

C2 ∫ Auto 2 µs/ Run 2022-10-16 21:29
1.48 V 104 MSa/s 2.371 2 µs High Res.

	Type	Current	Minimum	Maximum	Mean	StdDev	Wave count
2: C2 C1	Dly	5.9808 µs	4.6848 µs	12.4224 µs	8.5346304 µs	2.1847488 µs	1077

C1 500 mV/ DC 10:1 C2 500 mV/ DC 1:1 C3 C4

Figure 2: Response of a c program reading from sysfs as fast as possible. CPU usage: 100%


RTB2004; 1333.1005K04; 107713 (02.202 2018-11-06)

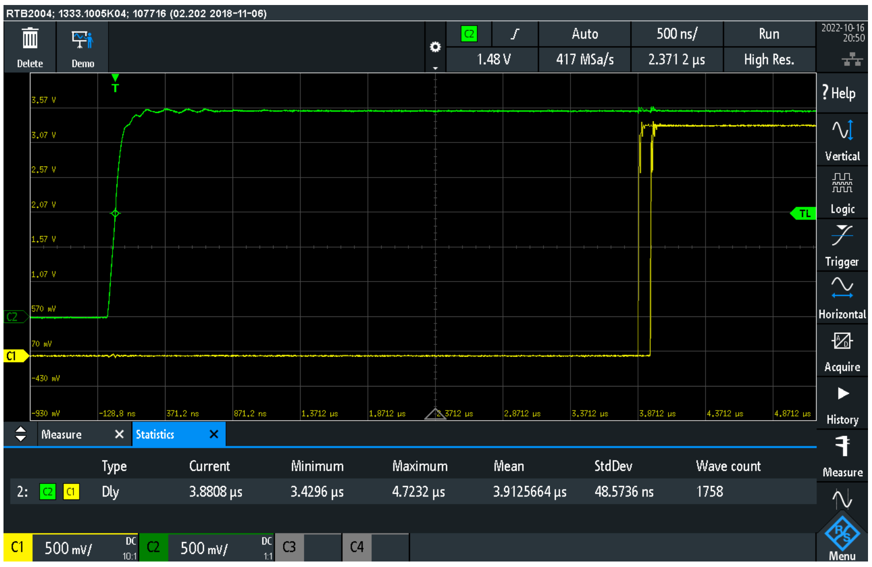
C2 ∫ Auto 11 ms/ Run 2022-10-15 23:41
1.05 V 303 kSa/s 159.980 016 ms High Res.

	Type	Current	Minimum	Maximum	Mean	StdDev	Wave count
1: C2 C1	Dly	148.5 µs	115.5 µs	339.9 µs	147.312 µs	24.3804 µs	136

C2 t1: 94.090016 ms t2: 148.320016 ms
Δt: 54.23 ms 1/Δt: 18.44 Hz

C1 1 V/ DC 1:1 C2 600 mV/ DC 1:1 C3 C4

Figure 3: Response time of c program that uses polling and sysfs to detect changes to input pin. CPU usage: 0.3%



	Type	Current	Minimum	Maximum	Mean	StdDev	Wave count
2: C2 C1	Dly	3.8808 µs	3.4296 µs	4.7232 µs	3.9125664 µs	48.5736 ns	1758

Figure 4: Response time of Linux Kernel Module that uses interrupts to detect changes to input pin. CPU usage: undetectable

2.3 Discussion

To count encoder pulses the Read Loop and Kernel Module methods would suffice however the read loop uses up an entire core of the CPU while the Kernel Module had no detectable effect on CPU usage. The cpu usage of the read loop could have been mitigated to some extend by sleeping some amount of microseconds each loop but that would in turn hurt the response time.

3 Part 3

The encoder counter was implemented in a kernel module. Two interrupt handlers were set up, one for each encoder input. Both were triggered on both rising and falling edges.

A user space program was written that based heavily on the arduino code from Project 3. The user space program fetched the number of pulses from the kernel space interrupt module and calculated the rpm from the number of pulses from the last run of update() and the approximate time between runs of update(). The program then wrote the calculated pwm to the pwm module built into the raspi using sysfs.

The motor controller was connected to the gpios on the raspi. After some time debugging, it worked perfectly but as was to be expected, there was quite considerable jitter.

The realtime when update() was run was taken for 1000 iterations and the results displayed on a histogram and a scatterplot to look at the jitter. On Figure 6 it may be seen that the time it takes varies from about 5.2ms to about 5.4ms but as may be seen in the histogram on Figure 5 that most of the runs are between 5.22ms to 5.24ms.

A better method of timing the main loop would have been beneficial but an argument could be made that for this snapshot taken of the controller running, no major actions are needed to improve the jitter between running the update() function of the controller. However, in the event that the cpu is more loaded, for example, if the raspberry pi had do other, compute heavy tasks, the jitter would certainly be worse and a more complicated implementation would be required. In figure 7 a plot may be seen where the cpu was more loaded and the jitter considerably worse, with the update() function being delayed up to 3ms. For example by recompiling the kernel with the real time preempt flag set, having a single core running the application solely or by simple adding a real time processor to the device which runs the PI controller.

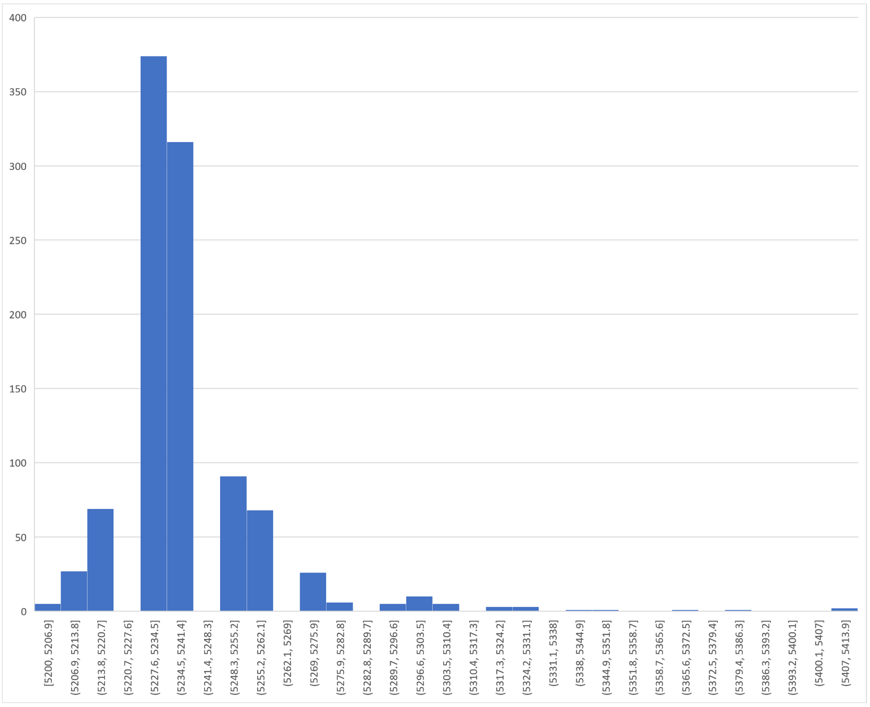


Figure 5: histogram plot of delta time between runs of the PI controller

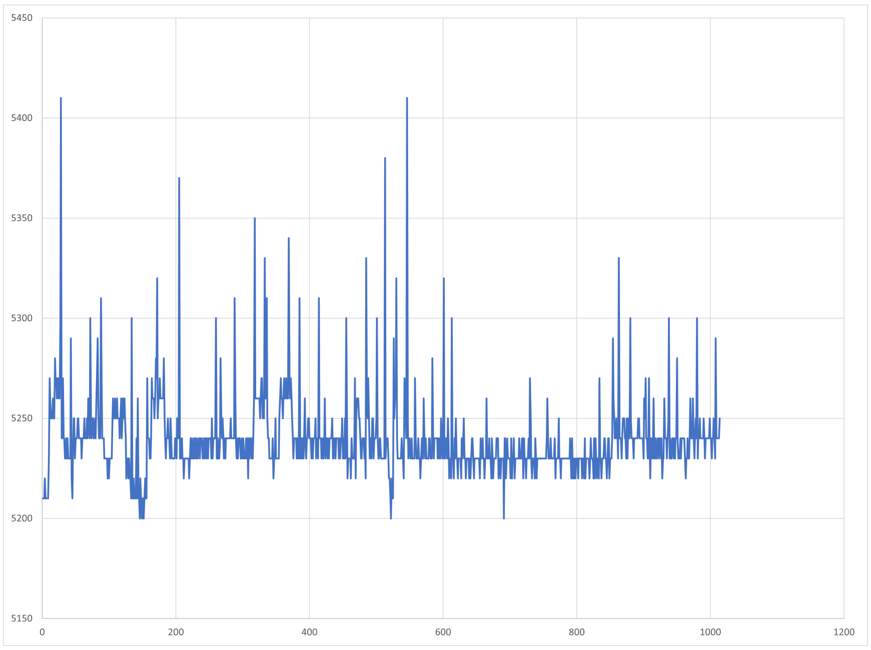


Figure 6: scatter plot of delta time between runs of the PI controller

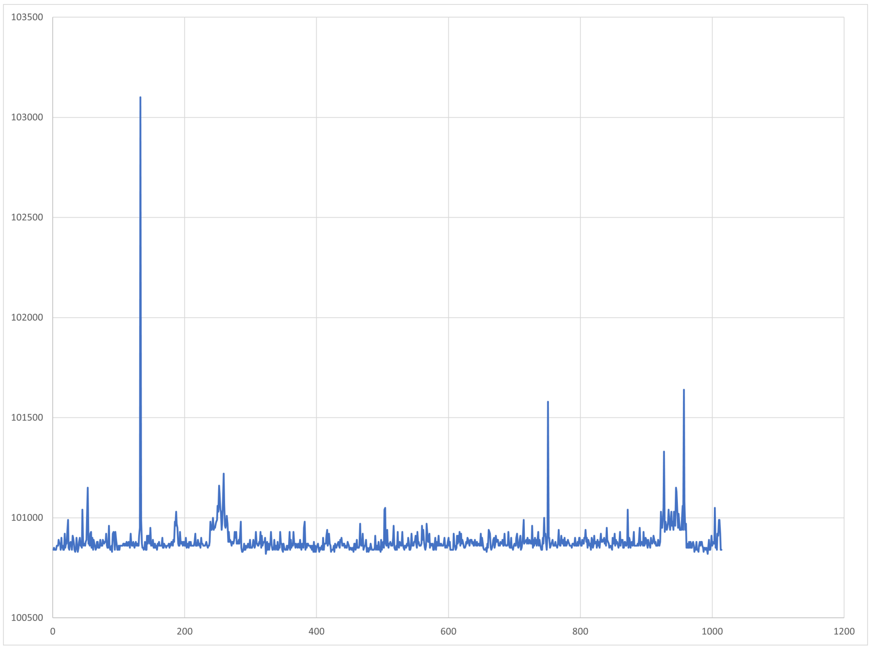


Figure 7: scatter plot of delta time between runs of the PI controller, when the cpu was more loaded

Appendix

A video of the control loop in action can be found at: `https://youtu.be/eprVd0OUXQw`

The code used can be found on github at `https://github.com/Steinarr134/EmbeddedGroupProjects/tree/8236f3d8a35ba01ab3f1a87bdadbbe8039d5d1cd/Project4KernelSpaceEncoderDriver`