Using Multiprocessing to Increase Sample Rate in Low-Cost Embedded Systems

Joseph Doyle

September 2023

Abstract

This paper investigates the effect multiprocessing has on the data collection rate in low-cost embedded systems. Specifically, this study ran embedded Rust on an Adafruit ESP32 Feather V2 to sample a 3-axis digital accelerometer continuously and to transmit the collected data over UART. Two programs were used to measure the average sample rate of a dual-buffer program in comparison to a sequential program. The dual-buffer program used both CPUs simultaneously to read and transmit our data, while the sequential program used only the PRO CPU. The programs showed that a single-threaded program was on average approximately 58 times slower than its asynchronous cousin with a buffer size of 575 (optimal). To determine that 575 was the smallest optimal buffer size, the program tested average sample rate for buffer sizes ranging from 300 to 900. This astronomical sample rate increase illustrates the possibilities of cheap and easily reproducible embedded systems being used in high-performance situations, once both CPU cores are used synchronously.

Acknowledgement

I am immensely grateful to Dr. George Anwar and Mr. Enrico Cruvinel of UC Berkeley for guiding my research and furthering my understanding of the topics presented in this paper. I would also like to thank the Cambridge Center for International Research for providing me with the opportunity and resources to conduct this research.

1 Introduction

In a world ruled by data, researchers continuously seek higher and higher fidelity measurements. More data points collected means a clearer picture of the physical phenomena affecting a system. However, embedded devices with higher clock speeds can be expensive and bulky. Thus, our goal in this article is to develop and test a program that utilizes resources efficiently and is able to sample data at speeds comparable to more expensive embedded systems. Our null hypothesis will be that a multi-cored approach offers no significant sample rate increase over a sequential approach while our alternate hypothesis is the opposite.

2 Hardware

2.1 ESP32

The ESP32 is a powerful dual-core 240MHz Xtensa processor running on the Harvard Architecture that is extremely popular due to its low cost and versatility. It boasts 8 MB Flash, 2 MB PSRAM, and built-in WiFi and Bluetooth capabilities.[3] This makes it the processor of choice for many affordable Internet of Things and data collection applications. Its dual-core abilities also make it perfectly suited to multiprocessing applications, meaning it is the best choice for our purposes.

2.2 LSM6DSO

The LSM6DSO is a "system-in-package featuring a 3-axis digital accelerometer and 3-axis digital gyroscope"[8] It supports the SPI, I2C, and I3C serial interfaces with main processor synchronization. Our program uses the fast mode I2C protocol to communicate with this device, setting the SCL clock frequency to 400kHz, the maximum speed of the LSM6DSO.[8] Conveniently, this sensor is that runs on 3.3V, which is the working voltage of the ESP32. This eliminates the need for any logic level converters.

3 Software

All software developed for this project was written in Rust.[9] Rust was an ideal choice because of its speed, low-level abilities, open source nature, and the fact that it is an emerging language in the embedded software domain that is likely to gain ground in the coming years.[1]

3.1 Embedded Software

All code written for the ESP32 uses the esp-hal library extensively.[5] This is the no-std version of the hardware abstraction layers provided by Espressif. The primary reasons for developing these programs in a no-std environment were less overhead and increased control over the CPU cores.[10]

3.1.1 Sequential Approach

The sequential approach to continuous sampling is quite straightforward: our ESP32 writes a sequence of bytes containing the OUTX_L_A command, the OUTY_L_A command, and the OUTZ_L_A command. This will prompt the accelerometer to write the acceleration in the x, y, and z directions, respectively.[8] After receiving these values, the ESP32 will immediately write a package of data including these values and the time elapsed to the USB connection using UART. The process then repeats.

[1] Software associated with this project is available at `https://github.com/JoeDoyle12/ESP32Multiprocessing`

3.1.2 Parallel Approach

In our multi-threaded program, two buffers of equal size are defined at the start of the program, and the APP CPU is started with a task that will write one buffer until it is full, then write the next one until it is full, and then switch back to the first one. The PRO CPU will wait until the first buffer is full, write its contents to the UART connection, then wait until the second buffer is full, and so on. A visual demonstration of these mechanics is provided in Figure 1. Safe passing of references to and from these buffers is achieved with synchronization primitives provided by the spin-rs crate.[7] Instead of requiring a Real Time Operating System (RTOS), as the objects provided by the standard library do, these sync primitives work on the principle of "spinning" the CPU (giving it "busy" work to do while attempting to acquire a lock that is not available). Because they do not rely on a RTOS, these primitives can be used in a no-std environment, which is perfect for our purposes.
This approach, like the sequential approach, offers continuous sampling of the accelerometer, but the delays between samples are much smaller, due to the utilization of both CPUs in parallel. This means our sample rate can be pushed much higher, without sacrificing either accuracy or continuous sampling.

3.2 Native Program

Writing our data to UART from the ESP32 is only useful if our computer can collect this data as it comes in from the ESP without falling behind. Our native Rust program uses the serialport-rs crate.[6] It first waits for a sequence of alignment bytes from the ESP32, and then read bytes totaling the entire size of a buffer. After repeating this sequence a pre-specified number of times, it translates these arrays of bytes into one unsigned 64 bit integer (time elapsed) and three signed 16 bit integers (x, y, and z acceleration) using the byteorder crate.[4] It then calculates the average sample rate from the time measurements collected by this program.

4 Results

We tested the speed of the sequential program by having it collect 100,000 data points one by one and send each one through UART before collecting the next one. This approach yielded an average time between samples of 135,177 μs, which means an average sample rate of 7.4 samples per second.
e Next, our program calculated the smallest optimal buffer size for our parallel program. There are infinite buffer sizes for which the APP CPU will never have to pause and wait for the PRO CPU, and this optimal buffer size will use the least possible amount of memory while still achieving this optimal amount. In real life however, sample rate can fluctuate greatly from one buffer size to the next depending on many internal factors. Despite this, our program will still seek out an "optimal" size. To do this, it tested numerous buffer sizes ranging from 300-900, and documented the average sample times and rates for each program over ∼15,000 data points. The data collected can be found in Table 1.

5 Discussion

Our data shows that utilizing both cores simultaneously can increase sample rate by about 5,800%. This has great implications for the continued development of low-cost continuous data collection systems; it shows that inexpensive microprocessors can be utilized efficiently to greatly increase sample rate.
Previous research in this area, especially using the ESP32, was mainly focused on sending data over WiFi in an environment containing the standard library. [1, 2] This means that during that research, a RTOS was present and was handling thread scheduling. This introduces significant overhead, because many of the resources on the APP CPU are devoted to the RTOS.[2] Future research could be done to determine by how much sample rate can be increased (if at all) on a microprocessor running a Real Time Operating System using the multiprocessing techniques outlined in this paper.

6 Conclusion

This research aimed to develop and test a multi-threaded continuous data sampling program using a low-cost microprocessor in order to demonstrate the viability of using such techniques to collect high-fidelity data affordably. Such systems are used widely in research, and this paper shows that if a Real Time Operating System is not needed to collect or transport data, these systems can be sped up significantly. With a sequential approach, the factor that determines speed is how fast our programs can transmit and receive bytes through UART. In our multi-threaded approach, the only limits are those imposed internally by the sensor and by the CPU clock.

References

[1] Ibrahim Allafi and M. Tariq Iqbal. "Design and implementation of a low cost web server using ESP32 for real-time photovoltaic system monitoring". In: Oct. 2017, pp. 1–5. DOI: 10.1109/EPEC.2017.8286184.

[2] Marek Babiuch, Petr Foltynek, and Pavel Smutný. "Using the ESP32 Microcontroller for Data Processing". In: May 2019, pp. 1–6. DOI: 10.1109/CarpathianCC.2019.8765944.

[3] *ESP32 Technical Reference Manual*. Version 5.0. Espressif Systems, 2023. URL: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.

[4] Github. *byteorder: Convenience Methods for Encoding and Decoding Numbers in Either Big-Endian or Little-Endian Order*. Version 1.4.3. URL: https://github.com/BurntSushi/byteorder.

[5] Github. *esp-hal: Hardware Abstraction Layer Crates for the ESP32, ESP32-C2/C3/C6, ESP32-H2, and ESP32-S2/S3 from Espressif (No-Std)*. Version 0.13.0. URL: https://github.com/esp-rs/esp-hal.

[6] Github. *serialport-rs: General-Purpose Cross-Platform Serial Port Library for Rust*. Version 4.2.2. URL: https://github.com/serialport/serialport-rs.

[7] Github. *spin-rs: Spin-based Synchronization Primitives*. Version 0.9.8. URL: https://github.com/mvdnes/spin-rs.

[8] *LSM6DSO Datasheet: iNEMO Inertial Module (Always-On 3D Accelerometer and 3D Gyroscope)*. STMicroelectronics, 2019. URL: https://cdn.sparkfun.com/assets/2/3/c/6/5/lsm6dso.pdf.

[2] This was another argument for using Rust, because other frameworks like MicroPython will simply bar you from running code on the APP CPU in order to reserve it for the RTOS alone.

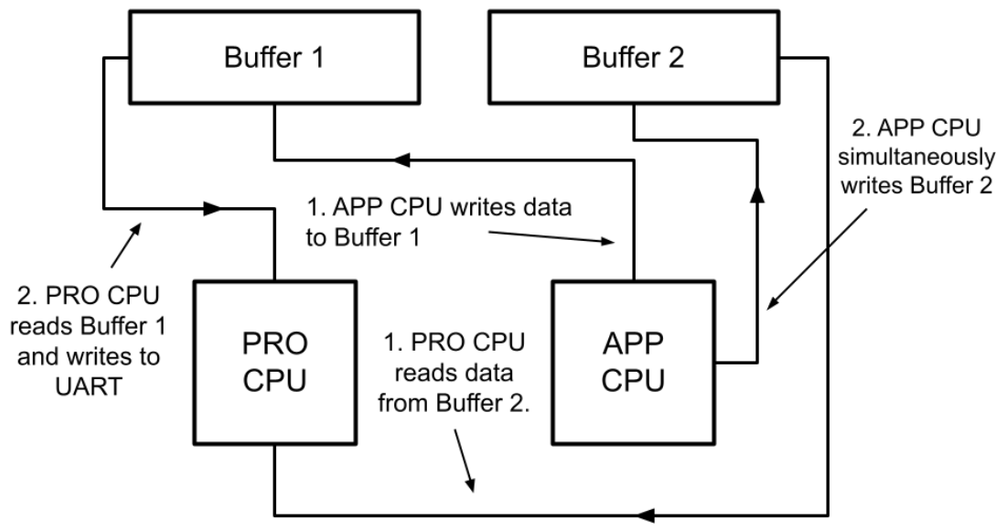

Figure 1: Illustration of Buffer-CPU Timing While Multiprocessing

Buffer Size (Data Points)	Average Sample Time (μs)	Average Sample Frequency (1/s)
300	2684.24	372.54
400	2752.16	363.37
500	2590.86	385.98
550	2748.75	363.80
575	2346.22	426.22
600	2364.69	422.89
625	2920.59	342.40
650	3063.60	326.41
700	2779.69	359.75
800	3026.53	330.41
900	2676.52	373.62

Table 1: Buffer Size vs Sample Rate

[9] *Rust: A Language Empowering Everyone to Build Reliable and Efficient Software*. Version 1.71.0. URL: https://www.rust-lang.org.

[10] *The Embedded Rust Book*. URL: https://docs.rust-embedded.org/book/.