گزارش نهایی

پروژه تهیه benchmark برای آزمون سیستم‌های نهفته (ارتباطات)

آزمایشگاه سخت‌افزار پاییز 1403

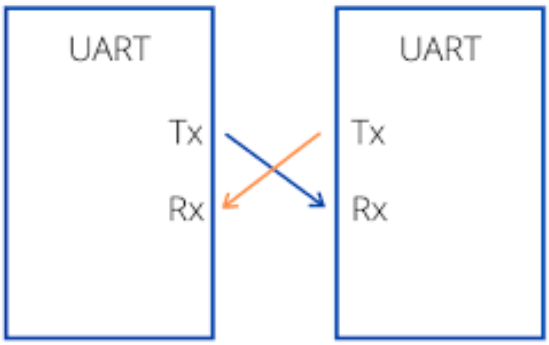
اعضای تیم:

نگار عسکری 99105612

هیراد داوری 99106136

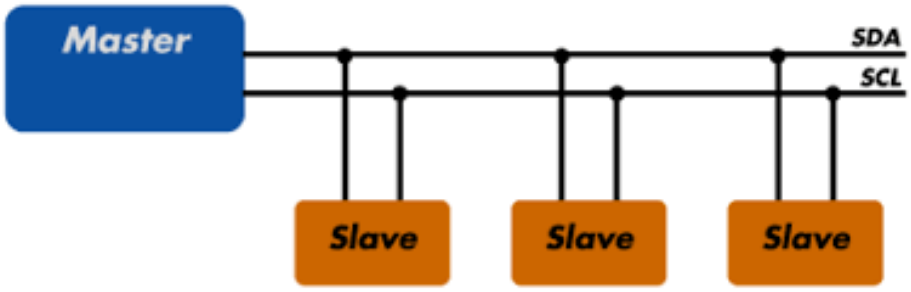
پروتکل‌های مورد بررسی

1. پروتکل UART یا Universal Asynchronous Receiver Transmitter ساختار بسیار ساده‌ای دارد. این پروتکل از اتصال دو سیم میان دو دستگاه استفاده کرده و یک ارتباط دو طرفه کامل (Full-Duplex) ایجاد می‌کند. انتقال داده در این پروتکل به صورت سریال و آسنکرون می‌باشد، یعنی دو دستگاه متصل شده نیازی به کلاک مشترک ندارند، اما نیاز است که پیش از برقراری ارتباط هر دو طرف روی متغیرهایی مانند baude rate به توافق برسند. همچنین یک ویژگی قابل توجه آن تقارن میان دو طرف اتصال است، برخلاف دو پروتکل دیگر، UART متقارن بوده و معماری Master/Slave ندارد.

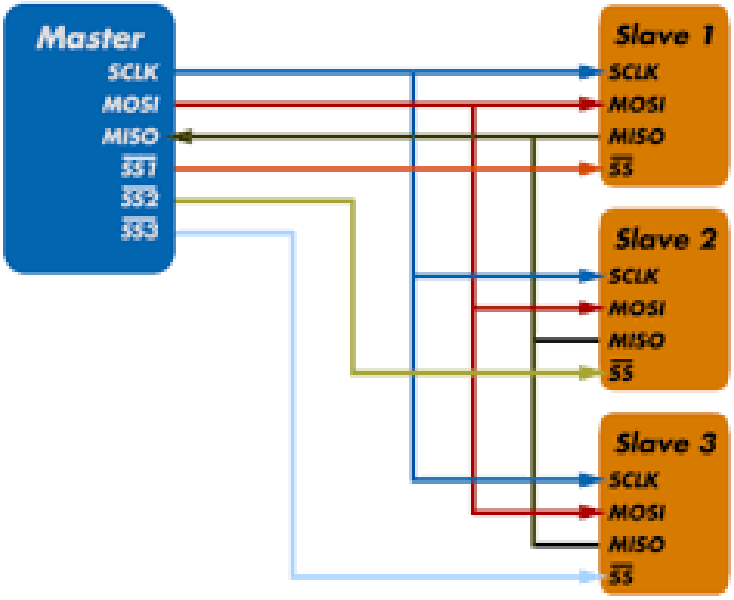


2. پروتکل I2C یا Inter Integrated Circuit معمولا برای انتقال داده داخل یک بورد به کار می‌رود. این پروتکل نیز از دو سیم برای برقراری ارتباط استفاده می‌کند، اما چون انتقال داده به صورت سنکرون است، یکی از این سیم‌ها برای

کلاک استفاده شده و امکان برقراری ارتباط دو طرفه کامل وجود ندارد. ارتباط در این پروتکل متقارن نیست، در واقع تعدادی دستگاه در یک Bus به یکدیگر متصل می‌شوند و سپس یکی از آن‌ها به عنوان Master شروع به کار می‌کند و با مشخص کردن آدرس Slave مد نظر خود، به آن داده ارسال کرده یا از او داده دریافت می‌کند.



3. پروتکل SPI یا Serial Peripheral Interface سرعت بالاتری نسبت به دو پروتکل قبلی دارد، اما تعدادی سیم‌های هر ارتباط حداقل ۴ می‌باشد. این پروتکل همانند I2C نامتقارن بوده و یک دستگاه به عنوان Master عمل کرده و انتقال داده را کنترل می‌کند. به کمک تعداد بالای سیم‌ها، یک Master می‌تواند به کمک سیم Chip Select یا CS متصل به هر Slave مشخص کند که آیا او در حال حاضر باید در حال انتقال داده باشد یا خیر. همچنین، علاوه بر یک سیم کلاک، دو سیم دیگر برای انتقال داده دو طرفه استفاده می‌شوند و این پروتکل توانایی برقراری ارتباط Full-Duplex را دارد.



به صورت خلاصه و در یک نگاه داریم:

	Speed	Range	Wires	Power Use
UART	1.2 kbps to 1 Mbps	<15 meters	2	low
I2C	100 kbps to 3.4 Mbps	~1 meter	2	low
SPI	<50 Mbps	~1 meter	>4	moderate

بررسی ارجاعات به اهمیت این پروتکل‌ها

برای اطمینان حاصل کردن از درستی انتخاب پروتکل‌ها و بررسی حوزه‌های استفاده و میزان کاربرد نسبی آن‌ها، در طول انجام این پروژه زمان زیادی را صرف بررسی مقالات منتشر شده در این حوزه کردیم.

هدف اصلی ما، دست یافتن به یک گزارش آماری در مورد میزان استفاده از این پروتکل‌ها در تکنولوژی‌های روز دنیا بود. برای این مرحله در سایت‌هایی مانند

- https://ieeexplore.ieee.org/
- https://scholar.google.com/
- https://arxiv.org/
- https://www.researchgate.net/

و غیره جست و جو کرده و مقالات مرتبط، به خصوص مقالات مروری، را مطالعه و بررسی کردیم.

در نهایت، با این که آمار عددی و کمی در این راستا یافت نشد، گزاره‌های کیفی زیادی با مضمون استفاده گسترده و اهمیت بالای پروتکل‌های انتخاب شده جمع آوری شدند. این نتایج در گزارش‌های هفتگی پیشین قابل مشاهده می‌باشند. چند نمونه به عنوان مثال در ادامه آورده شده‌اند:

> **Quote:** “Popular serial interfacing protocols include: USB, I2C, SPI/SSP, CAN and UART for communication between integrated circuits for low/medium data transfer speed with on board peripherals.”

Source: Kommu et al, 2014, Designing a learning platform for the implementation of serial standards using ARM microcontroller LPC2148

Quote: "Today, at the low end of the communication protocols we find the inter-integrated circuit (I2C) and the serial peripheral interface (SPI) protocols. Both protocols are well suited for communications between integrated circuits for slow communication with on-board peripherals. The two protocols coexist in modern digital electronics systems, and they probably will continue to compete in the future, as both I2C and SPI are actually quite complementary for this kind of communication."
Source: Leens, 2009, An introduction to I2C and SPI protocols

Quote: "Nowadays there are many hardware communications protocols to choose from, UART, SPI, I2C three protocols are the most representative, they are widely used."
Source: Chen & Huang, 2023, Analysis and Comparison of UART, SPI and I2C

Quote: “Today most major semiconductor manufacturers offer IC’s supporting I2C. Most device types support I2C such as Processors, EEPROMs, Temperature sensors, A2D, D2A, Video components, Real Time Clocks, Displays and even programmable devices like FPGAs can utilize I2C IP cores … The use of SPI is widespread and is used by many IC manufacturers, ASIC and FPGA implementers and peripheral devices. SPI can also be used for accessing EEPROM, Flash, A2D, D2A, Real Time Clocks, sensors and other devices.”

Source: Myers, 2007, Interfacing using Serial Protocols Using SPI and I2C

همچنین، پس از یک فاز اولیه بررسی دستی مقالات و برای اطمینان حاصل کردن از کامل بودن جست و جو، با تدارک دیدن ترکیبی از یک Web Crawler و یک Large Language Model، تعداد زیادی مقاله مرتبط به این حوزه را بررسی کرده و گزاره‌های مرتبط را از آن‌ها استخراج کردیم.

در این بخش، LLM منبع باز Mistral nemo را به صورت لوکال اجرا کردیم. با توجه به منابع سخت‌افزاری محدود، از مدل quantize شده آن استفاده کرده و آن را روی پلتفرم llama-cpp اجرا کردیم.

در ادامه، به کمک یک کد پایتون بخش crawler را ساختیم. چالش اصلی این بخش مسئله captcha بود که پس از امتحان کردن راه حل‌های متعدد و محدود کردن جست و جو به منبع https://www.semanticscholar.org/ از آن عبور کردیم.

در نهایت یک workflow تشکیل دادیم، به این صورت که با الگوریتم BFS مقالات مرتبط به موضوع را دریافت کرده و abstract آن را پس از parse کردن به ورودی مدل زبانی می‌دهیم. سپس، این مدل ورودی را بررسی کرده و آن یک امتیاز از ۰ تا ۱ می‌دهد. به کمک این امتیازها، مقالاتی که شانس بالاتری برای در بر داشتن آمار مفید دارند تفکیک شده و توسط ما به صورت دستی بررسی شدند.

در واقع، مدل در نهایت به ازای هر مقاله دریافت شده یک خروجی به فرمت زیر تولید می کند:


```
{   "score": 0.7,
    "title": "An Insight Comparison of Serial Communication Protocols",
    "topics": [
      "Integrated Circuits",
      "Data Link Layer",
      "Application Layers",
      "Data Rate"
    ],
    "abstract": "...",
    "analysis": "(7/10)",
    "noref": false,
    "url": "https://www.semanticscholar.org/paper/An-Insight-Comparison-of-Serial-Communication-Khurana-Goyal/77f60a2da3b35d07dce1ef79f3dd8b2ed30b34a8"}
```

پس از بررسی دستی نتایج این مرحله نیز به گزاره‌هایی مشابه مرحله قبل رسیدیم:

Quote: "UART abbreviated as Universal Asynchronous Receiver Transmitter is one of the essential element of communication system. It is being mostly used when there is a short-distance, between computer and peripherals. Whenever there is low-cost data exchange or the speed required for transmission is not high, UART's are also being used there."

Source: Gupta et al, 2015, 28nm FPGA based Power Optimized UART Design using HSTL I/O Standards

Quote: "I2C and SPI are the most commonly used serial protocols for both inter-chip and intra-chip low/medium bandwidth data-transfers."

Source: Solheim et al, 2015, A comparison of serial interfaces on energy critical systems

بررسی بنچمارک‌های پیشین

یک مرحله مهم و تحقیقاتی دیگر در انجام این پروژه، مروری بر بنچمارک‌های پیشین در حوزه آزمون سیستم‌های نهفته بود. برای انجام این کار، ابتدا با جست و جوی کلی و با راهنمایی دستیاران آموزشی، لیستی از بنچمارک‌های مرتبط تهیه کرده و توضیحات ابتدایی در مورد آن‌ها ارائه کردیم.

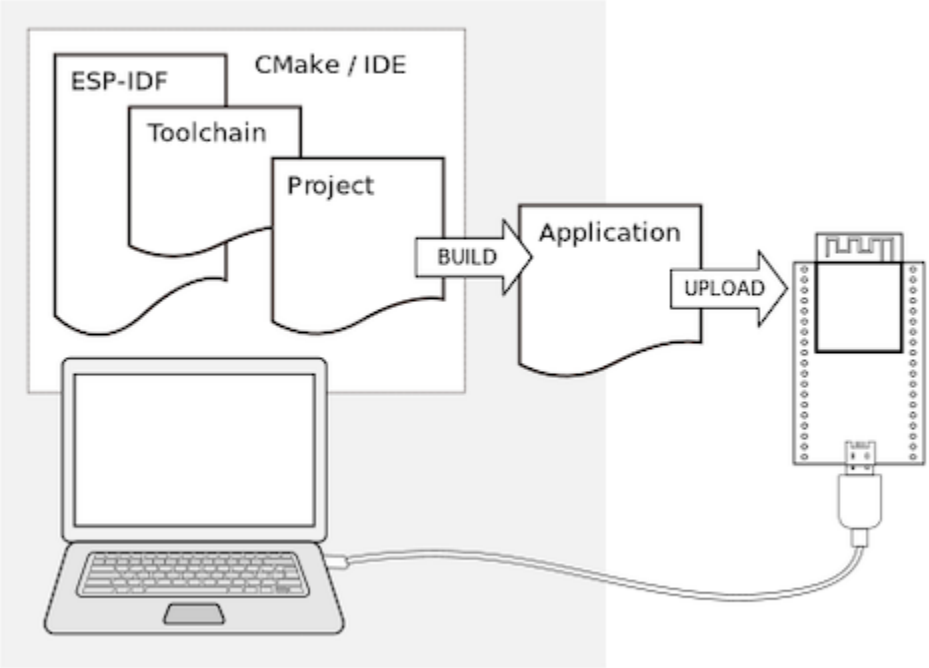
سپس پس از انتخاب مرتبط ترین موارد، یک مقایسه جامع میان آن‌ها انجام دادیم تا برای مرحله پیاده‌سازی آماده شویم. لیست و بررسی کامل، به همراه توضیح معیارهای ارزیابی شده در گزارشات هفتگی ارائه داده شدند، با این حال در ادامه خلاصه‌ای از نتایج این بخش از پژوهش ارائه می‌شود:

	Portability	Changeability	Completeness	Level
MiBench	low	medium	medium	(mostly) low
Embench	high	high	low	low
PapaBench	low	medium	high	high
RT-Bench	medium	high	high	low

پیاده‌سازی

پلتفرم ESP-IDF

برای توسعه برنامه روی برد ESP از این پلتفرم استفاده کردیم. پس از نوشتن برنامه به کمک این پلتفرم، ابتدا به کمک Toolchain پروژه را کامپایل می کنیم، سپس به کمک CMake و Ninja آن را Build می‌کنیم. در نهایت، برنامه به دست آمده را روی برد آپلود می کنیم. به کمک کتابخانه ها و کد سطح پایین موجود در ESP-IDF، این برنامه روی برد اجرا خواهد شد.



این پلتفرم ویژگی‌ها و قابلیت‌های منحصر به فرد و مفید زیادی دارد که تعدادی از آن‌ها را در ادامه نام می‌بریم:

- درایورهای مختلف برای دستگاه‌های موجود روی برد

- چهارچوب کامل برای logging
- کرنل FreeRTOS دارای قابلیت‌های:
  - اجرای تسک‌های multi threaded روی هسته‌های مختلف
  - قابلیت ایجاد تسک‌های real time و اعمال ددلاین‌ها
  - پیاده‌سازی صف و سمافور
- زیرمجموعه stdlib پیاده‌سازی شده برای سخت‌افزار ESP32
- پروژه‌های CMAKE که توانایی customize و reuse کردن کد را به ما می‌دهند
- امکان ایمپورت کردن پروژه‌های دیگر به عنوان dependency
- وجود قابلیت‌های دیگر POSIX Compliant

تعریف بنچمارک

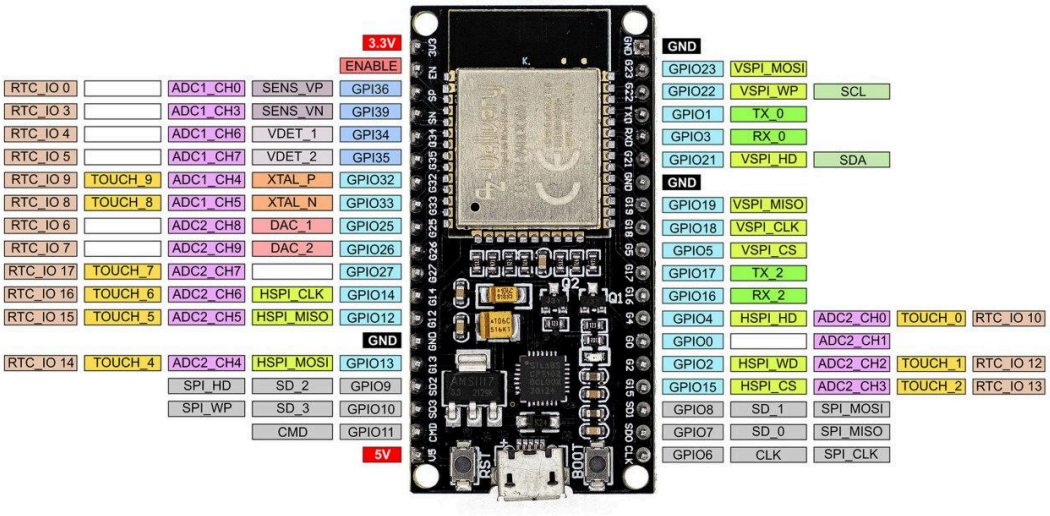
برای تولید این بنچمارک، یک سناریو مانند یک سیستم digital twin در نظر گرفتیم که در آن یک سیستم نهفته بی‌درنگ به صورت متناوب و با یک ددلاین مشخص ثابت (مثلا 100ms) باید استاتوس خود را در سیستم دیجیتال آپدیت کند و پیامی شامل وضعیت فعلی خود برای دستگاه دیگری ارسال کند. همچنین پس از ارسال هر پیام، یک پاسخ از آن دستگاه دریافت کند.

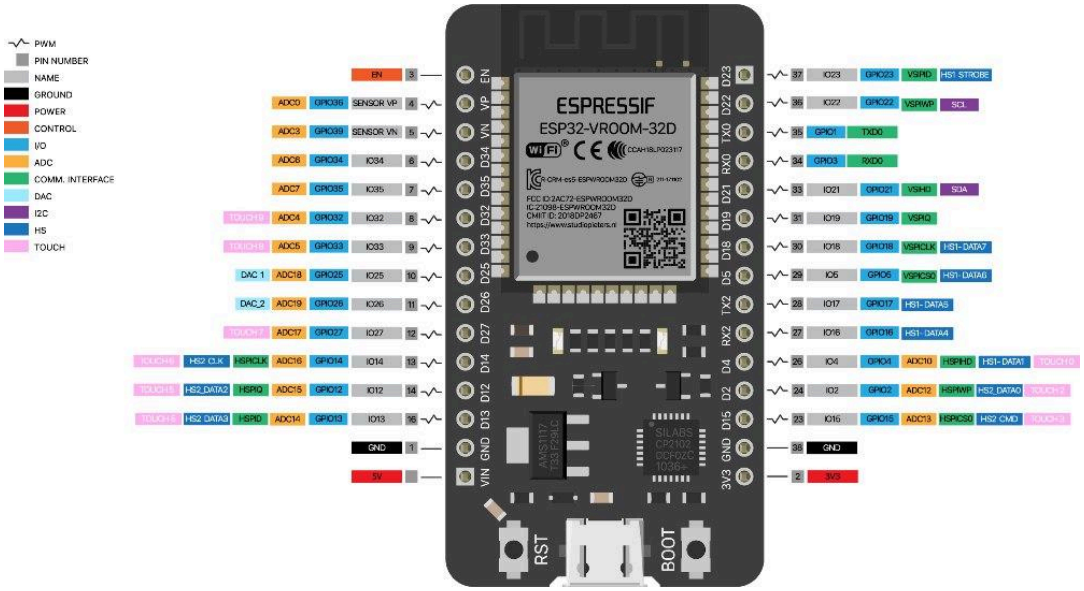
با اجرای این سناریو، می‌توانیم زمانی که دستگاه در هر تناوب صرف read و write کردن می‌کند را اندازه‌گیری کنیم، Worst Case Execution Time و Average Execution Time را محاسبه کنیم، و بررسی کنیم که چه درصدی از ددلاین‌های این دستگاه meet و چه درصدی miss می‌شوند. همچنین با محاسبه Standard Deviation

این زمان‌ها، می‌توانیم میزان Variation آن‌ها و در نتیجه میزان قابل اطمینان بودن نتایج را بررسی کنیم.

مراحل و موانع

با توجه به دیتاشیت و دیاگرام pinout برد ESP32، پلتفرم را کانفیگ می‌کنیم تا پایه‌ها را به درستی به دستگاه‌های مختلف روی برد map کند.





متغیرهای دیگر مانند تعداد راندهای شبیه‌سازی و همچنین میزان داده تولید شده برای بنچمارک را نیز می‌توان در این بخش تنظیم کرد. پس از مشخص کردن این متغیرها، لازم است برنامه را کامپایل کنیم. در نهایت می‌توان این نتیجه را روی برد فلش کرد و خروجی‌های آن را با یک اتصال ساده به صورت مستقیم از مانیتور خواند.

برای انتقال داده، ما دو دستگاه مشابه رو یک برد را به یکدیگر متصل کرده و داده را میان آن‌ها منتقل می‌کنیم. یک تفاوت این سناریو با سناریوهای معمول دنیای واقعی، یکی بودن پردازنده کنترل کننده هر دو دستگاه است، که البته تاثیر قابل توجهی روی نتایج نخواهد داشت.

ما در ابتدا این مراحل را برای پروتکل UART انجام دادیم که با توجه به سادگی پروتکل، به نسبت پیچیدگی کمی داشت و بدون مشکل خاصی انجام شد.

پروتکل‌های SPI و I2C با توجه به نامتقارن بودنشان پیچیدگی بیشتری در پی داشتند. ما در آزمون خود، انتقال داده را از Master به Slave انجام دادیم.

یک مشکل زمانبر که در پیاده‌سازی تست پروتکل SPI به آن برخوردیم، محدودیت ۳۱ بیتی روی حجم داده ارسالی بود. پس از مقدار زیادی آزمون و خطا و همچنین جست و جو در Forum های مربوطه، متوجه شدیم که این مشکل ناشی از کد ما نبوده و یک باگ داخلی و حل نشده از درایور espressif می‌باشد.

نتایج

خروجی حاصل از اجرای بنچمارک روی سه پروتکل مشخص شده به صورت زیر بود:

1. UART

```
D (10231) UART: Transmit::Time taken: 20498 microseconds
D (10321) UART: Receive::Time taken: 21373 microseconds
D (10331) UART: Transmit::Time taken: 20498 microseconds
D (10421) UART: Receive::Time taken: 21372 microseconds
D (10451) UART: Transmit::Time taken: 40968 microseconds
D (10541) UART: Receive::Time taken: 41380 microseconds
D (10551) UART: Transmit::Time taken: 20498 microseconds
D (10641) UART: Receive::Time taken: 21372 microseconds
D (10651) UART: Transmit::Time taken: 20498 microseconds
D (10741) UART: Receive::Time taken: 21373 microseconds
D (10771) UART: Transmit::Time taken: 40968 microseconds
D (10861) UART: Receive::Time taken: 41380 microseconds
D (10871) UART: Transmit::Time taken: 20498 microseconds
D (10961) UART: Receive::Time taken: 21372 microseconds
D (10971) UART: Transmit::Time taken: 20498 microseconds
D (11061) UART: Receive::Time taken: 21372 microseconds
D (11091) UART: Transmit::Time taken: 40968 microseconds
D (11181) UART: Receive::Time taken: 41381 microseconds
D (11191) UART: Transmit::Time taken: 20498 microseconds
D (11281) UART: Receive::Time taken: 21372 microseconds
D (11291) UART: Transmit::Time taken: 20498 microseconds
D (11381) UART: Receive::Time taken: 21372 microseconds
D (11411) UART: Transmit::Time taken: 40968 microseconds
W (11621) UART: INCOMPLETE DATA RECEIPT
D (11621) UART: Receive::Time taken: 160672 microseconds
I (11701) UART: BOTH TASKS ENDED, calculating metrics
I (11701) RESULT_UART: ======== Average Time ========
I (11701) RESULT_UART: Average read time: 29286 us
I (11701) RESULT_UART: Average write time: 27321 us
I (11711) RESULT_UART: Standard deviation of read times: 16480.9987 us
I (11711) RESULT_UART: Standard deviation of write times: 9700.3052 us
I (11721) RESULT_UART: ============ WCET ============
I (11731) RESULT_UART: Read WCET: 160672 us
I (11731) RESULT_UART: Write WCET: 40968 us
I (11741) RESULT_UART: ========== Deadlines =========
I (11741) RESULT_UART: Read Deadlines missed (80 ms): 1
I (11751) RESULT_UART: Write Deadlines missed (80 ms): 0
I (11751) RESULT_UART: ========== FINISH ==========
I (11761) main_task: Returned from app_main()

Done
```

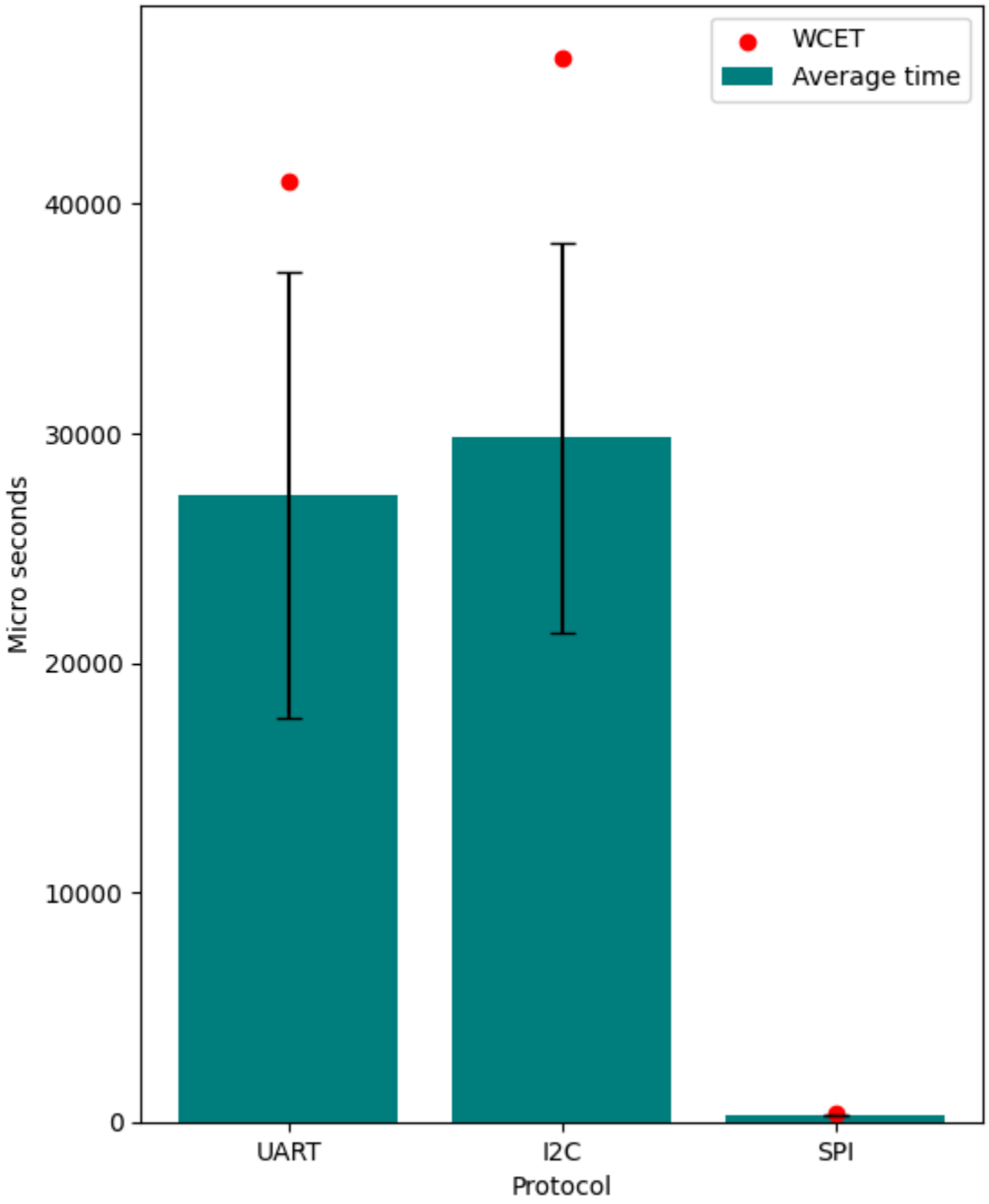
2. I2C

```
D (11561) I2C: Transmit::Time taken: 25432 microseconds
D (11561) I2C: Receive::Time taken: 29337 microseconds
D (11661) I2C: Transmit::Time taken: 25525 microseconds
D (11691) I2C: Receive::Time taken: 29386 microseconds
D (11791) I2C: Transmit::Time taken: 25480 microseconds
D (11811) I2C: Receive::Time taken: 29454 microseconds
D (11911) I2C: Transmit::Time taken: 25468 microseconds
D (11951) I2C: Receive::Time taken: 29372 microseconds
D (12051) I2C: Transmit::Time taken: 46310 microseconds
D (12051) I2C: Receive::Time taken: 56738 microseconds
D (12151) I2C: Transmit::Time taken: 25480 microseconds
D (12151) I2C: Receive::Time taken: 29434 microseconds
D (12251) I2C: Transmit::Time taken: 25455 microseconds
D (12281) I2C: Receive::Time taken: 29424 microseconds
D (12381) I2C: Transmit::Time taken: 25445 microseconds
D (12401) I2C: Receive::Time taken: 29341 microseconds
D (12501) I2C: Transmit::Time taken: 25533 microseconds
D (12501) I2C: Receive::Time taken: 29453 microseconds
D (12601) I2C: Transmit::Time taken: 25493 microseconds
D (12611) I2C: Receive::Time taken: 29384 microseconds
D (12711) I2C: Transmit::Time taken: 25425 microseconds
D (12741) I2C: Receive::Time taken: 29428 microseconds
I (12841) UART: BOTH TASKS ENDED, calculating metrics
I (12841) RESULT_I2C: ======== Average Time ========
I (12841) RESULT_I2C: Average read time: 40290 us
I (12841) RESULT_I2C: Average write time: 29813 us
I (12841) RESULT_I2C: Standard deviation of read times: 30270.3819 us
I (12851) RESULT_I2C: Standard deviation of write times: 8491.6111 us
I (12851) RESULT_I2C: ============ WCET ============
I (12861) RESULT_I2C: Read WCET: 198674 us
I (12861) RESULT_I2C: Write WCET: 46341 us
I (12871) RESULT_I2C: ========== Deadlines =========
I (12871) RESULT_I2C: Read Deadlines missed (100 ms): 3
I (12881) RESULT_I2C: Write Deadlines missed (100 ms): 0
I (12881) RESULT_I2C: ============ FINISH ===========
I (12891) main_task: Returned from app_main()
```

3. SPI

```
D (11800) UART: Receive::Time taken: 99862 microseconds
D (11800) UART: Transmit::Time taken: 278 microseconds
D (11900) UART: Receive::Time taken: 99862 microseconds
D (11900) UART: Transmit::Time taken: 278 microseconds
D (12000) UART: Receive::Time taken: 99862 microseconds
D (12000) UART: Transmit::Time taken: 278 microseconds
D (12100) UART: Receive::Time taken: 99862 microseconds
D (12100) UART: Transmit::Time taken: 278 microseconds
D (12200) UART: Receive::Time taken: 99862 microseconds
D (12200) UART: Transmit::Time taken: 278 microseconds
D (12300) UART: Receive::Time taken: 99862 microseconds
D (12300) UART: Transmit::Time taken: 278 microseconds
D (12400) UART: Receive::Time taken: 99862 microseconds
D (12400) UART: Transmit::Time taken: 278 microseconds
D (12500) UART: Receive::Time taken: 99862 microseconds
D (12500) UART: Transmit::Time taken: 278 microseconds
D (12600) UART: Receive::Time taken: 99862 microseconds
D (12600) UART: Transmit::Time taken: 278 microseconds
D (12700) UART: Receive::Time taken: 99862 microseconds
D (12700) UART: Transmit::Time taken: 278 microseconds
D (12800) UART: Receive::Time taken: 99862 microseconds
D (12800) UART: Transmit::Time taken: 278 microseconds
D (12900) UART: Receive::Time taken: 99862 microseconds
D (12900) UART: Transmit::Time taken: 278 microseconds
D (13000) UART: Receive::Time taken: 99862 microseconds
I (13000) SPI: RX DONE
D (13000) UART: Transmit::Time taken: 354 microseconds
I (13100) SPI: TX DONE
I (13100) UART: BOTH TASKS ENDED, calculating metrics
I (13100) RESULT_UART: ======== Average Time ========
I (13100) RESULT_UART: Average read time: 99863 us
I (13100) RESULT_UART: Average write time: 277 us
I (13110) RESULT_UART: Standard deviation of read times: 1.0043 us
I (13120) RESULT_UART: Standard deviation of write times: 7.2496 us
I (13120) RESULT_UART: ============ WCET ============
I (13130) RESULT_UART: Read WCET: 99864 us
I (13130) RESULT_UART: Write WCET: 354 us
I (13140) RESULT_UART: ========== Deadlines =========
I (13140) RESULT_UART: Read Deadlines missed (100 ms): 0
I (13150) RESULT_UART: Write Deadlines missed (100 ms): 0
I (13160) RESULT_UART: ============ FINISH ===========
I (13160) main_task: Returned from app_main()
```

با توجه به نحوه عملکرد پروتکل SPI و نیاز به ارسال ack از سمت slave، مقایسه مقادیر read time ارزش معنی داری برای ما ندارد. اما با مقایسه مقادیر write time داریم:



مشخص است که پروتکل SPI با فاصله چشمگیری از دو پروتکل دیگر پیشی گرفته و سرعتش با آن‌ها قابل مقایسه نیست. علاوه بر این در مثال اجرا شده پروتکل SPI هیچ ددلاینی را از دست نداد، در حالی که این مسئله برای پروتکل‌های دیگر درست نمی‌باشد.

در نهایت، contribution اصلی ما، ایجاد یک framework و کد modular برای نوشتن بنچمارک‌های دیگر و هموارسازی راه برای پروژه‌های آینده بود. جزئیات بیش‌تر کد در بخش Readme آن موجود هستند.