UNIVERSITY OF VICTORIA

Faculty of Engineering

SPRING 2025 ENGR 446 Final Report

Viability of Modern Firmware Platforms for Mechatronic Applications

University of Victoria
Faculty of Engineering
Victoria, British Columbia

Benjamin Say
V00927942
ENGR 446
Electrical Engineering
Email: benbsay@gmail.com

April 11th, 2025

In partial fulfillment of the requirements of the B.Eng. Degree

Benjamin B. Say
4048 Gordon Head Rd.
Victoria, BC
V8N 3X9

April 11th, 2025

Mr. Monty Raisinghani
Faculty of Engineering
University of Victoria
P.O. Box 1700
Victoria, B.C. V8W 2Y2

Dear Mr. Raisinghani,

Please accept the attached technical report “Viability of Modern Firmware Platforms for Mechatronic Applications” in partial fulfillment of the University of Victoria’s ENGR 446 requirements.

I am currently completing my sixth year of electrical engineering at the University of Victoria, with a focus on mechatronic systems. During my time both studying and completing co-ops in fulfilment of this degree, I have observed the importance of selecting the correct software framework before attempting to implement a mechatronic design, but I have often been left wondering how to choose between the available frameworks. The attached report will provide a comprehensive performance analysis of popular firmware platforms C++, Rust, CircuitPython, and MicroPython in terms of GPIO delay, processing speed, and memory allocation speed.

Thank you for taking the time to evaluate my submission, and I look forward to hearing your feedback.

Sincerely,

Benjamin B. Say
Electrical Engineering

Table of Contents

Table of Figures

List of Tables

Executive Summary

There is no existing comparison between different embedded firmware development platforms for the RP2040, or even for any microprocessor. This report evaluates the performance of C++, Rust, MicroPython, and CircuitPython for mechatronic applications using the RP2040 microprocessor. These platforms were ranked by analyzing fifty iterations of tests for GPIO latency, memory allocation speed, and processing efficiency (measured via bubble sort and floating-point Fibonacci calculations).

C++ emerged as the top-performing platform, excelling in all tested categories and making it ideal for all high-performance mechatronic systems. Rust provides slightly less performance but provides inherent memory safety, lagging behind C++ in speed (76–167% slower in tests) and exhibiting higher GPIO latency (105% slower).

Python-based platforms (MicroPython and CircuitPython) are significantly slower than both C++ and Rust (up to 10615% in bubble sort tests) due to dynamic typing overhead but offer ease of use and rapid prototyping capabilities. CircuitPython outperformed MicroPython in floating-point operations, while MicroPython was better for array manipulation.

The results show a clear performance-versus-usability trade-off, emphasizing the need to align platform choice with project priorities. Future work should explore power consumption, interrupt latency, and library availability to further refine recommendations.

Glossary

Term	Definition
ADC	Analog to digital converter.
Array	A programming object used to store a list of values.
Bubble sort	A simple sorting algorithm that compares every pair of adjacent values in an array, swapping them if they are in the incorrect order.
C++	A general-purpose programming language created by Danish computer scientist Bjarne Stroustrup in 1985.
CircuitPython	CircuitPython is an open-source derivative of the MicroPython programming language supported by Adafruit Industries.
Decision matrix	A decision matrix is a list of values in rows and columns that allows comparison between potential solutions.
Embedded System	A specialized computer system designed to control the mechanical and electrical functions of a device or system.
Floating-point	A numerical data type used to represent real numbers, including those with fractional parts.
GPIO pin	General purpose input output pin
Latency	The delay before a transfer of data begins following an instruction for its transfer.
Library	A set of packages that include functions and objects intended for use in a codebase.
Mechatronic	A multidisciplinary field that combines mechanical engineering, electronics, and computing to design and develop complex systems
Microprocessor	An integrated circuit that contains all the functions of a central processing unit of a computer.
MicroPython	MicroPython is a programming language largely compatible with Python 3 that is optimized to run on a microcontroller.
Pico	Raspberry Pi's first board based upon a single microcontroller chip; the RP2040, which was designed by Raspberry Pi in the UK.
Python	A general-purpose programming language emphasizing readability.
Rising edge	A low level to high level signal transition.
RP2040	A 32-bit dual-core ARM microcontroller integrated circuit by Raspberry Pi Ltd released in January 2021.
Rust	A general-purpose programming language emphasizing performance, type safety, and concurrency.

1. Introduction

In mechatronics engineering, microprocessing units (MPUs) are frequently used to implement control systems because how cheap, easy, and effective they are. This popularity has resulted in a massive range of MPU options in the hobbyist and professional markets, and consequently a wide range of firmware platforms that can be used to control them.

1.1. Background

There is plenty of information available about the capabilities of different MPUs, for which performance varies depending on the chosen architecture and features. Table 1 provides an example comparison between three different MPUs that all use the RP2040 chip [1]. However, there is comparatively little documentation from developers on the performance differences between different firmware platforms. There are a multitude of firmware platforms available, each presenting different advantages and disadvantages for firmware development. In the future, it would be useful to have a performance comparison between them so that developers may choose the appropriate platform for their project.

Table 1: RP2040 MPU Comparison

	Arduino Nano RP2040 Connect [2]	**Raspberry Pi Pico 1 [3]**	**Adafruit Feather RP2040 [4]**
# GPIO Pins	20	23	21
ADC channels	8 x 12 bit	3 x 12 bit	4 x 12 bit
Input voltage range	4 – 20 V	1.8 – 5.5 V	3.7/4.2V LiPo battery *or* 5V via USB C
Flash memory	16 MB	2 MB	8 MB
Max Current Supply	800 mA	~300 mA	500 mA

In a February 2024 interview, “all [Raspberry Pi CEO Eben Upton] could confirm [was] that "it's more than that," referring to four million Pico sales.” [5] At that time, the Pico 2 had not been released, meaning all those MPUs were based on the RP2040 chip. The RP2040 is widely used for prototyping and robotics applications, and as shown in Table 1, it is also sold as part of many other MPUs. It is often chosen by educational organizations for students to learn about embedded system and mechatronic development, and as such it remains a cornerstone of low-level computing. Determining the best overall firmware platform for development on this chip could not only improve the efficiency of future projects using the RP2040, but also help introduce students to more efficient development techniques by providing them with firmware systems that allow them to take full advantage of the chip’s resources.

While there have been a few recent reviews and comparisons of different IoT and embedded system development platform hardware [6][7] that provide a valuable perspective on , there have been no formal reviews and tests of the different firmware platforms. This paper aims to provide that.

1.2. Objective

There is no existing comparison between different embedded firmware development platforms for the RP2040, or even for any microprocessor. This means that the choice is often made by considering the ease of use alone, and performance is left unconsidered when it could provide significant efficiency to the project. There needs to be a way to evaluate which firmware platform to use depending on the project criteria.

This report will provide a comprehensive performance comparison between firmware platforms for the RP2040 in terms of GPIO delay, processing speed, and memory allocation speed.

There are some significantly impactful considerations for firmware platforms that will not be reviewed or tested as part of this research. Importantly, power consumption will not be considered due to the assumption that the control system's consumption will always be insignificant compared to that of the rest of the system. This paper is concerned with mechatronic system and prototyping applications, where the MPU usually controls the use of some number of electric motors and sensors, which will always dwarf the MPU in power consumption. The differences in power consumption are therefore insignificant.

The differences in safety (both cyber and physical) between software platforms will not be tested as the security concerns vary mostly based on the application of the MPU and not the platform itself. While some programming languages are known to be less "safe" than others, it will be assumed that the programmer either possesses sufficient skill to ensure their application is safe, is responsible for their own safety if they are a hobbyist, or is being guided by a professional if they are a student.

2. Discussion

There are various potential solutions that are all uniquely popular, however this paper will only consider four of the most popular for the sake of brevity. The platforms that will be considered are as follows:

1. C++
2. Rust
3. MicroPython
4. CircuitPython

The Raspberry Pi foundation has a popular SDK for C and C++ development, and in this paper, it will be used for C++ development due to the wider set of tools that C++ provides over C. C++ is also already widely used in Arduino development (another popular embedded systems platform), so it is a good choice to test due to it's cross-platform compatibility.

Rust for embedded systems has become more popular in recent years due to it's strong typing system and set of rules that explicitly prevent unsafe memory operations, creating unprecedented safety with no effort from the developer. According to Vandervelden et al., there are multiple papers that conclude that Rust could be a good replacement for C in the future due to its comparable performance and unparalleled safety [8]. This makes it an excellent candidate for testing.

MicroPython and CircuitPython are both Python platforms that are popular for prototyping and IoT devices due to Python's ease of use. MicroPython is recommended by Raspberry Pi for Pico boards, making it a default for the RP2040, and CircuitPython is the counterpart to MicroPython developed and provided by Adafruit industries. CircuitPython is popular for educational projects and hobbyist devices for its easy compatibility and library access for other Adafruit products, including sensors, motors, and other chipsets.

2.1. Assessment Methodology

The best overall firmware platform (referred to as a "platform" herein) and the system with which to determine the best solution for a given project will be found through a set of tests. For each test, a script will be written for each platform that is as similar as possible. Then, the program will be uploaded to a Raspberry Pi Pico 1 (referred to as a "Pico" herein) and monitored during runtime. The Pico was chosen because it is the most limited RP2040 platform, designed to have only the most necessary components included on the board to allow the RP2040 to function by the company that made it. This is imperative, because it ensures that the platform has the absolute minimum number of things to manage, in the case that it is attempting to manage peripherals in the background and therefore limiting its own performance. The tests are therefore being performed on the minimum required hardware at minimum load, improving test accuracy. In the case that one of the boards fails for any reason, multiple Picos have been purchased from the same source and batch to ensure consistency across tests.

First, the IO processing speed will be tested. This will be done by sending a square step signal from a separate Pico dedicated to timing the tests and waiting for a returned square step signal from the tested Pico, and the time between the sent step and the received step will be compared between platforms. The tested Pico will wait for the step response, continually polling the selected input pin, and return a square step signal as soon as possible on a different GPIO pin. The IO response time is a relevant statistic because it can determine how quickly a mechatronic system can respond to a change in its environment and how quickly a device can communicate with other parts of a system. It would be slightly more valuable to test using an interrupt-based system, but unfortunately CircuitPython does not support interrupts without async-await, which is not the same as a hardware-based interrupt. This same timing method will be used for all of the other tests; the tested processor will begin processing the test when it receives a rising edge signal from the control processor, and will indicate completion by sending a rising edge signal in return.

Next, memory allocation speed will be measured for each of the platforms by timing the allocation of an array of size 800. Different allocation methods for different languages result in different times required to allocate memory for data, which can significantly impact the performance of a system. Robotic and mechatronic systems often must store large amounts of information to evaluate system state and respond accordingly, and as such, the time taken to allocate space for data can be as impactful as the time taken to read or calculate the data itself. The impact of the selected firmware platform on this statistic may be a relevant consideration when choosing a development strategy.

Finally, processing speed will be evaluated using two separate tests. The first is a series of precise signed floating-point operations, which are notoriously difficult for low level systems. This will be done by computing a Fibonacci sequence of length 800, starting with 0.0001 and 1.0001 as the start of the sequence. This setup ensures that the same series of floating-point operations is performed for all tests at significant precision. The second test will be a bubble sort algorithm for an array of size 1000. The ordered array will be reversed before the test begins to ensure that every test uses the same unordered sequence, and because that setup requires the most swaps. The algorithm will be implemented with no optimization to maximize performance impact and therefore maximize program runtime, allowing the testing to exemplify the differences between platforms in terms of processing speed. Processing speed is a necessary test because it determines how the platform responds to processing large amounts of data, which is useful when there are a significant number of sensors or communications happening in a system.

Setup for the tests will be completed before the test begins rather than being part of the test time. For each platform, setup will be completed first, and then each test will be run sequentially to minimize total runtime. An example of the code running on the tested processor is shown in CircuitPython in Appendix A – Example Test Code in CircuitPython A, with CircuitPython being chosen as an example because it is the simplest and shortest of the test code versions.

2.2. Results

The following data was gathered over a period of one hour. To ensure consistency by removing optimizations from Python's interpreter at runtime, each test was run once on each platform and then all the tested processors were hard reset by cycling their shared power supply. Each test was completed 50 times to ensure accuracy. The testing setup is shown below in Figure 1.

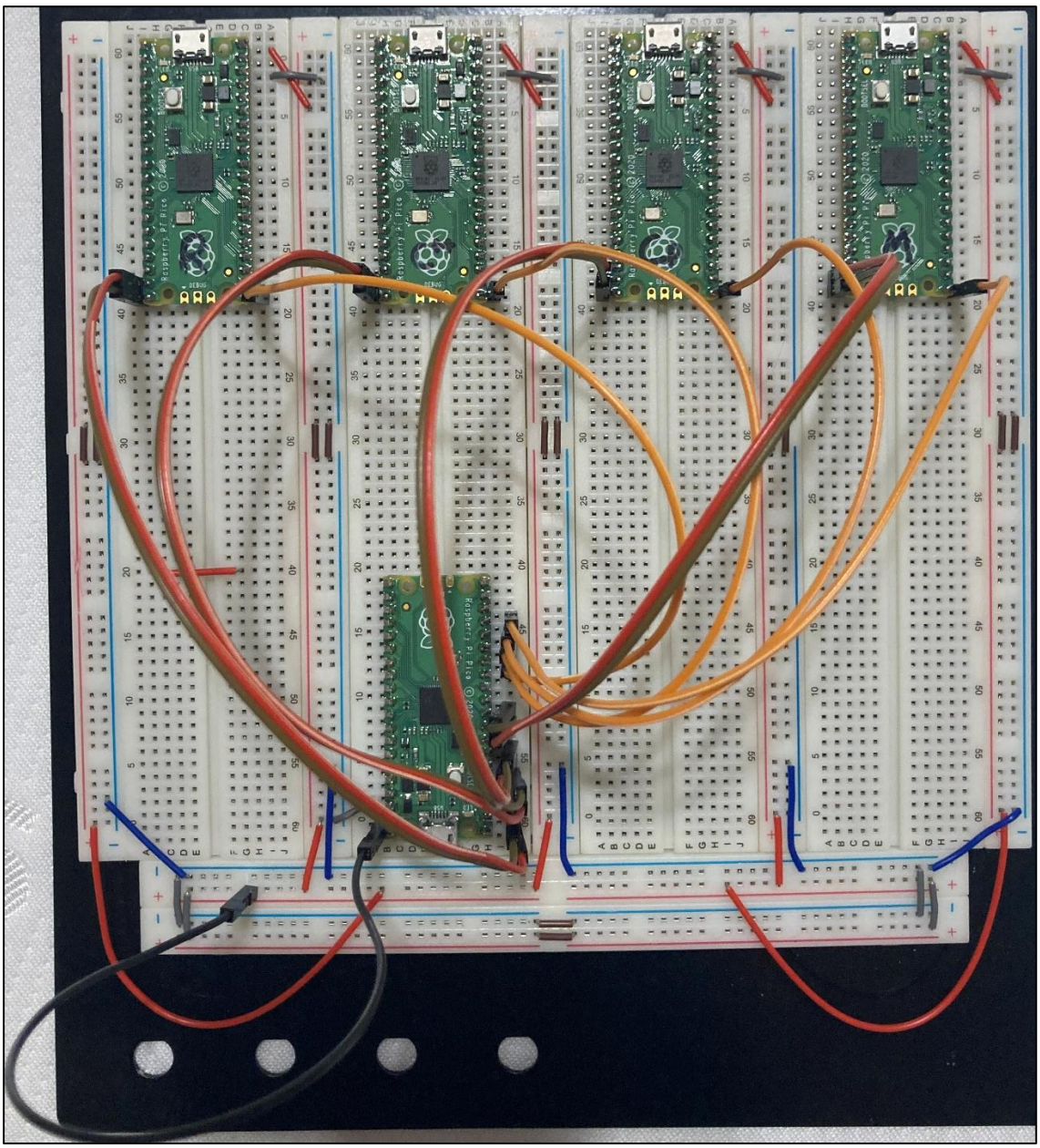

Figure 1: Testing Setup

2.2.1. GPIO Speed

GPIO speed tests yielded the data shown in Table 2. Percentage increase is calculated from the difference between the time of a given platform and the time of the platform before it.

Table 2: GPIO Latency

Platform	Average Time (ms)	Variance (ms^2)	Increase (%)
C++	0.0284	0.000003	-
CircuitPython	0.0290	0.000004	2%
MicroPython	0.0294	0.000003	1%
Rust	0.0601	0.000003	105%

Despite each of these times being insignificant in terms of human perception, differences of milliseconds can be critical in time-limited procedures. This can be especially noticeable when many GPIO pins are being read at a time. PWM signals and various forms of serial communication rely on GPIO pins being set at accurate intervals, and delayed changes to GPIO voltage may result in data transmission errors.

The differences between C++, CircuitPython, and MicroPython are extremely small, with CircuitPython being only 2% slower on average compared to C++ and MicroPython being only 1% slower on average compared to CircuitPython. While this difference of fractions of microseconds will not usually impact the performance of a system, the difference between the leaders and the Rust framework is significant. The Rust framework saw an increase of 105% from MicroPython, which can likely be attributed to the use of Rust's "unwrap" function on calls to GPIO pins for both input and output values. This function ensures that the process halts safely when the GPIO object returns a value associated with an error in retrieving or setting a pin value.

While halting the program gracefully is slightly safer, good programming and circuit design practices in C++, CircuitPython, or MicroPython should be used over choosing the Rust framework when GPIO response speed is the most important deliverable of a design.

2.2.2. Memory Allocation Speed

Memory allocation speed tests yielded the data shown in Table 3. Percentage increase is calculated from the difference between the time of a given platform and the time of the platform before it.

Table 3: Allocation Speed for an 800-Index Array

Platform	Average Time (ms)	Variance (ms^2)	Increase (%)
C++	2.4657	0.000004	-
Rust	6.5822	0.000007	167%
MicroPython	393.3118	0.280852	5875%
CircuitPython	457.7326	26.729341	16%

While delays due to memory allocation can typically be optimized by allocating memory during system setup, processes like vector analysis and linear algebra (often used in robotics processing) often necessitate significant amounts memory allocation at runtime. Slow memory allocation can also result in significant latency for systems that do a lot of data analysis, such as any kind of image recognition.

The C++ platform is again the fastest, with Rust following it by approximately four milliseconds or a 167% increase. This appears to be a significant difference, but both Python-based frameworks were more than fifty times slower on average, and CircuitPython had a variance of $26\ ms^2$, which is more than can be attributed to measurement error considering the GPIO times discussed in section 2.2.1. MicroPython and CircuitPython are both significantly slower and significantly less reliable than the lower-level C++ and Rust frameworks, with CircuitPython being the slower of the two by 16%. This can be attributed to Python-based frameworks inheriting Python's dynamically typed variable system, which creates significant overhead when modifying or allocating memory. Python also assumes index sizes, which means that an array of n indexes in Python may use significantly more memory than an array of n indexes in C++, Rust, or any other statically typed languages.

Aside from the unlikely case where using a dynamically typed array is more optimal than a statically typed array, C++ or Rust should be used over MicroPython or CircuitPython when memory allocation speed is important to the performance of the system. For the most time-critical systems, C++ should be used over Rust. In the case where the functionality of a dynamically typed array outweighs the cost of the increased processing speed, MicroPython should be chosen due to its relative speed and reliability.

2.2.3. Processing Speed

Bubble sort processing speed tests yielded the data shown in Table 4. Fibonacci sequence processing speed tests yielded the data shown in Table 5. Percentage increase is calculated from the difference between the time of a given platform and the time of the platform before it.

Since processing speed can mean different things depending on the application, two different tests were used. The sorting tests are a popular for benchmarking because they make use of comparing operations and in-place array manipulation, and a floating-point Fibonacci sequence calculation was chosen to ensure consistency between tests and determine relative speed in floating-point operations, which are often slow on embedded systems.

Table 4: Bubble Sort Time for a Reversed 1000-Index Array

Platform	Average Time (ms)	Variance (ms^2)	Increase (%)
C++	99.9573	0.000006	-
Rust	176.1393	0.000008	76%
MicroPython	18874.0861	0.000024	10615%
CircuitPython	20892.6039	0.000028	11%

In the bubble sort test, C++ takes the lead again with Rust following at a 76% increase in processing time. MicroPython is 9.05 times slower than Rust, and CircuitPython is again the slowest framework at only 11% slower than MicroPython. Due to the nature of the bubble sort test being both a compare and array-manipulation heavy test, the reason for the significant decrease in processing speed when moving to Python based languages is unclear, but it may be due to the dynamically typed arrays used for this test creating overhead on compare operations.

Table 5: Processing Time for a Fibonacci Sequence of Length 800

Platform	Average Time (ms)	Variance (ms^2)	Increase (%)
C++	0.8245	0.000007	-
Rust	2.0510	0.000005	149%
CircuitPython	20.6186	0.000076	905%
MicroPython	38.2682	0.000024	86%

In the floating-point Fibonacci sequence test, C++ is again the fastest, and Rust follows at a 149% increase. However, CircuitPython is faster than MicroPython this time, with MicroPython being 86% slower than CircuitPython in this test.

In terms of floating-point processing, C++ is the fastest option, outperforming every other framework by a significant margin for both tests. Rust, CircuitPython, and MicroPython are the next best options, in that order. In the case where floating-point operations are calculated frequently, such as calculating values from ADC output, frameworks should be chosen in that order.

In terms of general processing speed, C++ is clearly the best option, followed closely by Rust. CircuitPython and MicroPython are generally very similar, with CircuitPython being better than MicroPython in floating-point operations.

2.3. Analysis

To determine the best overall framework, the test performances will be ranked using a decision matrix. The frameworks will be given a value between one and four for each test based on their test ranking, and each test score will be multiplied by a weight determined by the test's value to the user from one to four. The more important the test or the higher a framework's placement in the test, the higher the value will be. The unweighted decision matrix is shown below in Table 6.

Table 6: Unweighted Decision Matrix

	GPIO Latency	Memory Allocation	Bubble Sort	Fibonacci Sequence	Score
C++	4	4	4	4	16
Rust	1	3	3	3	10
MicroPython	2	2	2	1	9
CircuitPython	3	1	1	2	7

To determine the weighting for this analysis, a general mechatronic system will be considered. The platforms differed most in processing speed, and the majority of clock cycles in most mechatronic systems are taken up by data analysis or calculations. Processing tests should therefore be weighted highest, with floating point math being generally more useful (and therefore weighted higher) due to its application in analog data analysis, which is often used for temperature sensors, proximity sensors, reflection sensors, etc. The difference in GPIO latency between the platforms is in the order of microseconds, and most mechatronic systems do not need response times faster than a tenth of a millisecond, so memory allocation speed will be given the third most weight and GPIO latency will be given the least amount of weight. The weighted decision matrix is shown in Table 7.

Table 7: Weighted Decision Matrix

	GPIO Latency (1)	Memory Allocation (2)	Bubble Sort (3)	Fibonacci Sequence (4)	Score
C++	$4 * 1 = 4$	$4 * 2 = 8$	$4 * 3 = 12$	$4 * 4 = 16$	40
Rust	$1 * 1 = 1$	$3 * 2 = 6$	$3 * 3 = 9$	$3 * 4 = 12$	28
MicroPython	$2 * 1 = 2$	$2 * 2 = 4$	$2 * 3 = 6$	$1 * 4 = 4$	16
CircuitPython	$3 * 1 = 3$	$1 * 2 = 2$	$1 * 3 = 3$	$2 * 4 = 8$	16

From the results of the weighted decision matrix for a general mechatronic system, C++ is the best framework for most use cases. Rust is best after C++, while MicroPython and CircuitPython are tied but significantly lower ranked.

The use of MicroPython compared to CircuitPython should therefore be decided by the kind of peripherals used in the system. CircuitPython is developed by Adafruit Industries, which provides an extensive, accessible, and easy to use library of packages for the peripherals they sell, while MicroPython is significantly less fleshed out. For a simple proof of concept, teaching tool, or amateur application, CircuitPython trades slow operation and patchy implementation (such as a lack of support for interrupts) for simplicity of use with Adafruit's own peripherals.

It is also worth noting that Rust is well known for being a completely memory safe language, making safety critical code easy to create. This can be a significant consideration in security-critical applications but was not explored in this paper for the sake of brevity.

This analysis provides an effective comparison of platforms when performance is critical but makes Python-based firmware frameworks seem ineffective and downright poor in terms of performance. However, for a proof of concept where development time and ease of use is more critical (especially for teaching or experimentation), Python-based frameworks can be both effective and fun if used correctly.

3. Conclusions

This report effectively provides an objective performance comparison of programming frameworks for the RP3040. For high performance mechatronic systems, C++ has been shown to be the best programming framework for performance, decreased latency, and fast memory allocation. It was first in every test by a significant margin while also being a well developed and time-tested framework that is already in use all over the world. The newer Rust programming framework follows closely behind C++, placing second in every test other than GPIO latency, where it places last, but has a lower barrier of entry to create memory safe code. MicroPython and CircuitPython – both Python-based frameworks – trail behind significantly, with MicroPython being slightly better at in-place array manipulation and CircuitPython being slightly better at floating point calculations.

To use this data to evaluate the correct platform for a different application, the weights of the decision matrix should be modified to fit the deliverables of the application.

The results highlight a clear trade-off between performance and ease of use, emphasizing the need for developers to align their platform choice with project goals, whether they are performance, safety, or development simplicity.

4. Recommendations

For performance-critical applications, use C++ to optimize processing speed and remove as much latency as possible from GPIO usage or complex calculations.

For safety focused applications where the user is not confident in their ability to create safe C++ code, use Rust to ensure safety with a small reduction in processing speed and increase in latency.

For prototyping and education, choose CircuitPython or MicroPython to reduce development time and simplify integration with peripherals, accepting slower performance as a trade-off. Choose CircuitPython where floating-point math is critical, and MicroPython where large arrays of data are being analyzed.

In the future, these tests should be run again if significant changes occur in any of the platforms, including but not limited to interrupt latency testing should CircuitPython gain interrupt support and updated processing tests as the Rust language develops. Power consumption data should also be measured for idle and running modes, and analysis of available libraries and packages could provide valuable information for prototyping applications.

5. References

[1] Raspberry Pi LTD, "RP2040 Datasheet," Datasheet 1, Jan. 2021 [Revised Oct. 2024], Available: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

[2] Arduino, "Arduino Nano RP2040 Connect," Datasheet 1, May 2020 [Revised Feb. 2025] Available: https://docs.arduino.cc/resources/datasheets/ABX00053-datasheet.pdf

[3] Raspberry Pi LTD, "Raspberry Pi Pico Datasheet," Datasheet 1, Jan. 2021 [Revised Oct. 2024] Available: https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf

[4] Adafruit Industries, "Introducing Adafruit Feather RP2040," Datasheet 1, Feb 2025 Available: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-feather-rp2040-pico.pdf

[5] L. Pounder, "Raspberry Pi celebrates 12 years as sales break 61 million units," Tom's Hardware, Feb. 29, 2024. Available: https://www.tomshardware.com/raspberry-pi/raspberry-pi-celebrates-12-years-as-sales-break-61-million-units

[6] A. Mishra, "Embedded Development Platforms To Design Prototypes Of Internet Of Things (Iot) Applications: A Study," International Journal of Research in Advent Technology, vol. 7, no. 4, Apr. 2019. Available: https://www.researchgate.net/profile/Ayaskanta-Mishra/publication/332863259_Embedded_Development_Platforms_To_Design_Prototypes_Of_Internet_Of_Things_IoT_Applications_A_Study/links/5ccda89692851c4eab83478e/Embedded-Development-Platforms-To-Design-Prototypes-Of-Internet-Of-Things-IoT-Applications-A-Study.pdf

[7] D. Singh, A. Sandhu, A. Thakur, and N. Priyank, "An Overview of IoT Hardware Development Platforms," International Journal on Emerging Technologies, vol. 11, no. 5, Aug. 2020. Available: https://www.researchgate.net/profile/Dhawan-Singh/publication/344207338_An_Overview_of_IoT_Hardware_Development_Platforms/links/5f5b92d1299bf1d43cf9e447/An-Overview-of-IoT-Hardware-Development-Platforms.pdf

[8] T. Vandervelden, R. D. Smet, D. Deac, K. Steenhaut, and A. Braeken, "Overview of Embedded Rust Operating Systems and Frameworks," Sensors, vol. 24, no. 17, pp. 5818–5818, Sep. 2024, Available: https://www.mdpi.com/1424-8220/24/17/5818

Appendix A – Example Test Code in CircuitPython

```
import time
import board
import digitalio
import array

ARRAY_SIZE = 1000
MALLOC_AMT = 800
FIB_SIZE = 800
INPUT_PIN = board.GP14
OUTPUT_PIN = board.GP15
READY_PIN = board.GP16


def is_sorted(arr, n):
    for i in range(n - 1):
        if arr[i] > arr[i + 1]:
            return False
    return True


# Set the LED to be an output
led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

# Set the input pin
input_pin = digitalio.DigitalInOut(INPUT_PIN)
input_pin.direction = digitalio.Direction.INPUT
input_pin.pull = digitalio.Pull.DOWN

# Set the output pin
output_pin = digitalio.DigitalInOut(OUTPUT_PIN)
output_pin.direction = digitalio.Direction.OUTPUT
output_pin.value = False

# Set ready pin
ready_pin = digitalio.DigitalInOut(READY_PIN)
ready_pin.direction = digitalio.Direction.OUTPUT
ready_pin.value = False

################################################################################################################
######################################## CODE FOR TESTING STARTS HERE ##########################################
################################################################################################################

while True:

    # create a reversed array for the bubble sort test, which is the worst case
    numbers = array.array('i', [ARRAY_SIZE - i for i in range(ARRAY_SIZE)])

    # allocate array for fibonacci test
    fib = [0.0] * FIB_SIZE
    fib[0] = 0.0001
    fib[1] = 1.0001

    # indicate setup complete
    for i in range(2):
        led.value = True
        time.sleep(0.5)
        led.value = False
        time.sleep(0.5)

    # indicate ready for tests
    ready_pin.value = True
    output_pin.value = False

    ################### GPIO SPEED TEST ###########################################################################

    while not input_pin.value: # wait for continue pin to go HI
        pass
    output_pin.value = True    # indicate test complete
    while input_pin.value:     # wait for continue pin to go LOW
        pass
    time.sleep(1)
    output_pin.value = False   # indicate ready for next test
```

```
    ################### BUBBLE SORT PROCESSING TEST ############################################################

    while not input_pin.value: # wait for continue pin to go HI
        pass

    for i in range(ARRAY_SIZE): # compute bubble sort
        for j in range(ARRAY_SIZE - i - 1):
            if numbers[j] > numbers[j + 1]:
                numbers[j], numbers[j + 1] = numbers[j + 1], numbers[j]

    output_pin.value = True # indicate test complete

    # check if array is properly sorted
    if not is_sorted(numbers, ARRAY_SIZE):
        while True:  # halt program if improperly sorted
            led.value = not led.value
            time.sleep(0.1)

    while input_pin.value:   # wait for continue pin to go LOW
        pass
    time.sleep(1)
    output_pin.value = False # indicate ready for next test

    ################### HEAP ALLOC MEMORY TEST #################################################################

    while not input_pin.value: # wait for continue pin to go HI
        pass

    for _ in range(MALLOC_AMT):
        lst = [i for i in range(128)]   # simulate memory allocation
        del lst

    output_pin.value = True  # indicate test complete
    while input_pin.value:   # wait for continue pin to go LOW
        pass
    time.sleep(1)
    output_pin.value = False # indicate ready for next test

    ################### FIBONACCI PERFORMANCE TEST ##############################################################

    while not input_pin.value: # wait for continue pin to go HI
        pass

    for i in range(2, FIB_SIZE):
        fib[i] = fib[i - 1] + fib[i - 2]

    output_pin.value = True  # indicate test complete
    while input_pin.value:   # wait for continue pin to go LOW
        pass
    time.sleep(1)
    output_pin.value = False # indicate ready for next test

    # Indicate testing complete
    ready_pin.value = False

    # attempt to keep the interpreter from optimizing out the fib code
    st = ""
    for i in fib:
        st += str(i)
    print(st)

    for _ in range(5):
        led.value = True
        time.sleep(0.1)
        led.value = False
        time.sleep(0.1)

    time.sleep(1)

#############################################################################################################
######################################## CODE FOR TESTING ENDS HERE ########################################
#############################################################################################################
```