electronics

MDPI

Article

Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller

Ignas Plauska, Agnius Liutkevičius * and Audronė Janavičiūtė

Department of Computer Sciences, Kaunas University of Technology, 44249 Kaunas, Lithuania
* Correspondence: agnius.liutkevicius@ktu.lt

Abstract: The rapid growth of the Internet of Things (IoT) and its applications requires high computational efficiency, low-cost, and low-power solutions for various IoT devices. These include a wide range of microcontrollers that are used to collect, process, and transmit IoT data. ESP32 is a microcontroller with built-in wireless connectivity, suitable for various IoT applications. The ESP32 chip is gaining more popularity, both in academia and in the developer community, supported by a number of software libraries and programming languages. While low- and middle-level languages, such as C/C++ and Rust, are believed to be the most efficient, TinyGo and MicroPython are more developer-friendly low-complexity languages, suitable for beginners and allowing more rapid coding. This paper evaluates the efficiency of the available ESP32 programming languages, namely C/C++, MicroPython, Rust, and TinyGo, by comparing their execution performance. Several popular data and signal processing algorithms were implemented in these languages, and their execution times were compared: Fast Fourier Transform (FFT), Cyclic Redundancy Check (CRC), Secure Hash Algorithm (SHA), Infinite Impulse Response (IIR), and Finite Impulse Response (FIR) filters. The results show that the C/C++ implementations were fastest in most cases, closely followed by TinyGo and Rust, while MicroPython programs were many times slower than implementations in other programming languages. Therefore, the C/C++, TinyGo, and Rust languages are more suitable when execution and response time are the key factors, while Python can be used for less strict system requirements, enabling a faster and less complicated development process.

Keywords: performance evaluation; microcontroller; ESP32; C/C++; MicroPython; TinyGo; Rust

Citation: Plauska, I.; Liutkevičius, A.; Janavičiūtė, A. Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics* **2023**, *12*, 143. https://doi.org/10.3390/electronics12010143

Academic Editors: Abdelhafid El Ouardi, Sergio Rodriguez and Bastien Vincke

Received: 25 November 2022
Revised: 16 December 2022
Accepted: 22 December 2022
Published: 28 December 2022

1. Introduction

The increasingly widespread IoT applications related to the development of various embedded systems and signal processing tasks require specialized hardware. This technical equipment must be characterized by small dimensions, low energy consumption, efficient memory use, and sufficient performance for the implementation of different signal processing functions. The main role in this case is played by various microcontrollers, which usually collect data from sensors and end-user devices, process those data, and forward results to higher-level systems. Currently, the market offers a whole range of specialized signal processing microcontrollers specially adapted for IoT tasks. One of the popular choices has become the ESP32 microcontroller, which is attractive to developers due to its technical characteristics and good software support, as well as the ability to use various programming languages. As concluded by [1], ESP32 is an excellent option for IoT devices due to the price and performance achieved by a dual core structure and a significant extension of operational features.

Recent scientific publications have proven that ESP32 chips are widely used in various fields. Aghenta and Iqbal proposed several SCADA systems [2,3] that use the ESP32 microcontroller for sensor data processing and brokering. Allafi and Iqbal [4] used ESP32 for the implementation of a low-cost web server to monitor and collect real-time

photovoltaic data. Carducci et al. [5] utilized the ESP32 microcontroller for the implementation of a building automation system. Sangeethalakshmi et al. [6] created an IoT patient health monitoring system consisting of sensors, a data acquisition unit, and an ESP32 microcontroller. Taştan and Gökozan [7] proposed a real-time indoor air quality monitoring system, where air quality data is measured by the sensor array monitored via the 32-bit ESP32 Wi-Fi controller. An ESP32-based solar irrigation pumping control system was proposed by Biswas and Iqbal [8], while Hangan et al. [9] reviewed information and communication technology systems for monitoring, control, and management of water resources, and concluded that although the Raspberry Pi 4B shows the highest processing score and an average interfacing score, the ESP32 is the most versatile for IoT applications, showing the highest overall score.

Since ESP32 microcontrollers are quite popular among researchers and system developers, several programming languages are available for ESP32 programming. Although C/C++ is the leading programming language for the development of IoT and embedded systems, there is also the possibility to use Rust, TinyGo, MicroPython, CircuitPython, and even JavaScript for ESP32-based systems as well. Few studies related to the evaluation of the performance of programs written in some of these languages have been conducted on ESP32 and other IoT platforms. Usually, such studies use the C/C++ language as the gold standard and compare its performance with other languages. Ionescu and Enescu [10] investigated the performance of the MicroPython and C languages on ESP32 and STM32 microcontrollers, finding that the MicroPython performance level is lower, but that it has better portability and is more suitable for inexperienced students. Dokic et al. [11] compared MicroPython with Arduino C and concluded that Arduino IDE is a faster platform than MicroPython for the development of neural networks on edge devices. Grunert [12] discussed the advantages and disadvantages of using the JavaScript language for microcontroller development and compared several JavaScript engines, suitable for ESP32 development, but left the evaluation of the performance and memory aspects of the interpreters for future work. To our knowledge, no one has so far conducted a performance evaluation of the Rust and Golang (TinyGo) programming languages on ESP32 or similar platforms. Therefore, this paper evaluates the efficiency of the C/C++, MicroPython, Rust, and TinyGo programming languages by comparing their execution performance on the ESP32 platform.

The previously cited performance evaluation study [10] used Secure Hash Algorithm (SHA-256) and Cyclic Redundancy Check (CRC-32) algorithms, while the authors of [11] implemented Machine Learning (ML) algorithms (neural networks) for comparison purposes. Security-related algorithms are also evaluated in Suárez-Albela et al.'s study [13], which compares Elliptic Curve Digital Signature Algorithm (ECDSA) and Rivest–Shamir–Adleman (RSA), using a resource-constrained IoT node based on the ESP32 system-on-chip. Further IoT and ESP32-related literature analysis shows that there are many applications of digital signal processing (DSP) algorithms, mainly Fast Fourier Transform (FFT) and various filtering algorithms as well. For example, Kodithuwakku et al. [14] used FFT for patient monitoring utilizing the ESP32 development environment. Fabregat et al. [15] used FFT to create a real-time sound-source localization system implemented on the ESP32 microcontroller. Shinde and Mundada [16] used ESP32-based FFT implementation to develop a bike engine health monitoring system.

Since many IoT applications include signal processing and security-related algorithms which are computationally demanding, the comparison of the programming languages in this paper is based on several popular data and signal processing algorithms, including FFT, CRC, SHA, Infinite Impulse Response (IIR), and Finite Impulse Response (FIR) filters. These algorithms were implemented in C/C++, MicroPython, Rust, and TinyGo programming languages, and their execution times were compared. The aim of this research is to find out whether more user-friendly (higher-level, less error-prone, with simpler and less code-demanding syntax) programming languages, including

MicroPython, Rust, and TinyGo, have similar execution performance compared to C/C++, which is officially supported by ESP32.

This study explores, for the first time, the execution performance of the quite popular Rust and TinyGo languages and compares them with widely used C and Python (MicroPython). The present study is expected to contribute to our understanding of the suitability of the programming languages available for the development of IoT systems using the ESP32 platform. The results of the study are important both for the developers of IoT systems and academia, which use ESP32 microcontrollers extensively for research and teaching purposes. They show that programs written in the C programming language are fastest in most cases, but this advantage is not very great compared to TinyGo- and Rust-written programs. Moreover, in some cases, C programs are outperformed by both C and Rust implementations.

This article is organized as follows. Section 2 presents the performance evaluation methodology and experimental setup. Section 3 shows the results of the execution comparison of digital signal processing algorithms implemented in selected programming languages. Section 4 discusses the results and compares them with previous research.

2. Materials and Methods

2.1. Programming Language Features

The most important aspect when choosing a microcontroller programming language is the support of its hardware and peripherals, such as GPIO pins and communication modules like WiFi, Bluetooth, or SPI. Microcontroller vendors almost always offer hardware abstraction libraries (HALs) to access specific registers and peripherals. Without such libraries, even an otherwise very powerful language is not useful for development on a chosen platform, in this case, ESP32.

Another aspect is the support for the language itself on a device. It is not uncommon that more advanced features of a high-level programming language are limited or not supported at all on certain platforms.

Memory management is also an important feature of any programming language, as it is closely related to overall safety and performance of any project developed in that language. The three most common memory management types are automatic, manual, and garbage collector-based. In automatic management, the memory is allocated by the compiler without an explicit instruction from the programmer, mainly occurring with stack management. In contrast, manual memory management requires the programmer to do all the hard work of allocating and deallocating memory. This is usually used for a heap. Finally, with garbage collection, memory allocation can be manual, but deallocation is performed automatically at certain intervals. While manual memory management can introduce bugs and safety issues, garbage collection comes with a performance penalty and unpredictability, since the runtime environment must stop the execution of the actual code to search for memory that is not in use. This is especially important in real-time applications, where execution times must be well known and controlled.

It is Important to consider what compiler and, by extension, toolchain will be used to compile the code for ESP32. In embedded environments, the ability to optimize code size is usually important. For example, GNU Compiler Collection (GCC) offers one optimization level of size (-Os), while Low Level Virtual Machine (LLVM) compiling technology offers two optimization levels (-Os and -Oz). The alternative is an interpreted language, which allows one to immediately run the code on a platform with a suitable interpreter.

Finally, a programming language runtime system handles tasks that include setting and managing stack and heap, handling garbage collection, threading, and other dynamic features available in that language [17]. The features of runtime are provided by the standard library of the language (or interpreter for an interpreted language) and, by extension, an operating system (typically, a real-time operating system on embedded hardware). The

so called bare-metal runtime is also possible, where no OS, or even standard library support, is available; however, it is usually limited in features.

2.2. Programming Languages Used for Evaluation

This paper evaluates four programming languages available for ESP32: C/C++, Rust, TinyGo, and MicroPython. Each language represents a different set of important features relevant to embedded and real-time programming, which are provided in Table 1.

Table 1. Overview of the programming languages compared on ESP32.

Language	ESP32 Peripherals Support	Language Features Support on ESP32	Memory Management	Compiler for ESP32	Runtime System on ESP32
C/C++	Full	Full	Manual/Automatic	GCC	C/C++ standard library on FreeRTOS
Rust	High	High	Fully automatic	LLVM	Rust standard library on FreeRTOS
TinyGo	Limited	Limited	Garbage collection	LLVM	Go standard library on bare-metal
MicroPython	Medium	Limited	Garbage collection	Interpreted	MicroPython interpreter on FreeRTOS

The C language was developed in 1978 by Brian Kernighan and Dennis Ritchie [18]. Considered the standard language in low-level system programming, it is also usually the default choice for embedded programming, since most microcontroller vendors provide tools for their products primarily in C. It is no exception with ESP32, since its vendor Espressif provides a development environment (ESP-IDF) and multiple libraries for interacting with hardware in C. It offers full support for ESP32 by the vendor via hardware abstraction libraries. It is also used in the popular Arduino framework extension for the ESP platform. In the context of embedded programming, C++ can be considered an extension of C (sometimes called C with classes) and is generally supported by vendor-provided tools (i.e., it is possible to use C++ on ESP32), but most available libraries and frameworks use only C. Most of the low-level C functionality is provided by its standard library. The C language is relatively low level, has manual memory management (for heap allocation), and is weakly typed. On ESP32, the C runtime environment includes FreeRTOS [19], which is embedded in ESP-IDF (and other development environments that use ESP-IDF, like Arduino [20]). While in principle it is possible to fully disable the real-time operating system (RTOS), it is not the standard use case (no support for this is offered by the vendor, or FreeRTOS itself), and was not considered for this paper.

Rust was originally developed by Graydon Hoare and later overtaken by Mozilla as a community-driven project [21]. Rust 1.0 was released in 2015 and since then has grown in popularity as a multipurpose programming language. A quite unusual feature of Rust is its fully automatic memory management without using garbage collection. All memory allocations are managed during compilation time, making many programming mistakes (which can easily be left in the C code) impossible [22]. Rust's ecosystem Cargo offers features such as building benchmarking and documentation generation [23]. Like C, Rust provides a large part of its functionality via the standard library (called crate in Rust). For ESP32 it is also possible to build Rust projects using the bare-metal environment, which does not use the standard library. However, this restricts the languages features (e.g., heap allocation or stack overflow protection is not supported, external custom libraries are needed) [24]. There is extensive Rust binding for ESP-IDF (esp-idf-sys crate) [25], which provides a high support for ESP32 features. However, support for ESP32 in Rust is in the so-called "Tier 3" [26], which does not guarantee that the project will build and work correctly.

TinyGo [27] is a recent version of the Go programming language (created at Google in 2007) that is oriented to embedded systems. Since Go uses an LLVM compiler like Rust, its ecosystem is also quite similar. The language is designed to be easy to parse and, by extension, easy to manipulate. Its garbage collector is predictable and easy to see for a programmer [28]. The TinyGo community lists such advantages over Rust as built-in support for concurrency without the need to rely on an RTOS-like framework, and architecturally better support for bare-metal applications [29]. As indicated in [30], TinyGo uses a cooperative scheduler and does not preempt tasks such as RTOS. Currently, support for ESP32 is not complete, and interfaces such as WiFi, Bluetooth, and even ADC are not available. Furthermore, not all standard library packages (Go library version) are supported [31]. Access to available peripherals is provided through the machine package.

MicroPython is an interpreter of Python 3 for microcontrollers and was developed by Damien George in 2014. Python is an interpreted language with garbage collector and is often perceived as a scripting language. It is widely used for application programming, especially by inexperienced programmers, due to its relative ease of use. The interpreter must be flashed into the supported microcontroller first. Some vendors now include the MicroPython interpreter as a default development environment in their products [32] as an easier alternative for beginners. MicroPython includes a subset of Python functions and libraries that are optimized for limited embedded environments [33]. MicroPython aims to be as compatible with normal Python as possible to allow an easy migration of desktop code to a microcontroller or embedded system. On ESP32, many peripherals are supported, including WiFi [34]. Since ESP32 does not usually contain a MicroPython interpreter, it can be downloaded and flashed following the instructions on the creator's website [35]. Python code can be uploaded and interpreted dynamically through a serial interface, or stored on microcontroller flash memory and run at boot time.

2.3. Algorithms Used for Performance Comparison

The ideal choice for performance comparison and evaluation would be to use an already existing comprehensive benchmark suite which includes a wide selection of different algorithms. However, few exist that are specifically targeted at embedded systems, and none exist that would consider relatively new languages like Rust or TinyGo. Existing embedded-oriented benchmarks include Bristol Energy Efficiency Benchmark Suite (BEEBS) benchmark, aimed at evaluating the energy consumption of embedded processors [36], MiBench [37], and EEMBC suite [38]. They all categorize used algorithms into different application categories, such as security, automotive, network, telecommunication, etc. These benchmarks test different types of embedded system applications in real-life use. A subset of algorithms was chosen from the benchmark suites mentioned above while considering these aspects:

- Presence in more than one embedded-oriented benchmark suite and more than two test categories. Algorithms that were already used in several benchmarks and grouped into different test categories were preferred, as they are known to be suitable for a more comprehensive performance evaluation.
- Presence in related works. Algorithms that were already used in similar performance comparisons on ESP32 were considered to be better tested and well suited for this work. Currently, the authors of [10] compare CRC-32 and SHA-256 in C and MicroPython on ESP32.
- Availability in vendor libraries. Algorithms that are implemented in Espressif (ESP32 vendor) officially provided libraries were assumed to be well tested and suited for ESP32, as well as faster to implement and port to other languages, due to their comprehensive documentation and use examples.
- Ease of use and verification. Since each selected algorithm had to be implemented in four different languages, it was crucial to be able to verify that each version outputs the correct results. Algorithms that can take a simple stream of data (such as an array)

and similarly output another stream of data or a single value were preferred. Then the input and expected output data could be easily generated and verified.

- Open source. The algorithm code should be available as an open source in any of the compared languages.

Five algorithms were chosen for comparison: popular hash functions CRC-32 and SHA-256, and three signal processing functions, FFT, IIR, and FIR. Many well-tested open source implementations of these functions can easily be found. The reasons for the selection of each algorithm are given in Table 2.

The CRC-32 or 32-bit Cyclic Redundancy Check is used in data integrity checks, while the SHA-256 (256-bit Secure Hash Algorithm) is used in authentication, encryption algorithms, and even cryptocurrencies. They take a byte stream for input. These functions were chosen to test how relatively simple operations perform on ESP32 while compiled in different languages, since their implementation involves simple bitwise shifting, logical, and arithmetic operations. They are used in two embedded-oriented benchmarks, the Bristol Energy Efficiency Benchmark Suite (BEEBS) and MiBench, and fit into the network, telecommunication, and security categories.

Another set of functions for this test were signal processing functions: Fast Fourier Transform (FFT), Infinite Impulse Response (IIR), and Finite Impulse Response (FIR) filters. EPS32 is powerful enough to be used for various signal processing tasks onboard; its vendor Espressif provides a comprehensive open source DSP library in Ansi C, and assembly and benchmark results for this library [39]. This enabled an easy comparison of other programming languages with vendor-provided code in C. To have more variety in data types, FFT with integer data points as inputs was selected, while IIR and FIR take float32. These algorithms are also used in the MiBench, BEEBS, and EEMBC benchmark suites and fit into the telecommunication, consumer, and automotive test categories.

Three of the selected algorithms can be further parameterized: CRC-32 by its polynomial, while FIR and IIR filters by their coefficients. For this work, no special considerations were made to select these parameters. Their values and amount (for FIR and IIR) were used as they appeared in the original source code or its usage examples. FIR was implemented as a 255th-order bandpass filter, while IIR was biquadratic type with five coefficients from their usage examples in the original source code. IEEE polynomial (0xEDB88320 in hexadecimal) was used for CRC-32.

The source codes for the selected functions were taken from free open sources in C (except for CRC-32, which was adapted from Go), then ported to other languages. Some functions were slightly adapted to make them stand-alone: library-wide error code definitions were removed from DSP functions; union type was removed from FFT code, as it has no close alternatives in other languages. The general structure of the code was kept as close as possible in all languages, while using some higher-level features of Rust, TinyGo, and MicroPython (e.g., using array length properties, instead of passing an additional length parameter like in C; using methods for structures). The details of each function are summarized in Table 2.

Table 2. The details of the functions used for the performance evaluation in selected programming languages.

Function	Source	Input Data Type (Passed as Array)	Comment	Reasons for Selection	Areas of Use
CRC32	[40]	uint8	IEEE polynomial	Presence in more than one benchmark (BEEBS, MiBench) Presence in related works Ease of use and verification	Network Telecommunication Security
SHA256	[41]	uint8	-	Presence in more than one benchmark (BEEBS, MiBench) Presence in related works	Network Telecommunication Security/Cryptography

FFT	[42]	int16	-	Ease of use and verification Presence in more than one benchmark (MiBench, EEMBC) Availability in vendor libraries Ease of use and verification	Telecommunication Consumer Automotive
FIR	[43]	float32	256 coefficients	Presence in more than one benchmark (EEMBC, BEEBS) Availability in vendor libraries Ease of use and verification	Telecommunication Consumer Automotive
IIR	[44]	float32	biquad type	Presence in more than one benchmark (EEMBC, BEEBS) Availability in vendor libraries Ease of use and verification	Telecommunication Consumer Automotive

MATLAB models for each function were also written. They were used to generate output data, which were transferred to the source code for each language, and used as a reference to verify the correct execution of the functions (more details in Section 2.3). The full source code with compilation and uploading instructions is provided in GitHub repository: https://github.com/ignasp/ProgLangComp_onESP32 (accessed on 27 December 2022).

2.4. Performance Comparison Methodology

While there are many benchmarking libraries available for each language (Rust toolchain even has a built-in benchmarking capability), they all greatly differ in implementation and use details. To provide a unified way to benchmark the selected functions, a simple custom benchmark library was first implemented in C and then ported to Rust, TinyGo, and Python.

To measure execution time, a timer structure (or object) and associated methods *start* and *stop* were defined, as detailed in pseudocode below:

```
STRUCTURE Timer : a
    tStart    : Time value
    tDuration : Time value

FUNCTION start(timer : TIMER) :
    timer.tStart = current time

FUNCTION stop(timer : TIMER) :
    timer.tDuration = calculated with timer.tStart as a ref-
    erence
```

In each language, functions were used that return a monotonically increasing clock. They were provided by either the standard library of the language or by the vendor of the ESP32 (Espressif) library. Table 3 lists the exact functions used for each language.

Table 3. Functions and code used to measure execution time in each language.

Language	Function for tStart	Function to Calculate Duration
C/C++	esp_timer_get_time()	esp_timer_get_time()—tStart
Rust	esp_idf_sys::esp_timer_get_time();	esp_idf_sys::esp_timer_get_time()—self.tStart
TinyGo	time.Now()	time.Since(tStart)
MicroPython	utime.ticks_us()	ticks_diff(ticks_us(), tStart)

Next, a test function type *RunFp* was defined:

```
TYPE RunFp : FUNCTION RezVerification(
                    data_len : Integer,
                    Timer : TimerObject/Struct)
```

Different versions were implemented for every function tested since they all differ in the types and size of input and output data, the necessary initializations, and the cleanup. The function takes two parameters—data length, to enable testing for different input length sizes, and a timer object, to measure the execution time. It also compares the generated output data with predefined expected values and returns a *RezVerification* enumeration type, which indicates whether the output data match the reference result data. Figure 1 presents the *RunFp* function algorithm.

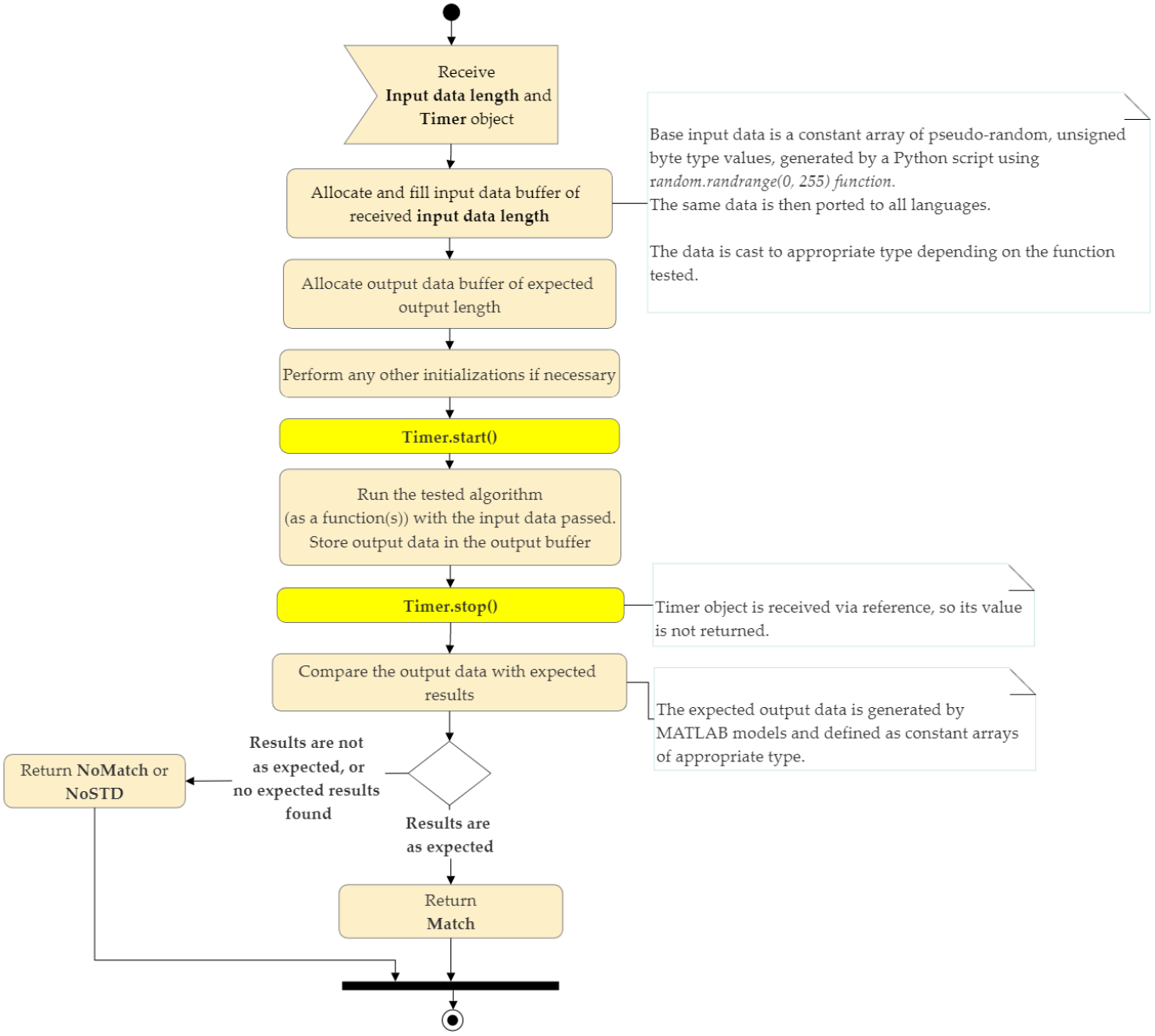

Figure 1. The algorithm of the execution time measurement function.

Finally, another structure and a *bench_Run* method/function were defined to automate execution of the testing function for different number of iterations and input data lengths:

```
STRUCTURE Tester :
    TestName       : String
    TestLengths    : Integer array
    Niterations    : Unsigned integer array
    Ptype      : Test result printing type (Readable, CSV)
    RunFn      : RunFp (Test runner function)
```

```
FUNCTION bench_Run (test : Tester) :
    INITIALIZE timer : Timer
    FOR data length IN test.TestLengths :
    FOR: iteration number IN test.Niterations :
         test result : RezVerification = run test.RunFn(data
    length, timer)
         PRINT(
              Language Name
              test.test name
              CPU frequency
              Iteration number
              timer.tDuration in microseconds
              test result)
```

The structure stores the test name for readability, an array of tested input data lengths, a number of iterations, a result printing type (which defines how the test results are printed to a serial interface), and a reference to the testing function type *RunFp*. The *bench_Run* method takes a *Tester* as a parameter and runs the referenced testing function for every defined data length and number of iterations, while printing the results of each iteration to the serial console, as detailed in Figure 2.

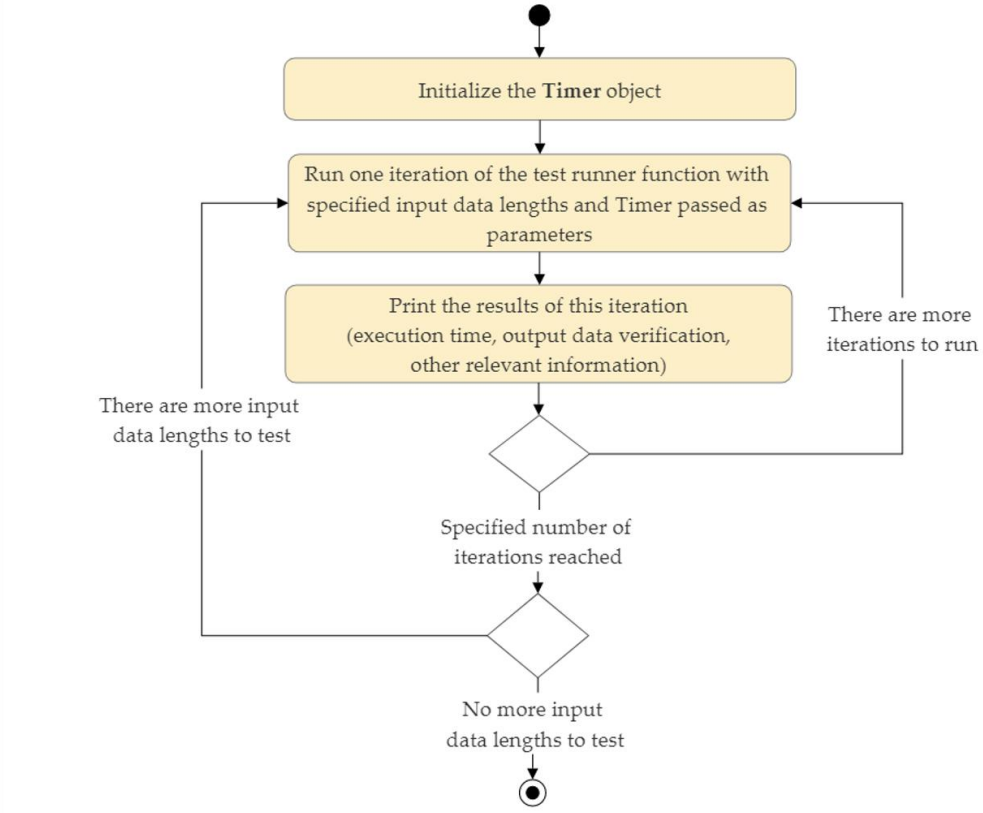

Figure 2. The general algorithm of the execution time measurements.

Two ways of results printing are available: readable for quick verification, and CSV. Figure 3 shows an example of readable results in a seral terminal emulator.

Lang	Test	Freq(Mhz)	Iter	Length	CPUcyc	Timeus	Rez
C	CRC32	160	1	32	154	730	OK
C	CRC32	160	2	32	154	742	OK
C	CRC32	160	3	32	154	730	OK

Figure 3. Test results for 3 iterations of CRC-32 in C programming language with data length of 32, displayed in a serial terminal.

The results printed in CSV format can be easily transferred to another program (such as Microsoft Excel) for data consolidation and further analysis. In C programming language, CPU cycles were also measured as a reference for DPS algorithms (FFT, IIR, and FIR), which have their benchmark result in CPU cycles provided by the vendor Espressif.

The tester structure is defined in the main body of the program, and the *bench_Run* method is called for every algorithm tested. For this study, all tests were run for 100 iterations, with input data lengths of 0, 16, 32, 64, 128, 256, 512, and 1024. Since the code on ESP32 is executed on top of FreeRTOS (except for TinyGo), it is expected to see some variation in execution times of different iterations due to its preemptive scheduling (SysTick). No RTOS specific functions were used in the code.

It is also important to note that the tests were performed using a 160 MHz CPU frequency. While the ESP32 can work with up to 240 MHz, the default value after boot is 160 MHz, and it currently not possible to set the custom frequency in all languages (as their standard libraries do not have functions for that and 160 MHz is hardcoded), so the default 160 MHz was used, and verified by reading and printing the CPU frequency by available functions (provided in all languages).

During development, it was discovered that TinyGo fails to link the full code, with all functions and reference output data included. To solve this, each function was first compiled and executed separately, with full reference output data included (the problem was traced to the linker script, where it is indicated that constant global variables are loaded in DRAM, and not flash memory, and this should be fixed eventually [45]). After being convinced that all functions return the correct result, the final tests were executed with 32 reference output data points included in the code. Since the verification of the output data is outside of any time measurements, it is not expected to affect the results in any way.

A similar problem arose with MicroPython (the interpreter failed to load the full code), so the same solution was introduced.

Initially, different optimization levels were attempted for the compiled languages. TinyGo, compiled with levels other than -Oz (highest optimization for size), produced incorrect results (verification failed; it was asserted that this was due to functions producing results different from reference data), or caused program panic. Since a full comparison for other optimization levels could not be made, the tests were performed with -Os for C and -Oz for Rust and TinyGo.

2.5. Hardware Setup

ESP32 is a family of powerful microcontrollers, based on the Xtensa 32 bit architecture and manufactured by Espressif [46]. It has integrated WiFi Wi-Fi 802.11 b/g/n, dual-mode Bluetooth version 4.2, and a variety of peripherals. ESP32 has a dual core processor with a frequency of up to 240 MHz, 520 Kilobytes of SRAM, and 16 Megabytes of flash program memory. ESP32 is supported by various popular integrated development environments (IDE), such as Arduino (for C/C++) and PlatformIO (for various languages through extensions).

Due to its relatively low price, the ESP32 is used in numerous prototyping and development boards [47], aimed both at professionals and enthusiasts. For this test, an M5 Stack Basic development kit with ESP32-D0WDQ6-V3 (Figure 4) was used.

(**a**) (**b**)

Figure 4. (**a**) M5 Stack Basic kit used for tests; (**b**) the kit opened, showing the ESP32.

No special preparation is needed to use the kit. It is simply connected to the computer via USB and discovered as a serial device, on which programs can then be deployed by any of the available IDEs, or simply by using a vendor-provided tool (which is also used by the aforementioned IDEs) [48]. For this test, no internal or external peripherals were used.

2.6. Software Development Environments and Compilers

For code development, integrated development environments (IDE) were used. Visual Studio Code (VS Code) [49] was used for C, Rust, and TinyGo, and Thonny [50] was used for MicroPython.

Thonny is an open source Python IDE, which also allows using MicroPython via serial port. It was used on a Windows 10 machine.

VS Code is an open source multiplatform IDE developed by Microsoft. Its features can be highly customized by installing extensions, which are available for a wide variety of languages and scripts, including C/C++, Rust, and TinyGo.

VS Code with the PlatformIO extension [51] was used on a Windows 10 machine for the development of C/C++ code. PlatformIO allows development for various embedded platforms, including ESP32. The C/C++ extension was automatically installed as a dependency. A new project was created for the M5stack core, with the Arduino framework. All other configurations were handled by the extension.

For Rust, VS Code with the rust-analyzer extension [52] was used on a Debian 11 machine. Debian was chosen over Windows as it appeared to be easier to install the necessary toolchain on a Linux machine. The rust-analyzer extension only provides syntax highlight and checking; the toolchain needed to compile Rust for ESP32 was installed following instructions provided in [24]. With the development environment ready, a Rust project was created using template [53].

Finally, for TinyGo, VS Code with the TinyGo extension [54] was used on the same Debian 11 machine. The TinyGo extension automatically installs the Go extension as a dependency. As with Rust, the actual toolchain was installed separately, following [55].

Table 4 lists the specific versions of the IDE, extension, and toolchain versions installed for this study.

Table 4. Overview of the development environment used for each programming language.

Language	IDE	Relevant Toolchain Versions
C/C++	VS Code with plugins: PlatformIO v2.5.5 and Espressif 32 v5.2.0 C/C++ v1.12.4	no additional version output available, toolchain is fully managed by Espressif platform via PlatformIO

	Arduino framework	
Rust	VS Code with plugins: rust-analyzer v0.3.1285	Xtensa toolchain: esp-2021r2-patch5-8_4_0 cargo 1.65.0-nightly (4bc8f24d3 20 October 2022) rustc 1.65.0-nightly (5b08d0476 4 November 2022) esp-idf-sys 0.31.9
TinyGo	VS Code with plugins: TinyGo v0.4.0 Go v0.36.0	Tinygo—version 0.25.0 linux/amd64 (using go version go1.19.1 and LLVM version 14.0.0)
MicroPython	Thonny v4.0.1	micropython 3.4.0; MicroPython v1.19.1 on 18 June 2022

3. Results

This section presents the results of the comparison of different data and signal processing algorithms, including CRC-32, SHA-256, FFT, FIR, and IIR. The algorithms were implemented using four programming languages, namely, C, Rust, TinyGo, and MicroPython. The results of this comparison are presented in the diagrams and table below. Each algorithm has a corresponding graph that shows the average execution time for each programming language using a logarithmic scale. In addition, Table 5 shows the execution times together with the standard deviations. For this study, all tests were run for 100 iterations, with input data lengths of 0, 16, 32, 64, 128, 256, 512, and 1024 bytes. Therefore, there are eight measurements in each figure.

Figure 5 presents a comparison of the average execution times of the CRC-32 algorithms. Since MicroPython average execution times are several orders of magnitude higher than times of the other programming languages, the results are presented using the logarithmic scale. As we can see in Figure 5, in most cases, the TinyGo implementation was the fastest, except for data sizes of 0, 16, and 32, where the C-based algorithm was slightly faster or equal. In all cases, the Rust implementation showed the third result, while the MicroPython program showed the worst execution times.

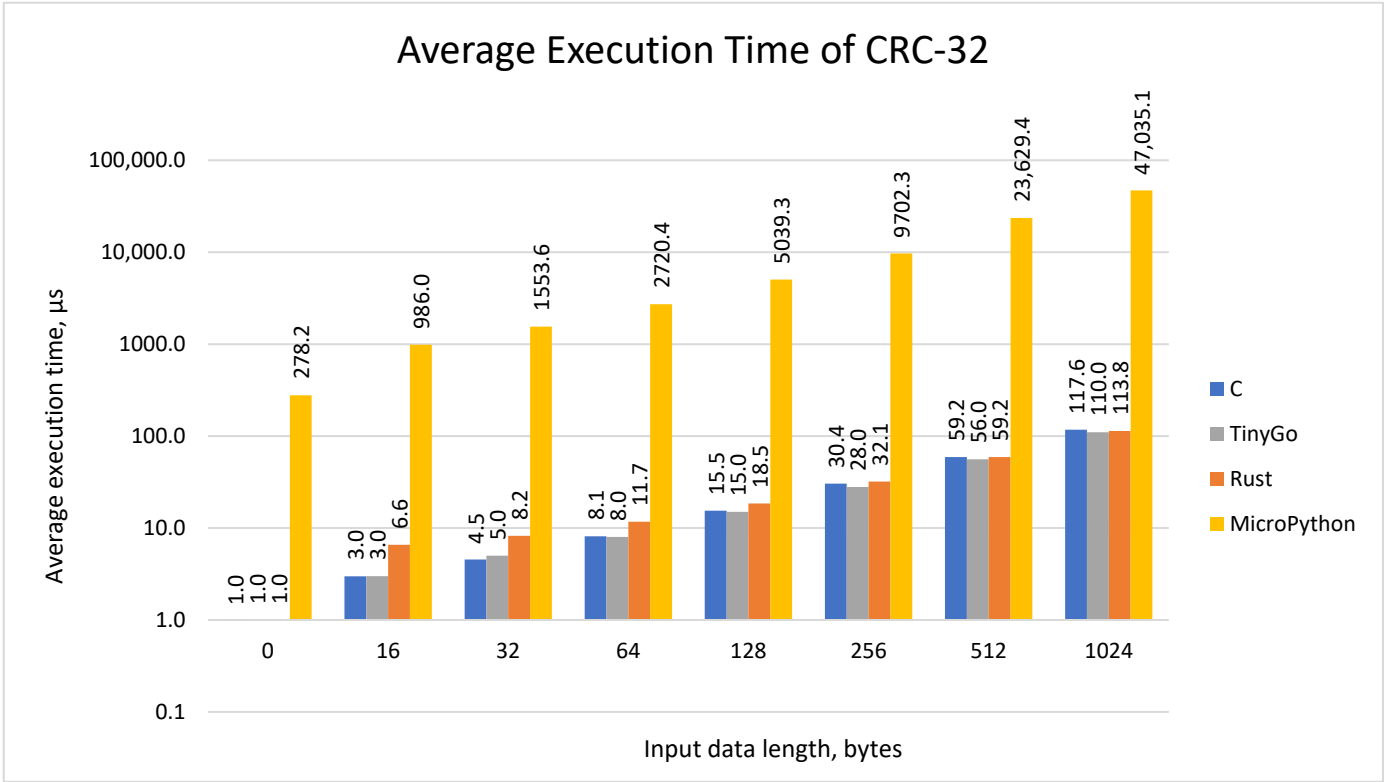

Figure 5. Average execution times of the CRC-32 algorithm (please note the logarithmic time scale).

Figure 6 presents a comparison of the average execution times of the SHA-256 algorithms. As we can see in Figure 6, in all cases, the algorithm implemented in C language was the fastest, followed by the TinyGo and Rust programs. The TinyGo algorithm showed the second result in most cases, except for data lengths of 512 and 1024, where the Rust implementation was slightly faster. The MicroPython program again showed the worst execution times.

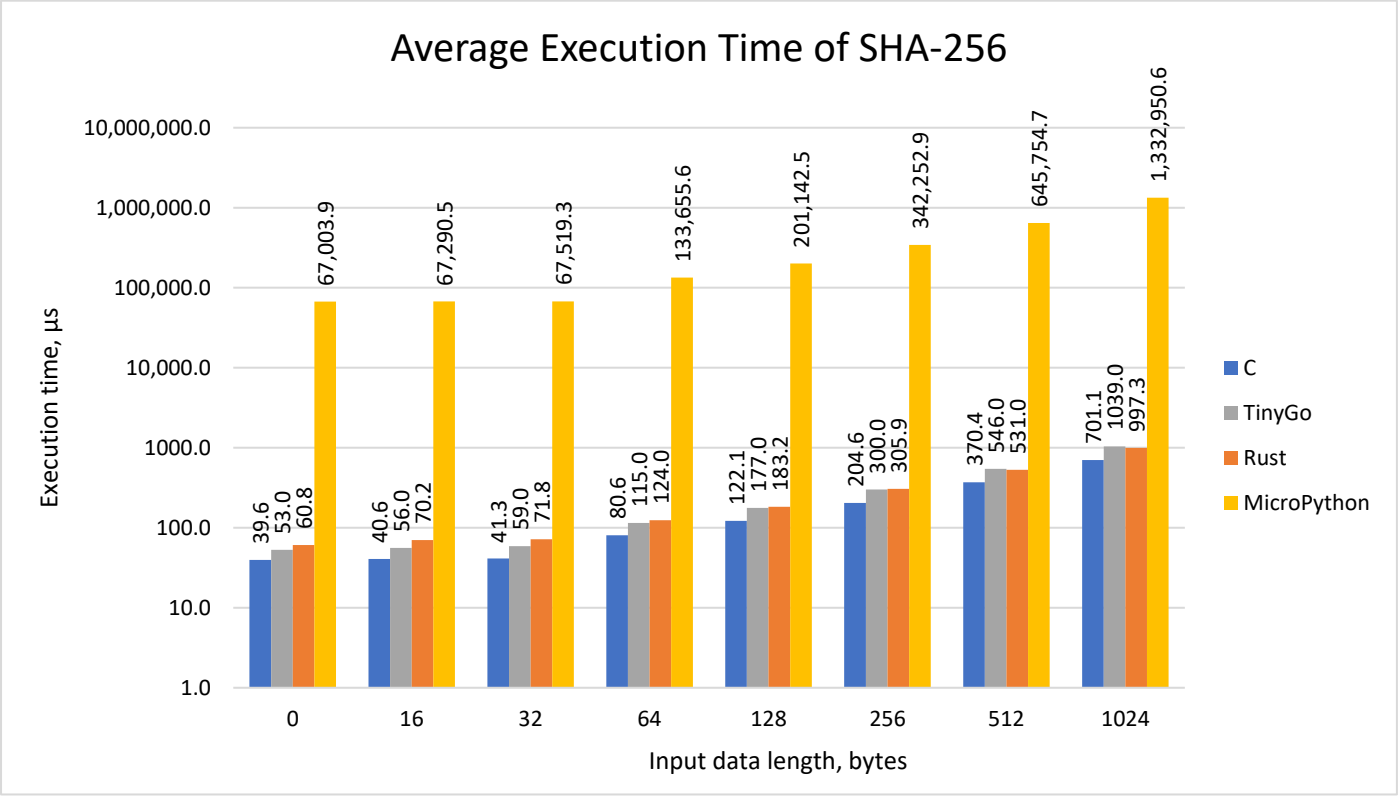

Figure 6. Average execution times of the SHA-256 algorithm (please note the logarithmic time scale).

Figure 7 presents a comparison of the average execution times of the FFT algorithms. As we can see in Figure 7, in all cases, the algorithm implemented in the C language was the fastest, followed by the TinyGo and Rust programs, except for the function calls with zero data, which resulted in equal average times. The MicroPython program showed the worst execution times, as expected.

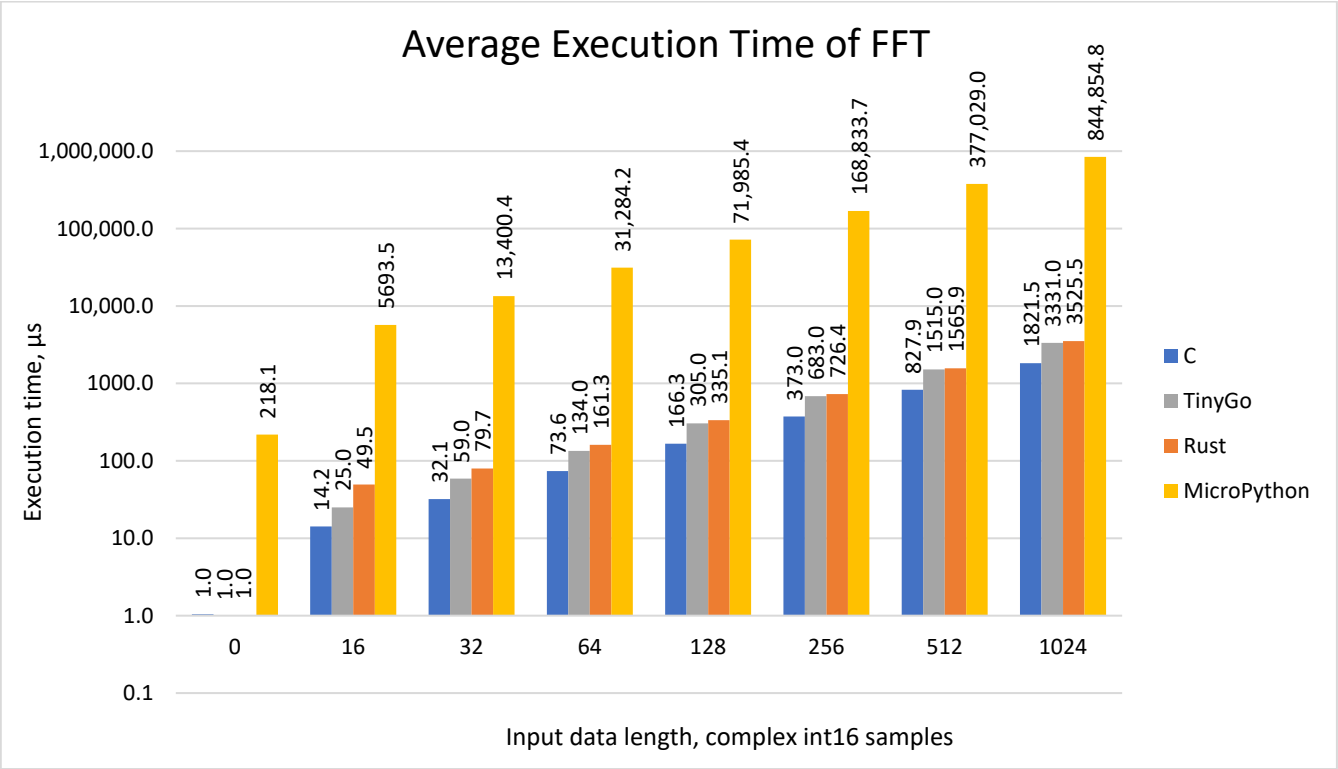

Figure 7. Average execution times of the FFT algorithm (please note the logarithmic time scale).

Figure 8 presents a comparison of the average execution times of the FIR filter algorithms. It is quite interesting that in this case, the algorithm implemented in the Rust language was the fastest, followed by the TinyGo and C programs, except for the function calls with zero data and data length of 16, where TinyGo showed a slightly better result. No surprise, the MicroPython algorithm was slowest again.

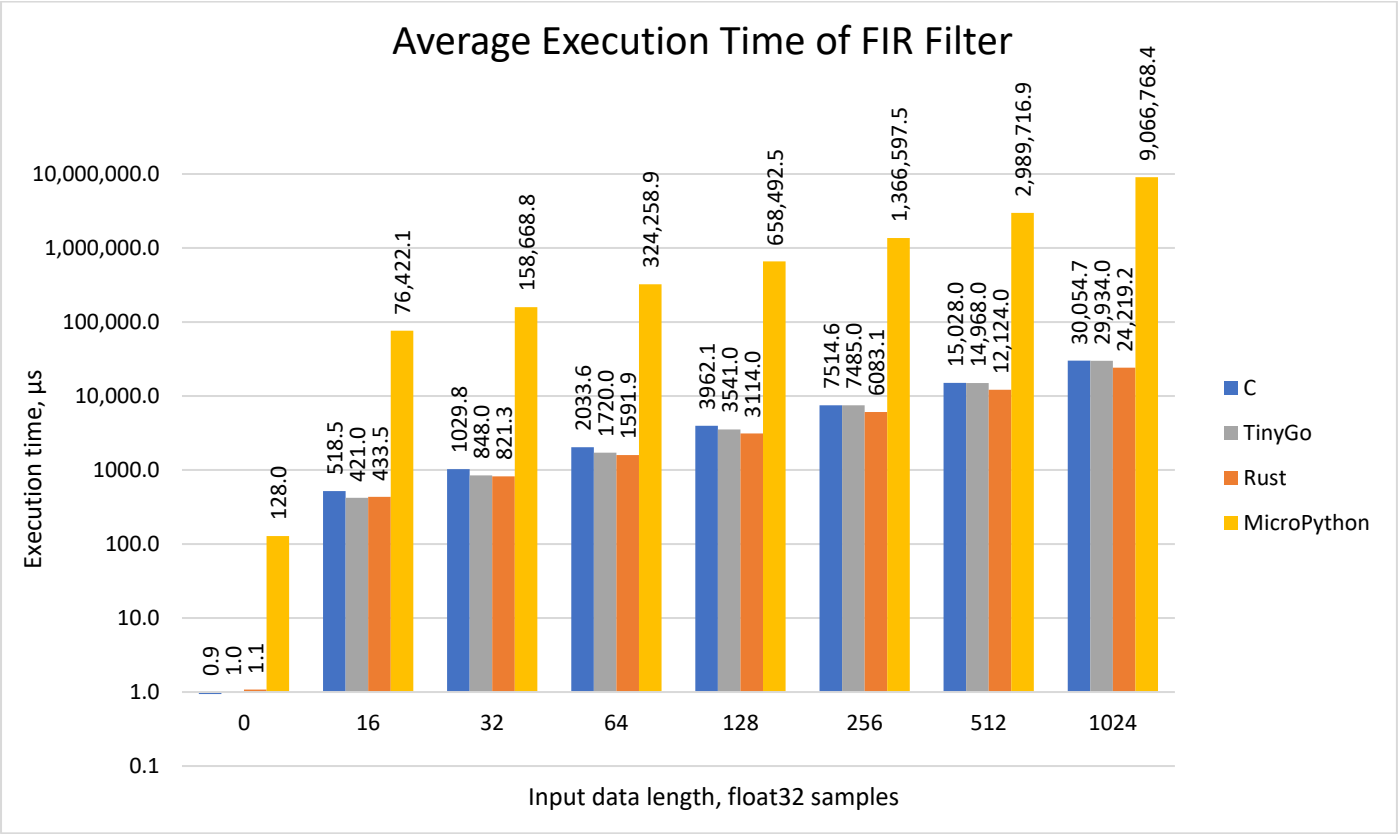

Figure 8. Average execution times of the FIR filter algorithm (please note the logarithmic time scale).

Figure 9 presents a comparison of the average execution times of the IIR filter algorithms. As we can see in Figure 9, in all cases, the algorithm implemented in the C language was the fastest, followed by the TinyGo and Rust programs, except for the data length of 32 samples, where the TinyGo program was only 0.1 µs faster on average. The MicroPython program showed the worst execution times again.

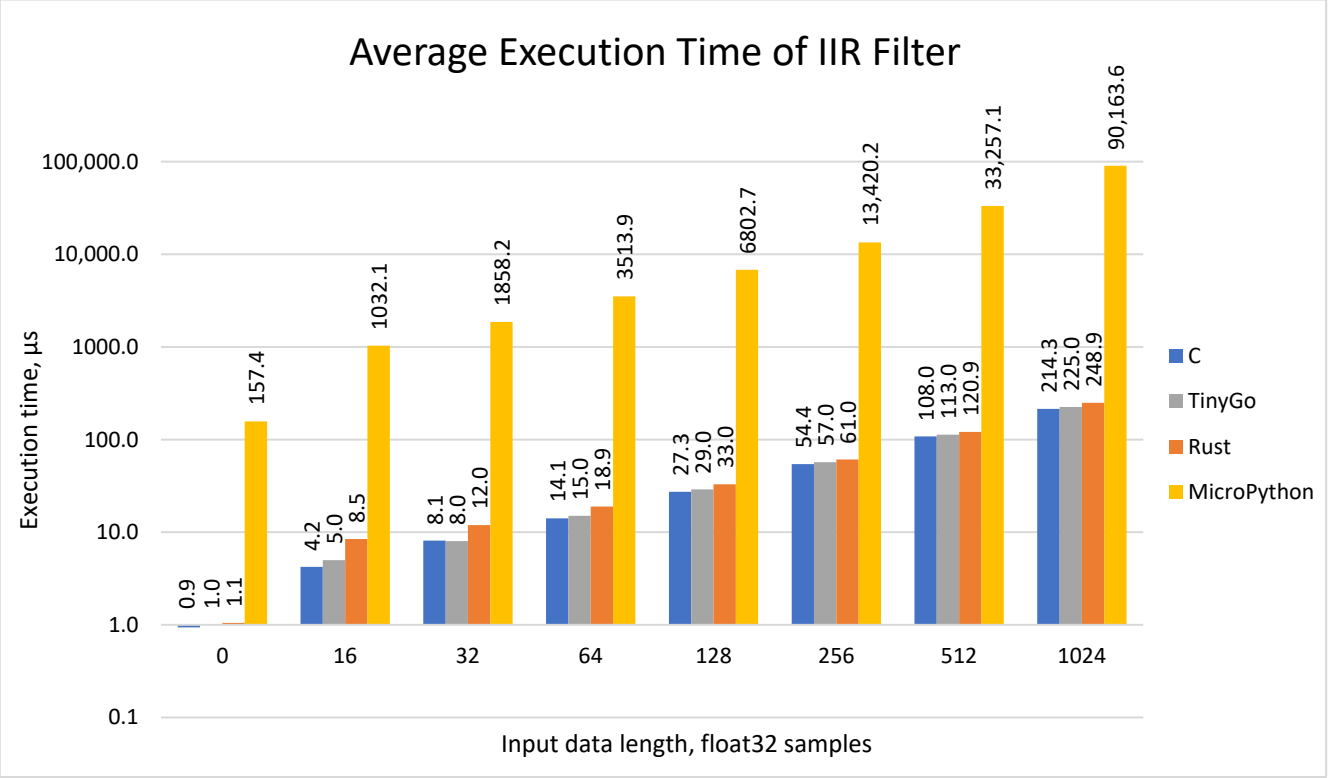

Figure 9. Average execution times of the IIR filter algorithm (please note the logarithmic time scale).

Table 5. Average execution times and standard deviations of the algorithms compared (Time AVG—average execution time, SD—standard deviation).

Algorithm	Input Data Length	C		TinyGo		Rust		MicroPython	
		Time AVG, µs	SD, µs	Time AVG, µs	SD, µs	Time AVG, µs	SD, µs	Time AVG, µs	SD, µs
CRC-32	0	1.0	0.1	1.0	0.0	1.0	0.0	278.2	15.2
	16	3.0	1.6	3.0	0.0	6.6	0.6	986.0	12.7
	32	4.5	0.5	5.0	0.0	8.2	0.6	1553.6	10.9
	64	8.1	0.3	8.0	0.0	11.7	0.6	2720.4	11.4
	128	15.5	1.2	15.0	0.0	18.5	0.6	5039.3	12.4
	256	30.4	2.7	28.0	0.0	32.1	0.5	9702.3	10.4
	512	59.2	2.7	56.0	0.0	59.2	0.6	23,629.4	11.3
	1024	117.6	3.8	110.0	0.0	113.8	0.9	47,035.1	12.7
SHA-256	0	39.6	2.7	53.0	0.0	60.8	2.2	67,003.9	42.1
	16	40.6	2.8	56.0	0.0	70.2	2.4	67,290.5	6.8
	32	41.3	2.0	59.0	0.0	71.8	2.7	67,519.3	6.7
	64	80.6	3.0	115.0	0.0	124.0	2.8	133,655.6	28.1
	128	122.1	3.8	177.0	0.0	183.2	2.7	201,142.5	40.0
	256	204.6	4.5	300.0	0.0	305.9	2.9	342,252.9	87.6
	512	370.4	5.6	546.0	0.0	531.0	2.9	645,754.7	363.3
	1024	701.1	5.4	1039.0	0.0	997.3	4.5	1,332,950.6	990.8
FFT	0	1.0	0.2	1.0	0.0	1.0	0.1	218.1	11.0
	16	14.2	0.4	25.0	0.0	49.5	2.5	5693.5	7.6
	32	32.1	0.3	59.0	0.0	79.7	2.1	13,400.4	6.9
	64	73.6	2.7	134.0	0.0	161.3	2.2	31,284.2	4.7
	128	166.3	4.0	305.0	0.0	335.1	2.8	71,985.4	6.0
	256	373.0	5.5	683.0	0.0	726.4	6.0	168,833.7	4.2

	512	827.9	4.4	1515.0	0.0	1565.9	3.7	377,029.0	8.9
	1024	1821.5	4.3	3331.0	0.0	3525.5	21.0	844,854.8	15.0
FIR filter	0	0.9	0.2	1.0	0.0	1.1	0.3	128.0	5.0
	16	518.5	5.7	421.0	0.0	433.5	6.0	76,422.1	229.9
	32	1029.8	0.4	848.0	0.0	821.3	5.5	158,668.8	151.3
	64	2033.6	2.6	1720.0	0.0	1591.9	6.1	324,258.9	15.7
	128	3962.1	1.1	3541.0	0.0	3114.0	6.3	658,492.5	2062.2
	256	7514.6	5.5	7485.0	0.0	6083.1	6.4	1,366,597.5	33.1
	512	15,028.0	1.6	14,968.0	0.0	12,124.0	7.8	2,989,716.9	152.7
	1024	30,054.7	2.5	29,934.0	0.0	24,219.2	12.8	9,066,768.4	4479.6
IIR filter	0	0.9	0.2	1.0	0.0	1.1	0.2	157.4	5.6
	16	4.2	0.4	5.0	0.0	8.5	0.6	1032.1	4.7
	32	8.1	2.5	8.0	0.0	12.0	0.4	1858.2	5.9
	64	14.1	0.3	15.0	0.0	18.9	0.4	3513.9	7.2
	128	27.3	0.5	29.0	0.0	33.0	0.4	6802.7	6.1
	256	54.4	2.8	57.0	0.0	61.0	0.9	13,420.2	7.8
	512	108.0	3.9	113.0	0.0	120.9	1.1	33,257.1	15.6
	1024	214.3	4.4	225.0	0.0	248.9	4.3	90,163.6	28.7

Table 5 shows the average execution times for different algorithm implementations, as well as the standard deviations for each selected data length. Note that TinyGo deviations are equal to zero in all cases, which can be explained by the fact that TinyGo programs do not use the operating system and are deployed directly on the hardware. Therefore, TinyGo-written programs always have the same execution time on the ESP32 platform, which is a very valuable feature from the point of view of the real-time systems developer. The MicroPython-based algorithms showed the worst execution performance, which is logical, since this language is not compiled, but interpreted, resulting in very high computational overhead. On the other hand, MicroPython (like any Python version) is a higher-level language than the other evaluated languages. Therefore, theoretically, it allows faster and easier code development, resulting in only a few lines of code. This is true for high-level system development, but it is not always the case in embedded programming, where a code developer usually needs to create an algorithm himself according to some formula, like in our study.

4. Discussion

The data presented in the Results section strongly correlate with some previous work [10,11], showing that MicroPython-based programs currently have much worse performance on the ESP32 platform, compared to programs written in the C programming language. On the other hand, this study allowed us, for the first time, to evaluate execution performance of two additional languages, namely, Rust and TinyGo. Both are quite popular among system developers and were created as an alternative to the C programming language, which until now has been considered a gold standard for embedded and IoT system development. Therefore, it was interesting for us to find out how good these alternatives are.

The results show that, surprisingly, the C-based algorithms, although fastest in most cases, in some cases were not the best. The C-based programs were outperformed by TinyGo in several cases:

- CRC-32 implementations with data sizes of 64, 128, 256, 512, 1024;
- FIR filter implementations with all data sizes, except 0 (just a function/method call with zero data);
- IIR filter implementations with data size 32.

The C-based FIR algorithm was outperformed by Rust-based FIR implementation as well, with all data sizes except 0. In this case, the Rust-based FIR algorithm was the fastest, followed by TinyGo, C, and finally, MicroPython with all data lengths except 16, where TinyGo was slightly superior. In other cases, the Rust programs took the third place, except for SHA-256 with data sizes of 512 and 1024, outperforming the TinyGo algorithm by a few microseconds on average.

Summarizing the results, it can be concluded that in most cases C algorithms had the best execution times, followed by TinyGo, Rust, and MicroPython. The clear outsider in this case was MicroPython, whose execution times were thousands of times worse than implementations in other languages.

However, the difference between C and TinyGo programs in many cases was only a few microseconds, which is not a very big difference for most embedded applications and IoT systems development. In addition, TinyGo-based algorithms have a very important advantage over C and other programming languages, since they always have the same execution time, i.e., their standard deviation of all execution times is zero. This is explained by the fact that TinyGo programs are deployed directly to the hardware without any operating system; therefore, nothing interferes with the execution process. This means that currently TinyGo technology is the best choice for the implementation of hard real-time systems on the ESP32 platform, where time jitter is a problem. However, it is unclear whether this feature will not be lost in the future, since Go (on which TinyGo is based) now uses asynchronously preemptible routines (as of version 1.14) [56]. These routines would eliminate the jitter-free execution advantage if they were introduced in the ESP32 TinyGo implementation as well. Besides, TinyGo is still in early stages of development [57], while Rust now fully supports its standard library on ESP32 and is more mature.

Finally, the MicroPython language, which is gaining more popularity, is not the best choice for low-level high-performance programming, since its execution times are incomparably longer than C, TinyGo, or even Rust. Therefore, the MicroPython language can be recommended for general-purpose high-level system programming, especially for teaching purposes and student projects, because it allows for faster and easier system development.

4.1. Limitations

The main limitation of this performance evaluation is that it does not include any ESP32 hardware-specific tests (such as using any peripherals). A comprehensive benchmark for a specific embedded system would be expected to evaluate the performance of accessing and using hardware peripherals as well. Nevertheless, we believe that this work provides a fair comparison of the current versions of the languages and produces results that will be relevant for a longer time. MicroPython, TinyGo, and Rust are still relatively new and developing languages, suitable for the ESP32 platform. While the general non-platform specific features are not expected to significantly change in the future, the same cannot be said about the hardware libraries.

4.2. Threats to Validity

The main threat to the validity of this work is within the selection of the algorithms to test. As discussed in Section 2.2, compiling a comprehensive benchmark suite for any platform is a non-trivial task, more so for four different programming languages. Personal bias and insufficient analysis cannot be excluded. However, we are confident that the selected algorithms provide a sufficiently comprehensive (as detailed in Table 2 “Areas of use” column), and most importantly, novel insight into performance of the compared languages on ESP32.

5. Conclusions

In this paper, we have presented the evaluation of the execution performance of the C/C++, MicroPython, Rust, and TinyGo programming languages on the ESP32 microcontroller. For this purpose, five widely used embedded processing algorithms were utilized: FFT, CRC-32, SHA-256, IIR filter, and FIR filter. This study is the first attempt to evaluate the execution performance of the newly emerging Rust and TinyGo programming languages. The aim of this evaluation is to find out how good these user-friendly languages are compared to the C/C++ language, which is a gold standard for embedded applications.

The results of this study reveal that, though the C/C++ programming language is widely believed to be the most efficient for embedded programming, that is not always the case. Our experiments showed that in a few cases the C/C++ algorithms were outperformed by algorithms implemented in TinyGo and Rust. Even in those cases where C/C++ implementations were faster, the difference between its execution times and that of other languages was not very significant. Moreover, the TinyGo algorithms demonstrated jitter-free execution, making this language more preferable for hard real-time applications. Therefore, TinyGo and Rust can be recommended as efficient higher-level ESP32 programming languages, which are characterized by faster and simpler programming compared to the C/C++ language.

This work may be helpful for embedded software developers, researchers, and students who use the ESP32 platform for various application development and study processes and need to select the most suitable programming language which is currently available on this platform.

Author Contributions: Conceptualization, I.P., A.L. and A.J.; Formal analysis, A.J.; Funding acquisition, A.L. and A.J.; Investigation, I.P.; Methodology, A.L. and A.J.; Software, I.P.; Supervision, A.L.; Validation, I.P. and A.L.; Visualization, A.J.; Writing—original draft, A.L. and A.J.; Writing—review and editing, I.P., A.L. and A.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Maier, A.; Sharp, A.; Vagapov, Y. Comparative Analysis and Practical Implementation of the ESP32 Microcontroller Module for the Internet of Things. In *Proceedings of the 2017 Internet Technologies and Applications (ITA), Wrexham, UK, 12–15 September 2017*; IEEE: Piscataway, NJ, USA, 2017; pp. 143–148.
2. Aghenta, L.O.; Tariq Iqbal, M. Design and Implementation of a Low-Cost, Open Source IoT-Based SCADA System Using ESP32 with OLED, ThingsBoard and MQTT Protocol. *AIMS Electron. Electr. Eng.* **2020**, *4*, 57–86. https://doi.org/10.3934/ElectrEng.2020.1.57.
3. Aghenta, L.O.; Iqbal, M.T. Low-Cost, Open Source IoT-Based SCADA System Design Using Thinger.IO and ESP32 Thing. *Electronics* **2019**, *8*, 822. https://doi.org/10.3390/electronics8080822.
4. Allafi, I.; Iqbal, T. Design and Implementation of a Low Cost Web Server Using ESP32 for Real-Time Photovoltaic System Monitoring. In *Proceedings of the 2017 IEEE Electrical Power and Energy Conference (EPEC), Saskatoon, SK, Canada, 22–25 October 2017*; IEEE: Piscataway, NJ, USA, 2017; pp. 1–5.
5. Carducci, C.G.C.; Monti, A.; Schraven, M.H.; Schumacher, M.; Mueller, D. Enabling ESP32-Based IoT Applications in Building Automation Systems. In *Proceedings of the 2019 II Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0&IoT), Naples, Italy, 4–6 June 2019*; IEEE: Piscataway, NJ, USA, 2019; pp. 306–311.
6. Sangeethalakshmi, K.; Preethi Angel, S.; Preethi, U.; Pavithra, S.; Shanmuga Priya, V. Patient Health Monitoring System Using IoT. *Mater. Today Proc.* **2021**, *In press, corrected proof* S2214785321045545. https://doi.org/10.1016/j.matpr.2021.06.188.
7. Taştan, M.; Gökozan, H. Real-Time Monitoring of Indoor Air Quality with Internet of Things-Based E-Nose. *Appl. Sci.* **2019**, *9*, 3435. https://doi.org/10.3390/app9163435.
8. Bipasha Biswas, S.; Tariq Iqbal, M. Solar Water Pumping System Control Using a Low Cost ESP32 Microcontroller. In *Proceedings of the 2018 IEEE Canadian Conference on Electrical & Computer Engineering (CCECE), Quebec, QC, Canada, 13–16 May 2018*; IEEE: Piscataway, NJ, USA, 2018; pp. 1–5.
9. Hangan, A.; Chiru, C.-G.; Arsene, D.; Czako, Z.; Lisman, D.F.; Mocanu, M.; Pahontu, B.; Predescu, A.; Sebestyen, G. Advanced Techniques for Monitoring and Management of Urban Water Infrastructures—An Overview. *Water* **2022**, *14*, 2174. https://doi.org/10.3390/w14142174.

10. Ionescu, V.M.; Enescu, F.M. Investigating the Performance of MicroPython and C on ESP32 and STM32 Microcontrollers. In *Proceedings of the 26th International Symposium for Design and Technology in Electronic Packaging (SIITME), Pitesti, Romania, 21 October 2020*; IEEE: Piscataway, NJ, USA, 2020; pp. 234–237.
11. Dokic, K.; Radisic, B.; Cobovic, M. MicroPython or Arduino C for ESP32-Efficiency for Neural Network Edge Devices. In *Intelligent Computing Systems*; Brito-Loeza, C., Espinosa-Romero, A., Martin-Gonzalez, A., Safi, A., Eds.; Communications in Computer and Information Science; Springer International Publishing: Cham, Switzerland, 2020; Volume 1187, pp. 33–43, ISBN 978-3-030-43363-5.
12. Grunert, K. Overview of JavaScript Engines for Resource-Constrained Microcontrollers. In Proceedings of the 5th International Conference on Smart and Sustainable Technologies (SpliTech), Split, Croatia, 23 September 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–7.
13. Suarez-Albela, M.; Fernandez-Carames, T.M.; Fraga-Lamas, P.; Castedo, L. A Practical Performance Comparison of ECC and RSA for Resource-Constrained IoT Devices. In *Proceedings of the Global Internet of Things Summit (GIoTS), Bilbao, Spain, 4–7 June 2018*; IEEE: Piscataway, NJ, USA, 2018; pp. 1–6.
14. Kodithuwakku, J.; Arachchi, D.D.; Thiha, S.; Rajasekera, J. Two Optimized IoT Device Architectures Based on Fast Fourier Transform to Monitor Patient's Photoplethysmography and Body Temperature. *Comput. Sci. Math. Forum* **2022**, *2*, 7.
15. Fabregat, G.; Belloch, J.A.; Badia, J.M.; Cobos, M. Design and Implementation of Acoustic Source Localization on a Low-Cost IoT Edge Platform. *IEEE Trans. Circuits Syst. II* **2020**, *67*, 3547–3551. https://doi.org/10.1109/TCSII.2020.2986296.
16. Shinde, A.R.; Mundada, K. Bike Engine Health Monitoring Using Vibration. In *Proceedings of the 2020 IEEE Pune Section International Conference (PuneCon), Pune, India, 16 December 2020*; IEEE: Piscataway, NJ, USA, 2020; pp. 99–102.
17. Aho, A.V.; Aho, A.V. (Eds.). *Compilers: Principles, Techniques, and Tools*, 2nd ed.; Pearson/Addison Wesley: Boston, MA, USA, 2007; ISBN 978-0-321-48681-3.
18. Kernighan, B.W.; Ritchie, D.M. *The C Programming Language*, 2nd ed.; Prentice Hall: Englewood Cliffs, NJ, USA, 1988; ISBN 978-0-13-308621-8.
19. FreeRTOS. Available online: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html (accessed on 15 November 2022).
20. Arduino Core for the ESP32, ESP32-S2, ESP32-S3 and ESP32-C3. Available online: https://github.com/espressif/arduino-esp32 (accessed on 15 November 2022).
21. Klabnik, S.; Nichols, C. *The Rust Programming Language*; No Starch Press: San Francisco, CA, USA, 2018; ISBN 978-1-59327-828-1.
22. Bugden, W.; Alahmar, A. Rust: The Programming Language for Safety and Performance. *arXiv* **2022**, arXiv:2206.05503. https://doi.org/10.48550/ARXIV.2206.05503.
23. The Cargo Book. Available online: https://doc.rust-lang.org/cargo/index.html (accessed on 15 November 2022).
24. The Rust on ESP Book. Available online: https://esp-rs.github.io/book/introduction.html (accessed on 15 November 2022).
25. Rust Bindings for ESP-IDF. Available online: https://github.com/esp-rs/esp-idf-sys (accessed on 15 November 2022).
26. Platform Support. Rustc. Available online: https://www.rust-lang.org/ (accessed on 15 November 2022).
27. TinyGo. Available online: https://tinygo.org/ (accessed on 15 November 2022).
28. Meyerson, J. The Go Programming Language. *IEEE Softw.* **2014**, *31*, 104. https://doi.org/10.1109/MS.2014.127.
29. Why Go Instead of Rust? Available online: https://tinygo.org/docs/concepts/faq/why-go-instead-of-rust/ (accessed on 15 November 2022).
30. TinyGo Runtime Scheduler.Go. Available online: https://github.com/tinygo-org/tinygo/blob/release/src/runtime/scheduler.go (accessed on 15 November 2022).
31. Packages Supported by TinyGo. Available online: https://tinygo.org/docs/reference/lang-support/stdlib/ (accessed on 15 November 2022).
32. Adi, P.D.P.; Kitagawa, A. A Review of the Blockly Programming on M5Stack Board and MQTT Based for Programming Education. In Proceedings of the IEEE 11th International Conference on Engineering Education (ICEED), Kanazawa, Japan, 6–7 November 2019; pp. 102–107.
33. George, D.P.; Sokolovsky, P. MicroPython Documentation. Release 1.10. Available online: https://docs.micropython.org/en/v1.10/micropython-docs.pdf (accessed on 15 November 2022).
34. Quick Reference for the ESP32. Available online: https://docs.micropython.org/en/latest/esp32/quickref.html (accessed on 15 November 2022).
35. MicroPython-Python for Microcontrollers ESP32. Available online: http://www.micropython.org/download/esp32/ (accessed on 15 November 2022).
36. Pallister, J.; Hollis, S.; Bennett, J. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. *arXiv* **2013**, arXiv:1308.5174.
37. Guthaus, M.R.; Ringenberg, J.S.; Ernst, D.; Austin, T.M.; Mudge, T.; Brown, R.B. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization, WWC-4 (Cat. No.01EX538), Austin, TX, USA, 2 December 2001; pp. 3–14.
38. Poovey, J.A.; Conte, T.M.; Levy, M.; Gal-On, S. A Benchmark Characterization of the EEMBC Benchmark Suite. *IEEE Micro* **2009**, *29*, 18–29. https://doi.org/10.1109/MM.2009.74.
39. Espressif DSP Library. Available online: https://docs.espressif.com/projects/esp-dsp/en/latest/esp-dsp-library.html (accessed on 15 November 2022).

40. The Go Authors. CRC32 Algorithm Implementation. Crc32_generic.Go. Available online: https://cs.opensource.google/go/go/+/refs/tags/go1.19.3:src/hash/crc32/crc32_generic.go (accessed on 15 November 2022).
41. Conte, B. Sha256.c. Available online: https://github.com/B-Con/crypto-algorithms/blob/master/sha256_test.c (accessed on 15 November 2022).
42. Espressif Systems. FFT Algorithm Implementation. Dsps_fft2r_sc16_ansi.c. Available online: https://github.com/espressif/esp-dsp/blob/master/modules/fft/fixed/dsps_fft2r_sc16_ansi.c (accessed on 15 November 2022).
43. Espressif Systems. FIR Algorithm Implementation. dsps_fir_f32_ansi.c. Available online: https://github.com/espressif/esp-dsp/blob/master/modules/fir/float/dsps_fir_f32_ansi.c (accessed on 15 November 2022).
44. Espressif Systems. IIR Algorithm Implementation. Dsps_biquad_f32_ansi.c. Available online: https://github.com/espressif/esp-dsp/blob/master/modules/iir/biquad/dsps_biquad_f32_ansi.c (accessed on 15 November 2022).
45. Linker Script for the ESP32. Available online: https://github.com/tinygo-org/tinygo/blob/release/targets/esp32.ld (accessed on 15 November 2022).
46. ESP32 Product Page. Available online: https://www.espressif.com/en/products/socs/esp32 (accessed on 15 November 2022).
47. The Internet of Things with ESP32. Available online: http://esp32.net/ (accessed on 15 November 2022).
48. Flashing Firmware. Available online: https://docs.espressif.com/projects/esptool/en/latest/esp32s3/esptool/flashing-firmware.html (accessed on 15 November 2022).
49. Visual Studio Code. Version 1.73. Available online: https://code.visualstudio.com/ (accessed on 15 November 2022).
50. Annamaa, A. Thonny,: A Python IDE for Learning Programming. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, Vilnius, Lithuania, 22 June 2015*; ACM: New York, NY, USA, 2015; p. 343.
51. Professional Collaborative Platform for Embedded Development. Available online: https://platformio.org/ (accessed on 15 November 2022).
52. Rust.Analyzer. Available online: https://rust-analyzer.github.io/ (accessed on 15 November 2022).
53. Rust on ESP-IDF "Hello, World" Template. Available online: https://github.com/esp-rs/esp-idf-template (accessed on 15 November 2022).
54. Visual Studio Code Support for TinyGo. Available online: https://marketplace.visualstudio.com/items?itemName=tinygo.vscode-tinygo (accessed on 15 November 2022).
55. TinyGo Install Guide on Linux. Available online: https://tinygo.org/getting-started/install/linux/ (accessed on 15 November 2022).
56. Go 1.14 Release Notes. Available online: https://go.dev/doc/go1.14 (accessed on 15 November 2022).
57. Go Language Features. Available online: https://tinygo.org/docs/reference/lang-support/ (accessed on 15 November 2022).