Comparison of ESP programming platforms

Filip Rak, Józef Wiora
Department of Measurements and Control Systems, Silesian University of Technology, Gliwice, Poland

Article Info

Article history:

Received Oct 1, 2020
Revised Feb 17, 2021
Accepted Mar 4, 2021

Keywords:

C
ESP8266
FreeRTOS
Lua
MicroPython

ABSTRACT

The growing popularity of ESP boards has led to the development of several programming platforms. They allow users to develop applications for ESP modules in different programming languages, such as C++, C, Lua, MicroPython, or using AT Commands. Each of them is very specific and has different advantages. The programming style, efficiency, speed of execution, level of advancement, or memory usage will differ from one language to another. Users mostly base their choice depending on their programming skills and goals of the planned projects. The aim of this work is to determine, which language is the best suitable for a particular user for a particular type of project. We have described and compared the mentioned languages. We have prepared test tasks to indicate quantified values. There is no common rule because each of the languages is intended for a different kind of user. While one of the languages is slower but simpler in usage for a beginner, the other one requires broad knowledge but offers availability to develop very complex applications.

This is an open access article under the CC BY-SA license.

Corresponding Author:

Józef Wiora
Department of Measurements and Control Systems
Silesian University of Technology
Akademicka 2A, 44-100 Gliwice, Poland
Email: jozef.wiora@polsl.pl

1. INTRODUCTION

Nowadays ESP modules are becoming more and more popular. ESP systems are low-cost Wi-Fi microcontrollers with full TCP/IP stack [1]. They are produced by Espressif Systems, a Chinese manufacturer. Thanks to the integrated Wi-Fi module, they can be used in many internet of things (IoT) projects [2], [3]. IoT is a network of physical objects which are not the only network of computers. They are devices of all type and size which communicate with each other and share information based on stipulated protocols [4], [5]. Realization of the IoT systems can be found in such areas as smart buildings, smart cities, smart measuring systems, smart energetic systems, or environmental monitoring. Because of the integrated Wi-Fi module, ESP can work in two modes: Access Point (AP) or Station (STA) [6]. AP mode is frequently used in remote monitoring and robotics to control any objects and collect measurement data [7]. STA mode is used to connect the ESP board to a Wi-Fi network established by an access point. That allows users for the creation of their own server where they can build their website. To provide communication between server and client, users can use such implemented protocols as HTTP, WebSocket, or MQTT. These are only several existing technologies available for users. Wide applications of the ESP boards attract many people to develop their own projects. Several solutions and programming platforms have been created through the years which support programming ESP modules. Espressif System provides an official Software Development Kit (SDK) which is a bundle of utilities and device-level application programming interfaces with optimized and precompiled libraries. Based on that SDK, other solutions have been created. Nowadays users can choose between programming in C, C++, Lua, MicroPython (MP), and using AT Commands.

Due to several existing solutions, users may be confused about choosing which programming language is the most appropriate for their purposes. Nowadays, the information which can be found by the users concerns only the particular programming language. It is difficult to find a reliable comparison of the existing solutions and valuable information that would clarify the reasons supporting the usage of the concrete solution. The lack of good information may lead users to apply the most popular, not necessarily the most appropriate ones. Therefore, they even may not realize that other solutions are available. According to our best knowledge, there is no guide that familiarizes users with existing programming platforms that they can choose to start developing their applications on the ESP modules. We can find many tutorials and a lot of information on the internet regarding only specific language but none of them covers all platforms in one work. Every language is very specific and has its outstanding features.

The existing problem is that users need to know and understand every platform separately and still might be confused about which platform is the most appropriate to their needs. The inspiration for this work was the inaccessibility of good comparisons and language performance tests. The overload of the non-specific information causes those new users (students, hobbyists) may get lost and discouraged to use different programming languages. In this work, researches have been conducted orientated toward familiarizing potential users with the existing solutions by clearly discussing available platforms. Thanks to this work users can base their choice without wasting time searching separated knowledge. It delivers the most important information about each platform and compares them. It allows selecting the most suitable solutions for the individual types of users. Researches concern each one of the listed programming languages and provide clear conclusions that answer the questions: Which programming platform is the best for a particular user? Which one is the most suitable for the particular type of project?

2. ESP PROGRAMMING

In this research, four ways of programming the ESP boards are considered. Development in C low-level programming language is often connected with Real-Time Operating System (RTOS). The official supported platform by Espressif is ESP8266 RTOS SDK which uses FreeRTOS as an operating system. The second approach to program ESP modules is the C++ language which uses Arduino libraries. It is the most popular solution which is supported by several programming environments like Arduino IDE, Visual Studio Code (VSC), or even Eclipse. The next are to use MP or Lua languages. They both require a dedicated firmware that has to be deployed into the board memory. They both offer a powerful tool in the shape of a file system. It can store more than one developed program, out of which a user can run whichever program he wishes. The last existing solution is to use AT Commands which allow sending commands to the board via serial communication. That way of programming an ESP board has not been considered in this work because it is intended to use it as a Wi-Fi module [8]-[10]. It is a very fast way to check communication with the board or test some modules but it is hard to compare it with programming languages.

2.1. C++

Usually, users do not have a choice between programming languages that they can use to program a particular microcontroller. ESP boards are supported by several solutions. Users, who start programming, will be influenced by a popularity indicator like also by the number of available and already implemented solutions. In that case, the most popular way to program ESP microcontrollers is the C++ language. The internet is full of prepared tutorials in different forms, so beginners might think that there is no other way to program that microcontroller. This popularity may be good because it allows for fast and easy getting started with programming ESP modules. The popularity of microcontroller programming in C++ is grounded in the success of Arduino boards. They have become very popular thanks to the number of available projects. Based on the existing Arduino libraries, users have to implement Arduino core for ESP8266. That core allows for writing the code using familiar Arduino functions and libraries. Thanks to the dedicated environments, it is simple to run implemented sketches directly on ESP8266 without an external microcontroller. The ESP8266 Arduino core includes all necessary libraries which support: control GPIO, communication over Wi-Fi using TCP and UDP, set up HTTP, mDNS and DNS servers, use a file system in flash memory, work with SD cards, servos, and communication protocols: SPI, I2C, or UART [11]-[15].

Working with ESP8266 Arduino core is easier for users who already got familiar with programming Arduino boards. Thanks to very well implemented libraries, in most cases users have only to call initialization function and then some control functions. To compile the written code, users can use several Integrated Development Environments (IDE). During this work, we have tested three most popular IDEs which are: Arduino IDE, Visual Studio Code, and Eclipse. Each of those IDEs requires a couple of steps to be performed. These allow flashing the written sketches into the board memory. Arduino IDE requires adding a package with the ESP boards that allow installing the ESP8266 platform. That package offers also a lot of prepared sketches.

One thing needed to do is choosing the correct board and set a communication port. In those few steps, Arduino IDE is ready to flash programs. The second environment is the Visual Studio Code. It is a free code editor. It requires installing a special dedicated package called Platform IO. That package can be also installed in Atom IDE. Platform IO is an open-source ecosystem for IoT development with a cross-platform build system [12]. That IDE allows developing software for more than 800 embedded boards and more than 35 development platforms such as Arduino, ESP8266, STM32, or Raspberry Pi. It is a very powerful package with a useful handful of add-ons. The third tested environment is Eclipse. It is mostly known for users who work with boards like STM32 or AVR. Eclipse also requires a dedicated package called Sloeber, which is free, open-source, and designed for Arduino boards, therefore it requires adding the ESP boards in the same way as in Arduino IDE. After these steps, Eclipse is ready to create a project and flash the program.

2.2. Lua

Another approach to program ESP boards is to use NodeMCU firmware that is an open-source Lua based firmware for the ESP8266 [15]. Dedicated ESP8266 boards with already loaded NodeMCU firmware are available on the market. The ESP8266 SDK can be categorized into two types: Non-OS SDK and RTOS SDK. The Non-OS SDK does not base on an operating system, it uses timers and call-backs as the main way to perform various functions. NodeMCU is implemented in C programming language and it is layered on the Espressif Non-OS SDK. The software is nowadays community-supported and the firmware can be run on any ESP board. The main Lua designed goals are speed, portability, extensibility, and small kernel size. The Lua is an asynchronous and event-driven script language [16], [17]. Other languages are procedural that makes a clear flow of execution [18], [19]. Lua requires the asynchronous mode and other approaches to develop and structure their applications. Referring to event-driven programming style, tasks are associated with given events by using SDK API which registers call-back functions to the corresponding events. All events are queued internally in SDK. Then firmware calls the task associated with a particular event one at a time. The SDK contains a scheduler that executes queued tasks with FIFO (first in first out) formula. An important thing is to obey time limits because tasks that run for more than 15 ms can disrupt other tasks, for example, Wi-Fi working. The recommendation is to keep medium priority tasks under 2 ms and low priority tasks under 15 ms to prevent the responsive work of the system.

The running watchdog will reset the microcontroller if the task takes longer than 500ms. Another limitation is the availability of the free Random-Access Memory (RAM) space. ESP hardware executes code in RAM but it can be also executed from Flash-mapped address space. Developers must be careful because it is easy to go out of free memory when they develop applications using the Lua core Run-Time System (RTS). The Lua RTS assumes that both Lua data and code are stored in RAM. The solution is to use the Lua Flash Store (LFS) patch which modifies the Lua RTS. It allows the Lua code and its associated constant data to be executed directly out of flash memory, in the same way, like is NodeMCU firmware. The NodeMCU project uses the SPI Flash File System (SPIFFS) to store files in the flash memory. The SDK invokes a start-up hook within the firmware on boot-up mode. During boot, firmware code initializes the Lua environment, and next it looks for init.lua file. It is a default executed after booting. In this file, the user should do any required initializations, call timer alarms, bind call-back routines, or run other scripts. More than one script can be stored in the SPIFFS that can be executed from init.lua file or by calling dedicated dofile command. Lua firmware offers several commands which can be used to manage the file system, run scripts, or reset the board. That is possible after communicate with the board using serial communication.

The default baud rate is set in firmware to 115 200 bit/s. ESP boards running Lua scripts requires dedicated firmware. The building of firmware has to be done by users. The ready firmware is not more available because of the increasing number of modules. There are three ways: #1 using Cloud Build Service intended for users who need a ready-made firmware to start developing their applications. On the website, users may choose modules: GPIO, UART, PWM, Timers, ADC, or Wi-Fi. After the task and set up their properties, the firmware will be prepared and send to the email box as a bin file. #2 using Docker Image which is intended for users who do not need full control over the complete toolchain. #3 setting up a Linux virtual machine. It uses the Linux build environment and offers a powerful environment with the complete toolchain. Once the firmware is created, the user may use esptool.py to flash the firmware into board memory. It is supported by many operating systems such as Windows, Linux, or macOS. Windows and macOS users can also use NodeMCU PyFlasher. The ESPlorer or NodeMCU-Tool allows for uploading a written script. ESPlorer IDE is a multiplatform tool with a graphical interface. It supports programming in LUA like also in MP. After connecting with the ESP board using serial communication, users can also use AT Commands that are supported by this IDE. The NodeMCU-Tool is based on Node.js and it offers only a console terminal.

2.3. MicroPython

MicroPython (MP) is a tiny open-source, interpreted, and object-oriented scripting language that runs on microcontrollers. It is a high-level programming language based on Python core. The MP idea is to keep code as simple as possible, so it is friendly for beginners [20]-[23]. The structure of the code is similar to the structure of Arduino applications. In opposition to Lua, MP is procedural. This simplicity has its disadvantages. It comes at the expense of speed and memory usage compared to the Arduino core or NodeMCU core. One of the biggest differences between MP and Lua is that the former is an interpreted language whereas the latter is a natively compiled code. It reveals that MP might be slower and more memory consumed what will be examined in this work. Moreover, applications that have tight timings, might not work with this programming language [20]. Thanks to the extensibility of MP with low-level languages like C or C++, users can mix them to provide faster work when it is needed. The greatest unique purpose of MP is the possibility to use the Read Evaluate Print Loop (REPL). This tool allows for connecting with the board and executing the code without any need of compiling or uploading the code. The applications can be just run on the board. This is a very fast way to test developed applications and run commands. Users can access REPL either by serial communication through the UART or by a Wi-Fi connection. USB connection requires a terminal emulator such as Putty or Hercules. Linux users can use the picocom terminal application.

The communication works with speed 115 200 bit/s. Through the REPL, users can even write code line by line and each of the lines will be executed up to date. Working with MP requires dedicated firmware. In contrast to Lua firmware, users do not have to build it by themselves. It is already done and the user can choose between three versions. A stable version is recommended for beginner users. There are two types of daily versions that are intended for more experienced MP developers. To deploy the firmware onto the ESP module it is highly recommended to use esptool.py which is currently supported tool. It requires an installed Python package. Then through the system console, esptool can be installed. Once the firmware is deployed the board is ready to use with REPL. Thanks to the internal filesystem, users can store more developed applications onto the ESP board and run whichever they want. Moreover, it is possible to create own directories and manage the filesystem. After boot, the firmware runs the boot.py file, and then it looks for the main.py file [20]. If the user wants to ensure executing application after every boot, he has to name that file as main.py. REPL is a very useful tool at the beginning of working with MP but it is not a good solution for developing more complex applications. There are several tools to manage filesystem: REPL, ampy, rshell, or mpfshell. During this work, we used ampy, a cross-platform command-line tool. Any editor can be used to write code, such as the Visual Studio Code. There is also a dedicated environment called uPyCraft IDE [24]. This IDE is not further supported for Linux users but it works under the Windows system. It offers a code editor with syntax checking. It also includes needed tools to manage the filesystem and run applications.

2.4. C (RTOS)

The last approach is to use the C language which is the most beneficial for developing embedded systems. In the case of ESP8266 boards, using a C is related to Real-Time Operating System (RTOS). The official supported SDK by Espressif System is ESP8266 RTOS SDK which uses FreeRTOS as an operating system. FreeRTOS is intended for microcontrollers and small microprocessors which is distributed freely under the open-source license [25]. The system is multitasking which allows executing several tasks pseudoconcurrently [26]. To allocate time for particular tasks, RTOS uses a scheduler. It is also a pre-emptive kernel that means that the currently running task can be expropriated when an event occurs. Using RTOS SDK to develop applications for ESP modules is not recommended for beginner developers. It requires knowledge about programming in C like also the ability to use pointers. Furthermore, it is important to have at least a basic knowledge of RTOS working. The basic knowledge should at least include: creating tasks, task scheduling up to their priority, task states, queues, semaphores, and mutex. Using RTOS has several benefits which are unique in comparison to other solutions. It allows for creating multitask applications that are scheduled by the system, which creates the illusion of executing several tasks simultaneously. RTOS is more deterministic because events and interrupts are handled within a defined time. Each task is allocated in a defined stack space that allows predicting memory usage. It is much more powerful than C++ or MP for more complicated and complex applications [25]. Working with ESP8266 RTOS SDK does not require any dedicated firmware. The user has to start with setting a toolchain. It includes programs to compile and build applications. It also allows flashing them into the board memory.

The toolchain requires a built-in make environment that is not included in the Windows system. Due to this issue, the preparation of the environments depends on the operating system. Both Linux and macOS systems require only installing xtensa tool and add its source path to the PATH environment variable. That will allow using dedicated commands in the system terminal. The GNU-compatible environment has been prepared for Windows users and it uses the MSYS2 platform. After unpacking it, users can work using a terminal. After setting the toolchain, we need to get ESP8266 RTOS SDK. It includes all of the required libraries which are provided by Espressif. This repository includes also several examples that can be used to fast check the correctness of Toolchain working. They can act as a user's first project's base. Not all libraries were included in official ESP8266 RTOS SDK. The ESP OPEN RTOS SDK has been also used in research work. The SDK is a community-developed open-source FreeRTOS. It is intended to use in both commercial and open-source projects. This SDK contains much more practical examples and have more working libraries than official Espressif release. Eclipse can also be used for developing an application. Every example includes makefile which contains a set of directives used by the make build tool to generate a target. The project can be created in Eclipse from the existing makefile. Then, after several steps of settings project properties, it will be possible to build and compile the whole project. Compiled code can be flashed in two ways. The first is to use terminal calling 'make flash' command which is included in the xtensa tool. The second is to create a target in the Eclipse environment that allows flash code directly from the application. The more complicated situation is in the case of OPEN RTOS SDK. It requires to add in Eclipse all library sources included in SDK. Official ESP8266 RTOS SDK only requires adding one environment variable IDF_PATH which includes the full path to the directory where SDK is installed.

3. RESEARCH METHOD

Out of the ESP boards, the three series of the ESP modules can be distinguished which are ESP8266, ESP32, and ESP32-S2. All modules are systems on a chip (SoC) that use the 32-bit Xtensa microcontrollers developed by Tensilica. The ESP8266 and ESP32 modules use the same LX6 processor whereas the newest ESP32-S2 uses the LX7 processor. The ESP8266 module is the poorest out of the distinguished series. It runs at 80 MHz or 160 MHz, has a 10-bit analog to digital converter (ADC), and 18 available pins. It includes peripherals such as UART, GPIO, I2C, I2S, SDIO, PWM, ADC, and SPI. The ESP32 differs from an older predecessor that it includes adjustable clock frequency, ranging from 80 MHz up to 240 MHz. It has 38 pins and more peripherals such as Bluetooth, SD cards interface, HALL sensor, and better 12-bit ADC. The newest series is ESP32-S2 which has 43 GPIO pins and includes different peripherals. In the ESP32 series, the security has been improved, but it has been more refined in the S2 series. ESP8266 is weak in this respect, but the newest modules provide flash encryption, secure boot, signature verification, and provide additional algorithms such as SHA or RSA. In this work, the ESP8266 module integrated on the Wemos D1 mini board has been tested. Figure 1 presents the electrical wiring of the board with particular peripherals. The entire electrical system creates a development board that can be used to learn and test working with ESP8266 module. In order to perform tests, the following benchmarks have been prepared:

- #1. Blinked LED;
- #2. Serial communication using UART;
- #3. ADC conversion with sending read values through UART;
- #4. LED brightness control using PWM controlled by potentiometer;
- #5. Servomotor control using PWM controlled by potentiometer;
- #6. Temperature and humidity measurement with DHT22 1-wire sensor;
- #7. Communication with OLED display using I2C protocol;
- #8. Simple web-server.

The prepared benchmarks have been chosen to use different popular modules and protocols which allow us to fully test an ESP8266 module. They have been realized in the mentioned programming languages and environments. To compare the languages, the dedicated indicators have been used that are: number of code lines, flashed file size, flashing duration, and maximum Frames per Second (FPS) that can be obtained on OLED display. They allow judging the differences more objectively than mentioning the advantages and disadvantages of a particular language.

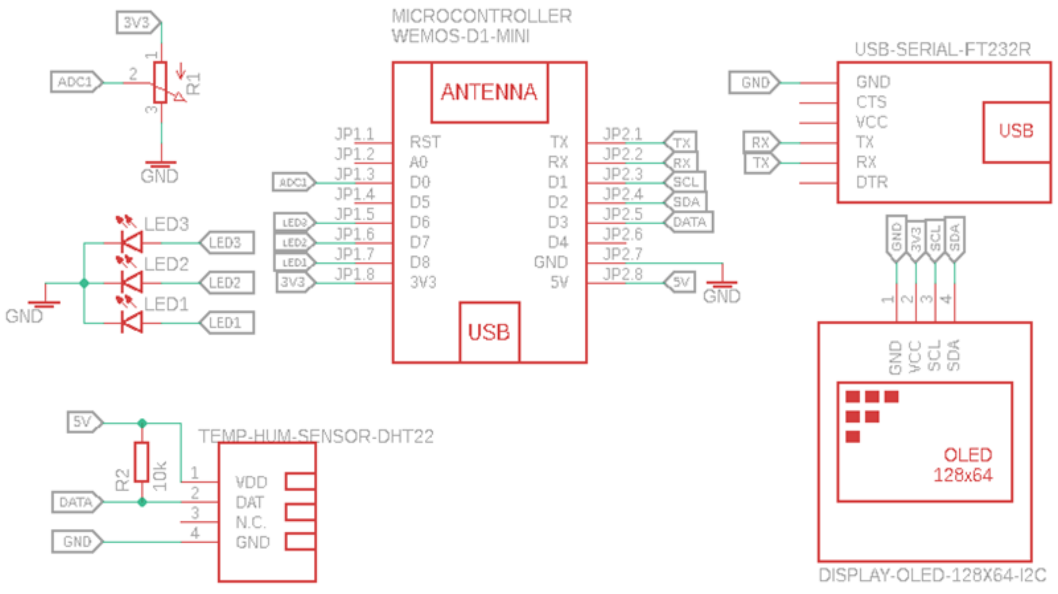

Figure 1. Electrical circuit applied in tests

4. RESULTS AND DISCUSSION

The tests performed led to quantified values presented in Tables 1-4. The first indicator, which is the number of code lines, has been presented in Table 1. Depending on the particular language, the values are different. The results confirm the main paradigm of MP and Lua languages to keep code as short as it is possible. The smallest value has been reached for those two languages. Arduino core is characterized by a few more code lines than Lua and MP. The worst results achieved a C language. It stems from a more complex environment in the form of RTOS. Users have to add extra libraries and do initializations on their own. There are not dedicated functions to initialize for example GPIO, but on the other hand, it has a big advantage. Users can initialize port in a more advanced way because they can setup properties such as pullups or external interrupts. MP and Lua keep code very short but users have to bear in mind that MP is not so powerful like Lua. In the case of MP, the reduction of the code led to a slowdown and to cause troubles with tight timings.

Table 1. Number of lines

Test No.	Test Name	C++	Lua	MP	C
#1	Blink LED	12	12	8	37
#2	Serial	8	3	5	32
#3	ADC	10	6	8	36
#4	Brightness	9	9	6	42
#5	Servo	13	10	7	43
#6	DHT22	18	7	11	43
#7	OLED	40	24	20	140
#8	Server	47	15	15	150

Table 2 presents the second indicator which is the size of the flashed file. We only check the size of the file which must be flashed each time the user makes a change in the code. The analysis concludes that the less optimal solution is using the Arduino core. In this case, MP and Lua have similar results and the scripts are very light-weight. They both owe it to the firmware which is flashed into Flash memory. In the case of Lua language, the firmware used during this work has 495616 bytes but it includes all modules necessary to implement benchmarks. When we take the firmware into consideration, the general size will be bigger than

Size reached using the Arduino core. The size of NodeMCU firmware depends on the number of included modules. For a particular application, the number of modules can be reduced to a minimum that will free more Flash space. Due to varying firmware size, it is not taken into consideration during comparison, but it is an important aspect to keep in mind. Even C language which uses RTOS and requires additional libraries to handle system working takes much less memory than Arduino core. In previous consideration, the RTOS SDK has the worst value of the indicator but it does not affect used memory space. Another checked indicator is the duration of the flashing process, summarized in Table 3. The best times reached Lua and MP. It takes about one second for the small programs. C++ and C languages have very similar results. Before flash, they both check for changes in code from the previous build and compile them using the xtensa tool. The big difference is between flashing duration using official ESP8266 RTOS SDK and OPEN RTOS SDK. The official distribution is faster than the open-source version. Developing with OPEN RTOS SDK is not so efficient in comparison to Lua or MP. Every flash, even after the small change, takes much more time. Developing using Lua or MP allows very fast test applications and check the impact of small changes. The internal file system, which allows store more applications or scripts, also supports faster development. There are also differences between particular environments that are used to flash and develop the application with Arduino core. The Arduino IDE and Eclipse have similar values of the indicator, but VSC needs double more time to flash applications. It is probably a result of different flashing speed set directly in the environment.

Table 2. Flashed file size in bytes

Test No.	Test Name	C++	Lua	MP	C
#1	Blink LED	261856	242	144	145664
#2	Serial	266448	89	75	149244
#3	ADC	267312	194	141	144288
#4	Brightness	263072	371	138	151104
#5	Servo	263408	421	158	151120
#6	DHT22	268784	323	199	154304
#7	OLED	282656	662	436 (5930)*	425984
#8	Server	316800	444	442	270336

*This application required an additional library to interface OLED display which is not included in the firmware. The value in parentheses is a summary of the script and the library.

Table 3. Flashing time in seconds

Test No.	Test Name	C++ (Arduino IDE)	C++ (VSC)	C++ (Eclipse)	Lua	MP	C
#1	Blink LED	10.0	22.3	10.6	< 1	1	10.8
#2	Serial	10.2	22.2	11.4	< 1	1	11.1
#3	ADC	10.3	22.6	13.7	< 1	1	10.7
#4	Brightness	10.2	22.8	13.5	< 1	1	11.2
#5	Servo	10.3	22.0	13.5	< 1	1	11.2
#6	DHT22	10.6	22.9	13.2	< 1	1-2	11.3
#7	OLED	13.5	23.4	13.8	< 1	1-2	39.0*
#8	Server	20.0	26.0	14.5	< 1	1-2	25.0*

* The applications have been implemented using OPEN RTOS SDK where the others use official ESP8266 RTOS SDK.

The last test was to check the number of FPS that can be reached by particular languages on the OLED display – Table 5. It uses the SSD1306 controller and I2C communication protocol. The results are presented in Table 4 and they prove the thesis that MP is slower than other solutions. It allows us to get only 8 FPS whereas Arduino core reaches up 35 FPS. In that case, the C language is slower than C++, because RTOS has to switch the threads. The available library could be also better written and optimized in C++ which affects the results. The Lua does not allow for using the while loop, which causes those users cannot do things as fast as they wish. The task waits for the appropriate event such as timer interrupt. Any try to create an infinity loop will cause a panic error and the board will be restarted by the watchdog. The board will be suspended and the only way to fix it is to reflash all firmware. The task which refreshes the display must be included in the callback function which is called by the registered timer. When the time was set to 500 ms (after which the function is executed) we obtained the expected 2 FPS. Then when we were reducing the time, the FPS was not growing up. For timer set to 200 ms we did not get the 5 FPS but 2/3 FPS. After reducing to a very small period, we got maximally 4 FPS. This value is much below the expectations, so we leave it as an open problem where more research is suggested. The good score reached by the Arduino core does not prove that it is the most powerful platform. The efficiency of the Lua and RTOS SDK will be noticeable only in more complex projects. The FPS test consisted of displaying one inscription and clearing the display as fast as possible. When the application has to do more tasks, the FPS drop in the case of the Arduino core could be higher.

Users, who decided to use the C++ language should also consider which programming environment they want to use. The Arduino IDE is the poorest of the three tested programs. Visual Studio Code and Eclipse allow setting the dark mode which is desired by the developers. They include also additional features that help users during programming. One of them is code autocompletion and syntax prompting which speed up the coding. They also allow us to divide the window horizontally or vertically and open in a new window another source code file. VSC and Eclipse help users in navigating in the code and they highlight errors instantly. Table 5 summarizes the above-mentioned languages. For each language, the pros and cons have been listed.

Table 4. Display refreshing frames per seconds

Language	FPS
C++	35
Lua	4*
MP	7 to 8
C	27

* Due to the asynchronous and event-driven programming style in Lua language, it is impossible to use while loop and to refresh display as fast as it is possible.

Table 5. Features of tested languages

Language	Advantages	Disadvantages
C++	supported by several environments availability of many examples easy way to start working with ESP require only elementary knowledge	size of the flashed file is not optimal
Lua	short programming codes internal file system which can store more scripts the small size of scripts size fast execution of the code	require to build firmware by user different programming style easy to go out of free RAM programs executed in RAM if users do not use LFS hard to debug a very bad description of code errors a small number of examples
MP	short programming codes the small size of applications users can use REPL run application without flashing them internal file system which can store more applications	much slower than rest languages applications that require tight timings may not work it is an interpreted language use more memory require firmware
C	work with RTOS allow developing more complex applications predictable working better memory management availability of several examples	require an advanced knowledge hard to get started longer codes than in other languages

5. CONCLUSION

The conducted researches prove that there is no perfect and unique solution for programming language choice. Every language has its advantages and disadvantages. The new knowledge that comes from the performed analysis allows us to answer the questions stated in the Introduction. It brings information for new users about efficiency and also allows the comparison of the particular language. This comparison gives the opportunity to choose a proper solution by a certain kind of developer. Previously it was not obvious which solutions should be used and it also was impossible to find a comparison of the given platforms. The users of the microcontrollers can be categorized by their knowledge and their developing experience. We decided to split them up into 3 groups: beginners, intermediate and advanced developers. Unambiguously, we can state that the best choices for beginners are MP and C++. They both do not require advanced knowledge about programming. MP is characterized by keeping code as short as it possible and it is not intended for a complex application. From the FPS tests, we conclude that the Arduino core is much more powerful than MP and allows more advanced development. MP will be also a good choice for users who are already familiar with Python language and their purpose is not to build complex applications with tight timings. It is also easy to get started working with these languages, which is desired by beginner users.

The intermediate developers are users, who worked previously with other microcontrollers and have already gained basic knowledge in embedded development. They can decide to use Lua, but we mark that not every user will be able to work with an event-driven programming style. The Lua keeps also code short but better efficiency and faster work differ Lua from MP. This platform is desired for users who want to create more complex applications but during conducting the benchmarks we also got unpredictable working and crashes. The users should stay aware and use LFS for official releases of the applications. We also suggest that C++ might be a good choice because it gives a lot of useful examples and the FPS test proved that its efficiency does not stand out from other solutions. For users who are very advanced developers and have knowledge about RTOS, the C programming language is intended. The ESP8266 RTOS SDK is desired to build very complex applications that require RTOS to handle all tasks. It is a very powerful tool and it is unnecessary to use that SDK to develop some simple applications. The better choice will be to use C++ or MP which are very easy to get started in opposite to Lua and RTOS SDK.

Another new knowledge result from the researches and tests is the possibility to adopt the proper solution for the users' aim. It leads to the second approach which is to categorized users by their project goals. We have distinguished three levels of the projects: Do It Yourself (DIY) projects, more complex applications but not intended for sale, and very complex applications desired for sale. For small applications and DIY projects, the C++ or MP should be used, because they offer the fastest way to get started and fast development. Due to the better efficiency of LUA and C++, we suggest to use them in more complex applications which use more modules and have to handle more tasks. For users who are building more complex devices that can be intended for sale, the C with RTOS and LUA languages are recommended. The RTOS provides stable and predictable work that is desired in these devices. The analysis of the result and summary based on the previously performed analysis led to clear conclusions that offer concrete solutions for specific users. They suggest which platform should be used and give the view for all languages in one work. Every user based on the conducted researches can regard given indicators and base their choice on them. The basic general work has been done and we encourage readers to continue our researches by testing more deeply the efficiency of the particular platforms. The next work should be more targeted at performing the speed tests and specify the conclusions about efficiency. We encourage to especially test the NodeMCU firmware and to solve the open problem with LUA available at https://github.com/MrHause/OLED_FPS_LUA.

ACKNOWLEDGEMENT

This work was funded by the Polish Ministry of Science and Higher Education.

REFERENCES

[1] A. Patel, P. Devaki. "Survey on NodeMCU and Raspberry Pi: IoT," *International Research Journal of Engineering and Technology*, vol. 06, no. 4, pp. 5101-5105, 2019.

[2] J. Mesquita, D. Guimarães, C. Pereira, C. Santos, L. Almeida, "Assessing the ESP8266 WiFi module for the Internet of Things," *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, Turin, 2018, pp. 784-791.

[3] A. G. Azwar, *et al.* "Smart Trash Monitoring System Design Using NodeMCU-based IoT," *2019 IEEE 13th International Conference on Telecommunication Systems, Services, and Applications (TSSA)*, Bali, Indonesia, 2019, pp. 67-71.

[4] K. K. Patel, S. M. Patel, "Internet of Things-IOT: Definition, Characteristics, Architecture, Enabling Technologies, and Application, Future Challenges". *International Journal of Engineering Science and Computing*, vol. 6, no. 5, pp. 6122-6131, 2016.

[5] L. K. P. Saputra, Y. Lukito, "Implementation of air conditioning control system using REST protocol based on NodeMCU ESP8266," *2017 International Conference on Smart Cities, Automation & Intelligent Computing Systems (ICON-SONICS)*, Yogyakarta, 2017, pp. 126-130.

[6] M. Gergeleit, "Autotree: Connecting Cheap IoT Nodes with an Auto-Configuring WiFi Tree Network". *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, Rome, Italy, 2019, pp. 199-203.

[7] H. Ouldzira, *et al*. "Remote monitoring of an object using a wireless sensor network based on NODEMCU ESP8266," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 16, no. 3, pp. 1154-1162, 2019.

[8] M. R. Sahay, M. K. Sukumaran "Environmental Monitoring System Using IoT and Cloud Service at Real-Time," EasyChair Preprint, no. 968, 2019

[9] S. P. Makhanya, E. M. Dogo, N. I. Nwulu, U. Damisa, "A Smart Switch Control System Using ESP8266 Wi-Fi Module Integrated with an Android Application," *2019 IEEE 7th International Conference on Smart Energy Grid Engineering (SEGE)*, Oshawa, ON, Canada, 2019, pp. 125-128.

[10] D. Bismor, "System for Vehicle Sound and Vibration Monitoring using MEMS Sensors," *2019 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, Poznan, Poland, 2019, pp. 50-55.

[11] Jenifer, D. J. Aravindhar, "Iot Based Air Pollution Monitoring System Using Esp8266-12 With Google Firebase", International Conference on Physics and Photonics Processes in Nano Sciences, vol. 1362, Art. no. 012072, 2019.

[12] I. Grokhotkov, "ESP8266 Arduino Core Documentation. Release 2.7.1-7-g4519db8". 2020.

[13] A. Ara1, S. Jawaligi, "NodeMCU (ESP8266) Control Home Automation using Google Assistant", *International Research Journal of Engineering and Technology*, vol. 6, no. 7, 3644-3648, 2019

[14] S. Barai, D. Biswas, B. Sau, "Estimate distance measurement using NodeMCU ESP8266 based on RSSI technique," *2017 IEEE Conference on Antenna Measurements & Applications (CAMA)*, Tsukuba, 2017, pp. 170-173.

[15] L. Shkurti, *et al*. "Development of ambient environmental monitoring system through wireless sensor network (WSN) using NodeMCU and "WSN monitoring," *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, Bar, 2017, pp. 1-5.

[16] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, "The evolution of Lua", *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2007, pp. 2-1 – 2-26.

[17] J. Ch. Herman, "Beginning Lua Scripting." in *Beginning Game Development with Amazon Lumberyard*. Apress, Berkeley, CA, 2019, pp. 167-190.
[18] M. Soldevila, B. Ziliani, B. Silvestre, D. Fridlender, F. Mascarenhas, "Decoding Lua: formal semantics for the developer and the semanticist," *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages*, 2017, pp. 75-86.
[19] J. T. P. Wibowo, B. Hendradjaya, Y. Widyani, "Unit test code generator for lua programming language," *2015 International Conference on Data and Software Engineering (ICoDSE)*, Yogyakarta, 2015, pp. 241-245, doi: 10.1109/ICODSE.2015.7437005.
[20] Ch. Bell, "MicroPython for the Internet of Things". Apress 2017.
[21] M. Khamphroo, N. Kwankeo, K. Kaemarungsi, K. Fukawa, "MicroPython-based educational mobile robot for computer coding learning," *2017 8th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES)*, Chonburi, 2017, pp. 1-6.
[22] S. Plamauer, M. Langer, "Evaluation of MicroPython as Application Layer Programming Language on CubeSats," *ARCS 2017; 30th International Conference on Architecture of Computing Systems*, Vienna, Austria, 2017, pp. 1-9.
[23] R. K. Kodali, K. S. Mahesh, "Low cost ambient monitoring using ESP8266," *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, Noida, 2016, pp. 779-782.
[24] K. Dokic, B. Radisic, M. Cobovic. "MicroPython or Arduino C for ESP32-Efficiency for Neural Network Edge Devices." *International Symposium on Intelligent Computing Systems*. 2020, pp. 33–43.
[25] R. Goyette, "An Analysis and Description of the Inner Workings of the FreeRTOS Kernel", Carleton Univ. 2007.
[26] L. Bogdano, R. Ivanov, "Flash Programming Low Power Microcontrollers over the Internet," *2019 IEEE XXVIII International Scientific Conference Electronics (ET)*, Sozopol, Bulgaria, 2019, pp. 1-4.

BIOGRAPHIES OF AUTHORS

Filip Rak was born in Poland in 1996. He received the Eng. degree in Automation and Robotic from Silesian Technical University in 2019. Currently a M.Sc. student in Automation and Robotic with Robotic specialization at the same university. His major research interests are the Internet of Things, embedded systems, and software development.

Józef Wiora is a professor at the Silesian University of Technology, Gliwice, Poland. He obtained his MSc, Eng. (2000) in Electronics, Ph.D. (2006), and D.Sc. (2019) in Automatic Controls and Robotics, all at the same university. His research focuses on improvement in measurement quality, optimization in measurement procedures, microcontroller-based measurement circuits. Currently also works at Bombardier Transportation, Katowice, Poland where deals with the European Train Control System.