Advances in Dynamical Systems and Applications.
ISSN 0973-5321, Volume 17, Number 2 (2022) pp. 599-614

https://www.ripublication.com/adsa.htm

Rust-like Programming Language for Low-resource Microcontrollers

Fernando Martinez Santa[1] **, Santiago Orjuela Rivera**[2] **and Fredy H. Martinez Sarmiento**[1]

[1]*Universidad Distrital Francisco José de Caldas, Facultad Tecnológica, Bogotá D.C, Colombia.*
[2]*Corporación Unificada Nacional de educación superior CUN, Escuela de Ingeniería, Bogotá D.C, Colombia.*

Abstract

This article proposes a programming structure for low-resource microcontrollers over the name of Sokae Project, this is inspired by Arduino and Micropython projects and pretends to work as a medium point between both programming structures taking the best features of each kind of languages and programming structures. The project's name is inspired by the crab pet of Rust programming language. Sokae means crab in Emberá Chamí language, so the project's name is a tribute to Emberá native people. The programming structure is composed of three main components: a language based on Rust syntax, a transpiler that turns the defined Rust- like language into C, and a generic Application Programming Interface API. The main goal of this project is to obtain a cross- platform programming structure for programming low-resource microcontrollers over the same language and API. Sokae language is a small subset of Rust programming language following its overall syntax. On the other hand, the transpiler from Sokae language to C is implemented by using Python and the SLY module as lexer and parser. Likewise, the API is written for the native C compiler for each microcontroller, so it is necessary to use it as a part of the project. Several application examples are tested in order to check the correct working of the overall programming structure, just over the XC16 compiler for PIC24/dsPIC33 microcontroller family. Finally, after several tests, the proposed programming structure shows that it is possible to use modern language structure to program any kind of microcontroller no matteríts limited resources.

Keywords: Programming Language, Rust, Transpiler, Microcontroller, Embedded Systems, Compiled Language, Microcontroller API

1. INTRODUCTION

Microcontroller programming has always been specialized and dependent on each microprocessor architecture. First, when these were programmed by using assembly language, the pro- grammer had to learn the complete specific instruction set of the processor, so changing to another architecture was too complicated and took a long time. When the high level program- ming languages were introduced in the microcontrollers world, the development and debugging times were reduced signifi- cantly, but features such setup registers, peripherals and others, keep depends on a certain processor's architecture knowledge by the programmer [1], [2]. Trying to generate a cross-platform programming tool some projects have been born [3], including several virtual machine implementations [4] such as JavaScript [5]. One of the most popular projects is Arduino, which consists of a C language compiler (with some additions) plus an API, which makes it easy to program the microcontrollers over a specific hardware platform. Arduino was designed to be used by people with minimal or null electronics knowledge [6], however, a lot of different industrial and academic applications have been developed using it [7]. Other projects such as Micropython, implements a subset of Python language able to run an interpreterón microcontrollers with certain memory requirements. Nowadays, Micropython has been ported to a lot of different architectures [8] and has been used in a lot of different applications, mainly in Internet of Things IOT. Finally, *Tinygo* project, implement a version of Go language able to run on microcontrollers. *Tinygo* offers a modern syntax and featured language, and the advantage of being compiled [9]. All of these projects implement complete programming structures for programming microcontrollers with ease, but most of the time sacrificing memory room and execution time. Most of the small or low-resource microcontrollers are out of the scope of those projects, basically due to the amount of memory available. In order to obtain the best level of optimization it is necessary to use the native compileróf the microcontrollers manufacturer [10], [3], generally over *C* language. Therefore, a programming structure that includes as a part the native *C* compiler of each architecture, and an upper modern language layer could reach similaróptimization levels with high level features.

A multilayer programming structure like the described needs to have a transpiler, which compiles or translates the upper layer language to the native *C* [11]. Those transpilers are very common nowadays [12], translating among compiled [13], in- terpreted [14], [15] and virtual-machine-based [16] languages, depending on the necessity. The aim of the transpilers is generally to reuse source code that comes from other different languages [13], orímprove the performance of the program changing the platform or language (for instance turn *Python* into *Rust* [14]), even translating source code to processor-less hardware [17].

On the other hand, nowadays several new programming languages have emerged to solve some of the common problems of the standard languages such as memory management and safety. One of the most popular new languages is *Rust*, which is an open source statically-typed programming language with a lot of modern

features that make the development easy. *Rust* is preferred specially by systems programmers [18], so much so the new versions of Linux kernel will include *Rust* along with *C*, due to its safety features. The popularity of *Rust* is spreading to the microcontroller world too, having different applications such as IOT [19] or even on multi-core microcontrollers [20],[21].

The programming structure proposed in this paper uses a high level language based on *Rust* as the upper layer language, and a transpiler from this *Rust*-like to the specific architecture native *C* compiler, which is finally used to generate the programming binary file. Likewise, a cross-platform API is proposed in order to make it easy to program across different architectures, this one is implemented over the native *C* compileróf each architecture (in this case XC16 compiler). The transpilerís proposed to be implemented by means of using free software, in this case all of the algorithms will be implemented in the *Python* language using the modules SLY.

The paperís organized as follows: Section 2 presents the methodology to implement the overall proposed programming structure, including the Sokae language definition (subsection 2.1), the Sokae-to-C transpiler implementation in Python (subsection 2.2), and the first version of the API implemented for the XC16 compiler (subsection 2.3). Section 3 presents the results of implementing the proposed programming structure by several test source codes. Finally, Section 4 shows the conclusions about this research's main ideas, including possible future work.

2. DESIGN and METHODOLOGY

A complete structure for programming microcontrollers using the proposed Rust-like language (named *Sokae*) was designed and developed. This programming structure includes the transpiler from *Sokae* language to *C*, the Application Programming Interface (API) in both languages, and the native *C* compiler for the specific microcontroller, as shown in Figure 1. The main goal is the user writes the code in *Sokae* language over a standard API and obtains the executable or binary file for a specific microcontroller without taking care about the inner workings of the programming architecture. For the scope of this article, the test only were done using the Microchip® PIC24FJ128GA010 microcontroller (on an Explorer 16 board) and the XC16 Compiler, but the programming structure is modular so it is relatively easy to include other microcontrollers or boards.

2.1. *Sokae* language definition

Sokae is the name given to the proposed language. This is based on Rust programming language, which is one of the most preferred new languages nowadays [18], mainly for its performance and safety features [22], [23], [24]. The name was taken from the *Emberá Chamí* language and means crab. This was inspired by the crab pet of *Rust* programming language, and at the same time is a

tribute to Emberá people who live in the pacific coasts of Panamá, Colombia and Ecuador.

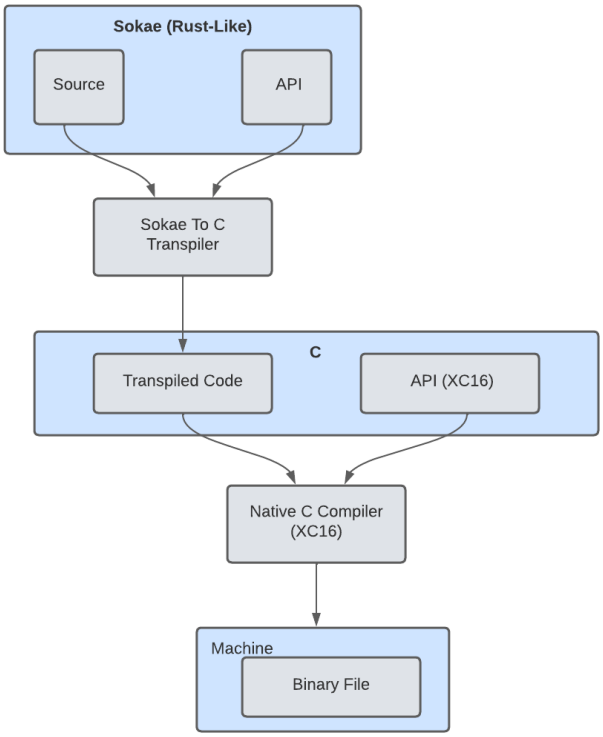

Figure 1: Complete proposed programming structure.

Sokae is a statically typed programming language based on the *Rust* syntax [13], designed to be used on low-resource microcontrollers. Just like *Rust*, *Sokae* shares most of the syntax basis of *C*. Features like the final semicolon, curly braces as code block delimiters, the *main* function, etc. makes *Sokae* easy to understand and of course easy to translate (transpile). Table 1 shows an example code written in *Sokae* and using Sokae API, which implements a blinking LED in the pin 0 of the port B using a PIC24FJ microcontroller. Likewise, the second column of Table 1 shows the *C* equivalent of the same code using the same API for the XC16 compiler. Some of the main features of *Sokae* language are shown in the left column of Table 1, such as:

- It is imperative to have a *main* function.
- All instructions end with a semicolon or curly bracket close.
- In a code block (always delimited by curly brackets), the last instruction can omit the final semicolon.
- All function declarations start with the reserved word **`fn`**.
- The identifier names prefer to use *snake* case as in Rust language (like **`sleep_ms()`** function, but this is not imperative like the **`pinHigh()`** function, which uses *camel* case in this API version.

Table 1: *Sokae* main function comparing

Sokae	*C (XC16)*

`use machine::pin;`
`use time::sleep_ms;`
 	
`fn main() {`
`pin(A6,OUT);`
`loop {`
 	
`pinHigh(A6);`
 	
`sleep_ms(500);`
 	
`pinLow(A6);`
 	
`sleep_ms(500)`
`}`
`}`
	`#include "./settings.h"`
	`#include "./ports.h"`
	`#include "./machine/pin.h"`
	`#include "./time/sleep_ms.h"`
	
	`int main(void) {`
	`pin(A6,OUT);`
	`while(true) {`
	`pinHigh(A6);`
	`sleep_ms(500);`
	`pinLow(A6);`
	`sleep_ms(500);`
	`}`
	`return 0;`
	`}`

- There are other loop instructions supported like **`loop`**, which is used for implementing infinity loops. This is really useful in multi-tasking applications.
- For including different modules or libraries, the reserved word **`use`** is utilized.
- The library including instruction can use the double colon `::` to include only a specific component instead of the complete library. This is mainly applicable to the implemented API (explained in detail in subsection 2.3).

Other *Rust* features has been implemented in *Sokae* such as:

- **`let`** keyword for declaring variables.
- **`i8`**, **`i16`**, **`i32`**, **`i64`**, **`isize`**, **`u8`**, **`u16`**, **`u32`**, **`u64`**, **`usize`** and **`char`** types.
- Default type inference in declaring.
- Suffix annotation in declaring.
- Underscore in literals for improving its readability.

On the other hand, all of the variables in *Sokae* are mutable by default unlike *Rust*, in which all variables are immutable by default. For that reason, **`mut`** *Rust's* keyword is not implemented in *Sokae*. This difference was implemented thinking

of being compatible with most of the native *C* microcontroller's compilers, and therefore for being easier to transpile. The rest of *Sokae* features will be explained by examples in the *Sokae* to *C* transpiler subsection.

2.2. *Sokae* to *C* Transpiler

By definition, a transpiler differs from a compiler in that the transpiler translates source code between programming languages with the same abstraction levels. In this case a transpiler from *Sokae* to *C* was implemented, being *Sokae* based on *Rust* and thus with the same abstraction level of *C*. The Transpiler from *Sokae* language to *C* was implemented using *Python* programming language and the *PLY* module as lexer and parser analysers for the input language. First, in the lexer analyzer, all of the tokens of *Sokae* language are defined, such as keywords, operands and other punctuation symbols, as shown in the source code extract of List 1.

List 1: *Sokae* lexer implemented in *Python* and *SLY* (extract)

```
tokens = { MODULUS,IDENTIFIER,
    KW_LET, KW_STATIC,  KW_FN, KW_RETURN,
          …
        I8, I16, I32, I64, U8, U16, U32, U64,
        …  }
U8          = r'u8'   #Tokens
U16         = r'u16'
U32         = r'u32'
…
literals = {',', ';', ':', '(', ')', '{', '}', … }
ignore = ' \t'
ignore_comment = r'//.*'     #Comments
…
```

Once the Lexer reduces the character flux to an easier-to-analyze token flux, the parser implements all of the syntactic rules of *Sokae* language. The List 2 shows an extract of the production syntactic productions implemented using *SLY*. The *Sokae* grammar implements a subset of the syntactic production of the official *Rust* grammar [25].

List 2: *Sokae* Parser implemented in *Python* and *SLY* (extract)

```
…
@_( 'KW_LET IDENTIFIER ":" Type "=" Expression ";"',
    'KW_LET IDENTIFIER "=" Expression NumericType ";"',
    'KW_LET IDENTIFIER ":" Type ";"' )
def LetStatement(self, p):
    self.exType = ''
    if len(p) == 7:
        return  p.Type  +'  '  +p.IDENTIFIER  +'  =  '
+p.Expression +';'
    elif len(p) == 6:
        return p.NumericType +' ' +p.IDENTIFIER +' = '
+p.Expression +';'
    elif len(p) == 5:
        return p.Type + ' ' + p.IDENTIFIER + ';'
…
@_( 'Expression "&" "&" Expression', 'Expression "|" "|"
Expression' )
def LazyBooleanExpression(self, p):
    return p[0] + ' ' + p[1] + p[2] + ' ' + p[3]
…
@_( 'U8', 'U16', 'U32', 'U64' )
def NumericType(self, p):
    return p[0].replace( 'u', 'uint' ) + '_t'
…
```

In *SLY* each syntactic production is implemented by a function definition as shown in List 2, where there are three different syntactic production examples: `LetStatement`, `LazyBooleanExpression` and `NumericType`. *SLY* uses Python's function decorators to implement the different syntactic rules applied to each syntactic production, for instance the production `LetStatement` is the implementation of three of the four possible variable declarations in *Sokae* language. That syntactic production is equivalent to the one Backus-Naur Form (BNF) shown in List 3, where the simple variable declaration, the declaration plus assignment and the declaration plus assignment by suffix annotation are defined.

List 3: *Sokae* syntactic production for variable declaration in BNF

```
LetStatement  ::=  KW_LET  IDENTIFIER  ":"  Type  "="
Expression ";"
               |  KW_LET  IDENTIFIER  "="  Expression
NumericType ";"
              | KW_LET IDENTIFIER ":" Type ";"
```

The transpiler reads the source code written in *Sokae* and creates a second source file code in *C*, with the same name of the original and different file extension as shown in Figure 2. For compatibility with standard source code editors, *Sokae* uses the `.rs` file extension, which is the same one of *Rust* language.

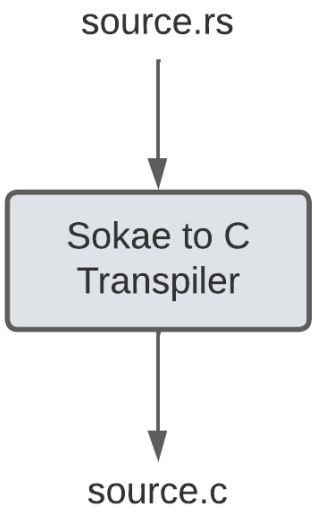

Figure 2: Transpiler's input and output files.

2.3. API definition

A simple Application Programming Interface was implemented in order to make the microcontroller's programming process easier. The main idea is designing a general cross-platform API implemented in *Sokae* language and in the specific microcontroller native *C* compiler, which include the basic features of peripherals of most microcontrollers. This proposed API includes the followings peripherals and features:

- General purpose input/output.
  - pin type declaration
  - setting high and low
  - setting specific binary value
  - reading an input value

- Analog to Digital Converter (ADC).
  - ADC setting up
  - ADC reading value

- Universal Asynchronous Receiver Transmitter (UART) .
  - UART setting up
  - single byte transmitting
  - single byte receiving

- Timing features.
  - delays in microseconds
  - delays in milliseconds

The API has to be implemented for each of the supported microcontrollers or boards following the folder structure shown in Figure 3, in order to maintain the compatibility across all the hardware devices. In a project, this folder structure allows to include complete modules such as `machine` or individual sub-modules such as `machine/pin.h`, this is pretty important in low-resource microcontrollers in order to save program memory. This last module includings are possible in *Sokae* language using the syntax: **`use`** `module;` for complete modules and: **`use`** `module::submodule;` for submodules, which will be transpiled to `#include "./module.h"` and `#include "./module/submodule.h"` respectively. When a complete module is include, the `./module.h` includes all of the *.h* files in the `Module` folder in the folder's API structure. For instance if the follow code is traspiled: **`use`** `time;` , the `sleep_us.h` and `sleep_ms.h` header files are included in the transpiled *.c* code.

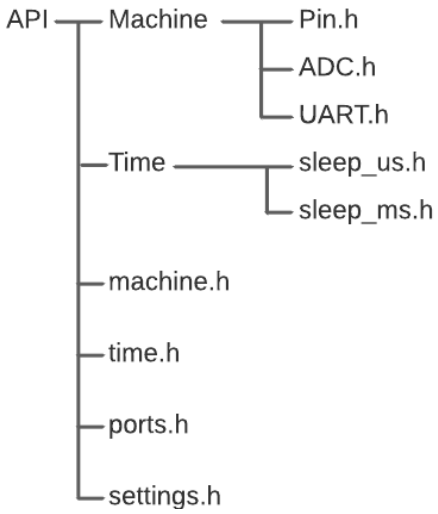

Figure 3: Folder tree for the implemented API.

3. RESULTS

The proposed programming structure including the *Sokae* to *C* transpiler and API definition was released as a free software project named Sokae Project in the https://gitlab.com/fermarsan/sokae-project URL, with the proposed to be a seed project for the embedded systems programming community.

The complete programming structure was tested by using the Microchip® PIC24FJ128GA010 microcontroller on the Explorer 16 development board. This is a 16-bit microcontroller with 125kB of program memory and 8kB of data RAM. Several tests were done for most of the *Sokae* features. Table 2 shows the comparison of the primitive variable declaration in *Sokae* and its corresponding *C* transpiling for the XC16 compiler. The advantage of using the Rust-Like variable definition is that each variable is bit-wide explicit no matter the hardware. This also includes the **`usize`** and **`isize`** integer types which adjust to the processor's registers wide.

Table 2: Primitive types declaration

Sokae	*C (XC16)*
`let var0: bool; //boolean`	`bool var0;`
`let var1: i8;  //integer`	`int8_t var1;`
`let var2: i16;`	`int16_t var2;`
`let var3: i32;`	`int32_t var3;`
`let var4: i64;`	`int64_t var4;`
`let var5: u8;  //unsigned integer`	`uint8_t var5;`
`let var6: u16;`	`uint16_t var6;`
`let var7: u32;`	`uint32_t var7;`
`let var8: u64;`	`uint64_t var8;`
`let var9: f32;  //floating point`	`float var9;`
`let var10: f64;`	`long double var10;`
`let var11: char;  //char`	`char var11;`
`let var12: usize; //processor's`	`uint16_t var12;`
`let var13: isize; //base integers`	`int16_t var13;`

Also, Table 3 shows the transpiled code for declaring and assignment instructions in *Sokae*. That example shows the use of the underscore symbol "`_`", which is used for improving the readability of large numbers. Likewise, the special notation for hexadecimal, octal and binary literals are shown, being only different to *C* the octal literals which start with the "`0o`" (zero + o) sequence.

Another interesting *Rust* feature implemented in *Sokae* language is the suffix annotation, with which it is possible to add a suffix to the literal value to indicate its type. As previously shown in List 3, the syntax for the suffix annotation avoids the use of colon "`:`" character and includes the assignment symbol and the specific suffix. The suffixes use the same type keywords defined in subsection 3.1, as shown in Table 4.

Table 3: Primitive types declaration and assignment (including underscore character)

Sokae	*C (XC16)*
`let var0: bool = true;`	`bool var0 = true;`
`let var1: bool = false;`	`bool var1 = false;`
`let var2: i8 = 129;`	`int8_t var2 = 129;`
`let var3: i64 = -6_835_292;`	`int64_t var3 = -6835292;`
`let var4: u8 = 0b0011_0101;`	`uint8_t var4 = 0b00110101;`
`let var5: u16 = 0o073452;`	`uint16_t var5 = 0073452;`
`let var6: u32 = 103_937_465;`	`uint32_t var6 = 103937465;`
`let var7: u64 = 0xAAFF_7625;`	`uint64_t var7 = 0xAAFF7625;`
`let var8: f32 = 1_342.56;`	`float var8 = 1342.56;`
`let var9: f64 = -34.035_440;`	`long double var9 = -34.035440;`
`let var10: char = 'f';`	`char var10 = 'f';`

Table 4: Primitive types declaration and assignment with suffix annotation

Sokae	*C (XC16)*
`let var2 = 129i8;`	`int8_t var2 = 129;`
`let var3 = -6_835_292i64;`	`int64_t var3 = -6835292;`
`let var4 = 0b0011_0101u8;`	`uint8_t var4 = 0b00110101;`
`let var5 = 0o073452u16;`	`uint16_t var5 = 0073452;`
`let var6 = 103_937_465u32;`	`uint32_t var6 = 103937465;`
`let var7 = 0xAAFF_7625u64;`	`uint64_t var7 = 0xAAFF7625;`
`let var8 = 1_342.56f32;`	`float var8 = 1342.56;`
`let var9 = -34.035_440f64;`	`long double var9 = -34.035440;`
`let var10 = -45isize;`	`int16_t var10 = -45;`
`let var11 = 9731usize;`	`uint16_t var11 = 9731;`

One of the most useful features of modern languages like *Rust* is the type inference, which can simplify the programming process in most cases. Type inference makes the programmer unworried about variable types when not necessary, and then makes the development time shorter. The type inference is also implemented in *Sokae* by using default types for integer and floating points variables. In the case of the XC16 compiler, the `uint16_t` and `float` type were defined as the default type for the inference. Table 5 shows the equivalent transpiled code for different variable declarations by using inference, including boolean, integer, floating point and character literals.

Table 5: Primitive types declaration by inference

Sokae	*C (XC16)*
`let var0 = true;`	`bool var0 = true;`
`let var1 = false;`	`bool var1 = false;`
`let var2 = 1345;`	`int16_t var2 = 1345;`
`let var3 = 71.4;`	`float var3 = 71.4;`
`let var4 = -457;`	`int16_t var4 = -457;`
`let var5 = -10.445;`	`float var5 = -10.445;`
`let var6 = 'd';`	`char var6 = 'd';`

On the other hand, *Sokae* gives support for some of the loop instructions of *Rust*, such as: regular **`while`** loop, infinity **`loop`**, and regular **`for`** loop. Table 6 shows the transpiler results for the supported loop instructions in Sokae. The **`for`** loop includes the integer range notation using the syntaxes: `i..f` and `i..=f`, being `i` the initial value and `f` the final value.

Table 6: Supported loops

Sokae	*C (XC16)*

`while a < 10 {`
`  a += 1`
`}`
	`while(a < 10){`
	`  a += 1;`
	`}`

`loop {`
`  a += 1`
`}`
	`while(true){`
	`  a += 1;`
	`}`

`for i in 0..10 {`
`  arr[i] = 0`
`}`
	`for(int i=0, i<10, i++){`
	`  arr[i] = 0;`
	`}`

`for i in 0..=9 {`
`  arr[i] = 0`
`}`
	`for(int i=0, i<=9, i++) {`
	`  arr[i] = 0;`
	`}`

4. CONCLUSIONS

The proposed programming structure allows the microcontroller programmer to use a modern high level programming language, with the advantage of being compiled but with some modern features. *Sokae* language pretends to be a high level programming language for microcontroller, that uses modern Rust-based features like type inference but at the same time obtaining binary files with similar optimization levels of standard compiled languages like *C*.

The amount of memory necessary to run a program written in an interpreted language such as *Micropython* or *Javascript* on microcontrollers, limits the range of these in which it is possible to do it. *Sokae* language along with the proposed programming structure could allow programming any microcontroller which has a native *C* compiler, taking advantage of its modern features. At the same time, *Sokae* could reach almost the same execution times of *C* language.

The transpiling process between *Sokae* and *C* is successful due to both languages are very similar. As all variables are mutable by default in *Sokae*, that language is closer to *C* even than *Rust* which is its inspiration.

Like *Arduino*, *mbed* and *Micropython* among others, *Sokae* language and programming structure could allow people with minimal electronics knowledge to program easily embedded systems. Likewise, *Sokae* could allow experienced embedded systems programmers to program a wide range of them learning only one language and API.

This programming proposal is perfectly functional as was demonstrated by tests done, even without implementing most of the modern features of *Rust*, which means that this project can significantly improve implementing more of them.

By using the proposed API implementation, it is possible to use *Sokae* and this programming structure in the classroom in basic courses of microcontrollers and embedded systems. The learning curve of *Rust* and therefore *Sokae* is not that fast, this fact gives the opportunity to use as base another simpler language to improve the general learning curve of the proposed programming structure, maintaining the same API. Two language candidates are the *V* programming language and *Peregrine* language; the first one is inspired mainly by *Rust* and the second one by *Python*.

As future work, the implementation of other *Rust* useful features is proposed. For instance, the array definition, the direct array indexing using the **`for`** loop, variable shadowing among others. Likewise, it is imperative to give support to other microcontrollers especially the ones with low program and data memory, which this project could show their advantages.

Finally, implementing PC-based graphical user interfaces directly in *Rust* for applications in *Sokae* could be advantageous, due to both languages sharing the same base, making it easy to develop complete applications like SCADA or other control and GUI applications.

5. ACKNOWLEDGEMENT

This work was supported by Universidad Distrital Francisco José de Caldas and Corporación Unificada Nacional de Educación Superior CUN. The views expressed in this paper are not necessarily endorsed by Universidad Distrital or CUN. The authors thank ARMOS and IDECUN research groups for the simulations and tests done.

REFERENCES

[1] A. Radovici and I. Culic, Embedded Systems Software Development. Berkeley, CA: Apress, 2022, pp. 27–47.

[2] E. Kusmenko, B. Rumpe, S. Schneiders, and M. von Wenckstern, "Highly-optimizing and multi-target compiler for embedded system models: C++ compiler toolchain for the component and connector language embeddedmontiarc," in Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, ser. MODELS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 447–457. [Online]. Available: https://doi.org/10.1145/3239372.3239388

[3] A. K. Rachioti, D. E. Bolanakis, and E. Glavas, "Teaching strategies for the development of adaptable (compiler, vendor/processoríndependent) embedded c code," in 2016 15th International Conference on Informa- tion Technology Based Higher Education and Training (ITHET), 2016, pp. 1–7.

[4] K. Zandberg and E. Baccelli, "Minimal virtual machines on iot microcontrollers: The case of berkeley packet filters with rbpf," in 2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN). IEEE, 2020, pp. 1–6.

[5] K. Grunert, "Overview of javascript engines for resource-constrained microcontrollers," in 2020 5th International Conference on Smart and Sustainable Technologies (SpliTech), 2020, pp. 1–7.

[6] D. E. Bolanakis, "A survey of research in microcontroller education," IEEE Revista Iberoamericana de Tecnologias del Aprendizaje, vol. 14, no. 2, pp. 50–57, 2019.

[7] S.-M. Kim, Y. Choi, and J. Suh, "Applications of the open-source hardware arduino platform in the mining industry: A review," Applied Sciences, vol. 10, no. 14, p. 5018, 2020.

[8] V. M. Ionescu and F. M. Enescu, "Investigating the performance of micropython and c on esp32 and stm32 microcontrollers," in 2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME), 2020, pp. 234–237.

[9] A. Suarez Ruiz, "Diseño de hardware y firmware para un sistema inal´ambrico de adquisición de datos daq de bajo costo," Departamento de Ingeniería El´ectrica, Electrónica y Computación, 2019.

[10] H. Wu, C. Chen, and K. Weng, "An energy-efficient strategy for microcontrollers," Applied Sciences, vol. 11, no. 6, p. 2581, 2021.

[11] A. M. Karpi´nski, "Automatic translation of programs source codes from python to c# programming language," Ph.D. dissertation, Zakład Sztucznej Inteligencji i Metod Obliczeniowych, 2022.

[12] M. Szafraniec, B. Roziere, H. Leather, F. Charton, P. Labatut, and G. Synnaeve, "Code translation with compiler representations," 2022. [Online]. Available: https://arxiv.org/abs/2207.03578

[13] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, "In rust we trust – a transpiler from unsafe c to safer rust," in 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2022, pp. 354–355.

[14] H. Lunnikivi, K. Jylkk¨a, and T. H¨am¨al¨ainen, "Transpiling python to rust for ptimized performance," in Embedded Computer Systems: Architectures, Modeling, and Simulation, A. Orailoglu, M. Jung, and M. Reichenbach, Eds. Cham: Springerínternational Publishing, 2020, pp. 127–138.

[15] M. Marcelino and A. M. Leit˜ao, "Extending PyJL - Transpiling Python Libraries to Julia," in 11th Symposium on Languages, Applications and Technologies (SLATE 2022), ser. Open Access Series in Informatics (OASIcs), J. a. Cordeiro, M. J. a. Pereira, N. F. Rodrigues, and S. a. Pais, Eds., vol. 104. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum f¨urínformatik, 2022, pp. 6:1–6:14. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/16752

[16] B. F. Andr´es and M. P´erez, "Transpiler-based architecture for multi-platform web applications," in 2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM), 2017, pp. 1–6.

[17] K. Takano, T. Oda, and M. Kohata, "Approach of a coding conventions for warning and suggestion in transpiler for rust convert to rtl," in 2020 IEEE 9th Global Conference on Consumer Electronics (GCCE), 2020, pp. 789–790.

[18] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," arXiv preprint arXiv:2206.05503, 2022.

[19] T. Uzlu and E. S¸aykol, "On utilizing rust programming language forínternet of things," in 2017 9th International Conference on Compu- tational Intelligence and Communication Networks (CICN), 2017, pp. 93–96.

[20] K. I. Vishnunaryan and G. Banda, "Harsark multi rs: A hard real- time kernel for multi-core microcontrollers in rust language," in Smart Intelligent Computing and Applications, Volume 2, S. C. Satapathy, V. Bhateja, M. N. Favorskaya, and T. Adilakshmi, Eds. Singapore: Springer Nature Singapore, 2022, pp. 21–32.

[21] J. Aparicio Rivera, "Real time rust on multi-core microcontrollers," Master's thesis, Lule˚a University of Technology, Computer Science, 2020.

[22] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020, pp. 763–779.

[23] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: rust as a case study," in Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021), 2021, pp. 597–616.

[24] N. Lagaillardie, R. Neykova, and N. Yoshida, "Implementing multiparty session types in rust," in International Conference on Coordination Languages and Models. Springer, 2020, pp. 127–136.

[25] Rust programming language development community. (2022 September 2). "The Rust Reference" [Online]. Available: https://doc.rust-lang.org/reference/introduction.html.