Microcontrollers Programming Framework based on a V-like Programming Language

Fernando Martínez Santa[1], Santiago Orjuela Rivera[2], Fredy H. Martínez Sarmiento[3]
Universidad Distrital, Francisco José de Caldas, Bogotá, Colombia[1,3]
Corporación Nacional Unificada, de Educación Superior CUN, Bogotá, Colombia[2]

***Abstract*—This paper describes the design of a programming framework for microcontrollers specially the ones with low program and data memory, using as a base a programming language with modern features. The proposed programming framework is named *Aixt Project* and took inspiration from other similar projects such as *Arduino*, *Micropython* and *TinyGo* among others. The project's name is inspired on the weasel pet of the *V* programming language and at the same time it is a tribute to *Ticuna* people who live in the Amazon rain-forest, just between Colombia, Perú and Brasil. *Aixt* comes from *Aixtü* or *Aitü rü* which means otter in *Ticuna* language. The proposed programming framework has three main components: the *Aixt* language based on the *V* syntax, a transpiler that turns the defined V-like source code into *C*, and a generic cross-platform Application Programming Interface (API). The target of this project is obtaining a cross-platform programming framework over the same language modern language an the same API, for programming different microcontrollers especially the ones with low memory resources. *Aixt* language is based on the syntax of *V* programming language but it uses mutable variables by default. *V* language was selected to be used as base of this project due to it is a new compiled programming language with interesting modern features. In order to turn the *Aixt* source code into *C*, a transpiler is implemented using *Python* and the some specialized libraries to design each part of its translation process. The transpiled code is compiled by the native *C* compiler of each microcontroller to obtain the final binary file, that is why the API has to be adapted for each native *C* compiler. The complete project is released as a free and open source project. Finally, different application test were done over the XC8 and XC16 compilers for the PIC16, PIC18, PIC24 and dsPIC33 microcontrollers families, demonstrating the correct working of the overall framework. Those tests show that the use modern language framework to program any microcontrollers is perfectly feasible using the proposed programming framework.**

***Keywords*—Microcontroller; transpiler; API; programming language; V; V-lang; Aixt project**

I. Introduction

The different processor architectures used by the commercial microcontrollers, make the programming process dependent on those architectures and thus not universal. Even, when the microcontrollers are programmed on high level languages, tasks such as peripherals, timers, setup registers, and others, keep depending on the programmer's knowledge of the processor's architecture [1], [2]. There are some different projects which pretends to generate cross-platform programming frameworks [3], using different programming languages like *JavaScript* [4], and other implementations using virtual machines [5], [6], [7]. An example of those programming frameworks (and one of the most popular) is *Arduino*[8], [9], [10], which is based on *C* language in addition to an API which makes the programming process easier. That API works on a predefined hardware setup to reduce the setup process by the programmer. Another popular programming framework for microcontrollers is *Micropython* which implements on several devices a subset of Python language. *Micropython* has specific relatively high memory requirements which makes it impossible to run on small microcontrollers, but it has been ported to a large number of different architectures [11] mainly in internet of things IoT implementations. *Arduino* is compiled but its *C* syntax lacks modern features, on the other hand *Micropython* is interpreted and therefore non time optimized as compiled language, but there is an intermediate framework named *Tinygo* which implements *Go* language on Microcontrollers, offering modern features like *Python* and the advantage of being compiled [12] like *Arduino* (*C*). However, most of the microcontroller with limited memory features does not fit to the memory requirements of the projects previously described, so for those ones it is necessary to use their native *C* compiler.

In order to obtain the best execution times and the best code optimization level [13], [3] it is necessary to use the native *C* compiler of each architecture. Then, if there is a programming framework with an upper modern language layer, a transpiler to *C* and the native *C* compiler as a part of the framework, this could have high level language features along with optimization levels similar to the ones reached with only the native compilers. The described programming framework needs to have a transpiler [14], which is a translator from the upper layer language to the native *C* [15]. Transpilers are highly utilized nowadays [16], [17], in several languages both compiled and interpreted [18], [19], [20], and even in languages based on virtual machines [21]. Those transpilers are mainly used in order to reuse source code that comes from another different language [18], or improve the execution times or another performance feature of the program [22], [23] changing the platform or language (for instance turn *Python* (interpreted) into *Rust* (compiled) [19]), even translating source code to gate-based hardware [24] like FPGAs or other processor-less devices.

Several new programming languages have emerged nowadays, mainly to solve some of the issues of the traditional ones such as safety, memory management among others. Among these new languages are *Go*, *Swift*, *Dart*, *F#* and *Rust*, being this last is one of the most preferred ones[25], having even implementations on microcontrollers [26], [27], [28]. There are some other other languages such as *Peregrine* which is based on the *Python's* syntax and the *V* programming language [29]

wish is inspired on *Rust* and other languages. *V* is an statically-typed programming language with several modern features that make the development easy, and a better learning curve than other modern languages like *Rust*.

This paper proposed a programming framework for microcontrollers that is composed by a high level language based on *V* as the main language, a transpiler from this *V-like* language named *Aixt* to *C*, and the microcontrollers' native *C* compiler which finally generates output binary file. In order to generalized the programs across the different microcontrollers, a general API is designed, which is implemented on each *C* compiler of the supported devices (in this first stage for XC8 and XC16 compilers). For the transpiler implementation *Python* and the module *SLY* were used, to write the lexer analyzer and the Parser. This project is based on a previous one named *Sokae* [30] developed by the same authors.

The paper is organized as follows: Section **II** presents the methodology for implementing the overall proposed programming framework, including the *Aixt* language definition (Section **II** - *A*), the *Python* implementation of the Aixt-to-C transpiler (Section **II** - *B*), and the API implementation for the XC16 compiler and PIC24 microcontrollers family (Section **II** - *C*). Section **III** shows the *Aixt* language functionality by implementing several examples, as well as it presents the results of implementing the proposed programming framework by several test source codes. Finally, Section **IV** shows the conclusions about this research's main ideas, including possible future jobs.

II. Methodology

With the name *Aixt Project*, a microcontroller programming framework is implemented. This framework uses an homonym language which is based on the *V* programming language. A transpiler from *Aixt* language to *C* is the most important block of this framework, as well as an Application Programming Interface (API) written in both languages. As part of the proposed structure, the native *C* compiler of the specific microcontroller finally generates the output binary file, as shown in Fig. 1. Using the proposed framework, the users will be able to write the source code in *Aixt* language using a standard API and obtains the binary file for a specific microcontroller or board without having further knowledge of the programming architecture. This framework pretends to be highly modular and relatively easy to include other microcontrollers or boards. The Fig. 1 shows the general structure of the programming framework indicating that for each new microcontroller to be supported it is necessary to adapt the API (Fig. 1 right) to this and invoke its specific native *C* compiler (Fig. 1 left down). The specific test done for this paper were implemented on some different Microchip® microcontroller families such as PIC16, PIC18, PIC24 and dsPIC33 using the XC8 and XC16 compilers, these microcontrollers were selected because their limited amount of implemented memory.

A. Aixt Language

Aixt is the name given to the proposed language and the overall programming framework. This language is based on the *V* programming language [29] and shares most of its syntax. Due to its relatively short learning curve, *V* language was

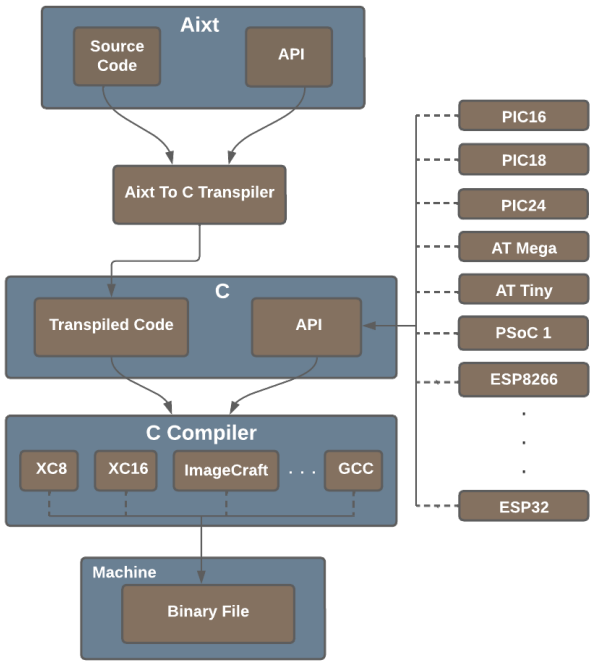

Fig. 1. General programming framework diagram

selected for this implementation instead other new languages like *Rust* [30]. The framework and language name is inspired in the Weasel pet of *V* Language, and at the same time is a tribute to *Ticuna* people who live in the Amazon rain-forest in the borders between Colombia, Brasil and Perú. Weasels are mustelids just like otters, so the name *Aixt* comes from *Aixtü* or *Aitü rü* which is a way to say otter in *Ticuna* language.

Aixt is a compiled and statically typed programming language based on the *V* syntax. This is designed to be used on a wide range of microcontrollers no matter their memory limitations. *Aixt* syntax shares some feature with languages such as *Rust* and *Go*, therefore also it shares syntax features with *C*, which makes *Aixt* easy to understand and transpile.

Listing 1 shows an example code using *Aixt* language and API, which makes blinking a LED for a specific microcontroller's pin. Likewise, the Listing 2 shows the *C* equivalent of the same *Aixt* source code.

Some of the basic features of *Aixt* language are listed as follows:

- the `:=` operator is used for declaring variables.
- Unlike *V*, variables are mutable by default in *Aixt*.
- `ix`, `ux` and `fx` variable types for regular integers, unsigned integers and floating point variables.
- `isize` and `usize` for integers with same size of the processor.
- `rune` type for character variables.
- Default type inference in declaring.
- Underscore character in literals for improving its readability.

- The `main` function is the first entry of a *V* program. In case of having only one source code, the main function definition can be ommited.
- All instructions end with a new line character, whit a semicolon or with a curly bracket close.
- The semicolon is optional. It has to be used when having two simple instructions in the same code line.
- All the code blocks are delimited by curly braces.
- All function declarations start with the reserved word `fn`.
- The names for all the identifiers (variables, constants, functions, etc.) prefer to use snake case as in *V*, for instance the function `pin_low()`. This feature is implemented in order to keep a standard format for all the *V* source code.
- There is only a loop instruction which is used for implementing all the supported loops, changing only its input parameters syntax.
- The reserved word `import` is used for including different complete modules or libraries.
- In order to reduce the *C* obtained code, it is possible to include individual components from a global module using the curly braces following the syntax: `import module { comp1, comp2, ...}`

Listing 1: Blinking LED example in *Aixt*

```
import machine { pin }
import time { sleep_ms }

pin(A6,OUT)
for {
    pin_high(A6)
    sleep_ms(500)
    pin_low(A6)
    sleep_ms(500)
}
```

Listing 2: Resultant *C* code for the Blinking LED example

```
#include "./settings.h"
#include "./machine/pin.h"
#include "./time/sleep_ms.h"

int main(void) {
    pin(A6,OUT);
    while(true) {
        pin_high(A6);
        sleep_ms(500);
        pin_low(A6);
        sleep_ms(500);
    }
    return 0;
}
```

B. Transpiler

A transpiler is a program that translates source code between programming languages with the same abstraction levels, by contrast a compiler translate source code generally to another low level language. The proposed programming framework does not compile directly the *Aixt* source code but transpile it to *C*. The Transpiler from *Aixt* language to *C* is implemented with *Python* and using the PLY module in order to implement the lexer analyzer and parser for the input source code. The complete working diagram of the implemented transpiler is shown in Fig. 2, where and input file with `.v` extension get in to the transpiler and it generates the output `.c` file. The transpiler implementation is based on part of the *V* language grammar, the Listing 3 shows and extract of that grammar in Backus-Naur form (BNF). This part of the grammar shows the definition of the four different ways to do loops in *Aixt* using the reserved word `for`, including infinite loops.

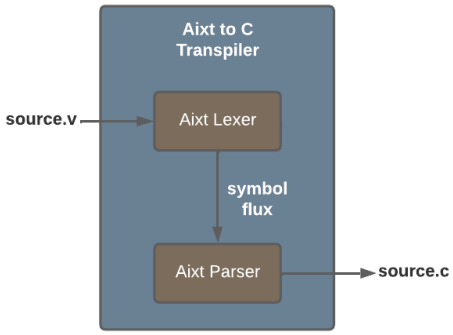

Fig. 2. Transpiler diagram

Listing 3: *Aixt* language BNF definition (extract)

```
...
forStmt ::= for block
          | for expr block
          | for forClause block
          | for inClause block

forClause ::= simpStmt ; expr ; simpStmt

inClause ::= exprList in IDENTIFIER
...
```

For the implementation of the lexer analyzer, all of the tokens of *V* language are supported, such as keywords, operands and other punctuation symbols, as shown in the code extract of Listing 4.

Listing 4: *Aixt* Lexer implementation (extract)

```
...
tokens = {
    I8, I16, I32, I64, ISIZE,
    ...
    F32, F64, BOOL, RUNE,
    IMPORT, IN, MAP, MATCH, RETURN,
```

```
    ...
}

BOOL        = r'bool'    # Types
RUNE        = r'rune'
...
IDENTIFIER = r'[a-zA-Z_][a-zA-Z0-9_]*'
...
BINARY_LIT = r'0b[01_]+'
...
literals = { '(', ')', '{', '}', '[',
             ']', ';', ',', '.',
           }
...
```

Once the Lexer analyzer reduced the character flux of the source code to an token flux, the parser analyzes the syntactic rules of language in order to find possible syntactic error and transpile it to *C*. The most of the syntactic rules of *V* are implemented in *Aixt* using the SLY module as shown in the extract source code of Listing 5, which matches with the BNF definition shown in Listing 6.

Listing 5: *Aixt* Parser implementation (extract)

```
...
@_( 'identList␣DECL_ASGN␣exprList',
    )
def varDecl(self, p):
...
    return ret_value

@_( 'IDENTIFIER',
    'identList␣","␣IDENTIFIER'
    )
def identList(self, p):
...
    return p[0]
...
```

Listing 6: *Aixt* BNF rules (extract)

```
varDecl ::= identList DECL_ASGN exprList

identList := IDENTIFIER
           | identList "," IDENTIFIER
```

SLY library uses *Python's* function decorators to implement the syntactic rules of the language to be compiled or transpiled, applying them to each syntactic production, for example the production varDecl is the implementation of variable declarations in *Aixt* language.

As previously said, the transpiler reads the source code written in *Aixt*, which for compatibility with standard source code editors, the `.v` file extension.

C. Application Programming Interface

One of the main goals of the proposed framework is designing a cross-platform API, which includes the basic features and peripherals of most microcontrollers. In order to make the microcontroller's programming process easier, a general Application Programming Interface is implemented in both the *Aixt* programming language and *C* for the specific native compiler. This API includes the peripherals and features shown in Tables from I to IV.

TABLE I. GENERAL PURPOSE INPUT/OUTPUT

Description	Function name
pin type declaration	`pin()`
setting high and low	`pin_low() pin_high()`
setting specific binary value	`pin_value()`
reading an input value	`pin_value()`

TABLE II. ANALOG TO DIGITAL CONVERTER (ADC)

Description	Function name
ADC setting up	`adc()`
ADC reading value	`adc_read()`

TABLE III. UNIVERSAL ASYNCHRONOUS RECEIVER TRANSMITTER (UART)

Description	Function name
UART setting up	`uartx()`
single byte transmitting	`uartx_put()`
single byte receiving	`uartx_get()`

TABLE IV. TIMING FEATURES

Description	Function name
delays in microseconds	`sleep_us()`
delays in millisecond	`sleep_ms()`
delays in second	`sleep()`

Table I shows the pin and GPIO functions like setup, input capture and output set. Some devices even could support exchange state functions (`pin_toggle`). The rest of API functions follow the same rules:

- The setup function has the same name of the module.
- The rest of name functions of the same module follow the syntax: `module_function()`. For instance: `adc_read()` function of `machine { adc }` module (Table II).
- Devices with more than one peripheral of the same time follow this name function syntax: `modulex_function()` where the `x` refers to the number that identify each peripheral. For instance: `uart2_get()` as shown in Table III.
- Some API modules refers to a inner features of the device different to hardware peripherals, for instance software delays (Table IV).

The Fig. 3 shows the folder structure designed for the overall API, this structure has to be followed for each of the supported microcontrollers and boards to maintain the compatibility across all the hardware devices. Following strictly this folder structure allow the transpiler to correctly redirect the module including tasks when it is necessary to include to the project isolated components of a module.

As previously mentioned, the module including follows the next syntax in *Aixt*: `import module`

for complete modules, which will be transpiled as `#include \./module.h"`. Likewise, the sub-modules or module components including follows the syntax: `import module { sub1, sub2, ...}`, which will be transpiled to `#include \./module/sub1.h"` etc. That is very important to optimize the resultant binary file. On the other hand, when a complete module is included, the `./module.h` header file has to include all of the `.h` files in the correspondent folder on the folder's API structure.

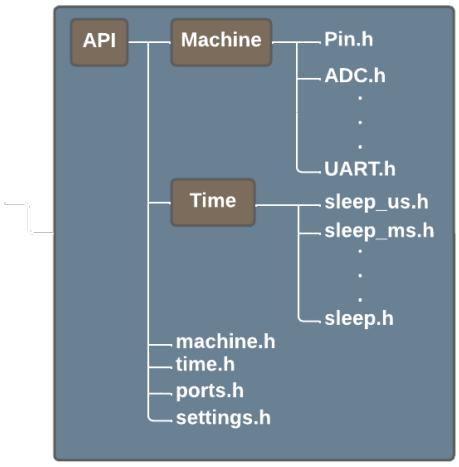

Fig. 3. General API structure

III. Results

The overall project including the *Aixt* language definition, the transpiler from *Aixt* to *C* and the API, is published by the authors as a free software project at the URL https://gitlab.com/fermarsan/aixt-project. The authors hope this project works as a starting point of a great free programming framework for microcontrollers or as seed for other similar projects.

The complete programming framework was successfully tested using some of the 8-bit and 16-bit PIC microcontroller families from Microchip ®. Those devices were selected due to their low amount of implemented data and program memories.

Several different working tests have been performed to check the correctness of most *Aixt* features. Listings 7 and 8 show a comparison of the variable declarations in *Aixt* and the corresponding transpiled *C* code for XC8 and XC16 compilers. In *Aixt* the variable declaration is always along with an assignment. The declaration and assignment process uses the operator `:=` to differentiate with only assignment `=`. At the same time it is necessary to use the conversion predefined functions such as `i8()`, `u32()` and `f64()` among others, in order to specify the number of bits and the type of integer and floating point variables. One of the benefits of using the conversion functions of *V* for the variable definitions is that each variable is bit-width explicit, independent of hardware device. Listing 7 shows too the use of the underscore symbol `"_"` for improving the large numbers readability. Also the special notations for hexadecimal, octal, and binary literals, are shown. The only difference with *C* is the octal literals beginning with the sequence `"0o"` (zero + o), instead of only `0` as in *C*.

Listing 7: *Aixt* variable declaring and assignment example.

```
var2 := i8(129)
var3 := i64(-6_835_292)
var4 := u8(0b0011_0101)
var5 := u16(0o073452)
var7 := u64(0xAAFF_7625)
var8 := f32(1_342.56)
var9 := f64(-34.035_440)
```

Listing 8: *C* resultant variable declaring and assignment example.

```
int8_t var2 = 129;
int64_t var3 = -6835292;
uint8_t var4 = 0b00110101;
uint16_t var5 = 0073452;
uint64_t var7 = 0xAAFF7625;
float var8 = 1342.56;
long double var9 = -34.035440;
```

Modern programming languages like *V* has some useful features such as the type inference, which simplifies programming in most cases. Type inference gives programmers peace of mind about variable types when they are not needed, thereby reducing development time. The implementation of this feature in *Aixt* is reached by using the standard types for integer and floating point variables. In the case of XC8 compiler the standard integer type is `int8_t` and for XC16 compiler `int16_t`. For the floating point variables the default type is `float`. Listings 9 and 10 show the transpiling result for some variable declarations by inference, including Boolean, character (named runes), integer and floating point literals, for the XC8 compiler.

Listing 9: *Aixt* variable declaring and assignment by inference example.

```
var0 := true
var1 := false
var2 := 1345
var3 := 71.4
var4 := -457
var5 := -10.445
var6 := 'd'
```

Listing 10: *C* resultant variable declaring and assignment by inference example.

```
bool var0 = true;
bool var1 = false;
int8_t var2 = 1345;
float var3 = 71.4;
int8_t var4 = -457;
float var5 = -10.445;
char var6 = 'd';
```

On the other hand, *Aixt* language syntax provides support for some of *V's* looping statements, such as: condition for (`while` in *C*), bare for or infinite loop (`while(true)` in *C*), infinite loops, regular for loop and C-like for loop. Listing 10 shows an example of the loop statements currently supported by the *Aixt* syntax and Listing 11 shows the *C* equivalent of each one. The Aixt-like for loop includes and integer range notation with the syntax: `i..f`, where `i` is the initial value and `f` is the final value.

Listing 11: *Aixt* available loops.

```
// condition for
for a < 10 {
    a += 1
}
// bare for
for {
    a += 1
}
//range for
for i in 0..10 {
    arr[i] = 0
}
//c for
for i:=0; i<=10; i++ {
    arr[i] = 0
}
```

Listing 12: *C* equivalent loops.

```
while(a < 10){
    a += 1;
}
while(true){
    a += 1;
}
for(int i=0, i<10, i++){
    arr[i] = 0;
}
for(int i=0, i<=10, i++){
    arr[i] = 0;
}
```

A. Microcontrollers Setting Up

In order to setup a specific new microcontroller or board added to the *Aixt* programming framework, a configuration file has to be written. The chosen format for this configuration file is *YAML* which means Yet Another Markup Language, and is a very simple format to implement setup file for software projects. In that configuration file the designer can setup features such as: type equivalences between *Aixt* and the native *C* compiler, the microcontroller fuses or configuration bytes, the part or device number, the default header files among others. This configuration file is expected to be modified once by the designer and not to be modified by a regular user. The Listing 12 shows an extract of the configuration file for a PIC24FJ device.

Listing 13: *YAML* microcontroller or board configuration file (extract).

```
i8:       int8_t
...
u16:      uint16_t
...
default_int:      int16_t
...
device:    p24FJ128GA010
...
headers:
  - <xc.h>
  - <stdint.h>
...
configuration:
  - "POSCMOD = XT"
  - "OSCIOFNC = ON"
...
```

On the other hand, a batch file has to be included for each new device. This file works as a Makefile, following the steps and invoking the different component of the framework, in order to obtain the final binary file starting from the *Aixt* source code. The batch file has to be provided in `.ps1` (PowerShell) format for Windows and in `.sh` format for Linux.

IV. Conclusion

Using the proposed programming framework, the microcontrollers programmer can utilize a modern high level language programming environment, using a compiled language with its benefits and at the same time taking advantage of the modern features of the language. *Aixt* Language pretends to be a highly level programming language for microcontrollers with a short learning curve due to its simplicity compared with other modern languages. *Aixt* utilizes modern *V*-based features such as type inference but at the exact same time obtains binary files with similar optimization degrees of standard compiled languages such as *C* and similar execution times. *Aixt* Language and the proposed programming framework could enable programming microcontrollers with ease, as long as they have a native *C* compilers. At the same time *Aixt* does not need a fixed amount of memory to work, the finally binary file depends only the source code. So it has not the problem of the memory needed to run a program written with an interpreted language such as *MicroPython* or *Javascript*.

The transcompilation process between *Aixt* and *C* is successful because both languages are similar, mainly due to all variables in *Aixt* are mutable by default like in *C*, and some other similar features like curly braces and others. Transpile another language such *Python* to *C* for instance, would be a little bit difficult because the differences between both languages.

Aixt Language and programming framework could allow individuals with little electronics know how to program easily embedded systems, just like *Arduino*, *mbed*, and *MicroPython* among other frameworks. Likewise, *Aixt* could enable experienced embedded system programmers only learning one programming language and API, to program a wide variety of microcontrollers no matter their memory sizes.

All of the features in the proposed programming framework was completely tested, however not all the modern features of

the *V* programming language were implemented. That means this project can highly improve implementing more features and adapting it to other microcontrollers and boards. It is perfectly able to use *Aixt* language and this framework in the classroom in Basic courses of microcontroller and embedded systems, due to currently this project is highly functional.

In spite of the short learning curve of *V* and therefore *Aixt* languages, it is possible to explore another simple languages to improve the proposed programming scheme, or even giving support to another main languages maintaining the same API. One of the candidates is the *Peregrine* Language who is based on the *Python* syntax.

As future work, the development of other useful features of *V* language are proposed. For example, the array definition, direct array indexing using the array for loop, array interpolation, matching statements among others. Likewise, it is important to keep giving support to other MCUs and board especially those with low program and data memories, which are the motivation for this project. For instance Atmel ® AT mega and AT tiny will be included to the project soon due to they use also the XC8 compiler. Finally, it is possible to combine PC graphical application developed in *V* with embedded application developed in *Aixt*, taking the advantage of learning only one programming basis to develop a complete embedded-based graphical application.

ACKNOWLEDGMENT

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

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/processor independent) embedded c code," in *2016 15th International Conference on Information Technology Based Higher Education and Training (ITHET)*, 2016, pp. 1–7.

[4] 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.

[5] 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.

[6] S. Varoumas, B. Pesin, B. Vaugon, and E. Chailloux, "Programming microcontrollers through high-level abstractions," in *Proceedings of the 12th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, 2020, pp. 5–14.

[7] R. Gurdeep Singh and C. Scholliers, "Warduino: a dynamic webassembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2019, pp. 27–36.

[8] 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.

[9] 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.

[10] H. K. Kondaveeti, N. K. Kumaravelu, S. D. Vanambathina, S. E. Mathe, and S. Vappangi, "A systematic literature review on prototyping with arduino: Applications, challenges, advantages, and limitations," *Computer Science Review*, vol. 40, p. 100364, 2021.

[11] 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.

[12] A. Suarez Ruiz, "Diseño de hardware y firmware para un sistema inalámbrico de adquisición de datos daq de bajo costo," *Departamento de Ingeniería Eléctrica, Electrónica y Computación*, 2019.

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

[14] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

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

[16] 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

[17] F. A. Bastidas and M. Pérez, "A systematic review on transpiler usage for transaction-oriented applications," in *2018 IEEE Third Ecuador Technical Chapters Meeting (ETCM)*, 2018, pp. 1–6.

[18] 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.

[19] H. Lunnikivi, K. Jylkkä, and T. Hämäläinen, "Transpiling python to rust for optimized performance," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, A. Orailoglu, M. Jung, and M. Reichenbach, Eds. Cham: Springer International Publishing, 2020, pp. 127–138.

[20] M. Marcelino and A. M. Leitão, "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 für Informatik, 2022, pp. 6:1–6:14. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/16752

[21] B. F. Andrés and M. Pérez, "Transpiler-based architecture for multiplatform web applications," in *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, 2017, pp. 1–6.

[22] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 662–676. [Online]. Available: https://doi.org/10.1145/3062341.3062381

[23] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021.

[24] 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.

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

[26] T. Uzlu and E. Şaykol, "On utilizing rust programming language for internet of things," in *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*, 2017, pp. 93–96.

[27] 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.

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

[29] N. P. Kumar Rao, *Getting Started with V Programming*. Packt Publishing, 2021. [Online]. Available: https://www.packtpub.com/product/getting-started-with-v-programming/9781839213434

[30] F. Martínez Santa, S. Orjuela Rivera, and F. H. Martínez Sarmiento, "Rust-like programming language for low-resource microcontrollers," *Advances in Dynamical Systems and Applications*, 2022.