ForEmb: Un Intérprete en Tiempo-Real para Sistemas Embebidos Inspirado en Forth

ForEmb: A Forth-Inspired, Real-Time Interpreter for Embedded Systems

XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX

XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX
XXXXXXXXXXXXX

Resumen — **Se implementó un intérprete en tiempo-real, el cual se portó a varios sistemas, incluyendo la tarjeta de desarrollo Raspberry Pi Pico. Además, se simuló una señal de información, que utiliza las funciones trigonométricas seno y coseno, así como la función exponencial en ForEmb, donde se obtuvo un error absoluto porcentual de 0.18% respecto a la respuesta obtenida de Matlab, con una varianza de 1.06×10−8. Por otro lado, se implementaron las señales de los sensores CMP y CKP para un motor de Neon 2.0L en ForEmb, permitiendo su emulación sobre el microcontrolador RP2040, donde se realizaron mediciones experimentales que validaron el funcionamiento en tiempo-real.**

Palabras Clave - Intérprete; Tiempo-Real; Sistema Embebido.

Abstract — **ForEmb, a real-time interpreter, was implemented and ported to various systems, including a Raspberry Pi Pico board. ForEmb successfully simulated an information signal -using sine, cosine and exponential functions- with an absolute error percentage of 0.18% compared to the response of Matlab and a variance of 1.06×10−8. The CMP and CKP sensor signals for Neon 2.0 L engine were also emulated by implementing them into ForEmb running on a RP2040 microcontroller; experimental measurements validate real-time implementation.**

Keywords - Interpreter; real-time; embeded system.

I. INTRODUCTION

Society is immersed in a variety of technologies that facilitate daily activities -such as the everyday, like cell phones, to the more specialized, like bio-metrically access-controlled authentication systems, and service systems. It is, therefore, essential to point out that all these technologies use embedded systems to operate optimally. Embedded systems are dedicated to specific tasks [1]–[3].

Microcontrollers/microprocessors are common components inside embedded systems that allow the implementation of digital algorithms to solve specific tasks [1], [4], [5]. In fact, there are different schemes aimed at improving microcontroller performance by enhancing software [6]–[8] and even hardware architectures [9], [10].

Embedded software is an essential component of an embedded system [11]. Algorithm implementation relies heavily on the resources and software design of the embedded system. It takes full advantage of the hardware to implement its best version. Moreover, programmers must adhere to the syntactical rules of the language to implement the designed algorithm in an embedded system [12]. Because, challenges and limitations can arise from the programming language and increase design time, more flexibility in the language can thereby reduce design time. Hence, Forth emerge as the subject of study for this investigation.

Forth is a simple, but Turing-complete programming language that uses data stack and RPN notation. Everything in Forth is a *word* -that is, characters which are not spaces- and every word is defined inside a dictionary and does something useful. Words can a) accept parameters passed on the data stack, b) execute programmed operations by using other word definitions, and c) return the answer back on the data stack. It is this simplicity and extendibility that attracts many designers, to use Forth in developing embedded systems [6], [7], [13].

Real-time embedded systems need to execute required operations and guarantee response times within a specified

period [14]. For example, synchronization and communication between a fire sensor, microcontroller, and alerting system need to be performed in fractions of the time to warn people when fire-related emergencies are detected [5].

Overall, Forth is a powerful and flexible programming language that is particularly well-suited to low-level embedded systems programming and other limited resource applications, i.e., real-time. While it may take some time to learn, Forth's unique approach to programming can make it a valuable tool in the hands of skilled developers.

In this work, Forth philosophy has been chosen to design a real-time interpreter for embedded systems due to its flexibility allowing for the efficient implementation of algorithms. Programs were written as a series of words -primitives and user defined- each of which performs a specific task or operation. These words are stored in dictionaries that can be combined and executed in various ways, enabling the creation of complex programs.

II. Architecture Overview

The interpreter ForEmb was designed to execute on multiple platforms, including embedded systems, personal computers, and servers [15]. The options available for the interpreter depend on the platform -each system has varying resources such as memory, processor frequency, and operating system.

A. Stacks

The interpreter architecture was inspired by the Forth programming language, as shown in Fig. 1, where it has five stacks to hold values for word evaluation:

- Data stack -This is the main stack and the most used for word evaluation; here unsigned and signed integers operations are computed, as well as character operations, or symbols, respectively.

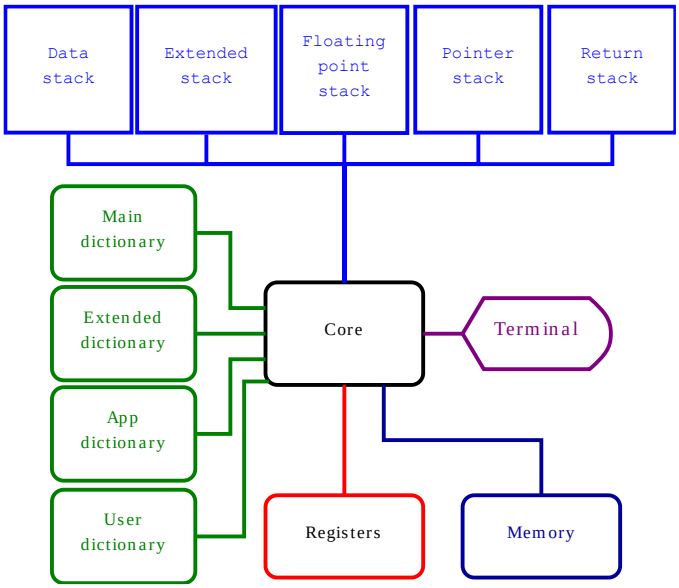


Figure 1. Interpreter architecture.

- Floating point stack -Words for floating-point numbers use this stack, which includes arithmetic operations, stack manipulation, and comparisons.
- Pointer stack -This stack is mainly used for arrays stored in memory, which include text strings (null-terminated) and lists. This stack includes words for string concatenation, integer and floating-point number conversion to text string representation, list operations, variables, constants, among others.
- Extended stack -This is an auxiliary stack that allows for the evaluation of complex words that require more platform resources. This stack is only available for personal computers and servers. Database and file system words are examples of such complex words.
- Return stack -Finally, script execution control inside the interpreter uses this stack to store the code memory address that is on evaluation -instruction pointer address, or IP- to implement control flow structures and create new words -user words- that include code instruction jumps and loops, where they require saving code memory address before making a conditional jump.

There are several advantages to using a stack over variables:

- Memory efficiency -Stacks use a small amount of memory because they store data in a last-in-first-out (LIFO) order. This means that only the most recent data needs to be stored in memory.
- Simplicity -Stacks are simple to use and require minimal programming knowledge. They have a basic set of operations -such as push and pop- that are easy to learn and use.
- Recursion -Stacks are well-suited for recursive algorithms, which are algorithms that call themselves repeatedly. When a function is called recursively, each call creates a new stack frame, which allows the function to save its state and return to it later.
- Flexibility -Stacks can be used for a wide range of applications, such as parsing, expression evaluation, and backtracking algorithms.
- Speed -Stacks are generally faster than variables because they operate on a fixed set of data structures that can be easily accessed and manipulated. This makes them ideal for applications that require fast processing, such as real-time systems.

In a stack, the top cell is called c_1, the next cell is c_2, and so on. However, stack notation is used to describe word operations, where the basic form is given by [13]

$$(\text{before} -- \text{after}) \quad (1)$$

The dash separates those cells that should be on the stack before evaluating the word from the items that will remain there afterwards.

In Table I, manipulation words for data stack are shown. There are five words that require a stack depth of one, whereas only *pick* and *roll* words require a stack depth greater than three. Furthermore, by using those words, it is possible to execute all the necessary operations on data stack, and most importantly, it is possible to implement new user-defined words.

Arithmetic operations for integer numbers, signed integer numbers, and symbols are shown in Table II, where *s/* and *s%* words are used to compute signed integer division and modular division, respectively. However, the rest of words in Table II can be used for any kind of cell in a data stack.

Words to operate over other stacks have been implemented in similar ways to the words shown in Table I. They exhibit the same behavior but use another stack. In fact, words names are formed by using the first preceding character of the stack name. For example, the word to delete cell c_1 in pointer stack is called *pdrop*.

TABLE I. DATA STACK MANIPULATION WORDS.

Word	Operation
drop	(c_1 – –), deletes cell c_1.
dup	(c_1 – – c_1 c_1), duplicates cell c_1.
emit	(c_1 – –), prints symbol cell c1 at terminal.
nip	(c_2 c_1 – – c_1), deletes cell c_2.
over	(c_2 c_1 – – c_2 c_1 c_2), duplicates cell c_2.
pick	(c_n ... c_2 c_1 – – c_n ... c_2 c_n), duplicates cell c_n, where n is given by c_1.
print	(c_1 – –), prints integer cell c_1 at terminal.
roll	(c_n c_{n-1}... c_2 c_1 – – c_{n-1} ... c_2 c_n), moves cell c_n to top, where n is given by c_1.
rot	(c_3 c_2 c_1 – – c_2 c_1 c_3), moves cell c_3 to top.
swap	(c_2 c_1 – – c_1 c_2), swap cells c_1 and c_2.
s.	(c_1 – –), prints signed integer cell c_1 at terminal.

TABLE II. ARITHMETIC WORDS FOR DATA STACK.

Word	Operation
+	(c_2 c_1 – – c_3), $c_3 = c_1 + c_2$
-	(c_2 c_1 – – c_3), $c_3 = c_2 - c_1$
*	(c_2 c_1 – – c_3), $c_3 = c_1 \times c_2$
/	(c_2 c_1 – – c_3), $c_3 = c_2 / c_1$
%	(c_2 c_1 – – c_3), $c_3 = c_2$ mod c_1
++	(c_1 – – c_2), $c_2 = c_1 + 1$
--	(c_1 – – c_2), $c_2 = c_1 - 1$
s/	(c_2 c_1 – – c_3), $c_3 = c_2 / c_1$
s%	(c_2 c_1 – – c_3), $c_3 = c_2$ mod c_1

B. Dictionaries

The interpreter has four dictionaries, which allow it to increase language flexibility according to the application requirements. Words are searched for in a specific order: main, extended, user, and applications dictionaries, respectively.

- Main dictionary -This dictionary has the primitive words that allow interpreter core operation. It has 87 words, which include arithmetic and logic operations, cell manipulation in different stacks, program control flow, and others. Those primitive words are the basis of implementing a new language.
- Extended dictionary -There are 71 new words added to the language with this dictionary which includes database and file system operations. However, such words are only present in personal computers and server versions of interpreter.
- User dictionary -This dictionary is the only one that starts off empty -does not have any defined words- where a programmer can add new words, thereby increasing capacity to the programming language. Words added to this dictionary can be updated with new versions, which is not possible to do in the previous dictionaries.
- Applications dictionary -Words contained in this dictionary are implemented using primitive words -found in the main dictionary. It is important to say that such words are available for all interpreter versions and can be replaced by user words, as defined in the user dictionary.

The main, extended, and applications dictionaries utilize a perfect hash function to efficiently search for words, employing a binary breakdown algorithm. A perfect hash function ensures that there are no collisions for a specific set of words -each word is mapped to a unique index in a hash table without any overlaps.

This perfect hash function was implemented using the GNU Gperf tool -a perfect hash function generator. To create the function and table, a list of strings is required in a file, and Gperf generates the hash function in the form of C or C++ code.

However, the user dictionary is unique in that it uses the Knuth multiplicative hashing algorithm [16]. To apply Knuth's multiplicative hashing to strings, it is necessary to process the string character by character and multiply each character by the golden ratio. One common method for doing this is to treat the string as a sequence of integer values that represent the characters in the string, and operate over these values.

Once there are integer values representing the string characters, it is possible to apply the basic formula for Knuth's multiplicative hashing, given by [16]

$$h(s) = m\ (\ s\ \varphi \bmod 1) \qquad (2)$$

where m is the size of the hash table, s is the numeric value representing the string and φ is the golden ratio, which is approximately equal to 1.618.

C. Memory Management

Memory management is closely related to the pointer stack; it is precisely in this stack where memory addresses for text

strings, lists, variables, and constant words are stored. Different operations over text string symbols and list atoms could be performed, as shown in Table III.

While the word *allocate* holds space in memory to create a new list, the word *free* releases the list reserved space back into memory. Similarly, the words *$allocate* and *$free* reserve and releases space in memory for text strings, respectively. Those words can be used in combination with the ones in Table III to perform more complex operations, such as text string concatenation or arithmetic operations between two integer lists.

D. Core and other Blocks

Core block, shown in Fig. 1, is the main part of interpreter; it controls word evaluation by using stacks, dictionaries, and the memory management block that administers memory occupancy. The terminal is used to display information to the user and could be in any device, such as screen monitors, OLED displays, LCDs, USB communications.

There are eight registers available for core operation which allow for defining integer and floating-point numbers format, i.e., *base*, *decimals*, and *digits* words, respectively. The remaining five registers are only read, and contain information on extended dictionary availability, number of kernel bits, version, number of system bits, and platform information where script is running, respectively.

Most of the words implemented in ForEmb operates over one stack; however, there are a few words which use more than one stack, as described in the last section. The conversion words shown in Table IV allow conversion from one cell type to another, and they use two stacks to perform such a conversion.

TABLE III. MEMORY WORDS FOR POINTER STACK.

Word	Operation
@	PS (c_1 – –), retrieves integer number from address in cell c_1.
!	PS (c_1 – –), stores integer number to address in cell c_1.
c@	PS (c_1 – –), retrieves symbol from address in cell c_1.
c!	PS (c_1 – –), stores symbol to address in cell c_1.
r@	PS (c_1 – –), retrieves real number from address in cell c_1.
r!	PS (c_1 – –), stores real number to address in cell c_1.
l@u	PS (c_1 – – c_1), retrieves integer number from list in cell c_1.
l!u	PS (c_1 – – c_1), stores integer number to list in cell c_1.
l@r	PS (c_1 – – c_1), retrieves real number from list in cell c_1.
l!r	PS (c_1 – – c_1), stores real number to list in cell c_1.
$@c	PS (c_1 – – c_1), retrieves symbol from text string in cell c_1.
$!c	PS (c_1 – – c_1), stores symbol to text string in cell c_1.

Using conversion words, it is possible to send information to the terminal. For example, to send a real number to the terminal, the *r>$* word could be used to convert the real number in cell c_1 at the floating-point stack into a text string in cell c_1 at the pointer stack. After that, it is necessary to retrieve each symbol using the *$@c* word and print it with the word *emit*, as shown in Listing 1.

Listing 1. Real number print on the terminal.

```
1   r>$
2   0
3   while dup $@c dup
4   {
5   emit
6   ++
7   }
```

E. Automatic Testing

An automatic testing script was designed to test words from all dictionaries. Such automatic testing was organized according to the complexity of words, starting from primitive words -those found in the main dictionary- and moving on to words in the applications dictionary.

Fig. 2 shows the results of the automatic testing on Linux PC, where 173 tests were performed successfully, with a maximum of eight cells used for the data stack, five cells used for the floating-point and pointer stacks, and three cells used for the return stack.

In Fig. 3 an automatic testing was performed on a server running FreeBSD 13.1-RELEASE-p3; where 173 tests were successfully executed. By comparison, Fig. 4 shows the automatic testing results on a Raspberry Pi Pico board, where 161 tests were performed successfully. It is important to note that there are fewer tests to perform compared to the PC and server versions because the extended dictionary is not available on the microcontroller version of the ForEmb interpreter.

Real numbers arithmetic testing is implemented in Listing 2, where a user-defined word *pass++* prints an *'ok'* message to the terminal and increments the number of successful tests,

TABLE IV. CONVERSION WORDS BETWEEN STACKS.

Word	Operation
>r	Converts integer to real number.
>$	Converts integer number to text string.
s>r	Converts signed integer to real number.
s>$	Converts signed integer number to text string.
r>s	Converts real to signed integer number.
r>$	Converts real number to text string.
$>s	Converts text string to signed integer number.
$>r	Converts text string to real number.

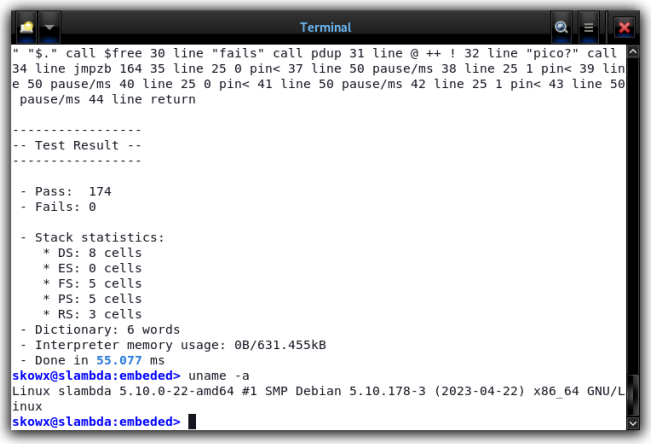


Figure 2. Automatic testing results for Linux.

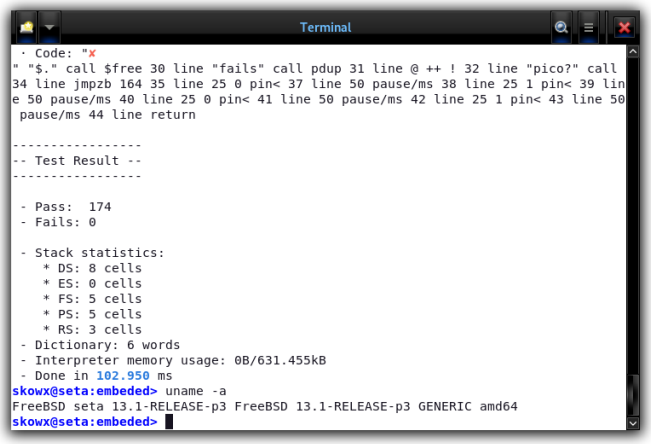


Figure 3. Automatic testing results for FreeBSD.

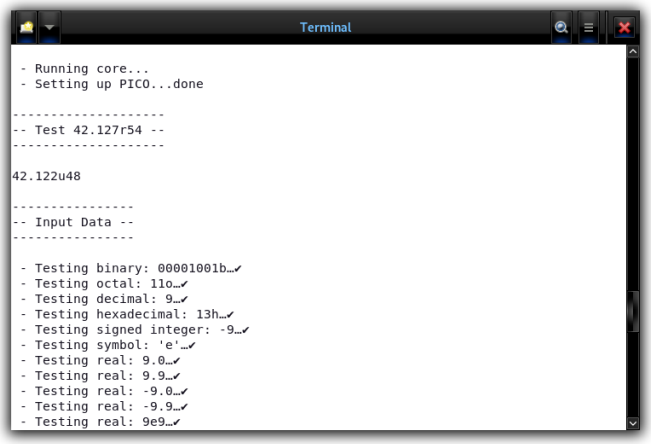


Figure 4. Automatic testing results for Raspberry Pi Pico.

while the user-defined word *fails++* prints an *'error'* message to the terminal and increments the number of failed tests. Additionally, a visual confirmation is performed on the Raspberry Pi Pico board, by using board LED. All arithmetic words for real numbers have been included in automatic testing.

Listing 2. Automatic testing for real numbers in ForEmb.

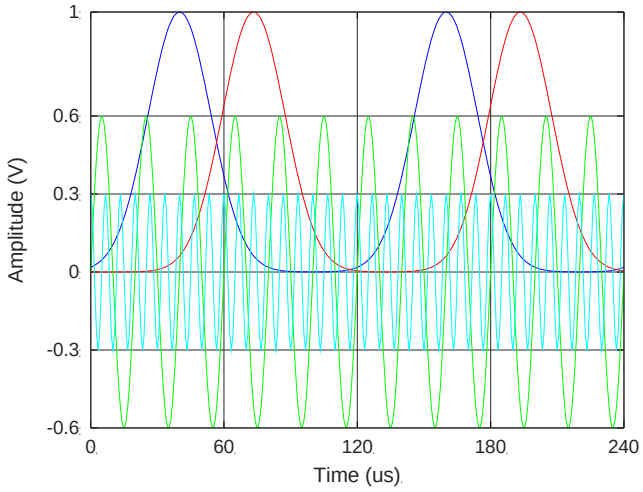
```
0 variable pass

: pass ++
“ok\n” $.f
pass pdup
@ ++ !
if pico?
{
25 0 pin<
100 pause/ms
25 1 pin<
100 pause/ms
}
end

“ – Testing: r+...” $.f
3.3 9.9 r+
if 13.2 r=
{
pass++
}
else
{
fails++
}
```

To evaluate the performance of the ForEmb interpreter, a signal information used to test communication systems as a simulation. This simulated signal was then compared with the response obtained from Matlab. The details of this process are described in detail in the next section.

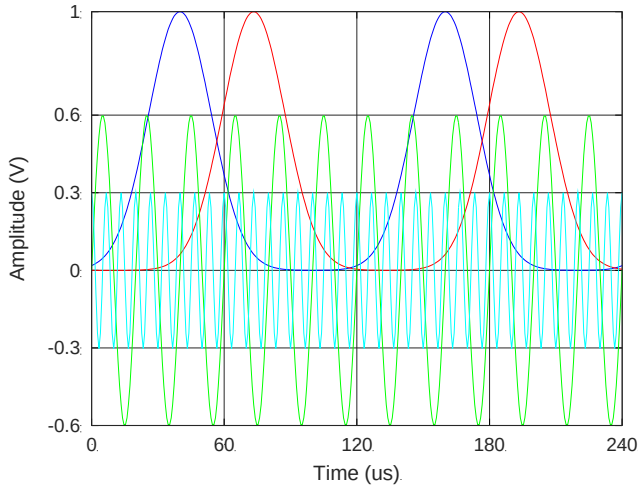
III. Simulation Results

Software simulation is an acknowledged method for evaluating real-time applications [8]. To measure interpreter performance, an information signal that is commonly used for evaluating communication systems during the design phase was simulated using a 150kHz base frequency and compared with a Matlab simulation on a PC with an Intel Core i9 CPU running at 2.80GHz with 16 GB of RAM. The information signal was computed from sine, cosine, and exponential signals, as shown in Fig. 5, where fundamental signals calculated in ForEmb follow the signals obtained from Matlab.

Exponential signals are multiplied by sine and cosine signals, respectively, in order to create periodic signals, as shown in Fig. 6, and compute the information signal, whose model is



(a) Matlab simulation.



(b) ForEmb simulation.

Figure 5. Fundamental signals for computing information signal.

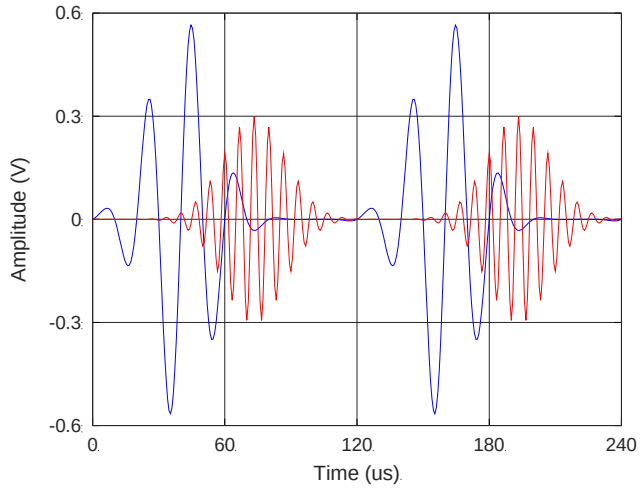


Figure 6. Periodic signals simulated in ForEmb.

given by

$$s = V_{bias}\left(e_0\, V_p \sin(\omega_3\, t) + e_0 \frac{V_p}{2} \cos(\omega_1\, t)\right) \quad (3)$$

with

$$e_0 = e^{-\left(\frac{\frac{t}{T}-c}{\omega}\right)^2} \quad (4)$$

$$\omega_1 = 2\,\pi\, f \quad (5)$$

$$\omega_3 = \frac{2\,\pi\, f}{3} \quad (6)$$

where V_{bias} is the biasing voltage, V_p is the peak voltage, T is the fundamental period, c is time shifting variable, w is signal duration and t is time vector used for simulation.

Model implementation from (3) was required to perform arithmetic operations between lists. To perform these operations, a new user-defined word *rmap* was created, which executed a given word for each atom in a real list. This included operations such as addition, subtraction, multiplication, and others, as shown in Listing 3. The word *atoms* returned the size of the list in the data stack, while the word *call* searched for a word in the user and application dictionaries and executed it when found. Memory words *l@r* and *l!r* were used to retrieve and store real numbers from list, respectively. Real numbers were passed to the floating-point stack, where they could be operated using any word to perform mathematics on them, and then the real number in the list could be updated with this new value.

Listing 3. *rmap* word implementation in ForEmb.

```
1   : rmap
2   atoms
3   pswap
4   while dup
5   {
6   --
7   dup l@r
8   $over call
9   dup l!r
10  }
11  drop
12  pswap free
13  end
```

Trigonometric functions such as sine and cosine, as well as the exponential function, have been implemented in ForEmb to simulate the information signal from (3). Taylor series were used to compute these functions, which were given by

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} \quad (7)$$

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} \quad (8)$$

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} \quad (9)$$

where the values of *n* used to implement the cosine, sine and exponential functions in ForEmb were chosen to be 7, 6 and 13, respectively.

Listing 4 shows the cosine word implemented in ForEmb. Factorial number fractions in (7) are precomputed by using 20 decimals to improve function evaluation and reduce execution time.

Listing 4. Cosine, exponential and sine words implementation in ForEmb.

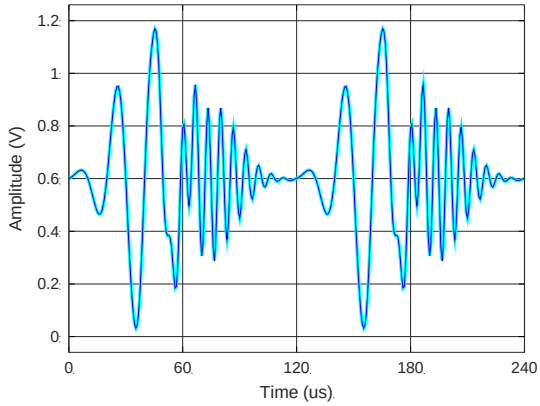
```
1   : cos
2   rabs
3   2pi r/ r% 2pi r*
4   if rdup pi r>=
5   {
6   2pi r−
7   }
8   1.0
9   rover 2 ^
10  −5.0e−01 r* r+
11  rover 4 ^
12  4.16666679084300994873e−02 r* r+
13  rover 6 ^
14  −1.388888922519981861l1e−03 r* r+
15  rover 8 ^
16  2.4801587642286904l562e−05 r* r+
17  rover 10 ^
18  −2.75573199814971303567e−07 r* r+
19  rover 12 ^
20  2.08767558795841523533e−09 r* r+
21  rswap 14 ^
22  −7.81894271550953590122e−10 r* r+
23  end
```

Table V shows the description of words used to implement the cosine function. Most of these words were used for arithmetic operations with real numbers. However, new word *cos* can only compute cosine function from cell c_1 in the floating-point stack. To compute cosine function for all real numbers on a list in cell c_1 in the pointer stack, it is necessary to use word *rmap*, as shown in Listing 3. The same situation occurred with sine and exponential functions, which are needed to compute information signal from (3).

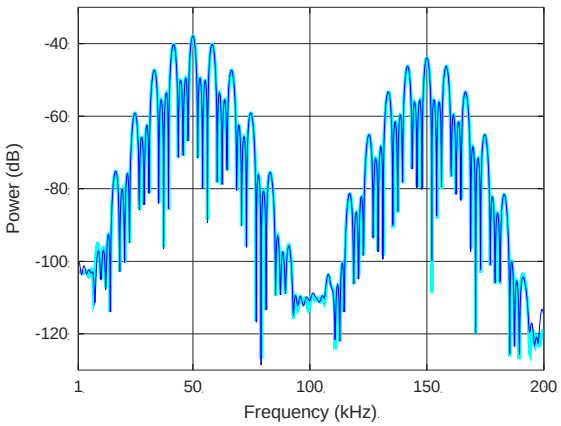
In Fig. 5(a), the information signal model from (3) was simulated using Matlab, whereas in Fig. 5(b) the same in formation signal model was simulated with ForEmb, resulting in an accurate response compared with Matlab. The sine and

TABLE V. USED WORDS FOR COSINE.

Word	Operation
:	Makes new user word.
2pi	2π constant.
pi	π constant.
^	Raise real number to power.
rabs	Gives absolute value from real number.
rdup	Duplicates c_1 cell from floating point stack.
rover	Duplicates c_2 cell from floating point stack.
rswap	Interchanges cells c_1 and c_2 from floating point stack.
r>=	Compares real numbers from cells c_1 and c_2, where inserts 1 in data stack when c_2 is greater or equal than c_1, and 0 otherwise.
r+	Real number addition.
r-	Real number subtraction.
*r**	Real number multiplication.
r/	Real number division.
r%	Gives integer part from real number.



(a) Information signal simulation.



(b) Information signal spectrum.

Figure 7. Simulation for information signal in Matlab and ForEmb, when the base frequency is 150kHz.

cosine signals have a frequency difference of 3, as shown in (3), where the cosine signal was faster than the sine signal.

When signals shown in Fig. 5 were combined according to (3), information signal shown in Fig. 7(a) was obtained, where the information signal simulated in ForEmb followed the Matlab simulation response, with a maximum absolute error percent of 0.18% and variance of $1.06\times10-8$.

Frequency domain analysis is commonly used to evaluate communication systems performance [17]. The spectrum of the information signal obtained from ForEmb simulation -in darker color- as shown in Fig. 7(b), is similar to the spectrum of the signal obtained from Matlab -in lighter color.

Another evaluation test was conducted to validate the performance of the ForEmb interpreter. This particular test involved real-time operation and generated two signals for controlling fuel injectors. The details of this test are described in the following section. It is worth noting that the test was measured and conducted experimentally to obtain accurate results.

IV. Experimental Results

An automobile engine is a complex machine equipped with an Engine Control Unit (ECU). The ECU's main functions is to read sensor signals and analyze them to determine the engine status at any given time. Based on this information, the ECU generates control signals in accordance with the driver requirements.

For instance, the Crankshaft Position Sensor (CKP) and Camshaft Position Sensor (CMP) provide essential signals that enable the ECU to ascertain the engine position in the Otto cycle. This information is then utilized by the ECU to generate fuel injector signals, ensuring the adjustment of the engine revolutions.

The Otto cycle is a thermodynamic cycle that describes the idealized process of a four-stroke internal combustion engine. It is named after Nikolaus Otto -the German engineer who invented the four-stroke engine [18].

The Otto cycle consists of four processes:

- Intake -The piston moves downward, creating a vacuum that draws the air-fuel mixture into the combustion chamber.
- Compression -The piston moves upward, compressing the air-fuel mixture to increase its pressure and temperature.
- Combustion (Power) -At the top of the compression stroke, a spark ignites the compressed mixture, causing a rapid combustion and generating high pressure. This high-pressure expansion pushes the piston downward, producing power.
- Exhaust -The piston moves upward again, expelling the burned gases from the combustion chamber.

Fuel injection timing has an important role in increasing engine performance, where such injection timing must be controlled according to CMP and CKP sensors [19], [20].

Raspberry Pi Pico is a low-cost, high-performance microcontroller board; it has a RP2040 dual core microcontroller running up to 133 MHz with a 12-bit ADC, real-time counter, and 26 multi-function GPIO PINs, which allow to send and receive signals from real world, and it supports SPI, I2C, UART and PWM channels, where it works with a supply voltage of 3.3 V, as shown in Fig. 8 [21], [22].

To evaluate the real-time performance of the interpreter on embedded systems, ForEmb was ported to the Raspberry Pi Pico board with a compiled size of 50.3kB. It was used to emulate the CMP and CKP signals to evaluate injection timing, as shown in Fig. 9. The injectors were mounted in fuel injection rail, which was connected to a fuel deposit, and the pressure was measured with a pressure gauge and an electronic sensor. It is important to note that the pressure variable had an important role inside control algorithm to adjust injection timing, along with the CMP and CKP sensors signals.

The CMP and CKP sensor signals were measured from a Neon 2.0 L engine -as shown in Fig. 10- using a Tektronix TDS 210 oscilloscope. Then, such sensor signals were digitized to emulate them. A period was considered to be two turns of the crankshaft for every one turn of the camshaft. This period was then divided into smaller time intervals by taking the smallest time from the CKP sensor signal when it was highest. This step resulted in a time interval of approximately 10 ms. Thus, the period was divided into time intervals of 10 ms, where a clock signal transition -from low to high- was used as a reference signal.

PINs 12 and 13 from the RP2040 microcontroller were chosen for the CMP and CKP signals, respectively, while PIN 11 was used for reference clock signal, and PIN 25 was a LED mounted in Raspberry Pi Pico board, which indicated when the emulated ECU signal is running. Listing 5 shows the PINs mask value -where each bit correspond to one PIN- in hexadecimal format inside a constant integer word named *pins*.

Listing 5. Raspberry Pico PINs mask for output signals.

```
1    2003800h constant pins
```

Reference clock signal period could be controlled by *delay* user-defined word, which implementation is shown in Listing 6, where *pause/ms* word made a pause in program execution by required time given in ms, and a 10 ms pause was executed when such delay user-defined word was found by interpreter.

Figure 8. Raspberry Pi Pico board.

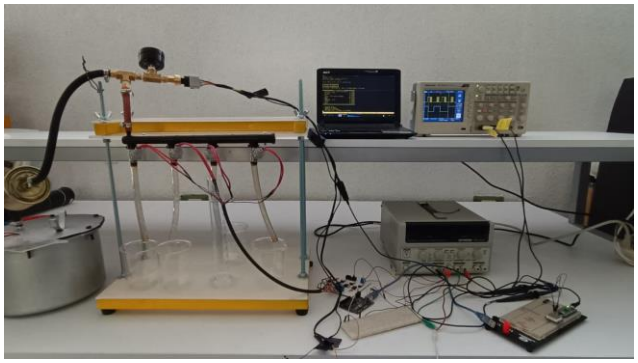


Figure 9. Fuel injector test bench.

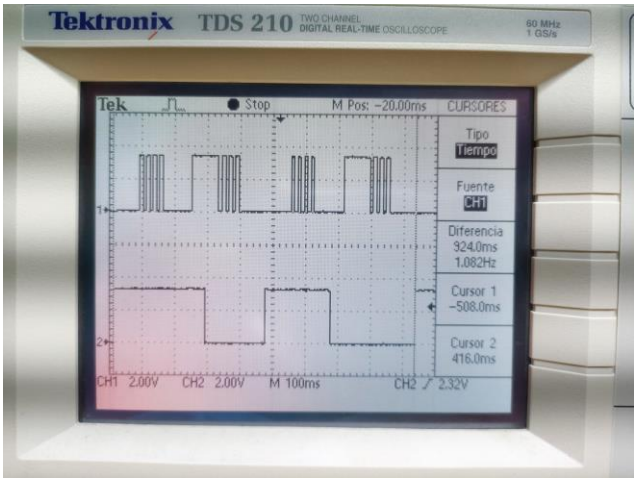


Figure 10. CKP and CMP signals for Neon 2.0 L engine.

Listing 6. Delay user word for reference clock signal period.

```
1    : delay
2    10 pause/ms
3    end
```

The minimum time interval corresponds to one clock signal period -low and high time- and was emulated by a user-defined word. There were several possible combinations for CKP and CMP sensors signals, where the simplest time interval was when such signals did not change at all during the clock signal period.

The digitized time interval shown in Table VI, where the CKP and CMP sensors signals do not change their values at all during the clock signal period, was implemented in user- defined word *s01* shown in Listing 7. It is important to note that clock signal and LED on board changed from 0 to 1, and all signals were synchronized.

In Listing 7, the cell c_1 from data stack was used to specify the number of times to execute the user-defined word *s01* using *while* loop. Meanwhile, the word *pins<* wrote the given value from cell c_1 on the data stack in output-enabled PINs.

TABLE VI. S01 TIME INTERVAL FOR CKP AND CMP SENSORSL SIGNALS.

CLK	CKP	CMP
0	0	1
1	0	1

Listing 7. *s01* user word for CKP and CMP sensor signals.

```
1    : s01
2    while dup
3    {
4    pins 2001800h pins<
5    delay
6    pins 0001000h pins<
7    delay
8    --
9    }
10   drop
11   end
```

One of the most complex time intervals occured when CKP or CMP sensor signals change their values during the clock signal period, as shown in Table VII. In this case, the CKP sensor signal maintained its value, while CMP sensor signal changed its value from 0 to 1 when the clock signal made a transition from low to high.

The user-defined word *s0001* implementation is shown in Listing 8, for the digitized time interval shown in Table VII. Cell c_1 from the data stack was used to specify the number of times to repeat the signal using a *while* loop. The word *pins<* wrote 0 for CKP and CMP sensors when reference clock was low, and changed CMP to 1 when reference clock was high.

Listing 8. *s0010* user word for CKP and CMP sensor signals.

```
1    : s0010
2    while dup
3    {
4    pins 2000800h pins<
5    delay
6    pins 0002000h pins<
7    delay
8    --
9    }
10   drop
11   end
```

Using a similar approach to how the words *s01* and *s0010* were implemented, the necessary time intervals were digitized and implemented to emulate the signals from the CKP and CMP sensors over a period of time for the ECU of Neon 2.0 L engine. Listing 9 shows the implementation of the CKP and CMP signals based on time intervals. The word *include* inserts the code from the file *signals.leo* into the ECU script, while

TABLE VII. S0001 TIME INTERVAL FOR CKP AND CMP SENSORSL SIGNALS.

CLK	CKP	CMP
0	0	0
1	0	1

the word *pins:out* enables the *pins* integer constant word mask as outputs for the RP2040 microcontroller. The integer number before the user-defined word indicates how many times such user-defined word would be repeated, i.e., 4 *s01* indicates that signal in user-defined word *s01* will be repeated four times. This integer number allows for a reduction in the number of words necessary for implementation of the ECU signal.

Listing 9. CKP and CMP sensor signals for ECU emulation.

```
1   include signals.leo ;
2
3   " − Setting up PICO...\n" $.f
4   pins pins:out
5   " − Starting simulation...\n" $.f
6   " − ECU Neon 2.0L signal...\n" $.f
7
8   while 1
9   {
10  4 s01
11  4 s1101
12  4 s01
13  2 s11
14  2 s10
15  3 s0010
16  4 s00
17  4 s01
18  4 s1101
19  2 s01
20  2 s00
21  4 s10
22  3 s0010
23  4 s00
24  }
25  end
```

By executing Listing 9 in ForEmb interpreter running on a personal computer, it was possible to simulate the ECU signal in the terminal, as shown in Fig. 11. This simulation allowed for verification of correct functionality before inserting the code into the embedded system; code depuration could also be done in PC when necessary. It is possible to see that required PINs, 11 to 13 and 25, are output enabled and have a required signal which changes over time.

When the simulation was correct, the next step was to insert code into the embedded system and perform experimental

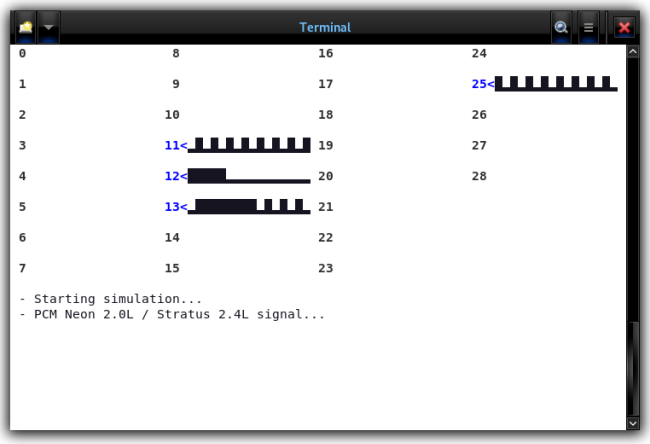


Figure 11. CKP and CMP signals simulation.

characterization. Doing so allowed for the validation of real-time operation for the ECU signal generated by ForEmb, while it was running on Raspberry Pi Pico board.

Fig. 12 shows CKP and CMP sensor signals measured with a Tektronix TDS 2024C digital storage oscilloscope while they were active. A computer located in the left allowed for stablishing communication with the Raspberry Pi Pico board using USB, in order to monitor parameters from microcontroller when ECU signal was running.

Emulated CKP and CMP signals were sent to the control circuit, which used them to control fuel injector timing, has shown in Fig. 9.

V. CONCLUSIONS AND FUTURE WORK

ForEmb was compiled and successfully tested on personal computers and servers running Linux, FreeBSD, and a Raspberry Pi Pico board, respectively. An automatic test script allowed it to reduce development time and validated each code improvement.

An information signal for communication systems evaluation was implemented in ForEmb using real number lists, where trigonometric and exponential signals were performed successfully. Simulation results show an absolute error percentage of 0.18% with respect to the Matlab response, validating ForEmb precision.

The Raspberry Pi Pico port of the ForEmb interpreter allowed for the evaluation of scripts in RP2040 microcontroller. An ECU signal for a Neon 2.0 L engine was successfully simulated on a PC and then downloaded to a Raspberry Pi Pico board, where an experimental characterization of the real-time embedded system showed a valid signal emulation.

ECU signal emulation was used on fuel injectors test bench, where timing control model could be developed.

Dynamic compilation in ForEmb, allowed to change code behavior without the need to recompile source code and reprogram microcontroller.

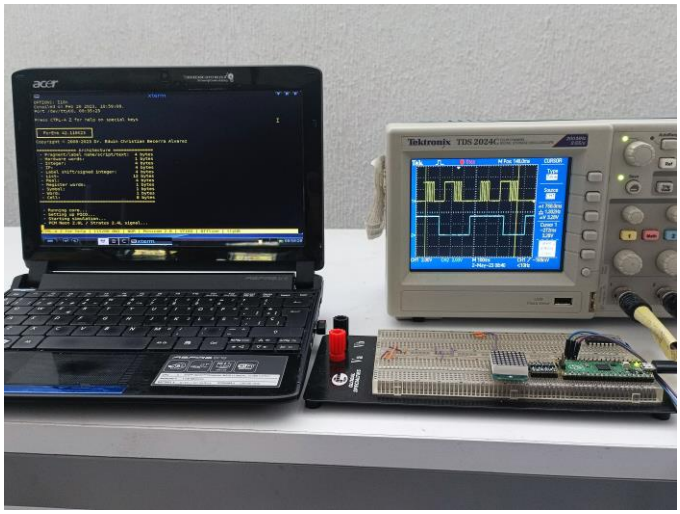


Figure 12. CKP and CMP emulated signals in real-time for ECU.

For future work, it is planned to port the ForEmb interpreter to the ATmega2560 microcontroller to increase the number of available platforms. Additionally, user-defined words will be packed in dictionaries for general applications, such as trigonometric and logarithmic functions.

ACKNOWLEDGMENT

This work is supported by the National Council of Science and Technology (CONACYT), Secretariat of Public Education (SEP) and the University of Xxxxxxx.

REFERENCES

[1] J. S. Furter and P. C. Hauser, “Interactive control of purpose built analytical instruments with Forth on microcontrollers - A tutorial,” Analytica Chimica Acta, vol. 1058, pp. 18–28, 2019.

[2] V. B. Y. Kumar, D. Shah, M. Datar, and S. B. Patkar, “Lightweight Forth Programmable NoCs,” in Proc of the 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), pp. 368–373, 2018.

[3] N. Shriethar, N. Chandramohan, and C. Rathinam, “RASPBERRY PI-BASED SENSOR NETWORK FOR MULTI-PURPOSE NONLINEAR MOTION DETECTION IN LABORATORIES USING MEMS,” Revista de F´ısica, no. 65, pp. 52–64, 2022.

[4] J. B. Thiyagarajan and M. Thothadri, “FITNESS MONITORING SYSTEM WITH RASPBERRY PI PICO,” International Journal of Science Academic Research, vol. 2, no. 7, pp. 1840–1845, 2021.

[5] S. M. Sonti, A. Harika, C. Dileep, and B. G. Raju, “FIRE DETECTING AND ALERTING SYSTEM USING RASPBERRY PI PICO,” International Research Journal of Modernization in Engineering Technology and Science, vol. 4, no. 6, pp. 4565–4569, 2022.

[6] D. M. Hanna, B. Jones, L. Lorenz, and S. Porthun, “An embedded Forth core with floating point and branch prediction,” in Proc of the 2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS), pp. 1055–1058, 2013.

[7] M. Bos˘njak, T. Rockta¨schel, J. Naradowsky, and S. Riedel, “Programming with a Differentiable Forth Interpreter,” in Proc of the 34th International Conference on Machine Learning, vol. 70, pp. 547–556, 06-11 Aug 2017.

[8] S. Baranov, “Real-time multi-task simulation in Forth,” in Proc of the 2016 18th Conference of Open Innovations Association and Seminar on Information Security and Protection of Information Technology (FRUCT-ISPIT), pp. 21–26, 2016.

[9] Z. Lie-Ping and L. Xiao-Jing, “Improvement on MISC computer memory architecture based on forth,” in Proc of the 31st Chinese Control Conference, pp. 4734–4738, 2012.

[10] P. Leong, P. Tsang, and T. Lee, “A FPGA based Forth microprocessor,” in Proc of the IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251), pp. 254–255, 1998.

[11] D.-H. Kim, J.-P. Kim, and J.-E. Hong, “Practice Patterns to Improve the Quality of Design Model in Embedded Software Development,” in Proc of the 2009 Ninth International Conference on Quality Software, pp. 179–184, 2009.

[12] S. Nadkarni, P. Panchmatia, T. Karwa, and S. Kurhade, “Semi natural language algorithm to programming language interpreter,” in Proc of the 2016 International Conference on Advances in Human Machine Interaction (HMI), pp. 1–4, 2016.

[13] B. Brey, Thinking Forth. United States of America, New Jersey: Pearson: Prentice Hall, 1 ed., 2014.

[14] K. V. Prashanth, P. S. Akram, and T. A. Reddy, “Real-time issues in embedded system design,” in Proc of the 2015 International Confer- ence on Signal Processing and Communication Engineering Systems, pp. 167–171, 2015.

[15] “ForEmb repository in GitHub.” https://github.com/labcibernetica/ForEmb. Accessed: 2023-05-05.

[16] D. E. Knuth, The Art of Computer Programming. United States of America: Addison-Wesley, 1 ed., 2011.

[17] S. Lara, J. Azocar, I. Soto, and S. Gutierrez, “Performance analysis of a hybrid RF/FSO comunication system with QKD for ventilation monitoring,” in Proc of the 4th West Asian Symposium on Optical and Millimeter-wave Wireless Communications (WASOWC), pp. 1–5, 2022.

[18] Y. Haseli, Entropy Analysis in Thermal Engineering Systems. MI, USA: Academic Press, 2020.

[19] H. A. Dhahad, M. A. Fayad, M. T. Chaichan, A. Abdulhady Jaber, and T. Megaritis, “Influence of fuel injection timing strategies on per- formance, combustion, emissions and particulate matter characteristics fueled with rapeseed methyl ester in modern diesel engine,” Fuel, vol. 306, p. 121589, 2021.

[20] M. A. Fayad, “Effect of fuel injection strategy on combustion performance and NO x /smoke trade-off under a range of operating conditions for a heavy-duty DI diesel engine,” Springer Nature, vol. 1, no. 9, 2019.

[21] Raspberry Pi Ltd, Raspberry Pi Pico Datasheet, 3 2023. Rev. ae3b121-clean.

[22] S. Smith, RP2040 Assembly Language Programming: ARM Cortex-M0+ on the Raspberry Pi Pico. Berkeley, CA: Apress, 2022.