DOI: 10.15439/2022F275
ISSN 2300-5963 ACSIS, Vol. 31

Small Footprint Embedded Systems Paradigm Based on a Novel and Scalable Implementation of FORTH

Bogusław Cyganek
AGH University of Science and Technology, Poland
Al. Mickiewicza 30, 30-059 Kraków, Poland
cyganek@agh.edu.pl

***Abstract*—This paper describes architecture of the novel implementation of the Forth interpreter-compiler. The architecture follows the object- and component-oriented design paradigms. The implementation is done with the modern C++ 20 language taking full advantage of such constructs as lambda functions, variadic templates, as well as the coroutines and concepts. The system is highly modular and easily scales for small footprint embedded systems. We propose to extend Forth with the coroutine words that allow for async operations and lightweight cooperative multi-threading. We show successful deployment of the proposed Forth implementation on three platforms, two PC frameworks running Linux and Windows, respectively, as well as on tiny embedded system NodeMCU v3 with the 32-bit RISC ESP8266 microprocessor and 32/80KB memory. The platform also has educational value, showing intrinsic operation of Forth and modern C++. Software is available free from the Internet.**

***Keywords—*: Forth, compiler-interpreter, multi-tasking, co-routines, co-operative systems, IoT**

I. Introduction

Forth is a computer language developed by Charles Moore in early 70s as a system to control the radio telescope when he worked in the National Radio Astronomy Observatory [11][13][19]. Its name was coined to commemorate the fourth generation of computers but since the file system restricted names to five letters only, Moore skipped the middle "U" and left Forth. The fascinating story of Forth is described in The Evolution of Forth [11], while a short biography of Charles Moore is in Wikipedia [13]. Forth has always been very outstanding, original and interesting computer language [1][10][12][14][15]. Although not in the mainstream, slightly forgotten today, we are deeply convinced it can still serve many purposes. This is especially true in the context of small embedded systems that need interactive features, such as ones for the Internet of Things (IoT), and also if Forth can be shown in the new light of a novel implementation in modern C++, as presented in this paper.

There are many free and commercial implementations of Forth, such as *Gforth*, which is a free GNU portable implementation of the ANS Forth standard for Linux/Unix, Windows, and other operating systems [20]. Another implementation is *Swift Forth*® by Forth Inc. [21]. On the other hand, a popular implementation with many follow ups is *jonesforth* project [22]. We just named few of the available projects, many more can be found online [19][23].

However, to the best of our knowledge, none of the above mentioned implementation uses modern C++, i.e. ver. 17 or 20 [5][24]. On the other hand, having a Forth implementation done with modern C++ allows to use the latest very efficient and productive features of C++, such as STL containers, variadic templates, on-time compilation, regular expressions, lambda functions, and coroutines. Especially the latter offers new ways of efficient implementation of the async IO operations, state machines, or lightweight multithreading, as will be discussed. Hence, the proposed implementation greatly reduces system complexity, at the same time allowing for scalable solutions. The complete Forth project presented in this paper, named *BCForth*, is available free from the Internet [16]. This also makes it a good teaching platform for the computer classes.

But most of all, what can be interesting in Forth when confronted with e.g. modern C++? The main difference is presence of the interpreter and compiler, at a relatively small footprint on the other hand. This means that, contrary to C++, which to add a new software component requires recompilation and rebuild, a Forth based system is very interactive and extensible. That is, the user can run the existing words but also can extend the system by his/her defined new words, which are immediately compiled and instantly become available for construction of next words, and so on. Not less important is the mentioned small footprint of Forth, which renders it useful for small embedded platforms, IoT, or even in the so called bare-metal systems. Hence, we can easily imagine a simple but smart sensor, which is run by Forth alone and allows communication, as well as extensions, in the run time.

The rest of the paper is organized as follows. Architecture of the Forth platform is presented in Section II. It is organized in four subsections: core architecture (II.A), key data structures (II.B), hierarchy of Forth words (II.C), and finally the system activity (II.D). The coroutine component – a proposed novel add-on to the Forth language – is dealt with in Section (III).

©2022, PTI

System deployment and experiments are presented in Section (IV). The paper ends with conclusions in Section (V).

II. Architecture of the Novel Forth Platform

The main purpose of the *BCForth*, is to provide a flexible implementation of Forth with the modern C++20, which can be easily ported to various embedded platforms endowed with the C++ compiler. *BCForth* contains also an extension in the form of the coroutines, as will be discussed. Contrary to some older implementations in assembly or C, modern C++ allows clear, understandable and extensible code. For instance, if necessary *BCForth* can be reduced of its components (e.g. it can run only with the interpreter), or it can be even ported to the older version e.g. C++ 11.

In this section we present basic assumptions behind the architecture of *BCForth*, while its implementation can be accessed free from the GitHub [16].

A. Core Architecture

Fig. 8 depicts the overall architecture of the Forth language defined in the project *BCForth*. The role and responsibilities of each class in the hierarchy are as follows.

- `TForth` – the base class defining all basic data structures, such as: the data stack represented by `DataStack`, the words' dictionary `WordDict` (`std::unordered_map`), as well as the auxiliary return stack `RetStack`.
  `TForth` defines the `WordEntry`, which is the structure holding all necessary information about a word and kept as a value of each word in the dictionary. `InsertWord_2_Dict` inserts a newly created word to the dictionary, whereas `GetWordEntry` retrieves a word from the dictionary by providing its name as a key; `WordOptional` is returned to cope with situations of non-existing words. Various words are represented as objects from the rich `TWord` family. These have access to the data stack defined in `TForth`. Each word present in the `TForth` dictionary is ready to be executed by calling the `ExecWord` with the `word_name` as its parameter. Hence, `TForth` alone, is sufficient to handle the pre-defined and non-contextual words (i.e. ones that don't need any other tokens from the input stream). This makes `TForth` alone a minimalistic Forth system. `TForth` defines also an auxiliary vector `NodeRepo` to hold objects that need to be present but that do not go to the dictionary of words (Fig. 8). These are e.g. compiled-in literals.
- `TForthInterpreter` – derived from `TForth` is responsible for handling the interpreter mode, in which a stream of `tokens` is processed and executed. Operation of `TForthInterpreter` mostly relies on interpreting the incoming stream of tokens, as integer or floating-point literals (these are distinguished by the dot `.` inside the literal), or as word names to be executed and their optional parameters. However, no new words can be defined (this is a role left for `TForthCompiler`).
- `TForthCompiler` – extends `TForthInterpreter` by providing `the` ability of entering definitions of new words. New words can be entered to the dictionary (Fig. 2) with the defining construction colon-semicolon (`: ;`). For instance,

  ```
  : ACTION DO I . CR LOOP ;
  ```

  defines a new word `ACTION` which upon a call

  ```
  23 0 ACTION
  ```

  prints all values 0-22, each in a new line.
  However, apart from the calls to the words already defined and registered in the dictionary, word definitions can contain nested structural words, such as `IF` … `THEN` ... `ELSE`, `DO` … `LOOP`, etc., as well as the two-stroke `CREATE` … `DOES>` creational pattern, or the `IMMEDIATE` / `POSTPONE` handling modes.
- `TForthReader` – an auxiliary class for converting a text stream, such as a terminal window or a text file, into a stream of Forth's tokens. This is done by text splitting over the white symbols (space, tab, new line), as well as after stripping off the Forth's comments. This way obtained stream of text tokens is fed to the interpreter and/or compiler objects, as described in Section (II.D).

One of the main architectural assumption is a strong separation of the input stream processing components, the token stream processing components, and the word defining objects. In other words, the latter does not bother with any variants of the input and output terminals. On the other hand, the streams of Forth tokens are obtained by the `TForthReader` object. If this is a word definition, tokens are passed to `TForthCompiler`, in order to compile-in a new word. Otherwise, tokens go to `TForthInterpreter` for word(s) execution.

B. Key Data Structures

Details of Forth can be found in many sources [1][4]. Here we focus mostly on the basic data structures and operations which they are used for. Fig. 1 depicts few characteristic operations on the Forth's data stack.

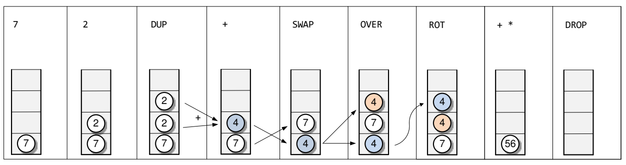


Fig. 1 Examples of the most common stack operations in Forth. All values are entered in the RPN. DUP duplicates the top value of the stack. A binary operator, such as +, removes the two topmost values, performs the operation, and pushes the result onto the stack. SWAP changes order of the two topmost values. OVER copies the second operand and pushes it to the top of the stack. ROT does the rotation of the three topmost stack values. DROP removes the topmost value from the stack.

The operations are straightforward once we recall that all operations are in the Reverse Polish Notation (RPN) [5]. It can be observed that each newly entered value (object) is pushed onto the data stack. Each operation, on the other hand, such as the + operator, or a DUP (duplicate) operation, pops off the necessary number of parameters, performs its specific action, and pushes the result, if there is any (for + this will be the sum, whereas DUP simply duplicates the top value of the stack).

In all of the aforementioned operations an error is thrown if the stack does not contain a number of operands (values) expected by a word. This breaks execution of a word and the special on-error cleaning procedure is launched, after which Forth gets good chances to enter the interpretation mode again, waiting for new commands.

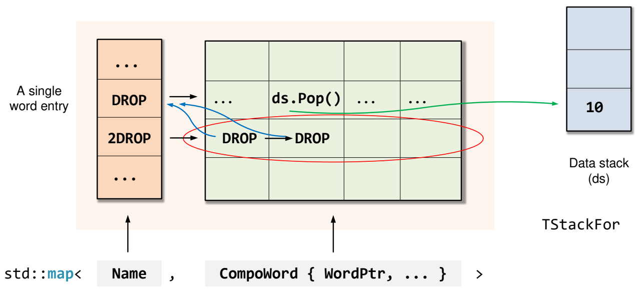


Fig. 2 Forth words are kept in the dictionary data structure, implemented as the C++ `std::unordered_map` with the key being any Name (`std::string`), while definitions are kept in a hierarchical *CompoWord* structures (based on `std::vector` containing pointers to already defined words and other procedures). Frequently, new words call words already present in the dictionary, such as 2DROP which two times calls DROP. Also, the words have an access to the data stack.

The second data structure characteristic to Forth is the word dictionary. Fig. 2 depicts structure of the Forth's dictionary in the *BCForth* implementation that holds definitions of the words, i.e. procedures. Each word is identified by its name.

As shown in Fig. 2, words can access the stack which holds the input and output parameters. Such definition is first scanned by the lexical tokenizer (

Fig. 8) to produce valid tokens, such as numeral literals and names of other words. The tokens are then parsed by the Forth compiler and, upon success, new definition is entered to the word dictionary

C. Hierarchy of Words

Fig. 9 depicts hierarchy of word defining classes, which has been already outlined in the general architecture shown in

Fig. 8. The roles of the classes in the TWord hierarchy are as follows.

- TWord is a template base class defining functional objects (functors) for the word hierarchy. The F template represents a class that defines all necessary data structures. Currently for this purpose TForth from Fig. 9 is used. Its main functionality, as well as of all its descendant, is *the action* defined by the virtual functional operator (). Naturally, invoking any Forth's word will be translated into calling the corresponding operator (). Hence, the entire TWord hierarchy can be seen as *the command design patter* [8].
- StructuralWord originates the sub-group of the structural words, such as the conditional statement IF … ELSE … THEN, the counted loop DO … LOOP and many more. However, StructuralWord is only a type-holder, whereas the most important function-holder is CompoWord.
- CompoWord defines *the composite design pattern* [8][5] to hold any sequence of Forth's words, also of the same type; such a recursive hierarchy allows composition of nested statements, such as DO … IF … THEN … LOOP, etc.
- IF is an example of *a composite to hold other composites* (similarly other objects in this sub-group). In this case it holds two branches: fTrueBranch representing a set of operation (another composite) chosen if, in the run-time, a condition (a value on the data stack) before the IF statement evaluates to true, and fFalseBranch which stores operations executed on the false condition.
- TValFor and TDataContainer are the two classes to represent a compiled-in value or a container of values, respectively. The type of the stored objects is given by the second template parameter V.
- StackOp – is a variadic template originating the suite of its specializations for defining data stack operations with various number of input and output parameters. For this purpose any function with 0, 1 or 2 input parameters, as well as 0 (void) or 1 return value, can be provided. These are supplied in the form of lambda functions passed to the constructor of the StackOp. Thanks to combination of this variadic template and the lambda functions dozens of stack operations are defined which otherwise required definition of separate classes in the TWord hierarchy [16].
- Dot, Comma, etc. – are examples of specialized system words.

As already mentioned, the key architectural assumption is expression of any Forth's word as the composite pattern, composed of other words, possibly also being composites, and so forth. Such a hierarchical structure provides a flexibility to define language constructions composed of structural statements nested to any depth

D. System Activity

In this section a brief overview of the activity of the Forth's interpreter and compiler are outlined.

The TForthInterpreter class was already outlined in Section (II.A). As shown in

Fig. 8, it is directly derived from the base TForth class. Since the main data structures TForthInterpreter inherits from its base, its key role is to execute words from the stream of text tokens, as outlined in the activity diagram shown in Fig. 3.

TForthCompiler is the last and the most complex class in the hierarchy in

Fig. 8. As mentioned, its main responsibility is parsing a word defining stream of tokens, contained in-between the **:** (colon) **;** (semicolon) symbols, and accordingly composing corresponding code of the newly created word.

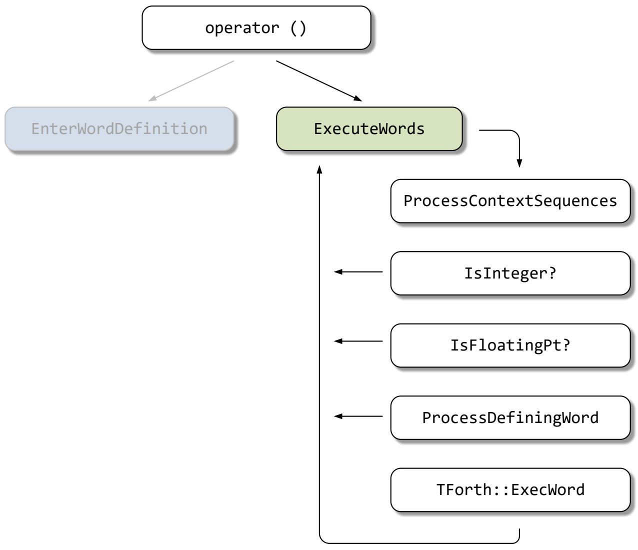


Fig. 3 UML activity diagram of the *TForthInterpreter*. The *ExecuteWords* executes a series of steps after which it is recursively called until the input stream of text tokens is emptied. The *EnterWordDefinition* is the compiler branch

If this operation is successful, the new word is placed in the Forth's dictionary, from which it can be invoked by the interpreter, as well as used in definitions of future words, again processed by the compiler, and so on. Its activity diagram is shown in Fig. 5.

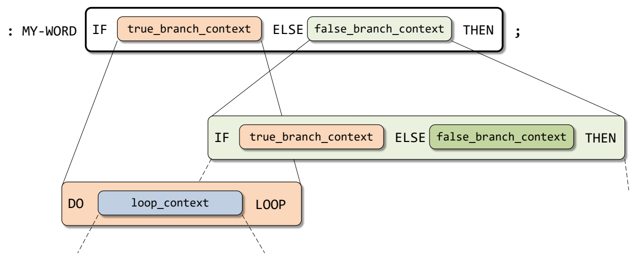


Fig. 4 Changing context concept. Each word, as well as each branch of a structural construction such as `IF` … `ELSE` … `THEN`, `DO` … `LOOP`, etc. has its own context implemented with its own composite *CompoWord*. Each such object has links to other words, also other *CompoWords*, and so on. The entire structure is parse by a successive recursive call to the parsing procedure

Fig. 4 depicts an example of the nested structure constructions. The key observation is that each sub-branch opens *a new context*, which can be treated as a separate sub-word construction, and so on. This creates a hierarchical composition which can be processed in a recursive manner – at each level the sub-branch is processed *independently* as a separate sub-word and in *its own context*.

Finally, the remaining Forth words are defined in separate Forth modules. These are special classes (Fig. 9) to enter word definitions for various domains, such as floating-point, string & memory processing, and from different sources, such as hard coded, string or file stream. The pure abstract root `TForthModule` starts their class hierarchy

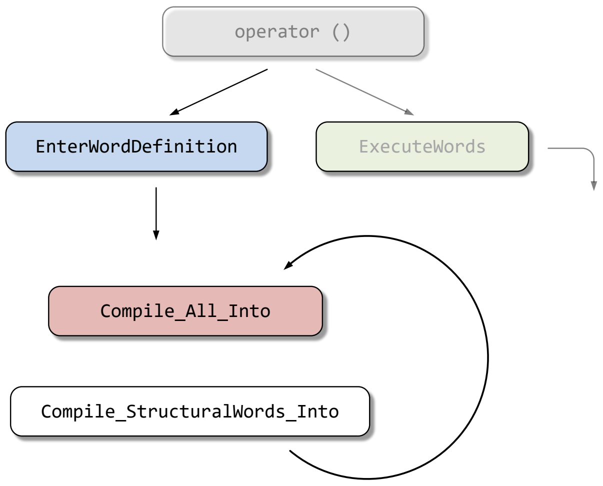


Fig. 5 Activity diagram of *TForthCompiler*. *EnterWordDefinition* implements the principal functionality of the Forth compiler – parsing word defining stream of text tokens and constructing the corresponding implementation. To process structural statements, which can be nested to any depth, each structural statement enters into a new context represented by a separate composite object. Processing is done by recursive calls of the *Compile_All_Into* function until the entire defining stream is processed. *Compile_StructuralWords_Into* processes the structural statements such as conditional IF, DO, etc. in interaction with the structural words of the *TWord* hierarchy

III. FORTH ENDOWED WITH THE COROUTINES

Existence and roles of functions, or routines, in computer programs are ubiquitous and well known. However, there is a special type of a routine called a coroutine, which can suspend its execution preserving its state to be resumed later [9], as shown in Fig. 6.

For such functionality coroutines need to have associated memory to store local data and the resumption point. In this respect there are two groups: stackfull and stackless coroutines. Modern C++20 provides the framework and mechanisms for the latter [7][3]. That is, they suspend execution by returning to the caller and the data that is required to resume execution is stored separately from the stack. This allows for sequential code that executes asynchronously e.g. to handle non-blocking I/O without explicit callbacks, allows for the so called lazy-computations e.g. to generate infinite series of values, but most of all it allows for cooperative multitasking purely on the Forth platform. The latter is very useful feature especially on small and resource constraints platforms that nevertheless require the kind of multitasking [2]. Forth built in coroutines allow for such operation in much more lightweight way compared to the preemptive multitasking. Hence, coroutines are a unique feature of *BCForth*.

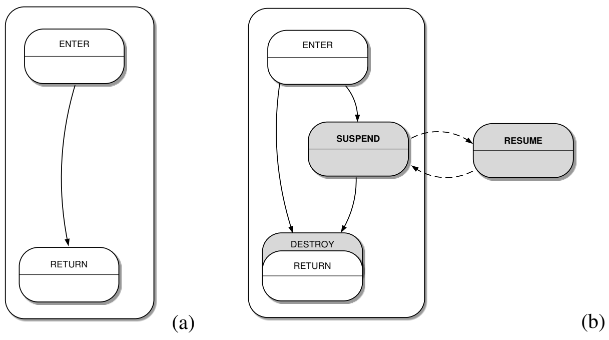


Fig. 6 State diagram of an ordinary routine (a) and a coroutine (b). The latter can also suspended, preserving its state, to be resumed later. This allows for async operations or lightweight threading

The main proposed idea is to introduce new Forth words, which will operate as the stackless co-routines (however, they have an access to the Forth's stack). For this purpose a new word named **CORO** is proposed, which if put after a word's definition, makes it a coroutine (this is similar to the IMMEDIATE post word). In the Forth's nomenclature we propose to call them *co-words*. For instance, the following defines the word FIBER_0 that does XOR of the first cell in a buffer BUF, then reads the second cell from that buffer and pushes it onto the Forth's stack

```
: FIBER_0 BUF @ 0xAB XOR BUF !   0x02 BUF + @
                    ;   CORO  [155]
```

However, CORO with the optional parameter [155] makes it a Forth's coroutine that toggles some bits, and suspends after 155 ms, or terminates if the second cell in BUF is not 0. This is possible thanks to the CORO_Frame and FiberTask<T> C++ coroutine structure that operates as a wrapper around any WorkerWord, such as FIBER_0, in our example, while time_slice becomes 155. An outline of CORO_Frame looks as shown in Algorithm 1. FiberTask<T> in line [1] is a structure with the nested class promise_type, as required by the C++20 framework [7]. On the other hand, GetTimePoint in lines [3,5,10] does the time management, resulting with the suspend via co_await in [9].

The next proposed new word is **COYLD** (from *co-yield*) that suspends a given word leaving its value on the top of the Forth's stack. Thanks to this, the value generating words can be defined. With its help the CO_RANGE word has been created which, upon each call, generates and pushes onto the Forth's stack consecutive values from a predefined range. For example 10 20 2 CO_RANGE creates a generator of values 10 to 20 with step of 2. Then each call to CO_RANGE leaves 12, 14, …, 18 on the stack.

The last from the proposed words is the **COSUS** (from *co-suspend*). It suspends a Forth's word at its point of call, from which that word will resume if called again (naturally, an 'ordinary' Forth word would start from the beginning).

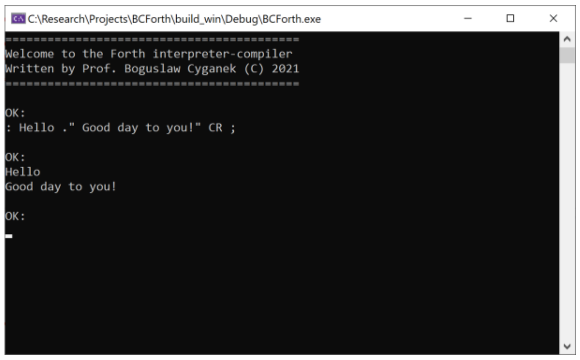
IV. System Deployment and Experiments

The complete C++ implementation of *BCForth* with exemplary Forth programs is available from the GitHub [16]. This is *a multi-platform header only library* aimed at Linux/Unix, Windows, and MacOS. It was successfully built and deployed on the following platforms:

1. PC computer with Linux Ubuntu 18.4 and 20.4, run on laptop Dell Precision 7710. Compiled with the gcc version 10 and 11. The latter allows co-routines.
2. PC computer with Windows 10 run on laptop Dell Precision 7760. Compiled with the Microsoft Visual C++ 2019 v. 16.9.2, as well as MV 2022 v. 17.2.6.
3. Embedded system NodeMCU v3 with the 32-bit RISC ESP8266 microprocessor [17][18], controlled by the 80 MHz clock (based on Tensilica Diamond Standard 106Micro architecture). The system equipped with the 32 KB instruction memory and 80 KB data RAM. The system contains built-in Wi-Fi, 10 GPIO ports, ADC converter and USB-UART CH340 link, allowing also external programming. Built in the PlatformIO Arduino equipped with the gcc version 10. This is an example of a IoT tiny platform with its own system but yet without co-routines.

Fig. 7(a) depicts the NodeMCU board, while *BCForth* run in the interactive mode in the terminal window is shown in Fig. 7(b).

(a)


C:\Research\Projects\BCForth\build_win\Debug\BCForth.exe

```
==========================================
Welcome to the Forth interpreter-compiler
Written by Prof. Boguslaw Cyganek (C) 2021
==========================================

OK:
: Hello ." Good day to you!" CR ;

OK:
Hello
Good day to you!

OK:
```

(b)

Fig. 7 Embedded system NodeMCU v3 with the 32-bit RISC ESP8266 microprocessor 32KB+80KB RAM, 80 MHz clock (a). *BCForth* running in the terminal window (b)

Although both Linux and Windows 10 allowed for a complete implementation, special attention deserves the third platform which is a tiny NodeMCU v3 embedded systems with only the 32 KB instruction memory and 80 KB data RAM.

Nevertheless, with some minor modifications, it was also possible to run *BCForth*. This shows that despite C++ the footprint of the *BCForth* can be as small as to fit to the small (and cheap) embedded platforms and/or IoT systems.

However, even more important is fast time (approx. three weeks) of *BCForth* system tuning to the new platform done by Mr. W. Gałecki & Ms. K. Rapacz, students of the 1st year of the graduate studies Electronics & Telecommunication, as a completion of their project to the Systems Design and Modeling Methodologies classes under author's supervision at the AGH University of Science and Technology. This proves that *BCForth* implementation is straightforward for all persons with at least medium competitions in the modern C++ programming, as well as that it can be easily deployed on similar tiny embedded frameworks. This also adds the teaching aspect of the presented system and, hopefully, can be used with educational and technical benefits by a broader group of students and enthusiasts of embedded systems

V. Conclusions

In this paper a novel and free Forth language platform *BCForth* [16], aimed at embedded systems of various sizes, is proposed. The main advantage of Forth is coexistence of the compiler and interpreter that allows for direct communication with a user and easy composition of new words (procedures). Unique *BCForth* features are as follows: (i) modular C++20 based implementation, (ii) implementation of coroutines for *async* operations and lightweight multithreading, (iii) educational/teaching platform for students of electrical engineering faculties. Envisioned things to do are: (i) modules with new words (e.g. file operations, graphics, etc.), (ii) GUI for Forth development and debugging, (iii) auto setup for easier deployment on the limited footprint platforms. We are deeply convinced that this novel implementation of Forth will be beneficial for embedded systems, as well as in education and further popularization of Forth and C++.

References

[1] Brodie L.: Thinking Forth. A Language and Philosophy for Solving Problems, Creative Commons, 2004.

[2] Belson B., W. Xiang, J. Holdsworth and B. Philippa, C++20 Coroutines on Microcontrollers – What We Learned, IEEE Embedded Systems Letters, vol. 13, no. 1, pp. 9-12, 2021.

[3] Belson B., et al.. A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems. ACM Trans. Embed. Comput. Syst. 18/3, 2019.

[4] Conklin E.K., Rather E. D.: Forth Programmer's Handbook, FORTH Inc. 2010.

[5] Cyganek B.: Introduction to Programming with C++ for Engineers. Wiley-IEEE Press, 2021.

[6] Dunkels A., Schmidt O., Voigt T., Muneeb A. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. 4th international conference on Embedded networked sensor systems (SenSys '06). ACM, 29–42, 2006.

[7] https://en.cppreference.com/w/cpp/language/coroutines

[8] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[9] Knuth D. E. The art of computer programming, Vol. 1: Fundamental algorithms (3rd. ed.), Addison-Wesley, 1997.

[10] Pelc S.: Programming Forth. MicroProcessor Engineering Limited, 2005.

[11] Rather E. D., Colburn Donald R., and Moore Charles H.: The evolution of Forth. History of programming languages-II. Assoc. for Comp. Machinery, New York, USA, 625–670, 1996.

[12] Rather E. D.: Forth Application Techniques, 6th edition, FORTH Inc. 2019.

[13] https://en.wikipedia.org/wiki/Charles_H._Moore#cite_note-2

[14] https://forth-standard.org/

[15] https://theforth.net/

[16] https://github.com/BogCyg/BCForth

[17] https://en.wikipedia.org/wiki/ESP8266

[18] https://en.wikipedia.org/wiki/NodeMCU

[19] https://en.wikipedia.org/wiki/Forth_(programming_language)

[20] https://gforth.org/

[21] https://www.forth.com/

[22] http://git.annexia.org/?p=jonesforth.git;a=summary

[23] https://awesomeopensource.com/projects/forth

[24] https://cppreference.com

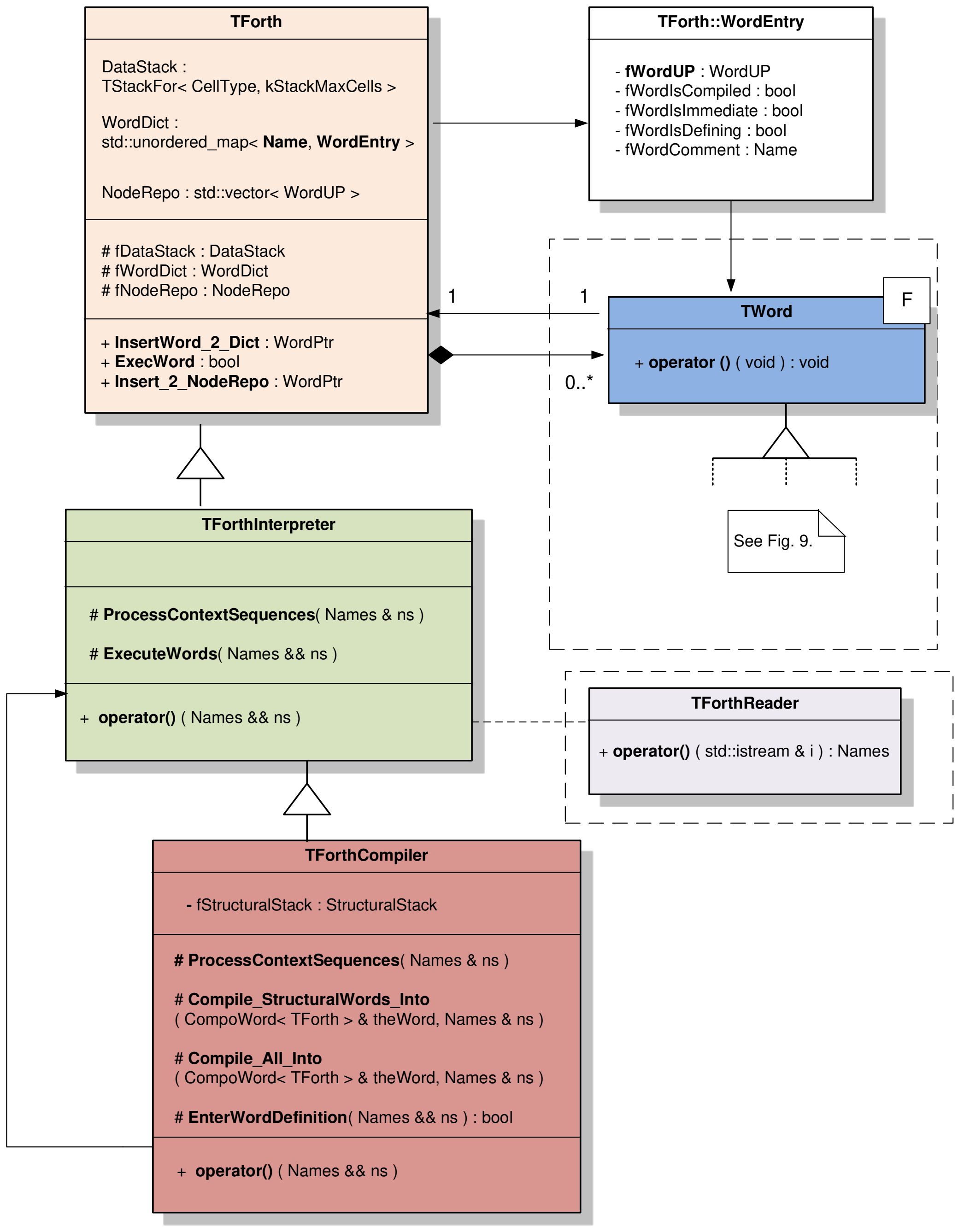

TForth
DataStack : TStackFor< CellType, kStackMaxCells >
WordDict : std::unordered_map< Name, WordEntry >
NodeRepo : std::vector< WordUP >
# fDataStack : DataStack
# fWordDict : WordDict
# fNodeRepo : NodeRepo
+ InsertWord_2_Dict : WordPtr
+ ExecWord : bool
+ Insert_2_NodeRepo : WordPtr
TForth::WordEntry
- fWordUP : WordUP
- fWordIsCompiled : bool
- fWordIsImmediate : bool
- fWordIsDefining : bool
- fWordComment : Name
1
1
0..*
TWord
F
+ operator () (void) : void
See Fig. 9.
TForthInterpreter
# ProcessContextSequences(Names & ns)
# ExecuteWords(Names && ns)
+ operator() (Names && ns)
TForthReader
+ operator() (std::istream & i) : Names
TForthCompiler
- fStructuralStack : StructuralStack
# ProcessContextSequences(Names & ns)
# Compile_StructuralWords_Into (CompoWord< TForth > & theWord, Names & ns)
# Compile_All_Into (CompoWord< TForth > & theWord, Names & ns)
# EnterWordDefinition(Names && ns) : bool
+ operator() (Names && ns)

Fig. 8 Architecture of the *BCForth* system. The main branch is composed of three classes: *TForth*, *TForthInterpreter* and *TForthCompiler*. These use the hierarchy of word nodes, originated from the *TWord* base (*Names* denotes a collection of text tokens). The input/output operations are interfaced by the *TForthReader* class, which transforms an input stream (terminal or a file) to a series of tokens.

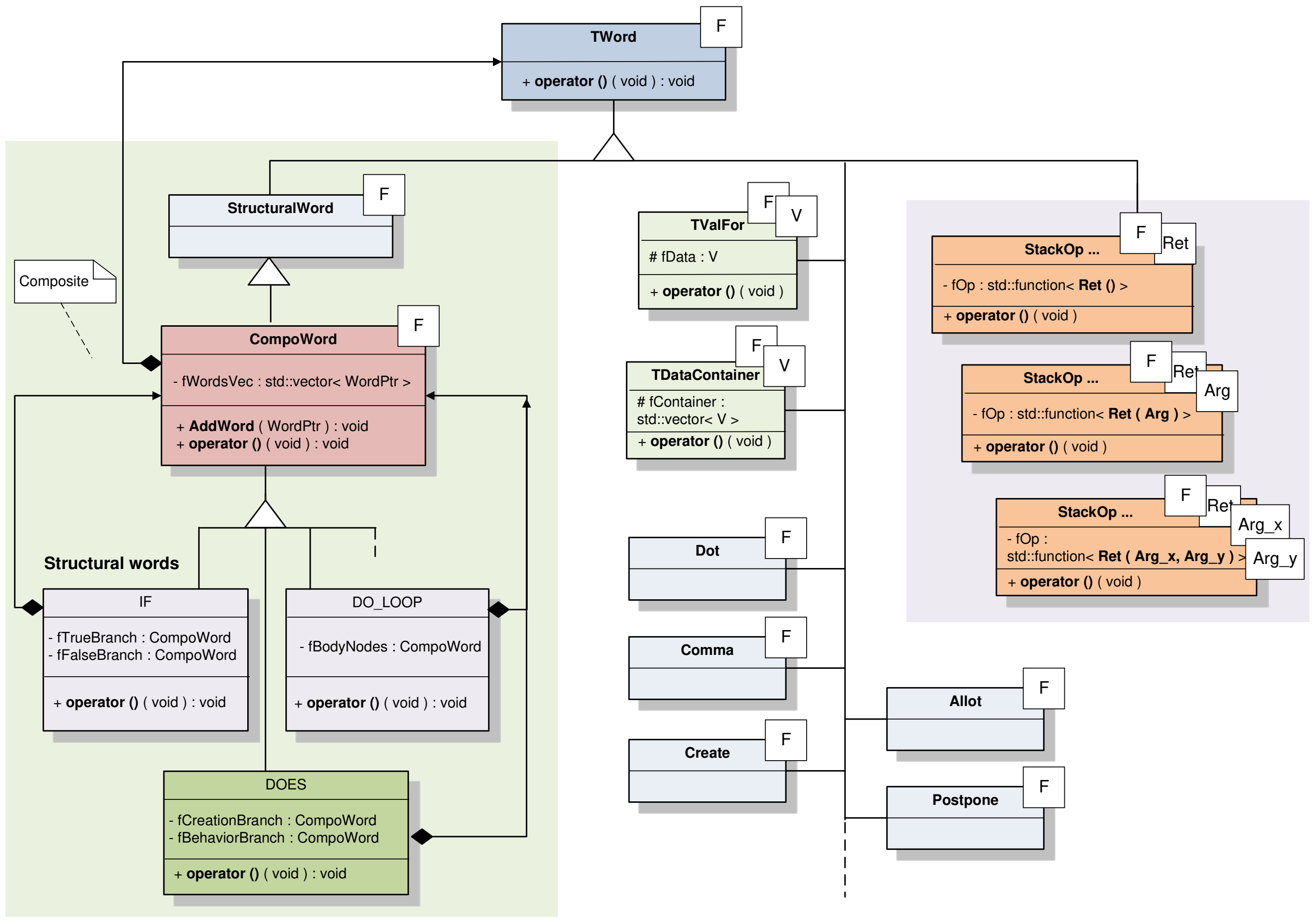

TWord
F
+ operator () (void) : void
StructuralWord
F
Composite
CompoWord
F
- fWordsVec : std::vector< WordPtr >
+ AddWord (WordPtr) : void
+ operator () (void) : void
Structural words
IF
- fTrueBranch : CompoWord
- fFalseBranch : CompoWord
+ operator () (void) : void
DO_LOOP
- fBodyNodes : CompoWord
+ operator () (void) : void
DOES
- fCreationBranch : CompoWord
- fBehaviorBranch : CompoWord
+ operator () (void) : void
TValFor
F
V
# fData : V
+ operator () (void)
TDataContainer
# fContainer : std::vector< V >
+ operator () (void)
Dot
Comma
Create
Allot
Postpone
StackOp ...
Ret
- fOp : std::function< Ret () >
+ operator () (void)
StackOp ...
Arg
- fOp : std::function< Ret (Arg) >
+ operator () (void)
StackOp ...
Arg_x
Arg_y
- fOp : std::function< Ret (Arg_x, Arg_y) >
+ operator () (void)

Fig. 9 Hierarchy of classes defining the Forth words. The composite design pattern, implemented with the *CompoWord* branch, constitutes the main building block of all words registered to the Forth dictionary. The next branch constitute system specific words, such as *Dot* or *Create*. The *StackOp* branch, implemented as a variadic template and its specializations, is responsible for majority of the data stack operations, such as arithmetic and logical operations.

Algorithm 1. Scheme of the *CORO_Frame* routine.

```
template < typename T, auto time_slice, auto WorkerWord >
FiberTask< T > CORO_Frame ( auto worker_load ) {
    auto tp0 = GetTimePoint(); // coroutines suspend on time elapsed
    for( ;; ) {
        // embed/call worker word WorkerWord with worker_load
        // if done, then co_await or break for co_return

        if( GetTimePoint() - tp0 > time_slice ) {
            co_await std::suspend_always{}; // suspend if time elapsed
            tp0 = GetTimePoint();
        }
    }
    co_return -1;
}
```