DOI: 10.32703/2617-9040-2025-45-1

UDC 004.4

Nuru Dashdamirli[1]

[1]Doctoral student, Azerbaijan Technical University, pr. H. Javid 25, Baku, Azerbaijan, AZ 1148. ORCID: https://orcid.org/0009-0009-0545-7855.

* **Corresponding author**: nurudashdamirli@gmail.com.

Cross-platform development for microcontrollers: design of a virtual machine based portable programming language

The fundamental role of microcontrollers in embedded systems and the Internet of Things (IoT) environments necessitates efficient software development approaches. Resource limitations of microcontrollers, the complexity of low-level programming languages, and the challenges of implementing multitasking slow down the development process considerably. Additionally, the diversity of the microcontroller landscape creates substantial barriers to code portability, leading to increased development time to support different hardware platforms. This paper presents the design of a virtual machine-based programming approach to enable cross-platform development for microcontrollers. The proposed portable programming language integrates with a custom virtual machine, Mico8-Chip, to suit modern microcontroller applications. This allows intuitive control over peripherals and built-in support for concurrent execution. The provided abstraction layer significantly improves code portability and accelerates development by isolating application logic from underlying hardware specifics. The primary purpose of this work is to address the fragmented microcontroller ecosystem and the challenges of low-level programming by introducing a unified and portable development solution.

Keywords: *microcontroller, embedded systems, virtual machine, programming, bytecode, concurrency, Internet of Things.*

Introduction. The increasing need for smart and independent devices within the Internet of Things (IoT) ecosystem has caused a rise in the popularity of microcontroller-based systems. Microcontrollers are now widely used in many types of applications, including automation systems, wearable technologies, home appliances, medical devices, industrial equipment, and robotics. Their key advantages, such as low power use and small size, make them very important for many applications, including various IoT-based technologies [1]. However, using traditional programming methods for microcontrollers has some limits. While the C programming language and similar low-level languages are still commonly used in microcontroller systems and IoT networks, they require careful attention from programmers about important things like memory management, software security, and multitasking [2]. Although other high-level programming languages like MicroPython offer an easy-to-use way to develop software, they often have problems with program execution speed and how they use available memory, so they are not suitable for use in microcontroller environments where resources are limited [3].

A significant challenge in the domain of microcontroller programming comes from the characteristic limitations in hardware resources. These limitations typically evident as restricted Random Access Memory (RAM), small flash memory capacity, and reduced processing power compared to general-purpose computing systems. As a result, developers are required to write highly optimized code to guarantee efficient resource utilization. Furthermore, implementing complex functionalities like multitasking becomes considerably challenging due to the scarcity of available resources. When an embedded system demands concurrent execution of multiple tasks, such as processing data from sensors, managing communication protocols, and handling user interface interactions through a control panel, the complexity of software development increases substantially. The necessity of ensuring precise memory management and the efficient scheduling of these concurrent tasks inevitably leads to delays in software development cycle.

In this paper, a new approach is proposed that integrates a portable programming language specifically designed for the microcontroller environment with a virtual machine to overcome the previously mentioned difficulties. The primary purpose of this work is to develop a comprehensive solution for cross-platform microcontroller programming that balances ease of development with resource efficiency. To achieve this, this research focuses on creating a portable, high-level programming language with a C-like syntax for microcontroller applications to enhance developer productivity and code readability. A compiler is also developed to translate this language into compact bytecode executable by the Mico8-Chip virtual machine. Additionally, the approach integrates built-in concurrency support within the language and virtual machine to simplify multitasking in embedded systems without external dependencies, demonstrating improved code portability across microcontroller architectures through the virtual machine abstraction layer. The target virtual machine, Mico8-Chip, is a redesigned version of the CHIP-8 architecture, modified to meet the requirements of modern microcontroller systems [4]. CHIP-8 is an interpreted programming language and virtual machine developed in the 1970s for use on microcomputers. CHIP-8 which was designed for the development of interactive programs and games [5]; however, it is still being used in educational environments [6]. The proposed programming language is designed to provide control of microcontroller peripherals, handling of input/output operations, and concurrent program execution. One of the important features of this approach is its built-in multitasking environment, which allows multiple applications to run simultaneously without the need for complex scheduling mechanisms.

The syntax of the proposed programming language is intentionally designed as a simplified derivative of C syntax. The motivation for this similarity is based on C's extensive application in microcontroller systems, thereby minimizing the necessity for programmers to get proficient in an entirely new syntax. Tasks developed in this programming language are converted into special bytecode and executed by the virtual machine. The proposed approach both provides easier control of the hardware in microcontroller systems and increases the portability of the code. Since the same bytecode can be used on different microcontroller platforms without changes, the efficiency and speed of software development increase significantly.

The problem statements. Despite the widespread adoption of microcontrollers in various applications, the current landscape of embedded systems development faces significant issues that decrease its efficiency. One major challenge lies in the inherent fragmentation of the microcontroller ecosystem. There are different microcontroller architectures and vendors, each often requiring specialized toolchains, programming languages, and development environments. This diversity makes it difficult for developers to port their software across different hardware platforms, leading to significant increase of effort and development costs when projects need to scale or adapt to new hardware. Valuable engineering resources are frequently spent on platform-specific optimizations and re-writes rather than on developing core application logic and new features.

Furthermore, the widespread use of low-level languages like C, while offering fine-grained control over hardware, introduces a steep learning curve and requires careful attention to low-level details. This can be particularly challenging for developers entering the embedded systems domain or those working on projects with tight deadlines. The complexity associated with manual memory management, complicated peripheral configurations, and the need to handle hardware-specific nuances can lead to increased development time and a higher likelihood of introducing subtle and hard-to-debug errors. This situation is further compounded by the growing sophistication of embedded applications, which often demand more complex functionalities and interactions, pushing the limits of what can be efficiently managed with traditional low-level programming paradigms within resource-constrained environments.

While higher-level languages offer a potential solution for simplifying development, their inherent overhead in terms of execution speed and memory footprint often renders them unsuitable for the resource-constrained nature of many microcontroller platforms. This creates a trade-off between ease of development and performance efficiency. Developers are frequently forced to choose between the faster development cycles offered by higher-level languages and the necessary performance and resource optimization achievable with lower-level languages, often leading to compromises that may not fully satisfy the requirements of the target application. This restricts the accessibility of advanced programming concepts and tools within the embedded systems domain, potentially preventing the development of more sophisticated and feature-rich applications on microcontrollers.

The increasing demand for embedded systems to perform multiple tasks concurrently also presents a significant challenge in the context of limited resources. Implementing robust and efficient multitasking capabilities on microcontrollers using traditional methods often requires complex real-time operating systems (RTOS) or custom scheduling mechanisms. This adds another layer of complexity to the development process, demanding specialized knowledge and potentially consuming significant amounts of the already scarce memory and processing power. The need for effective concurrency management becomes even more critical in IoT applications, where devices must interact with their environment and communicate with other systems in a timely and reliable manner.

In essence, the current landscape of microcontroller development is characterized by a tension between the need for efficient resource utilization, the growing complexity of applications, and the aspiration for faster development cycles and increased code portability. These limitations collectively delay innovation, increase development costs, and complicate the creation of sophisticated and adaptable embedded systems. This paper aims to address these fundamental challenges by proposing a unique approach centered around a virtual machine-based portable language specifically designed to bridge the gap between development ease and resource efficiency in the microcontroller domain. This process involves developing a high-level programming language with a C-like syntax and an accompanying compiler that compiles this source code to bytecode for the targeted virtual machine, Mico8-Chip. This approach abstracts hardware complexities and simplifies application logic development while allowing to integrate native concurrency without relying on external scheduling mechanisms. Furthermore, it improves code portability and accelerates development cycles compared to traditional methods.

Related work. Overcoming the characteristic limitations of resource-constrained embedded systems to enable the use of higher-level programming languages has been a persistent focus of research, frequently involving the development of dedicated virtual machines. BIT and PICBIT were two of the early approaches in this domain that sought to bring the benefits of the Scheme programming language to microcontrollers. The development of BIT detailed a highly compact implementation of the Scheme programming language, focusing on achieving space efficiency and real-time garbage collection through a compact byte-code compiler and runtime system [7]. This work demonstrated the feasibility of running meaningful Scheme programs on microcontrollers with limited memory, with the limitations of relatively lower execution speed and the omission of error-checking. Further investigation into Scheme for resource-constrained devices was conducted with PICBIT, a specific adaptation of a Scheme system for the PIC microcontroller family. PICBIT utilized a 24-bit object representation and incorporated a byte-code interpreter and optimizing compiler [8]. The PICOBIT system was later modeled after these early virtual machine approaches, which included an optimizing C compiler named SIXPIC [9].

Addressing the challenge of executing Java applications on resource-constrained microcontrollers, the Darjeeling Virtual Machine was designed to provide a memory-efficient solution for environments such as wireless sensor networks. The primary objective was to design and implement a virtual machine that supports a subset of Java functionality suitable for microcontrollers. Darjeeling introduced a system comprising a runtime and offline tools that implement a modified Java Virtual Machine (JVM) designed for minimal memory usage. Key innovations of this approach include the selective omission of certain JVM features, a compact static linking model using infusions, and a unique memory model. The system also uses a basic mark-and-sweep garbage collector and supports preemptive multithreading.

Performance evaluations on AVR128 and MSP430 microcontrollers demonstrated effective operation with limited memory resources and execution speeds up to 70,000 JVM instructions per second [10].

Efforts to overcome the constraints of microcontroller environments have also led to the development of alternative JVM designs. The TakaTuka JVM specifically designed for microcontrollers with limited RAM, storage, and processing power. The TakaTuka JVM introduces techniques such as offline garbage collection, which allows deallocation of reachable but unused objects, and a variable slot size scheme for efficient memory management. The system also implements extensive Java binary optimizations, including bytecode compaction and constant pool optimizations, and utilizes a direct threading mechanism for bytecode interpretation. The main limitation of this approach is the potential restriction in dynamic adaptability due to the reliance on offline analyses for memory management [11].

Methods for improving microcontroller programming were also explored through unique virtual machine approaches, such as the OCaLustre system, which was developed as a synchronous extension to the OCaml programming language. The system works along with OCaPIC, a virtual machine designed to run OCaml bytecode on PIC microcontrollers. The primary aim was to enable safer, more expressive, and concurrent software for microcontrollers by integrating a higher-level programming model with efficient memory use. The OCaLustre system introduces nodes for deterministic concurrent tasks and features a compilation model that translates these nodes into sequential OCaml functions. Benchmarking analyses demonstrated OCaLustre's advantages in both static and dynamic resource efficiency compared to other concurrency models [12]. Later work introduced a generic virtual machine approach for programming microcontrollers with the OMicroB project. Building on earlier findings with OCaPIC, this research focused on enhancing the safety and portability of microcontroller programming by utilizing the OCaml language and the OMicroB VM, which is implemented in C. This approach involves compiling OCaml to bytecode, optimizing it with the "ocamlclean" tool, and then converting it into C code using a custom tool called "bc2c". Key findings demonstrated that the OMicroB VM could effectively execute OCaml programs on resource-constrained microcontrollers, such as the ATmega328P used in Arduino Uno boards [13].

Recognizing the potential of WebAssembly beyond web environments, its application was explored as an alternative programming platform for microcontrollers. The core objective was to enhance the efficiency, safety, and ease of programming for microcontrollers. WARDuino, a virtual machine built upon WebAssembly and improved with features such as live code updates, remote debugging, and modular runtime configuration was created and examined. The architecture of WARDuino integrates WebAssembly and Arduino functionalities, exposing hardware capabilities through specific modules. Benchmarks demonstrated that WARDuino achieved significantly faster execution compared to Espruino, a JavaScript interpreter for microcontrollers, while maintaining smaller binary sizes [14].

Researchers have also explored the feasibility and performance of implementing minimal virtual machines on microcontroller-based IoT devices. The research focused on evaluating two virtual machine architectures: a stack-based VM using WebAssembly and a register-based VM built upon extended Berkeley Packet Filters, called rBPF. Their goal was to identify solutions for process isolation and software modularity with minimal resource overhead. Evaluations demonstrated that the rBPF interpreter had a significantly smaller flash memory footprint compared to WebAssembly. However, rBPF exhibited longer execution times. The study highlighted rBPF's advantages in memory efficiency and its potential for process isolation on microcontrollers without requiring hardware memory protection units [15].

The complexity and safety issues in microcontroller programming was addressed by proposing SenseVM, a virtual machine that implements a higher-order concurrency model. The principal objective of the research was to streamline the programming of reactive and concurrent embedded systems by introducing a high-level message-passing interface based on functional programming. SenseVM introduces a concurrency model where synchronous operations are encapsulated as first-class values called "events", which can be composed using various combinators. The virtual machine, SenseVM, is an interpreter built upon the Categorical Abstract Machine (CAM) architecture and includes a low-level bridge that interfaces with Zephyr OS for hardware-agnostic interaction. Evaluations demonstrated code portability across different microcontroller platforms, with power consumption comparable to optimized C implementations, although response times revealed a performance overhead [16].

Summing up the above works, the field of microcontroller programming has seen a significant push towards adopting higher-level languages and virtual machine technologies to address the limitations and complexities associated with traditional low-level approaches. Various studies have explored the implementation of languages like Scheme, Java, OCaml, and WebAssembly through custom-made virtual machines, each addressing the unique challenges of memory efficiency, performance, and concurrency on microcontrollers. These efforts have introduced innovative techniques in memory management, garbage collection, code optimization, and concurrency models. Different architectures have been investigated, often coupled with specialized compilers and optimization tools. While each approach presents its own strengths and limitations in terms of performance, memory usage, and portability, the overarching goal is to simplify microcontroller programming, enhance developer productivity, and enable the development of more sophisticated and safer applications for embedded systems. The main contributions of this paper are summarized as follows:

• A minimalist virtual machine-based architecture designed to enable cross-platform development for embedded systems.

• The design and implementation of a portable language specifically for microcontrollers, enabling code reuse across different hardware platforms.

• An approach that aims to balance the performance of low-level programming with the abstraction of high-level languages in the context of embedded systems.

Proposed programming language and compiler architecture. Microcontrollers typically have limited processing power and RAM resources compared to microprocessors. Therefore, it is not advisable for applications targeting microcontroller environments to carry out intensive calculations and allocate large chunks of memory. Understanding these limitations is important for selecting the right microcontroller for a specific application and designing systems that can operate effectively within these limitations. Microcontrollers are essential in devices that require low power consumption, especially for battery-powered wearable technologies. The proposed portable programming language was designed taking into account the limitations often encountered in microcontroller environments. The syntax and keywords of the language are similar to the C programming language, simplified to enable compilation into minimalistic virtual machine systems, such as Mico8-Chip, and to ensure low resource usage. Therefore, functionalities such as dynamic memory allocation and the definition of complex data models have not been integrated into the language.

To run the developed programs in the microcontroller environment, it is first necessary to convert the human-readable source code into bytecode that can be executed by a Mico8-Chip virtual machine. For this, a compiler designed for the proposed programming language is used. The compiler's conversion of source code into bytecode is a multi-stage process, which determines whether the program code is syntactically and semantically correct before execution. The compilation process goes through the following stages:

1. Reading source code. Human-readable source code is read from a file. This code is written in the syntax of the proposed language.

2. Lexical analysis. Source code is broken down into tokens. In programming languages, tokens are the smallest meaningful units processed by compilers or interpreters. Examples of tokens include keywords, operators, identifiers, and numbers.

3. Syntax analysis. Tokens are checked against the grammatical rules of the language. A syntax tree is constructed, representing the hierarchical structure of the code. This stage ensures that the code is

syntactically correct.

4. Semantic analysis. The code is checked for semantic correctness. This includes checking types, determining whether variables and functions are declared before they are used, and ensuring that function calls are correct.

5. Creation of intermediate representation code. After semantic analysis, the syntax tree is converted into intermediate representation code. Intermediate representation code is a form that can be more easily optimized and manipulated than source code.

6. Optimization. Various optimization techniques are used to improve the execution performance. This stage may include eliminating code duplication, removing unreachable code, and optimizing loops.

7. Bytecode generation. Bytecode is a low-level, platform-independent code that can be executed by the proposed virtual machine. Bytecode generation is performed based on the intermediate representation code.

All errors and warnings encountered at various stages of the compilation process are logged by the compiler. Errors result in the compilation process failing to complete, while warnings reveal potential problems in the source code. This helps in identifying and fixing problematic parts of the code. This systematic approach ensures that the source code is validated and transformed into an efficient bytecode format that is suitable for execution in the virtual machine environment (Fig. 1).

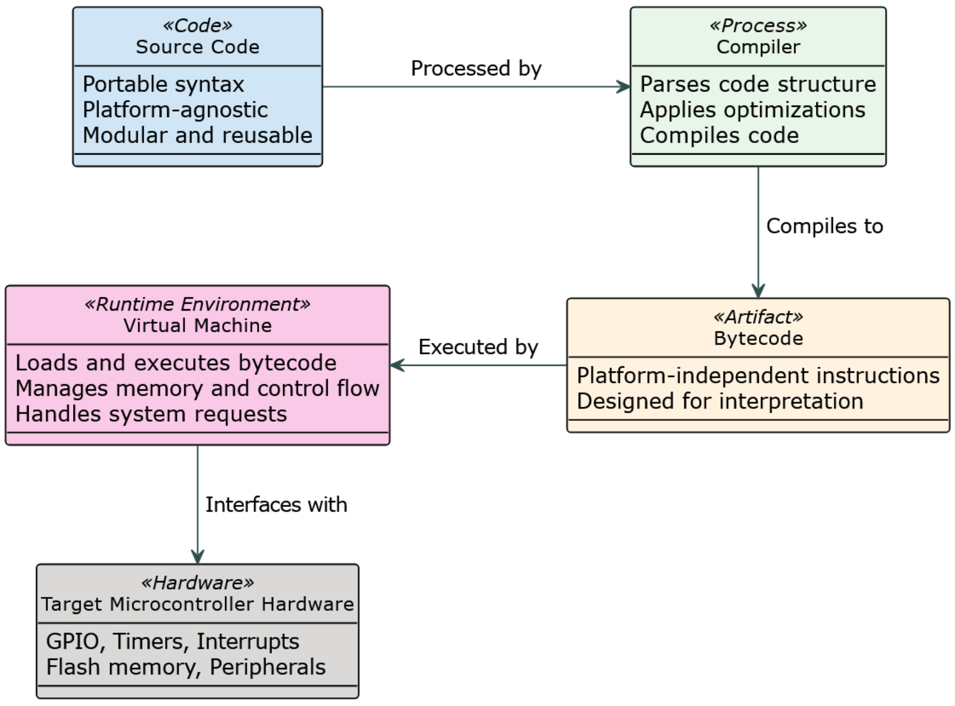


Fig. 1. **Execution flow for the proposed portable programming language for microcontrollers**

Tokens and lexical structure. The initial phase of compilation transforms unstructured text source code into a sequence of meaningful symbols that can be processed by following compiler stages. These symbols are tokens, which act as blocks of source code and are categorized based on their role in the syntax of the proposed language. The process of breaking down source code into tokens is called tokenization or lexical analysis and is the initial step in the compilation or interpretation process where the source code is scanned and tokens are identified.

The operational process of lexical analysis involves the compiler's lexer systematically reading the source code, character by character. As it reads, it applies the defined lexical structure rules for grouping characters into lexemes and assign them the appropriate token types. For instance, upon encountering the sequence 'i', 'f', the lexer recognizes it as the keyword "if" rather than two separate identifier tokens. This recognition is based on the defined keywords list and the longest match principle. Whitespace and comments encountered during this scan are recognized and filtered out. If a sequence of characters is encountered that does not fit to any valid token pattern, a lexical error is reported. The output of this phase is a list of tokens (Fig. 2).

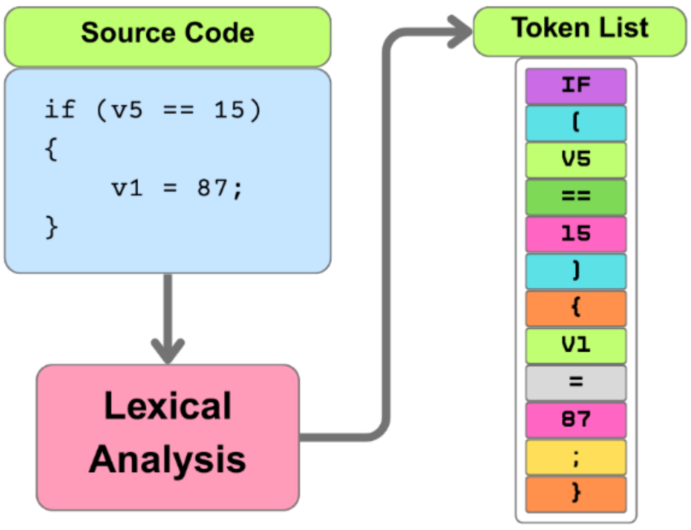


Fig. 2. **Tokenization process of the source code**

The types of available tokens are keywords, identifiers, literals, operators, and comments. Identifiers are names given to tokens such as variables, functions, and labels. Identifiers must follow certain naming conventions, starting with a letter or underscore and consisting of letters, digits, or underscores. Keywords are words with special meanings, such as "if", "else", "while", and "for", which cannot be used as identifiers. Literals represent constant values that are directly represented in the code, like integer and string literals. Operators are tokens that express operations on operands, and include arithmetic, comparison, and logical operators. There are also tokens that provide structure to the language but have no independent meaning, such as commas, semicolons, parentheses, curly braces, and square brackets. Comments are single-line or multi-line annotations that allow developers to make notes within the source code, although they do not affect the execution of the program.

The proposed language has a similar token structure to C, with the elimination of tokens that are not implemented due to different constraints of the targeted virtual machine, Mico8Chip (Fig. 3). While this limits the use of syntactic patterns, it is necessary for achieving the minimal memory and storage footprint for the virtual machine implementation and resulting executable.

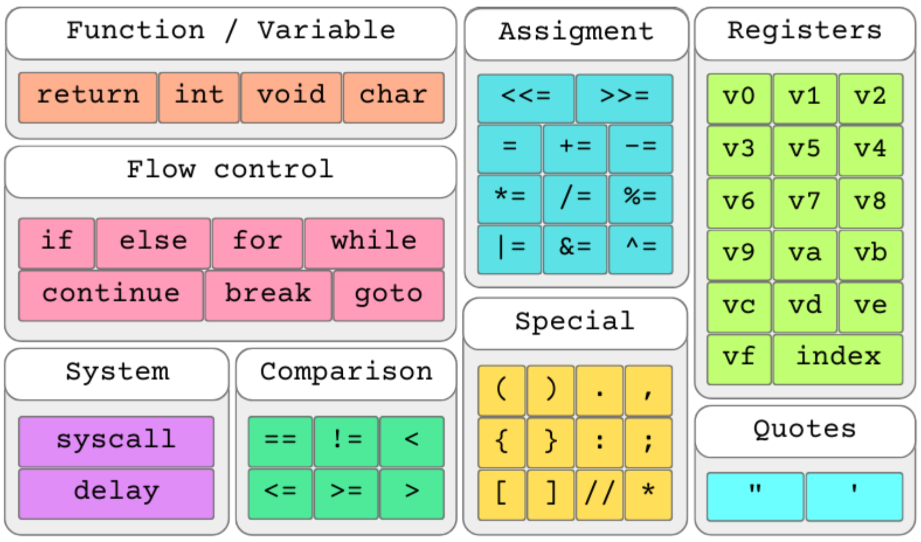
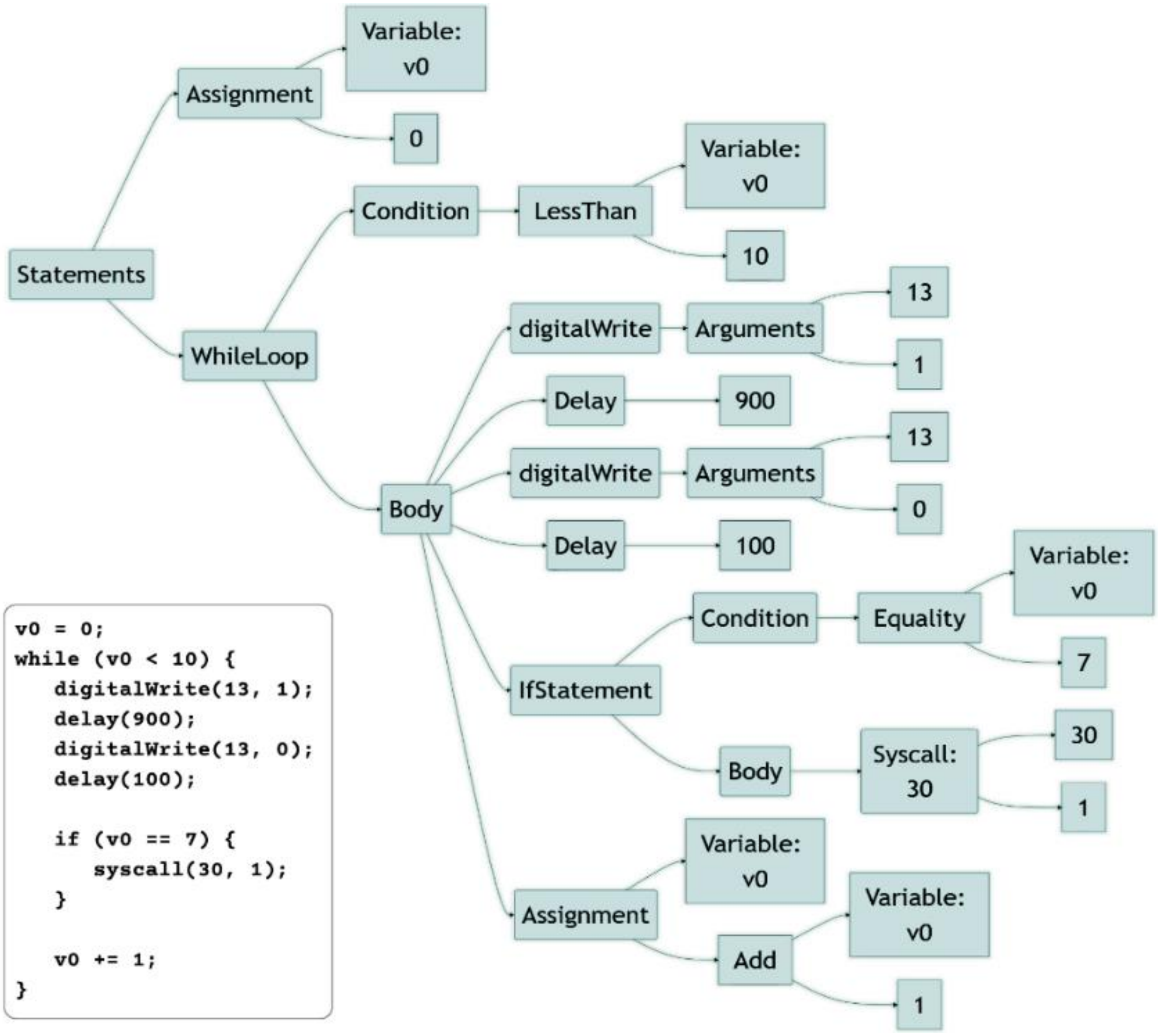


Fig. 3. **Categorization of available tokens of the proposed programming language**

Syntax analysis. The token stream produced by the lexical analysis phase serves as the input for the subsequent stage of compilation: syntax analysis. By converting the raw character data into a structured sequence of meaningful tokens, the lexical analyzer simplifies the task of the parser. The parser can then operate on these higher-level tokens rather than individual characters, allowing it to focus on verifying the grammatical structure and hierarchical relationships within the code according to the language's syntax rules. This separation is a fundamental principle in compiler design, contributing to modularity and simplifying the complexity of both the lexical and syntactic analysis phases.

The primary objective of syntax analysis is to determine if the sequence of tokens produced by the lexer conforms to the formal grammatical rules defined for the proposed programming language. This process verifies the structural correctness of the code and simultaneously constructs a hierarchical representation that reflects the relationships between the tokens, providing a structured view of the program. The syntax of the portable language is formally defined by a context-free grammar, which specifies the valid sequences in which tokens can appear to form syntactically correct language constructs such as expressions, statements, and definitions. The parser, the component of the compiler responsible for this stage, reads the token stream and applies these grammar rules to group the tokens into phrases and verify their arrangement. This process involves recognizing patterns in the token sequence that correspond to the language's syntactic constructs, ensuring that the ordering and combination of tokens strictly follow the defined grammar. As the parser successfully recognizes these constructs, it builds a hierarchical representation of the source code in the form of an Abstract Syntax Tree (AST). The AST provides a concise and abstract representation that captures the essential structure and relationships of the code while omitting unnecessary grammatical details, offering a convenient structure for following compilation phases like semantic analysis and code generation. During parsing, if the parser encounters a sequence of tokens that does not follow the language's grammatical rules, a syntax error is detected and reported to the user, indicating the location of the structural mistake. Syntax analysis serves as an intermediary phase, transforming the linear token stream into a structured representation that is crucial for understanding the program's meaning and aiding the later stages of the compilation process.

The AST of the proposed language is simpler compared to the C programming language because of the reduced number of tokens and simplicity of the design (Fig. 4).



```
v0 = 0;
while (v0 < 10) {
   digitalWrite(13, 1);
   delay(900);
   digitalWrite(13, 0);
   delay(100);

   if (v0 == 7) {
      syscall(30, 1);
   }

   v0 += 1;
}
```

Fig. 4. **Abstract Syntax Tree of example program written in the proposed language**

Semantic analysis. Following the validation of the program's structural correctness during the syntax analysis phase, the compilation process proceeds to semantic analysis. This stage is responsible for checking the logical consistency of the source code, ensuring that it adheres to the proposed language's rules beyond just grammatical form. The input to the semantic analyzer is the AST produced by the parser. The primary goals are to detect semantic errors that are syntactically valid but logically inconsistent and to gather essential information required for following stages like intermediate code generation or optimization.

Key checks performed during semantic analysis include static type checking, which verifies that operations are applied to operands of compatible data types according to the proposed language's type system. This prevents illogical operations, such as attempting to perform arithmetic on a non-numeric value or assigning a value of an incompatible type to a variable, unless explicit and valid type coercion or casting is specified. Another important purpose of this stage is scope and declaration checking, ensuring that all identifiers for variables and functions are declared within an accessible scope before they are used and that there are no conflicting declarations within the same scope. This process involves constructing and managing a symbol table, a data structure that stores information about each declared identifier, including its type and scope. Name resolution is performed by querying the symbol table to link each use of an identifier in the code to its corresponding declaration. Additional semantic checks include verifying correct function call arguments and ensuring control flow statements like break or return are used in appropriate contexts.

Semantic analysis performs static checks that can be done at compile time without executing the code. Unlike syntax errors, semantic errors often relate to the context in which language constructs are used. If semantic errors are detected such as using an undeclared variable or type mismatch, the compiler reports these issues to the user, providing details about the nature and location of the error. Successfully analyzed code is represented by an annotated AST, where nodes are decorated with semantic information such as resolved types and symbol table references. This enhanced representation serves as the validated and semantically meaningful input for the next stage of the compiler, building a connection between the purely structural view provided by the parser and the requirements for generating executable code for the Mico8-Chip virtual machine.

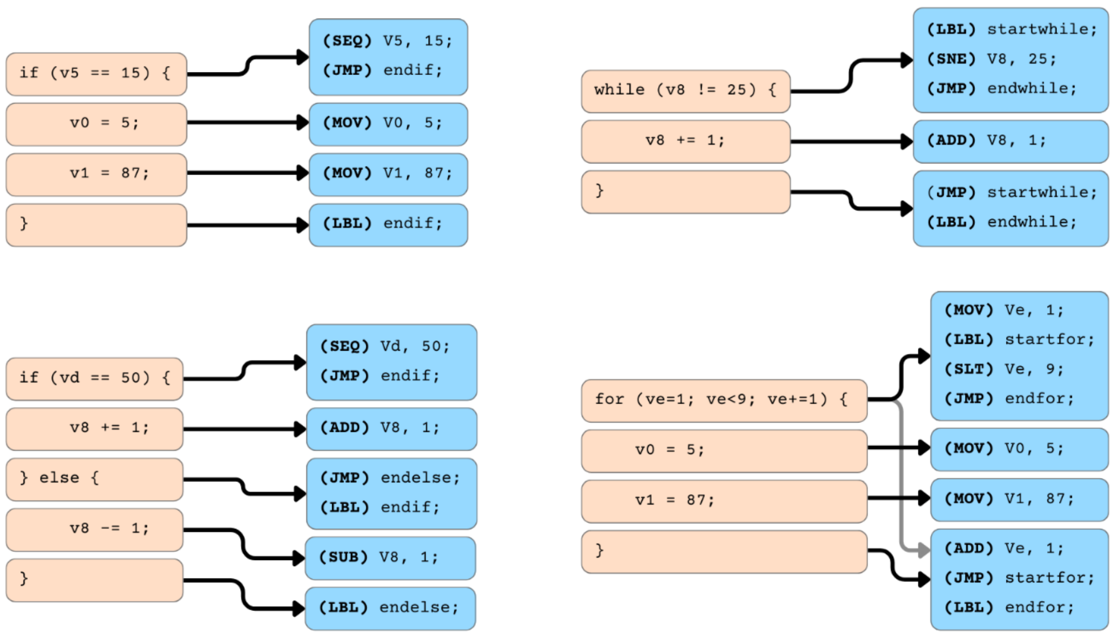
Intermediate Representation generation. After the semantic analysis process, the compiler process proceeds to generate an Intermediate Representation (IR). The IR serves as a bridge between the language-specific front-end of the compiler, which includes lexical, syntax, and semantic analysis, and the target-specific back-end, which involves optimization and code generation. Its primary purpose is to transform the program from a high-level, language-dependent structure like the AST into a lower-level, more machine-like representation that is independent of the specific target architecture, in this case, the Mico8-Chip virtual machine. IR is still more abstract than raw machine code or bytecode.

The adoption of an IR provides effective separation between the front-end and back-end. The front-end translates the source language into the IR, and the back-end translates the IR into the target code. This modularity is important for compiler development and maintenance. For instance, adding support for a new source language only requires building a new front-end that targets the existing IR, and supporting a new target architecture requires only building a new back-end that uses the existing IR. The IR is the representation at which many compiler optimizations are performed. IRs are designed to provide opportunities for optimization more effectively than either the source-level AST or the final target code. These optimizations include dead code elimination and loop transformations. Performing optimizations on a single IR allows them to be language-independent and target-independent, reducing redundant implementation effort. IRs can take various forms, ranging from graphical structures, to tree-based IRs simpler than the AST, to various forms of linear code. The compiler of the proposed portable language generates the IR by traversing the semantically analyzed AST and translating the high-level constructs into a sequence of simpler, sequential operations. This step systematically breaks down complex statements and expressions into a more elementary form that is closer to the operations the target virtual machine can execute (Fig. 5).

The IR generated at this stage then becomes the input for the subsequent optimization phase, where various algorithms are applied to improve the code's efficiency. Following optimization, the IR serves as the basis from which the final Mico8-Chip bytecode is generated. Therefore, the IR acts as an intermediate format that standardizes the program representation for both optimization and target code generation, contributing to the compiler's modularity, efficiency, and portability.

Bytecode generation. Bytecode generation is the final stage in the compiler pipeline, following the generation and optimization of the IR. This phase is responsible for translating the optimized IR into the specific instruction set that the target virtual machine, the Mico8-Chip, can directly interpret and execute. It represents the final transformation of the source code program into its portable, executable form. The input to this stage is the optimized IR code produced by the compiler's front-end, and the output is the sequence of bytes containing the Mico8-Chip bytecode.

The core task of bytecode generation is to map the operations and control flow structures expressed in the optimized IR onto the opcodes defined by the Mico8-Chip virtual machine's instruction set architecture. This involves iterating through the IR and, for each operation or sequence of operations, selecting the appropriate Mico8-Chip opcodes that perform the equivalent function. This selection process must consider the specifics of the Mico8-Chip's architecture, including its register set, memory model, and available operations. Control flow constructs within the IR, such as conditional branches, loops, and function calls are translated into the corresponding jump and branch instructions of the Mico8-Chip virtual machine, requiring careful management of addresses within the target bytecode space. Data access operations specified in the IR are mapped to the instructions for loading from and storing to memory.



```
if (v5 == 15) {      (SEQ) V5, 15;
                     (JMP) endif;
    v0 = 5;          (MOV) V0, 5;
    v1 = 87;         (MOV) V1, 87;
}                    (LBL) endif;
```

```
while (v8 != 25) {   (LBL) startwhile;
                     (SNE) V8, 25;
                     (JMP) endwhile;
    v8 += 1;         (ADD) V8, 1;
}                    (JMP) startwhile;
                     (LBL) endwhile;
```

```
if (vd == 50) {      (SEQ) Vd, 50;
                     (JMP) endif;
    v8 += 1;         (ADD) V8, 1;
} else {             (JMP) endelse;
                     (LBL) endif;
    v8 -= 1;         (SUB) V8, 1;
}                    (LBL) endelse;
```

```
for (ve=1; ve<9; ve+=1) {   (MOV) Ve, 1;
                            (LBL) startfor;
                            (SLT) Ve, 9;
                            (JMP) endfor;
    v0 = 5;                 (MOV) V0, 5;
    v1 = 87;                (MOV) V1, 87;
}                           (ADD) Ve, 1;
                            (JMP) startfor;
                            (LBL) endfor;
```

Fig. 5. **Example conversion of "if", "while", "if-else", and "for" blocks to IR code**

This phase is the most target-dependent part of the compiler's back-end, as it is intimately tied to the specific instruction set of the Mico8-Chip virtual machine. While the preceding IR provides a degree of abstraction, effective bytecode generation requires a detailed understanding of the operations, operand types, and encoding formats. The output is a linear sequence of 16-bit opcodes, formatted precisely as expected by the Mico8-Chip interpreter. This bytecode contains the compiled program logic and data in a form that is ready to be loaded into the Mico8-Chip virtual machine running on a microcontroller. The portability of the compiled program across different microcontroller platforms is achieved because the Mico8-Chip virtual machine, which interprets this bytecode, is designed to abstract away the underlying hardware differences, executing the same bytecode regardless of the specific microcontroller it is running on. Thus, bytecode generation concludes the compilation process by producing the artifact that enables cross-platform execution via the virtual machine.

Experimental results. The Mico8-Chip virtual machine is implemented in the C programming language for each specific microcontroller platform. However, benchmarking of the virtual machine's runtime performance is outside the scope of this paper, as the primary focus is on the design of the portable programming language and its compiler. Evaluating the virtual machine's execution efficiency depends heavily on its specific native implementation for each microcontroller architecture, a separate research effort from the language and compiler design presented.

Evaluation of compilation speed on host development platforms revealed it does not create a practical bottleneck. A large source file exceeding 20,000 lines (~600 kilobytes) compiles in under one second on a standard workstation, resulting in a bytecode ~150 kilobytes in size. The Mico8-Chip virtual machine's typical 8 kilobyte bytecode limit already renders such large source files unrealistic for the target. Moreover, the compiler's implementation in Python suggests even faster speeds are achievable with a compiled language, supporting that host compilation time is not a limiting factor in the development workflow.

A set of programs was developed using the proposed portable language and compiled into Mico8-Chip bytecode for testing purposes. As demonstrated by the data presented in Table 1, the generated bytecode for these programs has comparable size to functionally equivalent code implemented manually at a low level, such as hand-crafted Mico8-Chip instruction sequences or highly optimized low-level code. This competitive code size is achieved while offering developers the advantage of working with a higher level of abstraction through the proposed language's syntax, which improves source code clarity, increases readability, and reduces the complexity associated programming for the target virtual machine.

Table 1. **Compiled size of example programs**

Example program written in the proposed language	Bytecode size
Blinking an LED with a simple delay mechanism	10 bytes
Controlling an LED with a button click event	14 bytes
Outdoor light activation when it is dark outside	16 bytes
Servo motor control using two separate buttons	34 bytes
Ultrasonic distance measurement for object detection	52 bytes
Animated lights on an RGB LED array display	68 bytes
Basic sound-sensitive alarm system for security purposes	82 bytes
Periodic watering system designed for indoor plants	102 bytes
6 button sound pad with system calls for audio playback	140 bytes
Toggling a device on and off with hand clapping	176 bytes
Temperature and humidity logger for environmental monitoring	192 bytes
Automated feeding system for domestic pets	234 bytes
Controlling IoT devices using system calls	310 bytes

Conclusion and future work. The increasing complexity of microcontroller-based systems within the Internet of Things and other embedded applications highlights significant challenges associated with traditional development approaches. As discussed in this paper, the characteristic resource constraints of microcontrollers, coupled with the complexity of low-level programming and the fragmentation of the hardware landscape, decrease developer productivity and limit code portability. These factors require repetitive development efforts when targeting different platforms. This paper presented the design and implementation of a virtual machine-based portable programming language aimed at addressing these fundamental challenges. The core purpose of this paper was to create a cross-platform development solution for microcontrollers. The program codes written in this language compiles to bytecode for Mico8-Chip virtual machine, a re-engineered architecture based on CHIP-8, adapted to meet the demands of modern microcontroller systems. Mico8-Chip virtual machine includes a built-in concurrent task execution environment and to serves as the universal execution target for the proposed language. The language features C-like syntax, incorporating features necessary for embedded development. Through this work, the compiler pipeline of this language was detailed, which is responsible for compiling the source code into the final Mico8-Chip bytecode. This integrated approach directly addresses the problem of code portability, as demonstrated by the ability to generate uniform bytecode for different microcontroller platforms, and lowers development complexity by offering a higher level of abstraction.

In conclusion, this work contributes a programming language that address the issue of fragmentation and portability in the microcontroller ecosystem. By providing a consistent compilation target and execution environment, the system simplifies the development process, reduces porting effort, and makes embedded programming more accessible while remaining mindful of the limitations of the target hardware. The effectiveness of the designed compiler pipeline in generating compact code for resource-constrained environments is demonstrated by the bytecode sizes of various example programs, as presented in Table 1. The small bytecode sizes for various functional programs confirm that the solution successfully balances high-level abstraction with the memory constraints of microcontrollers. This directly contributes to achieving the goal of accelerating development without significant performance or memory overhead, thus making embedded programming more accessible and portable across diverse hardware platforms.

Several paths exist for future research and development. Expanding the feature set of the portable language to include more complex data structures or standard libraries would enhance its applicability. Further optimizations to the compiler's back-end and the Mico8-Chip interpreter could potentially reduce execution overhead and memory footprint. Porting the Mico8-Chip virtual machine to a wider array of microcontroller families would broaden the system's reach. Additionally, exploring advanced features like developing integrated debugging and profiling tools could further enhance the system's utility for embedded software engineers. This research provides a solid foundation for continued exploration into high-level, portable development paradigms for the ever-growing domain of microcontroller-based systems.

REFERENCES

1. Wu, Z., Qiu, K., & Zhang, J. (2020). A Smart Microcontroller Architecture for the Internet of Things. *Sensors*, 20(7), 1821. https://doi.org/10.3390/s20071821.
2. Krishnamurthy, J., & Maheswaran, M. (2016). Programming frameworks for Internet of Things. In *Internet of Things* (pp. 79-102). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-805395-9.00005-8.
3. Bell, C. (2024). *MicroPython for the Internet of Things: A Beginner's Guide to Programming with Python on Microcontrollers Second Edition*. Apress eBooks. https://doi.org/10.1007/978-1-4842-9861-9.
4. Dashdamirli, N. (2025). Hybrid scheduling approach for concurrent task execution on microcontroller-based systems. *Romanian Journal of Information Technology and Automatic Control*, 35(1), 79–90. https://doi.org/10.33436/v35i1y202506.
5. Priyadarshini, S. B. B., Mahapatra, A., Mohanty, S. N., Nayak, A., Jena, J. P., & Samanta, S. K. S. (2022). myCHIP-8 emulator: An innovative software testing strategy for playing online games in many platforms. In *Optimization of Automated Software Testing Using Meta-Heuristic Techniques* (pp. 133-154). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-031-07297-0_9.
6. Cruz, N., Ruiz-Ferrández, M., Redondo, J. L., Álvarez, J., & Ortigosa, P. (2019). Applications of chip-8, a virtual machine from the late seventies, in current degrees in computer engineering. In *EDULEARN19 Proceedings* (pp. 1720-1729). IATED. https://doi.org/10.21125/edulearn.2019.0501.
7. Dubé, D., & Feeley, M. (2005). BIT: A Very Compact Scheme System for Microcontrollers. *Higher-Order and Symbolic Computation*, 18(3), 271-298. https://doi.org/10.1007/s10990-005-4877-4.
8. Feeley, M., & Dubé, D. (2003, November). PICBIT: A Scheme system for the PIC microcontroller. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming* (pp. 7-15).
9. St-Amour, V., & Feeley, M. (2010). PICOBIT: A Compact Scheme System for Microcontrollers. *Implementation and Application of Functional Languages*, 1-17. https://doi.org/10.1007/978-3-642-16478-1_1.
10. Brouwers, N., Corke, P., & Langendoen, K. (2008, December). Darjeeling, a Java compatible virtual machine for microcontrollers. In *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion* (pp. 18-23). https://doi.org/10.1145/1462735.1462740.
11. Aslam, F. (2011). *Challenges and Solutions in the Design of a Java VirtualMachine for Resource Constrained Microcontrollers* (Doctoral dissertation, University of Freiburg).

12. Varoumas, S., Vaugon, B., & Chailloux, E. (2016, January). Concurrent Programming of Microcontrollers, a Virtual Machine Approach. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)* (pp. 711-720).
13. Varoumas, S., Vaugon, B., & Chailloux, E. (2018, January). A generic virtual machine approach for programming microcontrollers: the OMicroB project. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018).*
14. Gurdeep Singh, R., & Scholliers, C. (2019, October). WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (pp. 27-36). https://doi.org/10.1145/3357390.3361029.
15. Zandberg, K., & Baccelli, E. (2020, December). 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)* (pp. 1-6). IEEE. https://doi.org/10.23919/PEMWN50727.2020.9293081.
16. Sarkar, A., Krook, R., Svensson, B. J., & Sheeran, M. (2021, September). Higher-order concurrency for microcontrollers. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (pp. 26-35). https://doi.org/10.1145/3475738.3480716.

Нуру Дашдамірлі[1]

[1]Докторант, Азербайджанський Технічний Університет, пр. Г.Джавід, 25, Баку, Азербайджан, AZ 1148. ORCID: https://orcid.org/0009-0009-0545-7855.

Крос-платформенна розробка для мікроконтролерів: проєктування переносимої мови програмування на основі віртуальної машини

Фундаментальна роль мікроконтролерів у вбудовуваних системах та середовищах Інтернету речей потребує ефективних підходів до розробки програмного забезпечення. Обмеження ресурсів мікроконтролерів, складність мов програмування низького рівня та проблеми реалізації багатозадачності значно уповільнюють процес розробки. Крім того, різноманітність екосистеми мікроконтролерів створює суттєві бар'єри для портативності коду, що призводить до збільшення часу розробки для підтримки різних апаратних платформ. У цій статті представлено розробку підходу програмування на основі віртуальної машини, що забезпечує кросплатформенну розробку для мікроконтролерів. Пропонована мова програмування, що переноситься, інтегрується з користувальницькою віртуальною машиною Mico8-Chip для відповідності сучасним додаткам мікроконтролерів. Це забезпечує інтуїтивне керування периферійними пристроями та вбудовану підтримку паралельного виконання. Наданий рівень абстракції значно покращує переносимість коду та прискорює розробку, ізолюючи логіку застосування від базових апаратних особливостей.

***Ключові слова**: мікроконтроллер, вбудовані системи, віртуальна машина, програмування, байт-код, паралелізм, інтернет речей.*