Vol. 7, No. 2, 2025

Ruslan Holovatskyy

Computer Design Systems Department, Lviv Polytechnic National University, 12, S. Bandery str., Lviv, Ukraine, E-mail: ruslan.i.holovatskyi@lpnu.ua, ORCID 0009-0001-3096-1115

RESEARCH OF THE EFFICIENCY OF CODE GENERATION BY THE ARDUINO IDE DEVELOPMENT ENVIRONMENT USING THE EXAMPLE OF ARITHMETIC OPERATIONS OF ADDITION AND SUBTRACTION

Recieved: September 01, 2025 / Revised: September 09, 2025 / Accepted: September 15, 2025

https://doi.org/10.23939/cds2025.02.030

Abstract. This article examines the efficiency of code generation by the Arduino IDE development environment when performing elementary arithmetic operations of addition and subtraction. This environment is a popular tool among developers for working with microcontrollers, as it has a convenient interface for rapid prototyping. One of the key characteristics of such environments is the quality of the generated code, which affects the speed of program execution, memory usage, and overall system performance. Therefore, studying the efficiency of code generation by the specified development environment is a relevant task. This study analyzes performance, memory usage, and code optimization by the compiler. Experimental results are presented and conclusions are drawn regarding the feasibility of using the environment for tasks with high performance requirements.

Keywords: Arduino IDE, code generation, efficiency, arithmetic operations, addition, subtraction, microcontrollers, AVR, firmware file.

Introduction

Arduino IDE is one of the most popular development environments for programming various microcontrollers used in the Arduino hardware platforms [1]. Today, the Arduino hardware platforms include the following families: Arduino Nano, Arduino MKR, Arduino Mega, and Arduino Classic.

The Arduino Nano family is a set of miniature boards with a lot of features. It ranges from the inexpensive basic Nano Every to the multi-functional Nano 33 BLE Sense / Nano RP2040 Connect with Bluetooth® / Wi-Fi® radios. These boards also have a set of built-in sensors such as temperature/humidity, pressure, gesture recognition, microphone, etc.

The Arduino MKR family is a series of boards, shields, and carriers that can be combined to create fairly complex projects without any additional circuitry. Each board is equipped with a radio module (except the MKR Zero) that supports Wi-Fi®, Bluetooth®, LoRa®, Sigfox®, NB-IoT communication. All boards in the family are based on the low-power 32-bit Arm® Cortex®-M0 processor SAMD21 and are equipped with a cryptographic chip for secure communication.

The Arduino Mega family is a board for projects based on AVR microcontrollers that require a lot of processing power and a large number of GPIO pins.

The Arduino Classic family is a classic family of boards based on AVR microcontrollers, ranging from the legendary Arduino UNO to boards such as Leonardo & Micro.

Despite the variety of Arduino boards, the Arduino IDE provides a user-friendly interface for writing, compiling, and uploading code without requiring a deep understanding of the low-level details of the microcontroller. This allows even beginners to quickly create their own projects using sensors, actuators, displays, and other electronic components.

Review of Modern Information Sources on the Subject of the Paper

The Arduino hardware platform [1] based on AVR microcontrollers [2] is widely used for educational purposes [3–5]. It also allows you to create interesting and quite complex projects that are used by enthusiasts around the world in everyday life [6–9]. Many developers use the Arduino platform not only for learning, but also as a tool for creating more complex systems or even as a replacement for expensive devices and machines [10–12]. For programming and rapid prototyping, the Arduino IDE development environment [1] is usually used, which already contains ready-made libraries. However, the question arises: how effectively does this environment generate code? Therefore, studying the efficiency of code generation by the aforementioned development environment, which will answer the question posed, is a relevant task. This article investigates the efficiency of code generated by the Arduino IDE, using the example of elementary arithmetic operations of addition and subtraction, which are the basis of any serious project.

Problem Statement

After analyzing a large number of Arduino projects, it can be noted that, despite its convenience, the Arduino IDE does not always generate optimal machine code. Microcontrollers used in Arduino platforms have limited resources, in particular, a small amount of FLASH memory for storing the program, limited random access memory (RAM) and a relatively low clock frequency [2]. Because of this, a large and unoptimized program can lead to crashes, slow down the device, or even lack of memory to perform basic operations.

In addition, the performance of the device depends not only on the characteristics of the microcontroller, but also on the quality of the written code. Some standard functions used in the Arduino IDE (for example, digitalWrite() or delay()) greatly simplify programming, but are slower than direct access to the microcontroller registers. The use of inefficient algorithms, inappropriate data types, incorrect and inefficient variable descriptions or excessive function calls can lead to unnecessary memory usage and delays in program execution.

That is why studying the efficiency of code generation by the Arduino IDE development environment with the aim of further optimizing it plays a key role in developing efficient programs for microcontrollers. Efficient code allows you to reduce memory usage, increase data processing speed, and extend the battery life of the device, which is especially important for power-dependent projects.

By efficient code we mean code that performs the assigned tasks with minimal resource consumption, such as processor time, memory, and power consumption, without losing clarity, readability, and maintainability [13, 14].

The main characteristics of effective code include the following:

1. Performance – the code runs quickly, minimizing unnecessary operations and function calls.
2. Optimal memory usage – excessive consumption of RAM and FLASH memory is avoided.
3. Readability and maintainability – the code is well-structured, understandable to other developers and easily adaptable to new requirements.
4. Compactness – code duplication is avoided, efficient algorithms are used.
5. Modularity – the code is divided into independent functional blocks, which simplifies its modification and testing.
6. Balance between speed and resource consumption – a compromise between performance and energy saving is achieved, which is important for embedded systems.

In the context of Arduino, efficient code allows the device to run faster, more stable, and more economically using the microcontroller’s limited resources. In the case of power supply from an autonomous power source, the operating time of the developed microcontroller system, which runs under the control of optimized software code, increases.

For AVR microcontrollers used in the Arduino Nano, Arduino Mega and Arduino Classic families, and operating at frequencies up to 20 MHz, in the vast majority of projects [6–12], the amount of available memory is crucial: both FLASH for storing code and RAM. Therefore, in this study, the main focus is on the size of the generated code.

Main material presentation

To investigate the efficiency of code generation by specialized development environments for the AVR architecture using the example of elementary arithmetic operations of addition and subtraction, we will use Arduino IDE version 2.3.4. As a hardware platform for research, we will choose the classic Arduino UNO R3 board, Fig. 1.

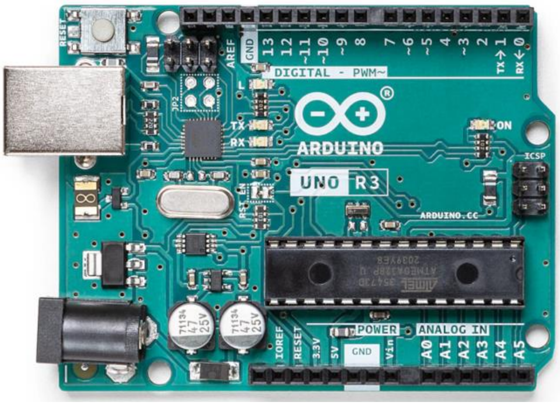

Fig. 1. The Arduino UNO R3 board under study

Addition operation. First, we will find the most optimal implementation of the addition operation on integer variables of type byte. The byte data type occupies 1 byte in memory, accepts values: 0–255 (unsigned integers). To do this, we will determine the size of the code generated by the Arduino IDE for the following ways of performing this operation:

1. Local variables without additional functions.
2. Global variables, an addition function without parameters and without returning a result.
3. Global variables, an addition function without parameters with returning a result.
4. Global variables, an addition function that accepts arguments and returns a result.
5. Local variables, an addition function that accepts arguments and returns a result.

A sketch that implements the addition operation of two variables of type byte using local variables and without additional functions is shown in Fig. 2.

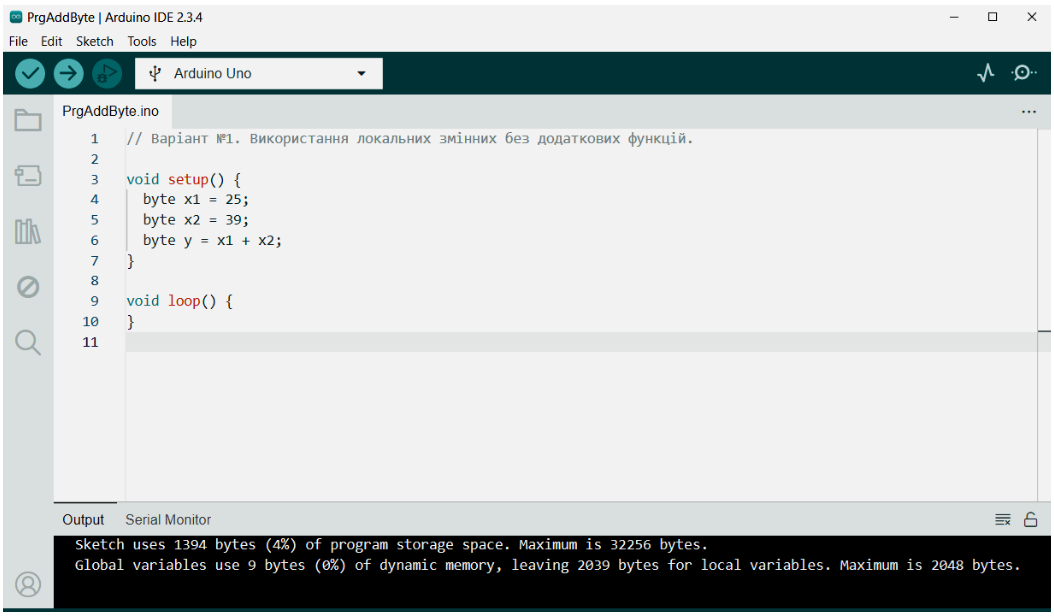

Fig. 2. The operation of adding two variables of type byte using local variables and without additional functions

As can be seen from Fig. 2, the size of the firmware file, without the bootloader, takes up 1394 bytes out of 32256 available bytes for the ATMega328P microcontroller. This is approximately 4.32 % of the available FLASH memory. For the ATMega16 microcontroller, which has 16128 bytes of FLASH memory available, this will be approximately 8.64 % of the available memory. And for the ATMega8 microcontroller, which has 8064 bytes of FLASH memory available, this will be approximately 17.3 % of the available memory. With the bootloader, this percentage will already be 23.5 % of the available FLASH memory.

A sketch that implements the operation of adding two variables of type byte using global variables and the addition function without parameters and without returning the result is shown in Fig. 3.

Fig. 3. The operation of adding two variables of type byte using global variables and the addition function without parameters and without returning a result

A sketch that implements the operation of adding two variables of type byte using global variables and the add function without parameters but returning the result is shown in Fig. 4.

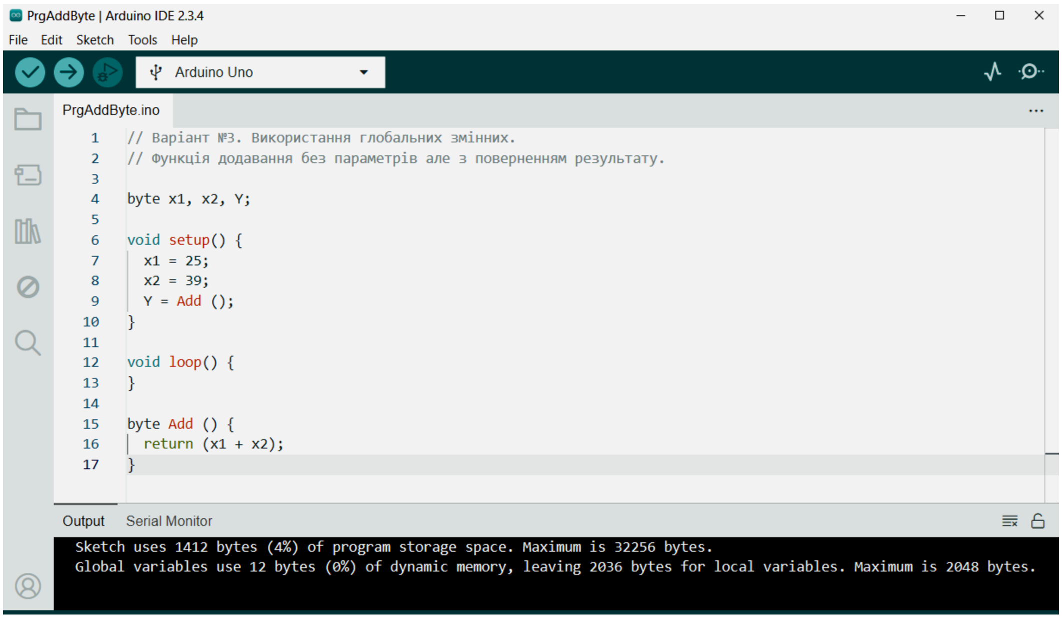

Fig. 4. The operation of adding two variables of type byte using global variables and the addition function without parameters but returning the result

As can be seen from Fig. 3, the size of the firmware file, without the bootloader, takes up 1412 bytes out of 32256 available bytes for the ATMega328P microcontroller. This is approximately 4.38 % of the available FLASH memory. For the ATMega16 microcontroller, which has 16128 bytes of FLASH memory available, this will be approximately 8.75 % of the available memory. And for the ATMega8 microcontroller, which has 8064 bytes of FLASH memory available, this will be approximately 17.51 % of the available memory. With the bootloader, this percentage will already be 23.74% of the available FLASH memory.

As can be seen from Fig. 4, the size of the firmware file, without the bootloader, takes up 1412 bytes out of 32256 available bytes for the ATMega328P microcontroller. This is approximately 4.38 % of the available FLASH memory. For the ATMega16 microcontroller, which has 16128 bytes of FLASH memory available, this will be approximately 8.75 % of the available memory. And for the ATMega8 microcontroller, which has 8064 bytes of FLASH memory available, this will be approximately 17.51 % of the available memory. With the bootloader, this percentage will already be 23.74 % of the available FLASH memory. This implementation of the operation of adding by the size of the firmware file is completely similar to the previous one, shown in Fig. 3.

A sketch that implements the operation of adding two variables of type byte using global variables and an addition function that accepts arguments and returns a result is shown in Fig. 5.

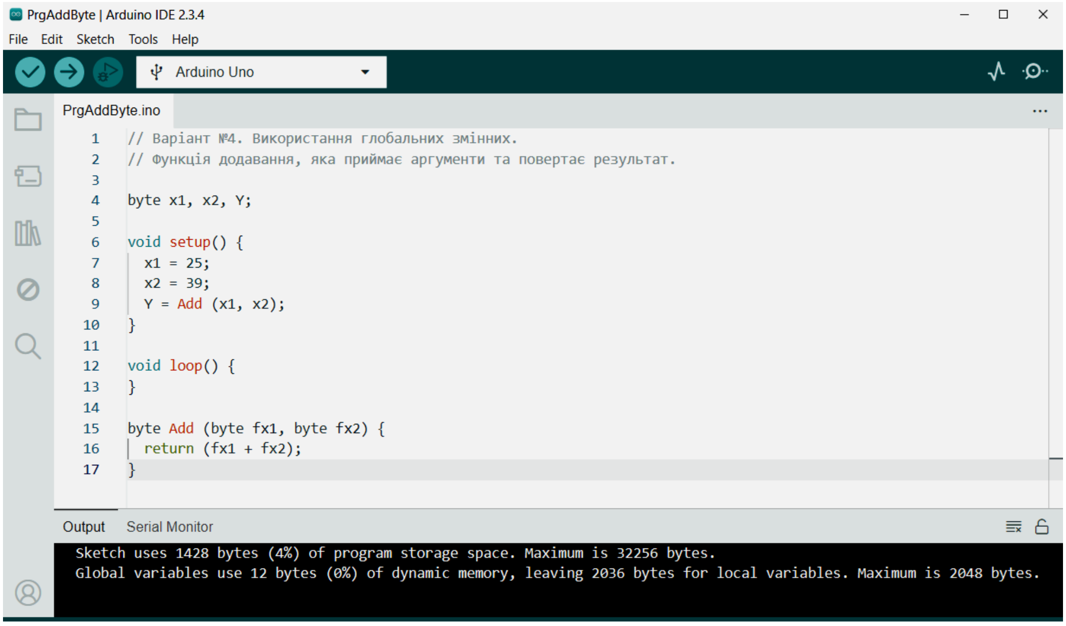

Fig. 5. The operation of adding two variables of type byte using global variables and an addition function that accepts arguments and returns the result

As can be seen from Fig. 5, the size of the firmware file, without the bootloader, takes up 1428 bytes out of 32256 available bytes for the ATMega328P microcontroller. This is approximately 4.43 % of the available FLASH memory. For the ATMega16 microcontroller, which has 16128 bytes of FLASH memory available, this will be approximately 8.85 % of the available memory. And for the ATMega8 microcontroller, which has 8064 bytes of FLASH memory available, this will be approximately 17.71 % of the available memory. With the bootloader, this percentage will already be 23.93 % of the available FLASH memory! This implementation of the operation of adding by the size of the firmware file is the worst of all those studied within the framework of this work.

A sketch that implements the operation of adding two variables of type byte using local variables and an addition function that accepts arguments and returns the result is shown in Fig. 6.

As can be seen from Fig. 6, the size of the firmware file, without the bootloader, takes up 1426 bytes out of 32256 available bytes for the ATMega328P microcontroller. This is approximately 4.42 % of the available FLASH memory. For the ATMega16 microcontroller, which has 16128 bytes of FLASH memory available, this will be approximately 8.84 % of the available memory. And for the ATMega8 microcontroller, which has 8064 bytes of FLASH memory available, this will be approximately 17.68 % of the available memory. With the bootloader, this percentage will already be 23.91 % of the available FLASH memory.

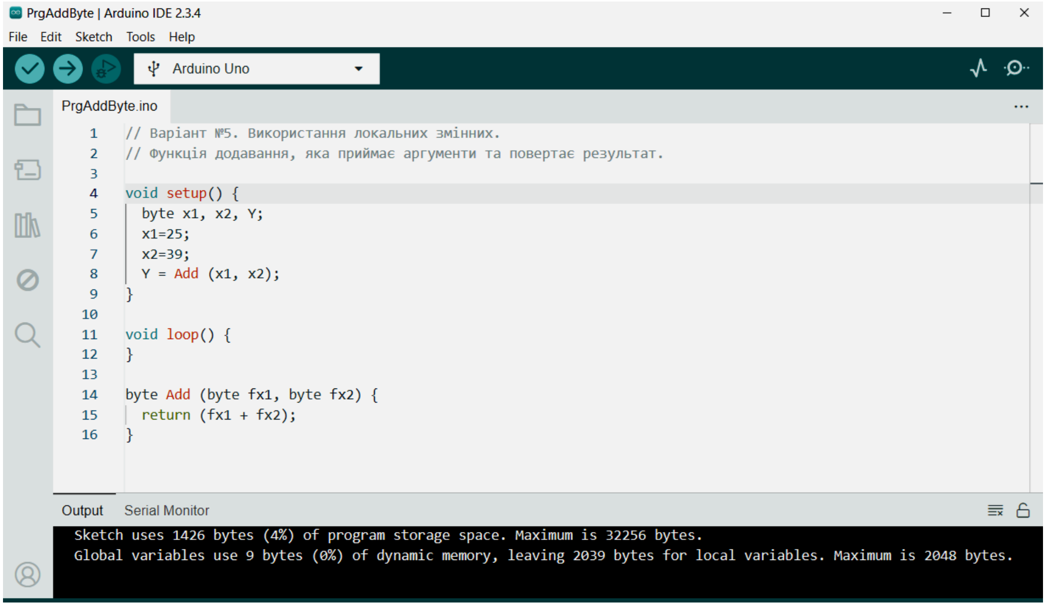

Fig. 6. The operation of adding two variables of type byte using local variables and an addition function that accepts arguments and returns the result

Let us present the obtained results in the form of a bar chart for the convenience of their visual comparison (Fig. 7).

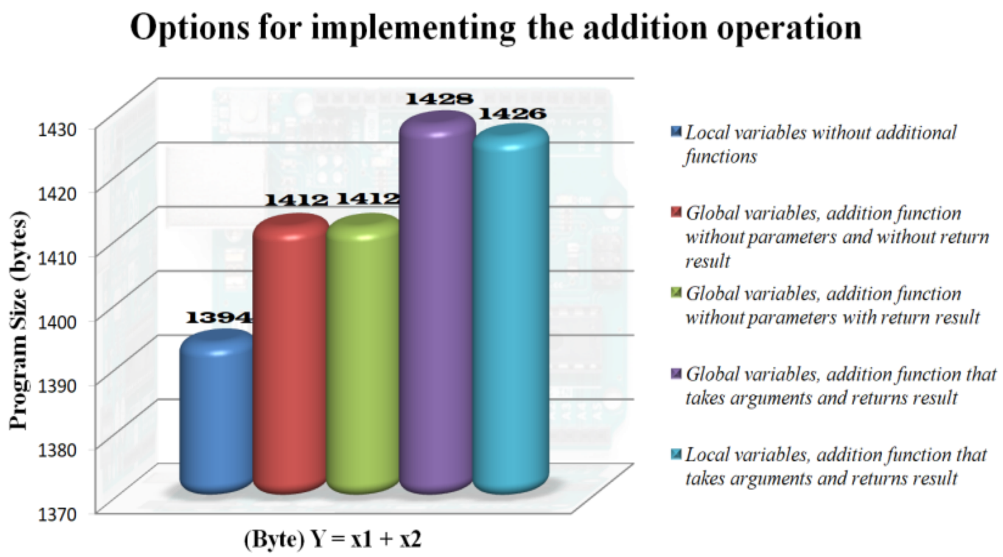

Fig. 7. Size of the MK firmware file without a bootloader for different ways of performing the operation of adding two byte type variables

From the conducted research it is clear that for the operation of adding two single-byte variables, the Arduino IDE generates a firmware code of a minimum size of 1394 bytes when using local variables without using additional functions. Therefore, in the future, for our experiments, we will use exactly this variant of implementing an elementary operation.

Now let's see how the size of the generated code depends on changing the data type from byte to int. Int is an integer type of 16-bit signed numbers in the range: –32 768–32 767, occupies 2 bytes in memory for Arduino UNO R3 boards (for ESP8266 / ESP32 – 4 bytes). A sketch that implements the operation of adding two variables of type int using local variables and without using additional functions is shown in Fig. 8.

Fig. 8. The operation of adding two variables of type int

Now let's replace the int data type with long. Long is a 32-bit integer type of signed numbers in the range: –2 147 483 648–2 147 483 647, occupies 4 bytes in memory for Arduino UNO R3 boards. A sketch that implements the operation of adding two variables of type long using local variables and without using additional functions is shown in Fig. 9.

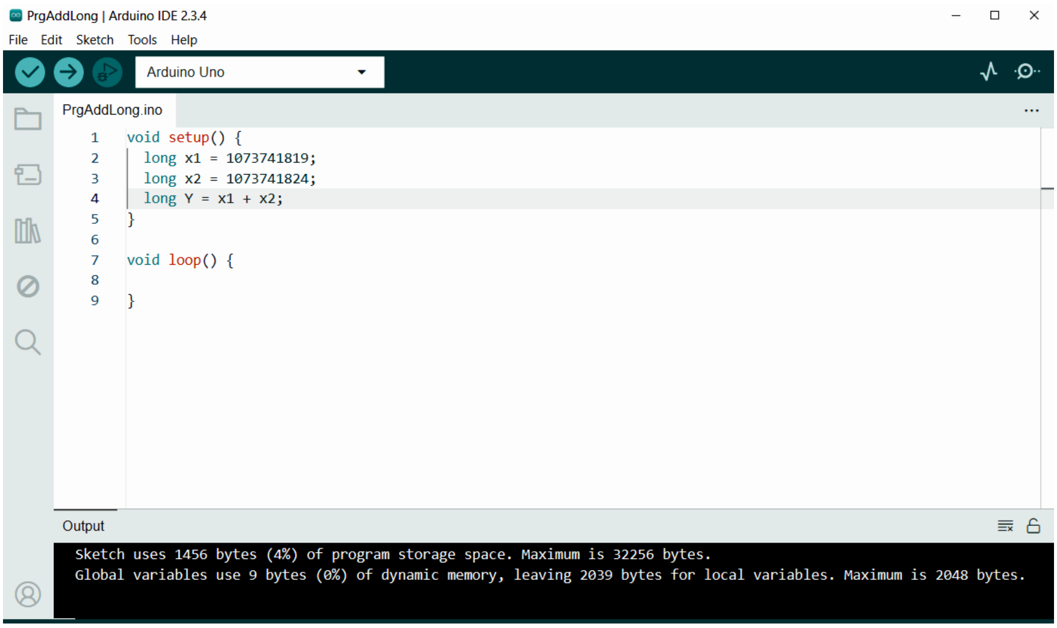

Fig. 9. The operation of adding two variables of type long

As can be seen from Fig. 8, the size of the firmware file, without the bootloader, takes up 1418 bytes out of 32256 available bytes for the ATMega328P microcontroller. This is approximately 4.39 % of the available FLASH memory. For the ATMega16 microcontroller, which has 16128 bytes of FLASH memory available, this will be approximately 8.79 % of the available memory. And for the ATMega8 microcontroller, which has 8064 bytes of FLASH memory available, this will be approximately 17.58 % of the available memory. With the bootloader, this percentage will already be 23.81 % of the available FLASH memory.

As can be seen from Fig. 9, the size of the firmware file, without the bootloader, takes up 1456 bytes out of 32256 available bytes for the ATMega328P microcontroller. This is approximately 4.51 % of the available FLASH memory. For the ATMega16 microcontroller, which has 16128 bytes of FLASH memory available, this will be approximately 9.03 % of the available memory. And for the ATMega8 microcontroller, which has 8064 bytes of FLASH memory available, this will be approximately 18.06 % of the available memory. With the bootloader, this percentage will already be 24.3 % of the available FLASH memory.

Subtraction operation. Now let's examine the efficiency of Arduino IDE code generation for the subtraction operation using the example of using local variables and without using additional functions.

The sketch that implements the subtraction operation of two byte variables is shown in Fig. 10.

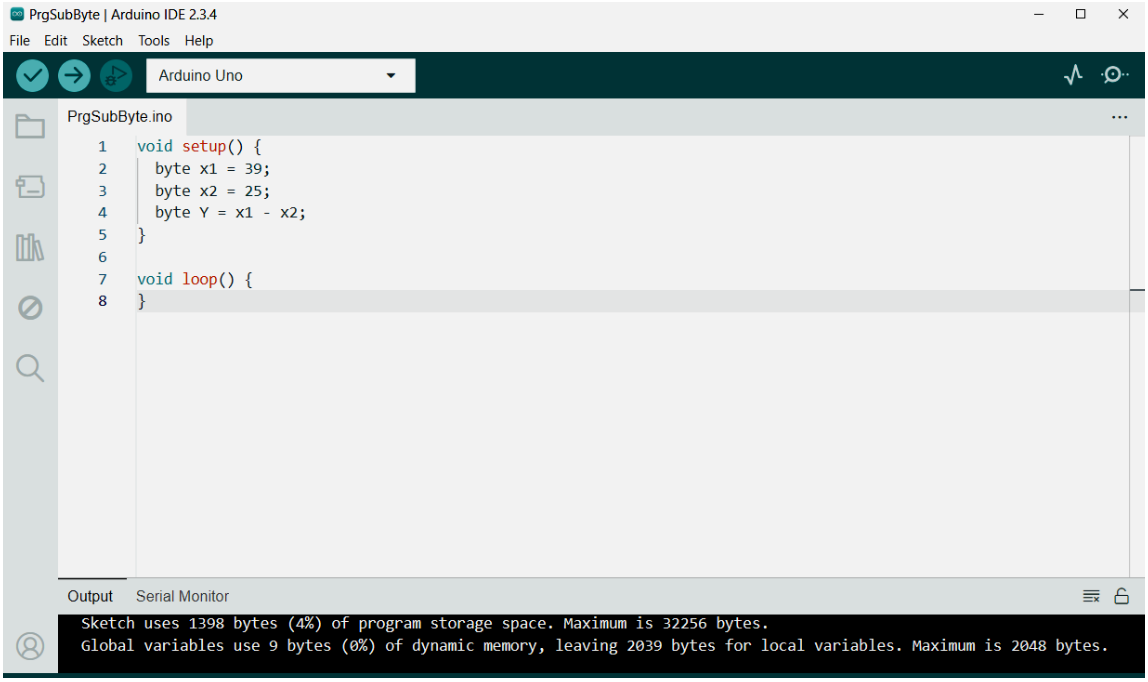

Fig. 10. Subtraction operation of two byte type variables

As can be seen from Fig. 10, the size of the firmware file generated by the Arduino IDE version 2.3.4 environment, without the bootloader, takes up 1398 bytes out of 32256 available bytes for the ATMega328P microcontroller. This is approximately 4.33 % of the available FLASH memory. For the ATMega16 microcontroller, which has 16128 bytes of FLASH memory available, this will be approximately 8.69 % of the available memory. And for the ATMega8 microcontroller, which has 8064 bytes of FLASH memory available, this will be approximately 17.34 % of the available memory. With the bootloader, this percentage will already be 23.56 % of the available FLASH memory.

The sketch that implements the subtraction operation of two int variables is shown in Fig. 11.

As can be seen from Fig. 11, the size of the firmware file is 1422 bytes. This is approximately 4.41 % of the available FLASH memory for the ATMega328P microcontroller. For the ATMega16 microcontroller, which has 16128 bytes of FLASH memory available, this will be approximately 8.82 % of the available memory. And for the ATMega8 microcontroller, which has 8064 bytes of FLASH memory available, this will be approximately 17.63 % of the available memory. With the bootloader, this percentage will already be 23.86 % of the available FLASH memory.

The sketch that implements the subtraction operation of two long variables is shown in Fig. 12.

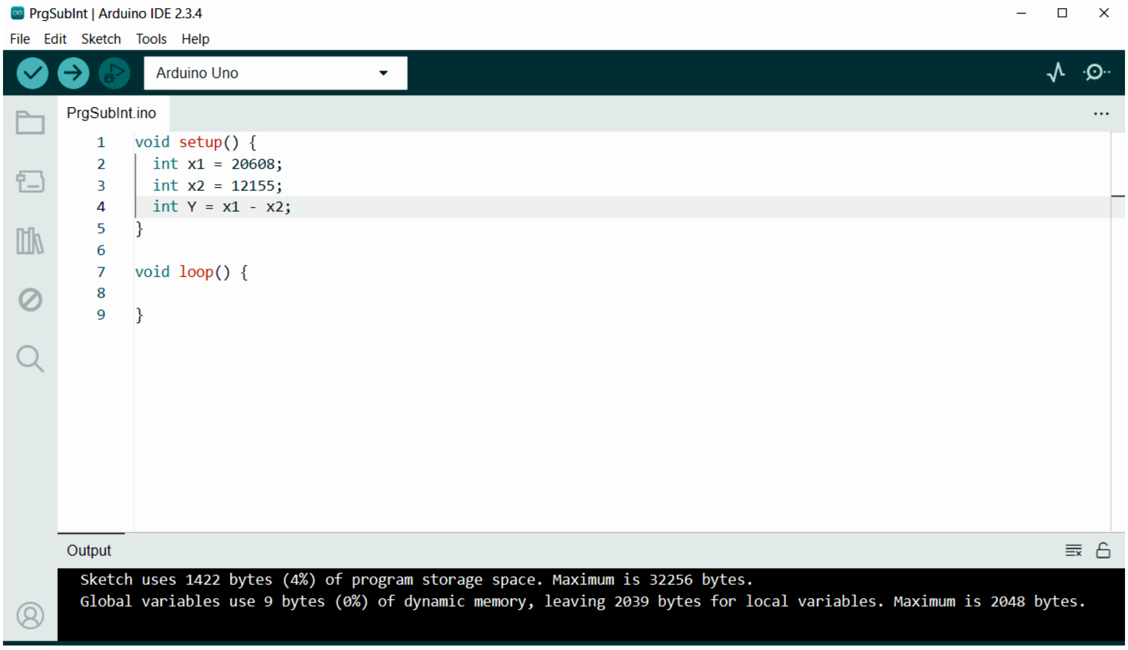

Fig. 11. The operation of subtracting two variables of type int

Fig. 12. Subtraction operation of two variables of type long

As can be seen from Fig. 12, the size of the firmware file generated by the Arduino IDE version 2.3.4 environment, without the bootloader, takes up 1480 bytes. This is approximately 4.59 % of the available FLASH memory for the ATMega328P microcontroller. For the ATMega16 microcontroller, this will be approximately 9.17 % of the available memory. And for the ATMega8 microcontroller, respectively, 18.35 %. With the bootloader, this percentage will already be 24.58 % of the available FLASH memory.

Microcontroller firmware file. An example of a firmware file generated by the Arduino IDE version 2.3.4 development environment without a bootloader and without optimization, with a size of 1394 bytes, compatible with Arduino Classic microcontrollers for the operation of adding two single-byte variables using local variables and without using additional functions is shown in Fig. 13.

PrgAddByte.ino.hex

```
:100000000C9434000C9451000C9451000C94510049
:100010000C9451000C9451000C9451000C9451001C
:100020000C9451000C9451000C9451000C9451000C
:100030000C9451000C9451000C9451000C945100FC
:100040000C949D000C9451000C9451000C945100A0
:100050000C9451000C9451000C9451000C945100DC
:100060000C9451000C94510011241FBECFEFD8E026
:10007000DEBFCDBF11E0A0E0B1E0E2E7F5E002C0F5
:1000800005900D92A030B107D9F721E0A0E0B1E0D2
:1000900001C01D92A930B207E1F70E948A000C94BA
:1000A000B7020C940000CF93DF9300D01F92CDB71E
:1000B000DEB789E1898387E28A8399818A81890F02
:1000C0008B830F900F900F90DF91CF910895CF9376
:1000D000DF93CDB7DEB7DF91CF910895CF93DF9354
:1000E00000D0CDB7DEB79A83898380E090E00F908F
:1000F0000F90DF91CF910895CF93DF93CDB7DEB707
:10010000DF91CF910895CF93DF93CDB7DEB7DF9125
:10011000CF910895CF93DF93CDB7DEB70E9422022F
:100120000E947C000E9453000E94670080E090E0E3
:10013000892BD1F30E940000F7CF1F920F920FB6C8
:100140000F9211248F939F93AF93BF93CF93DF931D
:10015000CDB7DEB72597DEBFCDBF809104019091 6A
:100160000501A0910601B0910701898 39A83AB83B1
:10017000BC838091080 18D8389819A81AB81BC8188
:100180000196A11DB11D89839A83AB83BC838D81A8
:100190008D5F8D838D818D3770F08D818D578D832F
:1001A00089819A81AB81BC810196A11DB11D898392
:1001B0009A83AB83BC838D818093080189819A8166
:1001C000AB81BC81809304019093050 1A09306014B
:1001D000B0930701809100019091010 1A09102016B
:1001E000B09103010196A11DB11D80930001909370
:1001F0000101A0930201B09303010F900F900F90A3
:100200000F900F90DF91CF91BF91AF919F918F9100
:100210000F900FBE0F901F901895CF93DF9300D0D3
:1002200000D01F92CDB7DEB78FE590E0FC01808152
:100230008983F894809104019091050 1A0910601B1
:10024000B09107018A839B83AC83BD838FE590E0E7
:100250002981FC0120838A819B81AC81BD81BC0105
:10026000CD010F900F900F900F900F90DF91CF91D5
:100270000895CF93DF9300D000D000D0CDB7DEB784
:100280008FE590E0FC0180818D83F89480910001DE
:10029000909101 01A0910201B091030189839A8399
:1002A000AB83BC8386E490E0FC0180818E8385E390
:1002B00090E0FC01808188 2F90E081709927892B44
:1002C00071F08E818F3F59F089819A81AB81BC8119
:1002D0000196A11DB11D89839A83AB83BC838FE5F1
:1002E00090E02D81FC01208389819A81AB81BC81C2
:1002F0002227382F492F5A2F8E81882F90E0A0E097
:10030000B0E0820F931FA41FB51F880F991FAA1F6B
:10031000BB1F880F991FAA1FBB1FBC01CD012696CA
:100320000FB6F894DEBF0FBECDBFDF91CF91089519
:10033000EF92FF920F931F93CF93DF93CDB7DEB76A
:1003400028970FB6F894DEBF0FBECDBF6D837E83B6
:100350008F8398870E943901DC01CB0189839A83BE
:10036000AB83BC8337C00E94B00217C08D819E81D1
:10037000AF81B8850197A109B1098D839E83AF83B1
:10038000B88789819A81AB81BC8188519C4FAF4FDE
:10039000BF4F89839A83AB83BC838D819E81AF815C
:1003A000B885892B8A2B8B2BA9F00E9439019B01E0
:1003B000AC0189819A81AB81BC8179018A01E81AFB
:1003C000F90A0A0B1B0BD801C701883E9340A1050F
:1003D000B10560F68D819E81AF81B885892B8A2B0E
:1003E0008B2B09F628960FB6F894DEBF0FBECDBF53
:1003F000DF91CF911F910F91FF90EF900895CF93D0
:10040000DF9300D0CDB7DEB79A83898389819A8143
:10041000029798F089819A81880F991F880F991FF8
:100420009A838983898 19A8105979A83898389 81AF
:100430009A810197F1F79A8389830F900F90DF914A
:10044000CF910895CF93DF93CDB7DEB7789484E44E
:1004500090E0FC01208184E490E02260FC01208394
:1004600084E490E0FC01208184E490E02160FC01C0
:10047000208385E490E0FC01208185E490E0226007
:10048000FC01208385E490E0FC01208185E490E07C
:100490002160FC0120838EE690E0FC0120818EE645
:1004A00090E02160FC01208381E890E0FC01108253
:1004B00081E890E0FC01208181E890E02260FC016D
:1004C000208381E890E0FC01208181E890E02160B8
:1004D000FC01208380E890E0FC01208180E890E02E
:1004E0002160FC01208381EB90E0FC01208181EB05
:1004F00090E02460FC01208380EB90E0FC012081EF
:1005000080EB90E02160FC0120838AE790E0FC0111
:1005100020818AE790E02460FC0120838AE790E054
:10052000FC0120818AE790E02260FC0120838AE7B9
:1005300090E0FC0120818AE790E02160FC012083AB
:100540008AE790E0FC0120818AE790E02068FC01C6
:10055000208381EC90E0FC011082DF91CF9108951F
:10056000CF93DF93CDB7DEB7DF91CF910895F894A5
:02057000FFCFBB
:00000001FF
```

Fig. 13. Firmware file without a bootloader and without optimization for the operation of adding two single-byte variables using local variables and without using additional functions

An example of a firmware file generated by the Arduino IDE version 2.3.4 development environment with a bootloader without optimization, with a size of 1896 bytes, compatible with Arduino Classic microcontrollers for the operation of adding two single-byte variables using local variables and without using additional functions is shown in Fig. 14.

As can be seen from Fig. 14, the size of the bootloader occupies 502 bytes in FLASH memory. This is approximately 1.56 % of the available FLASH memory for the ATMega328P microcontroller. For the ATMega16 microcontroller, this will be approximately 3.11 % of the available memory. And for the ATMega8 microcontroller, respectively, 6.21 %.

Ways to improve code efficiency. First of all, to improve the efficiency of code written for the Arduino platform, based on and within the framework of the research conducted, the following steps can be suggested:

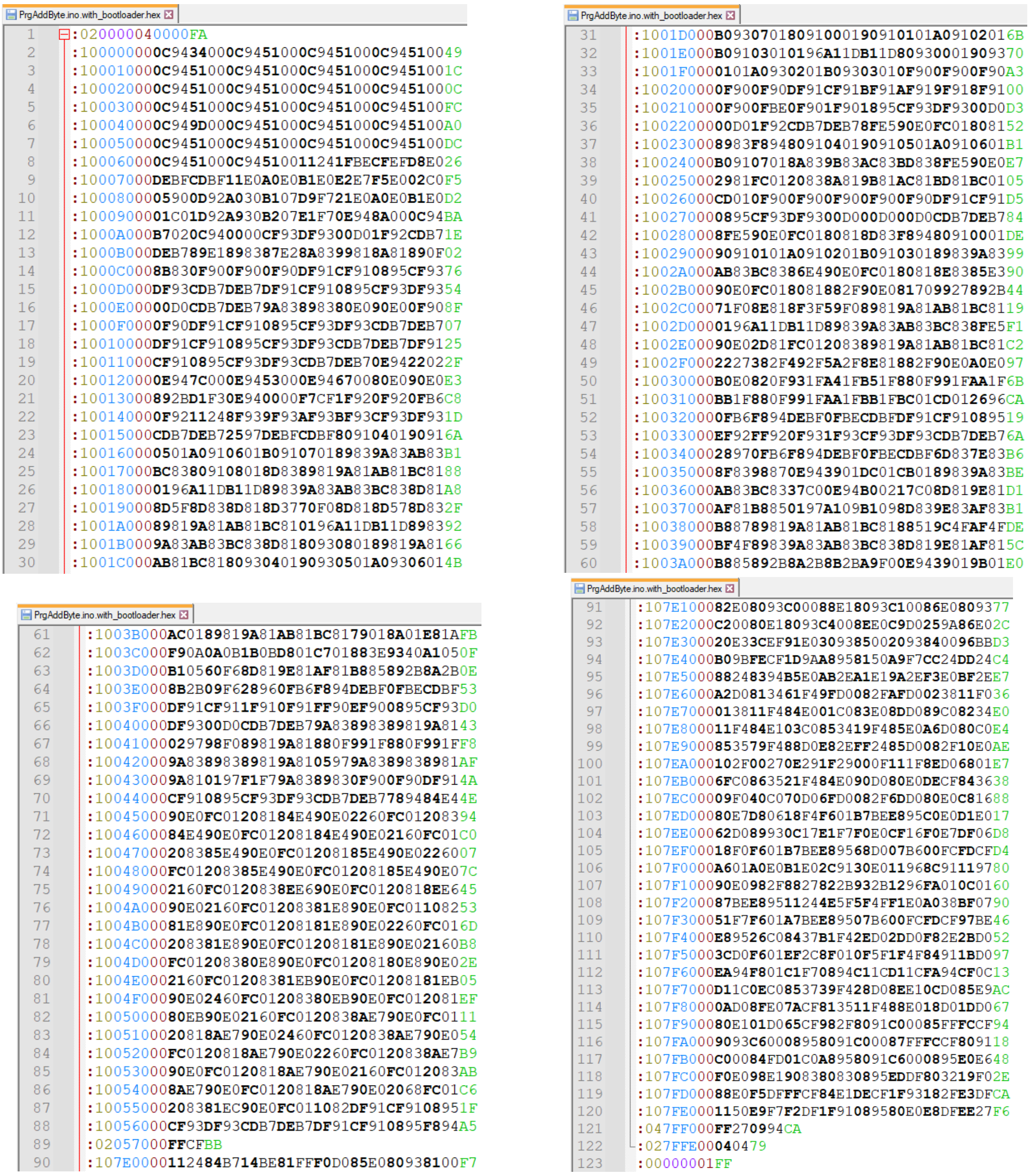

Fig. 14. Firmware file with a bootloader without optimization for the operation of adding two single-byte variables using local variables and without using additional functions

1. Use variables effectively. You need to use the correct data types, namely byte (0–255) instead of int if the variable does not exceed this range or int instead of long. Uint8_t, uint16_t, uint32_t instead of int if unsigned numbers are required. Bool instead of int if the variable only accepts true or false. Const or #define for immutable values to avoid wasting RAM.

2. Reduce the use of float data type. Working with float data type slows down code execution and takes up more memory. It is advisable to use integers, for example, multiply by 100 and work with int type if precision to two digits is required.

3. Choose the optimal way to store strings, for example char array[] consumes less memory than String. Use F() to store strings in FLASH memory:

Serial.println(F("This text will be stored in FLASH memory, not RAM!"));

4. The Arduino IDE also has an internal code optimizer that you can use to optimize your sketch. Let's demonstrate how it optimizes our code for byte data types (Fig. 15).

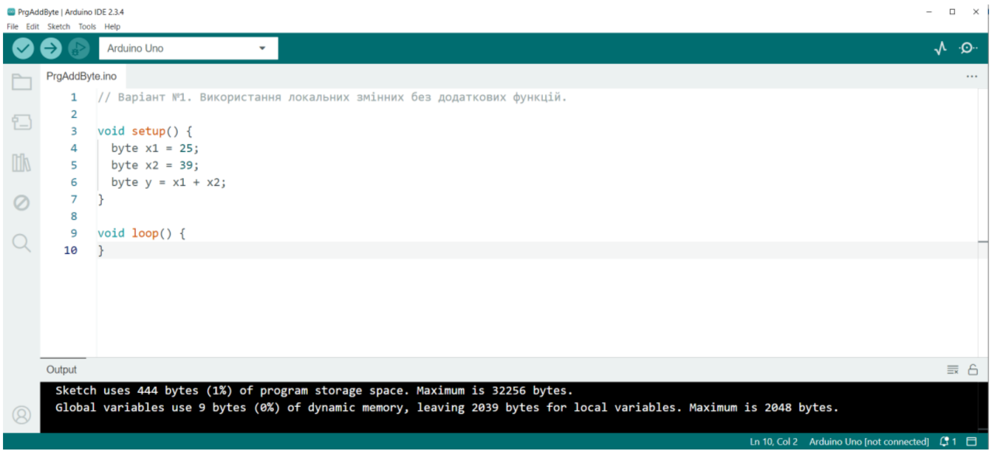

Fig. 15. Using the built-in code optimizer to optimize a written sketch

As can be seen from Fig. 15, the size of the firmware file generated by the Arduino IDE version 2.3.4 environment after optimization but without the bootloader takes up 444 bytes. This is approximately 1.38 % of the available FLASH memory for the ATMega328P microcontroller. For the ATMega16 microcontroller, this will be approximately 2.75 % of the available memory. And for the ATMega8 microcontroller, respectively, 5.5 %. Together with the bootloader, after optimizing the generated code, the program will take up 946 bytes (444 bytes of the code itself, which implements the operation of adding two single-byte variables + 502 bytes of the bootloader). In percentage terms, this percentage will already be 11.73 % of the available FLASH memory for the ATMega8 microcontroller. That is, the increase in the size of the firmware file when using the bootloader is obvious.

5. If possible, abandon the bootloader. Such abandonment will slightly complicate the programming of the microcontroller, but will save 502 bytes of valuable FLASH memory.

An example of a firmware file generated by the Arduino IDE version 2.3.4 development environment without a bootloader but with optimization, with a size of 444 bytes, compatible with Arduino Classic microcontrollers for the operation of adding two one-byte variables using local variables and without using additional functions is shown in Fig. 16.

Fig. 16. Optimized firmware file for the operation of adding two byte variables without a loader

PrgAddByte.ino.hex | PrgAddByte.ino.with_bootloader.hex

```
:020000040000FA
:100000000C9434000C9446000C9446000C9446006A
:100010000C9446000C9446000C9446000C94460048
:100020000C9446000C9446000C9446000C94460038
:100030000C9446000C9446000C9446000C94460028
:100040000C9448000C9446000C9446000C94460016
:100050000C9446000C9446000C9446000C94460008
:100060000C9446000C94460011241FBECFEFD8E03C
:10007000DEBFCDBF21E0A0E0B1E001C01D92A930FC
:10008000B207E1F70E9492000C94DC000C9400008F
:100090001F920F920FB60F9211242F933F938F93BD
:1000A0009F93AF93BF938091050190910601A0911A
:1000B0000701B09108013091040123E0230F2D378F
:1000C00058F50196A11DB11D209304018093050 1EF
:1000D0009093060 1A0930701B09308018091000 15D
:1000E0009091010 1A0910201B09103010196A11D1F
:1000F000B11D809300019093010 1A0930201B09380
:100100000301BF91AF919F918F913F912F910F90DC
:100110000FBE0F901F90189526E8230F0296A11D81
:10012000B11DD2CF789484B5826084BD84B58160DE
:1001300084BD85B5826085BD85B5816085BD8091B2
:100140006E00816080936E0010928100809181002A
:10015000826080938100809181008160809381002 2
:1001600080918000816080938000809 1B1008460E4
:100170008093B1008091B0008160 8093B000809145
:100180007A00846080937A0080917A008260809304
:100190007A0080917A00816080937A0080917A0061
:1001A000806880937A001092C100C0E0D0E0209770
:0C01B000F1F30E940000FBCFF894FFCF99
:107E0000112484B714BE81FFF0D085E080938100F7
:107E100082E08093C00088E18093C10086E0809377
:107E2000C20080E18093C4008EE0C9D0259A86E02C
:107E300020E33CEF91E0309385002093840096BBD3
:107E4000B09BFECF1D9AA8958150A9F7CC24DD24C4
:107E500088248394B5E0AB2EA1E19A2EF3E0BF2EE7
:107E6000A2D0813461F49FD0082FAFD0023811F036
:107E7000013811F484E001C083E08DD089C08234E0
:107E800011F484E103C0853419F485E0A6D080C0E4
:107E9000853579F488D0E82EFF2485D0082F10E0AE
:107EA000102F00270E291F29000F111F8ED06801E7
:107EB0006FC0863521F484E090D080E0DECF843638
:107EC00009F040C070D06FD0082F6DD080E0C81688
:107ED00080E7D80618F4F601B7BEE895C0E0D1E017
:107EE00062D089930C17E1F7F0E0CF16F0E7DF06D8
:107EF00018F0F601B7BEE89568D007B600FCFDCFD4
:107F0000A601A0E0B1E02C9130E011968C91119780
:107F100090E0982F8827822B932B1296FA010C0160
:107F200087BEE89511244E5F5F4FF1E0A038BF0790
:107F300051F7F601A7BEE89507B600FCFDCF97BE46
:107F4000E89526C08437B1F42ED02DD0F82E2BD052
:107F50003CD0F601EF2C8F010F5F1F4F84911BD097
:107F6000EA94F801C1F70894C11CD11CFA94CF0C13
:107F7000D11C0EC0853739F428D08EE10CD085E9AC
:107F80000AD08FE07ACF813511F488E018D01DD067
:107F900080E101D065CF982F8091C00085FFFCCF94
:107FA0009093C6000895 8091C00087FFFCCF809118
:107FB000C00084FD01C0A8958091C6000895E0E648
:107FC000F0E098E1908380830895EDDF803219F02E
:107FD00088E0F5DFFFCF84E1DECF1F93182FE3DFCA
:107FE0001150E9F7F2DF1F91089580E0E8DFEE27F6
:047FF000FF270994CA
:027FFE00040479
:00000001FF
```

Fig. 17. Optimized firmware file for the operation of adding two byte variables with a loader

Results and discussion

An example of a firmware file with a bootloader and optimization, 946 bytes in size, generated by the Arduino IDE version 2.3.4 development environment, compatible with Arduino Classic microcontrollers for the operation of adding two one-byte variables using local variables and without using additional functions, is shown in Fig. 17.

As a result of the study, the efficiency of the generated code for the AVR architecture on the classic Arduino UNO R3 platform by the Arduino IDE version 2.3.4 development environment was analyzed, when performing basic arithmetic operations of addition and subtraction. The study included a comparison of code variants using variables of different types and forms of their initialization, as well as using different types of functions or without them. The sizes of the generated code by the Arduino IDE environment were considered and determined for the following ways of performing the addition operation: Local variables without additional functions; Global variables, an addition function without parameters and without returning a result; Global variables, an addition function without parameters with returning a result; Global variables, an addition function that accepts arguments and returns a result; Local variables, an addition function that accepts arguments and returns a result.

The optimal implementation of the addition operation on integer variables was found. It turned out to be a method using local variables and without using additional functions (Fig. 7). Therefore, in the future, for our experiments with other dimensions of operands, this method of implementing the operation under study was chosen. The worst version of the implementation of the addition operation in terms of the firmware file size of all those studied within the framework of this work was also found. It turned out to be a version of implementing the addition operation using global variables and an addition function that accepts arguments and returns a result.

The obtained research results are summarized in graphs for ease of visual perception and are presented in Fig. 18–20 with a detailed explanation of the results obtained.

The size of the bootloader for the Arduino Classic family of boards has been determined. It occupies 502 bytes of FLASH memory, which is approximately 1.56 % of the available FLASH memory for the ATMega328P microcontroller. For the ATMega16 microcontroller, this will be approximately 3.11 % of the available memory. And for the ATMega8 microcontroller, respectively, 6.21 %.

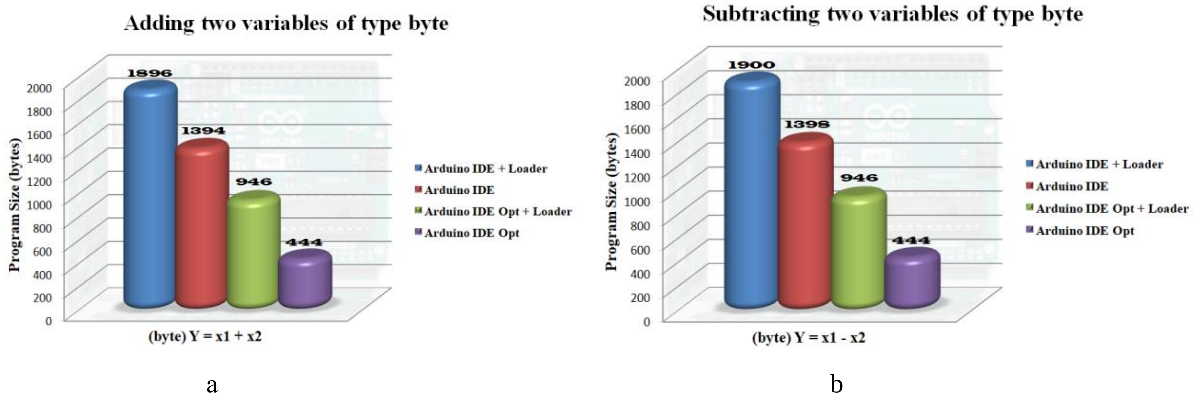

Fig. 18. Firmware file sizes for elementary arithmetic operations of two byte type variables:
a – for the addition operation; b – for the subtraction operation

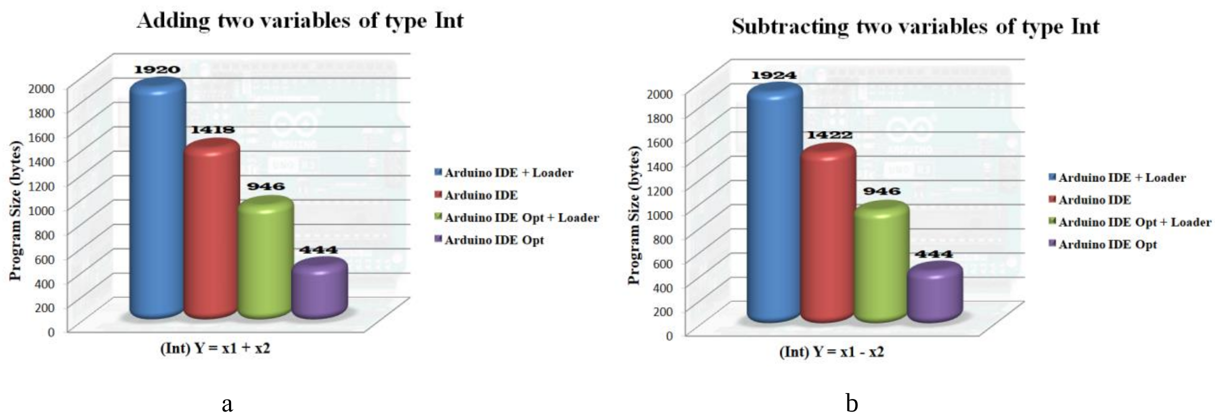

Fig. 19. Firmware file sizes for elementary arithmetic operations of two variables of type Int:
a – for the addition operation; b – for the subtraction operation.

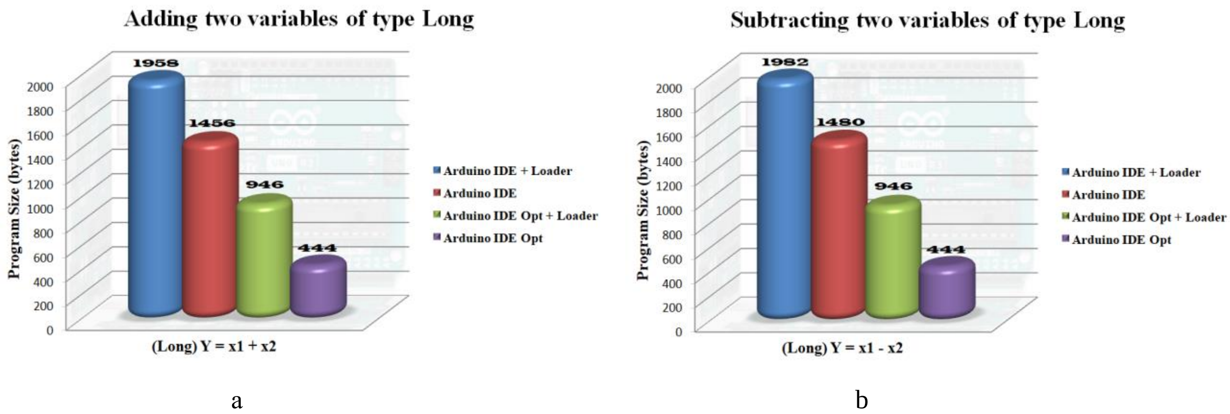

Fig. 20. Firmware file sizes for elementary arithmetic operations of two Long type variables:
a – for the addition operation; b – for the subtraction operation

For visual perception and better understanding of the material presented in the article, examples of microcontroller firmware files without a loader (Fig. 13) and with a loader (Fig. 14) but without optimization, as well as without a loader (Fig. 16) and with a loader (Fig. 17) but with optimization for the operation of adding two single-byte variables using local variables and without using additional functions are provided.

Based on the research, ways to improve the efficiency of code for sketches for the Arduino hardware platform are proposed. It is shown how effectively the internal code optimizer built into the Arduino IDE works.

Conclusions

The conducted research showed, using the example of elementary arithmetic operations of addition and subtraction, that the Arduino IDE development environment generates inefficient code. The situation improves if the built-in code optimizer is used correctly. With its help, the size of the firmware file with a loader for the operation of adding two single-byte variables was reduced by half. The size of the firmware file without a loader for the operation of adding two single-byte variables was reduced by 3.2 times. However, it can be said with confidence that there is great potential for further reduction of the firmware file size. Which can be the subject of further research.

The article also provides general recommendations for improving the efficiency of writing sketches for Arduino platforms. The main ones are: abandoning the bootloader; using constants instead of variables where possible, using data types with smaller bit sizes. It is shown that even simple arithmetic operations can have different hardware costs depending on the way they are written in the code.

These findings have practical implications for embedded system developers, allowing them to make more informed decisions when choosing data types and programming style to achieve maximum efficiency. In the future, the research can be extended to other types of operations, variables, and other microcontroller architectures.

References

[1] https://www.arduino.cc/ [E-resource]. Arduino hardware platform and Arduino IDE development environment.

[2] https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors [E-resource]. Microchip microcontrollers.

[3] Neil Cameron. Arduino Applied: Comprehensive Projects for Everyday Electronics. APRESS. ISBN 978-1-4842-3959-9. 2018. 552p. https://doi/org/10.1007/978-1-4842-3960-5_1

[4] Michael Margolis, Brian Jepson. Arduino Cookbook: Recipes to Begin, Expand, and Enhance Your Projects 3rd Edition. O'REILLY. ISBN 978-1-4919-0352-0. 2020. 800 p.

[5] Farzin Asadi. Essentials of Arduino™ Boards Programming: Step-by-Step Guide to Master Arduino Boards Hardware and Software. APRESS. ISBN 978-1-4842-9599-1. 2023. 332 p. https://doi/org/10.1007/978-1-4842-9600-4_1

[6] https://projecthub.arduino.cc/me_yogesh/diy-iot-plant-watering-system-using-arduino-b00eb3 [E-resource]. Plant Watering System.

[7] https://projecthub.arduino.cc/diytechos786/portable-mini-weather-station-real-time-weather-updates-with-web-configuration-1fa024 [E-resource]. Portable Mini Weather Station.

[8] https://projecthub.arduino.cc/rajeshjiet/iot-based-health-monitoring-system-arduino-project-27f2ba [E-resource]. IoT based health monitoring system.

[9] https://projecthub.arduino.cc/crepeguy/jumpman-lcd-game-c9aea0 [E-resource]. jumpman lcd game.

[10] https://projecthub.arduino.cc/Nagarajan-S/reprap-3-dimentional-additive-manufacturing-printer-with-iot-37621f [E-resource]. RepRap 3-Dimentional Additive Manufacturing Printer with IOT.

[11] https://projecthub.arduino.cc/Aboubakr_Elhammoumi/real-time-data-acquisition-of-solar-panel-using-arduino-9c72ef [E-resource]. Real-Time Data Acquisition of Solar Panel Using Arduino

[12] https://projecthub.arduino.cc/ghemml/cnc-arduino-winding-machine-652378 [E-resource]. CNC Arduino Winding Machine

[13] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools, 2nd Edition, 2007, 966 p.

[14] Niklaus Wirth, Jürg Gutknecht. Project Oberon. The Design of an Operating System and Compiler, Edition 2005, 441 p.

Руслан Головацький

Кафедра систем автоматизованого проєктування, Національний університет “Львівська політехніка”, вул. С. Бандери, 12, Львів, Україна, E-mail: ruslan.i.holovatskyi@lpnu.ua, ORCID 0009-0001-3096-1115

ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ГЕНЕРУВАННЯ КОДУ СЕРЕДОВИЩЕМ РОЗРОБКИ ARDUINO IDE НА ПРИКЛАДІ АРИФМЕТИЧНИХ ОПЕРАЦІЙ ДОДАВАННЯ ТА ВІДНІМАННЯ

Отримано: Вересень 01, 2025 / Переглянуто: Вересень 09, 2025 / Прийнято: Вересень 15, 2025

Анотація. У статті розглянуто ефективність генерування коду середовищем розробки Arduino IDE під час виконання елементарних арифметичних операцій додавання та віднімання. Це середовище є популярним інструментом серед розробників для роботи з мікроконтролерами, оскільки має зручний інтерфейс для швидкого прототипування. Однією з ключових характеристик таких середовищ є якість згенерованого коду, що впливає на швидкість виконання програм, використання пам’яті та загальну продуктивність системи. Тому дослідження ефективності генерування коду вказаним середовищем розроблення є актуальним завданням. У цьому дослідженні проаналізовано продуктивність, використання пам’яті та оптимізацію коду компілятором. Наведено експериментальні результати та зроблено висновки щодо доцільності використання середовища для задач із високими вимогами до продуктивності.

Ключові слова: Arduino IDE, генерування коду, ефективність, арифметичні операції, додавання, віднімання, мікроконтролери, AVR, файл прошивки.

* Corresponding author