Assessing the Impact of a Source Level Optimization Utility for Embedded Systems

by

Tanner Lisonbee

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2021 by the
Graduate Supervisory Committee:

Robert Heinrichs, Chair
Ruben Acuña
Shawn Jordan

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

Embedded software is different in many aspects to traditional software; as such, a software developer may face issues when attempting to transition from traditional to embedded software development. This thesis explores providing feedback and applying optimizations at the source code level of embedded software. The aim is to measure the impact of these optimizations on teaching embedded software design principles, as well as assessing the relative success of each optimization in terms of a variety of metrics.

There are many considerations when altering code and is a known limitation imposed by most software optimization schemes. By applying optimizations at the source level, the aim is to demonstrate what the optimizations do and how they provide value to the resulting software. In order to fulfill these goals, the Embedded C Source Optimizer has been developed, which is used to import and export code, select which optimizations are applied, and provide feedback to the end user.

This utility abstracts away the lower level operations performed by each optimization, while conveying the resulting changes to the end user. Since embedded systems are generally quite limited compared to modern computers, someone transitioning from traditional software design to embedded software may find it challenging to understand how to overcome these limitations. Clearly conveying means to improve a naive implementation of an embedded program aids through demonstrating what changes need to be made to satisfy embedded design rules.

The optimizations which the utility can apply range from simple replacement operations to more complex applications of implicit utilization of built-in hardware peripherals on supported microcontrollers. Each optimization comes with its own set of considerations, risks, and potential level of improvement to the resulting code. These optimization options are evaluated by comparing embedded software before and after each option is applied through a variety of metrics, allowing the relative success of each to be determined as effectively as possible.

The end goal for this utility is to aid in crossing the hurdle from traditional software to embedded software in a comprehensive and educational manner, with the provided optimization options acting as an avenue for teaching embedded concepts.

TABLE OF CONTENTS

CHAPTER Page

APPENDIX

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

LIST OF ABBREVIATIONS

ADC	Analog-to-Digital Converter
CCCC	C and C++ Code Counter
CFG	Control-Flow Graph
CoFlo	C and C++ Control Flow Graph Generator and Analyzer
CV	Coefficient of Variation
DC	Direct Current
EEPROM	Electrically Erasable Programmable Read-Only Memory
GCC	GNU Compiler Collection
GoF	Gang of Four
GPIO	General Purpose Input/Output
GPLv3	GNU General Public License v3.0
GUI	Graphical User Interface
I/O	Input / Output
IPC	Instructions Per Clock
ISR	Interrupt Service Routine
LED	Light-Emitting Diode
LLVM	Low Level Virtual Machine
LOC	Lines of Code
NOP	No Operation
MCC	MPLAB X Code Configurator
MCU	Microcontroller Unit
MVC	Model-View-Controller
PIC	Programmable Intelligent Computer
PWM	Pulse-Width Modulation

RAM	Random Access Memory
SLOC	Source Lines of Code
SRAM	Static Random Access Memory
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
WDT	Watchdog Timer

Chapter 1

INTRODUCTION

Embedded software design is different in many aspects to more traditional software; as such, an entry-level software developer may face a variety of issues when attempting to transition from traditional software projects to embedded ones. Aspects such as: code size, memory usage, hardware support, utilization of peripherals, are all critical facets of embedded software design which tend to have minimal to no consideration in traditional software. This required shift in mindset can be a daunting task for such a novice embedded developer to overcome, and has the propensity to discourage or even turn them off from embedded software development entirely.

With such a steep learning curve and a general lack of rudimentary information (outside of courses provided at formal educational institutions), there is a prominent gap in aiding developers in learning not just how to write embedded code, but why embedded code must be written in a certain way and the concepts / design patterns that come with it. It should be noted that there are many online sources for learning embedded concepts, but due to the fragmented nature of embedded hardware (and therefore how software is written for a given platform), it is not always straightforward to search for and find information which is appropriate for the task at hand. There is an opportunity to aid in filling this gap by providing a self-contained solution which aids the end user in not only becoming familiar with embedded concepts, but to explicitly demonstrate how their existing code can be altered to more effectively utilize embedded hardware while maintaining the same provided functionality. This thesis explores the process of designing, creating, and analyzing a tool which attempts to accomplish the goal defined previously.

Chapter 2

BACKGROUND

There are a variety of aspects which differ between traditional and embedded software design (which will be detailed in Section 2.1). Understanding these differences is critical to not only realizing what the primary issues are with this transition, but how the demonstrated Embedded C Source Optimizer utility is useful in easing the transition to embedded software design. There are a variety of existing solutions for automatically applying optimizations to existing embedded software, as well as documented methods of manually optimizing code. Furthermore, there are tools which allow for building programs at a higher level than traditional source code (some of which are provided in Subsection 2.2.1). While these are valid solutions for their intended use cases, there remains a potential gap between such solutions and easing the aforementioned transition to embedded software design. Discussion concerning filling this gap will be detailed in Subsection 2.2.2.

2.1 Fundamentals

Embedded software (depending on the hardware being used) is generally very compact and low-level in an effort to minimize the resources used and maximize performance, taking into consideration the limitations imposed by the hardware. Many introductory embedded development platforms utilize an 8-bit Microcontroller Unit (MCU) [3] which, in comparison to traditional computers, can be limited in Instructions Per Clock (IPC), Static Random Access Memory (SRAM) capacity, Input / Output (I/O) throughput, and connectivity [29]. As such, it is critical to write software which is appropriate for the hardware being used to achieve the intended result.

2.1.1 Embedded Code & Fixed-Function Hardware

Embedded hardware in the form of MCUs exists as a means of integrating relatively simple and low-cost logic into a given design/product. These MCUs are intentionally simplistic in nature in order to facilitate the creation of products requiring specialized logic to handle some portion of its functionality. Since the desired functionality is often so specialized for the product in question, the hardware itself is built to provide exactly the required amount of performance per cost for the product in which it is being integrated into [29]. The limitations imposed on software developers by this restrictive hardware set requires a significant paradigm shift from traditional software development, where many specific considerations must be made to accomplish any end goal in terms of: functionality, efficiency, power consumption, etc.

Modern MCUs contain a variety of specialized hardware units, referred to as ”peripherals,” which can be configured to perform tasks with minimal to no impact on the core of the MCU. These peripherals require specific registers to be configured correctly in order to obtain the intended functionality. This has the potential to be lost on novice embedded developers due to the differences in peripherals included between MCU architectures and families, as well as the specificities of configuring each peripheral. Such a developer may not even know about the existence of a given peripheral, further compounding the issue. It is expected that any required information needed to set up these peripherals can be found in the corresponding datasheet/documentation for the hardware being used, but without knowing exactly what to look for this can become quite an arduous task.

Having a simple utility to not only provide background on these peripherals, how they work, and how to configure them properly, but also attempt to apply changes to a piece of source code in order to utilize them implicitly is one of the primary focuses of the Embedded C Source Optimizer. Such an all-encompassing tool could improve understanding in novice embedded developers through demonstrating what changes should be made for a given peripheral without digging through documentation.

Despite the variations between families and architectures, there are many hardware peripherals which have become standard on most embedded hardware. For the sake of example, some of these peripherals found on the ATmegaXX8 family of MCUs include [6, 13]:

- Counter/Timer
  - Using the primary clock source on the MCU, this is used to count a number of cycles over a duration of time. A given timer instance need only be configured once, and from then on it will continue counting with no intervention from the MCU core. A prescaler value may be set to reduce the frequency which the counter increments, and in combination with the known primary clock source, can be used to calculate when a certain duration of time has passed. There are interrupts which can be enabled to execute an Interrupt Service Routine (ISR) when the counter reaches a specified value, or when the counter overflows.

The ATmegaXX8 line of MCUs contain multiple separate timer instances [6], allowing for more complex operations to be controlled, and many independent operations to be performed at different time intervals and frequencies.

- Pulse Width Modulation (PWM)
  - Generally included as a specific utilization of an existing counter/timer instance, PWM channels can be configured to output a fixed or variable frequency square wave. The output depends on two factors: duty cycle and switching frequency. Duty cycle defines how long the signal is held high relative to the total pulse. If a pulse lasts 10 ms and the signal is held high for 7 ms (with the remaining 3 ms being held low) then the duty cycle is represented as 7 / 10, or 70% (refer to Figure 2.1 for other examples of how duty cycle influences the resulting signal). Switching frequency corresponds to how long a pulse lasts, which is defined by the time between rising or falling edges. From the previous example, a pulse that lasts 10 ms would have a switching frequency of 10 ms / 1000 ms, or 100 Hz (meaning there are 100 pulses per second). Combining these two attributes allows for virtually any digital [1] wave to be generated. If the switching frequency is high enough, the output can be used as a direct replacement for analog voltage control since the average voltage which is generated is proportional to the input voltage times the duty cycle [14].

[1] A digital signal corresponds to a signal comprised of only two states (being on or off), whereas an analog signal is a continuous wave made up of infinitely-many states. Analog signals cannot be generated directly by PWM since the PWM output can only be on or off; however an analog signal can be approximated by continuously varying the duty cycle over time (thus varying the average output voltage over time), and using external components (such as capacitors) / circuits to ”smooth” the output signal to more closely match the intended analog wave.

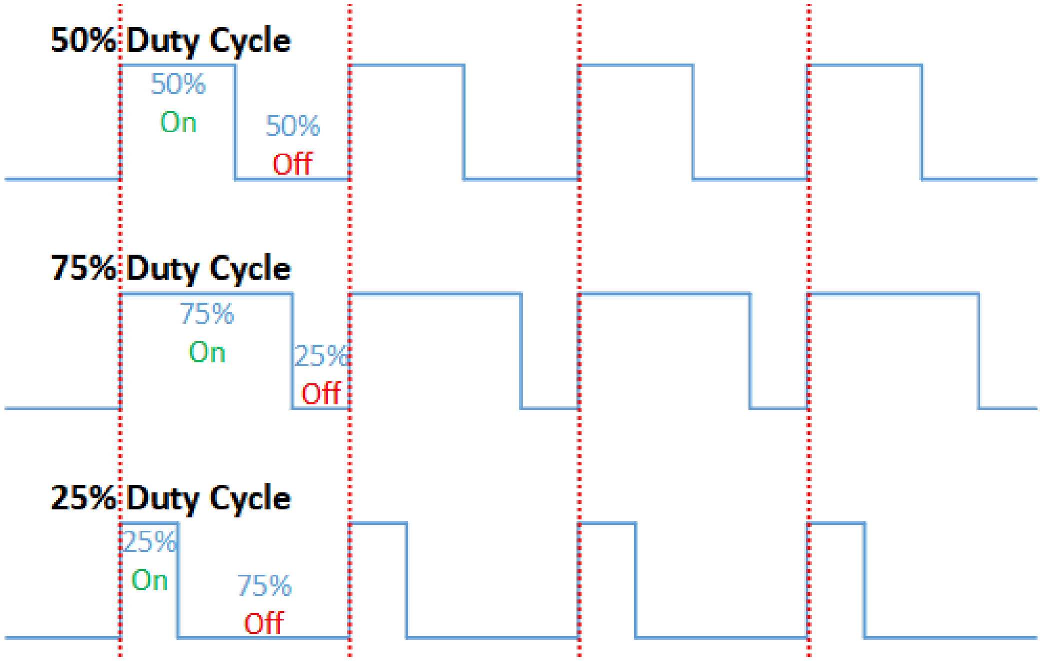


Figure 2.1: Visual example of PWM output with varying duty cycles [41]. These square wave signals with a fixed duty cycle over time will produce a constant average output voltage which is proportional to the high level voltage times the duty cycle. The dashed red lines running vertically correspond to the start of a signal's period, where the duty cycle is a proportion of the time the signal is on versus off for the duration of the period.

- Interrupt Handler Unit
  - In the context of embedded hardware, interrupts are a means to alter the flow of a program at runtime depending on the state of the system. There are two categories of interrupts in this case: internal and external. Internal interrupts are triggered by the MCU based on the result of some internal operation. An example of an internal interrupt would be a counter/timer instance overflowing (assuming it is configured to do so), and jumping to a separate subroutine. On the other hand, external interrupts are triggered outside of the MCU. An instance of an external interrupt might be a sensor or button setup to hold a pin high when active/pressed. Again, assuming the hardware is configured to do so, this signal can interrupt the flow of the program while it is executing and jump to a separate subroutine. The subroutine the interrupt causes the MCU to jump to

is known as an ISR, and may contain code to handle or respond to the action which triggered the interrupt. To achieve this functionality, a separate piece of hardware is used on the MCU to update the program counter and store the state of the remaining registers on the stack. All of this comes together to enable out-of-order execution which allows for better control over dynamic program execution [6].

- Analog Comparator
  - A relatively simple piece of hardware, this peripheral allows for comparing two analog inputs, and based on a variety of cases (one signal is higher than the other, the two signals are the same voltage, etc.) can interact with other peripherals to alter the flow of execution or modify the state of the MCU. For instance, the comparator can be set up to ”trigger the Timer/Counter1 input capture function” [6] or trigger an interrupt, causing the MCU to jump to a defined ISR. This peripheral may be used in a variety of cases from monitoring other components of an electrical circuit, to detecting when analog sensors exceed or drop below a specified voltage level [6].
- Watchdog Timer (WDT)
  - This specialized timer is used to ensure that the execution time of a program does not exceed a certain limit before reseting the timer. The ATmegaXX8 allows for a wide range of timeout values to be selected and includes a corresponding instruction to reset the timer. The use cases for the WDT include setting time limits for tasks within a program, reseting the MCU in the case that execution was halted due to a bug or other external factor, and/or safely shutting down the system in the case of a critical failure [6].

- Analog-to-Digital Converter (ADC)
  - The ADC peripheral is used primarily to measure analog values from an external source (such as an analog temperature sensor). The ADC internally uses an analog comparator to measure the external analog signal against a known reference voltage generated by the MCU. By repeatedly comparing the external signal to a known reference, a digital representation of that signal can be stored. The ATmegaXX8 has a 10-bit ADC, meaning each time an analog signal is measured, the process of comparing the external signal to a known reference voltage and determining which is greater is repeated 10 times in total. 10 bits of resolution yields 2^{10} possible analog values which can be interpreted and stored as a digital representation.

These peripherals represent fixed-function hardware which can be utilized to greatly expand the functionality of a given MCU. Choosing to perform tasks in software rather than using these peripherals has the potential to negatively impact a variety of aspects of the resulting program. This becomes somewhat of a double-edged sword however due to the relative complexity involved in configuring these peripherals. This gap may be filled through being able to take an existing software-based approach and having a utility attempt to configure the appropriate peripherals with little to no intervention from the programmer.

2.1.2 Embedded Optimization Methods

Software optimization has become a standard for many use-cases, especially in the embedded space. While not provided by default, compiling an embedded program under GNU Compiler Collection (GCC) [25] with the included size optimization is standard for embedded programs. This size optimization applied by the compiler is extremely useful in shrinking the final size of the resulting firmware to more effectively fit on the likely limited flash storage of a given MCU. As with any kind of software optimization, there is a chance for alterations to the firmware which impact the functionality and introduce unexpected behavior. This issue becomes more prominent when considering these compiler optimizations are applied at such a low level. Investigating why the functionality of a program changed after compiling with optimizations involves attempting to read and debug assembly level code, which is far less intuitive to do than working with source code. By applying optimizations at the source code level, it not only makes debugging potential issues with the optimization much more straightforward, but it also opens an avenue to better demonstrate to the end-user why changes were made in the first place.

Fundamentally, software optimizations are simply a means to improve various aspects of a program without changing the intended functionality. ”Optimization” is a term which covers many aspects in computing; it may refer to changes made by a software developer to improve their code, by the compiler to indicate it altered the generated assembly code to account for the hardware being used, or even at the hardware level for increasing the number or frequency of instructions being executed over a fixed period of time. Within the scope of this thesis, optimization is associated with the changes made at the source code level (the level of code which the end-user writes/modifies) which improve the software in terms of size, efficiency, and performance [21]. By altering source code directly, it is possible to affect the functionality of the resulting program independent of any compiler-level optimizations performed. Furthermore, the Embedded C Source Optimizer utility aims to make the process of automatically applying optimizations at the source code more streamlined through implicit utilization of fixed-function hardware (peripherals) included in the majority of AVR MCUs.

2.2 State of the Art

Embedded software optimization is included as a standard option within GCC, offering a range of optimization options when compiling a program. That being said, there are not many alternative options when it comes to tools which aim to optimize embedded code, especially at the source code level. Some high-level methods which can be used to optimize programs include: selecting what compiler is used, how the compiler is configured, manually optimizing code through how it is implemented, and avoiding the compilation step entirely by writing code at the assembly or machine level [21]. Aside from relying on a compiler (such as GCC) to perform any optimizations at the assembly level, most of the means used to optimize source code must be done manually [21, 23, 30].

2.2.1 Existing Optimization Solutions

One example of an existing optimization tool which can be used with embedded hardware (such as AVR) is LLVM, which is a compiler infrastructure containing various optimization options and techniques [40]. Despite its name, the LLVM Foundation claims that the tools it supports ”[have] little to do with traditional virtual machines” [40] (thus not to be confused with Low Level Virtual Machine (LLVM)). Rather, LLVM uses an intermediate representation for source code, where any optimizations which are applied exist at this intermediate level. Despite the wide range of options the LLVM Core supports in the way of optimizations [32], applying optimizations to the intermediate form of code obfuscates any changes made and potentially makes it difficult to map them to the original source code. Furthermore, the optimizations provided by the LLVM core are ”target-independent” [40], and therefore may not be able to take full advantage of embedded hardware due to the differences between embedded hardware, even within the same architecture or product family.

An example of a source-level tool which aids in implicitly generating embedded code and setting up hardware peripherals is included in Microchip's MPLAB X IDE [35]. This framework can be setup with MPLAB X Code Configurator (MCC) [34] to obtain a graphical means of configuring hardware while having to write a minimal amount of source code manually. This high-level development tool is useful for configuring hardware peripherals without the need to cross-reference data sheets, as well as abstracting away the lower level operations associated with configuring hardware peripherals. That being said, MCC itself is not intended to be an optimization tool, rather a high-level means of developing embedded code [34]. Any optimization would still be done by the user and/or compiler.

Beyond automated tools, there are countless ways to optimize embedded code manually which have been used some time; some of which include: loop unrolling, function inlining, reducing padding in data structures, direct insertion of assembly code, and general algorithmic optimization through choosing to use algorithm implementations with more favorable time and/or space complexity. The specificities of each of these optimization methods are outside the scope of this thesis; however, the notable observation from these is that software optimization both on embedded systems and in general is not a novel idea, and has been a topic of research and discussion for decades [46]. The primary concern with these manual optimization methods lies in the level of research and prior knowledge needed to implement them.

2.2.2 Gaps in Existing Solutions

Despite the previously mentioned tools being useful in their own right, they all tend to lack the level of explanation and clarity for understanding *how* any applied optimizations work. Furthermore, it may not be immediately obvious which fundamental components must be utilized in the first place to achieve a certain result. For instance, MCC can be utilized to set up a PWM output to control a Light-Emitting Diode (LED) or other actuator;

however, a novice embedded developer may not intuitively realize that they need to use PWM to accomplish this since the same functionality can be more readily achieved with a loop and software delay functions. This disconnect may manifest into bad practices as there is generally no single source of information, beyond the respective datasheet or other provided documentation, which conveys this concept.

Chapter 3

RESEARCH TOPIC

One primary concern a beginner embedded software developer may have is a lack of clarity of fundamental key concepts used in embedded software that remain absent in traditional software. The variety of hardware, compilers, and other software tools in the embedded space can lead to a significant amount of overhead in determining how to most effectively utilize hardware for a given task. This is a challenging problem for a variety reasons. Different MCU architectures (and even within families of MCUs) require different compilers [22], and contain different hardware peripherals, memory / flash capacity, etc. [13], potentially requiring the developer to delve deep into datasheets and other supplemental information regarding exactly what hardware they are using. Even performing simple tasks often require sorting through hardware documentation, and even then the programmer must have some requisite knowledge of what they need to look for in the first place. This is also why other implicit code-generating tools are not always simple for a novice embedded programmer to properly utilize.

The proposed solution described in this thesis differs from other tools in that it does not make assumptions about the end-user beyond what hardware they are using. Instead of attempting to provide a different, potentially more intuitive method of performing the same coding tasks, this solution instead focuses on analyzing a given interpretation of an embedded program, and from that making judgments on what the provided code is attempting to do. By taking a more retroactive approach to code generation, it has the potential to become more clear to the end-user how to best utilize a given piece of hardware.

3.1 Filling the Gap

Tools such as Microchip's MCC excel at obfuscating how a given piece of hardware must be configured to perform an intended task. For a novice embedded software developer, a solution like MCC may be a powerful tool in quickly building embedded programs. Furthermore, the optimization options provided in the GCC and LLVM compilers perform exceptionally well at applying low-level optimizations to compiled code which improve the resulting program in many ways. That being said, these various tools do not necessarily provide the needed background to aid the end-user in understanding how and why things are done in a certain way. It is assumed that a beginner embedded software developer likely has other coding experience and a certain level of competency in software development (even if they have little experience with embedded code). For instance, a beginner embedded software developer may be able to easily write a simple program which performs its intended task; however the way it is accomplished may be less than ideal without first understanding the limitations imposed by the hardware. In the same vein, utilizing MCC to create an embedded program may not be intuitive to an individual without background knowledge of how functions or peripherals provided by a given piece of hardware work on a fundamental level.

Continuing the example previously mentioned in Subsection 2.2.2, simply writing an infinite loop containing the code to change the state of an output pin and a software delay to flash a LED on and off should provide the intended result, but is objectively a poor solution considering these delay functions will block execution on the core until they are complete [19]. The same result can be obtained through utilizing a built-in PWM channel with essentially no impact on the MCU core, leaving it open to perform other tasks. Such a basic example does not necessarily show the full picture, as it is not complex enough to warrant setting up the hardware appropriately, especially when a "quick and dirty" solution will suffice. That being said, as soon as the programmer wishes to add additional functionality to their code, suddenly the inefficiencies caused by such a solution will undoubtedly result in a multitude of other problems.

Naive or otherwise inefficient solutions are the use case which the Embedded C Source Optimizer presented in this thesis intends to aid in. A programmer can take their working program and pass it through the utility in order to determine if there are changes they should consider making, especially if they intended to add additional functionality later. Such a utility which can interpret program code and provide feedback to a new coming embedded programmer is potentially valuable to their learning experience. An individual who has come from a traditional software development background likely will not find it intuitive to pick up on the specificities of embedded software. Having an all-in-one solution for providing feedback on programs is an useful tool for being able to make the necessary connections to embedded concepts. The Embedded C Source Optimizer discussed here can act as the essential first step to making these connections. Furthermore, the ability to improve various aspects of a piece of software with minimal intervention from the developer has the potential to speed up development time while simultaneously not compromising on efficiency or performance.

3.2 Research Questions

The primary interests for the proposed optimization utility have to do with the aspects of the source code after any selected optimizations are applied, and how these changes are presented to the end-user to aid in their understanding of embedded concepts. Investigating the effectiveness of these will become paramount in the outcome of this project as far as determining whether or not the utility can make the transition from traditional to embedded software design more streamlined. Furthermore, the breadth and depth of the supported optimizations and corresponding documentation will influence the usefulness of the tool, taking it beyond just a simple educational exercise.

The research questions this thesis will attempt to answer include:

1. *How can embedded code be automatically altered/modified by a utility at the source code level in order to improve its resulting size, efficiency, and performance, while also preserving the originally-intended functionality?*

2. *How can existing software be mapped to fixed-function hardware in order to improve embedded programs?*

3. *How could automatically applied source-level optimizations aid a user's ability to comprehend and extend upon the resulting program, as well as be conveyed in a way which illustrates how the optimizations work?*

Chapter 4

EMBEDDED C SOURCE OPTIMIZER

This chapter details the Embedded C Source Optimizer's design and architecture, metrics for analyzing embedded code, supported optimization options, as well as further considerations concerning educational aspects of the utility. Section 4.1 discusses the implementation-specific details of the Embedded C Source Optimizer and its use cases. Section 4.2 breaks down metrics for measuring and conveying various aspects of embedded software, as well how the Embedded C Source Optimizer applies optimizations at the source code level. Finally, Section 4.3 briefly introduces how the utility may be used to teach embedded programming principles.

4.1 Software Design

The Embedded C Source Optimizer utility is designed with the end-user at the forefront of its operation, and does not make assumptions about the state of the provided embedded program. This implies that the utility is *not* intended to make a program which otherwise does not compile or run suddenly work. Rather the intended use case for this utility is taking a working, but suboptimal solution to a problem, and improving on it by enforcing embedded design principles, as well as taking the hardware peripherals provided by the target MCU into consideration. These aspects aim to improve the program in a variety of ways, as well as demonstrate to the end-user how and why any changes were made.

The Embedded C Source Optimizer is built using Java, utilizing the Swing library for all Graphical User Interface (GUI)-related aspects. Java was chosen due to its availability of GUI libraries and tools, accessibility of lower-level operations such as process management and file I/O, as well as its wide adoption across computing applications [47]. This application was built using only natively-included Java libraries and toolkits in an effort to limit its dependence on external tools/applications, as well as allow for greater control over the organization and functionality used to implement the various components used by the utility. The GUI of the Embedded C Source Optimizer is shown in Figure 4.1.

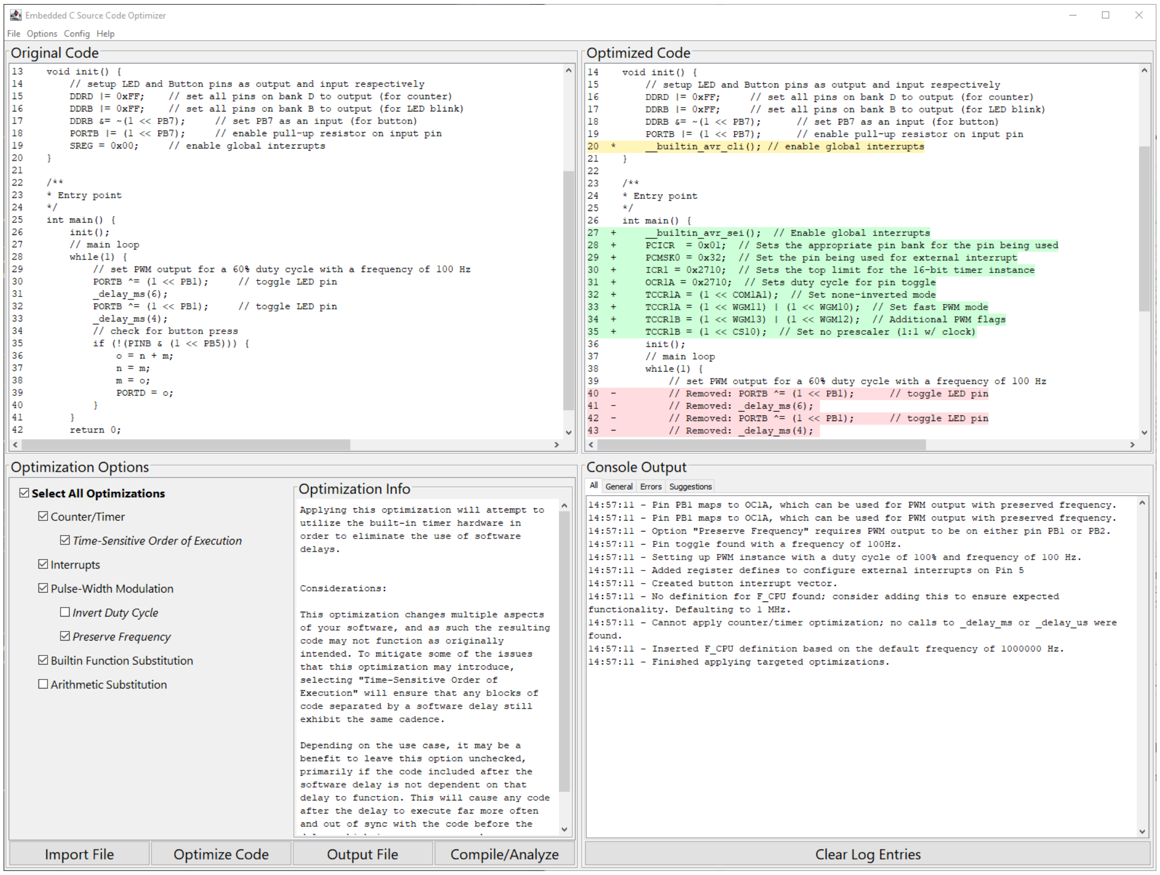
4.1.1 Fundamental Operation

Given the scope and schedule of this project, there are some limitations to what the Embedded C Source Optimizer is capable of in its current form. That being said, the primary use case of demonstrating to a novice embedded programmer how to improve their code is not lost on this. One of the most popular families of MCUs among beginners, the ATmegaXX8, is the primary target for this utility. Despite the Programmable Intelligent Computer (PIC) MCU architecture cumulatively selling more units [2], more hobbyists tend to start with AVR [24, 28, 42]. The initial testing for the utility was done targeting an ATmega168 [4]. The continuity of hardware support between different AVR MCUs means that the vast majority of MCUs in the same product stack will work with this utility to some degree [13].

At the most basic level, the Embedded C Source Optimizer searches through an embedded program written in C, attempting to match what is provided to determine if a better solution exists. In the case that the provided code is suboptimal and there is a known, better solution, the utility will attempt to add, remove, and/or modify the existing code depending on which optimization options the user has chosen to apply. The end-user is given control over which optimizations are applied, providing them the opportunity to better identify how each optimization works and which might best suit their needs. There are many considerations when it comes to altering source code, and as such it looks for very specific cases to ensure that the basic functionality of the provided program is not altered.

When a C source file is imported, the Embedded C Source Optimizer parses and displays the original code. The process of parsing the input program consists of matching keywords found in the C programming language to determine the general flow of execution. The code is broken up into a tree structure of “code elements,” which themselves represent a basic structure which may or may not be able to store other elements inside. To put this in perspective, a function would be considered a code element with other code elements nested within. A for loop in said function would be a child code element in relation to the function, and it itself can have other code elements nested within. As such, each code element can be observed as being either a branch or a leaf. Breaking up the source code in this manner allows for entire pieces of the program to be added, changed, or removed without affecting the other elements. More detail on the implementation of this structure will be provided in Subsection 4.1.3.

4.1.2 Intended use cases



Original Code

```
void init() {
    // setup LED and Button pins as output and input respectively
    DDRD |= 0xFF;    // set all pins on bank D to output (for counter)
    DDRB |= 0xFF;    // set all pins on bank B to output (for LED blink)
    DDRB &= ~(1 << PB7);     // set PB7 as an input (for button)
    PORTB |= (1 << PB7);     // enable pull-up resistor on input pin
    SREG = 0x00;     // enable global interrupts
}

/**
* Entry point
*/
int main() {
    init();
    // main loop
    while(1) {
        // set PWM output for a 60% duty cycle with a frequency of 100 Hz
        PORTB ^= (1 << PB1);     // toggle LED pin
        _delay_ms(6);
        PORTB ^= (1 << PB1);     // toggle LED pin
        _delay_ms(4);
        // check for button press
        if (!(PINB & (1 << PB5))) {
            o = n + m;
            n = m;
            m = o;
            PORTD = o;
        }
    }
    return 0;
```

Optimized Code

```
   void init() {
       // setup LED and Button pins as output and input respectively
       DDRD |= 0xFF;     // set all pins on bank D to output (for counter)
       DDRB |= 0xFF;     // set all pins on bank B to output (for LED blink)
       DDRB &= ~(1 << PB7);      // set PB7 as an input (for button)
       PORTB |= (1 << PB7);      // enable pull-up resistor on input pin
*      __builtin_avr_cli(); // enable global interrupts
   }

   /**
   * Entry point
   */
   int main() {
+      __builtin_avr_sei();  // Enable global interrupts
+      PCICR  = 0x01;  // Sets the appropriate pin bank for the pin being used
+      PCMSK0 = 0x32;  // Set the pin being used for external interrupt
+      ICR1 = 0x2710;  // Sets the top limit for the 16-bit timer instance
+      OCR1A = 0x2710;  // Sets duty cycle for pin toggle
+      TCCR1A = (1 << COM1A1);  // Set none-inverted mode
+      TCCR1A = (1 << WGM11) | (1 << WGM10);  // Set fast PWM mode
+      TCCR1B = (1 << WGM13) | (1 << WGM12);  // Additional PWM flags
+      TCCR1B = (1 << CS10);  // Set no prescaler (1:1 w/ clock)
       init();
       // main loop
       while(1) {
           // set PWM output for a 60% duty cycle with a frequency of 100 Hz
-          // Removed: PORTB ^= (1 << PB1);      // toggle LED pin
-          // Removed: _delay_ms(6);
-          // Removed: PORTB ^= (1 << PB1);      // toggle LED pin
-          // Removed: _delay_ms(4);
```

Optimization Options

☑ **Select All Optimizations**
- ☑ Counter/Timer
  - ☑ *Time-Sensitive Order of Execution*
- ☑ Interrupts
- ☑ Pulse-Width Modulation
  - ☐ *Invert Duty Cycle*
  - ☑ *Preserve Frequency*
- ☑ Builtin Function Substitution
- ☐ Arithmetic Substitution

Optimization Info

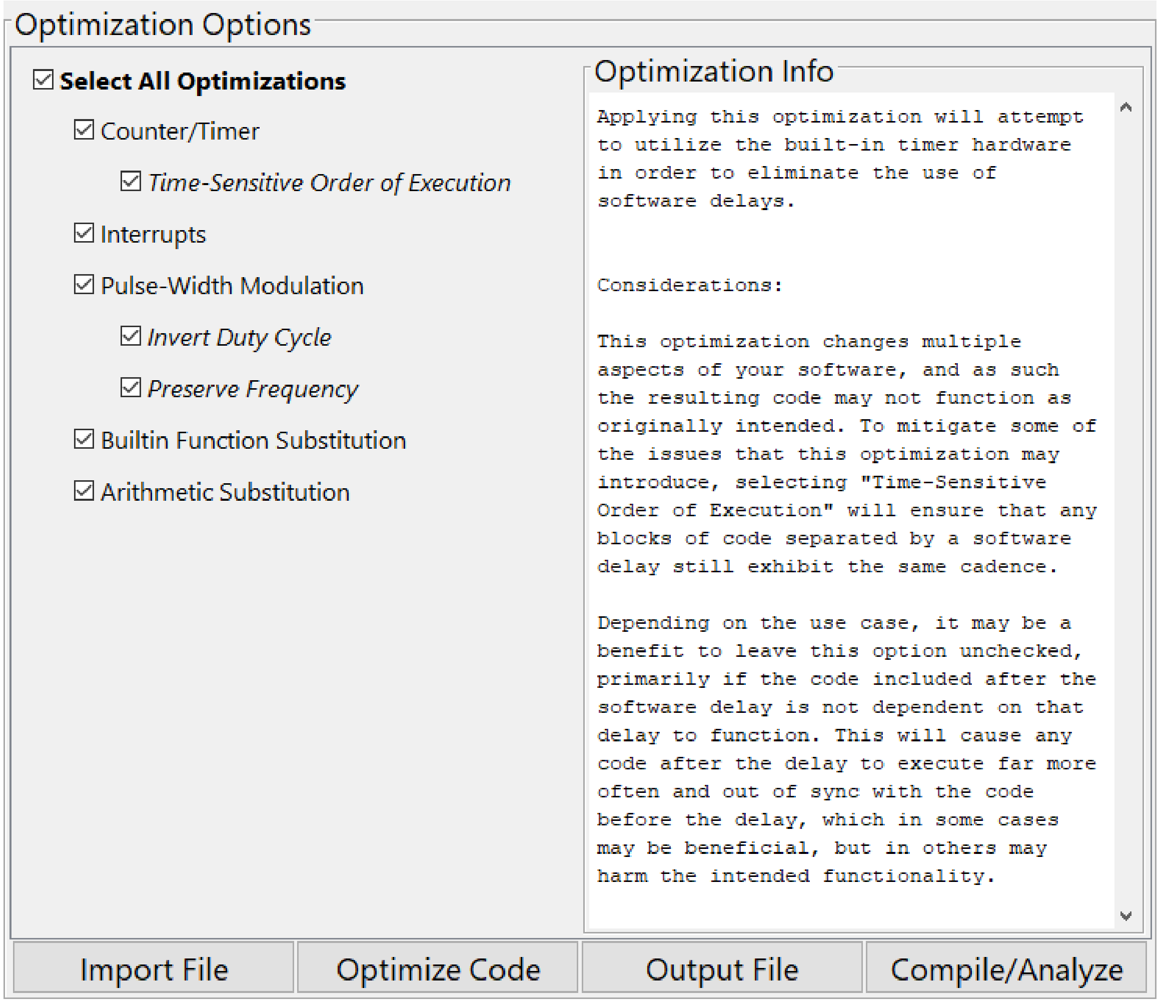
```
Applying this optimization will attempt to
utilize the built-in timer hardware in
order to eliminate the use of software
delays.


Considerations:

This optimization changes multiple aspects
of your software, and as such the resulting
code may not function as originally
intended. To mitigate some of the issues
that this optimization may introduce,
selecting "Time-Sensitive Order of
Execution" will ensure that any blocks of
code separated by a software delay still
exhibit the same cadence.

Depending on the use case, it may be a
benefit to leave this option unchecked,
primarily if the code included after the
software delay is not dependent on that
delay to function. This will cause any code
after the delay to execute far more often
and out of sync with the code before the
```

Console Output

```
14:57:11 - Pin PB1 maps to OC1A, which can be used for PWM output with preserved frequency.
14:57:11 - Pin PB1 maps to OC1A, which can be used for PWM output with preserved frequency.
14:57:11 - Option "Preserve Frequency" requires PWM output to be on either pin PB1 or PB2.
14:57:11 - Pin toggle found with a frequency of 100Hz.
14:57:11 - Setting up PWM instance with a duty cycle of 100% and frequency of 100 Hz.
14:57:11 - Added register defines to configure external interrupts on Pin 5
14:57:11 - Created button interrupt vector.
14:57:11 - No definition for F_CPU found; consider adding this to ensure expected
functionality. Defaulting to 1 MHz.
14:57:11 - Cannot apply counter/timer optimization; no calls to _delay_ms or _delay_us were
found.
14:57:11 - Inserted F_CPU definition based on the default frequency of 1000000 Hz.
14:57:11 - Finished applying targeted optimizations.
```

Figure 4.1: The Embedded C Source Optimizer GUI, showing the original user's code on the top left and the optimized code which was generated on the right. The optimization options are displayed as a list of check-boxes in the bottom left, and all the output from the utility on the bottom right.

As is shown in Figure 4.2, there is a list of optimization options which the user may select for their program. Upon selecting an option, a flag is set to attempt to apply that optimization on the imported code. Each option corresponds to a class with the necessary functionality contained within, which attempts to modify the existing code to implement that optimization. These options range from more complex functions which attempt to replace sections of code with implementations utilizing the built-in hardware peripherals on the selected MCU, to simple replacements attempting to save on code-size or cycle-time for certain instructions.



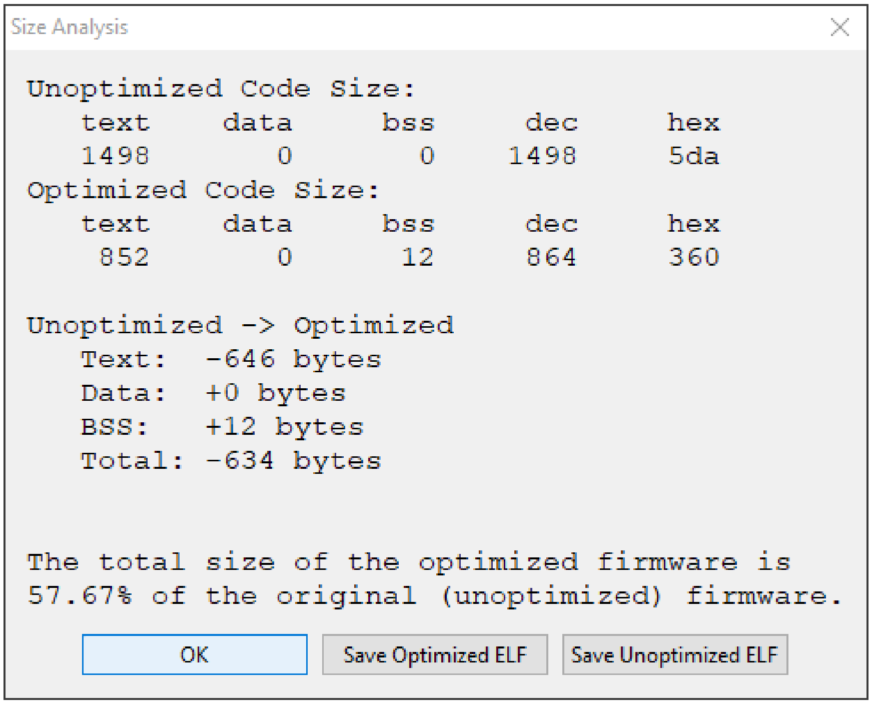
☑ **Select All Optimizations**
- ☑ Counter/Timer
  - ☑ *Time-Sensitive Order of Execution*
- ☑ Interrupts
- ☑ Pulse-Width Modulation
  - ☑ *Invert Duty Cycle*
  - ☑ *Preserve Frequency*
- ☑ Builtin Function Substitution
- ☑ Arithmetic Substitution

```
Applying this optimization will attempt
to utilize the built-in timer hardware
in order to eliminate the use of
software delays.


Considerations:

This optimization changes multiple
aspects of your software, and as such
the resulting code may not function as
originally intended. To mitigate some of
the issues that this optimization may
introduce, selecting "Time-Sensitive
Order of Execution" will ensure that any
blocks of code separated by a software
delay still exhibit the same cadence.

Depending on the use case, it may be a
benefit to leave this option unchecked,
primarily if the code included after the
software delay is not dependent on that
delay to function. This will cause any
code after the delay to execute far more
often and out of sync with the code
before the delay, which in some cases
may be beneficial, but in others may
harm the intended functionality.
```

Figure 4.2: The Embedded C Source Optimizer provides a variety of ”optimization options” which may be applied to an embedded program. These optimizations can be selected in any combination by the end-user to best fit their use case. Supplemental information is provided for each optimization option to increase the user’s understanding of what each optimization does and what they should expect in terms of changes it will make to their program. Furthermore, options are provided for importing C source code, applying the selected optimizations, exporting the resulting code, and compiling/performing a size analysis which compares the unoptimized and optimized version of the imported code.

As demonstrated in Figure 4.3, a selected set of optimization options (which will be discussed in greater detail in Subsection 4.2.3) can be applied and validated by compiling both versions of the program, and directly comparing the size of each section used by the final firmware. The standard output (shown in the Berkeley format by default [1]) gives indications of how the flash and Random Access Memory (RAM) are utilized by the firmware.

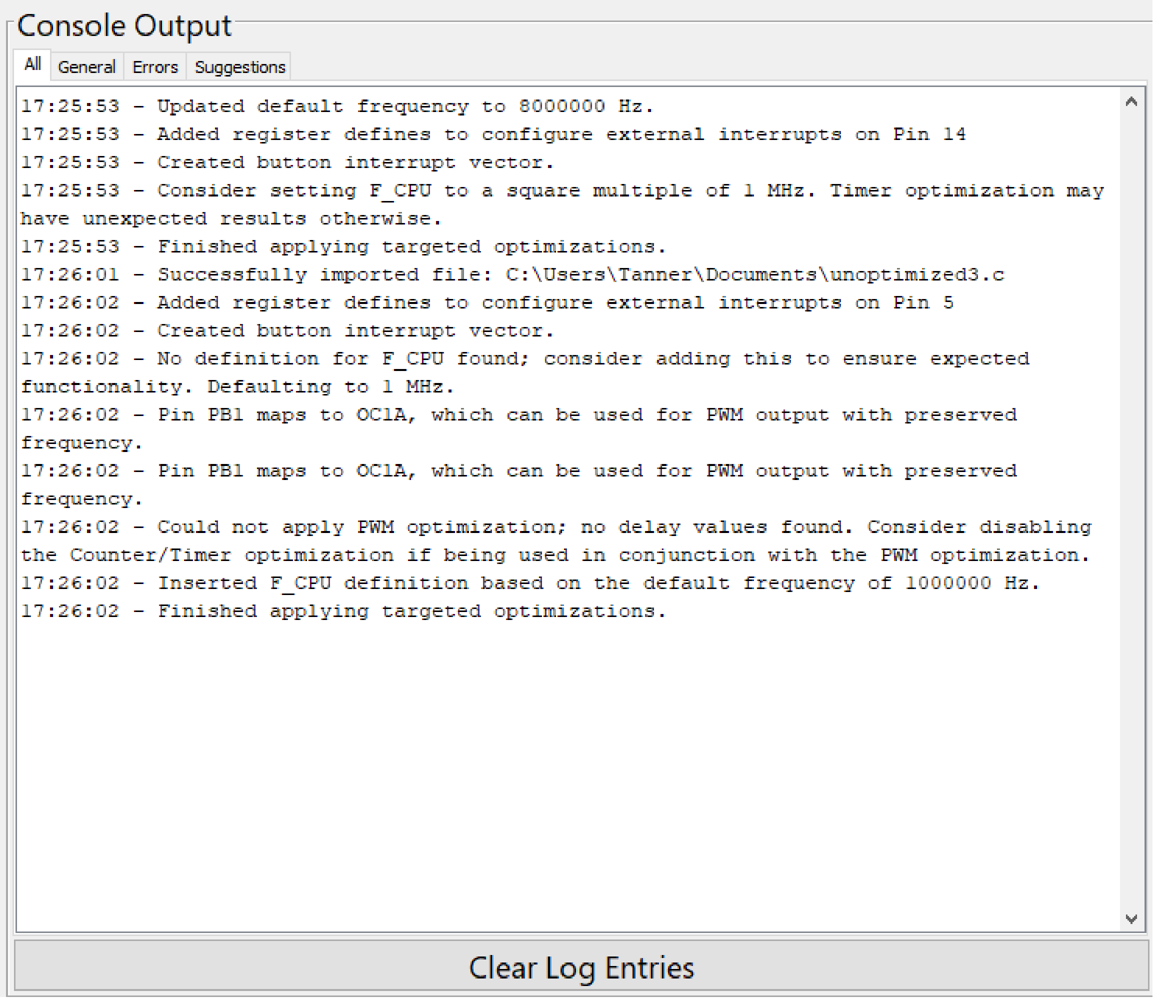
The output lists three distinct sections: ”text”, ”data”, and ”bss”, with the total of those displayed under the ”dec” and ”Total” labels. Additional information concerning what these sections of memory are used for will be provided in Subsection 4.2.1.



```
Unoptimized Code Size:
   text     data      bss     dec     hex
   1498        0        0    1498     5da
Optimized Code Size:
   text     data      bss     dec     hex
    852        0       12     864     360

Unoptimized -> Optimized
   Text:  -646 bytes
   Data:  +0 bytes
   BSS:   +12 bytes
   Total: -634 bytes


The total size of the optimized firmware is
57.67% of the original (unoptimized) firmware.
```

Figure 4.3: After selecting and running the desired optimization options, choosing the ”Compile/Analyze” option within the Embedded C Source Optimizer will compile the source code from both before and after the optimizations are applied. The ”avr-size” utility (included with the GNU AVR-GCC toolchain) is then called from the utility to display the measured difference in code size between both versions.

Figure 4.4 shows an example of the output generated by the Embedded C Source Optimizer when a set of selected optimizations are applied to a program. This GUI element allows filtering the log output to: general statements concerning how the selected optimizations are applied, errors which kept any selected optimizations from being applied, and suggestions for potential improvements that any given optimization option does not directly target.



```
17:25:53 - Updated default frequency to 8000000 Hz.
17:25:53 - Added register defines to configure external interrupts on Pin 14
17:25:53 - Created button interrupt vector.
17:25:53 - Consider setting F_CPU to a square multiple of 1 MHz. Timer optimization may
have unexpected results otherwise.
17:25:53 - Finished applying targeted optimizations.
17:26:01 - Successfully imported file: C:\Users\Tanner\Documents\unoptimized3.c
17:26:02 - Added register defines to configure external interrupts on Pin 5
17:26:02 - Created button interrupt vector.
17:26:02 - No definition for F_CPU found; consider adding this to ensure expected
functionality. Defaulting to 1 MHz.
17:26:02 - Pin PB1 maps to OC1A, which can be used for PWM output with preserved
frequency.
17:26:02 - Pin PB1 maps to OC1A, which can be used for PWM output with preserved
frequency.
17:26:02 - Could not apply PWM optimization; no delay values found. Consider disabling
the Counter/Timer optimization if being used in conjunction with the PWM optimization.
17:26:02 - Inserted F_CPU definition based on the default frequency of 1000000 Hz.
17:26:02 - Finished applying targeted optimizations.
```

Figure 4.4: The Embedded C Source Optimizer gives the end-user written output corresponding to: the actions taken by the utility to apply the selected optimization(s), any errors which kept the selected optimization(s) from being applied, and additional suggestions for the user-provided program which the utility does not directly target. This output can be filtered down to any one of these categories for improved readability.

4.1.3 Architectural Design

The architecture of the Embedded C Source Optimizer is reminiscent to the Model-View-Controller (MVC) pattern, in that there are highly cohesive classes (making up separate modules) for defining how the source code is modeled and stored, UI interactions, and functionality to control and manipulate the provided source code. The high-level modules are loosely coupled with one another, allowing for each to be expanded or changed with minimal impact to the rest of the system. Figure 4.5 shows how these modules and classes interact in greater detail.

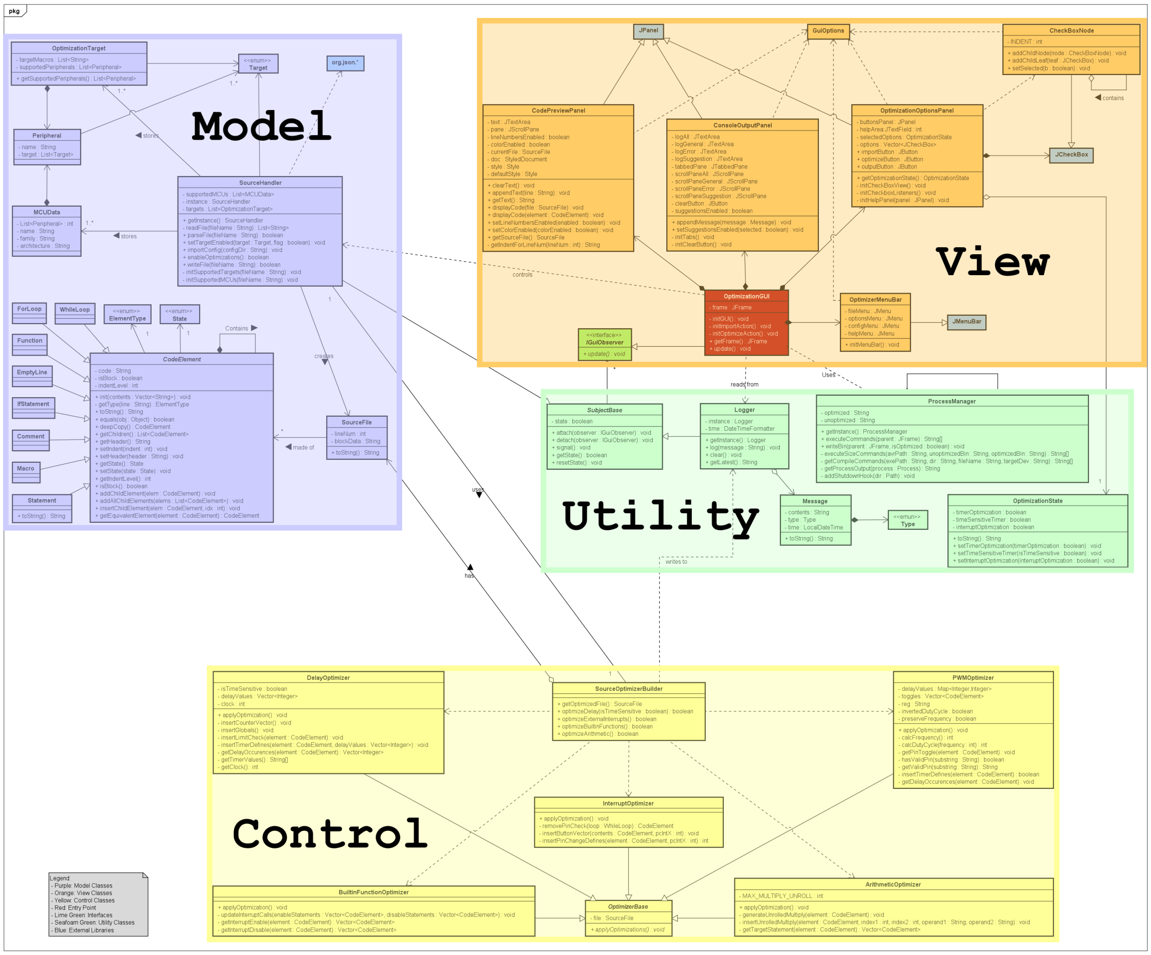


Figure 4.5: This class diagram details the high-level design of the Embedded C Source Optimizer, showing the organization of modules pertaining to the UI, optimization options, source code parsing, file and process I/O, etc.

A variety of Gang of Four (GoF) design patterns are used across the implementation of the Embedded C Source Optimizer [17]. These design patterns were implemented in an effort to both speed up development time and aid in an outside individual's ability to comprehend the software. The implemented patterns include:

- **Singleton** (found in the Logger, SourceHandler, and ProcessManager classes)
- **Strategy** (used by the different ”Optimizer” classes, inheriting from OptimizerBase)

- **Observer** (allowing the OptimizationGUI class to observe the Logger and SourceHandler class and update as necessary)
- **Builder** (implemented by the SourceOptimizerBuilder class which calls methods in the ”Optimizer” classes to build the optimized source file)
- **Composite** (used by the CodeElement class to recursively store a tree-like representation of C source code)
- **Decorator** (found in the CheckBoxNode class which is used to build the nested list of check boxes for the optimization options, as well as the UI classes which inherit and decorate various Java Swing components)

The ***Model*** module of the Embedded C Source Optimizer is responsible for handling all functionality pertaining to working with and manipulating C source code (Figure 4.6 shows the class structure of the model in greater detail). When a user imports a C source file, the SourceHandler class reads and parses it, storing the result in a SourceFile object, which can then be used by the ***Control*** module to apply various optimizations. Having a single entry point to this module (being the SourceHandler class) allows for the inner workings of its other classes to be abstracted away, enabling easier refactoring and improving its extensibility. The attributes which make up a SourceFile object are CodeElements. A CodeElement is defined here as being any structure found in embedded C code. At the most basic level, it may either be a block or a single line. For instance, functions, loops, and if statements would all be classified as blocks since they contain a nested body, whereas function calls, variable declarations/instantiations, macros, and single-line comments are not blocks since they do not have a body. This distinction allows a SourceFile object to be searched recursively for any type of element by simply iterating through every nested element and checking its type. From there, modifications can be made without disturbing the tree structure which it makes up. The CodeElement class provides all the necessary methods to add, remove, and modify any aspect of the element and (if it is a block) any of its nested elements. This module comes together to create a robust solution for storing and updating code as necessary, while providing encapsulated access to the data stored within.

The ***View*** module (shown in greater detail in Figure 4.7) is responsible for initializing the UI which is viewed and interacted with by the end-user. Decoration of classes implemented in the Swing library allows for a modular approach to user interface design, enabling individual components to be updated without affecting the rest of the UI. At the fundamental level, the base UI frame contains four panels: two for previewing code (one of which shows the original code and the other shows the resulting code after optimizations have been applied), one for selecting optimization options, and one for log output which details errors, warnings, suggestions, etc. Making up the two top panels, the CodePreviewPanel class contains functionality for displaying source code, as well as applying effects such as colored highlighting showing changes made after optimizations are applied and displaying line numbers for improved readability. The ConsoleOutputPanel stores log output from the Logger class (shown previously in Figure 4.4), and the OptimizationOptionsPanel provides options to select combinations of optimizations to be applied to the source code. These components are controlled by the OptimizationGUI class, being the entry point to the program. These classes come together to create the UI shown previously in Figure 4.1.

The ***Control*** module (shown in Figure 4.8) contains all the necessary functionality to apply the optimization options listed within the GUI to the imported C source code. The SourceOptimizerBuilder class is the entry point for this module, providing methods to selectively apply optimizations to a SourceFile depending on the state of the user-selected optimization options. The various ”Optimizer” classes (i.e. DelayOptimizer, InterruptOptimizer, etc.) contain functionality to search through and apply specific source-level optimizations (provided the original source file enables such optimizations to be applied in the

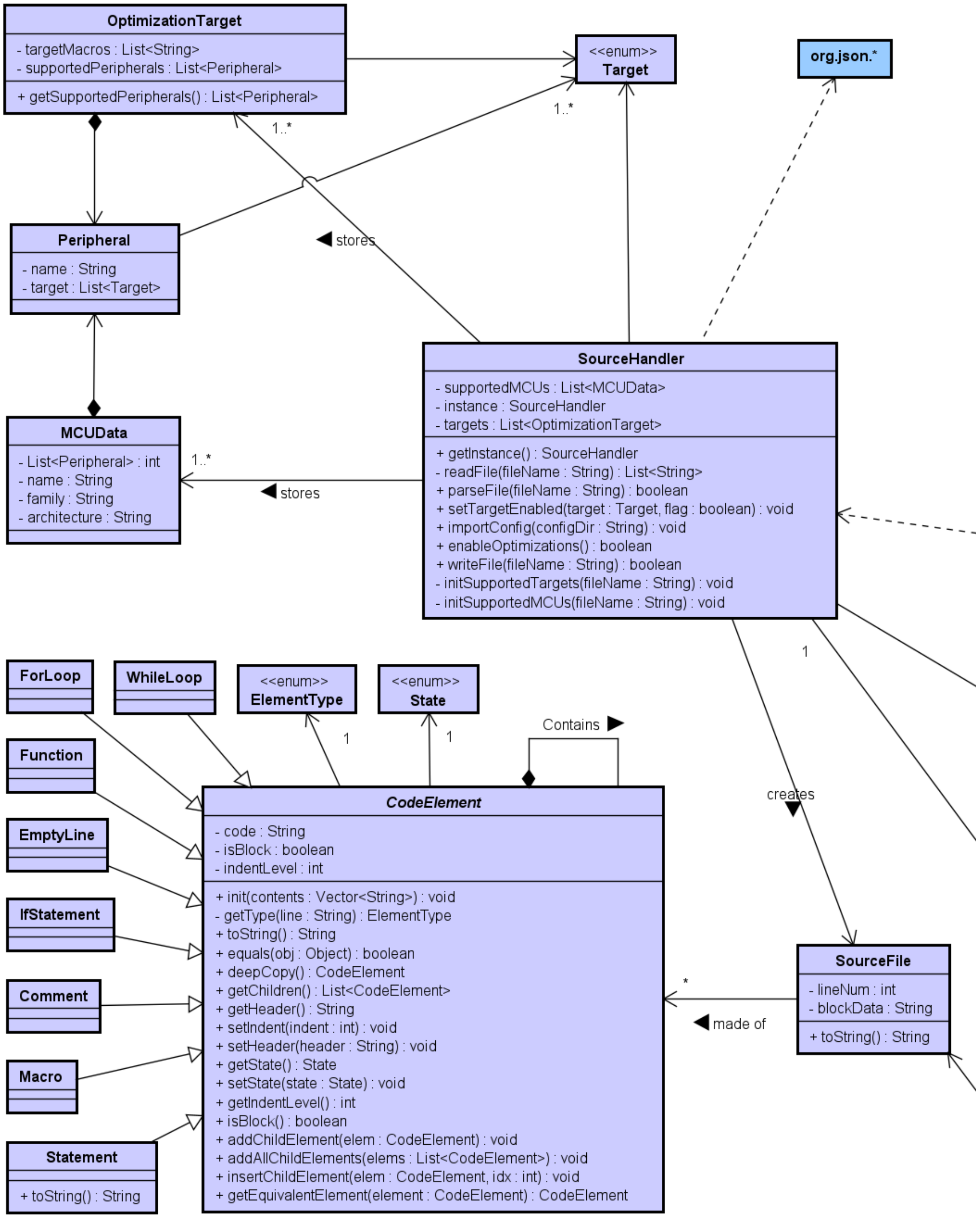

OptimizationTarget
- targetMacros : List<String>
- supportedPeripherals : List<Peripheral>
+ getSupportedPeripherals() : List<Peripheral>
<<enum>>
Target
org.json.*
1..*
1..*
Peripheral
- name : String
- target : List<Target>
stores
SourceHandler
- supportedMCUs : List<MCUData>
- instance : SourceHandler
- targets : List<OptimizationTarget>
+ getInstance() : SourceHandler
- readFile(fileName : String) : List<String>
+ parseFile(fileName : String) : boolean
+ setTargetEnabled(target : Target, flag : boolean) : void
+ importConfig(configDir : String) : void
+ enableOptimizations() : boolean
+ writeFile(fileName : String) : boolean
- initSupportedTargets(fileName : String) : void
- initSupportedMCUs(fileName : String) : void
MCUData
- List<Peripheral> : int
- name : String
- family : String
- architecture : String
1..*
stores
1
ForLoop
WhileLoop
<<enum>>
ElementType
<<enum>>
State
Contains
1
1
Function
creates
EmptyLine
CodeElement
- code : String
- isBlock : boolean
- indentLevel : int
+ init(contents : Vector<String>) : void
- getType(line : String) : ElementType
+ toString() : String
+ equals(obj : Object) : boolean
+ deepCopy() : CodeElement
+ getChildren() : List<CodeElement>
+ getHeader() : String
+ setIndent(indent : int) : void
+ setHeader(header : String) : void
+ getState() : State
+ setState(state : State) : void
+ getIndentLevel() : int
+ isBlock() : boolean
+ addChildElement(elem : CodeElement) : void
+ addAllChildElements(elems : List<CodeElement>) : void
+ insertChildElement(elem : CodeElement, idx : int) : void
+ getEquivalentElement(element : CodeElement) : CodeElement
IfStatement
Comment
Macro
Statement
+ toString() : String
*
made of
SourceFile
- lineNum : int
- blockData : String
+ toString() : String

Figure 4.6: The ***Model*** module contains classes pertaining to: importing, storing, managing, manipulating, and exporting source code.

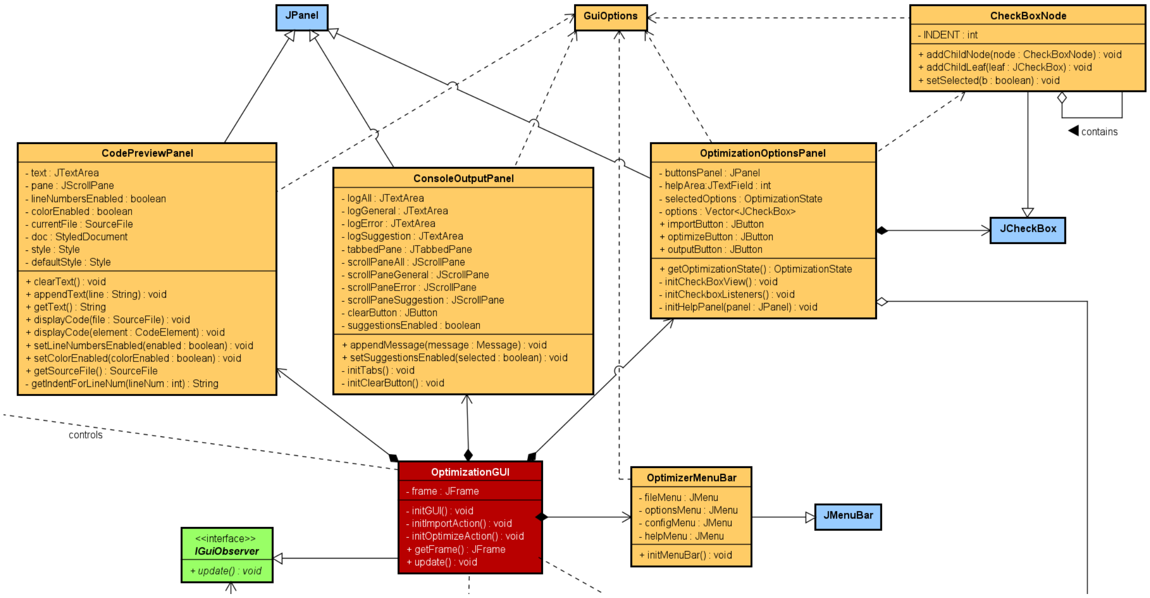

JPanel
GuiOptions
CheckBoxNode
- INDENT : int
+ addChildNode(node : CheckBoxNode) : void
+ addChildLeaf(leaf : JCheckBox) : void
+ setSelected(b : boolean) : void
contains
CodePreviewPanel
- text : JTextArea
- pane : JScrollPane
- lineNumbersEnabled : boolean
- colorEnabled : boolean
- currentFile : SourceFile
- doc : StyledDocument
- style : Style
- defaultStyle : Style
+ clearText() : void
+ appendText(line : String) : void
+ getText() : String
+ displayCode(file : SourceFile) : void
+ displayCode(element : CodeElement) : void
+ setLineNumbersEnabled(enabled : boolean) : void
+ setColorEnabled(colorEnabled : boolean) : void
+ getSourceFile() : SourceFile
- getIndentForLineNum(lineNum : int) : String
ConsoleOutputPanel
- logAll : JTextArea
- logGeneral : JTextArea
- logError : JTextArea
- logSuggestion : JTextArea
- tabbedPane : JTabbedPane
- scrollPaneAll : JScrollPane
- scrollPaneGeneral : JScrollPane
- scrollPaneError : JScrollPane
- scrollPaneSuggestion : JScrollPane
- clearButton : JButton
- suggestionsEnabled : boolean
+ appendMessage(message : Message) : void
+ setSuggestionsEnabled(selected : boolean) : void
- initTabs() : void
- initClearButton() : void
OptimizationOptionsPanel
- buttonsPanel : JPanel
- helpArea:JTextField : int
- selectedOptions : OptimizationState
- options : Vector<JCheckBox>
+ importButton : JButton
+ optimizeButton : JButton
+ outputButton : JButton
+ getOptimizationState() : OptimizationState
- initCheckBoxView() : void
- initCheckboxListeners() : void
- initHelpPanel(panel : JPanel) : void
JCheckBox
controls
OptimizationGUI
- frame : JFrame
- initGUI() : void
- initImportAction() : void
- initOptimizeAction() : void
+ getFrame() : JFrame
+ update() : void
<<interface>>
IGuiObserver
+ update() : void
OptimizerMenuBar
- fileMenu : JMenu
- optionsMenu : JMenu
- configMenu : JMenu
- helpMenu : JMenu
+ initMenuBar() : void
JMenuBar

Figure 4.7: The ***View*** module contains classes inheriting from various Swing components which are used by the OptimizationGUI class to create the UI which the end-user interacts with.

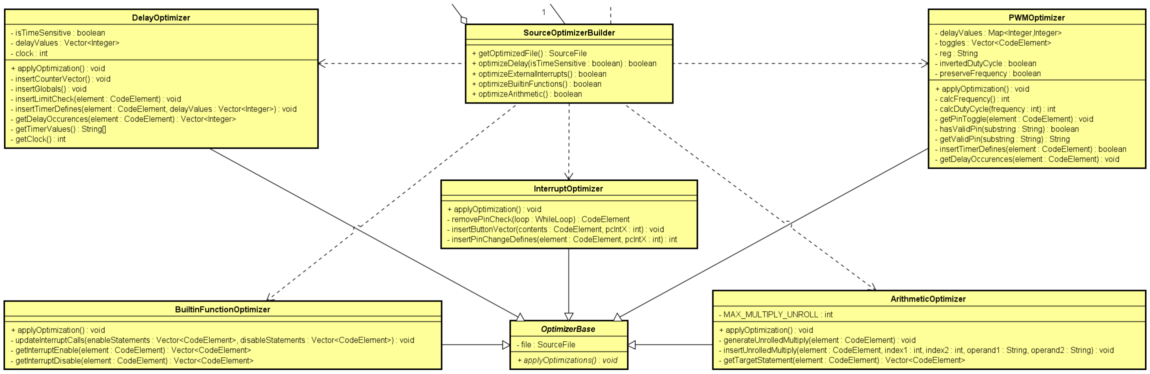

1
DelayOptimizer
- isTimeSensitive : boolean
- delayValues : Vector<Integer>
- clock : int
+ applyOptimization() : void
- insertCounterVector() : void
- insertGlobals() : void
- insertLimitCheck(element : CodeElement) : void
- insertTimerDefines(element : CodeElement, delayValues : Vector<Integer>) : void
- getDelayOccurences(element : CodeElement) : Vector<Integer>
- getTimerValues() : String[]
- getClock() : int
SourceOptimizerBuilder
+ getOptimizedFile() : SourceFile
+ optimizeDelay(isTimeSensitive : boolean) : boolean
+ optimizeExternalInterrupts() : boolean
+ optimizeBuiltinFunctions() : boolean
+ optimizeArithmetic() : boolean
PWMOptimizer
- delayValues : Map<Integer,Integer>
- toggles : Vector<CodeElement>
- reg : String
- invertedDutyCycle : boolean
- preserveFrequency : boolean
+ applyOptimization() : void
- calcFrequency() : int
- calcDutyCycle(frequency : int) : int
- getPinToggle(element : CodeElement) : void
- hasValidPin(substring : String) : boolean
- getValidPin(substring : String) : String
- insertTimerDefines(element : CodeElement) : boolean
- getDelayOccurences(element : CodeElement) : void
InterruptOptimizer
+ applyOptimization() : void
- removePinCheck(loop : WhileLoop) : CodeElement
- insertButtonVector(contents : CodeElement, pcIntX : int) : void
- insertPinChangeDefines(element : CodeElement, pcIntX : int) : int
BuiltinFunctionOptimizer
+ applyOptimization() : void
- updateInterruptCalls(enableStatements : Vector<CodeElement>, disableStatements : Vector<CodeElement>) : void
- getInterruptEnable(element : CodeElement) : Vector<CodeElement>
- getInterruptDisable(element : CodeElement) : Vector<CodeElement>
OptimizerBase
- file : SourceFile
+ applyOptimizations() : void
ArithmeticOptimizer
- MAX_MULTIPLY_UNROLL : int
+ applyOptimization() : void
- generateUnrolledMultiply(element : CodeElement) : void
- insertUnrolledMultiply(element : CodeElement, index1 : int, index2 : int, operand1 : String, operand2 : String) : void
- getTargetStatement(element : CodeElement) : Vector<CodeElement>

Figure 4.8: The ***Control*** module contains classes which analyze and apply optimizations to the source code stored within the ***Model*** module.

first place). All the ”Optimizer” classes inherit from a single class OptimizerBase, defining the basic attributes/methods each ”Optimizer” class must have. This enables polymorphic behavior by the SourceOptimizerBuilder, decoupling the implementation of each class from its definition. As a result, other ”Optimizer” classes can be added or existing ones changed with minimal modifications needed to the other classes within the module.

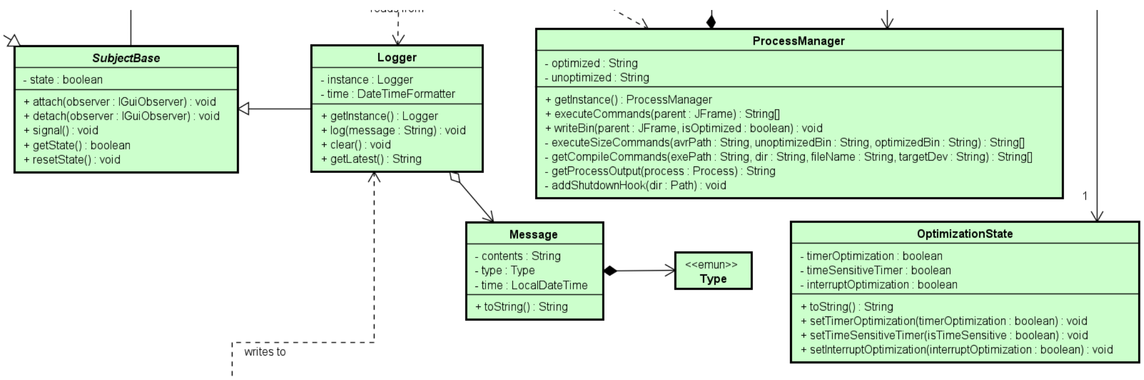

SubjectBase
- state : boolean
+ attach(observer : IGuiObserver) : void
+ detach(observer : IGuiObserver) : void
+ signal() : void
+ getState() : boolean
+ resetState() : void
Logger
- instance : Logger
- time : DateTimeFormatter
+ getInstance() : Logger
+ log(message : String) : void
+ clear() : void
+ getLatest() : String
ProcessManager
- optimized : String
- unoptimized : String
+ getInstance() : ProcessManager
+ executeCommands(parent : JFrame) : String[]
+ writeBin(parent : JFrame, isOptimized : boolean) : void
- executeSizeCommands(avrPath : String, unoptimizedBin : String, optimizedBin : String) : String[]
- getCompileCommands(exePath : String, dir : String, fileName : String, targetDev : String) : String[]
- getProcessOutput(process : Process) : String
- addShutdownHook(dir : Path) : void
1
Message
- contents : String
- type : Type
- time : LocalDateTime
+ toString() : String
<<emun>>
Type
OptimizationState
- timerOptimization : boolean
- timeSensitiveTimer : boolean
- interruptOptimization : boolean
+ toString() : String
+ setTimerOptimization(timerOptimization : boolean) : void
+ setTimeSensitiveTimer(isTimeSensitive : boolean) : void
+ setInterruptOptimization(interruptOptimization : boolean) : void
writes to

Figure 4.9: The utility module provides extraneous functionality to both the ***View*** and ***Control*** modules, pertaining to: executing commands and process management on the host system, keeping track of the state of selected optimization options, logging messages to be displayed to the end-user, and enabling observer functionality across other classes.

While not specified by the MVC pattern, the ***Utility*** module is included as a separate entry due to multiple other modules using the classes contained within. Furthermore, these classes (shown in detail in Figure 4.9) do not necessarily fit into any of the other modules, and as such are kept separate. The SubjectBase class is simply used to implement the observer pattern, acting as the ”observable” component. SubjectBase is used by classes within other modules to communicate and indicate when changes are made, allowing any observers of that class to update accordingly. The Logger class, which inherits from SubjectBase, contains functionality to store and retrieve detailed messages pertaining to operations performed by other areas of the system. These messages are then displayed to the end-user in an effort to clearly communicate success/failure and other extraneous changes made by the system. The Logger stores messages as Message objects, which have a type to indicate whether the message was an error, suggestion, or just a general statement. ProcessManager is used to execute commands and keep track of the state of running processes created by the utility. Finally, OptimizationState is used to keep track of the state of which optimization options have been selected. Overall, this module acts as a intermediate step for other modules to carry out tasks, while enabling the reuse of the provided functionality across many different classes/modules.

4.2 Optimization Methods & Classification

Embedded optimizations tend to focus on code size and memory usage, given the constraints imposed by the hardware. The size optimization provided by GCC generally does an excellent job in this regard [36]; however, there is still room for improvement in other key areas which this optimization option may not account for. Furthermore, there can be edge cases where this optimization option may not be able to reduce the size of the resulting firmware. To better illustrate the effects of each optimization option, Subsection 4.2.1 will introduce metrics which attempt to convey how the changes made by the Embedded C Source Optimizer impact the resulting code. Some of these metrics can easily be demonstrated as being better or worse through quantitative measurements, while others are more subjective and require a diverse range of perspectives to address with any level of confidence.

4.2.1 Relevant Metrics for Assessing Embedded Software

There are a variety of metrics and aspects which can be analyzed in order to determine the level of success of each optimization option. A set of metrics was narrowed down which are both widely used and accepted for use in embedded software [39, 18, 27, 45]. These metrics include:

I - Latency

II - Utilization & Execution Capacity

III - Extensibility

IV - Code Size

V - Cyclomatic Complexity

These will be described in detail as far as why they must be considered, what impact they have on embedded software, and how they are measured in order to compare different iterations of software. The results of addressing the effects of each of these aspects in regard to each optimization option provided by the Embedded C Source Optimizer will be discussed in Chapter 5.

I Latency

Latency in the case of software systems refers to the incurred delay caused by the time taken for a system to execute a series of instructions before generating some form of output or sending a response [43]. Latency directly affects the response time of a given task, meaning that reducing latency will improve the end-to-end time taken between sending an input to a system and receiving a corresponding output. Considering the real-time nature and generally lower levels of performance found in embedded systems, reducing software latency can be beneficial when attempting to create programs that are both efficient (in terms of the number of instructions executed) and responsive.

When measuring latency on any system, there are some special considerations which must be made for determining how much of the total response time is due to software execution rather than extraneous hardware/signaling delays. Take for instance a system which includes a MCU which interfaces with an external device over a serial connection. The external device can send a request to the MCU, to which the MCU will perform some set of tasks. This might include: interfacing with sensors, performing calculations, activating actuators, reading or writing to external storage, etc. The MCU then sends a response to the external device corresponding to the actions it performed. Measuring the delay between sending the request and receiving a response indicates the total response time of the system, of which software latency is only one factor. Figure 4.10 shows the potential aspects of a system other than software which contribute to the overall response time.

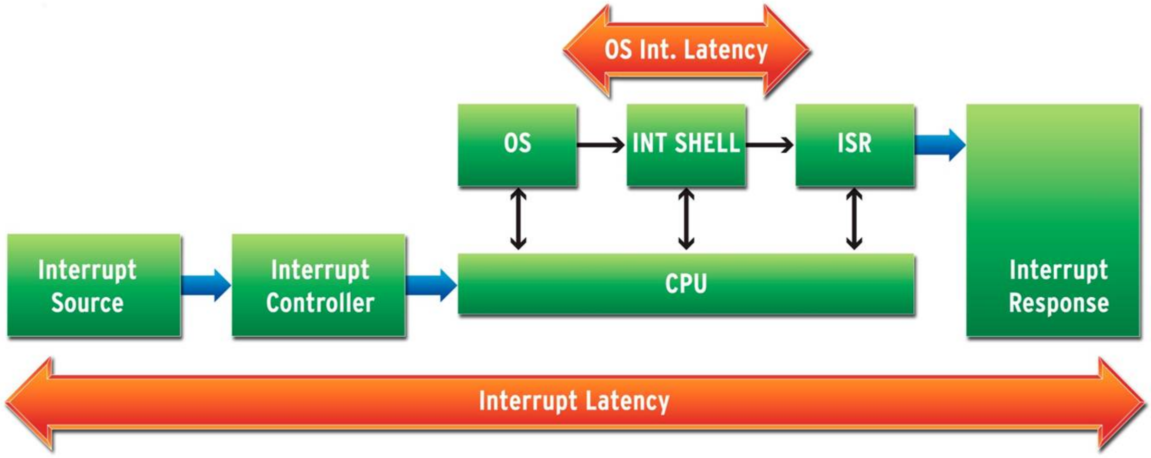


Figure 4.10: A flow diagram of a hypothetical pipeline for handling interrupts within a system, including the various stages which contribute to the over end-to-end response time. While many of the stages cannot be altered (such as the interrupt hardware), the primary concern within the scope of this metric concerns the latency introduced by the ”ISR” stage, being the software which runs when an interrupt is triggered. This example includes certain aspects which do not apply to most systems based on 8-bit MCUs, but the general idea of mapping end-to-end latency still holds [45].

In order to account for extraneous delay when measuring software latency, a control test may be performed to measure the end-to-end response time while attempting to minimize any incurred software latency as much as possible. The measured response time in such a case can be used as a baseline to then measure the impact of software latency relative to the total. While this method of measuring latency is not necessarily perfect, it is sufficient for comparing various implementations of programs without needing specialized external hardware, especially when accounting for as much of the extraneous delay (other than software latency) as possible.

II Utilization & Execution Capacity

Utilization within the scope of this thesis refers to the extent to which the MCU hardware is being used relative to its full potential. This both refers to what fixed-function hardware is utilized, as well as what proportion of time the MCU spends idle. [1] Both of these aspects of utilization are significant due to the restrictive nature of embedded hardware, and should be considered when attempting to maximize the potential of the software being developed.

Measuring the usage of hardware peripherals is a straightforward task, and gives a rough idea of what extent the provided fixed-function hardware is being utilized. On the other hand, measuring utilization as a proportion of time the hardware spends idle requires more setup, but yields potentially more granular data which is effectively hardware agnostic. In the same vein as attempting to measure latency on embedded systems, obtaining highly accurate utilization measurements for idle time likely would require external debug hardware, or some form of cycle-accurate hardware emulation. Within the scope of this thesis, the accuracy of measuring utilization must be compromised slightly in order to avoid the cost and other potential hidden factors of using external hardware or emulation. With simple tools and slight modifications to an existing program, it is possible to get an effective estimate of how the software is utilizing hardware in terms of the time the MCU spends idle versus its potential capacity to execute instructions.

Utilization can be represented as a percentage, which itself represents the ”headroom” leftover for additional functionality to be added. 0% utilization would indicate that the MCU is spending 100% of its time idle, and 100% utilization means the MCU is never idle at runtime and has no remaining capacity for additional functionality to be executed within an equal period of time. Idle in the traditional sense is defined as time the core spends executing No Operation (NOP) instructions; however within the scope of this thesis, idle time will refer to the remaining capacity for executing instructions regardless of what each instruction is. To better clarify execution capacity and utilization, there are some constraints which must be made - it is expected that a task takes a certain amount of time to execute,

[1] Idle as referred to in this thesis differs from the traditional definition, and will be discussed further in the following paragraphs.

and that task has a timing deadline it must meet. Furthermore, the scale of utilization will be skewed towards higher values as testing utilization in software inherently takes a certain number of instructions to measure; as such, utilization measurements must be taken relative to the smallest amount of instructions needed to track utilization.

$$\begin{aligned} Utilization &= 100\% - (\%\ time\ spent\ idle) \\ \%\ time\ spent\ idle &= \frac{(average\ period\ of\ an\ iteration\ with\ no\ load) * 100\%}{(average\ period\ of\ an\ iteration\ with\ some\ load)} \\ average\ period\ of\ an\ iteration &= \frac{(total\ execution\ time)}{(number\ of\ iterations)} \end{aligned}$$

Figure 4.11: Calculating utilization as a proportion of the average period taken to complete a task with some load versus the average period taken for a task which is idle (no load) [27].

In order to obtain a baseline measurement for an ”idle” state, the simplest way of measuring this utilization is to increment a counter every iteration of the main control loop of a program. Assuming the control loop does nothing else other than increment the counter, then the result of the counter over a fixed unit of time represents the lowest utilization which can possibly be measured in this manner. By adding functionality to the control loop, additional instructions will be executed every iteration of the loop, thus reducing the frequency which the counter is incremented. The relative difference between the final state of the counter over a known number of clock cycles represents the utilization of the measured program. As the number of clock cycles spent each iteration of the control loop increases, this relative difference will approach, but never actually reach 100%. See Figure 4.11 for exactly how utilization is calculated manner.

In order to accomplish this measurement process, a simple library has been developed (which is shown in Figure B.5) to track and output the measured utilization and execution capacity of a program. The premise of this library is to add as little overhead to the measured program as possible. This is done through utilizing a counter/timer instance which runs externally to the main MCU core, as well as a single 8 bit global variable which should be incremented at the end of every control loop iteration of the measured program. The header file shown in Figure B.4 provides preprocessor directives which affect the measurement period to account for the potentially variable execution period of the control loop from one program to another. A baseline test attempting to minimize work done in the main loop was developed which other programs can be compared against (which is provided in Figure B.6).

One potential issue with this method of measuring utilization is not accounting for what instructions are being executed, and as such the idle time (traditionally being any time the MCU is executing NOP instructions) cannot be fully represented by this metric. Rather, measurements derived in this manner relate more closely to the capacity for execution instead of the proportion of time the MCU spends idle. There is overlap between these metrics; however, it is not a one to one comparison. As stated previously, to accurately measure idle time on an embedded system would require additional tools which are unrealistic to obtain, setup, and use given the scope of this thesis. This discrepancy should be kept in mind when considering utilization measurements taken in this manner, but so long as a baseline for utilization is provided and all other factors are controlled, the relative measurements should relate closely to the measurement of actual idle time in most cases. [2]

III Extensibility

Extensibility refers to the extent which software can be expanded upon in terms of adding functionality in its current form [44]. Assuming embedded design patterns and general clean coding principles are used, a piece of software should take minimal refactoring

[2] A notable exception to this case is when software delay functions are used, as they are blocking calls which inflate the measurement for execution capacity (since the iteration period of the control loop is directly affected by these delays), but contribute towards what is traditionally referred to as idle time (executing NOP instructions) during execution. This added idle time will decrease utilization as it is traditionally defined, but the mentioned method of measuring utilization in software cannot account for this, thus inflating the utilization measurement. This distinction will become relevant in the evaluation provided in Chapter 5.

to modify, add, or remove functionality. This is the case for both traditional and embedded software, although embedded systems require additional forethought in their design due to their limited hardware.

Due to extensibility being an empirically derived metric, measuring how extensible a given program is can be challenging to represent. There do not appear to be any widely-accepted quantitative metrics for measuring extensibility; however, one potential method of demonstrating this metric is to track the difficulty of adding features to an existing program relative to its size. As such, extensibility can be defined by decomposing and measuring the relative size of each program segment, where these segments are the set of states the program may exist in during execution. The larger any given program segment is, the more refactoring would likely be required in order to add or modify its functionality. This is a result of program segments with a larger size taking longer to execute, thus reducing the execution frequency of any added functionality without implementing additional logic to split or otherwise change the existing program segments.

```
 1  #include <avr/io.h>
 2  #include <util/delay.h>
 3
 4  int main() {
 5    DDRB |= (1 << PB1); // set PB1 as an output (for LED)
 6    // control loop
 7    while (1) {
 8      PORTB ^= (1 << PB1); // toggle the state of PB1
 9      _delay_ms(500);
10    }
11    return 0;
12  }
```

Figure 4.12: A simple program which toggles the state of an output pin (shown on line 8) with a delay inserted (on line 9) for half a second (or 500 ms), with the intended use case being flashing an LED on and off with a period of 1 second.

Consider the simple embedded program provided in Figure 4.12 which toggles an LED by inverting the output of a I/O pin, with a 500 ms delay placed afterwards. Once the program is tested and working, now another task is added which checks for a button press on another I/O pin. Figure 4.13 shows how can be done by simply checking the state of the pin which the button is connected to, and using an if statement to evaluate to true in such a case. Ideally, the check for the button press should be done quickly to ensure the response is fast and a single press of the button is always detected. In this scenario, since the main loop is constantly changing the state of the LED and then calling a blocking delay function every iteration for 500 ms, simply adding the check for the button press to the main loop will mean that the MCU core can only perform the check every 500 ms at a minimum. This means there could be up to a half second delay between pressing the button and performing the task, and if the button is pressed too quickly (held down for less than 500 ms) then there is a chance it will not be detected at all. In order to check for a button press more frequently, additional logic would need to be added to split the single delay function into multiple which add up to 500 ms. An example of how this might be done is shown in Figure 4.14. The primary issue with this example is it adds additional complexity to the program and has the potential to throw off the timing of the LED blink, since the contents of the for loop will now be executed 50 times for every state change on the LED. This introduces extra overhead in the form of instructions which take additional time that the delay function cannot account for.

The code shown in Figures: 4.12, 4.13, and 4.14 are a simple demonstration of how extensibility impacts embedded code, but due to their simplicity may not fully demonstrate how extensibility scales with more complex programs. A program with many different tasks will likely take significantly more refactoring to add functionality, especially when embedded design patterns/rules are not implemented appropriately. By designing embedded software with the proper design patterns and using hardware peripherals as needed, much of this refactoring can be mitigated as software grows and becomes more complex. An example of how the program demonstrated in Figure 4.13 can be improved by the Embedded C Source Optimizer is shown in Figure 4.15. The ”Counter/Timer” and ”Interrupts” optimization options (which will be discussed in greater detail in Subsection 4.2.3) were selected and the resulting code contains additional logic for performing the same operations in hardware rather than software. Expanding on the optimized program becomes much more straightforward, as now adding a second task to the control loop is as simple as defining a second counter/limit instance and adding another conditional statement to the control loop which checks for that counter instance reaching its limit. Furthermore, any possible timing issues as a result of using software delay functions are resolved since the counter/timer peripheral operates independent of the MCU core.

```
 1  #include <avr/io.h>
 2  #include <util/delay.h>
 3
 4  int count = 0;
 5
 6  int main() {
 7    DDRB |= (1 << PB1); // set PB1 as an output (for LED)
 8    DDRB &= ~(1 << PB2); // set PB2 as an input (for button)
 9    while (1) { // control loop
10      PORTB ^= (1 << PB1); // toggle the state of PB1
11      _delay_ms(500);
12      // check for falling edge (button press)
13      if (PINB & (1 << PB2)) {
14        count++; // iterate count on button press
15      }
16    }
17    return 0;
18  }
```

Figure 4.13: An extension of the program shown in Figure 4.12 which attempts to add a second task which checks for a button press and increments a counter ”count” in the case that it was. This added task can be seen on lines 13-15, where the conditional checks if pin PB2 is high (in the case of a button press or another event) and increments the count variable in the case that condition evaluates to true.

IV Code Size

Code size is a relatively simple metric, representing the total amount of memory a program takes up after being compiled, assembled, and linked into a binary file containing

```
 1  #include <avr/io.h>
 2  #include <util/delay.h>
 3
 4  int count = 0;
 5
 6  int main() {
 7    DDRB |= (1 << PB1); // set PB1 as an output (for LED)
 8    DDRB &= ~(1 << PB2); // set PB2 as an input (for button)
 9    while (1) { // control loop
10      PORTB ^= (1 << PB1); // toggle the state of PB1
11      int i;
12      for (i = 0; i < 50; i++) {
13        // check for falling edge (button press)
14        if (PINB & (1 << PB2)) {
15          count++; // iterate count on button press
16        }
17        _delay_ms(10);
18      }
19    }
20    return 0;
21  }
```

Figure 4.14: A further modification of the program shown in Figure 4.13 which adds logic to poll the input pin connected to a button faster using a for loop and a shorter delay value. Lines 12 and 18 were added, corresponding to the for loop, and line 17 shows the delay function which was shortened to 10 ms from 500 ms.

machine code. In general, programs with a smaller final code size are favorable due to both the restrictive nature of embedded hardware, as well as indicating that there may be fewer instructions to execute overall, potentially providing a relative speedup to the program. GCC includes a compiler optimization which attempts to reduce the final code size, demonstrating the importance of improving this aspect of embedded software.[3]

As shown in Figure 4.3, there are different sections of memory which are considered when measuring code size, as not all memory is treated equally. The three different sections of memory which are considered in this case are: ”text”, ”data”, and ”bss”. ”Text” corresponds to the machine-level instructions which are generated after compiling a program,

[3] The performance of the program using the GCC size optimization option (-Os) will depend on the program itself; however in general should provide a speed up for a general case as it ”enables all -O2 optimizations except those that often increase code size,” where ”-O2” refers to the moderate speed optimization option [36].

```
 1  #define F_CPU 1000000
 2  #include <avr/io.h>
 3  #include <util/delay.h>
 4
 5  int count = 0;
 6
 7  volatile unsigned int count[1];  // Timer instances for each of the blocks of code
 8  volatile unsigned int limit[1];  // Timer limit instances to compare against 'count'
 9  int main() {
10      OCR0A  = 0x7c;  // Sets an output compare value for a 1ms tick
11      TCCR0A = 0x02;  // Clear compare register on compare match
12      TCCR0B = 0x82;  // Force output compare A, and sets a 1ms tick
13      TIMSK0 = 0x03;  // Enables timer interrupts for compare match A
14      count[0] = 500;
15      limit[0] = 500;
16      __builtin_avr_sei();  // Enable global interrupts
17      PCICR  = 0x01; // Sets the appropriate pin bank for the pin being used
18      PCMSK0 = 0x4;  // Set the pin being used for external interrupt
19      DDRB |= (1 << PB1);   // set PB1 as an output (for LED)
20      DDRB &= ~(1 << PB2);   // set PB2 as an input (for button)
21      // control loop
22      while (1) {
23          if (count[0] == limit[0]) {
24              unsigned char state = SREG;  // Retrieve SREG state
25              __builtin_avr_cli();  // Disable global interrupts
26              PORTB ^= (1 << PB1);   // toggle the state of PB1
27              count[0] = 0;  // Reset counter instance
28              SREG = state;  // Restore previous state of SREG
29          }
30          // Removed: _delay_ms(500);
31          // check for falling edge (button press)
32      }
33      return 0;
34  }
35  // Declare external interrupt vector as being called by the internal interrupt
36  void __vector_3(void) __attribute__ ((signal, used, externally_visible));
37  void __vector_3(void) {  // Interrupt vector for external pin change
38      if (PINB & (1 << PB2)) {
39          // iterate count on button press
40          count++;
41      }
42  }
43  // Declare timer interrupt vector as being called by the internal interrupt
44  void __vector_14(void) __attribute__ ((signal, used, externally_visible));
45  // Timer interrupt vector which will be called by the timer hardware on compare match
46  void __vector_14(void) {
47      if (count[0] < limit[0]) {
48          count[0]++;  // add 'tick' to count, indicating a 1ms increase
49      }
50  }
```

Figure 4.15: An optimized version of the program shown in Figure 4.13 with the ”Counter/Timer” and ”Interrupts” optimization options in the Embedded C Source Optimizer selected. The existing logic was updated to utilize one of the 8-bit counters on the ATmega168 to set the delay of the LED blink, as well as enable hardware interrupts for triggering an ISR which increments the counter on a button press. Lines 7 and 8 keep track of the delay value and check if the delay has reached its limit. 10-13 correspond to configuring the timer by setting the appropriate registers which enable a 1ms tick. 17 and 18 configure hardware interrupts for a state change on pin PB2. 23 mimics the functionality provided by a delay function, only evaluating to true when the counter instance has reached its limit. 36-42 correspond to the ISR which handles the button press, and 44-50 are the ISR used by the timer to increment the count instance every 1 ms.

and are stored in the non-volatile flash memory included on most MCUs. This data can be persistent as the instructions generated by the compiler will never need to change during execution. The ”data” section shows how much memory is used for initialized data in a given program. Initialized in this case simply means the initial value associated with a memory address or variable is known at compile-time, and can therefore be stored in flash memory alongside the program code. If the initialized variable is not defined as constant (meaning the value of that variable can be altered at runtime) then an equal amount of SRAM must be reserved to account for this. This overlaps with ”BSS”, which refers to any uninitialized data in a program. Uninitialized data is any variable or memory address which is defined at runtime versus compile-time, and therefore must be stored in SRAM as well. All three of these sections are added together to get the total program size (which is listed under the ”dec” and ”hex” labels in the Berkley format [1]).

V Cyclomatic Complexity

Credited with its development to Thomas J. McCabe, cyclomatic complexity is a metric used to indicate the complexity of a given piece of software, where complexity refers to the number of different paths the program may take during execution. The original intent for such a metric came as a result of McCabe attempting to find a better alternative to Source Lines of Code (SLOC) as a metric, as ”McCabe could see no obvious relationship between length and module complexity” [38]. Despite McCabe’s original intentions, cyclomatic complexity as a software metric has been scrutinized over time, with critics arguing that the metric itself is flawed at the fundamental level. Some of the issues which have been cited with the metric include: not recognizing data or functional complexity, being overly sensitive to the number of subroutines included in the measured program, not considering ”else” blocks within conditional statements, and the lack of any means to measure the complexity of a linear (non-branching) series of statements [38].

While the validity and use cases for this metric are mixed, it can still be used as a thought experiment and may give some insight into the complexity of simple embedded programs. Furthermore, it may be possible to draw correlations between cyclomatic complexity and other metrics to better determine how changes made to embedded software improve or detract from the program itself. An extension of measuring cyclomatic complexity would be generating a Control-Flow Graph (CFG) which can be analyzed more granularly in an effort to better understand how changes made to software impact the resulting execution, and therefore get an idea of how the functionality may differ.

To measure the cyclomatic complexity of embedded C code, the C and C++ Code Counter (CCCC) utility may be used. This tool performs static analysis on C code, and provides measurements for a variety of metrics, including cyclomatic complexity [31]. Larger cyclomatic complexity values indicate higher levels of complexity in terms of the number of paths which can be taken during execution. McCabe claims that higher measurements indicate less maintainable modules which are more difficult to test [38].

4.2.2 Optimization Categories

The optimization options which will be discussed in Subsection 4.2.3 can be broken down into two general categories: direct replacement of code, and implicit generation of code which configures and utilizes on-board peripherals. The former option is simpler to configure and generally more reliable as these cases are very specific, while the latter has the potential to introduce issues since there is some level of interpretation which must be done in order to preserve the intended functionality of the software. That being said, there is more potential to be unlocked through utilizing peripherals than the otherwise simple replacement steps.

I Direct Replacement

Direct replacement in this situation is defined as taking an existing line or section of code, and modifying it in a way which better utilizes the hardware for the given task without changing the actual functionality of the code. These optimizations are applied to very specific cases such that unwanted changes to the program's behavior are minimized. The ”Built-In Function Substitution” and ”Arithmetic Substitution” optimization options (which will be detailed in Subsection 4.2.3) included in the Embedded C Source Optimizer are considered to be direct replacement.

II Implicit Utilization of Hardware Peripherals

Hardware peripherals, such as those mentioned in Chapter 2, must be configured manually at the source code level in order to use them for a given task. This is one of the fundamental aspects of embedded software which the Embedded C Source Optimizer aims to improve. Since these specialized hardware blocks are associated with embedded systems, a software engineer without much experience in embedded may go through a substantial amount of trial and error to not only get them configured properly, but also research and determine what peripheral(s) they need to use in the first place. The optimization options: ”Counter/Timer,” ”Interrupts,” and ”Pulse-Width Modulation” (which will be introduced in Subsection 4.2.3) aim to take a naive software implementation and determine if said implementation can benefit from using these peripherals. The utility will then add all the necessary code to configure the peripheral(s) and rework the existing implementation to support the added functionality.

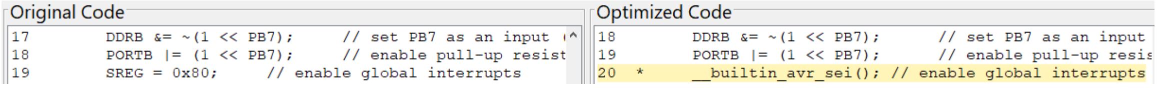
4.2.3 Supported Optimization Options

All of the optimizations supported by the Embedded C Source Optimizer are listed in this subsection. Implementation specific details will be provided for each as far as: what they target, how they apply changes to an existing program, and what potential benefits they provide to the end-user. While not mutually exclusive, some of the optimizations generally are intended to be used independent of one another (both due to the overlapping of hardware utilization and the code they target for refactoring). That being said, various use cases will often have one optimization or the other applied more effectively as a result. This leaves room for the end-user to experiment with each option on their code to determine which optimization more effectively improves their existing software.

I Built-in Function Substitution

There is a set of functions recognized by the AVR-GCC compiler which allow specific assembly-level instructions to be specified at the source code level. It is generally advantageous (or in certain cases required to achieve the desired functionality) to utilize these functions as relying on the compiler to perform these operations will generally result in a different set of instructions. Certain cases require specific sets of instructions to be executed within a strict time window, which is why relying on the compiler to determine the appropriate instructions may lead to unexpected behavior. Other cases are simply a matter of attempting to reduce the number of instructions in the firmware after compilation, reducing code size and possibly improving the performance of the program.

The Embedded C Source Optimizer identifies and replaces two specific cases where built-in functions can perform the exact same task. These include setting and clearing global interrupts, where the corresponding built-in functions are ”`__builtin_avr_sei()`” and ”`__builtin_avr_cli()`” respectively [7]. A naive approach to setting or clearing the flag which corresponds to whether interrupts are enabled or disabled often could be to do so directly by performing a bitwise operation on the register where the flag is stored. This register (SREG) is the status register in the ATmegaXX8 line of MCUs, where various flags are stored by the MCU core concerning the result of arithmetic operations, as well as the global interrupt enable flag [6]. This flag is generally the only bit that would be changed in software as the rest are dependent on the state of the MCU. As such, in the case that the global interrupt enable bit is being set by directly accessing SREG, the built-in functions can be used to ensure that the state of the remaining flags in the register are not changed, while simultaneously reducing the number of instructions needed to do so. Examples of how the Embedded C Source Optimizer handles these cases when the ”Built-In Function Substitution” optimization option is selected are shown in Figures 4.16 and 4.17, with the naive approach to setting and clearing global interrupts on the left, and the optimized solution using built-in functions on the right.



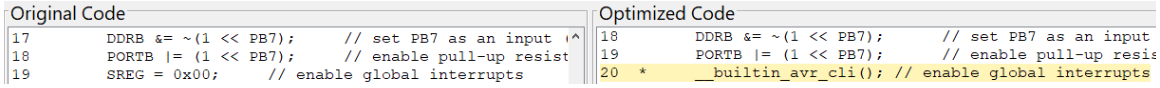
Original Code

```
        DDRB &= ~(1 << PB7);      // set PB7 as an input (
        PORTB |= (1 << PB7);      // enable pull-up resist
        SREG = 0x80;      // enable global interrupts
```

Optimized Code

```
        DDRB &= ~(1 << PB7);      // set PB7 as an input
        PORTB |= (1 << PB7);      // enable pull-up resis
   *    __builtin_avr_sei(); // enable global interrupts
```

Figure 4.16: An example of the ”Built-In Substitution” option modifying an existing line of code which directly modifies the Status Register (SREG) to use the built-in function which performs the same function (enabling global interrupts)



Original Code

```
        DDRB &= ~(1 << PB7);      // set PB7 as an input (
        PORTB |= (1 << PB7);      // enable pull-up resist
        SREG = 0x00;      // enable global interrupts
```

Optimized Code

```
        DDRB &= ~(1 << PB7);      // set PB7 as an input
        PORTB |= (1 << PB7);      // enable pull-up resis
   *    __builtin_avr_cli(); // enable global interrupts
```

Figure 4.17: An example of the ”Built-In Substitution” option modifying an existing line of code which directly modifies the Status Register (SREG) to use the built-in function which performs the same function (disabling global interrupts)

The savings in code size and performance uplift (through reducing the number of executed instructions) may vary between different programs. In the case of the replacement shown in Figures 4.16 and 4.17, the original code containing a direct assignment to SREG will generate four instructions when compiled; however, the optimized code which replaces the assignments with their corresponding built-in functions generate only one instruction.

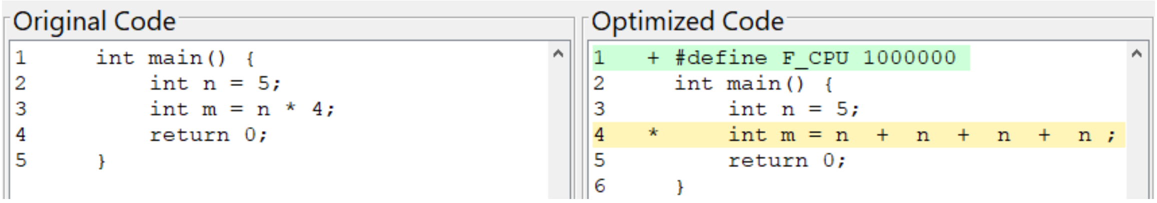
Since some programs may not rely on interrupts at all, this optimization would not yield any benefit in such a case. Other programs which do rely on interrupts may see an uplift in the two areas mentioned previously, depending on how often they are being executed.

II Arithmetic Substitution

Due to the lack of hardware support on some embedded hardware for various arithmetic operations, attempting to perform even simple combinations of arithmetic functions can end up generating less-than-ideal machine code. The primary issue with addressing this issue becomes the variations in hardware support for specific instructions. As such, attempting to target a broad range of MCUs yields varying levels of improvement depending on the capability of the hardware being used.

One potential optimization which can improve the number of instructions generated by the compiler would be to target slightly more complex arithmetic operations which the MCU may not support at the hardware level. For instance, some MCUs lack hardware multiplier units [13], and as such would need to perform any multiplication defined in code as repeated addition [20]. Depending on how the compiler handles these situations, it may be worthwhile to better define such an instance in terms of addition in the first place. This ”multiply unrolling” has the potential to save on generated instructions depending on how the compiler is set up to handle multiplication [15]. An example use case of this operation is provided in Figure 4.18. In this instance, running the ”Compile/Analyze” option within the Embedded C Source Optimizer on both the unoptimized and optimized program shows no change to the final code size. This is likely due to the target MCU containing a hardware multiplier.

Another potential (and more universally applicable) application of arithmetic substitution comes in the form of multiplication and division by powers of two. Embedded hardware is less likely to support division at the hardware level, and therefore is applicable to a wider

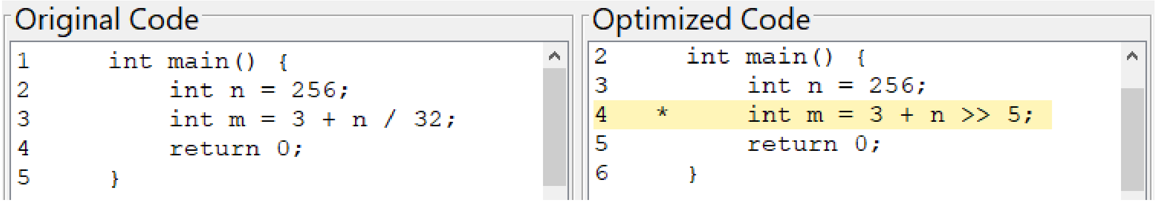


```
int main() {
    int n = 5;
    int m = n * 4;
    return 0;
}
```

```
+ #define F_CPU 1000000
  int main() {
      int n = 5;
*     int m = n  +  n  +  n  +  n ;
      return 0;
  }
```

Figure 4.18: An example of the ”Arithmetic Substitution” optimization which ”unrolls” a multiplication operation to use addition at the source code level. This specific example shows no difference in final code size after the optimization is applied, which is likely due to the inclusion of a hardware multiplier on the ATmega168.

range of hardware. By replacing multiplication and division operations when the multiplicand or divisor is a power of 2 with shift operations, fewer instructions may be generated as a result. This works as multiplying/dividing by a power of 2 is the same as shifting the binary representation of that number to the left or right respectively. Such operations are significantly less expensive in terms of clock cycles used, and by explicitly using them at the source code level may yield fewer instructions overall [16]. An example of such a case is shown in Figure 4.19, where the optimized code generates a binary file that is six bytes less than the original (an overall reduction of approximately three percent in terms of the final code size). This difference in size is a direct result of a fewer number of instructions generated by the compiler due to the lack of a hardware divider.



```
int main() {
    int n = 256;
    int m = 3 + n / 32;
    return 0;
}
```

```
  int main() {
      int n = 256;
*     int m = 3 + n >> 5;
      return 0;
  }
```

Figure 4.19: An example of the ”Arithmetic Substitution” optimization replacing a division operation where the divisor is a power of two with a right shift operation which yields the same result. Since the divisor is 32 in this case, shifting the binary representation of the value stored in variable ’n’ to the right 5 times will yield the same value (since 2^5=32). The final code size was reduced from 206 bytes to 200 bytes (an approximate 3 percent reduction) when targeting an ATmega168.

III Counter/Timer

The counter/timer peripheral is one of the most common groups of fixed-function hardware included on MCUs and has the potential to provide a great amount of additional functionality to embedded software. While the ”Built-in Function” and ”Arithmetic Substitution” optimizations match and replace very specific cases, the ”Counter/Timer” optimization (and those that follow) implement implicit utilization of hardware peripherals and must make certain inferences about the existing software. In the case of the ”Counter/Timer” optimization, it attempts to find an instance where blocks of code are executed with calls to software delay functions placed in-between. An example case might be as simple as toggling the state of an output on and off using these delay functions to slow down how often the output is toggled. Another instance could be interfacing with a digital sensor and waiting a specific amount of time before reading the output (assuming the sensor has some delay between the request, measurement, and response). There are countless other combinations of cases where delays can be used, which means the ”Counter/Timer” optimization is potentially applicable to many different use cases.

The simplest method of executing a delay within an embedded program is by using a software delay function. These functions take a delay value (generally in milliseconds or microseconds) and when called will wait for the specified amount of time before exiting. The clock frequency of the MCU and the specified delay value can be used to calculate the number of clock cycles the MCU must wait to achieve the desired amount of delay [19]. Since the delay function is only used to insert a set amount of delay within a program, the actual instructions which the MCU will execute are NOP instructions. These instructions explicitly perform no work, and generally take one clock cycle to execute [11].

For simple programs, the use of software delay functions may be fine, but there are two primary issues which may become apparent when expanding upon or building more complex embedded software. One issue is that software delays cannot account for processing time taken by other instructions used within the program. For instance, if a 50 ms delay is inserted into the main control loop (being a loop which iterates indefinitely) containing other code, the only thing that can be known for sure is that 50 ms will be added to the total execution time of a single iteration of the loop. This does not account for other instructions, so a loop containing many other complex operations/subroutine calls could add up to multiple milliseconds. For the sake of example, say the sum of the incurred delay due to these instructions is 5 ms. With a 50 ms delay included in the control loop, the total execution time of a single iteration will be 50 + 5, or 55 ms (a 10% increase). If the program is at all time sensitive or needs to synchronize with another piece of hardware at specific intervals, then this timing error will rapidly propagate and likely cause extraneous issues (or in the worst case make the program nonfunctional). Not all programs which use software delay functions will suffer from this as it depends on the length of the delay and the number of instructions executed outside of the delay, but regardless should be considered.

A second issue with software delay functions is how they impact the extensibility of the code, and how effectively the embedded hardware can be utilized to carry out useful instructions. Since a software delay function will sit executing NOP instructions for the length of the delay, the core of the MCU cannot be used for anything else until the delay function exits. This means the core will essentially be sitting idle for that entire period of time, which may impact any timing requirements of the system. Attempting to add functionality to the program after the fact will only continue to add to the total incurred delay from the software delay functions, and potentially throw off the timing of other code within the control loop.

A better alternative to software delay functions would be to utilize the fixed-function hardware included in the MCU, rather than relying on software to achieve the intended delay. By configuring an instance the counter/timer peripheral to trigger interrupts at specific periods, both of the previously mentioned issues can be addressed. Since the counter is an independent piece of hardware, it does not need any intervention by the core of the MCU to operate (other than the initial configuration and handling of ISRs). Due to this lack of dependence, the timing will not be affected by any instructions carried out on the core, resulting in a more accurate delay. Furthermore, since the core is not sitting idle during the delay, it becomes much more straightforward to expand an existing piece of software to include additional functionality without compromising how and when each timing-specific task is executed.

In order to convert existing embedded code which contains software delays to a functionally equivalent program which utilizes the counter/timer peripheral, the Embedded C Source Optimizer searches through the control loop of the provided source code, finds any calls to software delay functions, and remaps the other code into conditional statements which execute after the specified delay value is met (for a more detailed breakdown of this optimization option's behavior, see Figure 4.20). An example use case for the ”Counter/Timer” optimization is provided in Figure 4.21. Depending on if the software delay function being used was in terms of milliseconds or microseconds, the timer will be configured to either a 1 ms or 1 us ”tick” (meaning the timer will trigger an interrupt every 1 ms or 1 us). In the case of the program shown in Figure 4.21, the counter is setup for a 1 ms tick since the original software delay function was provided in terms of ms. The ISR (”`__vector_14(void)`”) which is executed will increment variables corresponding to the amount of time passed, and check whether they have reached the delay limit (which can be seen in the main control loop as ”`if (count[0] == limit[0])`”). Due to the limited number of counter instances available, storing and incrementing the delay and limit values in software allows a single timer instance to handle any number of delays in the original software.

One limitation imposed by the ”Counter/Timer” optimization is not preserving the original timing requirements imposed by the original program. This is a result of the ”Counter/-Timer” optimization not accounting for any additional code contained within the program after any included calls to existing software delay functions relying on the delay included by any previous calls to these software delay functions. In certain cases, the ”Counter/Timer” optimization may actually improve the functionality of the existing software (cases of this will be provided in Section 5.2), but in others can impact the functionality of the resulting program. To combat this, a secondary option of the ”Counter/Timer” optimization is included - the ”Time Sensitive Order of Execution” (or ”TSOoE”) option. This secondary option which can be selected eliminates this issue by assuming any tasks included after the final delay function call within the control loop of a program contains the same timing requirements imposed by the previous tasks. As such, the final task is provided a separate counter/limit instance which behaves in the same manner as any previous counter/limit instances added to a given program.

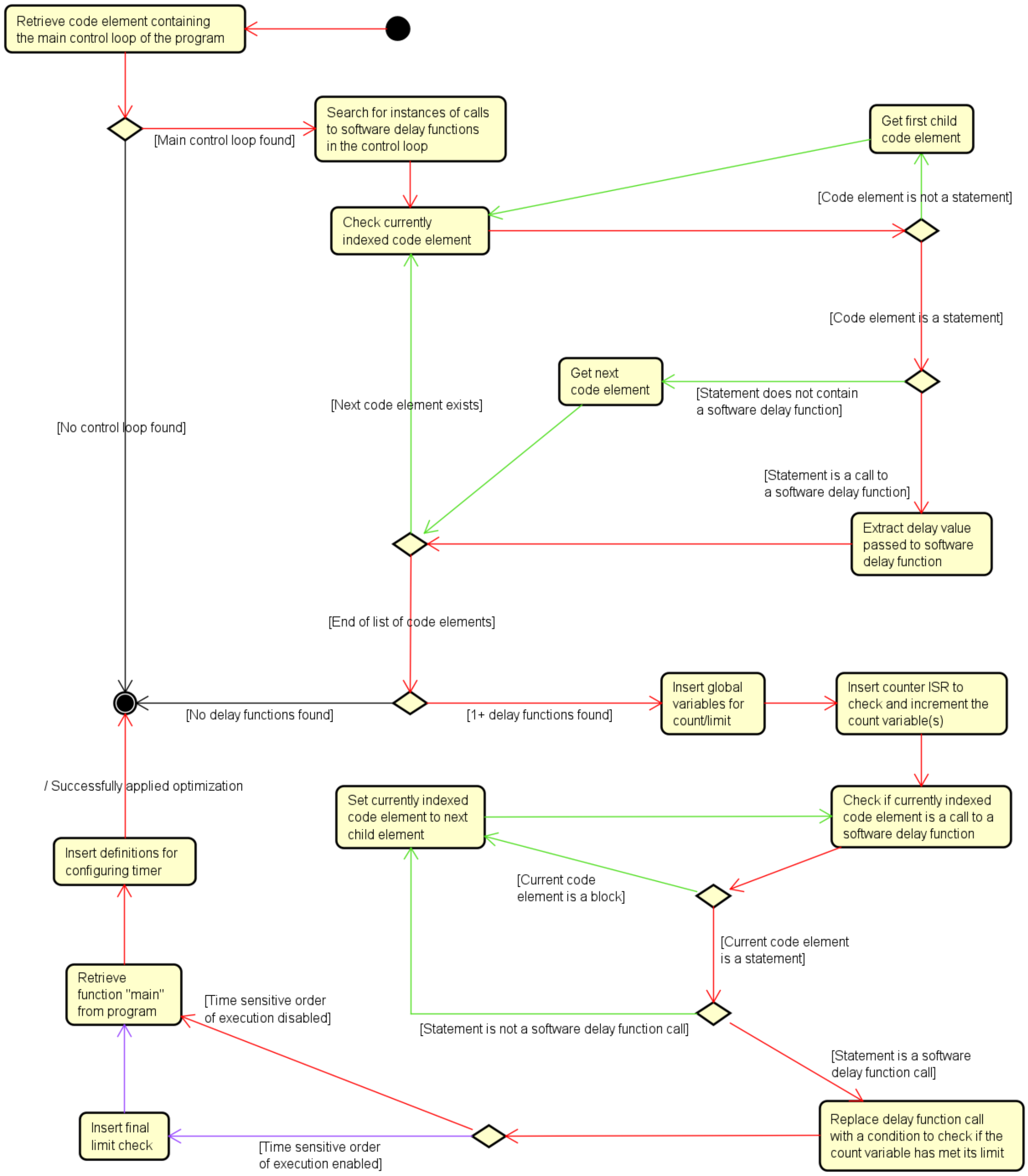
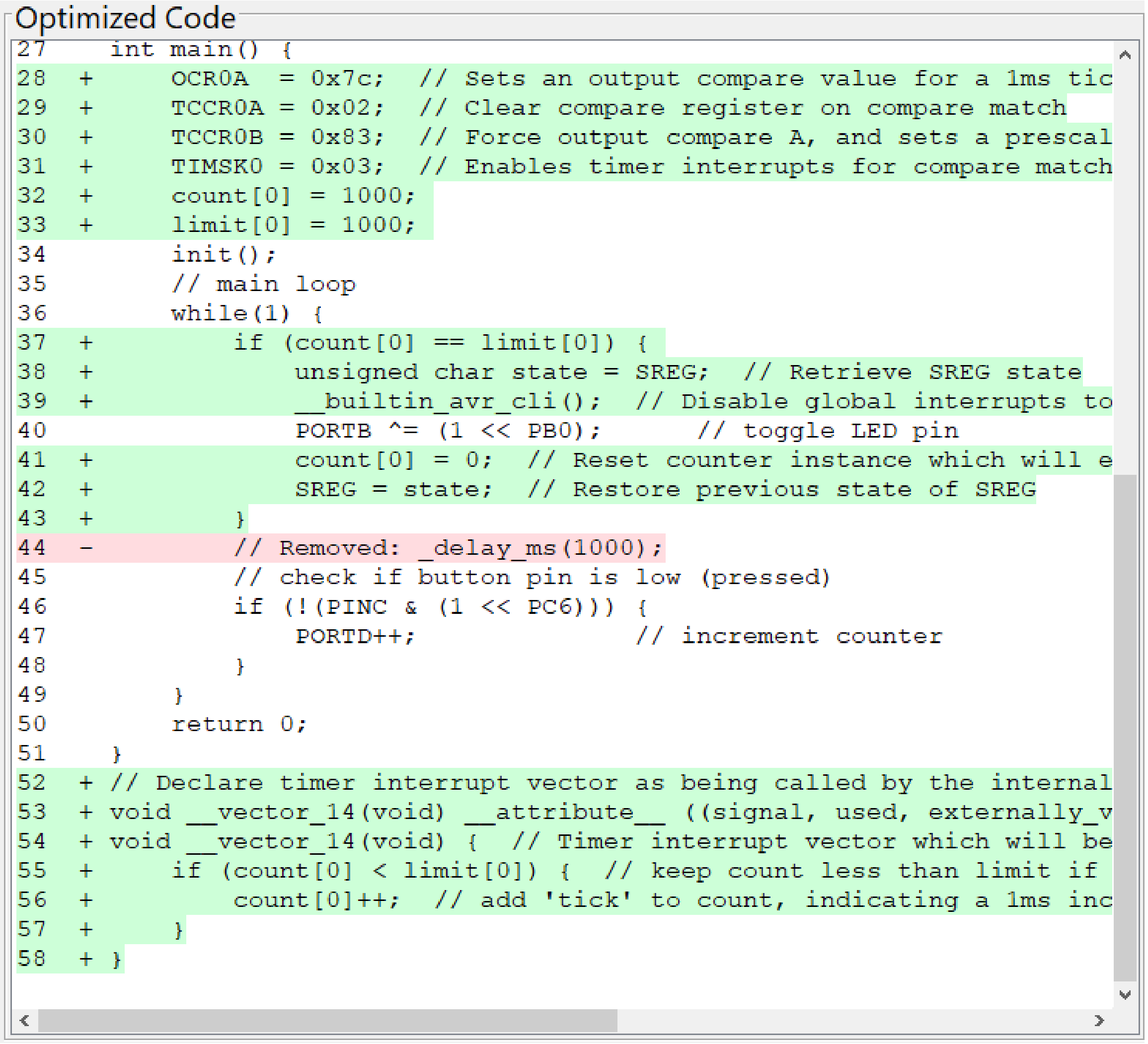

Retrieve code element containing the main control loop of the program
[Main control loop found]
Search for instances of calls to software delay functions in the control loop
Get first child code element
[Code element is not a statement]
Check currently indexed code element
[Code element is a statement]
Get next code element
[Statement does not contain a software delay function]
[Next code element exists]
[No control loop found]
[Statement is a call to a software delay function]
Extract delay value passed to software delay function
[End of list of code elements]
[No delay functions found]
[1+ delay functions found]
Insert global variables for count/limit
Insert counter ISR to check and increment the count variable(s)
/ Successfully applied optimization
Set currently indexed code element to next child element
Check if currently indexed code element is a call to a software delay function
Insert definitions for configuring timer
[Current code element is a block]
[Current code element is a statement]
Retrieve function "main" from program
[Time sensitive order of execution disabled]
[Statement is not a software delay function call]
[Statement is a software delay function call]
Insert final limit check
[Time sensitive order of execution enabled]
Replace delay function call with a condition to check if the count variable has met its limit

Figure 4.20: Demonstrates the behavior of the ”Counter/Timer” optimization and how it is applied to source code. The main path (indicating the optimization was successfully applied) is marked in red, with green showing loops, purple being optional paths, and black indicating the optimization could not be applied.



Optimized Code

```
27     int main() {
28  +      OCR0A  = 0x7c;  // Sets an output compare value for a 1ms tic
29  +      TCCR0A = 0x02;  // Clear compare register on compare match
30  +      TCCR0B = 0x83;  // Force output compare A, and sets a prescal
31  +      TIMSK0 = 0x03;  // Enables timer interrupts for compare match
32  +      count[0] = 1000;
33  +      limit[0] = 1000;
34         init();
35         // main loop
36         while(1) {
37  +          if (count[0] == limit[0]) {
38  +              unsigned char state = SREG;  // Retrieve SREG state
39  +              __builtin_avr_cli();  // Disable global interrupts to
40                 PORTB ^= (1 << PB0);      // toggle LED pin
41  +              count[0] = 0;  // Reset counter instance which will e
42  +              SREG = state;  // Restore previous state of SREG
43  +          }
44  -          // Removed: _delay_ms(1000);
45             // check if button pin is low (pressed)
46             if (!(PINC & (1 << PC6))) {
47                 PORTD++;              // increment counter
48             }
49         }
50         return 0;
51     }
52  + // Declare timer interrupt vector as being called by the internal
53  + void __vector_14(void) __attribute__ ((signal, used, externally_v
54  + void __vector_14(void) {  // Timer interrupt vector which will be
55  +      if (count[0] < limit[0]) {  // keep count less than limit if
56  +          count[0]++;  // add 'tick' to count, indicating a 1ms inc
57  +      }
58  + }
```

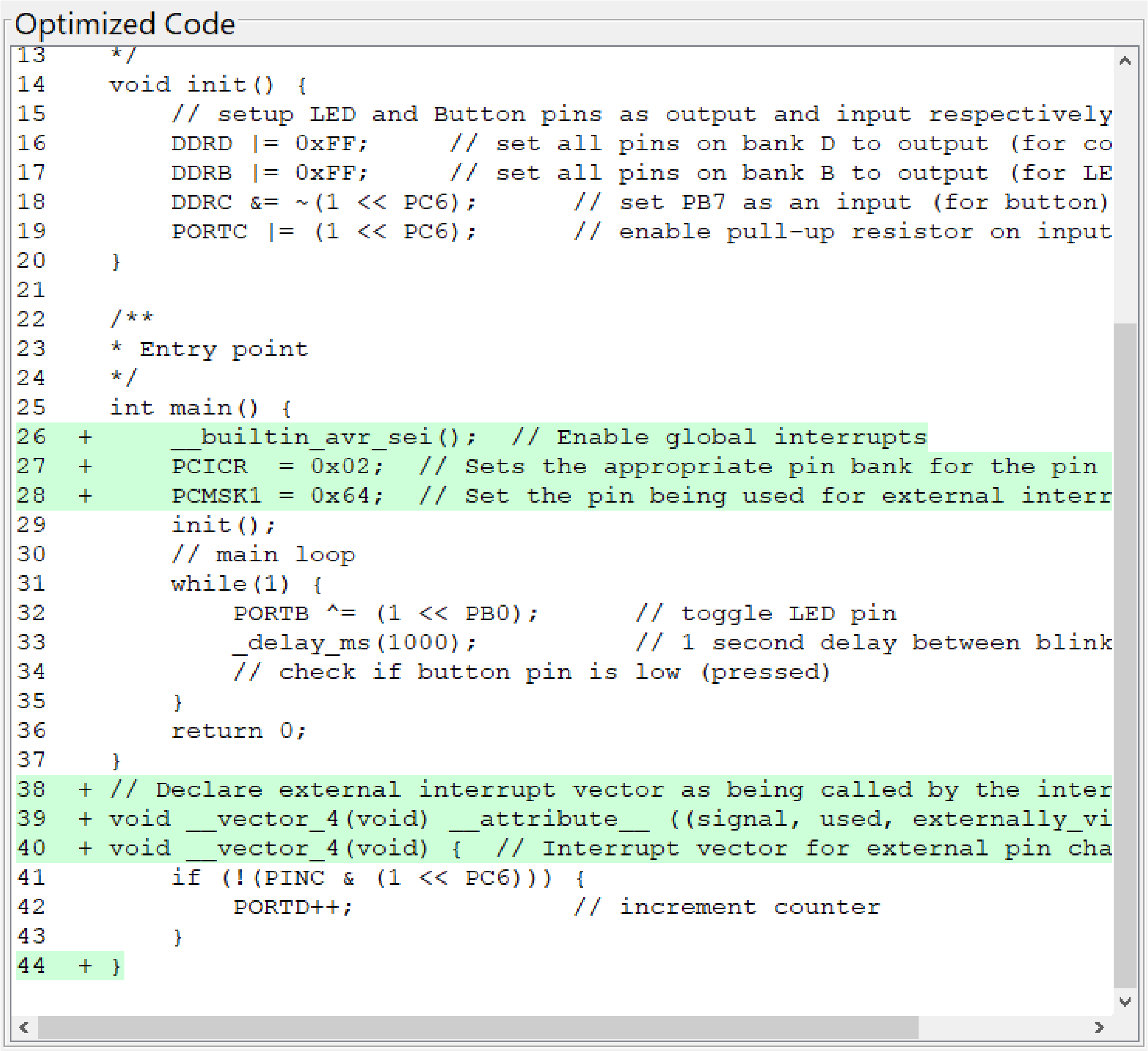
Figure 4.21: An example of the ”Counter/Timer” optimization which eliminates usages of software delay functions (`_delay_ms(...)` and `_delay_us(...)` provided by the util/delay.h library [19]). Lines 28-31 configure the 8-bit timer instance 0, lines 32 and 33 initialize the count and limit variables used to track the current delay value, lines 37-39 and 41-43 are added to check and execute the existing code (line 40) in the case the delay expires, and lines 52-58 are the ISR added to track and increment the variables corresponding to the delay value. The count and limit variables are declared globally at the beginning of the program.

IV Interrupts

To properly take advantage of the Interrupt Handler Unit, scenarios where a state change on a pin is checked and used to perform a task must be identified. For this optimization option, only external interrupts are considered. This would include changes on input pins on the MCU, such as a button press or other signal from a sensor or actuator. The simplest way of obtaining this functionality in software is to simply check the state of the connected pin every iteration of the control loop, and if there is a change (whether it be high to low or low to high) then execute some task. Such an implementation is potentially wasteful as most of the clock cycles are spent checking for the change, rather than performing more meaningful work. By utilizing hardware interrupts, no software is needed to check for changes; if/when a state change occurs, the interrupt handler unit will automatically execute code defined within the corresponding ISR.

Within the Embedded C Source Optimizer, the ”Interrupt” optimization option attempts to match a condition where a pin on a given bank is being checked within the control loop through a conditional statement, where a certain task is executed if the condition evaluates to true. Such a scenario might be checking for a pin going low (in the case of a button press) and performing a task such as: setting an output signal high, writing to the Electrically Erasable Programmable Read-Only Memory (EEPROM) (or other external device), or changing the state of an internal flag used by the MCU. To better demonstrate the changes made by the ”Interrupts” optimization option, an example case is provided in Figure 4.22. There are a variety of other use cases for interrupts, and as such are another potentially wide-reaching use case which the Embedded C Source Optimizer can handle.

Since the ATmegaXX8 line of MCUs supports hardware interrupts on all of the general-purpose I/O registers, the mapping which needs to be done is universal among all of the possible cases of input pins. Any instance where such a state change occurs, the Embed-



Optimized Code

```
13    */
14    void init() {
15        // setup LED and Button pins as output and input respectively
16        DDRD |= 0xFF;     // set all pins on bank D to output (for co
17        DDRB |= 0xFF;     // set all pins on bank B to output (for LE
18        DDRC &= ~(1 << PC6);      // set PB7 as an input (for button)
19        PORTC |= (1 << PC6);      // enable pull-up resistor on input
20    }
21
22    /**
23    * Entry point
24    */
25    int main() {
26  +     __builtin_avr_sei();  // Enable global interrupts
27  +     PCICR  = 0x02;  // Sets the appropriate pin bank for the pin
28  +     PCMSK1 = 0x64;  // Set the pin being used for external interr
29        init();
30        // main loop
31        while(1) {
32            PORTB ^= (1 << PB0);      // toggle LED pin
33            _delay_ms(1000);          // 1 second delay between blink
34            // check if button pin is low (pressed)
35        }
36        return 0;
37    }
38  + // Declare external interrupt vector as being called by the inter
39  + void __vector_4(void) __attribute__ ((signal, used, externally_vi
40  + void __vector_4(void) {  // Interrupt vector for external pin cha
41        if (!(PINC & (1 << PC6))) {
42            PORTD++;              // increment counter
43        }
44  + }
```

Figure 4.22: An example of the "Interrupts" optimization which attempts to replace any conditional statement checking for a change on an input pin for a corresponding ISR which performs the same function. In the case of the imported program, lines 41-43 were moved from the main control loop to the ISR corresponding to the pin which was being checked originally. The interrupt is configured to trigger on a state change of the checked input pin (done in lines 27 and 28), so the original conditional statement is preserved to ensure the included functionality only happens on the intended edge (being the falling edge in this case).

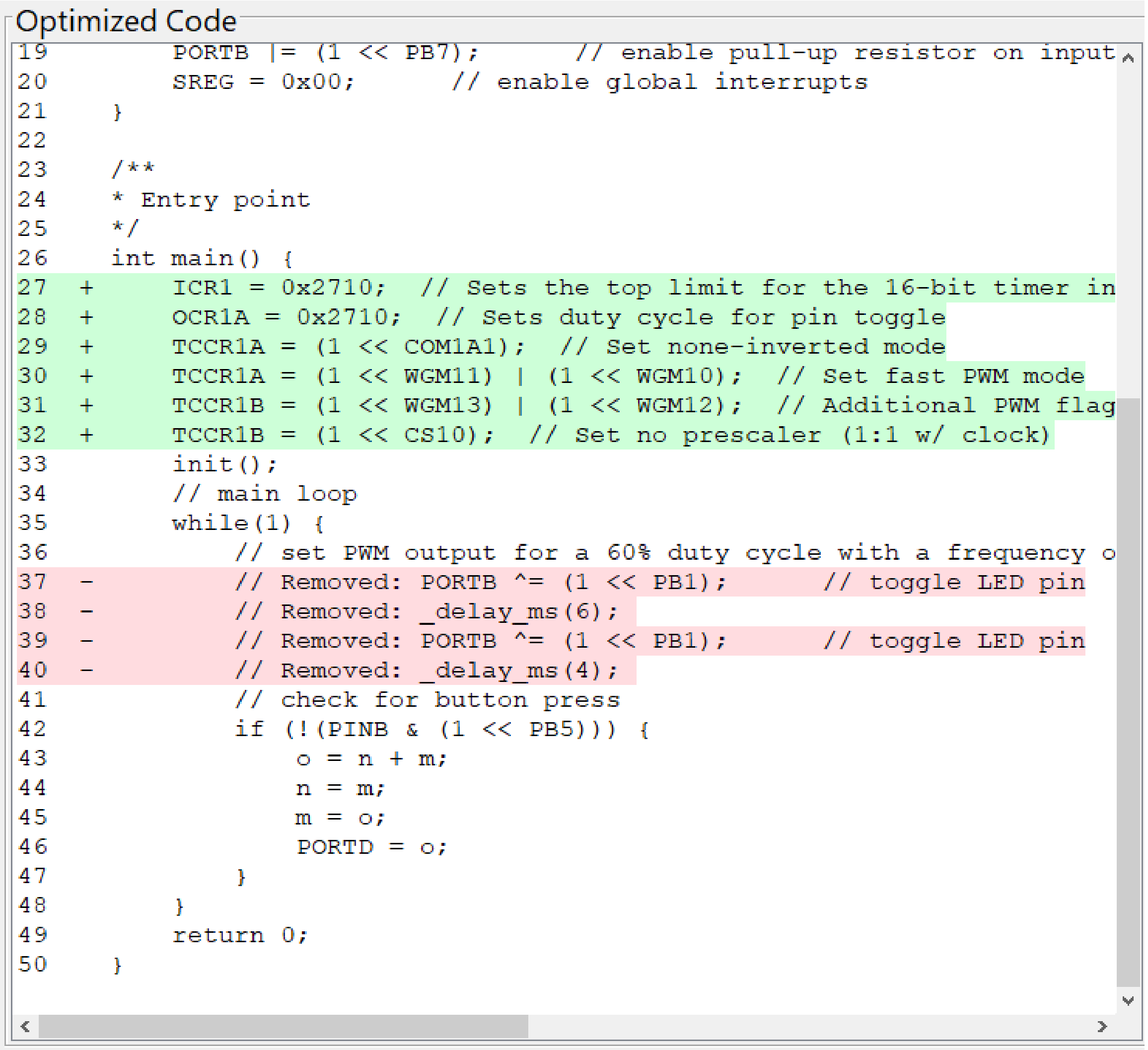
ded C Source Optimizer attempts to place any code within the body of a corresponding conditional statement into an ISR ("`__vector_4(void)`") corresponding to the external interrupt, as well as configuring the specified pin to trigger an interrupt on a state change (seen on lines 27 and 28 in Figure 4.22). The result of these changes is the rising or falling edge (depending on the initial state of the condition from the imported code) on the input pin will be redirected to an ISR rather than manually checking the state of the pin each iteration of the control loop. This increases the potential throughput of the MCU and eliminates any latency imposed by software for executing tasks based on state changes on inputs of the MCU (it should be noted that the Interrupt Handler Unit does introduce some latency into the embedded system; however, the degree to which this affects the overall end-to-end latency as compared to software-driven approaches are negligible) [33].

V Pulse-Width Modulation

PWM is commonly used on embedded platforms for a variety of reasons (for additional information concerning PWM, refer to subsection 2.1.1). In hardware, PWM is implemented as a specific utilization of the counter/timer peripheral, and can generally provide PWM output on pins across multiple different banks. With the use case of PWM being to control the average amplitude and frequency of an output, there are a variety of cases where such functionality can be utilized within embedded software. These include: outputting a fixed-frequency signal (used by an actuator or other device), generating an approximated Direct Current (DC) wave, and controlling actuators which require pulse trains with specific timing requirements (such as servo motors).

Since the PWM outputs on MCUs are generally limited to the same counter/timer hardware, there is some overlap between use cases requiring either of these functionalities. The Embedded C Source Optimizer makes the distinction between these two optimizations at both the scope and depth of the use cases they target. The ”PWM” optimization is more narrow in what it attempts to modify, only matching cases where software delay function calls are nested between assignments to change the state of a given output pin. An example case is provided in Figure 4.23 with further explanation of the implementation. On the other hand, the counter/timer optimization targets a more broad range of use cases, where the only strict requirement is the use of software delay functions within the main control loop of the program. This distinction is made for two reasons: to setup the counter/timer hardware appropriately for either PWM functionality or for triggering interrupts, and to setup the resulting optimized code in a manner which bodes well for each respective optimization and what it is attempting to accomplish. The ”PWM” optimization can be thought of as a more specific application of the counter/timer optimization, where the resulting code is more hardware driven as a result (rather than managing delay and limit values in software to support many different points of execution).

The ”PWM” optimization supports two additional options to modify the underlying behavior of the optimization. The ”Invert Duty Cycle” (or ”IDC”) option will purposefully invert the duty cycle of the PWM signal. This is necessary as there are certain cases where the ”PWM” optimization is unable to determine the duty cycle of the software-generated PWM signal due to not accounting for the initial state of the pin. In the event that the ”PWM” optimization is applied and the duty cycle is not the intended value, this option may help mitigate the root issue. The ”Preserve Frequency” (or ”PF”) option is another secondary option implemented by the ”PWM” optimization and is used to ensure the frequency of the original, software-driven PWM signal is maintained in the optimized implementation. Since the PWM channels may only be used on specific pins, this option is not enabled by default in order to support the maximum number of use cases (since only the PWM channels associated with the 16-bit counter/timer instance 1 support this use case in the case of the ATmega168). In the event that the program relies on the PWM having a set frequency *and* duty cycle, this option may be selected to ensure the functionality is maintained. In the event that the pin used for the software-driven PWM signal cannot be used with the counter/timer instance which supports the ”PF” option, an error message will be displayed to the user.



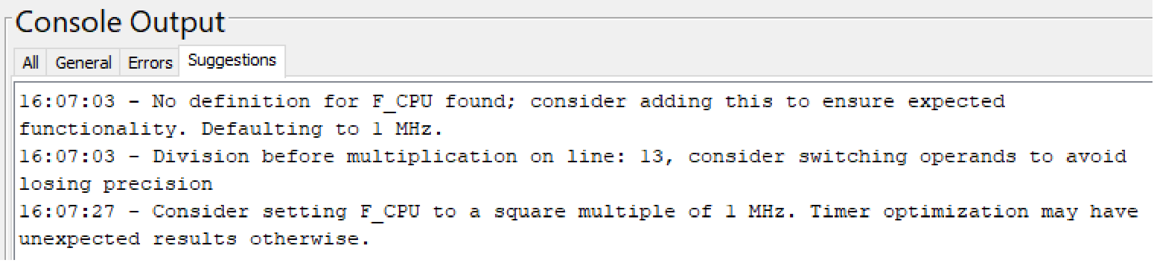
Optimized Code

```
19          PORTB |= (1 << PB7);      // enable pull-up resistor on input
20          SREG = 0x00;       // enable global interrupts
21      }
22
23      /**
24      * Entry point
25      */
26      int main() {
27  +       ICR1 = 0x2710;  // Sets the top limit for the 16-bit timer in
28  +       OCR1A = 0x2710;  // Sets duty cycle for pin toggle
29  +       TCCR1A = (1 << COM1A1);  // Set none-inverted mode
30  +       TCCR1A = (1 << WGM11) | (1 << WGM10);  // Set fast PWM mode
31  +       TCCR1B = (1 << WGM13) | (1 << WGM12);  // Additional PWM flag
32  +       TCCR1B = (1 << CS10);  // Set no prescaler (1:1 w/ clock)
33          init();
34          // main loop
35          while(1) {
36              // set PWM output for a 60% duty cycle with a frequency o
37  -           // Removed: PORTB ^= (1 << PB1);      // toggle LED pin
38  -           // Removed: _delay_ms(6);
39  -           // Removed: PORTB ^= (1 << PB1);      // toggle LED pin
40  -           // Removed: _delay_ms(4);
41              // check for button press
42              if (!(PINB & (1 << PB5))) {
43                  o = n + m;
44                  n = m;
45                  m = o;
46                  PORTD = o;
47              }
48          }
49          return 0;
50      }
```

Figure 4.23: An example of the ”Pulse-Width Modulation” optimization which attempts to use a counter/timer instance on the MCU to output a fixed duty-cycle signal based on software-based PWM. The original program outputs a software-driven PWM signal by inverting the state of the output pin (which can be seen on the ”removed” lines 37 and 39), with calls to a software delay function, ”`_delay_ms(...)` (seen on the ”removed” lines 38 and 40) setting the duty cycle of the signal. This code shows the optimized output after selecting the ”Pulse-Width Modulation” optimization with the option ”Preserve Frequency” which will initialize the timer to output a PWM signal with the same period and duty cycle as the original (with the timer configuration shown on lines 27-32).

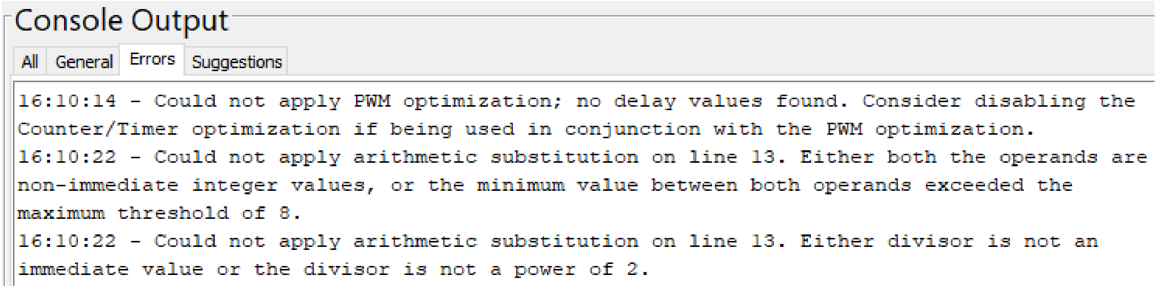
4.3 Educational Considerations

Alongside attempting to objectively improve embedded code by a number of metrics, the Embedded C Source Optimizer also relays how and why certain operations are performed in an effort to teach the end-user how and when they should make similar changes to their code in the future. Some of the ways the utility attempts to do this is: providing detailed comments in the optimized code output (examples of which were shown previously in Figures: 4.21, 4.22, and 4.23), conveying the steps each optimization option takes to apply any changes, and giving suggestions to the user in the form of log output (shown previously as log output in Figure 4.4) which indicate aspects of their code they should consider changing to get the most out of provided optimization options. These are an attempt to make learning embedded concepts more clear, while taking away some of the guess work and the need to review datasheets or other documentation. Further examples of these attempts at aiding the end-user are detailed in Figures 4.24 and 4.25. Furthermore, the information provided for each optimization option which explains what the optimizations do and what considerations should be made are shown in Figures 4.26 and 4.27.



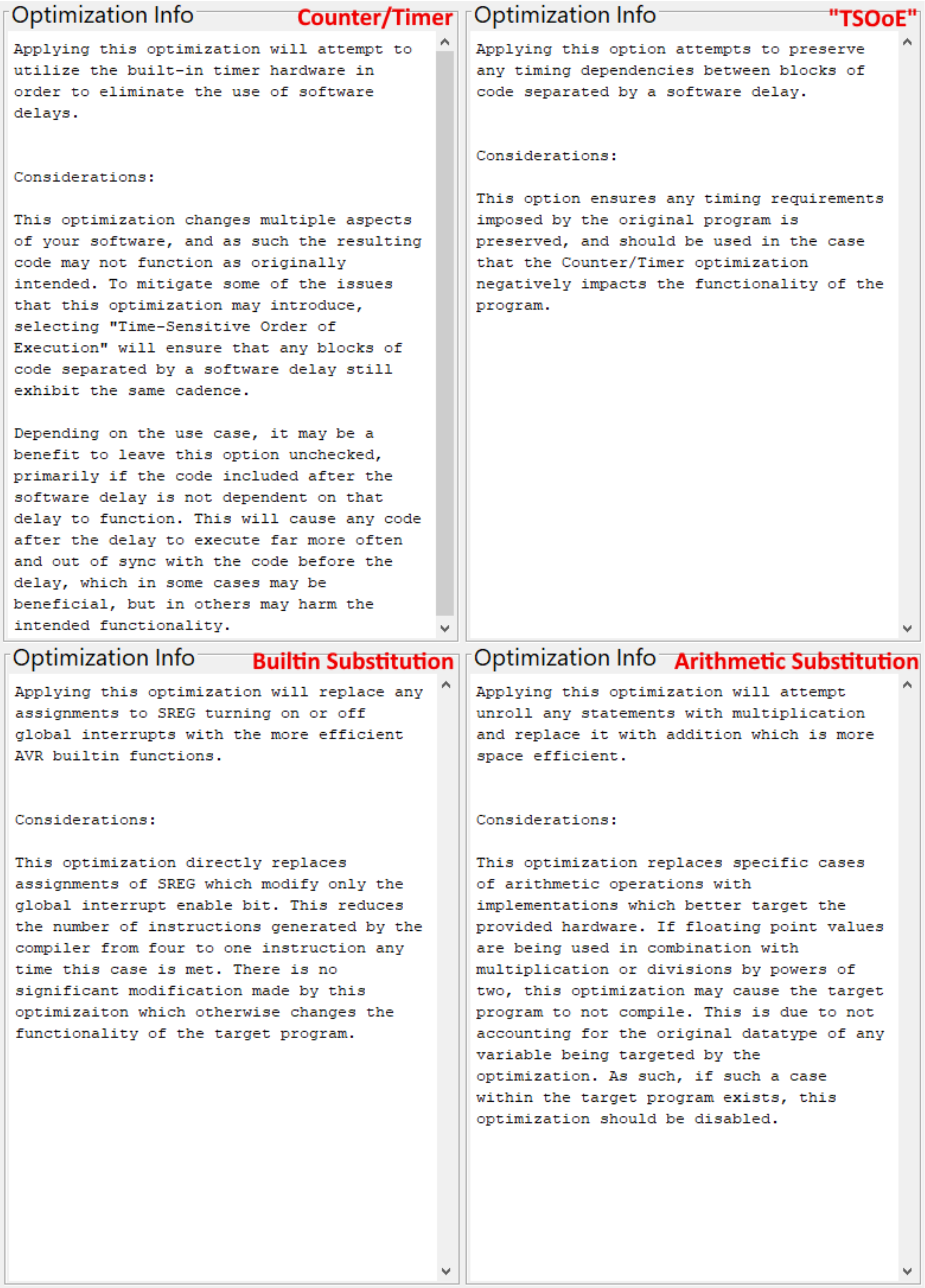
```
16:07:03 - No definition for F_CPU found; consider adding this to ensure expected
functionality. Defaulting to 1 MHz.
16:07:03 - Division before multiplication on line: 13, consider switching operands to avoid
losing precision
16:07:27 - Consider setting F_CPU to a square multiple of 1 MHz. Timer optimization may have
unexpected results otherwise.
```

Figure 4.24: An example of suggestions provided to the end-user based on the optimizations they selected to be run on their code.



```
16:10:14 - Could not apply PWM optimization; no delay values found. Consider disabling the
Counter/Timer optimization if being used in conjunction with the PWM optimization.
16:10:22 - Could not apply arithmetic substitution on line 13. Either both the operands are
non-immediate integer values, or the minimum value between both operands exceeded the
maximum threshold of 8.
16:10:22 - Could not apply arithmetic substitution on line 13. Either divisor is not an
immediate value or the divisor is not a power of 2.
```

Figure 4.25: An example of errors messages concerning why selected optimization options could not be applied to the end-user's code.



Applying this optimization will attempt to utilize the built-in timer hardware in order to eliminate the use of software delays.

Considerations:

This optimization changes multiple aspects of your software, and as such the resulting code may not function as originally intended. To mitigate some of the issues that this optimization may introduce, selecting "Time-Sensitive Order of Execution" will ensure that any blocks of code separated by a software delay still exhibit the same cadence.

Depending on the use case, it may be a benefit to leave this option unchecked, primarily if the code included after the software delay is not dependent on that delay to function. This will cause any code after the delay to execute far more often and out of sync with the code before the delay, which in some cases may be beneficial, but in others may harm the intended functionality.

Applying this option attempts to preserve any timing dependencies between blocks of code separated by a software delay.

Considerations:

This option ensures any timing requirements imposed by the original program is preserved, and should be used in the case that the Counter/Timer optimization negatively impacts the functionality of the program.

Applying this optimization will replace any assignments to SREG turning on or off global interrupts with the more efficient AVR builtin functions.

Considerations:

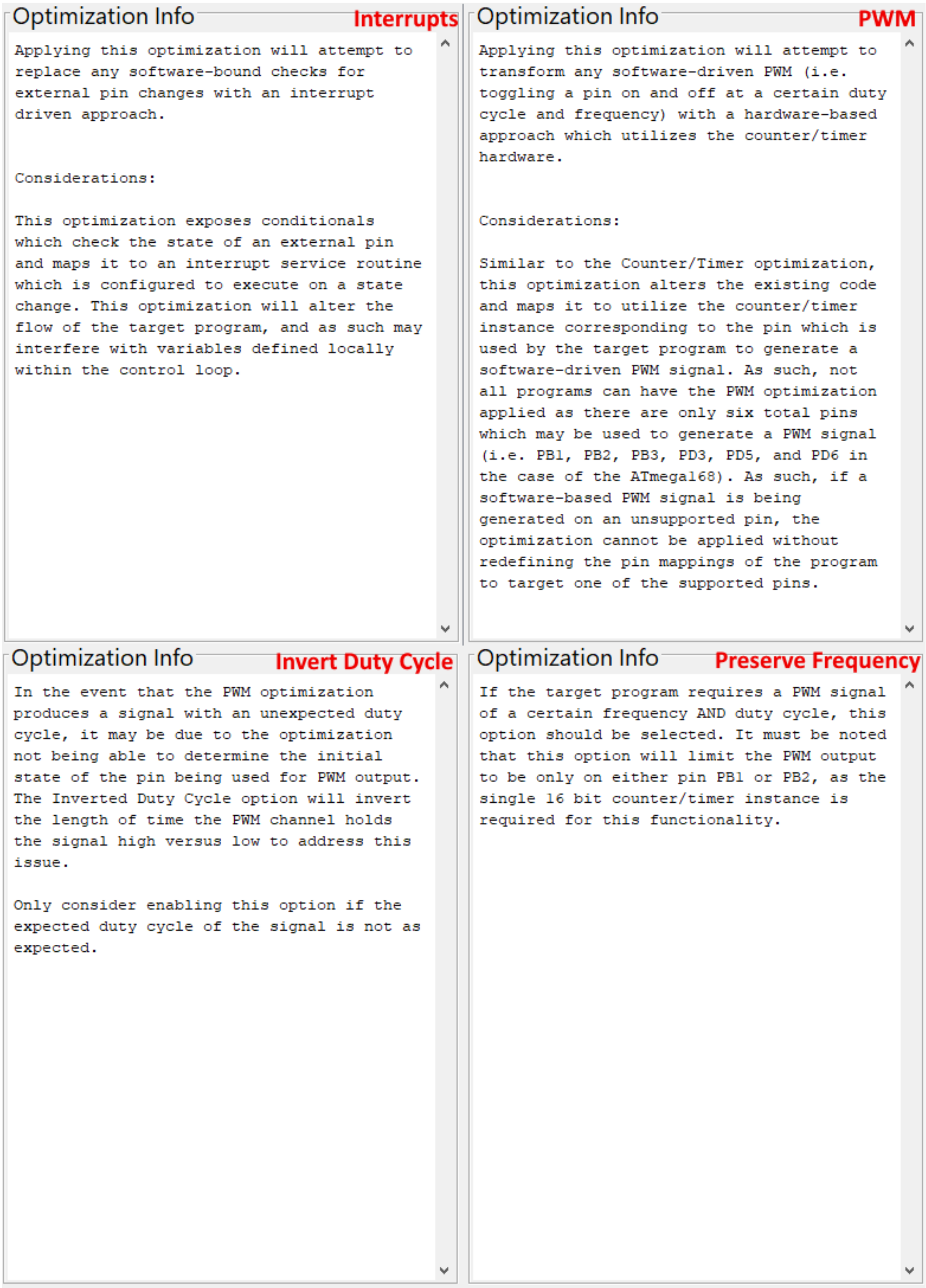
This optimization directly replaces assignments of SREG which modify only the global interrupt enable bit. This reduces the number of instructions generated by the compiler from four to one instruction any time this case is met. There is no significant modification made by this optimizaiton which otherwise changes the functionality of the target program.

Applying this optimization will attempt unroll any statements with multiplication and replace it with addition which is more space efficient.

Considerations:

This optimization replaces specific cases of arithmetic operations with implementations which better target the provided hardware. If floating point values are being used in combination with multiplication or divisions by powers of two, this optimization may cause the target program to not compile. This is due to not accounting for the original datatype of any variable being targeted by the optimization. As such, if such a case within the target program exists, this optimization should be disabled.

Figure 4.26: The optimization info provided by the Embedded C Source Optimizer for the ”Counter/Timer,” ”Time Sensitive Order of Execution” (or ”TSOoE”), ”Arithmetic Substitution,” and ”Built-in Function Substitution” options to aid in the user’s understanding of each.



Optimization Info — Interrupts

Applying this optimization will attempt to replace any software-bound checks for external pin changes with an interrupt driven approach.

Considerations:

This optimization exposes conditionals which check the state of an external pin and maps it to an interrupt service routine which is configured to execute on a state change. This optimization will alter the flow of the target program, and as such may interfere with variables defined locally within the control loop.

Optimization Info — PWM

Applying this optimization will attempt to transform any software-driven PWM (i.e. toggling a pin on and off at a certain duty cycle and frequency) with a hardware-based approach which utilizes the counter/timer hardware.

Considerations:

Similar to the Counter/Timer optimization, this optimization alters the existing code and maps it to utilize the counter/timer instance corresponding to the pin which is used by the target program to generate a software-driven PWM signal. As such, not all programs can have the PWM optimization applied as there are only six total pins which may be used to generate a PWM signal (i.e. PB1, PB2, PB3, PD3, PD5, and PD6 in the case of the ATmega168). As such, if a software-based PWM signal is being generated on an unsupported pin, the optimization cannot be applied without redefining the pin mappings of the program to target one of the supported pins.

Optimization Info — Invert Duty Cycle

In the event that the PWM optimization produces a signal with an unexpected duty cycle, it may be due to the optimization not being able to determine the initial state of the pin being used for PWM output. The Inverted Duty Cycle option will invert the length of time the PWM channel holds the signal high versus low to address this issue.

Only consider enabling this option if the expected duty cycle of the signal is not as expected.

Optimization Info — Preserve Frequency

If the target program requires a PWM signal of a certain frequency AND duty cycle, this option should be selected. It must be noted that this option will limit the PWM output to be only on either pin PB1 or PB2, as the single 16 bit counter/timer instance is required for this functionality.

Figure 4.27: The optimization info provided by the Embedded C Source Optimizer for the ”Interrupts,” ”Pulse Width Modulation” (or ”PWM”), ”Preserve Frequency,” and ”Inverted Duty Cycle” options to aid in the user’s understanding of each.

Chapter 5

EVALUATION

This chapter introduces the findings for each of the aspects and metrics (defined in Section 4.2) as applied to examples of embedded software, and how these findings indicate the relative success of each optimization option. The combination of quantitative and qualitative metrics will be assessed individually, and then considered together in order to determine what value each optimization adds to the utility.

5.1 Experimental Design

A number of tests were designed which attempt to provide an evaluation of each metric discussed in Section 4.2 and run on various embedded programs to assess their effectiveness on a variety of use cases. The programs tested are comprised of multiple purpose-built programs designed to loosely target the optimizations applied by the Embedded C Source Optimizer (see Figures B.1, B.2, and B.3). These programs were designed to represent embedded code that a novice embedded developer may create, and therefore are likely applicable to the provided optimization options. Furthermore, a selection of externally sourced embedded programs (provided in Figures B.10 - B.21) were tested to increase the overall sample size and otherwise confirm the general nature of the optimizations provided by the Embedded C Source Optimizer.

5.1.1 Latency Results

Latency was tested using a Raspberry Pi 3 Model B [37] which was used to simulate signals and track the time between sending and receiving responses to and from the ATmega168. The Raspberry Pi 3 Model B was connected to the ATmega168's input and output pins through two of its General Purpose Input/Output (GPIO) pins, and the program provided in Figure 5.1 was executed on the Raspberry Pi to record and export a set of measured time values corresponding to the delay between the falling edge of the ATmega168's input and the output becoming high. Results for latency testing will be provided in Section 5.2, with the raw data provided in Table B.1.

The script shown in Figure 5.2 simulates an external signal by setting the output pin of the Raspberry Pi low, which is equivalent to the state change the ATmega168 expects. The ATmega168 ”responds” to this low signal by setting a separate external pin high, which the Raspberry Pi then reads. In between these two events, the Raspberry Pi tracks the

```cpp
#include <iostream>
#include <fstream>
#include <cstdio>
#include <ctime>
#include <wiringPi.h>

#define ITERATIONS 1000
#define MAX_DELAY_MS 100
#define MIN_DELAY_MS 1
#define RANDOM

int main(int argc, char **argv) {
  if (argc != 2) {
    std::cout << "No file name argument provided\n";
    return 1;
  }
  std::ofstream file;
  file.open(argv[1]);
  wiringPiSetup();
  pinMode(0, OUTPUT);
  pinMode(1, INPUT);
  // initialize pin as high held high
  digitalWrite(0, HIGH);
  delay(MAX_DELAY_MS);
  std::clock_t timer;
  for (int i = 0; i < ITERATIONS; i++) {
#ifdef RANDOM
    // random time delay
    delay((rand() % (MAX_DELAY_MS - MIN_DELAY_MS)) + MIN_DELAY_MS);
#endif
    // set input pin high
    digitalWrite(0, LOW);
    // start timer
    timer = std::clock();
    // wait for output pin to go high
    while (!digitalRead(1));
    // calculate latency based on start time
    file << ((std::clock() - timer) / (double) CLOCKS_PER_SEC)
         << ((i == (ITERATIONS - 1)) ? "" : ",");
    std::cout << i << '/' << ITERATIONS << '\r';
    // reset input pin
    digitalWrite(0, HIGH);
  }
  file.close();
  return 0;
}
```

Figure 5.1: A simple program developed in C++ which simulates a button press and tracks the time between sending and receiving a response, allowing the end-to-end latency to be measured for a given embedded system. Macros have been defined which allow for changing the number of iterations, the amount of delay included between ”button presses,” and whether or not the delay values should be randomized to better represent the asynchronous nature of external interrupts.

time taken for the ATmega168 to respond, which represents the latency imposed by the ATmega168 system. An additional randomized delay is included within the Raspberry Pi script to ensure no synchronization occurs between the two systems which would otherwise impact the measured results. This ensures asynchronous operation between the two systems, which is amicable to external events which are generally independent of the system state.

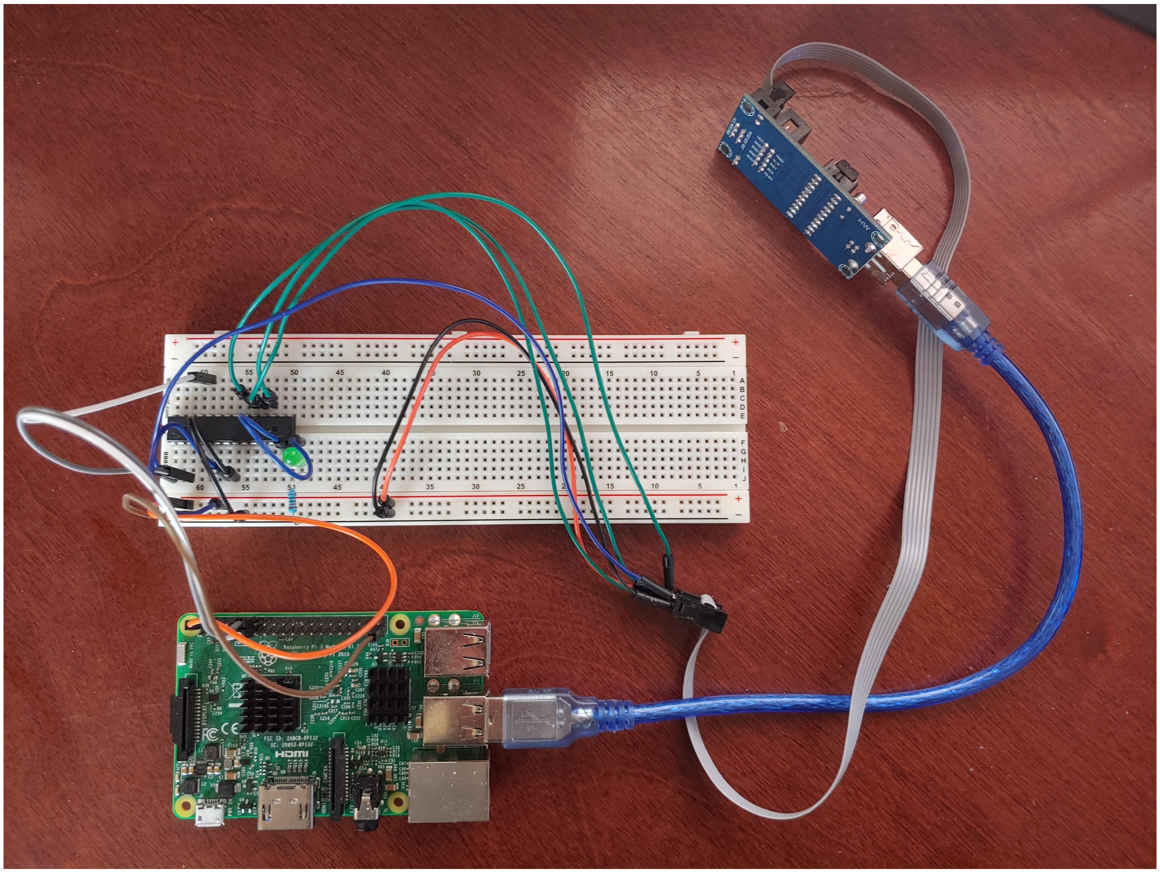


Figure 5.2: The testing configuration used to measure latency across the tested programs. The setup consists of a Raspberry Pi 3 Model B connected to an ATmega168, where two of the GPIO pins of the Raspberry Pi (GPIO 0 assigned to output, GPIO 1 assigned to input) are connected to two pins of the ATmega168 (PC1 assigned to input and PD1 assigned to output).

5.1.2 Utilization Results

Utilization and execution capacity was tested by including the ”utilization” library shown in Figure B.4 and Figure B.5. A Universal Synchronous/Asynchronous Receiver/Transmitter (USART) to Universal Serial Bus (USB) adapter was connected to the serial RX and TX pins of the ATmega168 and a host computer to read the serial data sent by the MCU. Figure 5.3 shows the testing configuration for collecting the utilization data across the various example programs. In combination with the ”utilization” library, it allows for the total cycles passed and the number of times the control loop was iterated to be measured, and as such the cycles per iteration and utilization to be calculated.

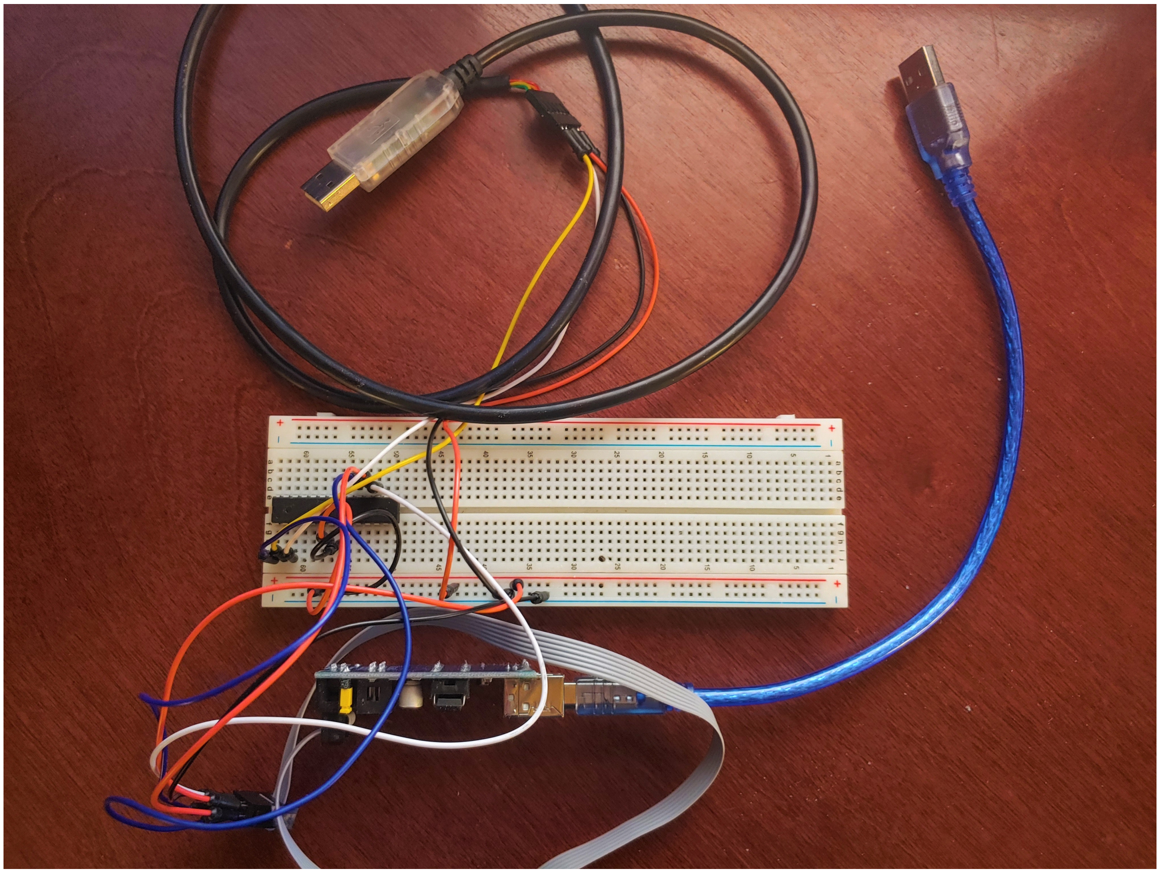


Figure 5.3: The testing configuration used to measure utilization across the tested programs. The setup consists of a USART to USB cable which converts the serial USART connection provided by the ATmega168 to an interface which can be read using a computer with a USB connection. The cycles per iteration measurement can then be read on the host computer, and used to calculate the utilization of the provided firmware.

5.1.3 Extensibility Results

In order to represent extensibility in a quantifiable manner, a program may be analyzed in terms of how many instructions are executed at runtime, and based on that figure project how feasible it would be to add functionality to a given portion of the program. Since extensibility primarily considers adding functionality, a program which executes fewer instructions within a given block of code will be more amicable to adding functionality, while maintaining any requisite deadlines or timing requirements imposed by the existing code. Table 5.1 provides approximate values for how many instructions may be generated by a given piece of code, and will be used to provide a rough idea of how various implementations of a program may be extended upon in their current form.

Table 5.1

This table shows the approximate instructions per source-level construct. This data was gathered through analyzing listing files and comparing the relative number of instructions generated for each of the mentioned source element. These values are not accurate for all possible cases, but they provide a general case for each which makes it straightforward to convert lines of code to assembly-level instructions.

Category	Source Lines	Approx. Instructions	Instructions/line
Load/Store	1	1	1
Declaration	1	3	3
Assignment	1	4	4
Bitwise	1	1	1
Subroutine	1	8	8
If Statement	1	4	4
If Else	2	5	2.5
Loop	2	5	2.5

By utilizing the approximate instructions versus source lines of code (provided in Table 5.1) a program may be evaluated in terms of the number of instructions it executes at runtime. For a given program, the source code constructs (e.g. conditional statements, loops, assignments) are converted into an approximate number of instructions based on the flow of the program. For example, a for loop containing a single assignment which iterates 10 times would execute 5 instructions for the loop, and 10 times 4 instructions for the assignments. Such a program would yield 45 instructions executed. This calculation may provide insight into both the states (referred to here as ”program segments”) which the program may occupy at runtime, and as such the relative difficulty of adding/modifying any given program segment to be determined.

5.1.4 Code Size Results

As discussed previously in Subsection 4.2.1, code size is measured by running the ”avr-size” executable (included in the AVR-GCC toolchain) on compiled embedded programs. The total size (in bytes) is recorded for each example program with each optimization option applied, such that the resulting differences between each can be compared. This is done automatically by the Embedded C Source Optimizer by selecting the ”Compare/Analyze” option after applying any selected optimizations, or by manually exporting each optimized implementation of a program from the Embedded C Source Optimizer, compiling all of the iterations of the program in question, and running ”avr-size” on each compiled program.

5.1.5 Cyclomatic Complexity Results

The cyclomatic complexity of each implementation across the supported optimization options was measuring using CCCC [31]. This is done by running the executable provided by CCCC on each implementation of a program in question and recording the cyclomatic complexity value for each. Each example program provided was tested individually, such that any changes to the cyclomatic complexity can be determined. The optimized implementations of each program were exported from the Embedded C Source Optimizer and tested in the same manner, allowing for a relative comparison between the cyclomatic complexity before and after each optimization option was applied.

5.2 Findings

The resulting measurements taken for each metric discussed previously in Section 4.2.1 are provided in this section. Each example program had optimization options applied individually, and measurements were taken for each implementation with all other factors controlled.

5.2.1 Example Program 1

The following measurements are based on the program provided in Figure B.1.

Latency

Example Program 1 was modified to accommodate testing latency on a controlled hardware setup which would be reused across testing all three programs and their optimized variants. These modifications include updating the pin configuration (in order to support the same set of pins across all tested programs), as well as modifying the conditional branch to ”respond” when detecting a falling edge on the input pin by briefly toggling the state of the output pin. One thousand samples were collected using the testing methodology described previously in Subsection 5.1.1, and the aggregate values for mean, minimum, maximum, and standard deviation for each optimized implementation of Example Program 1 were calculated. Figure 5.4 shows both the average latency and standard deviation for each applicable optimization. In the case of Example Program 1, these optimizations include: ”Counter/Timer”, ”Counter/Timer with Time-Sensitive Order of Execution (TSOoE)”, and ”Interrupts” respectively (additional information concerning the functionality provided by each option was previously detailed in Subsection 4.2.3).

Based on the results shown in Figure 5.4, the ”Counter/Timer” optimization option reduced the average latency between sending and receiving a response to and from the ATmega168 by nearly three orders of magnitude (approximately 1/850). Enabling ”TSOoE” on the ”Counter/Timer” optimization does not have a significant impact on the average latency, although this is to be expected as this option attempts to preserve the original timing of the program. The ”Interrupts” option, similar to the ”Counter/Timer,” provides an average latency of approximately 1/440 of the original. The results demonstrated by the ”Counter/Timer” and ”Interrupts” optimization options contribute positively to the latency of the final program considering the substantial reduction in end-to-end response time. The result is a functionally equivalent program which is much more responsive to external input.

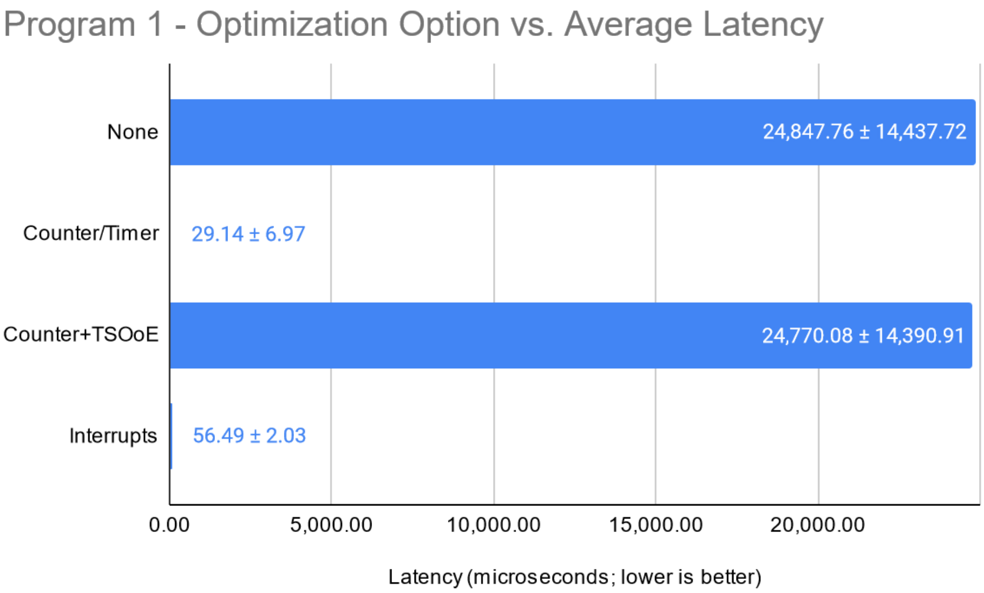


Figure 5.4: A comparison of average latency of Example Program 1 for each applicable optimization across 1000 samples. The standard deviation of each test is shown following the average value data label as ”plus-minus” the average.

To compare the standard deviation of each iteration of the program, the Coefficient of Variation (CV) was calculated by dividing the standard deviation by the mean, providing a normalized representation of latency variance. Figure 5.5 demonstrates the difference in CV between the applicable optimizations for Example Program 1.

Relative to the original (unoptimized) iteration of the program, the ”Counter/Timer” optimization reduced the CV by approximately a factor of 2.4, which indicates that the standard deviation relative to the mean is approximately 2.4 times less than the unoptimized version. This is a positive change, as lower CV in this case correlates to more consistent and reliable response times. The ”Counter/Timer+TSOoE” optimization has an equivalent CV to the unoptimized version, which is expected as the ”TSOoE” option intends to make the original program function exactly the same as it did before, but with improved extensibility and utilization. The ”Interrupts” optimization shows the best result of the tested optimization options, with the CV being reduced by a factor of approximately 16. This low CV indicates the response times are notably consistent, and a worst case scenario for latency will be only slightly above the mean latency.

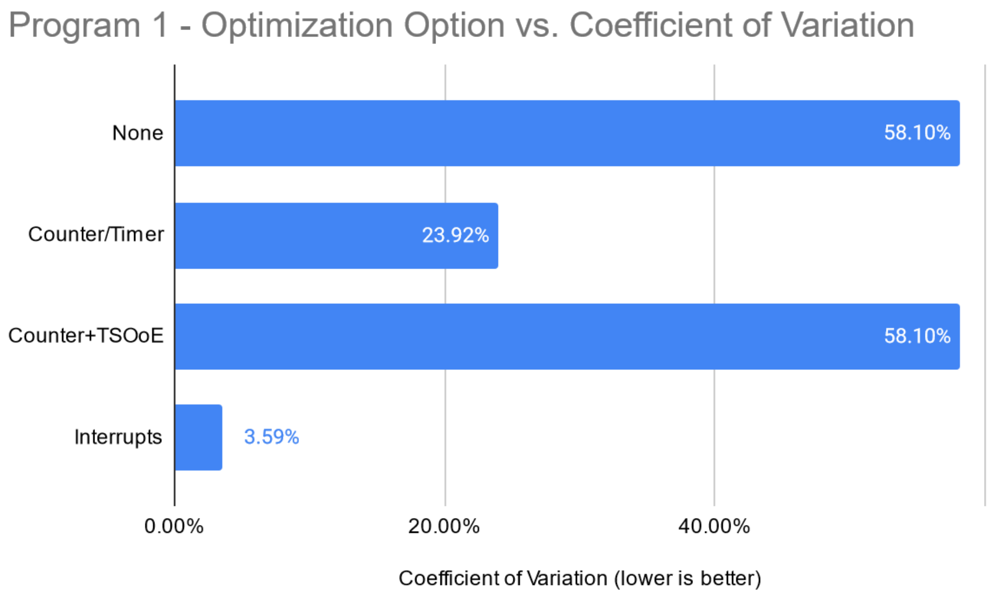


Figure 5.5: A comparison of the CV for each optimization option applied to Example Program 1 using the same data which was collected and aggregated for Figure 5.4.

Utilization & Execution Capacity

Similar to the process for measuring latency, utilization requires the existing program to be modified slightly in order to perform the measurements. The changes made to Example Program 1 in order to perform this test include: including ”latency.h” (shown in Figure B.4), adding a call to ”`util_init()`” during the initialization of the target program, and incrementing ”`iterationCount`” at the end of each iteration of the control loop. The timer instance configured in ”`util_init()`” (with the implementation being provided in Figure B.5) handles calculating the cycles taken by the control loop over a known amount of time, and sends that over USART to the host computer to be recorded.

Figure 5.6 shows the results of running Example Program 1 with each of the optimization options applied, and the corresponding cycles taken by the control loop per iteration. Optimization options which could not be applied are grayed out as they have no effect on the final utilization. Due to the significant disparity between some of the values, the horizontal axis (showing cycles per control loop) is plotted on a log scale. The ”Counter/Timer” optimization improved the measured cycles per iteration by a factor of approximately 32,000, meaning the control loop of the optimized program executes approximately 32,000 times as fast as the unoptimized program. This massive difference is the result of included software delay functions within the main loop, having relatively large delay values (in this case, one second). Since the ”Counter/Timer” optimization eliminates these delay function calls in favor of using the dedicated counter/timer hardware, the added overhead of blocking for the duration of a delay is removed. The ”TSOoE” option improves the utilization in a similar manner, although introduces a bit more overhead through adding checks for an additional count/limit instance (which is used to preserve the original timing of the program). The remaining optimization options do not have any affect on the utilization as they do not remove calls to software delay functions, and therefore the cycles per iteration is left unchanged.

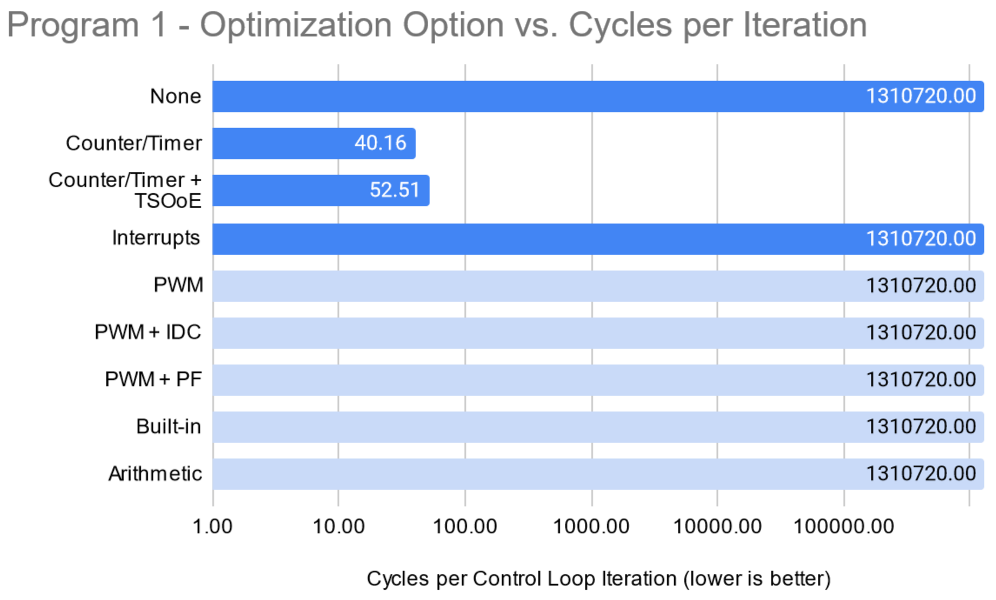


Figure 5.6: A comparison of cycles per iteration for Example Program 1 tested against each optimization option, which indicate the programs ”execution capacity.” Optimizations which could not be applied are grayed out.

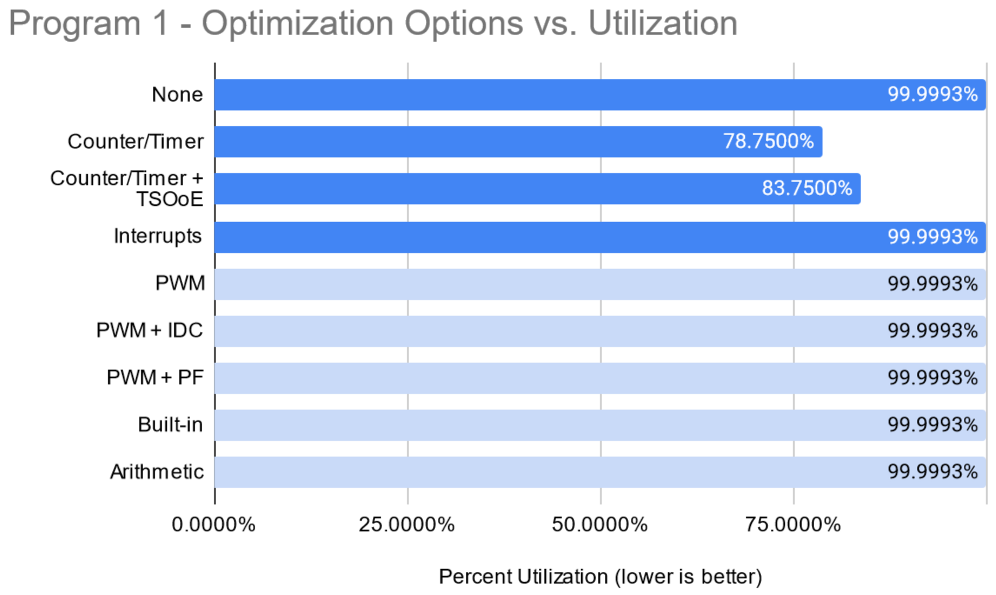


Figure 5.7: A comparison of utilization represented as a percentage for Example Program 1 tested against each optimization option. Optimizations which could not be applied are grayed out.

Figure 5.7 shows utilization as a percentage, where the value is expressed based on the calculation shown previously in Figure 4.11. The same improvements made by the ”Counter/Timer” optimization with and without the ”TSOoE” option can be seen as a reduction in overall utilization. The scale provided by this metric differs from cycles per iteration in that the percentage asymptotically approaches 100%, with greater cycles per iteration contributing less and less to increasing the utilization. Yet, the unoptimized program (and those where the optimization could not be applied) still show a significant increase in utilization over the ”Counter/Timer” optimization.

Extensibility

In the case of Example Program 1, the applicable optimization options and their affect on the extensibility of the program are shown in Figure 5.8. The values included for instructions executed per program segment are derived from extrapolating the approximate number of instructions generated for a given code construct (which can be seen in Table 5.1) from the target program. These values are plotted in order to better understand how the various implementations of the same program contribute to the total instructions executed. As it relates to extensibility, the number of instructions executed can be considered a rough indication of how feasible it is to add functionality to the program in its current form.

Figure 5.8 shows the results of measuring the extensibility of Example Program 1 with each applicable optimization option applied. The ”Delay Function” segment contributes the most amount of instructions executed to the implementations which contain a call to ”`_delay_ms()`.” Since the ”Counter/Timer” optimization (with and without the ”TSOoE” option) attempt to remove any calls to software delay functions, the most significant program segment in terms of instructions executed can be eliminated. Due to the ”Delay Function” segment contributing directly to the iteration period of the control loop (since the subroutine call is nested inside the loop), this program segment sets an upper bound for

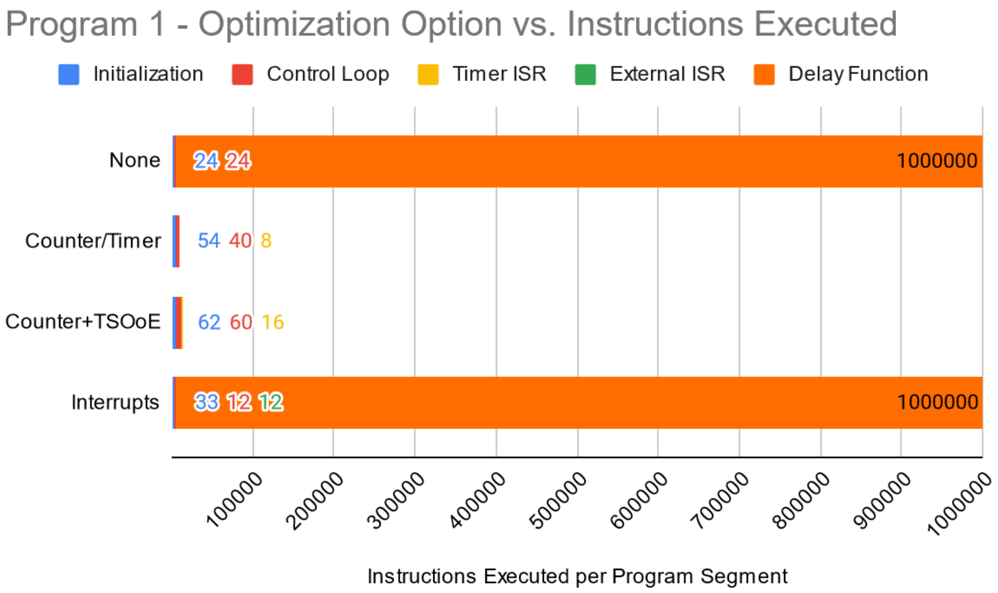


Figure 5.8: A comparison of the approximate number of instructions executed per program segment for Example Program 1 against all applicable optimization options.

how often an additional (added) task can be performed without adding additional logic to split the delay into increments.

In the case of Example Program 1, the difference between the unoptimized and ”Counter/Timer” versions show a difference of approximately 1/10,000 the executed instructions for the optimized iterations. This reduction manifests as making it objectively less complex to add additional functionality when any timing/execution requirements of the functionality do not align with the existing implementation. Due to the reduced execution period by such a substantial factor, a task can be added to the control loop and will be executed approximately 10,000 times as often, reducing the need to add additional logic/overhead to account for different timing requirements. As for the ”Interrupts” optimization option, the number of instruction executed overall is essentially unchanged as it does not target software delay functions. That being said, this representation of instructions executed cannot account for the out-of-order execution provided by the interrupt handler configured by the ”Interrupts” optimization. In the event that the task being added is triggered by an external event, the generated ISR can be used to bypass the iteration period of the control loop entirely.

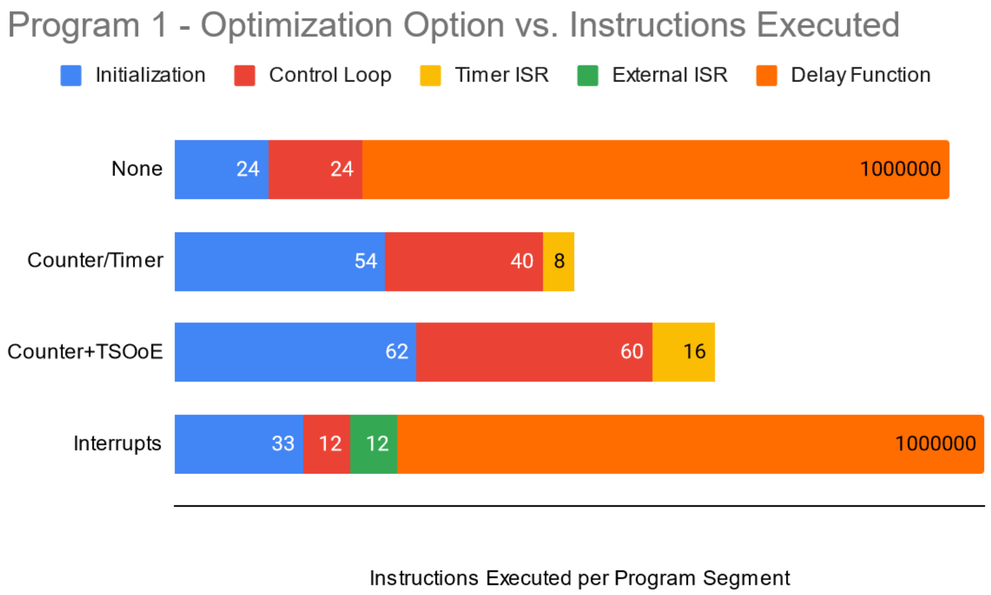


Figure 5.9: A scaled representation of Figure 5.8 to better show the differences between all program segments. The ”Delay Function” segment is largely underrepresented in this figure as a result.

To better understand the relative differences between program segments other than ”Delay Function,” Figure 5.9 shows the same data as Figure 5.8 but with the ”Delay Function” segment heavily underrepresented. This representation is useful as the impact of each optimization option can be analyzed more granularly. For instance, the ”Counter/Timer” optimization increases the initialization segment by a factor of ~2.5, and the control loop segment by ~1.5. With the ”TSOoE” option enabled, these program segments increase to ~3 and ~2 respectively. Furthermore, a new program segment ”Timer ISR” is introduced to check and increment the count/limit variables corresponding to each instance. The increases to the initialization and control loop segments are a result of the additional declarations, assignments, and logic to setup and utilize the counter/timer hardware; however, this additional overhead is vastly outweighed by the elimination of the ”Delay Function” segment. As for the ”Interrupts” optimization, the initialization segment increases for the same reason as the ”Counter/Timer” optimization, but the control loop segment decreases by a factor of one half. The reduction to the control loop comes as a result of moving the logic to check for a state change on a pin to a separate ISR which is called by the interrupt handler.

Code Size

Since code size is a significant factor for embedded systems, attempting to reduce the size of a compiled embedded program is generally considered ideal. The impact on code size of each optimization option as applied to Example Program 1 is represented in Figure 5.10. The ”Counter/Timer” optimization reduces the final code size of the program by roughly one half (the optimized program’s code size is ~45% of the original). This substantial decrease in code size is likely the result of not needing to include the compiled delay function within the final firmware. This difference includes the addition of extra overhead for initialization and handling the counter/limit instances in software, so the net result of removing the delay function calls is likely even greater than what is shown by this representation. The ”TSOoE” option for the ”Counter/Timer” optimization increases code size slightly, as it adds additional logic to ensure any timing requirements of the original program are preserved. The ”Interrupts” optimization increases the final code size over the unoptimized version by about 6% due to the additional overhead added to configure external interrupts. Since this optimization does not do anything to remove calls to delay functions, any overhead added by these calls is still included in the final size.

The other combinations of optimization options were tested but ultimately did not impact the resulting code size due to not being applicable to Example Program 1.

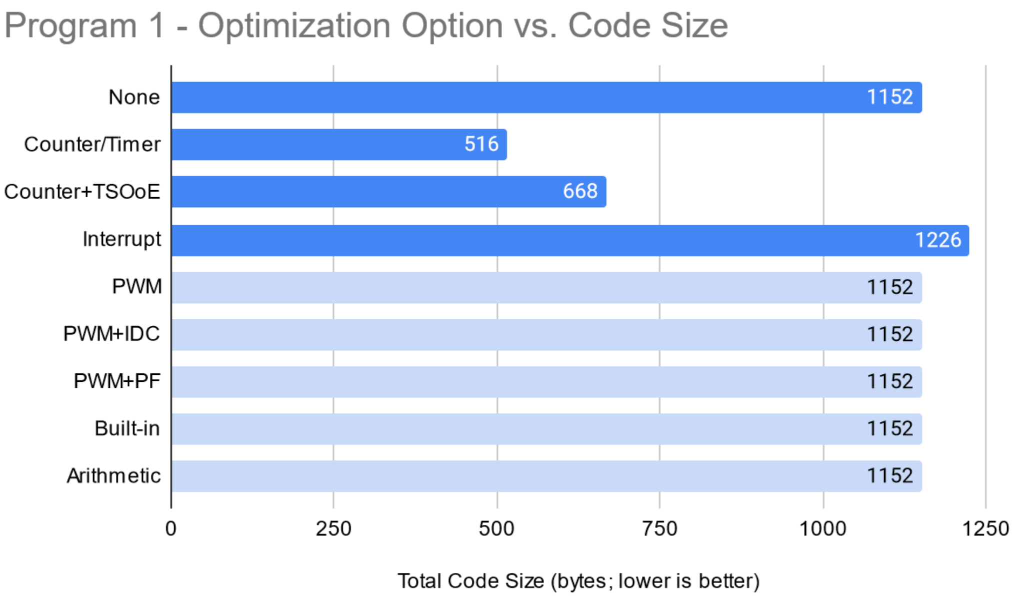


Figure 5.10: A comparison of the size of Example Program 1 after being compiled with each optimization option applied. Optimizations which are not applicable to the program are grayed out.

Cyclomatic Complexity

Cyclomatic complexity is represented as a value, where higher measurements indicate higher levels of complexity (where complexity refers to the total number of paths through a program). Figure 5.11 shows the measured cyclomatic complexity for each optimization option as applied to Example Program 1. The Counter/Timer optimization increases the cyclomatic complexity over the unoptimized implementation by ~1.6, and ~2.6 with the ”TSOoE” option applied. These increases are a direct result of the logic added to increment and check the counter/limit instances which control the delay. Since cyclomatic complexity is relative to the number of paths and this optimization adds as many if statements as are needed to eliminate all the delay function calls, the complexity inevitably scales with this added logic. While not ideal, the benefits as shown by the recorded measurements for previously introduced metrics outweigh this added complexity.

The ”Interrupts” optimization option, while applicable to Example Program 1, does not

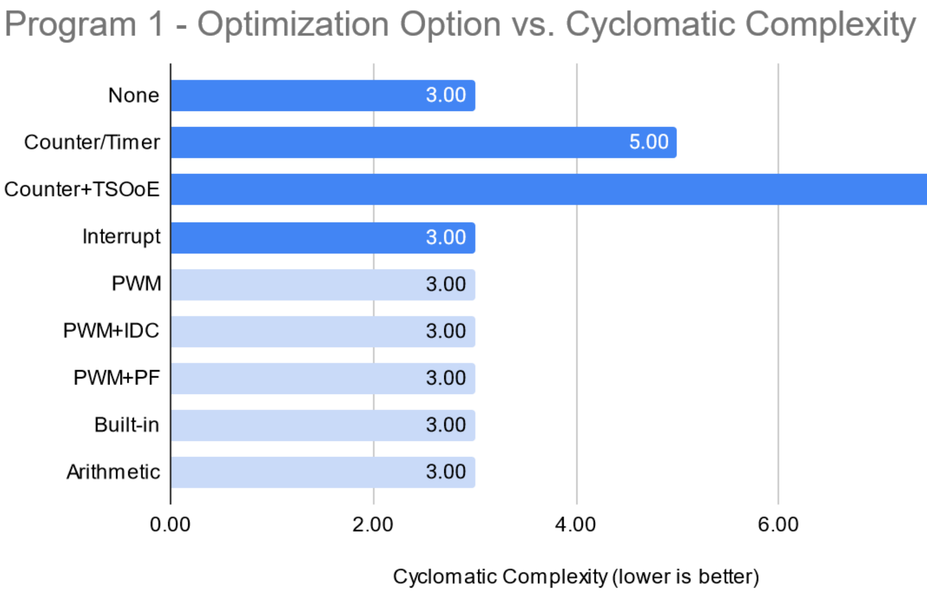


Figure 5.11: A comparison of cyclomatic complexity of Example Program 1 with each optimization option applied. Optimizations which are not applicable to the program are grayed out.

affect the cyclomatic complexity of the resulting implementation. All further optimization options listed in Figure 5.11 do not apply to this program, and are therefore grayed out.

5.2.2 Example Program 2

Example Program 2 (shown in Figure B.2) is another instance of a simple, beginner-level embedded program which can be improved upon in a variety of ways.

Latency

Figure 5.12 shows the average latency for Example Program 2 with the applicable optimization options applied, and shows similar improvements to that of Example Program 1 (previously discussed in Subsection 5.2.1), in that the ”Counter/Timer” and ”Interrupts” optimizations show a substantial decrease in average latency. The ”TSOoE” option performs similarly as well, with no significant difference between the optimized and unoptimized versions (which is to be expected due to ”TSOoE” attempting to preserve the original timing).

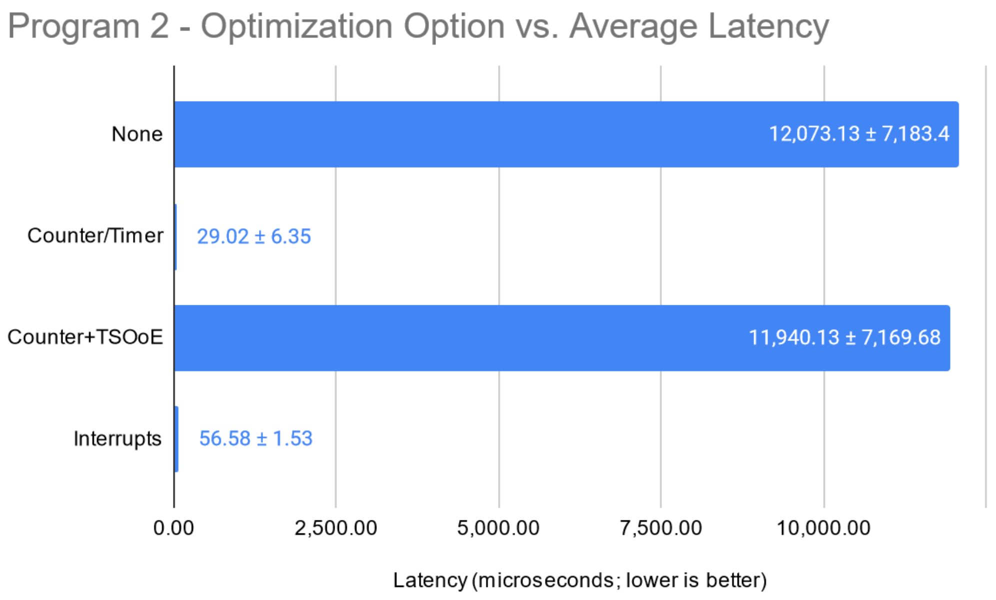


Figure 5.12: A comparison of average latency of Example Program 2 for each applicable optimization across 1000 samples. The standard deviation of each test is shown following the average value data label as ”plus-minus” the average.

The CV for Example Program 2 (shown in Figure 5.13) is also similar to Example Program 1, with the ”Counter/Timer” optimization showing improved consistency across all samples collected for latency. The ”Interrupts” option also shows substantially lower CV than any of the other options. Once again, the ”TSOoE” option does not have a significant impact in this regard.

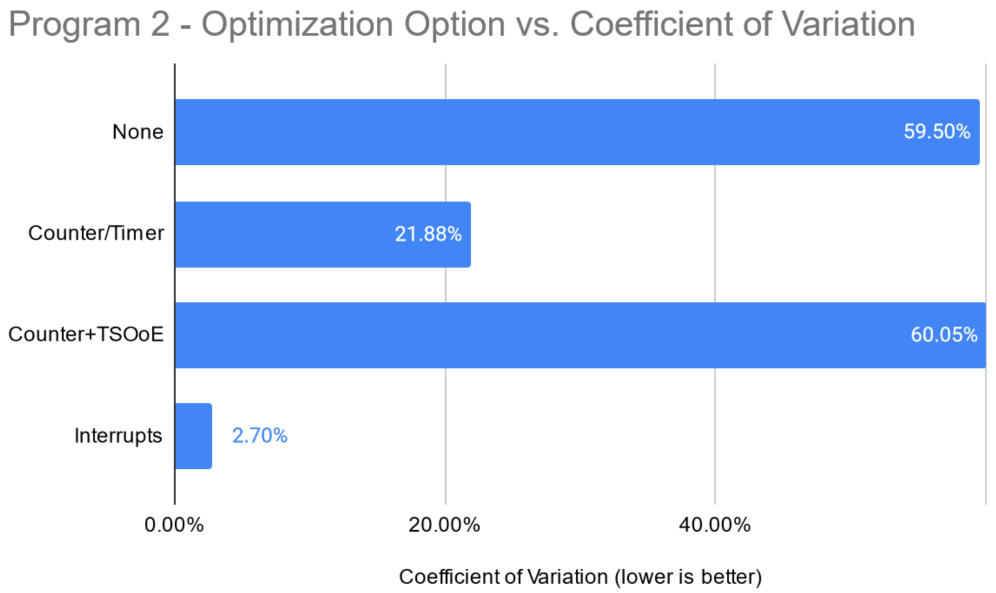


Figure 5.13: A comparison of the CV for each optimization option applied to Example Program 2 using the same data which was collected and aggregated for Figure 5.12.

Utilization & Execution Capacity

Utilization and cycles per iteration for Example Program 2 was decreased substantially by the ”Counter/Timer” optimization both with and without ”TSOoE” enabled (shown in Figure 5.14). The reduction is on the level of five orders of magnitude due to the blocking nature of software delay functions. The ”Interrupts” optimization does not influence the measured cycles per iteration as it does not attempt to remove calls to delay functions. The remaining optimization options are not applicable to Example Program 2, and thus are grayed out.

Figure 5.15 shows utilization for Example Program 2 as a percentage, and shows similar results to Example Program 1, where the unoptimized program (and those where the optimization option could not be applied) approaches 100%. An interesting distinction between Program 1 and 2 is how the ”TSOoE” option affects the utilization. In the case of Program 2, the utilization decreases when enabling this option, where the opposite is true for Program

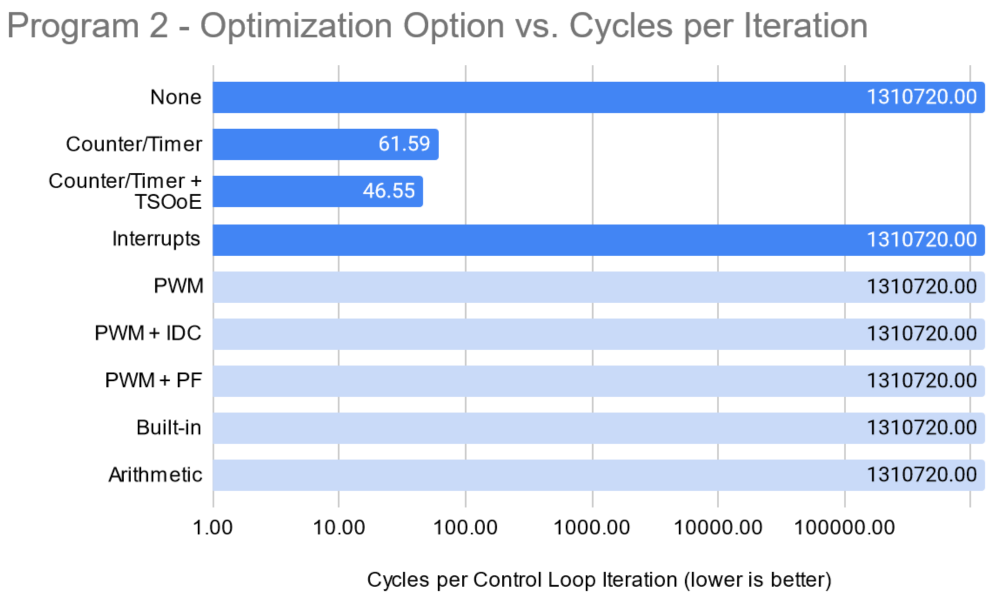


Figure 5.14: A comparison of cycles per iteration for Example Program 2 tested against each optimization option, which indicate the programs ”execution capacity.” Optimizations which could not be applied are grayed out.

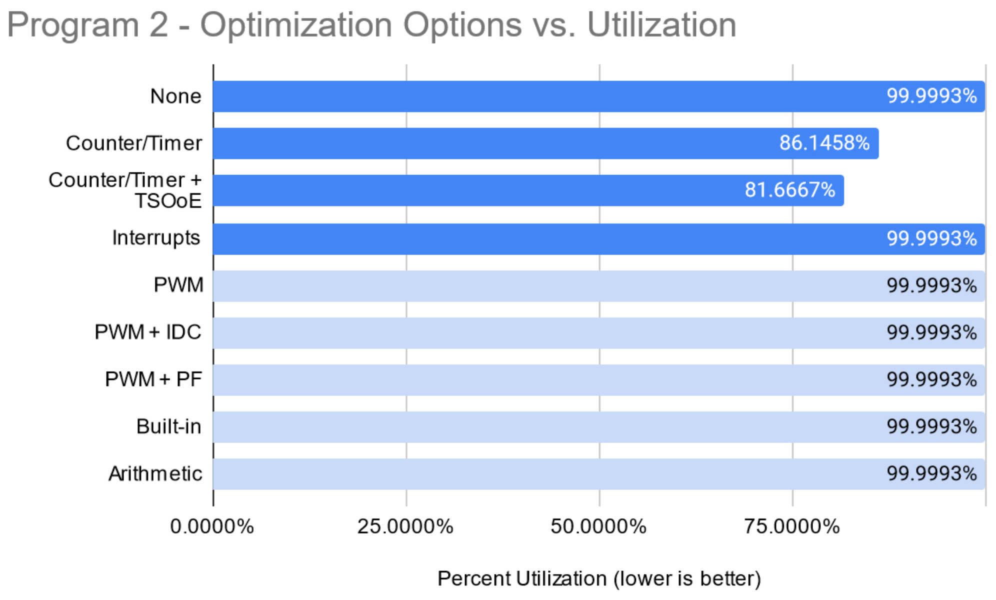


Figure 5.15: A comparison of utilization represented as a percentage for Example Program 2 tested against each optimization option. Optimizations which could not be applied are grayed out.

1. This is likely due to the size of the task (in both instructions and lines of code) following the call to the software delay function. Since the conditional statement which is added by the ”TSOoE” option is more or less constant across various programs, the utilization will be affected more as the size of the code executed after such a delay increases.

Extensibility

The extensibility of Example Program 2 with applicable optimizations applied shows a similar result to that of Example Program 1, in that the ”Counter/Timer” optimization significantly reduces the overall instructions executed due to eliminating calls to software delay functions. The results shown in Figure 5.16 show the relative difference between these optimizations. This representation does not show a meaningful improvement made by the ”Interrupts” optimization, although still arguably improves the extensibility of the program due to allowing out-of-order execution.

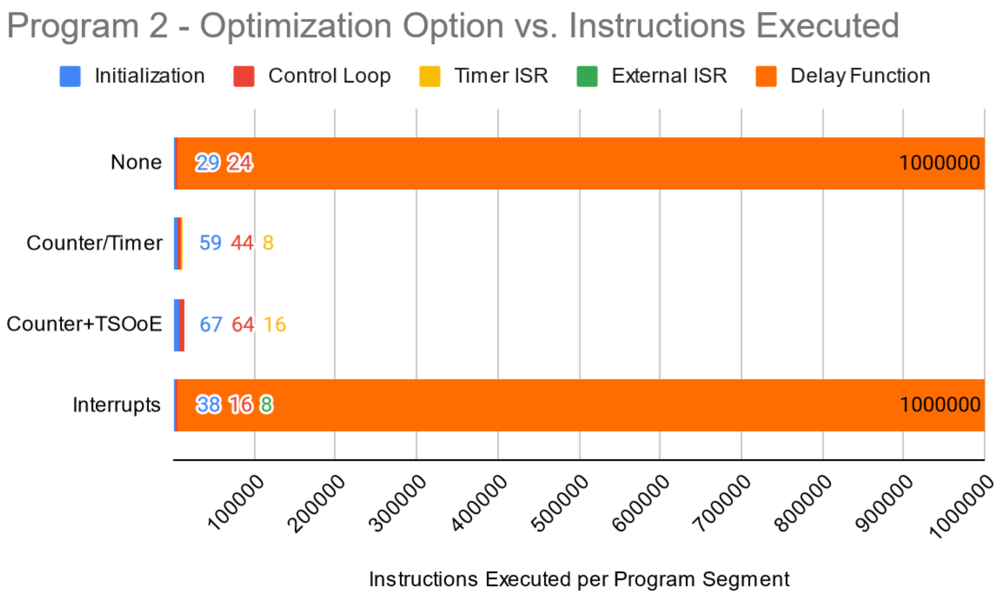


Figure 5.16: A comparison of the approximate number of instructions executed per program segment for Example Program 2 against all applicable optimization options.

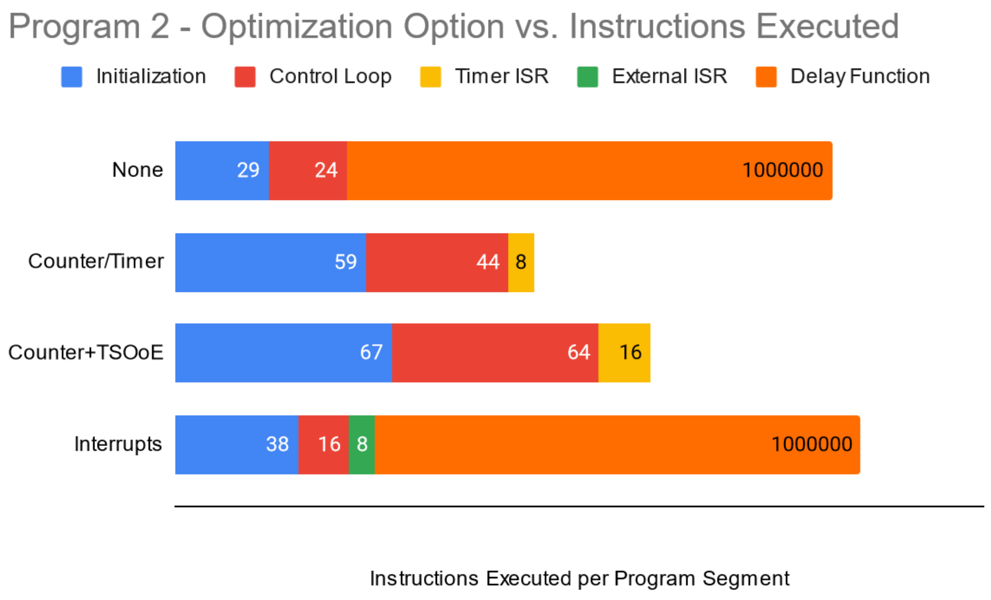


Figure 5.17: A scaled representation of Figure 5.16 to better show the differences between all program segments. The ”Delay Function” segment is largely underrepresented in this figure as a result.

A scaled representation of Figure 5.16 is provided in Figure 5.17, where the size of the Delay Function segment is massively underrepresented in an effort to better display the differences between the other segments. Similarly to the results from Program 1, this program behaves similarly when applying each applicable optimization option. In general, the optimizations all increase the initialization segment due to adding code to set up/initialize the hardware peripherals. Also the control loop after applying the ”Counter/Timer” optimization with and without ”TSOoE” increases in terms of the number of instructions executed due to checking for any given counter instance expiring each iteration. The ”Interrupts” optimization reduces the size of the control loop as it moves the logic for checking the state of a pin from the control loop to a separate ISR.

Code Size

The code size for Example Program 2 (shown in Figure 5.18) shows similar results to that of Program 1, in that the ”Counter/Timer” optimization greatly reduces the final code size as compared to the unoptimized implementation, and the ”Interrupts” optimization slightly increases the final code size. These results are promising as even in the case where code size increases, it only does so by about 7%. By combining various optimization options, more ideal results could potentially be achieved.

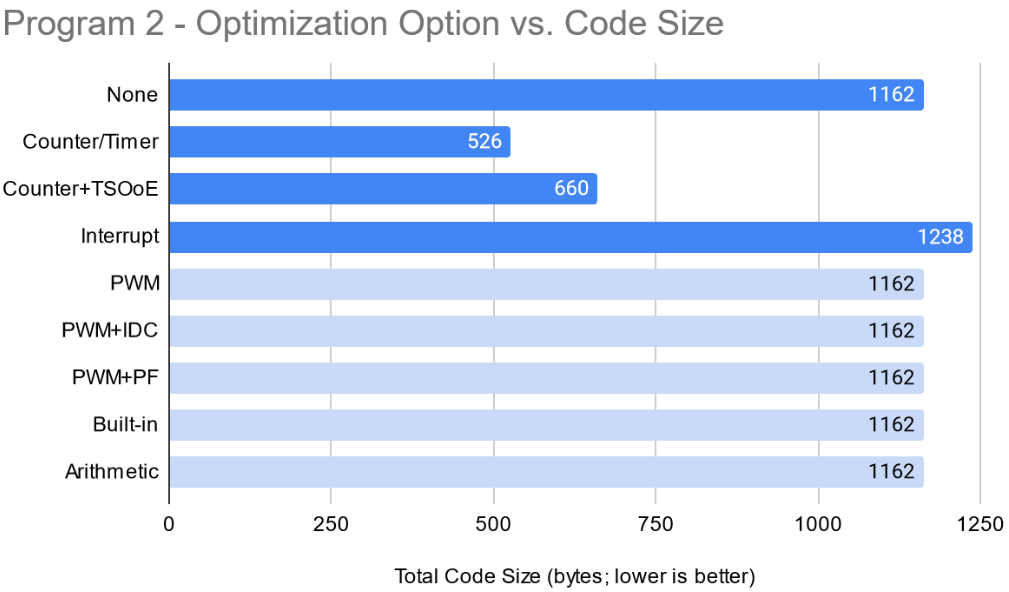


Figure 5.18: A comparison of the size of Example Program 2 after being compiled with each optimization option applied. Optimizations which are not applicable to the program are grayed out.

Cyclomatic Complexity

The ”Counter/Timer” optimization with and without ”TSOoE” shows an increase in cyclomatic complexity by a factor of 1.5 and 2.25 respectively due to the additional logic added to control the timer instances. The only other applicable optimization option (”Interrupts”) shows no change to the final complexity. While not desirable, this level of increase to the complexity after applying these optimizations is outweighed by the other potential benefits they introduce.

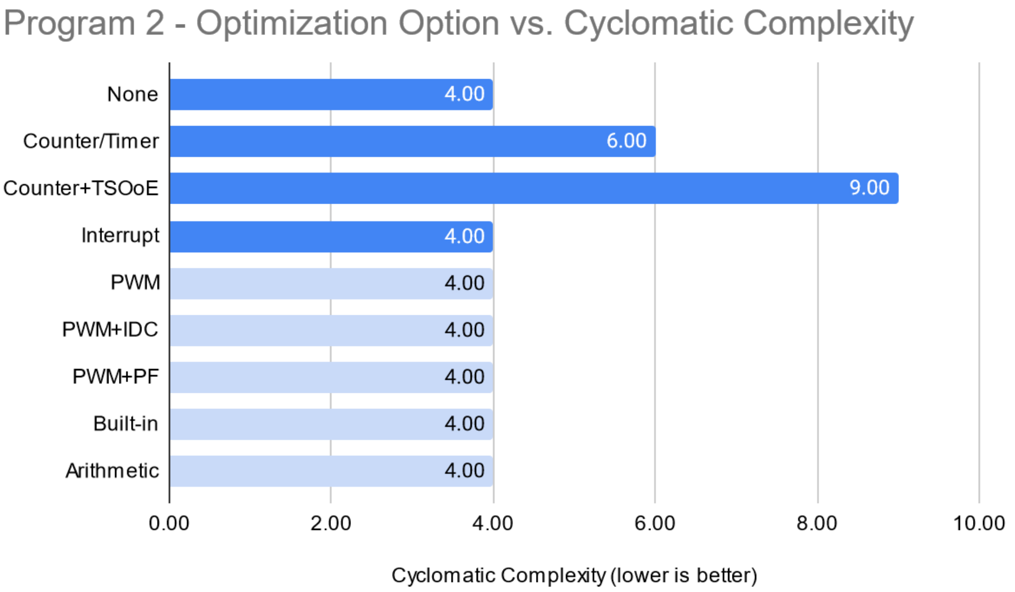


Figure 5.19: A comparison of cyclomatic complexity of Example Program 2 with each optimization option applied. Optimizations which are not applicable to the program are grayed out.

5.2.3 Example Program 3

Example Program 3 (shown in Figure B.3) is of a similar level of complexity (in terms of functionality) to both previous example programs, but the function of the program is designed to take advantage of some optimization options which the others do not.

Latency

Average latency for Example Program 3 is shown in Figure 5.20 with all applicable optimization options listed. The ”Counter/Timer” optimization shows a substantial improvement in average latency over the unoptimized version (having about 1/80 the average latency), although with ”TSOoE” enabled the results align more closely to the original. While previous tests showed the ”TSOoE” option having nearly identical latency to the unoptimized version of the same program, Program 3 shows approximately a 13% reduction in average latency versus the original. It is not immediately obvious why this difference exists between both versions, and may be the result of outlier data points (considering the standard deviation is notably similar between both). The ”Interrupts” optimization shows a similar decrease to average latency as the ”Counter/Timer” optimization, having roughly 1/70 the average latency as the unoptimized version. The ”PWM” optimization option is applicable to this program, and as such when applied shows even lower average latency (having roughly 1/120 the average latency versus the unoptimized version, both with and without the ”Inverted Duty Cycle [IDC]” option applied). The ”Built-In Function Substitution” option shows no change to average latency, as the only change it makes is performed to the initialization phase of the program.

As for the CV of each optimization option (shown in Figure 5.21), the ”Counter/Timer” optimization shows little improvement in consistency over the unoptimized version, despite the average latency being much lower. To further argue that there may be outliers included

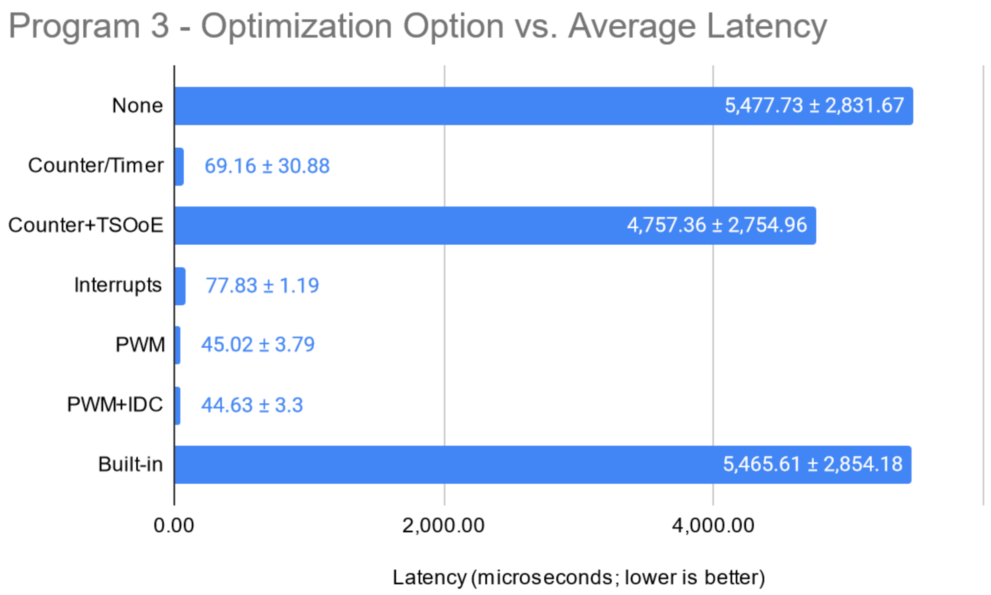


Figure 5.20: A comparison of average latency of Example Program 3 for each applicable optimization across 1000 samples. The standard deviation of each test is shown following the average value data label as ”plus-minus” the average.

in the ”TSOoE” data, the CV is notably higher than the unoptimized version (by about 12%), indicating there was more variance in the data relative to the mean. The ”Interrupts” optimization shows the best CV of any optimization applied to Program 3, suggesting the maximum expected latency is very close to the mean. The ”PWM” optimization both with and without ”IDC” applied show similar levels of improvement to the CV over the unoptimized implementation, although are not quite as effective as the ”Interrupts” optimization in this regard (despite having lower average latency). Once again, the ”Built-in” optimization shows the same result as the unoptimized version of Program 3 due to not modifying anything in the control loop. For additional information concerning each of the introduced optimization options, refer to Subsection 4.2.3.

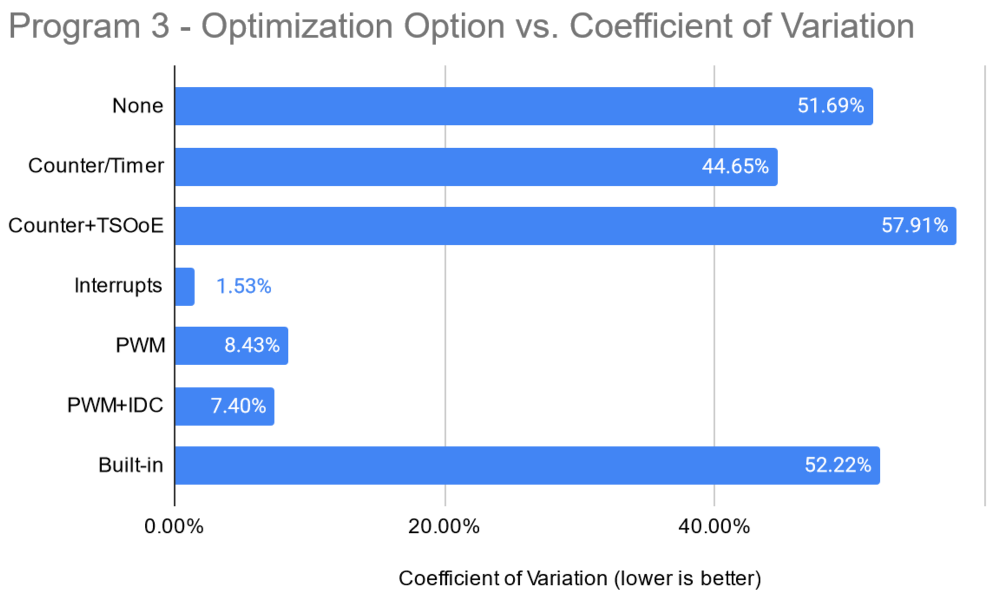


Figure 5.21: A comparison of the CV for each optimization option applied to Example Program 3 using the same data which was collected and aggregated for Figure 5.20.

Utilization & Execution Capacity

The cycles per iteration measurements for Program 3 (shown in Figure 5.22) show similar results to the previous two programs. The ”Counter/Timer” optimization shows a significant reduction in cycles per iteration over the unoptimized version both with and without ”TSOoE” applied. The ”Interrupts” optimization does not significantly change the cycles per iteration due to the software delay function calls still being included. The ”PWM” optimization with and without both options (”IDC” and ”Preserve Frequency [PF]”) demonstrate the best result for utilization of any optimization tested, with a reduction in cycles per iteration by a factor of approximately 1/270. Another way to convey this measurement would be that with the ”PWM” optimization applied, the control loop executes 270 times as fast as the unoptimized version does. This is primarily due to the removal of software delay function calls, as well as having less overhead than the ”Counter/Timer” optimization (as the ”PWM” optimization does not need to track counter/limit instances in software). Both the ”Built-in” and ”Arithmetic Substitution” optimizations had no affect on the cycles per iteration or utilization.

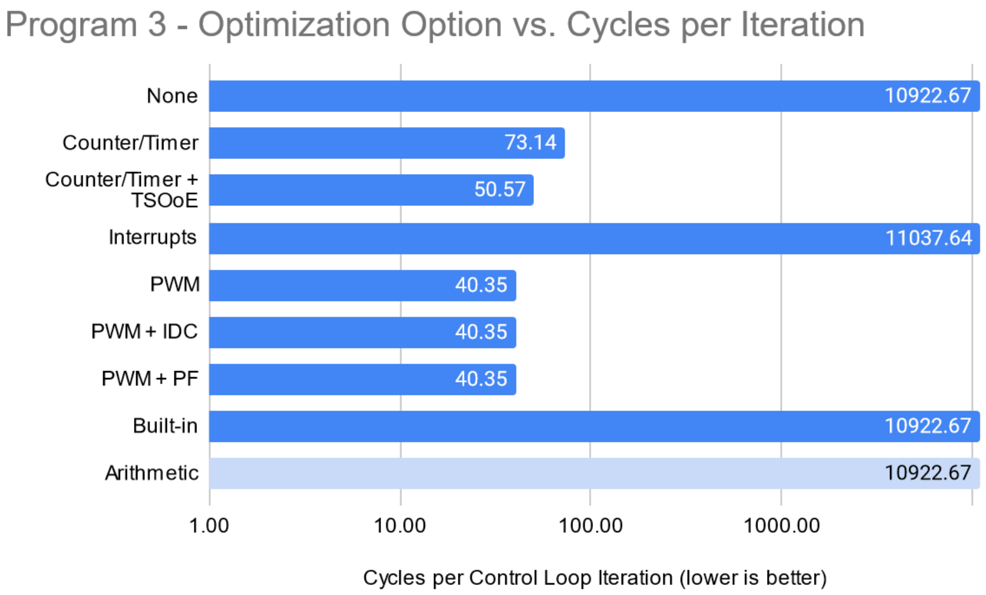


Figure 5.22: A comparison of cycles per iteration for Example Program 3 tested against each optimization option, which indicate the programs ”execution capacity.” Optimizations which could not be applied are grayed out.

Figure 5.23 shows utilization as a percentage, with the effective optimization options substantially reducing the corresponding value. As these measurements are relative to those taken of the control program (which is shown in Figure B.6), the resulting utilization values do not scale linearly with the cycles per iteration.

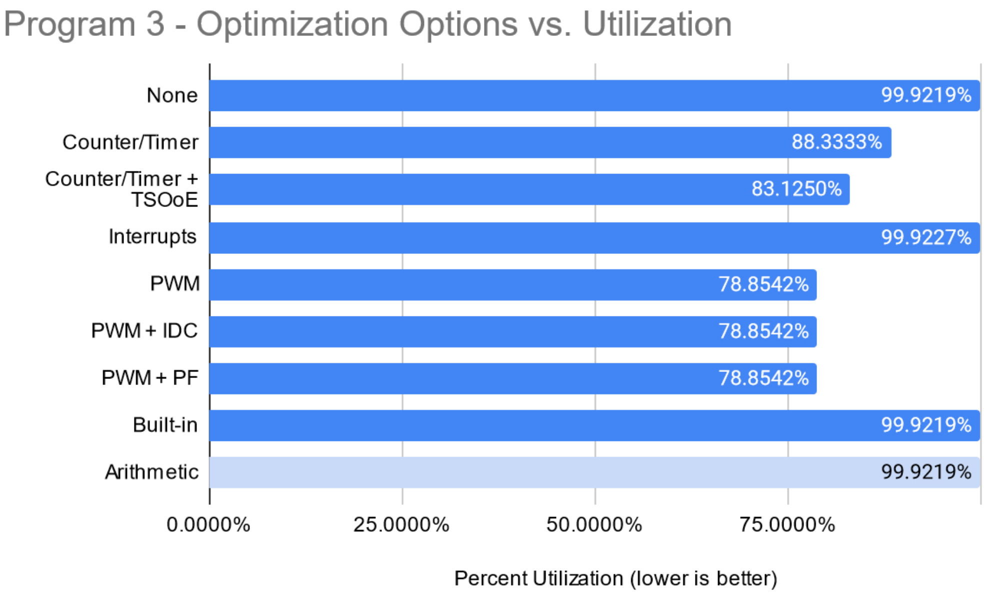


Figure 5.23: A comparison of utilization represented as a percentage for Example Program 3 tested against each optimization option. Optimizations which could not be applied are grayed out.

Extensibility

Figure 5.24 shows the approximate number of instructions executed per program segment of Example Program 3 as it relates to extensibility. The ”Counter/Timer” optimization both with and without ”TSOoE” enabled show similar levels of improvement, allowing additional functionality to be added and have it execute much more frequently than what the unoptimized version would be capable of in its existing form. The ”Interrupts” optimization had negligible impact on the instructions executed as most of its total is still taken up by the delay function segment. The ”PWM” optimization (and its variants with the ”IDC” and ”PF” options applied) shows a substantial decrease in the total number of instructions executed for the same reason as the ”Counter/Timer” optimization. Finally, the ”Built-in” optimization has marginal impact on the total number of instructions executed.

Figure 5.25 shows a scaled version of the data represented in Figure 5.24 to better see the differences between the smaller program segments. The same behavior from the ”Coun-

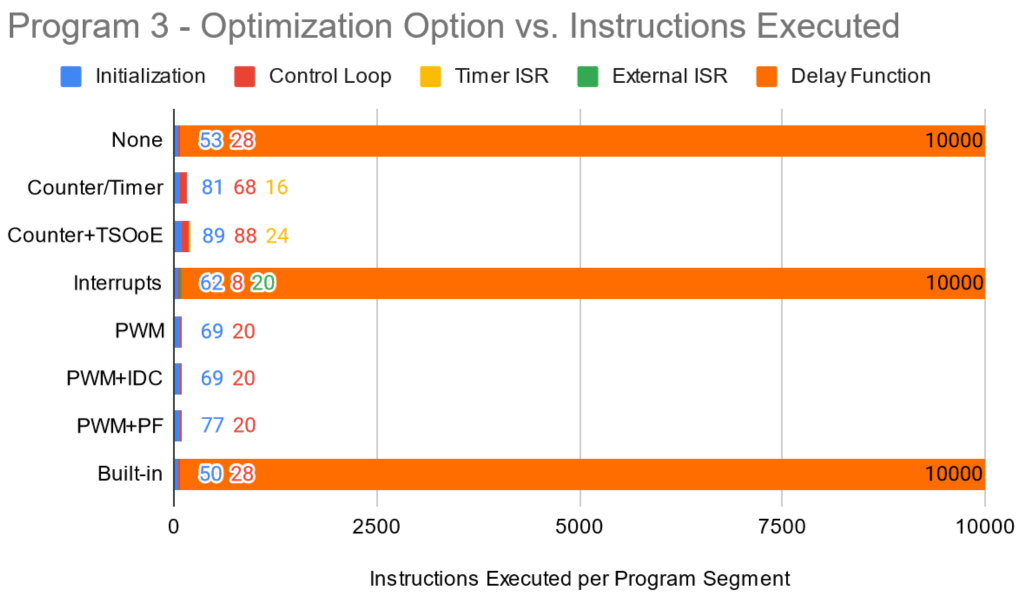


Figure 5.24: A comparison of the approximate number of instructions executed per program segment for Example Program 3 against all applicable optimization options.

ter/Timer" and "Interrupts" optimizations as was shown in Program 1 and 2, with additional overhead needing to be added to the initialization segment, and increased instructions executed within the control loop in the case of "Counter/Timer." The "Interrupts" optimization reduces the size of the control loop substantially due to most of the functionality being moved to a separate ISR. The "PWM" optimization inflates the initialization section of the program somewhat, but in every case it reduces the number of instructions executed within the control loop by about 30% over the unoptimized version. Both with and without the "IDC" option, the "PWM" optimization shows the same control loop segment size, and the "PF" option adds slightly to the initialization segment due to the additional constraint of preserving the frequency from the original version. The "Built-in" optimization reduces the number of instructions executed during the initialization segment slightly due to replacing explicit assignments of SREG with a function call corresponding to a single instruction. The remaining sections remain unchanged as this optimization only applies to the initialization segment in this case.

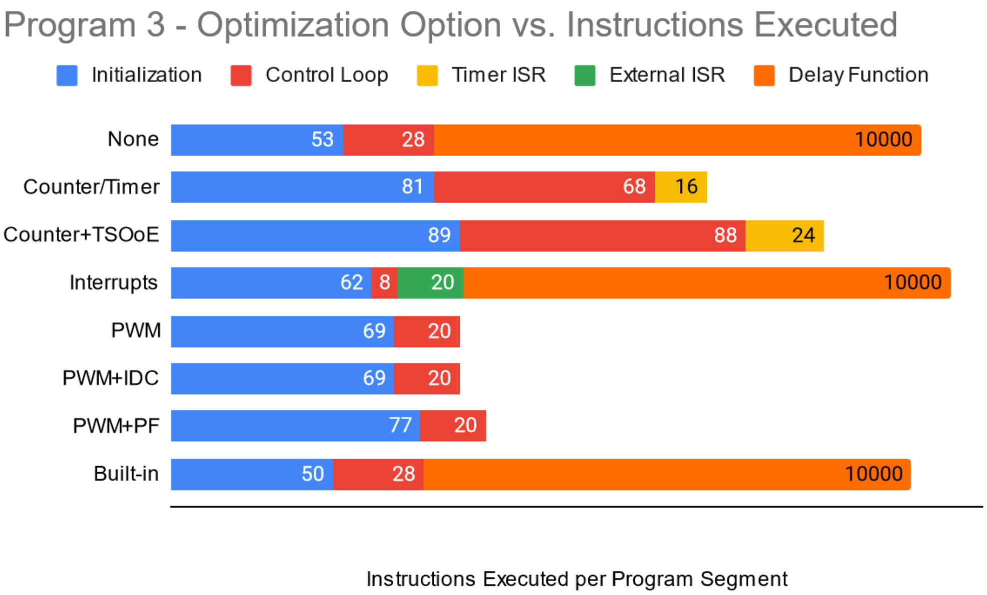


Figure 5.25: A scaled representation of Figure 5.24 to better show the differences between all program segments. The ”Delay Function” segment is largely underrepresented in this figure as a result.

Code Size

The code size of Example Program 3 as it relates to the optimization options can be seen in Figure 5.26. The ”Counter/Timer” optimization reduces the code size by about one half, and the ”Interrupts” optimization increases the final size slightly. Both of these results are consistent with the previous two programs. The ”PWM” optimization shows the best case code size for Program 3, shrinking the final size by approximately 75%. The ”IDC” option does not affect the size significantly, and the ”PF” option increases the size slightly over the default due to the additional logic added for preserving the original signal’s frequency. The ”Built-in” optimization reduces the size slightly as all the applicable assignments of SREG are replaced with a single instruction. The ”Arithmetic” optimization did not impact the program in this instance, and is therefore grayed out.

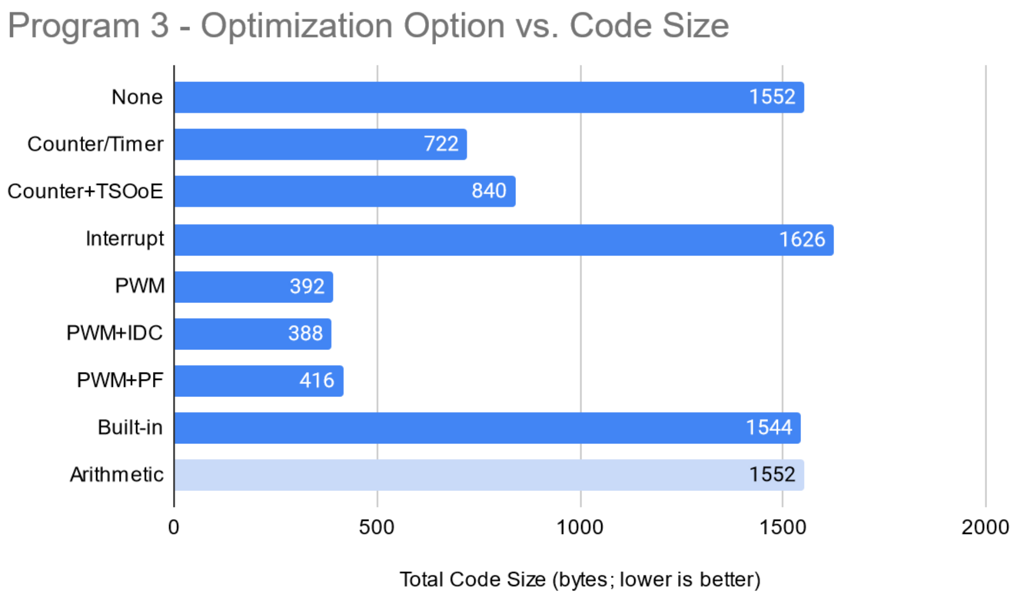


Figure 5.26: A comparison of the size of Example Program 3 after being compiled with each optimization option applied. Optimizations which are not applicable to the program are grayed out.

Cyclomatic Complexity

Similar to the previous programs, the cyclomatic complexity of Example Program 3 with various optimization options applied (shown in Figure 5.27) follows a similar pattern. The ”Counter/Timer” optimization increases the cyclomatic complexity over the unoptimized version by a factor of ~2.3, and the ”TSOoE” option increases that to ~3 times that of the original. The remaining optimization options do not impact the cyclomatic complexity. The only optimization option which could not be applied is ”Arithmetic” and is therefore grayed out.

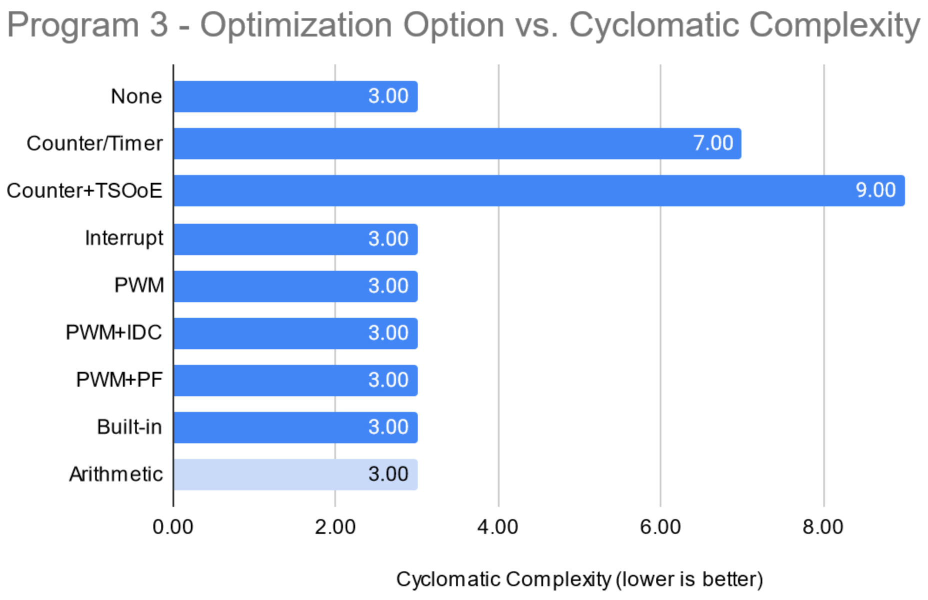


Figure 5.27: A comparison of cyclomatic complexity of Example Program 3 with each optimization option applied. Optimizations which are not applicable to the program are grayed out.

5.2.4 Alternate Example Programs

The following section details findings from performing the same testing as with the initial three programs on externally sourced programs. This is done in an effort to increase the sample size and further validate the results shown by the previous three examples. Three of the externally sourced programs (which are shown in Figures: B.10, B.11, and B.12.) will be tested at the same level of detail as the previous three example programs, and a further nine externally sourced programs (shown in Figures B.10 - B.21) will be tested at a higher level to further evaluate how applicable and generalizable the optimization options provided by the Embedded C Source Optimizer are.

These programs were sourced from various forums, tutorials, and repositories. Each program is relatively simple in nature, and represent real-world examples of programs written both by and for novice embedded software developers.

Latency

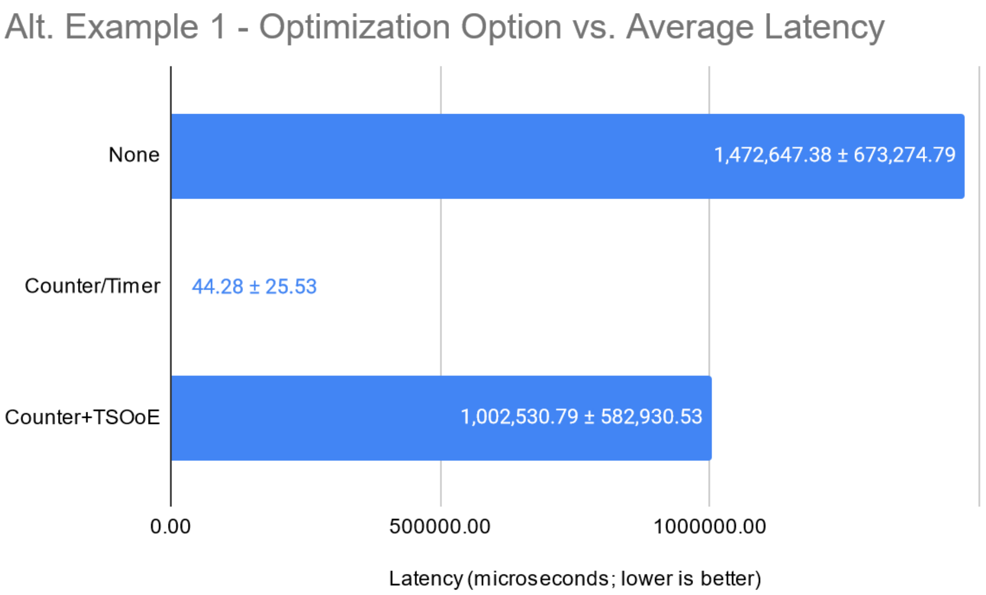


Figure 5.28: Results for average latency tested against each applicable optimization option for Alternate Example Program 1.

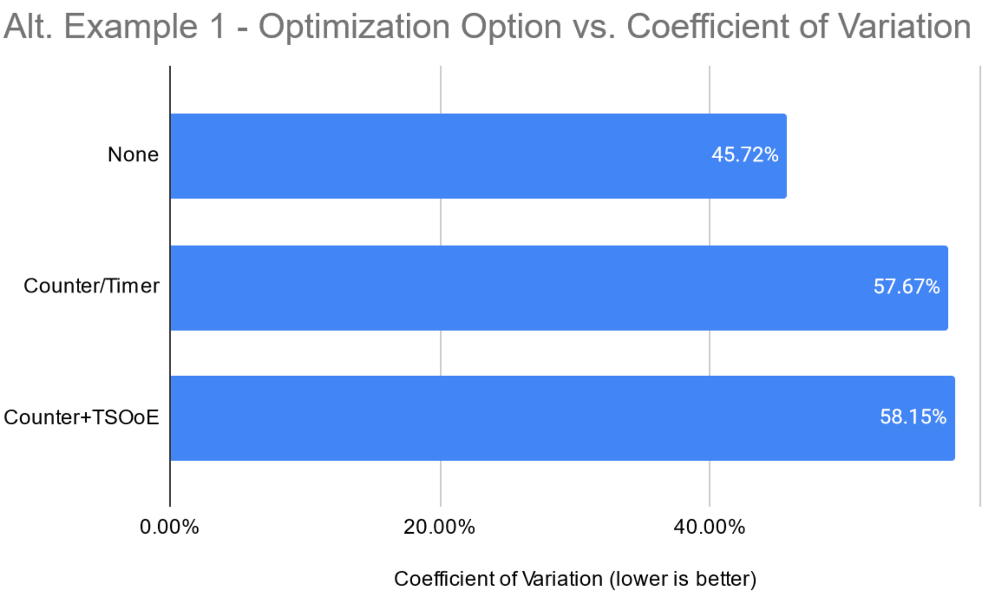


Figure 5.29: Results for the coefficient of variation of the collected latency data for Alternate Example Program 1.

Figures 5.28 and 5.29 show the resulting latency and CV data across the applicable optimization options for Alternate Example Program 1. The ”Counter/Timer” optimization, both with and without ”TSOoE” enabled, shows similar behavior to the results from Example Programs 1-3, where the latency is significantly reduced in the former case, and mostly unchanged in the latter. The CV increased for both optimized implementations of Alternate Example Program 1, indicating slightly less consistent latency over the samples collected.

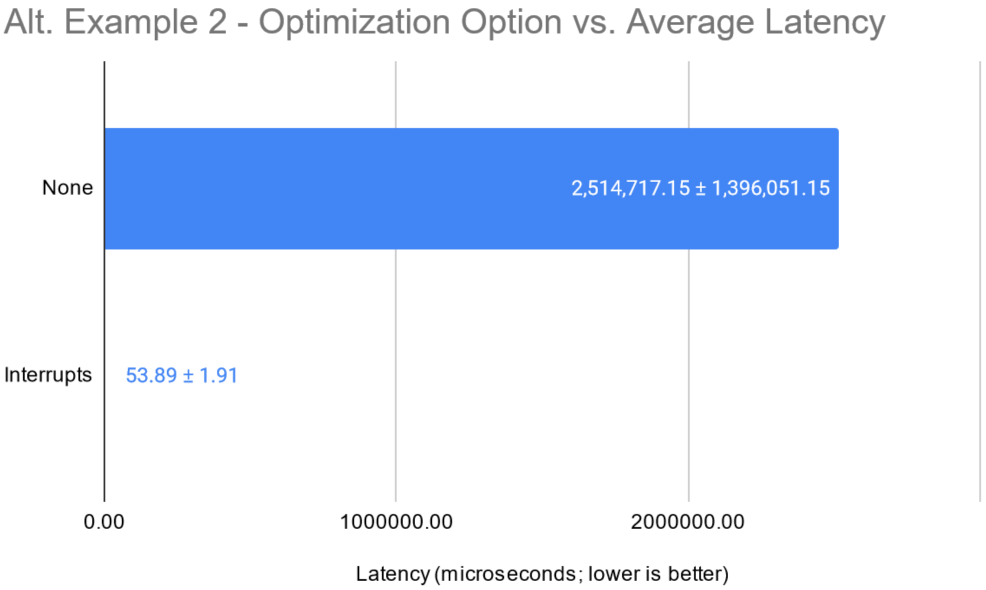


Figure 5.30: Results for average latency tested against each applicable optimization option for Alternate Example Program 2.

Figures 5.30 and 5.31 show both a significant reduction in average latency and CV for the interrupts optimization as applied to Alternate Example Program 2, which is also consistent with the results from earlier tests.

Figures 5.32 and 5.33 (corresponding to Alternate Example Program 3) yields similar results for the ”Counter/Timer” optimization in terms of average latency, although produces significantly higher CV. The relative difference in CV for the ”Counter/Timer” optimization is not immediately obvious, considering the resulting CV with the ”TSOoE” option enabled is in line with the unoptimized implementation. The ”Arithmetic Substitution” optimization

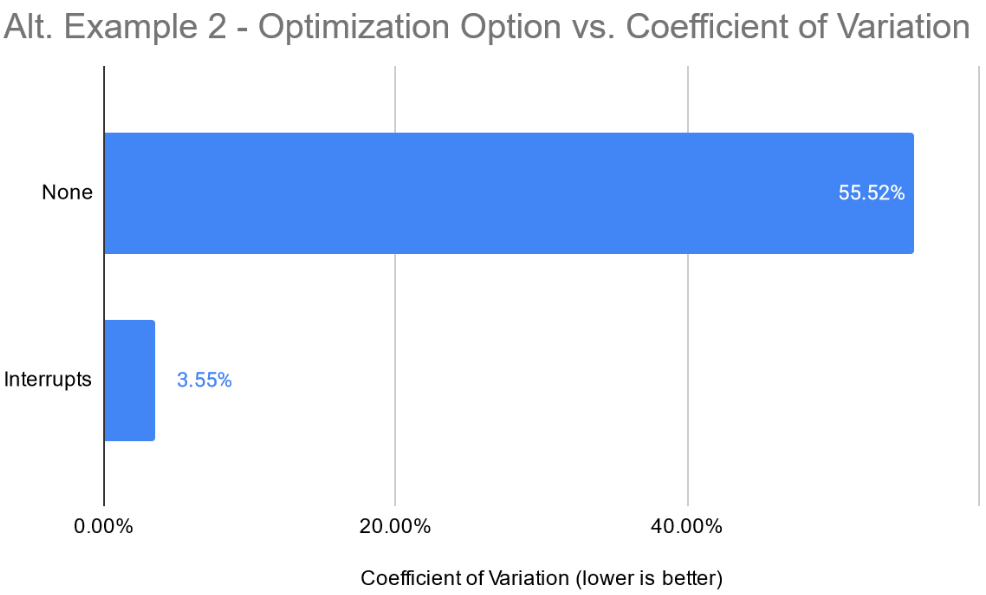


Figure 5.31: Results for the coefficient of variation of the collected latency data for Alternate Example Program 2.

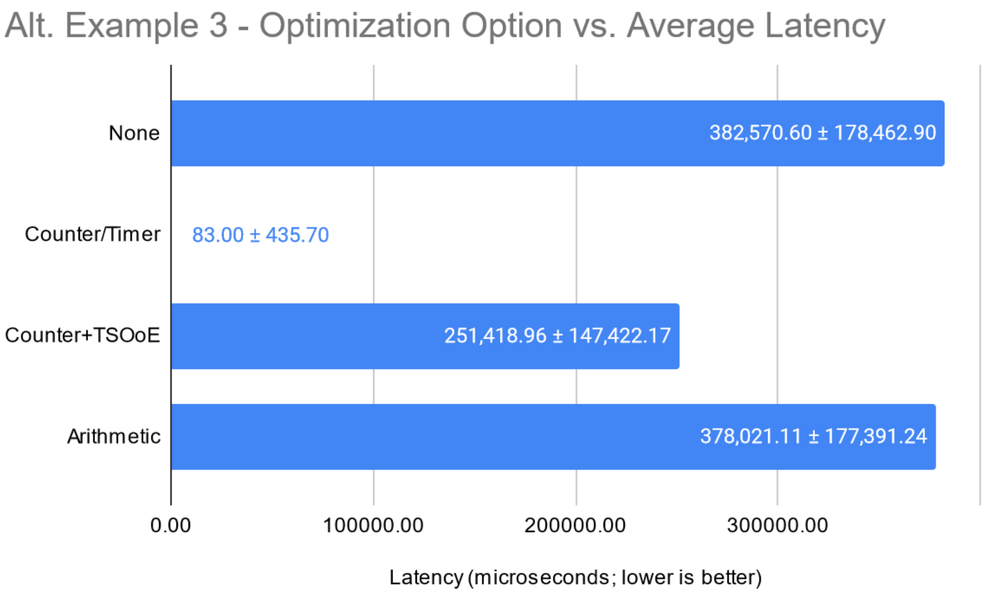


Figure 5.32: Results for average latency tested against each applicable optimization option for Alternate Example Program 3.

was applicable to Alternate Example Program 3, and demonstrated a marginal reduction in average latency (approximately one percent) and no significant difference in terms of CV.

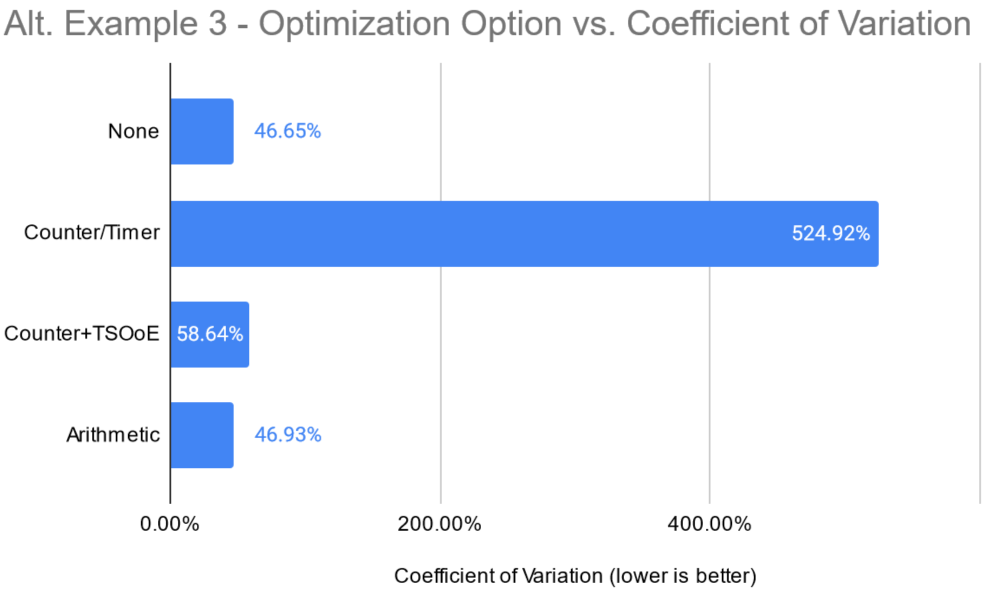


Figure 5.33: Results for the coefficient of variation of the collected latency data for Alternate Example Program 3.

The slight reduction in latency can be associated with fewer instructions generated due to the ”Arithmetic Substitution” optimization replacing the division operation with a right shift operation.

This particular program indicates a limitation with the ”Arithmetic Substitution” optimization, where the data type of any variable being used in the target statement is not considered, and therefore floating point values cause the optimization to introduce a corresponding error into the program. In order to continue testing with this optimization enabled, the program was slightly modified to change the floating point variable to be an integer with the same data width (32 bit). This change must be considered when evaluating this optimization, as changing the data type may have unforeseen influence on the results.

Utilization & Execution Capacity

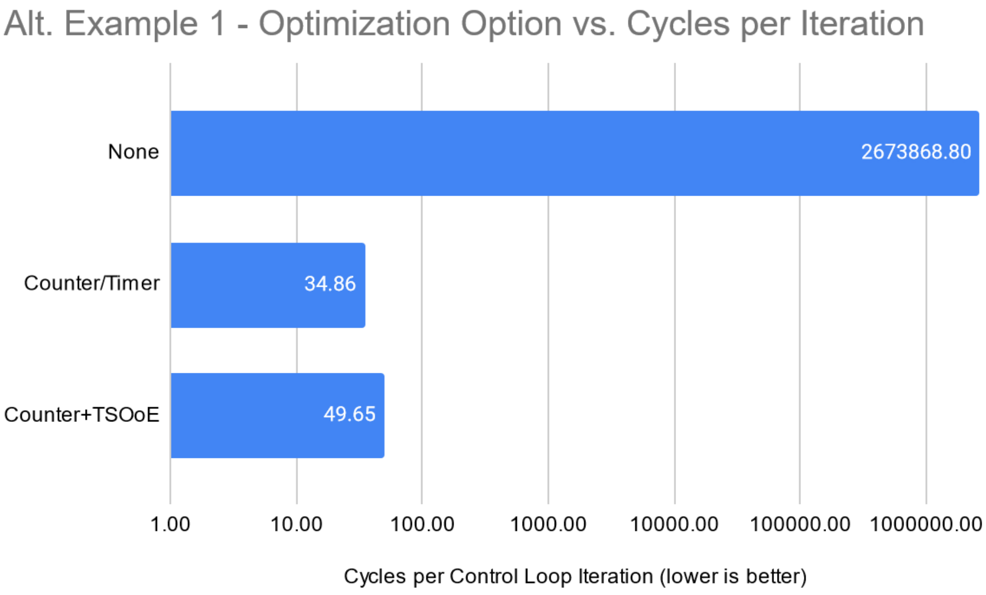


Figure 5.34: The measured cycles per iteration for each applicable optimization option for Alternate Example Program 1.

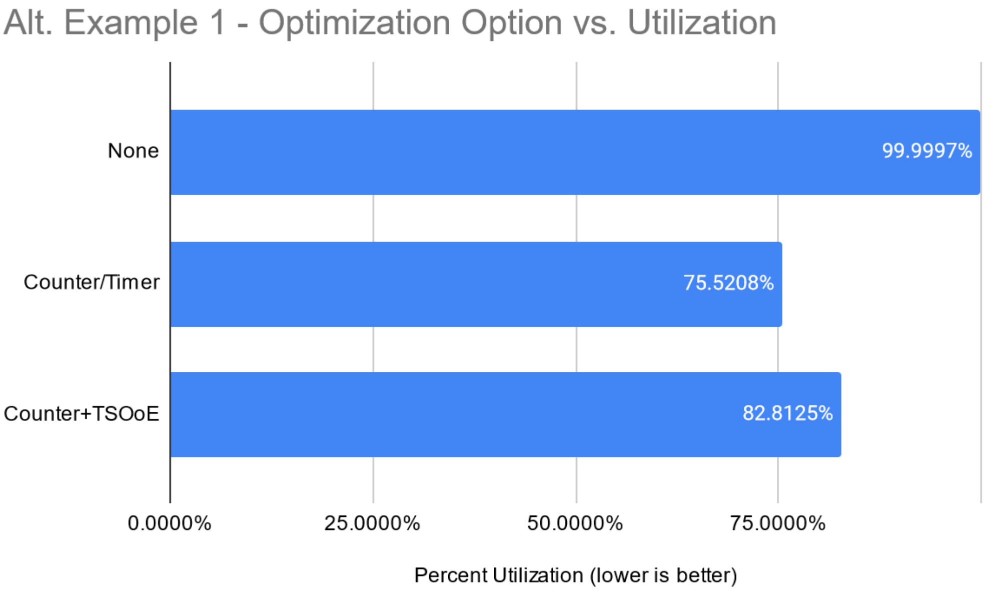


Figure 5.35: The measured utilization for each applicable optimization option for Alternate Example Program 1.

Figures 5.34 and 5.35 demonstrate the expected results for both cycles per iteration and utilization for the applicable optimization options as applied to Alternate Example Program 1. The ”Counter/Timer” optimization significantly reduced the cycles per iteration over the unoptimized implementation, and similarly reduced utilization.

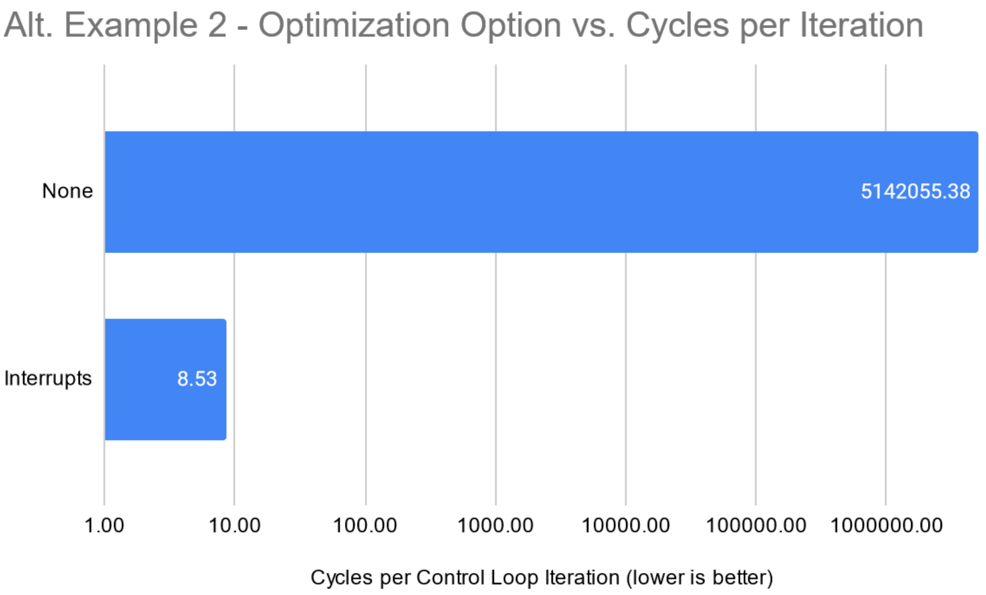


Figure 5.36: The measured cycles per iteration for each applicable optimization option for Alternate Example Program 2.

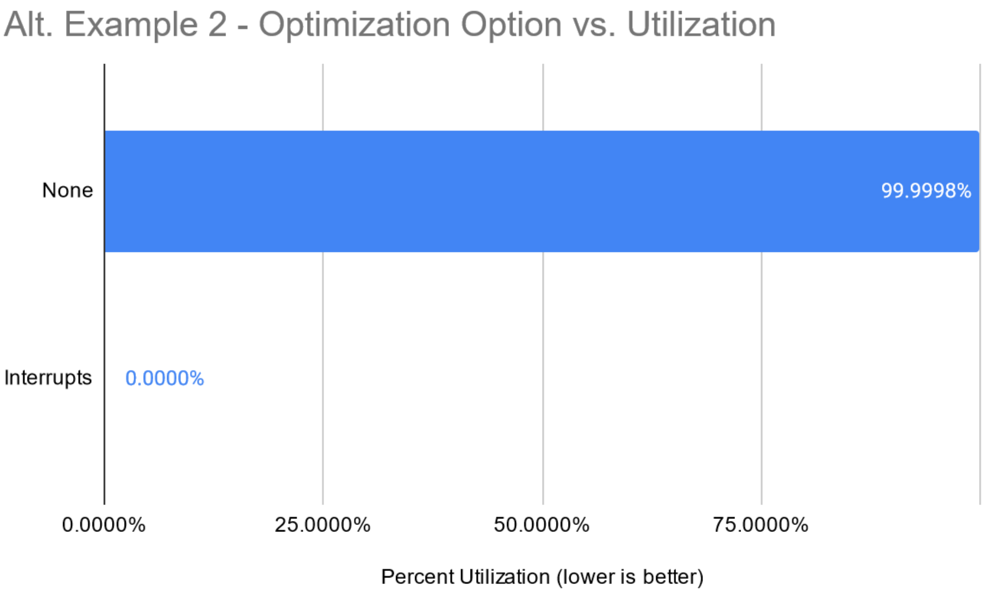


Figure 5.37: The measured utilization for each applicable optimization option for Alternate Example Program 2.

Figures 5.36 and 5.37 also show the expected results for cycles per iteration and utilization when applied to Alternate Example Program 2 for the ”Interrupts” optimization. In this instance, the contents of the control loop was entirely moved to a separate ISR, and as such the cycles per iteration was reduced to the minimum value which can be measured by this test. This results in a utilization of zero percent.

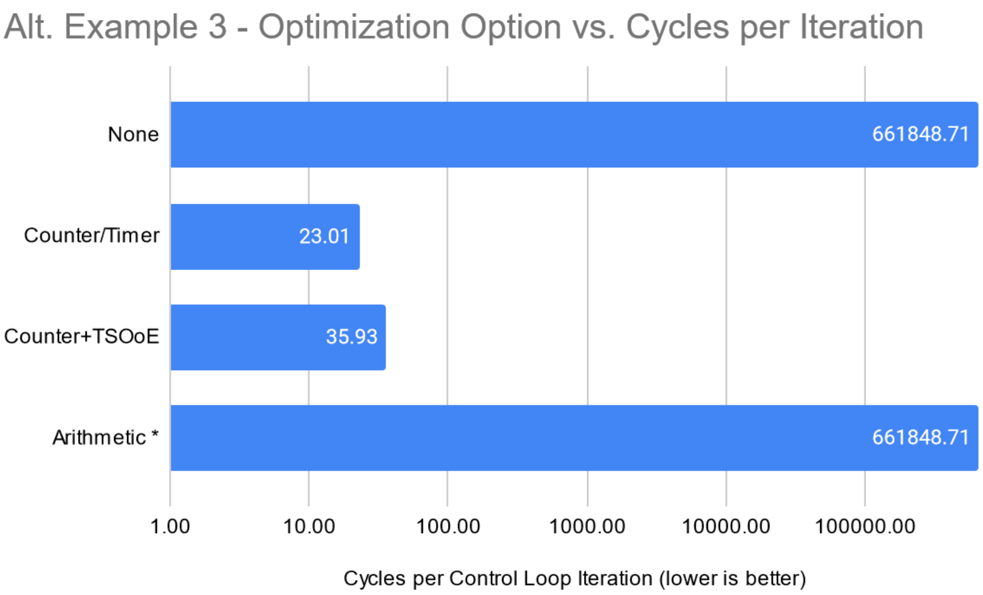


Figure 5.38: The measured cycles per iteration for each applicable optimization option for Alternate Example Program 3.

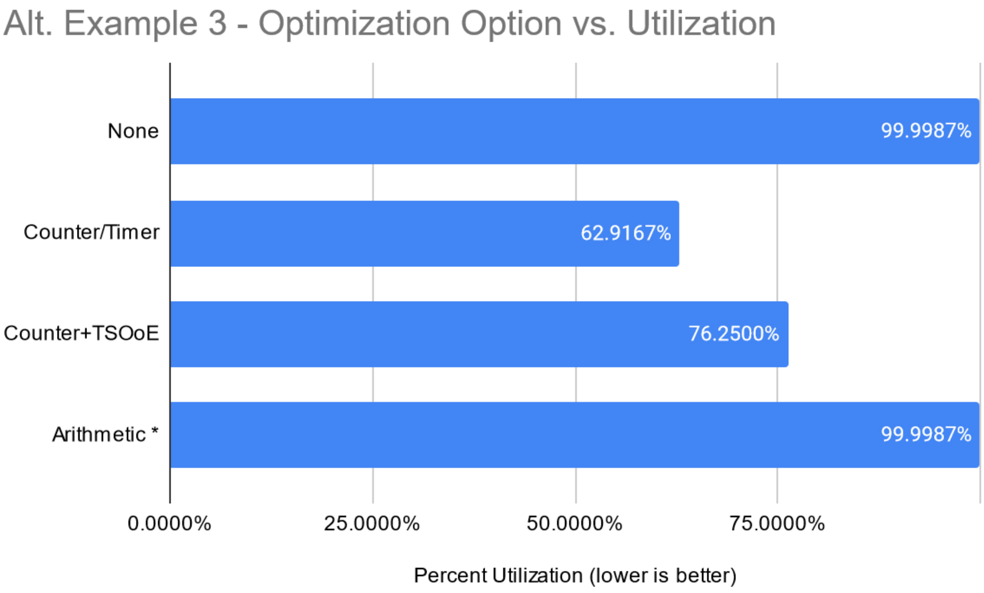


Figure 5.39: The measured utilization for each applicable optimization option for Alternate Example Program 3.

Figures 5.38 and 5.39 show expected results for the ”Counter/Timer” optimization, having reduced the cycles per iteration and utilization of Alternate Example Program 3 by about four orders of magnitude over the unoptimized implementation (both with and without ”TSOoE” enabled). The ”Arithmetic Substitution” optimization had no significant impact on utilization in this case, indicating any difference in utilization is outside of the range of precision provided by this test.

Extensibility

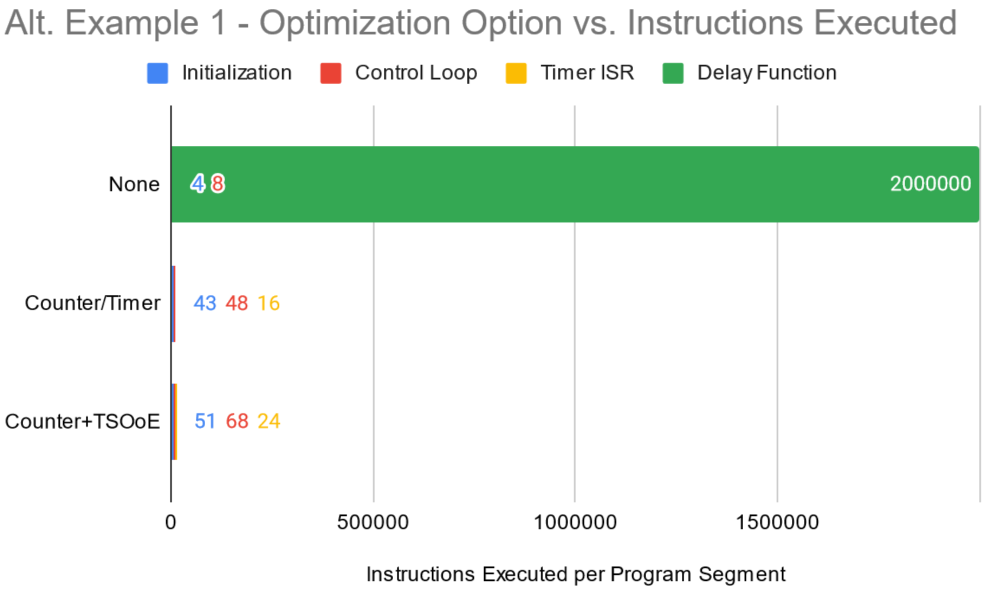


Figure 5.40: The measured extensibility (based on the number of instructions executed per program segment) for each applicable optimization option for Alternate Example Program 1.

Figure 5.40 shows a significant reduction in the total number of instructions executed for the ”Counter/Timer” optimization (both with and without ”TSOoE” enabled) as compared to the unoptimized implementation of Alternate Example Program 1. Similar to the earlier example programs, the Initialization and Control Loop segments increase as a result of the optimization, but eliminate the Delay Function segment, thus reducing the overall instructions executed by the control loop.

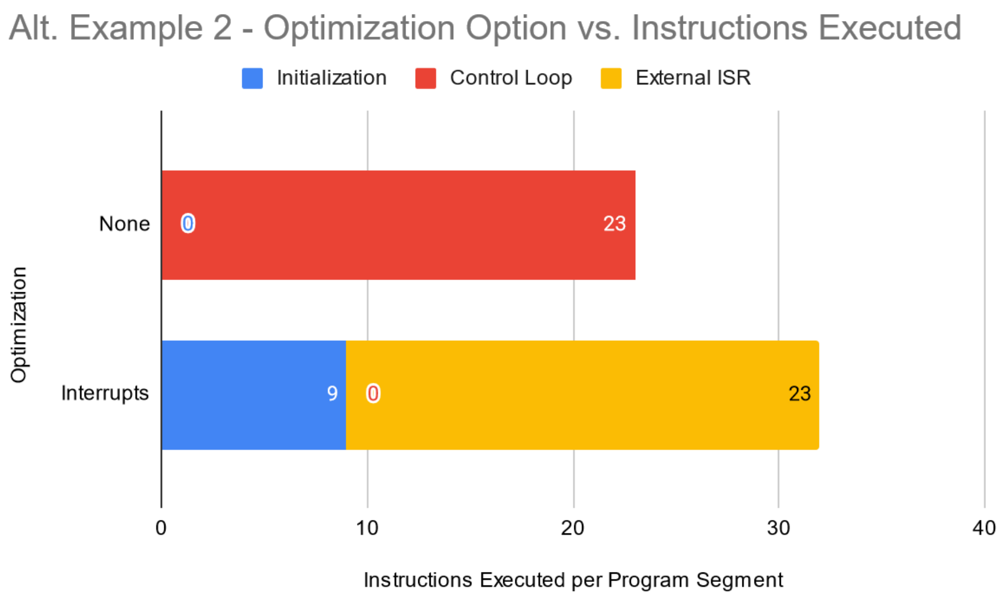


Figure 5.41: The measured extensibility (based on the number of instructions executed per program segment) for each applicable optimization option for Alternate Example Program 2.

Figure 5.41 shows an overall increase in the number of instructions executed by the ”Interrupts” optimization as applied to Alternate Example Program 2. Even though the overall instructions increases (due to the overhead added by configuring hardware interrupts on the pin being checked) it reduces the size of the Control Loop segment from approximately 23 instructions to zero. This is due to the entire contents of the control loop being moved to a separate ISR (which is shown by the External ISR segment). In return, it becomes more straightforward to add functionality to the control loop, as there is no concern of interfering with the check to the external pin as it can be controlled directly by the interrupt handler.

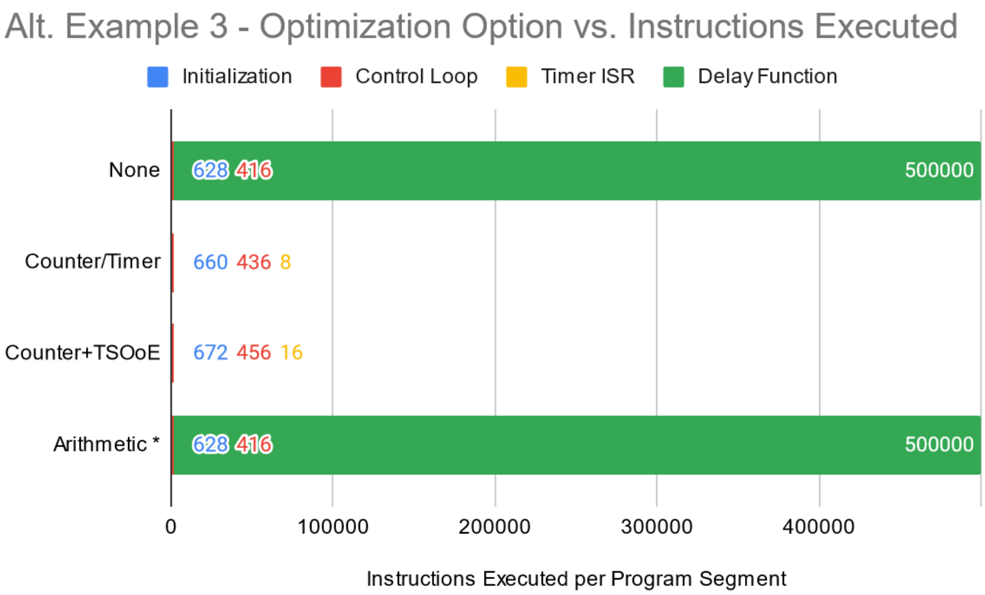


Figure 5.42: The measured extensibility (based on the number of instructions executed per program segment) for each applicable optimization option for Alternate Example Program 3.

Figure 5.42 shows the impact of each applicable optimization option on Alternate Example Program 3 as it relates to altering the size of each of its program segments. The ”Counter/Timer” optimization (with and without ”TSOoE”) show similar results to all previous example programs, having greatly reduced the overall number of instructions executed by the program. The ”Arithmetic Substitution” optimization shows no difference from the unoptimized program, as this testing only approximates the number of instructions generated by the source code and does not account for different arithmetic operations.

Code Size

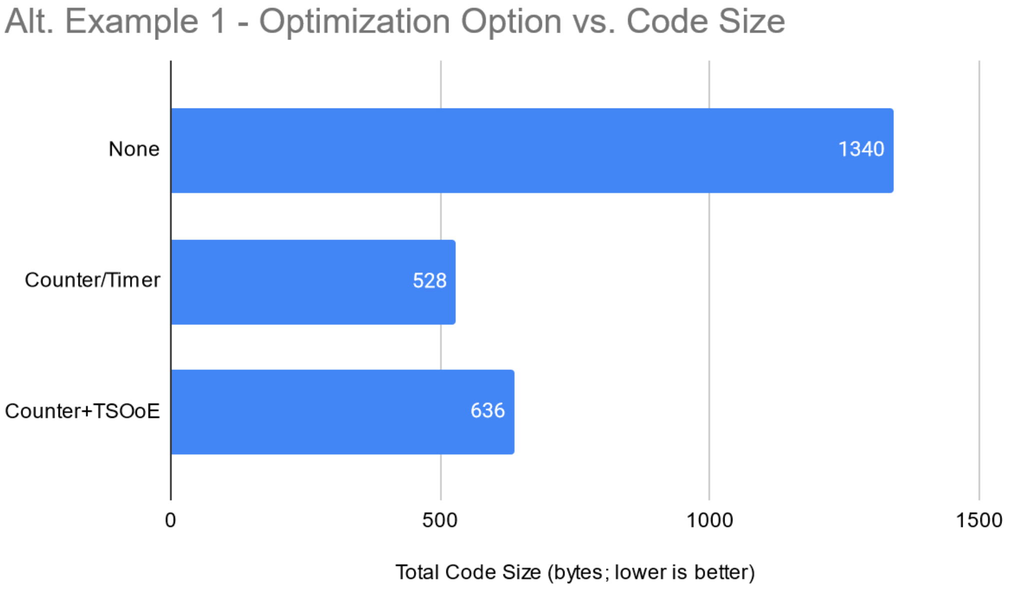


Figure 5.43: The measured code size for each applicable optimization option for Alternate Example Program 1.

Figure 5.43 demonstrates the results of applicable optimizations to Alternate Example Program 1 in terms of code size. The ”Counter/Timer” optimization with and without ”TSOoE” enabled shows the expected results, shrinking the final code size by about half. This again is likely a result of not including the delay function library in the compiled firmware.

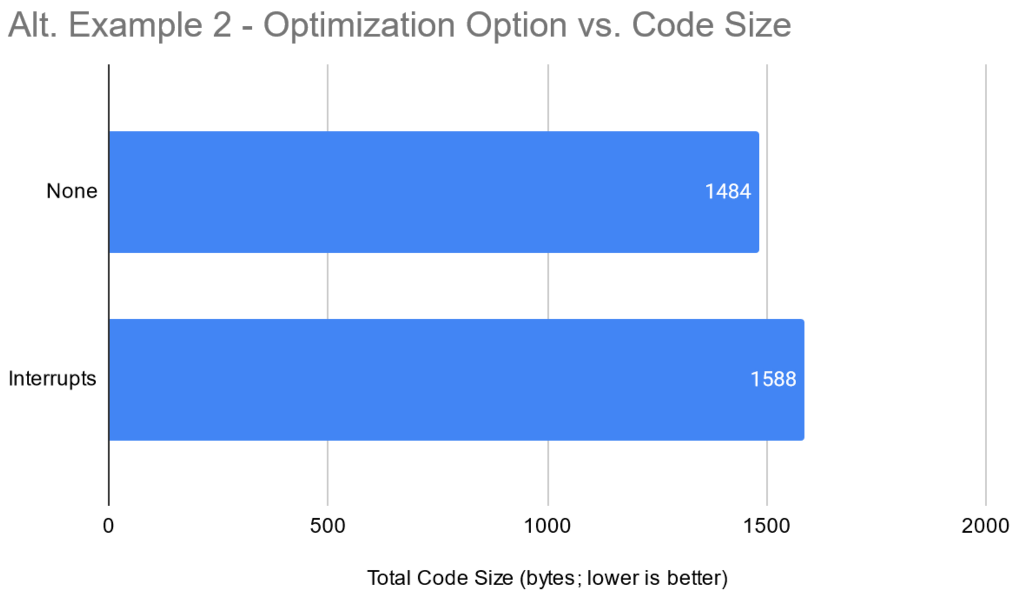


Figure 5.44: The measured code size for each applicable optimization option for Alternate Example Program 2.

Figure 5.44 shows similar results to previous programs with the ”Interrupts” optimization applied, increasing the final code size of Alternate Example Program 2 by seven percent over the unoptimized implementation. This is consistent with other examples of this optimization being applied, and is due to the overhead added to configure the interrupt handler.

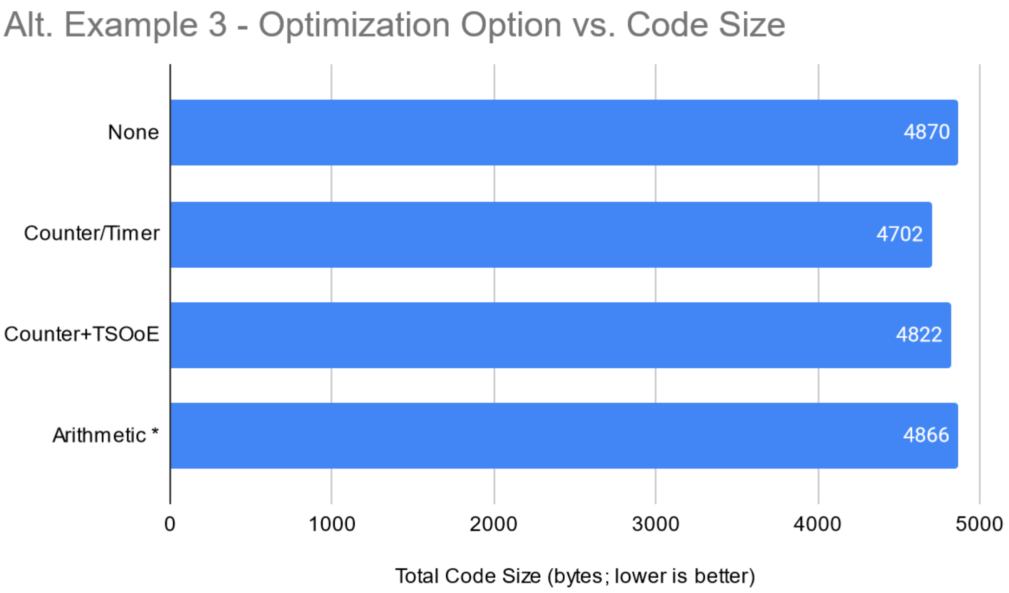


Figure 5.45: The measured code size for each applicable optimization option for Alternate Example Program 3.

Figure 5.45 shows a marginal decrease in the final code size of Alternate Example Program 3 for the ”Counter/Timer” optimization both with and without ”TSOoE” enabled. The ”Arithmetic Substitution” optimization also reduces the final code size of the program, but by a negligible amount. The small relative difference in code size of the optimized implementations of Alternate Example Program 3 is partially a result of the significantly larger initial size of the program as compared to previous example programs. It must also be considered that the program had to be modified slightly to apply the ”Arithmetic Substitution” optimization, which may have an impact on the final code size other than that of the transformation applied by the optimization.

Cyclomatic Complexity

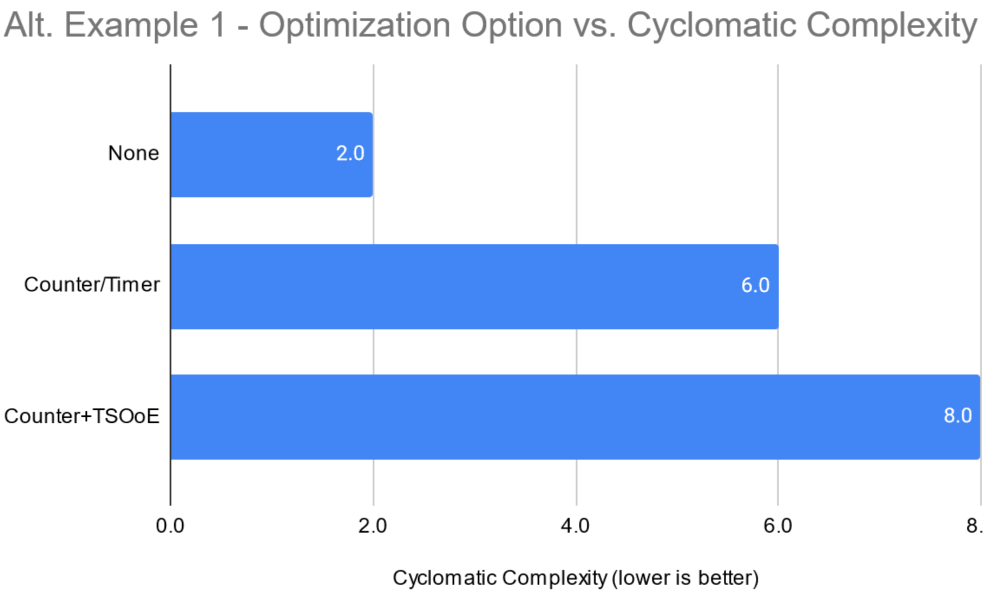


Figure 5.46: The measured cyclomatic complexity for each applicable optimization option for Alternate Example Program 1.

Figure 5.46 shows an increase in cyclomatic complexity when the ”Counter/Timer” optimization is applied to Alternate Example Program 1, and a further increase when ”TSOoE” is enabled. This is again a result of the ”Counter/Timer” adding additional logic to the program to utilize the counter/timer peripheral, thus increasing the number of paths through the program.

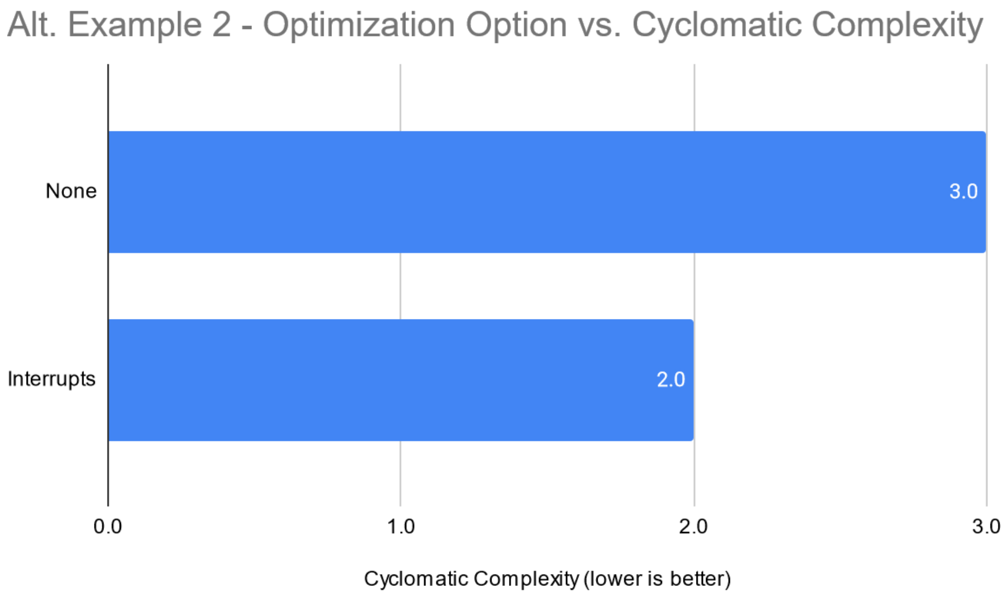


Figure 5.47: The measured cyclomatic complexity for each applicable optimization option for Alternate Example Program 2.

Figure 5.47 shows a decrease in the final cyclomatic complexity of Alternate Example Program 2 when the ”Interrupts” optimization is applied. This may be a result of eliminating the contents of the control loop, and since the added ISR executes asynchronously from the main code is not counted as a path through the program.

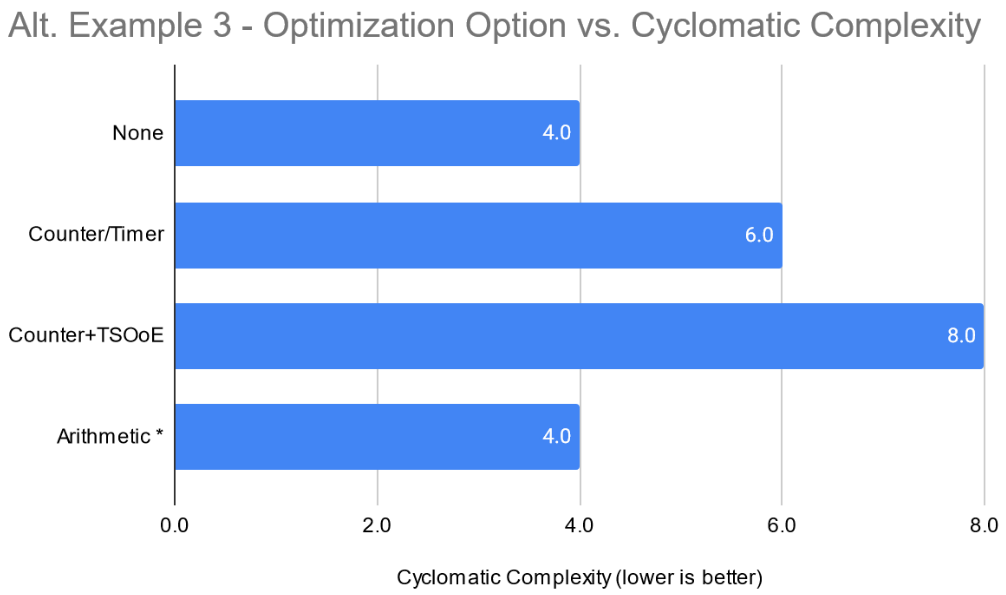


Figure 5.48: The measured cyclomatic complexity for each applicable optimization option for Alternate Example Program 3.

Figure 5.48 shows a similar increase to cyclomatic complexity when the ”Counter/-Timer” optimization is applied to Alternate Example Program 3, and a further increase when the ”TSOoE” option is enabled. The ”Arithmetic Substitution” optimization had no impact on the cyclomatic complexity of this program, which is due to this optimization not influencing the number of paths through the program.

Additional High-Level Tests

An additional nine externally sourced programs were tested at a high level to better determine whether the optimization options provided by the Embedded C Source Optimizer are general enough to apply to a wide range of use cases. This testing was performed by importing each program into the Embedded C Source Optimizer, selecting and applying all optimization options, and exporting the resulting code. The optimized iterations of each program were then evaluated in terms of code size and cyclomatic complexity to better correlate the results with the previous example programs.

Of these nine programs, seven had at least one optimization option successfully applied. The two programs which did not have any optimization applied were not altered by the optimizations in any way. All nine of the programs still compiled and were functionally equivalent after being passed through the Embedded C Source Optimizer.

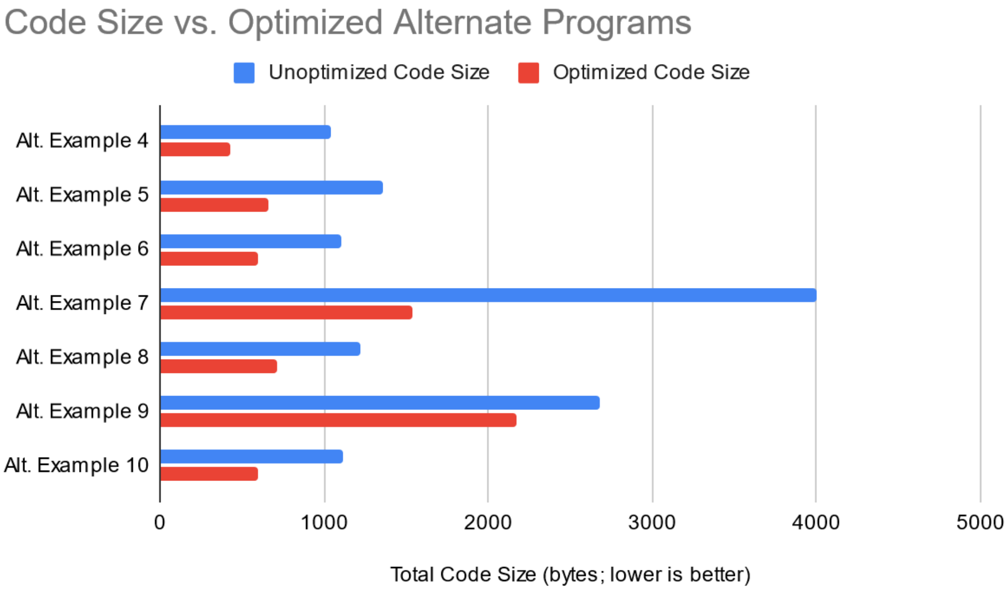


Figure 5.49: Resulting code size for the original and optimized variants (by selecting and applying all optimization options), showing the difference between both implementations.

Figure 5.49 shows the code size of the additional alternate example programs both before and after having any optimizations applied. The seven programs which had at least one optimization applied are shown as Alt. Example 4 - 10. In every instance, the optimizations which were applied reduced the final code size of the program.

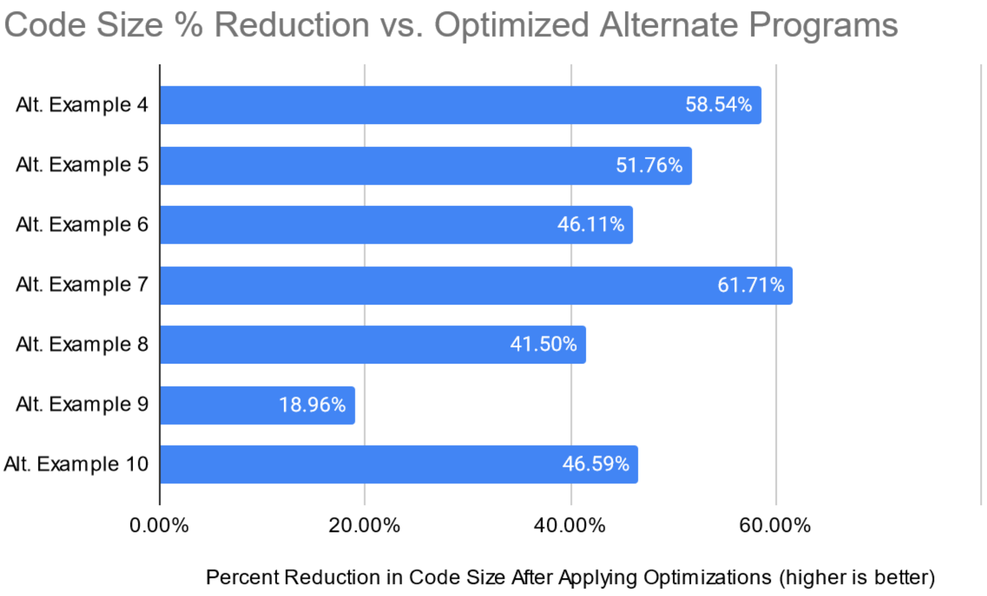


Figure 5.50: Relative percent difference in code size between the unoptimized and optimized variants of each of the additional tested programs.

Figure 5.50 shows the relative difference in code size between the original and optimized versions of Alternate Examples 4 - 10. The differences in code size are relatively consistent, producing a reduction in code size of between 41% and 62%, with one outlier case having only reduced the final size by 19%.

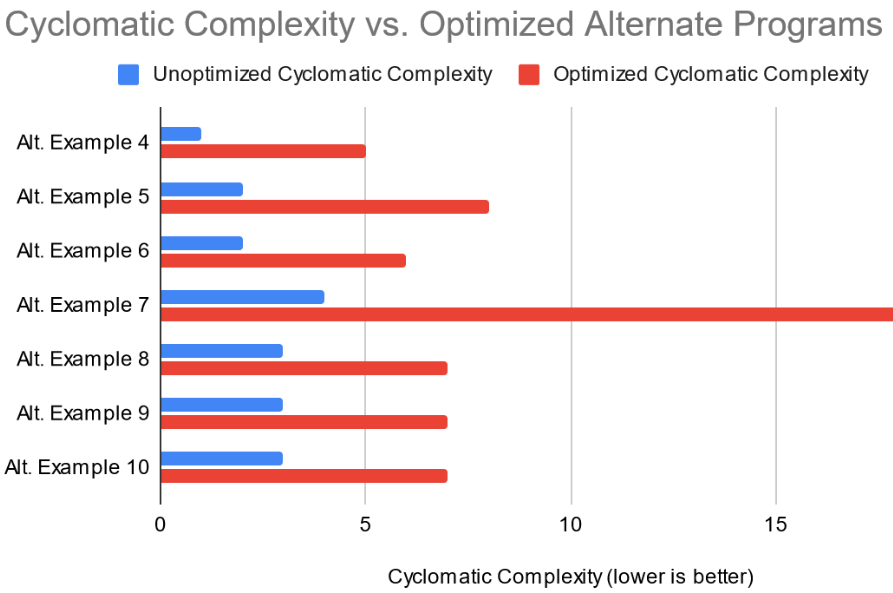


Figure 5.51: Resulting cyclomatic complexity for the original and optimized variants (by selecting and applying all optimization options), showing the difference between both implementations.

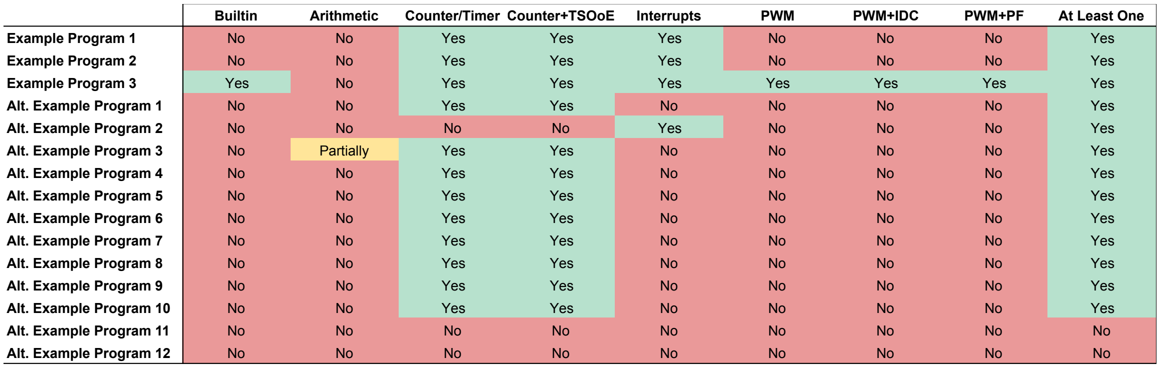
Figure 5.51 shows the cyclomatic complexity for the unoptimized and optimized version of alternate example programs 4 - 10. The cyclomatic complexity increased in each case, which is a result of the ”Counter/Timer” optimization being applicable to each of these cases. The results of previously tested programs show the ”Counter/Timer” optimization consistently increasing the cyclomatic complexity of a program due to the additional logic and conditional statements it adds. The differences in cyclomatic complexity between the unoptimized and optimized implementations vary substantially, as the more delay functions which the target program contains, the more conditional statements will be added to control counter/limit instances for each task.

5.3 Experimental Results

The resulting measurements presented in Section 5.2 demonstrate mostly positive changes to the tested embedded programs. While each optimization option has varying levels of applicability to programs and can have both positive and negative effects on the resulting code, there is evidence to support that these optimizations have a positive overall impact. With one of the main goals of the Embedded C Source Optimizer being to improve embedded code in a variety of ways, the measurements presented in this chapter more or less satisfy this with little downside. Being a voluntary option to apply optimizations by the end-user means that they retain the choice to accept any potential downsides for each optimization option. Furthermore, while not tested directly, applying combinations of optimizations may yield overall better results, producing code which may have the best attributes of each optimization.

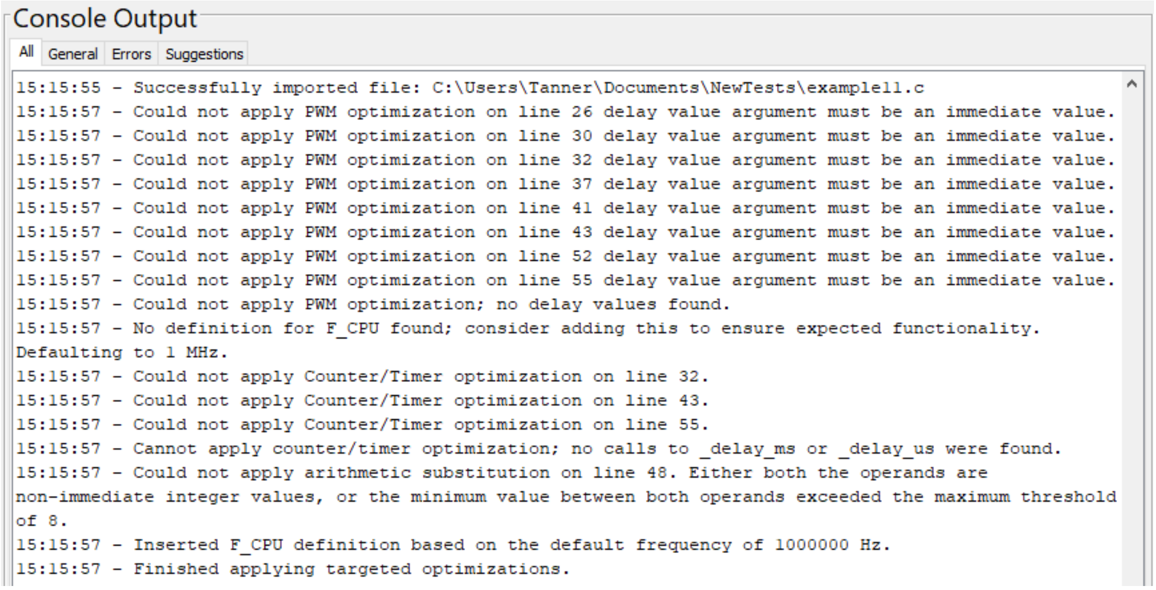
Figure 5.52 shows the applicability of each optimization option provided by the Embedded C Source Optimizer to each of the fifteen tested programs based on the evaluation previously provided Section 5.2. The twelve alternate example programs (which are sourced from various online forums, tutorials, and repositories) show similar levels of applicability to the purposefully designed solutions in terms of how each optimization option was applied. Of all fifteen programs, thirteen had at least one optimization applied. Furthermore, there was only a single case (Alternate Example Program 3) where the final result of running the program through the Embedded C Source Optimizer resulted in a non functional program. The ”Arithmetic Substitution” optimization does not take into account the data type of the values being used in an arithmetic operation, and as such has the capability to render a statement nonfunctional if floating point values are used. Since the ATmega168 does not have a hardware Floating Point Unit (FPU) [6], it is not necessarily expected that this problem will be common.

As can be seen in Figure 5.52, there were two cases in which none of the optimization options could be applied to the target program. These include Alternate Example Program 11 and 12 (which can be seen in Figures B.20 and B.21 respectively). The output from the Embedded C Source Optimizer when attempting to apply all optimizations to these programs is shown in Figures 5.53 and 5.54. The utility indicates to the user how and why any selected optimization options that are not conducive to the program could not be applied.



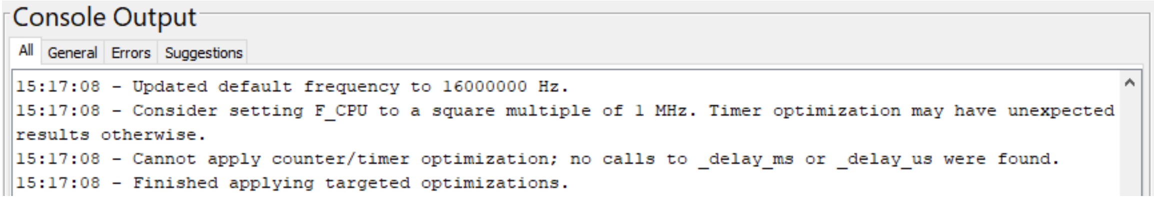
	Builtin	Arithmetic	Counter/Timer	Counter+TSOoE	Interrupts	PWM	PWM+IDC	PWM+PF	At Least One
Example Program 1	No	No	Yes	Yes	Yes	No	No	No	Yes
Example Program 2	No	No	Yes	Yes	Yes	No	No	No	Yes
Example Program 3	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Alt. Example Program 1	No	No	Yes	Yes	No	No	No	No	Yes
Alt. Example Program 2	No	No	No	No	Yes	No	No	No	Yes
Alt. Example Program 3	No	Partially	Yes	Yes	No	No	No	No	Yes
Alt. Example Program 4	No	No	Yes	Yes	No	No	No	No	Yes
Alt. Example Program 5	No	No	Yes	Yes	No	No	No	No	Yes
Alt. Example Program 6	No	No	Yes	Yes	No	No	No	No	Yes
Alt. Example Program 7	No	No	Yes	Yes	No	No	No	No	Yes
Alt. Example Program 8	No	No	Yes	Yes	No	No	No	No	Yes
Alt. Example Program 9	No	No	Yes	Yes	No	No	No	No	Yes
Alt. Example Program 10	No	No	Yes	Yes	No	No	No	No	Yes
Alt. Example Program 11	No	No	No	No	No	No	No	No	No
Alt. Example Program 12	No	No	No	No	No	No	No	No	No

Figure 5.52: The results of applying each optimization option to each of the fifteen tested programs. Thirteen of the fifteen programs had at least one optimization applied (shown in the right-most column).



```
15:15:55 - Successfully imported file: C:\Users\Tanner\Documents\NewTests\example11.c
15:15:57 - Could not apply PWM optimization on line 26 delay value argument must be an immediate value.
15:15:57 - Could not apply PWM optimization on line 30 delay value argument must be an immediate value.
15:15:57 - Could not apply PWM optimization on line 32 delay value argument must be an immediate value.
15:15:57 - Could not apply PWM optimization on line 37 delay value argument must be an immediate value.
15:15:57 - Could not apply PWM optimization on line 41 delay value argument must be an immediate value.
15:15:57 - Could not apply PWM optimization on line 43 delay value argument must be an immediate value.
15:15:57 - Could not apply PWM optimization on line 52 delay value argument must be an immediate value.
15:15:57 - Could not apply PWM optimization on line 55 delay value argument must be an immediate value.
15:15:57 - Could not apply PWM optimization; no delay values found.
15:15:57 - No definition for F_CPU found; consider adding this to ensure expected functionality.
Defaulting to 1 MHz.
15:15:57 - Could not apply Counter/Timer optimization on line 32.
15:15:57 - Could not apply Counter/Timer optimization on line 43.
15:15:57 - Could not apply Counter/Timer optimization on line 55.
15:15:57 - Cannot apply counter/timer optimization; no calls to _delay_ms or _delay_us were found.
15:15:57 - Could not apply arithmetic substitution on line 48. Either both the operands are
non-immediate integer values, or the minimum value between both operands exceeded the maximum threshold
of 8.
15:15:57 - Inserted F_CPU definition based on the default frequency of 1000000 Hz.
15:15:57 - Finished applying targeted optimizations.
```

Figure 5.53: The output generated when optimized alternate example program 11, showing the selected optimization options indicating (through log output) the reasons which the optimization could not be applied to the target program.



```
15:17:08 - Updated default frequency to 16000000 Hz.
15:17:08 - Consider setting F_CPU to a square multiple of 1 MHz. Timer optimization may have unexpected
results otherwise.
15:17:08 - Cannot apply counter/timer optimization; no calls to _delay_ms or _delay_us were found.
15:17:08 - Finished applying targeted optimizations.
```

Figure 5.54: The output generated when optimized alternate example program 12, showing the selected optimization options indicating (through log output) the reasons which the optimization could not be applied to the target program.

It should be noted that each optimization option targets a different use case, and therefore has varying levels of usefulness. For instance, the ”Counter/Timer” optimization intends to target a wider range of use cases, but in doing so may not always provide the most optimal results as compared to other options. In the case of Example Program 3 (which was discussed in detail in Subsection 5.2.3) the ”PWM” optimization option was more effective than the ”Counter/Timer” optimization in nearly every metric that was tested. That being said, the ”PWM” optimization targets a more specific use case, and was not applicable to Example Program 1 or 2, whereas the ”Counter/Timer” optimization was successfully applied to all three programs. These trade offs are intentional, as they each fulfill a different purpose and attempt to convey a different concept to the end-user. It is therefore up to the user to determine which optimization are most applicable to their program.

Chapter 6

CONCLUSION

A summary of the the research topic and how it is addressed is provided in Section 6.1 and the research questions (previously introduced in Section 3.2) will be answered in Section 6.2. Furthermore, the outlook of the Embedded C Source Optimizer will be addressed in Section 6.3, taking into consideration external constraints on the development of the Embedded C Source Optimizer and what potential it may have given additional development time.

6.1 Summary

The proposed Embedded C Source Optimizer is intended to act both as an educational tool in teaching embedded design principles to novice embedded software developers, as well as a more general utility which can improve a variety of aspects associated with embedded code. Provided the differences in considerations between developing traditional and embedded software, having access to such a tool may be beneficial to novice embedded software developers. As demonstrated by the results provided in Chapter 5, the Embedded C Source Optimizer is capable of improving many aspects of embedded code, as well as conveying how each improvement was made through documentation included in the generated code and log output of the utility.

6.2 Answering Research Questions

1. *How can embedded code be automatically altered/modified by a utility at the source code level in order to improve its resulting size, efficiency, and performance, while also preserving the originally-intended functionality?*

   One of the primary ways in which software can be automatically transformed to improve its size, performance, and efficiency (in terms of utilization) is through the removal of blocking functions which would otherwise reduce the throughput and increase the latency of a program. The results provided in Section 5.2 show multiple cases where removing blocking calls (specifically the blocking delay function calls included in most of the example programs) significantly reduces the average latency, utilization, and resulting code size while maintaining the same functionality as the original program. Another way which embedded programs can be improved is through the utilization of included hardware peripherals. The ”Counter/Timer,” ”Interrupts,” and ”PWM” optimization options provided by the Embedded C Source Optimizer apply transformations to software which utilize each respective peripheral they target. For the sake of example, programs where the ”Interrupts” optimization could be applied (shown previously in Section 5.2) show a significant decrease in average latency without affecting the functionality of the program. Furthermore, the ”Counter/Timer” and ”PWM” optimizations both target the counter/timer peripheral, which is ultimately how the blocking delay function calls can be eliminated from the program.

2. *How can existing software be mapped to fixed-function hardware in order to improve embedded programs?*

   Structures in existing software may be mapped/transformed to utilize hardware peripherals through matching source code constructs (e.g. conditional statements and function calls) and substituting them for functionally-equivalent code which maps to a given peripheral. In the case of the ”Counter/Timer” optimization, this is accomplished by searching for calls to blocking software delay functions within the control loop of a program, and from there the delay value passed to that function can be extracted and mapped to configure a series of counter/limit instances which ultimately control the execution of the code surrounding those delay function calls. These blocking delay function calls inflate the utilization and decrease the throughput of a given program, and removing them generally provides an uplift in this regard (shown previously in Section 5.2). An extension of utilizing the counter/timer peripheral is implemented by the ”PWM” optimization, where only cases in which an external pin is toggled with delays placed in between will be mapped to a PWM channel. This optimization provides a performance uplift through the same method as the ”Counter/Timer” optimization as it also removes calls to blocking software delay functions. The ”Interrupts” optimization targets conditional statements which check the state of an external pin; since every pin on the ATmega168 can be configured to trigger interrupts on a state change, then by configuring and mapping the existing code to only be executed on a state change, the final functionality may be preserved. By moving the conditional statement to an ISR and relying on the interrupt handler to control the execution of the program, the overhead of performing the check in software is removed, both increasing throughput and decreasing latency. The improvements made by each optimization option were demonstrated previously in Section 5.2.

The various conditions which must be met to apply optimizations are made to be quite specific, such that the final functionality of the program may be maintained with a high level of confidence. In the case of the ”direct replacement” optimizations (mentioned previously in Subsection 4.2.2), code can be mapped more simply by checking for specific cases of register assignments and arithmetic operations. For instance, the ”Built-in Function Substitution” optimization is applied only when SREG is being assigned directly, and only when the state of the ”Global Interrupt Enable” bit is being toggled. Furthermore, the ”Arithmetic Substitution” optimization matches cases where multiplication and division are used in a specific way, where functionally equivalent code can be inserted. Both the ”Built-in Function Substitution” and ”Arithmetic Substitution” optimizations attempt to improve performance through generating fewer instructions, which in turn decreases code size and may increase the throughput of the target program (refer to Chapter 5 for details).

3. *How could automatically applied source-level optimizations aid a user's ability to comprehend and extend upon the resulting program, as well as be conveyed in a way which illustrates how the optimizations work?*

   As previously demonstrated in Section 4.3, the Embedded C Source Optimizer supports multiple methods of conveying how each optimization works, any considerations which should be made when using a given optimization, and comments in the generated output which indicate what actions any added code performs. Furthermore, the Embedded C Source Optimizer log output provides additional suggestions to the user for operations which a given optimization option does not directly target. All of these attributes may improve the end-user's ability to work with and understand the optimizations provided by the utility. It should be noted the extent to which these attributes aid the end-user's understanding were not directly evaluated.

6.3 Outlook

The Embedded C Source Optimizer, while considered feature complete for the current iteration, is intended to have the capacity for expansion and further iteration thanks to its modular architecture and use of well-documented design patterns (which were discussed previously in Subsection 4.1.3). Some high-level improvements which could be made include: new optimization options, new features, wider hardware support, more granular source code analysis, and UI improvements.

Assuming additional development time, some of the highest priority additions/changes which would be made include:

- Updating the syntax handling for imported code to enforce that the format of the original code is preserved, as well as ensuring that code formatted in a variety of ways may be properly parsed.

- Updating existing optimization options to target a wider range of use cases.
  - The ”Counter/Timer” optimization requires delay functions to occur within the control loop of a program and have immediate values for arguments. Ideally it would work when delay function calls are nested in other functions/constructs, and when the delay argument is a non-immediate value.
  - The ”Interrupts” optimization ideally should be able to recognize a wider range of cases when checking for an external pin change; more specifically: when user-defined macros are used which map to registers/pins, identifying different formats for bit masks which target the checked pin, and checking for target conditional statements outside of the control loop of the program.
  - The ”Arithmetic Substitution” and ”Built-in Function Substitution” optimizations could be expanded to include additional targets (more built-in functions

and additional arithmetic operations) which would further broaden the use-case for each optimization, as well as potentially making each more effective.

- Widening hardware support for more MCU families and architectures in order to broaden the set of potential use cases.

- Adding new optimization options which target different peripherals to increase the likelihood that at least one optimization is applicable to the majority of embedded C programs.

- Adding support for importing multiple C source and header files so larger projects may be optimized.

- Including additional automated testing (besides code size) to better convey the changes made by each optimization option, as well as potentially selecting the optimal set of optimization options for a given program based on said testing.

Being an open source project, anyone is able to use, modify, and/or distribute the source code of the Embedded C Source Optimizer. This decision was made to facilitate the goal of providing an all-encompassing tool to novice embedded software developers to learn about and experiment with embedded code, and to improve existing implementations of embedded programs with minimal forethought or prerequisite knowledge.

REFERENCES

[1] Free Software Foundation, Inc. URL: http://web.mit.edu/gnu/doc/html/binutils_7.html (visited on 03/01/2021).

[2] *8-bit PIC® and AVR® Microcontrollers*. Microchip Technology Inc. 2018. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/30009630M.pdf (visited on 03/07/2021).

[3] *Arduino Products*. Arduino. URL: https://www.arduino.cc/en/main/products (visited on 03/10/2021).

[4] *ATmega168*. Microchip Technology Inc. 2021. URL: https://www.microchip.com/wwwproducts/en/ATmega168 (visited on 03/07/2021).

[5] *ATmega8 LED blinking project not working properly*. URL: https://electronics.stackexchange.com/questions/136482/atmega8-led-blinking-project-not-working-properly (visited on 04/18/2021).

[6] *ATmega88/ATmega168 Datasheet*. Atmel Corporation. 2016. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-9365-Automotive-Microcontrollers-ATmega88-ATmega168_Datasheet.pdf (visited on 02/08/2021).

[7] *AVR Built-in Functions*. Free Software Foundation, Inc. URL: https://gcc.gnu.org/onlinedocs/gcc/AVR-Built-in-Functions.html (visited on 02/08/2021).

[8] *AVR C : 8 Bit Counter using button*. URL: https://stackoverflow.com/questions/42336207/avr-c-8-bit-counter-using-button (visited on 04/18/2021).

[9] *AVR C : 8 Bit Counter using button*. URL: https://kartikmohta.com/tech/avr/tutorial/ (visited on 04/18/2021).

[10] *AVR C Programming two functions on button press with delay*. URL: https://stackoverflow.com/questions/26325687/avr-c-programming-two-functions-on-button-press-with-delay (visited on 04/18/2021).

[11] *AVR Instruction Set Manual*. Atmel Corporation. 2016, p. 131. URL: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf (visited on 02/15/2021).

[12] *AVR Tutorial - Getting Started: Blinking an LED*. Micah & Carrick. URL: https://www.micahcarrick.com/getting-started.html (visited on 04/18/2021).

[13] *AVR® Microcontrollers Peripheral Integration*. Microchip Technology Inc. 2020. URL: https://www.microchip.com/content/dam/mchp/documents/MCU08/ProductDocuments/Brochures/30010135E.pdf (visited on 03/04/2021).

[14] *AVR131: Using the AVR's High-speed PWM*. Atmel Corporation. 2003. URL: http://edge.rit.edu/edge/P08027/public/References/Robot/doc2542.pdf (visited on 02/08/2021).

[15] Salim Mohammed Benabadji. *10 simple tricks to optimize your C code in small embedded systems*. AspenCore. 2019. URL: https://www.embedded.com/10-simple-tricks-to-optimize-your-c-code-in-small-embedded-systems/ (visited on 03/05/2021).

[16] Jacob Beningo. *Tips and Tricks - Fast Divide and Multiply*. Beningo Embedded Group. 2015. URL: https://www.beningo.com/tips-and-tricks-fast-divide-and-multiply/ (visited on 03/05/2021).

[17] Richard Carr. *Gang of Four Design Patterns*. BlackWasp. 2009. URL: http://www.blackwasp.co.uk/gofpatterns.aspx (visited on 02/22/2021).

[18] Matt Chernosky. *Simple metrics for embedded software*. ElectronVector. 2019. URL: http://www.electronvector.com/blog/simple-metrics-for-embedded-software (visited on 03/04/2021).

[19] *Convenience functions for busy-wait delay loops*. 2016. URL: https://www.nongnu.org/avr-libc/user-manual/group__util__delay.html (visited on 02/15/2021).

[20] *Efficient Multiplication and Division Using MSP430™ MCUs*. Texas Instruments Incorporated. 2018. URL: https://www.ti.com/lit/an/slaa329a/slaa329a.pdf?ts=1614886162928&ref_url=https%253A%252F%252Fwww.google.com%252F (visited on 03/04/2021).

[21] *Embedded C Programming III*. Renesas Electronics Corporation. 2004. URL: https://www.renesas.com/us/en/document/apn/embedded-programming-iii-ecprogramiiiopt (visited on 03/06/2021).

[22] *Embedded Compilers*. CPU Technologies. 2021. URL: https://microcontroller.com/Embedded_Compilers/ (visited on 03/04/2021).

[23] *Engineering embedded software for optimum performance: Part 1 – basic C coding techniques*. AspenCore. 2014. URL: https://www.embedded.com/engineering-embedded-software-for-optimum-performance-part-1-basic-c-coding-techniques/ (visited on 03/06/2021).

[24] Lou Frenzel. Electronic Design. 2016. URL: https://www.electronicdesign.com/technologies/microcontrollers/article/21802086/and-the-best-micro-for-beginner-learning-is (visited on 03/01/2021).

[25] *GCC, the GNU Compiler Collection*. Free Software Foundation, Inc. 2021. URL: https://gcc.gnu.org/ (visited on 02/08/2021).

[26] *GNU GENERAL PUBLIC LICENSE*. Free Software Foundation, Inc. URL: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 04/09/2021).

[27] *How to calculate CPU utilization*. AspenCore. 2004. URL: https://www.embedded.com/how-to-calculate-cpu-utilization/ (visited on 02/16/2021).

[28] Aimal Khan. *10 Best Microcontroller Boards for Engineers and Geeks*. Engineering Passion. 2021. URL: https://www.engineeringpassion.com/10-best-microcontroller-boards-for-engineers-and-geeks/ (visited on 03/01/2021).

[29] Philip Koopman. *Embedded System Engineering Economics*. Carnegie Mellon. 2015. URL: http://users.ece.cmu.edu/~koopman/ece649/lectures/12_product_economics.pdf (visited on 03/10/2021).

[30] Michael E. Lee. *Optimization of Computer Programs in C*. University of Strasbourg. 1997. URL: http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html (visited on 03/06/2021).

[31] Tim Littlefair. *CCCC - C and C++ Code Counter*. Edith Cowan University. 2013. URL: http://cccc.sourceforge.net/ (visited on 02/09/2021).

[32] *LLVM's Analysis and Transform Passes*. LLVM Foundation. 2021. URL: https://llvm.org/docs/Passes.html (visited on 03/06/2021).

[33] *Minimizing Interrupt Response Time*. Atmel Corporation. 2005. URL: http://web.engr.oregonstate.edu/~traylor/ece473/pdfs/minimize_interrupt_response_time.pdf (visited on 02/15/2021).

[34] *MPLAB® Code Configurator*. Microchip Technology Corporation. URL: https://www.microchip.com/mplab/mplab-code-configurator (visited on 02/08/2021).

[35] *MPLAB® X Integrated Development Environment (IDE)*. Microchip Technology Corporation. URL: https://www.microchip.com/en-us/development-tools-tools-and-software/mplab-x-ide (visited on 02/08/2021).

[36] *Options That Control Optimization*. Free Software Foundation, Inc. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (visited on 03/01/2021).

[37] *Raspberry Pi 3 Model B*. Raspberry Pi Foundation. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (visited on 03/28/2021).

[38] Martin Shepperd. "A critique of cyclomatic complexity as a software metric". In: *Software Engineering Journal* (1988), pp. 30–36. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.6535&rep=rep1&type=pdf (visited on 02/17/2021).

[39] *Software Metrics*. JavaTpoint. URL: https://www.javatpoint.com/software-engineering-software-metrics (visited on 03/04/2021).

[40] *The LLVM Compiler Infrastructure*. LLVM Foundation. 2021. URL: https://llvm.org/ (visited on 03/06/2021).

[41] Thewrightstuff. URL: https://commons.wikimedia.org/wiki/File:Duty_Cycle_Examples.png (visited on 02/17/2021).

[42] *Top 10 Popular Microcontrollers Among Makers*. Electronics-Lab. 2020. URL: https://www.electronics-lab.com/top-10-popular-microcontrollers-among-makers/ (visited on 03/01/2021).

[43] Jonathan Valvano and Ramesh Yerraballi. *Embedded Systems - Shape The World*. .Chapter 11: Serial Interfacing. 2014. URL: https://users.ece.utexas.edu/~valvano/Volume1/E-Book/C11_SerialInterface.htm (visited on 02/09/2021).

[44] Eric Walkingshaw. *CS 609 Diskussionsseminar: Modular Extensibility*. Oregon State University. URL: http://web.engr.oregonstate.edu/~traylor/ece473/pdfs/minimize_interrupt_response_time.pdf (visited on 03/07/2021).

[45] Colin Walls. *Measuring interrupt latency*. Siemens Digital Industries Software. 2012. URL: https://blogs.sw.siemens.com/embedded-software/2012/06/05/measuring-interrupt-latency/ (visited on 02/18/2021).

[46] Dr. Gary Wang. *Brief History of Optimization*. Empower Operations Corp. 2018. URL: https://empowerops.com/en/blogs/2018/12/6/brief-history-of-optimization (visited on 03/06/2021).

[47] *What is Java and why is it important?* Code Institute. URL: https://codeinstitute.net/blog/what-is-java/ (visited on 02/22/2021).

[48] Elliot Williams. 2016. URL: https://github.com/hexagon5un/AVR-Programming/tree/master/AVR-Programming-Library (visited on 03/02/2021).

APPENDIX A

EMBEDDED C SOURCE OPTIMIZER CODE

The source code to the Embedded C Source Optimizer is hosted on GitHub (https://github.com/tblisonb/source-optimizer), and uses the open-source GNU General Public License v3.0 (GPLv3) [26].

APPENDIX B

TARGET PROGRAMS AND TOOLS

```
/**
 * Program toggles an LED and also waits for a button press which
 * increments a counter.
 */

#define F_CPU 8000000

#include <avr/io.h>
#include <util/delay.h>

/**
 * Initialize data direction registers and enable global interrupts
 */
void init() {
  // setup LED and Button pins as output and input respectively
  DDRD |= 0xFF;   // set all pins on bank D to output (for counter)
  DDRB |= 0xFF;   // set all pins on bank B to output (for LED blink)
  DDRC &= ~(1 << PC6);     // set PB7 as an input (for button)
  PORTC |= (1 << PC6);     // enable pull-up resistor on input pin
}

/**
 * Entry point
 */
int main() {
  init();
  // main loop
  while(1) {
    PORTB ^= (1 << PB0);     // toggle LED pin
    _delay_ms(1000);         // 1 second delay between blinks
    // check if button pin is low (pressed)
    if (!(PINC & (1 << PC6))) {
      PORTD++;             // increment counter
    }
  }
  return 0;
}
```

Figure B.1: Program 1 - A simple program which toggles an LED while simultaneously checking for a button press on an input pin which increments a counter (where a pin bank is used as the counter so it can be visualized).

```
/**
 * Program keeps track of two separate counters; one of which sets
 * a unique control signal on output bank D, and another counts the
 * number of button presses.
 */

#include <avr/io.h>
#include <util/delay.h>

void init() {
  // setup LED and Button pins as output and input respectively
  DDRD |= 0xFF;   // set all pins on bank D to output (for counter)
  DDRB |= 0xFF;   // set all pins on bank B to output (for LED blink)
  DDRC &= ~(1 << PC6);    // set PB7 as an input (for button)
  PORTC |= (1 << PC6);    // enable pull-up resistor on input pin
}

int main() {
  init();
  uint8_t n = 1;
  // main loop
  while(1) {
    _delay_ms(1000);        // 1 second delay between blinks
    // check if button pin is low (pressed)
    if (!(PINC & (1 << PC6))) {
      PORTB++;            // increment counter
    }
    PORTD = n;
    n = n << 1;
    if (n == 0) {
      n = 1;
    }
  }
  return 0;
}
```

Figure B.2: Program 2 - A program which modifies the state of two different bin banks, one of which increments on a button press, and the other left-shifts the state of the register every 1 second.

```
/**
 * Program sets software PWM output for 60% duty cycle on PB1 and
 * on button press calculates an iteration of the fibonacci sequence
 * starting at 1.
 */

#include <avr/io.h>
#include <util/delay.h>

uint8_t n = 0, m = 1, o;

/**
 * Initialize data direction registers and enable global interrupts
 */
void init() {
  // setup LED and Button pins as output and input respectively
  DDRD |= 0xFF;   // set all pins on bank D to output (for counter)
  DDRB |= 0xFF;   // set all pins on bank B to output (for LED blink)
  DDRB &= ~(1 << PB7);    // set PB7 as an input (for button)
  PORTB |= (1 << PB7);    // enable pull-up resistor on input pin
  SREG = 0x80;    // enable global interrupts
}

/**
 * Entry point
 */
int main() {
  init();
  // main loop
  while(1) {
    // set PWM output for a 60% duty cycle with a frequency of 100 Hz
    PORTB ^= (1 << PB1);    // toggle LED pin
    _delay_ms(6);
    PORTB ^= (1 << PB1);    // toggle LED pin
    _delay_ms(4);
    // check for button press
    if (!(PINB & (1 << PB5))) {
      o = n + m;
      n = m;
      m = o;
      PORTD = o;
    }
  }
  return 0;
}
```

Figure B.3: Program 3 - A program which outputs a software-driven PWM signal with a 60% duty cycle and frequency of 100Hz on a single pin, while also calculating and outputing a binary representation of the fibonacci sequence on a pin bank set to output.

```
/**
 * Definitions for adjusting parameters related to the measured utilization.
 *
 * Details:
 *   iterationCount keeps track of the number of iterations of the main
 *   control loop while timer instance 2 is counting. Timer instance 2 is
 *   configured to trigger an interrupt on overflow, incrementing
 *   overflowCount, and will output the contents of iteration counter over
 *   USART if the max number of overflows defined has been reached.
 *
 * Usage:
 *   The MAX_OVERFLOW_COUNT macro should be set according to how long
 *   execution is expected to take in the main control loop relative to the
 *   timer prescaler. The PRESCALER_DIVISOR macro can also be changed to affect
 *   the frequency which the timer ISR is called. The serial output shows the
 *   number of clock cycles passed over the duration and the iteration count,
 *   where the average period of an iteration is simply
 *   [(cycleCount/iterationCount)/F_CPU].
 */

#ifndef UTILIZATION_H
#define UTILIZATION_H

#include <stdint.h>

#define MAX_OVERFLOW_COUNT 8
#define PRESCALER_DIVISOR 1 // Values: 1, 8, 32, 64, 128, 256, 1024

volatile uint8_t iterationCount;
volatile uint8_t overflowCount;

void util_init();

#endif
```

Figure B.4: Header file included to add utilization measurement output to a given program.

```c
#include "utilization.h"
#include <avr/io.h>
#include "USART.h"

#define PD PRESCALER_DIVISOR
#define PRESCALER_VALUE (PD == 1 ? 1 : PD == 8 ? 2 : PD == 32 ? 3 : PD == 64 ? 4 : PD == 128 ? 5 : PD == 256 ? 6 : PD == 1024 ? 7 : 0)

void util_init() {
  iterationCount = 0;
  overflowCount = 0;
  __builtin_avr_sei();
  initUSART();
  // enable timer instance 2
  TCCR2A = 0x00;         // set normal mode operation
  TCCR2B = PRESCALER_VALUE; // set prescaler of clk/PRESCALER_DIVISOR
  TIMSK2 = 0x01;         // set overflow interrupt flag
  TIFR2  = 0x01;         // enable overflow interrupt
  TCNT2  = 0x00;         // initialize timer to 0
}

void __vector_9(void) __attribute__ ((signal, used, externally_visible));
void __vector_9(void) {  // Timer interrupt vector which will be called by the timer hardware on overflow
  overflowCount++;
  if (overflowCount >= MAX_OVERFLOW_COUNT) {
    printString("Clock cycles passed: ");
    uint32_t cycles = PRESCALER_DIVISOR * 256UL * ((uint32_t) overflowCount);
    // print number of clock cycles as a 4 digit hex value
    printHexByte((uint8_t) (cycles >> 24));
    printHexByte((uint8_t) (cycles >> 16));
    printHexByte((uint8_t) (cycles >> 8));
    printHexByte((uint8_t) cycles);
    // print carriage return + line feed
    printString("\r\n");
    printString("Control loop iterations: ");
    printHexByte(iterationCount);
    // print carriage return + line feed
    printString("\r\n\r\n");
    overflowCount = 0;
    iterationCount = 0;
  }
  TIFR2 = 0x01; // clear overflow flag
  TCNT2  = 0x00;  // reset timer to 0
}
```

Figure B.5: Source file which sets up a timer instance and adds an ISR for tracking the number of clock cycles passed and providing serial output [48] including cycle count and the number of iterations passed (stored in ’iterationCount’).

```c
#include ”utilization.h”

/**
 * Control program for measuring utilization.
 */
int main() {
  util_init();
  // main loop
  while(1) {
    iterationCount++;
  }
  return 0;
}
```

Figure B.6: Control program used as a baseline for measuring latency using the library in Figure B.4 and B.5.

```
#define F_CPU 1000000

#include <avr/io.h>
#include <util/delay.h>

void init() {
  DDRD |= (1 << PD1);   // set PD1 as output to indicate response
  DDRB |= 0xFF;   // set all pins on bank B to output (for LED blink)
  DDRC &= ~(1 << PC1);    // set as an input (for button press)
  PORTC |= (1 << PC1);    // enable pull-up resistor on input pin
}

int main() {
  init();
  // main loop
  while(1) {
    PORTB ^= (1 << PB0);  // toggle LED pin
    _delay_ms(50);  // delay to make LED blinks visible
    // check if button pin is low (pressed)
    if (!(PINC & (1 << PC1))) {
      // toggle output pin indicating response
      PORTD |= (1 << PD1);
      PORTD &= ~(1 << PD1);
    }
  }
  return 0;
}
```

Figure B.7: A modified version of Example Program 1 which supports measuring latency according to the testing described in Section 5.1.1.

```
#define F_CPU 1000000

#include <avr/io.h>
#include <util/delay.h>

void init() {
  // setup LED and Button pins as output and input respectively
  DDRD |= (1 << PD1); // set pin as output for response to button
  DDRB |= 0xFF; // set all pins on bank B to output (for counter)
  DDRC &= ~(1 << PC1); // set PB7 as an input (for button)
  PORTC |= (1 << PC1); // enable pull-up resistor on input pin
}

int main() {
  init();
  uint8_t n = 1;
  // main loop
  while(1) {
    PORTB = n;
    n = n << 1;
    if (n == 0) {
      n = 1;
    }
    _delay_ms(25);  // delay for LED counter
    // check if button pin is low (pressed)
    if (!(PINC & (1 << PC1))) {
      // toggle output pin indicating response
      PORTD |= (1 << PD1);
      PORTD &= ~(1 << PD1);
    }
  }
  return 0;
}
```

Figure B.8: A modified version of Example Program 2 which supports measuring latency according to the testing described in Section 5.1.1.

```
#define F_CPU 1000000

#include <avr/io.h>
#include <util/delay.h>

uint8_t n = 0, m = 1, o;

void init() {
  // setup LED and Button pins as output and input respectively
  DDRD |= (1 << PD1);     // set pin as output for response
  DDRD |= (1 << PD6);   // set pin as output for PWM signal
  DDRB |= 0xFF;       // set all pins on bank B to output
  DDRC &= ~(1 << PC1);  // set pin as input for button press
  PORTC |= (1 << PC1);    // enable pull-up resistor on input pin
  SREG = 0x80;    // enable global interrupts
}

int main() {
  init();
  // main loop
  while(1) {
    // set PWM output for a 60% duty cycle with a frequency of 100 Hz
    PORTD ^= (1 << PD6);    // toggle LED pin
    _delay_ms(6);
    PORTD ^= (1 << PD6);    // toggle LED pin
    _delay_ms(4);
    // check for button press
    if (!(PINC & (1 << PC1))) {
      // calculate next fibonacci number and display on PORTB
      o = n + m;
      n = m;
      m = o;
      PORTB = o;
      // toggle output pin indicating response
      PORTD |= (1 << PD1);
      PORTD &= ~(1 << PD1);
    }
  }
  return 0;
}
```

Figure B.9: A modified version of Example Program 3 which supports measuring latency according to the testing described in Section 5.1.1.

```c
/* REF: https://github.com/hexagon5un/AVR-Programming/blob/
  master/Chapter02_Programming-AVRs/blinkLED/blinkLED.c */

                                                       /* Blinker Demo */

// ------- Preamble -------- //
#include <avr/io.h>                        /* Defines pins, ports, etc */
#include <util/delay.h>                     /* Functions to waste time */


int main(void) {

  // -------- Inits --------- //
  DDRB |= 0b00000001;            /* Data Direction Register B:
                                   writing a one to the bit
                                   enables output. */

  // ------ Event loop ------ //
  while (1) {

    PORTB = 0b00000001;          /* Turn on first LED bit/pin in PORTB */
    _delay_ms(1000);                                           /* wait */

    PORTB = 0b00000000;          /* Turn off all B pins, including LED */
    _delay_ms(1000);                                           /* wait */

  }                                                  /* End event loop */
  return 0;                            /* This line is never reached */
}
```

Figure B.10: Alternate example program 1 [48].

```c
/* REF: https://stackoverflow.com/questions/26325687/
  avr-c-programming-two-functions-on-button-press-with-delay */

#include <avr/io.h> // added
#include <util/delay.h> // added

int i;
void led(void) {
  for (i = 0; i < 10; i++) {
    PORTB |= (1 << PB0);  //LED on
    _delay_ms(250);       //wait 250ms

    PORTB &= ~(1 << PB0); //LED off
    _delay_ms(250);       //wait 250ms
  }
}

int main() { // added
  while (1) {
    if (!(PINB & (1<<PB7)) ) {
      PORTB |= (1 << PB1); // Piezo on
      led();
    }
    else {
      PORTB &= ~(1 << PB1); // Piezo off
    }
  }
} // added
```

Figure B.11: Alternate example program 2 [10].

```c
/* REF: https://github.com/hexagon5un/AVR-Programming/blob/master/
  Chapter12_Analog-to-Digital-Conversion-II/voltmeter/voltmeter.c */

// ------- Preamble -------- //
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>                           /* for ADC sleep mode */
#include <math.h>                           /* for round() and floor() */

#include "pinDefines.h"
#include "USART.h"

#define REF_VCC 5.053
                                /* measured division by voltage divider */
#define VOLTAGE_DIV_FACTOR  3.114


// -------- Functions --------- //
void initADC(void) {
  ADMUX |= (0b00001111 & PC5);                      /* set mux to ADC5 */
  ADMUX |= (1 << REFS0);                  /* reference voltage on AVCC */
  ADCSRA |= (1 << ADPS1) | (1 << ADPS2);    /* ADC clock prescaler /64 */
  ADCSRA |= (1 << ADEN);                                 /* enable ADC */
}

void setupADCSleepmode(void) {
  set_sleep_mode(SLEEP_MODE_ADC);            /* defined in avr/sleep.h */
  ADCSRA |= (1 << ADIE);                       /* enable ADC interrupt */
  sei();                                   /* enable global interrupts */
}

EMPTY_INTERRUPT(ADC_vect);

uint16_t oversample16x(void) {
  uint16_t oversampledValue = 0;
  uint8_t i;
  for (i = 0; i < 16; i++) {
    sleep_mode();                   /* chip to sleep, takes ADC sample */
    oversampledValue += ADC;                        /* add them up 16x */
  }
  return (oversampledValue >> 2);          /* divide back down by four */
}

void printFloat(float number) {
  number = round(number * 100) / 100; /* round off to 2 decimal places */
  transmitByte('0' + number / 10);                       /* tens place */
  transmitByte('0' + number - 10 * floor(number / 10));        /* ones */
  transmitByte('.');
  transmitByte('0' + (number * 10) - floor(number) * 10);    /* tenths */
                                                   /* hundredths place */
  transmitByte('0' + (number * 100) - floor(number * 10) * 10);
  printString("\r\n");
}

int main(void) {

  float voltage;

  initUSART();
  printString("\r\nDigital Voltmeter\r\n\r\n");
  initADC();
  setupADCSleepmode();

  while (1) {

    voltage = oversample16x() * VOLTAGE_DIV_FACTOR * REF_VCC / 4096;
    printFloat(voltage);
    _delay_ms(500);

  }                                                  /* End event loop */
  return 0;                            /* This line is never reached */
}
```

Figure B.12: Alternate example program 3 [48].

```
// Ref: https://www.micahcarrick.com/getting-started.html

#define F_CPU 1000000UL

#include <avr/io.h>
#include <util/delay.h>

int main (void) {
    DDRB |= _BV(DDB0);

    while(1) {
        PORTB ^= _BV(PB0);
        _delay_ms(500);
    }
}
```

Figure B.13: Alternate example program 4 [12].

```
/* REF: https://electronics.stackexchange.com/questions/136482/
  atmega8-led-blinking-project-not-working-properly?rq=1 */

#include <avr/io.h>
#include <util/delay.h>

int main(void) {
    DDRD = 0b10000000;
    DDRD = 0b01000000;

    while(1) {
        PORTD = 0b10000000;
        _delay_ms(100);
        PORTD = 0b00000000;
        PORTD = 0b01000000;
        _delay_ms(100);
        PORTD = 0b00000000;
    }
    return 1;
}
```

Figure B.14: Alternate example program 5 [5].

```
/* REF: https://github.com/hexagon5un/AVR-Programming/blob/master/Chapter04
  _Bit-Twiddling/cylonEyes_quasiRandomToggle/quasiRandomToggle.c */

                                                         /* Cylon Eyes */

// ------- Preamble -------- //
#include <avr/io.h>                        /* Defines pins, ports, etc */
#include <util/delay.h>                     /* Functions to waste time */

#define DELAYTIME 45                                   /* milliseconds */
#define LED_PORT                PORTB
#define LED_PIN                 PINB
#define LED_DDR                 DDRB

int main(void) {

  // -------- Inits --------- //
  uint16_t x = 0x1234;
  uint8_t y;
  LED_DDR = 0xff;               /* Data Direction Register B:
                                   all set up for output */

  // ------ Event loop ------ //
  while (1) {

    x = 2053 * x + 13849;                 /* "random" number generator */
    y = (x >> 8) & 0b00000111;       /* pick three bits from high byte */
    LED_PORT ^= (1 << y);                            /* toggle one bit */
    _delay_ms(100);

  }                                                  /* End event loop */
  return 0;
}
```

Figure B.15: Alternate example program 6 [48].

```c
/* REF: https://github.com/hexagon5un/AVR-Programming/blob/master/Chapter15
   _Advanced-Motors/hBridgeWorkout/hBridgeWorkout.c */

// Simple demo of an h-bridge

// ------- Preamble -------- //
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "pinDefines.h"

static inline void setBridgeState(uint8_t bridgeA, uint8_t bridgeB) {
  /* Utility function that lights LEDs when it energizes a bridge side */
  if (bridgeA) {
    PORTD |= (1 << PD6);
    LED_PORT |= (1 << LED0);
  }
  else {
    PORTD &= ~(1 << PD6);
    LED_PORT &= ~(1 << LED0);
  }
  if (bridgeB) {
    PORTD |= (1 << PD5);
    LED_PORT |= (1 << LED1);
  }
  else {
    PORTD &= ~(1 << PD5);
    LED_PORT &= ~(1 << LED1);
  }
}


int main(void) {
  // -------- Inits --------- //

  DDRD |= (1 << PD6);               /* now hooked up to bridge, input1 */
  DDRD |= (1 << PD5);               /* now hooked up to bridge, input2 */
  LED_DDR |= (1 << LED0);
  LED_DDR |= (1 << LED1);

  // ------ Event loop ------ //
  while (1) {

    setBridgeState(1, 0);                                 /* "forward" */
    _delay_ms(2000);

    setBridgeState(0, 0);                      /* both low stops motor */
    _delay_ms(2000);

    setBridgeState(0, 1);                                 /* "reverse" */
    _delay_ms(2000);

    setBridgeState(1, 1);                /* both high also stops motor */
    _delay_ms(2000);

    // For extra-quick braking, energize the motor backwards
    setBridgeState(1, 0);
    _delay_ms(2000);
    setBridgeState(0, 1);
    _delay_ms(75);              /* tune this time to match your system */
    setBridgeState(0, 0);
    _delay_ms(2000);

  }                                                  /* End event loop */
  return 0;
}
```

Figure B.16: Alternate example program 7 [48].

```
/* REF: https://github.com/hexagon5un/AVR-Programming/blob/master/Chapter07_
        Analog-to-Digital-Conversion-I/lightSensor/lightSensor.c */

// Quick Demo of light sensor

// ------- Preamble -------- //
#include <avr/io.h>
#include <util/delay.h>
#include "pinDefines.h"

// -------- Functions --------- //
static inline void initADC0(void) {
  ADMUX |= (1 << REFS0);                  /* reference voltage on AVCC */
  ADCSRA |= (1 << ADPS2);                   /* ADC clock prescaler /16 */
  ADCSRA |= (1 << ADEN);                                 /* enable ADC */
}

int main(void) {

  // -------- Inits --------- //
  uint8_t ledValue;
  uint16_t adcValue;
  uint8_t i;

  initADC0();
  LED_DDR = 0xff;

  // ------ Event loop ------ //
  while (1) {

    ADCSRA |= (1 << ADSC);                     /* start ADC conversion */
    loop_until_bit_is_clear(ADCSRA, ADSC);          /* wait until done */
    adcValue = ADC;                                     /* read ADC in */
                        /* Have 10 bits, want 3 (eight LEDs after all) */
    ledValue = (adcValue >> 7);
                                   /* Light up all LEDs up to ledValue */
    LED_PORT = 0;
    for (i = 0; i <= ledValue; i++) {
      LED_PORT |= (1 << i);
    }
    _delay_ms(50);
  }                                                  /* End event loop */
  return 0;                            /* This line is never reached */
}
```

Figure B.17: Alternate example program 8 [48].

```c
/* REF: https://github.com/hexagon5un/AVR-Programming/blob/master/Chapter17
  _I2C/i2cThermometer/i2cThermometer.c */

                 /* Reads LM75 Thermometer and Prints Value over Serial */

// ------- Preamble -------- //
#include <avr/io.h>
#include <util/delay.h>
#include <avr/power.h>

#include "pinDefines.h"
#include "USART.h"
#include "i2c.h"

// -------- Defines -------- //

#define LM75_ADDRESS_W               0b10010000
#define LM75_ADDRESS_R               0b10010001
#define LM75_TEMP_REGISTER           0b00000000
#define LM75_CONFIG_REGISTER         0b00000001
#define LM75_THYST_REGISTER          0b00000010
#define LM75_TOS_REGISTER            0b00000011
// -------- Functions --------- //

int main(void) {

  uint8_t tempHighByte, tempLowByte;

  // -------- Inits --------- //
  clock_prescale_set(clock_div_1);                              /* 8MHz */
  initUSART();
  printString("\r\n====  i2c Thermometer  ====\r\n");
  initI2C();

  // ------ Event loop ------ //
  while (1) {
                         /* To set register, address LM75 in write mode */
    i2cStart();
    i2cSend(LM75_ADDRESS_W);
    i2cSend(LM75_TEMP_REGISTER);
    i2cStart();                       /* restart, just send start again */
                               /* Setup and send address, with read bit */
    i2cSend(LM75_ADDRESS_R);
                                /* Now receive two bytes of temperature */
    tempHighByte = i2cReadAck();
    tempLowByte = i2cReadNoAck();
    i2cStop();

    // Print it out nicely over serial for now...
    printByte(tempHighByte);
    if (tempLowByte & _BV(7)) {
      printString(".5\r\n");
    }
    else {
      printString(".0\r\n");
    }

                                                     /* Once per second */
    _delay_ms(1000);

  }                                                   /* End event loop */
  return 0;                             /* This line is never reached */
}
```

Figure B.18: Alternate example program 9 [48].

```
/* REF: https://stackoverflow.com/questions/42336207/
  avr-c-8-bit-counter-using-button */

#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>

/*
Board digital I/O pin to atmega328 registers for LEDS
| d2  | d3  | d4  | d5  | d6  | d7  | d8  | d9  |
| pd2 | pd3 | pd4 | pd5 | pd6 | pd7 | pb0 | pd1 |

Input Button
| d9  |
| pb2 |
*/


int main(void) {

    uint8_t x = 0;

    DDRD = 0b11111100;
    PORTD = 0b00000000;

    DDRB = 0b00000011;
    PORTB = 0b00000100;

    while(1) {

        if((PINB & 0b00000100) == 0) {

            ++x;

            PORTD = x << 2;
            PORTB = (PORTB & 0b11111100) | ((x >> 6) & 0b00000011);
        }

        _delay_ms(80);
    }
    return 0;
}
```

Figure B.19: Alternate example program 10 [8].

```
/* REF: https://github.com/hexagon5un/AVR-Programming/blob/master/Chapter04
  _Bit-Twiddling/showingOffBits/showingOffBits.c */

            /* Showing off some patterns to practice our bit-twiddling */

// ------- Preamble -------- //
#include <avr/io.h>
#include <avr/power.h>
#include <util/delay.h>                      /* Functions to waste time */

#define DELAYTIME 85                                    /* milliseconds */
#define LED_PORT                PORTB
#define LED_DDR                 DDRB

int main(void) {

  uint8_t i;
  uint8_t repetitions;
  uint8_t whichLED;
  uint16_t randomNumber = 0x1234;

  // -------- Inits --------- //
  LED_DDR = 0xff;                     /* all LEDs configured for output */
  // ------ Event loop ------ //
  while (1) {
                                                             /* Go Left */
    for (i = 0; i < 8; i++) {
      LED_PORT |= (1 << i);                    /* turn on the i'th pin */
      _delay_ms(DELAYTIME);                                     /* wait */
    }
    for (i = 0; i < 8; i++) {
      LED_PORT &= ~(1 << i);                  /* turn off the i'th pin */
      _delay_ms(DELAYTIME);                                     /* wait */
    }
    _delay_ms(5 * DELAYTIME);                                  /* pause */

                                                            /* Go Right */
    for (i = 7; i < 255; i--) {
      LED_PORT |= (1 << i);                    /* turn on the i'th pin */
      _delay_ms(DELAYTIME);                                     /* wait */
    }
    for (i = 7; i < 255; i--) {
      LED_PORT &= ~(1 << i);                  /* turn off the i'th pin */
      _delay_ms(DELAYTIME);                                     /* wait */
    }
    _delay_ms(5 * DELAYTIME);                                  /* pause */

                                    /* Toggle "random" bits for a while */
    for (repetitions = 0; repetitions < 75; repetitions++) {
                                           /* "random" number generator */
      randomNumber = 2053 * randomNumber + 13849;
                                       /* low three bits from high byte */
      whichLED = (randomNumber >> 8) & 0b00000111;
      LED_PORT ^= (1 << whichLED);                    /* toggle our LED */
      _delay_ms(DELAYTIME);
    }
    LED_PORT = 0;                                       /* all LEDs off */
    _delay_ms(5 * DELAYTIME);                                  /* pause */

  }                                                   /* End event loop */
  return 0;                             /* This line is never reached */
}
```

Figure B.20: Alternate example program 11 [48].

```
// Ref: https://kartikmohta.com/tech/avr/tutorial/

/* port_test4.c
 * This program checks the 0th pin of port D for input and if it is ON (Logic 1), keeps writing 1 to
 * 0th pin of port B else if it is OFF (Logic 0), it toggles pin 0 of port B
 */

#include <inttypes.h> // short forms for various integer types

#include <avr/io.h> // Standard include for AVR

#define F_CPU 16000000UL // Crystal frequency required for delay functions

#include <util/delay.h> // Delay functions

#define sbi(x, y) x |= _BV(y)                  // set bit
#define cbi(x, y) x &= ~(_BV(y))               // clear bit
#define tbi(x, y) x ^= _BV(y)                  // toggle bit
#define is_high(x, y) ((x & _BV(y)) == _BV(y)) // check if the input pin is high
#define is_low(x, y) ((x & _BV(y)) == 0)       // check if the input pin is low

int main() {
    DDRB = 0xff; // PORTB as OUTPUT
    PORTB = 0x00;
    DDRD = 0x00;  // PORTD as INPUT
    PORTD = 0xff; // Enable Pull-up on the input port

    while(1) { // Infinite loop
        uint8_t i;

        for(i = 0; i < 2; i++) {
            if(i == 0) {
                sbi(PORTB, PB0);
            }
            else if(i == 1) {
                if(is_low(PIND, PD0)) {
                    cbi(PORTB, PB0);
                }
                else {
                    sbi(PORTB, PB0);
                }
            }
            _delay_ms(100);
        }
    }
    return 0;
}
```

Figure B.21: Alternate example program 12 [9].

Table B.1

Results of latency testing, where each optimization option was applied independently across three example programs. Measurements for: mean, minimum, maximum, and standard deviation are provided (in microseconds) for each optimization option, with a sample size of one thousand. Optimization options which could not be applied to the example programs were not tested, and therefore not included in this table.

	Optimization	Mean	Min	Max	Std Dev.
Example 1	None	24847.8	365.0	48480.0	14437.7
	Counter/Timer	29.1	17.0	99.0	7.0
	Counter+TSOoE	24770.1	294.0	48290.0	14390.9
	Interrupts	56.5	12.0	65.0	2.0
Example 2	None	12073.1	65.0	24170.0	7183.4
	Counter/Timer	29.0	14.0	81.0	6.3
	Counter+TSOoE	11940.1	28.0	25252.0	7169.7
	Interrupts	56.6	24.0	63.0	1.5
Example 3	None	5477.7	402.0	14748.0	2831.7
	Counter/Timer	69.2	21.0	197.0	30.9
	Counter+TSOoE	4757.4	108.0	10902.0	2755.0
	Interrupts	77.8	50.0	86.0	1.2
	PWM	45.0	38.0	102.0	3.8
	PWM+IDC	44.6	29.0	56.0	3.3
	Built-in	5465.6	328.0	20395.0	2854.2