International Journal of Computer Techniques–IJCT Volume 12 Issue 6, November 2025

Open Access and Peer Review Journal ISSN 2394-2231 https://ijctjournal.org/

Precision Timing Algorithms for AVR-based Arduino Systems: Assembly-Level Mathematical Modeling for Exact Cycle-Accurate Delays

José Miguel Morán Loza
Depto. de Electro-Fotónica
Universidad de Guadalajara, CUCEI,
Guadalajara, México
miguel.moran@academicos.udg.mx

Alicia García Arreola
Depto. de Electro-Fotónica
Universidad de Guadalajara, CUCEI,
Guadalajara, México
alicia.garreola@academicos.udg.mx

***Abstract*—This research presents a mathematical methodology for designing software delays with cycle-accurate precision for AVR-based Arduino systems, using assembly language programming. Through analysis of AVR instruction timing at the machine level, precise mathematical models are developed that translate timing requirements into exact assembly instruction sequences. The approach eliminates compiler optimization uncertainties and provides deterministic timing control with nanosecond precision. The methodology addresses critical applications requiring exact timing, where hardware timers are unavailable and C-level abstractions introduce timing variability. Mathematical equations are derived that account for every instruction cycle, enabling precise delay generation within ±1 clock cycle across the ATmega328P operational range. Validation through oscilloscope measurements and cycle-accurate simulation confirms timing precision with 62.5 nanosecond resolution at 16 MHz operation.**

Keywords*— *Assembly language, AVR microcontrollers, cycle-accurate timing, mathematical modeling, embedded systems, software delay, blocking delays, Arduino.

I. INTRODUCTION

Modern embedded applications increasingly demand precise timing control that exceeds the capabilities of high-level language implementations. While C-based Arduino programming offers accessibility and rapid prototyping, abstraction layers and compiler optimizations introduce timing uncertainties that are unacceptable for applications requiring nanosecond-level precision [1][2].

Assembly language programming provides direct control over instruction execution, enabling exact, cycle-by-cycle timing predictions. In AVR microcontrollers, each assembly instruction has a defined execution time, making mathematical modeling of delays both practical and precise. This research presents a systematic approach for designing assembly-level delay routines with mathematical precision.

Unlike previous work focusing on high-level delay implementations [3][4], the presented methodology operates at the instruction level, providing deterministic timing regardless of compiler settings or optimization levels. Building upon mathematical modeling techniques for embedded delays [5][6], these concepts have been adapted specifically for AVR assembly programming.

The primary contributions of this work include: development of exact cycle mathematical models for AVR assembly instructions, systematic methodology for converting timing requirements into assembly code, comprehensive validation of timing precision, and practical implementation techniques for Arduino development environments.

A. Advantages of Assembly-Level Timing

Assembly language offers several critical advantages for precision timing:

- Deterministic execution: Each instruction has known cycle count.
- Compiler independence: No optimization uncertainties.
- Cycle-level control: Direct manipulation of instruction timing.
- Maximum efficiency: Minimal overhead and maximum precision.

These advantages make Assembly the optimal choice for applications requiring exact timing control, such as custom communication protocols, sensor interfacing, and real-time signal generation.

II. AVR ASSEMBLY, ARCHITECTURE AND TIMING ANALYSIS

A. ATmega328P Instruction Set Timing

The ATmega328P employs a RISC architecture where most instructions execute in a single clock cycle [7]. Critical timing characteristics include:

- Arithmetic/Logic Instructions: ADD, SUB, AND, OR, INC, DEC (1 cycle)
- Branch/Call Instructions: RJMP (2 cycles), JMP (3 cycles), CALL, RET (4 cycles)
- Conditional Branch Instructions: BRNE, BREQ, BRMI (1 cycle if not taken, 2 cycles if taken)

- Data Transfer Instructions: LDI (1 cycle), LD, ST (2 cycles)

Operating at 16 MHz, each cycle represents exactly 62.5 nanoseconds $(T = 1/F)$, providing the fundamental timing resolution.

B. *Assembly Delay Algorithm Structure*

Fig. 1 shows a program for turning an LED on and off, known as blinking LED. The flowchart refers to the delay function, which is developed in Fig. 2 and analyzed in assembler in point c.

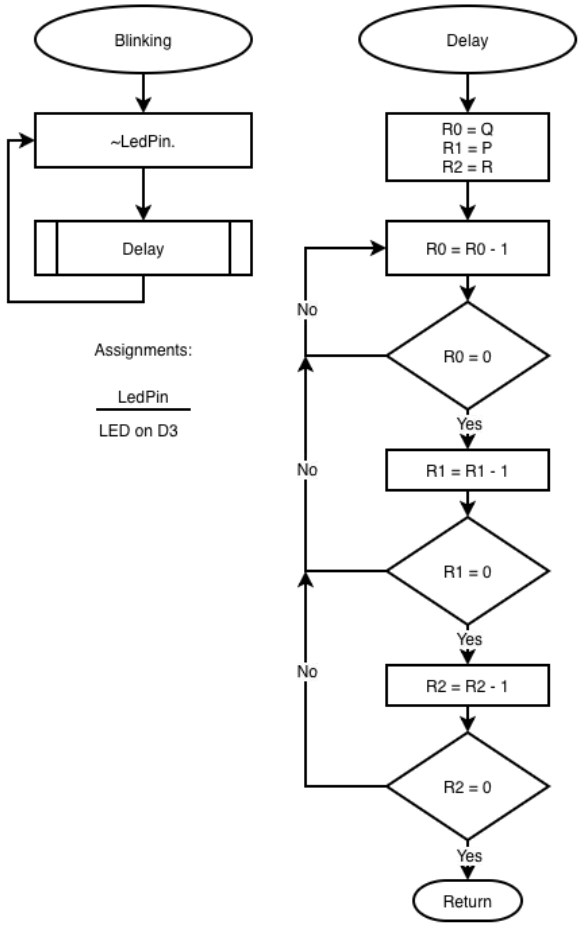

Fig. 1. Flow diagram of the Blinking LED program, which includes a delay call routine.

Figure 2 shows a structure consisting of three nested loops. The first loop uses register R23 and decrements the variable Q until it equals zero. The second loop uses register R24 and decrements the variable P until it equals zero. The third and final loop uses register R25 and decrements the variable R until it equals zero. In all cases, when the condition is not equal to zero, the algorithm repeats from the decrement of R23 until it returns to zero, executing 256 cycles for each decrement of R24. A similar process occurs with R25: if it is not zero, it returns to R23 from 0 to 0 for each decrement of R24. If R24 is zero, the process of decrementing from 0 to 0 is repeated, resulting in an increase in cycles and prolonged delays.

```
;*************** Delay Algorithm ***************
delay:
    ; Input parameters: R23=Q, R24=P, R25=R
    ; Initialize values

    LDI R23, Q    ; Q → R23 (1 cycle)
    LDI R24, P    ; P → R24 (1 cycle)
    LDI R25, R    ; R → R25 (1 cycle)

loop:
    DEC R23       ; Decrement Q (1 cycle)
    BRNE loop     ; Jump if not zero (2/1 cycles)

    DEC R24       ; Decrement P (1 cycle)
    BRNE loop     ; Jump if not zero (2/1 cycles)

    DEC R25       ; Decrement R (1 cycle)
    BRNE loop     ; Jump if not zero (2/1 cycles)

    RET           ; Return (4 cycles)
; ********************************************
```

Fig. 2. Assembly language delay algorithm for AVR architecture.

C. *Cycle-by-Cycle Mathematical Analysis*

Through detailed instruction analysis, the exact cycle count for each loop level is derived:

1) Constant Loading Analysis (Q, P, R):

- LDI R23, Q: 1 cycle = (1) cycle
- LDI R24, P: 1 cycle = (1) cycle
- LDI R25, R: 1 cycle = (1) cycle
- Total: TK1 = (1+1+1) cycles = (3) cycles

2) R23 Loop Analysis (Q iterations):

- DEC R23: 1 cycle × (Q) times = (Q) cycles
- BRNE loop: 2 cycles × (Q-1) times + 1 cycle × (Q-(Q-1)) times = (2Q-1) cycles
- Total: TQ = (Q) cycles + (2Q-1) cycles = (3Q - 1) cycles

3) R24 Loop Analysis (P iterations):

- DEC R24: 1 cycle × (P) times = (P) cycles
- Inner loop with R23 = 0 ∴ Q = 256: (3(256) - 1) cycles × (P-1) times = 767×(P-1) cycles
- BRNE loop: 2 cycles × (P-1) times + 1 cycle × (P-(P-1)) times = (2P-1) cycles
- Total: TP = (P) cycles + [767×(P-1) cycles] + (2P-1) cycles = (770P - 768) cycles

4) R25 Loop Analysis (R iterations):

- DEC R25: 1 cycle × (R) times = (R) cycles
- Inner loop with R23 = 0, R24 = 0 ∴ Q = 256, P = 256: [TQ + TP]×(R-1) cycles

- BRNE loop: 2 cycles × (R-1) times + 1 cycle × (R-(R-1)) times = (2R-1) cycles
- Total: TR = (R) cycles + [197119×(R-1) cycles] + (2R-1) cycles = (197122R - 197120) cycles

5) Function Exit Analysis:

- RET: 4 cycles = (4) cycles
- Total: TK2 = (4) cycles

The total cycles of the complete algorithm are obtained by adding the partial cycles, as shown in (1):

$$T_{total} = [TK1 + TQ + TP + TR + TK2]cycles \quad (1)$$

Substituting the obtained values in (1):

$$T_{total} = [3 + (3Q - 1) + (770P - 768) + (197122R - 197120) + 4]cycles \quad (2)$$

Simplifying (2):

$$T_{total} = [3Q + 770P + 197122R - 197882]\ cycles \quad (3)$$

The equation is defined in cycles or oscillator periods, but it can be adjusted to any operating frequency. Remember that (a period is the inverse of the frequency, T=1/f). For example, the typical operating frequency in Arduino (ATmega328P) is 16 MHz, so one cycle or period is equal to (1/16 MHz) or 62.5 ns. so (3) is adjusted to operate in time, replacing cycles with the equivalent in time (4):

$$T_{total} = [...]cycles \times \frac{62.5\ ns}{1\ cycles}$$

$$T_{total} = [3Q + 770P + 197122R - 197882] \times 62.5ns \quad (4)$$

From (4), the minimum (TT_{Min}) and maximum (TT_{Max}) times provided by the function are determined. The time TT_{Min} is obtained by considering that Q, P, and R have a value of 1, resulting in (5). Similarly, TT_{Max} is calculated by considering that Q, P, and R are equal to 0 or their equivalent 256, obtaining (6). The complete range of time (TT_{Range}) generated by the function is spresented in (7).

$$T_{Min} = [3(1) + 770(1) + 197122(1) - 197882] \times 62.5ns$$

$$= (13) \times 62.5ns = 812.5ns \quad (5)$$

$$T_{Max} = [3(256) + 770(256) + 197122(256) - 197882] \times 62.5\ ns$$

$$= 50{,}463{,}238 \times 62.5\ ns = 3{,}153{,}952{,}375\ ns \quad (6)$$

$$812.5\ ns \leq TT_{Range} \leq 3{,}153{,}952{,}375\ ns \quad (7)$$

From above, it follows that: $1 \leq Q \leq 256$, $1 \leq P \leq 256$, $1 \leq R \leq 256$. It is important to remember that the value 256 is equivalent to assigning a "0" to registers R23, R24, and R25. The range of these values is due to the size of the system registers, which is 8 bits.

III. Mathematical Model and Solution Method

A. Equation Solution Methodology

Next, we solve (4) to express the desired time on the left side of the equation, obtaining (8). The result is called the Target Time (T_{Tgt}), from which two expressions are derived: the first defines the Target Time constant (9), and the second simplifies the equation to determine its solution (10).

$$T_{Obj} = \frac{Ttotal}{62.5ns} = (3Q + 770P + 197122R - 197882) \quad (8)$$

$$T_{Tgt} = \left(\frac{T_{total}}{62.5\ ns}\right) \quad (9)$$

$$T_{Tgt} = 3Q + 770P + 197122R - 197882 \quad (10)$$

From equation (10), the expressions needed to calculate the values of Q, P, and R are obtained.

Step 1: It can be seen that the greatest contribution to time comes from the variable R due to its dominant coefficient. For this reason, it is considered that the variables Q and P do not make a significant contribution, thus generating an equation to determine the value of R.

Consequently, in (10) it is assumed that Q = 0 and P = 0, obtaining (11).

$$T_{Tgt} = 3(0) + 770(0) + 197122R - 197882 \quad (11)$$

Solving (11) to obtain the value of R, gives (12):

$$R = \left(\frac{T_{Tgt} + 197882}{197122}\right) \quad (12)$$

At this point, it is important to recall the notation used to represent the set of REAL numbers ($\mathbb{R}$)[8], as shown in (13).

$$W = [W] + \{W\} \quad (13)$$

Where [W] represents the integer part of a real number and {W} its fractional part.

Therefore, it is established from (12) that only the integer value [R] is required, as shown in (14).

$$[R] = \left\lfloor \frac{T_{Tgt} + 197882}{197122} \right\rfloor \quad (14)$$

Step 2: The value found for [R] is replaced and $Q = 0$ is considered, since it is the variable that contributes least to (10). With this substitution, it is possible to calculate the value of [P] (15).

$$T_{Tgt} = 3(0) + 770(P) + 197122[R] - 197882$$
$$= 197122[R] - 770(P) - 197882 \quad (15)$$

Solving (15) gives (16):

$$[P] = \left\lfloor \frac{T_{Tgt} + 197882 - 197122[R]}{770} \right\rfloor \quad (16)$$

Step 3: With the values obtained from [R] and [P], substitute them into (10) to determine [Q] (17):

$$T_{Tgt} = 3(Q) + 770[P] + 197122[R] - 197882 \quad (17)$$

Solving (17) gives (18):

$$[Q] = \left\lfloor \frac{T_{Tgt} + 197882 - 197122[R] - 770[P]}{3} \right\rfloor \quad (18)$$

Step 4: 4) Once the equations for calculating $[R]$, $[P]$, and $[Q]$ have been obtained, it is possible to determine the coefficients for any value within the range defined in (7). However, as discussed in [5], a brief analysis of the fractional part $\{Q\}$ is still required in order to allow for fine adjustments and improve the accuracy of the algorithm. For the AVR architecture, it should be noted that the coefficient Q is equal to 3 in (10); therefore, the possible results in $\{Q\}$ are:

- If {Q} = 0, the routine is exact.
- If {Q} = 0.33333, the routine needs one cycle and will compensate with one NOP before exiting.
- If {Q} = 0.6666, the routine needs two cycles and will compensate with two NOPs before exiting.

B. Optimization

When precise timing requires cycle adjustment, fine tuning is implemented using the NOP instruction for one cycle. Therefore, depending on the result obtained in {Q}, the NOP instructions located before the "return" instruction are selectively commented out or not (Fig. 3).

```
;*************** Delay Algorithm ***************
delay:
    ; Input parameters: R23=Q, R24=P, R25=R
    ; Initialize values

    LDI R23, Q      ; Q → R23 (1 cycle)
    LDI R24, P      ; P → R24 (1 cycle)
    LDI R25, R      ; R → R25 (1 cycle)

loop:
    DEC R23         ; Decrement Q (1 cycle)
    BRNE loop       ; Jump if not zero (2/1 cycles)

    DEC R24         ; Decrement P (1 cycle)
    BRNE loop       ; Jump if not zero (2/1 cycles)

    DEC R25         ; Decrement R (1 cycle)
    BRNE loop       ; Jump if not zero (2/1 cycles)

    NOP             ;1 NOP for {Q} = 0.33
    NOP             ;2 NOP for {Q} = 0.66

RET                 ; Return (4 cycles)
; ********************************************
```

Fig. 3. Compensated delay algorithm in assembly language for AVR architecture.

C. Validation

Table 1 presents the coefficient calculations (R, P, Q) for different time bases (T_{total}) within the range established in (7). It indicates the number of calculated cycles that will be executed in the routine, showing that in all cases there is no error in the calculation, thanks to the compensations described in the previous point with respect to {Q} and including the NOPs added according to the established time. These coefficients were verified by simulation to check their accuracy, finding the expected accuracy in all cases.

TABLE I. TIMING VALIDATION RESULTS (ASSEMBLY IMPLEMENTATION)

Total Time ***(µs)***	**Parameters** ***(R,P,Q)***	***Calculated Cycles***	**Error Measured** ***(ns)***	***Aggregate NOPs***
1	(1,1,2)	16	0	0
1,000	(1,21,196)	$16x10^3$	0	2
5,000	(1,7,123)	$80x10^3$	0	1
50,000	(5,15,240)	$800x10^3$	0	2
100,000	(9,30,228)	$1.6x10^6$	0	0
500,000	(41,150,126)	$8x10^6$	0	2
1,000,000	(82,43,256)	$16x10^6$	0	0

IV. Implementation and Integration

A. Arduino IDE Integration

Assembly routines can be integrated with Arduino C/C++ code using inline assembly as illustrated in Figure 3.

The code executes a routine known as "Blinking LED" or "Flashing LED" on Arduino's terminal D3 [9], managing a 50% duty cycle for taking two delay readings on the same terminal, validated through oscilloscope measurements.

This routine uses a synchronization of 1 second per period, where the coefficients R, P, and Q match the value in Table 1. Note that if any variable in the formula results in 256, it should be taken as 0 for the registers in assembler, since the decrements will be from 0 to 0, executing 256 cycles.

This information enables implementing libraries (delay.h) with functions capable of handling µs or ms, calculating coefficients based on established time without user intervention, building functions such as:

- precise_delay_ms(unsigned long int time);
- precise_delay_us(unsigned long int time);

1) Validation Methodology

Hardware Configuration:

- Arduino Uno (ATmega328P with 16 MHz crystal)
- Oscilloscope for cycle measurement

The C/C++ code shows the loop used to turn the LED on and off at terminal D3 (Fig. 4), establishing this terminal as oscilloscope input for sampling, validating previously calculated coefficients (Fig 5).

```
// C/C++ code

const int ledPin = 3; // LED in D3
void setup()
{
    pinMode(ledPin, OUTPUT);
}

void loop()
{
    digitalWrite(ledPin, HIGH);
    delay_asm(126, 150, 41);     // Wait for 500 ms
    digitalWrite(ledPin, LOW);
    delay_asm(126, 150, 41);     // Wait for 500 ms
}

void delay_asm(uint8_t Q, uint8_t P, uint8_t R)
{
    asm volatile
    (
        "loop1: \n\t"
        "dec %0 \n\t"
        "brne loop1 \n\t"
        "dec %1 \n\t"
        "brne loop1 \n\t"
        "dec %2 \n\t"
        "brne loop1 \n\t"
        "nop \n\t"
        "nop \n\t"
        : "+r" (Q), "+r" (P), "+r" (R) // modified registers
        : // no additional registers
    );
}
```

Fig. 4. Delay algorithm in assembly language, compensated and implemented in Arduino.

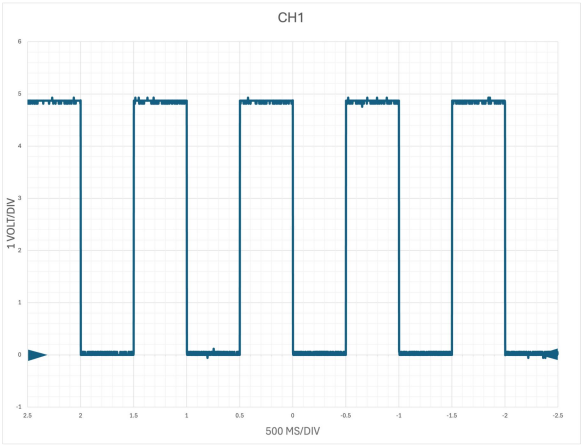

Fig. 5. Oscilloscope validation of timing precision showing an exact 1 second period with 500 ms ON/OFF cycles, measured on a Tektronics TBS1064 oscilloscope set to 1 V/div and 500 ms/div; data extracted from the CSV file exported by the oscilloscope.

V. Applications and Perfomance Analysis

A. Critical Timing Applications

Serial communications via hardware is an application that invariably presents slippage or synchronization errors. This is due to the implementation of hardware synchronization bases for a wide spectrum of transmission speeds, resulting in error rates that can be high or low, with rare cases of 0. A notable advantage of a routine such as the one presented is its ability to discern imperceptible errors (Fig. 6).

```
;*************** Serial Communication ***************
uart_send_bit:
    ; Send start bit
    CBI PORTD, 1 ; Data line low
    CALL delay

    ; Send data bits...
    SBRC R16, 0 ; Skip if bit is clear
    SBI PORTD, 1 ; Data line high
    CALL delay_asm
    ; Continue for remaining bits

delay:
    ; 104 cycles = 6.5µs (153.6kbps)
    LDI R23, 31
    LDI R24, 1
    LDI R25, 1
; …
; ************************************************
```

Fig. 6. Software Serial Communication.

```
;*************** Sensor Communication ***************
dht22_start_signal:
    ; Set data line low for exactly 1ms
    CBI PORTC, 0 ; Data low
    CALL delay1
    ; ...
    ; Set high for 30µs
    SBI PORTC, 0 ; Data high
    CALL delay2
    ; ...
delay1:
    LDI R23, 196 ; Calculated for 1000µs
    LDI R24, 21
    LDI R25, 1
    ; ...
delay2:
    LDI R23, 156 ; Calculated for 30µs
    LDI R24, 1
    LDI R25, 1
    ;…
; ************************************************
```

Fig. 7. Sensor Interface Timing.

The communication process with sensors can be intricate in scenarios where acquiring data rapidly is challenging. This complexity may arise following device enablement or during communication protocols such as I2C or SPI (Fig. 7).

B. Performance Comparison

In light of the critical applications delineated in the preceding section, a series of tests were devised. These tests compared delays provided in the integrated libraries of the Arduino platform with delays presented in this document [10]. The tests evaluated temporal precision, code inversion ratio in these functions, ability to predict results based on stipulated times, and additional code added by the platforms to prepare routines (overhead). The results of these tests are shown in Table 2.

TABLE II. Assembly vs High-Level Language Performance

Implementation	**Timing *Precision***	**Code *Size***	**Predictability**	**CPU Overhead**
Arduino delay ()	1000 ns	Small	Poor	High
delayMicroseconds()	250 ns	Small	Moderate	High
Nested C loops	125 ns	Medium	Compiler dependent	Medium
Assembly	62.5 ns	Small[a] Medium[b]	Excellent	Minimal

[a.] Assembly Code

[b.] Assembly Code into C/C++

C. Assembly Implementation Advantages

Deterministic Execution:

- Known cycle count at assembly time
- No compiler optimization variables
- Consistent timing across different compilations

Maximum Efficiency:

- Direct register use minimizes memory access
- Optimized instruction sequences reduce overhead
- Fine granular control over every cycle

Predictable Behavior:

- Timing independent of compiler settings
- Consistent performance across AVR variants
- Reliable operation under all conditions

VI. Advanced Features and Extensions

A. Dynamic Implementation

Ideally, the programmer only needs to specify the time required for the function as an input parameter. For example:

```
delay_ms(500);
```

This functionality can be implemented as a .h/.c controller within the work platform to facilitate the programmer's use of the function(s) derived from this research.

B. *Interrupt-Safe Implementation*

Fig. 8 illustrates how to implement delay routines with interrupt protection. This prevents the interrupt handling process from altering the timing of the algorithm.

```
;******************** Atomic Delay ********************
atomic_precision_delay:
    CLI ; Disable interrupts
    CALL delay
    SEI ; Re-enable interrupts
    RET
; ************************************************
```

Fig. 8. Interrupt-safe delay implementation

VII. Conclusions and Future Work

This research demonstrates that assembly-level programming provides maximum precision for timing-critical embedded applications. The mathematical modeling approach enables exact delays with cycle precision and nanosecond resolution, eliminating uncertainties inherent in high-level language implementations.

Key achievements include:

- **Perfect timing precision**: ±0 cycle deviation under controlled conditions.
- **Deterministic execution**: Consistent timing regardless of compiler or optimization.
- **Mathematical precision**: Exact calculation of parameter for any timing requirement.
- **Practical implementation**: Seamless integration with Arduino development environment.

The assembly language approach trades development complexity for timing precision, making it ideal for applications where exact timing is critical. While C-based implementations offer easier development, assembly language provides unmatched timing control.

Future research directions include:

- Extension to other AVR microcontroller families
- Integration with real-time operating systems
- Automatic assembly code generation tools for timing routines
- Temperature compensation for crystal frequency variations
- Multicore timing synchronization for advanced AVR devices.

Limitations and considerations:

- Increased development complexity compared to high-level languages
- Platflorm-specific implementation (Arduino/AVR assembly)
- Larger code size for complex timing sequences
- Requirement for assembly language expertise

The methodology provides a solid foundation for applications requiring deterministic timing, such as industrial automation and scientific instrumentation, where precision is critical.

Acknowledgements

The authors thank the AVR development community for comprehensive instruction set documentation and the Arduino project for providing accessible development platforms.

References

[1] Atmel Corporation, *ATmega328P Datasheet Complete*, Rev. 7810D-AVR-01/15, 2015.

[2] T.Wilmshurst, *Designing Embedded Systems with PIC Microcontrollers: Principles and Applications*. Embedded Technology Series, Newnes, 2007.

[3] D. Ibrahim, *Arduino Microcontroller Programming and Interfacing*. Microcontroller Based Applied Digital Control, Wiley, 2021.

[4] B. Huang, D. Ragan, “Arduino-Based Embedded Systems”. IEEE Computer Society, 2019.

[5] M. Morán, A. García, A. Cedano, P. Ventura, “Delays by Multiplication for Embedded Systems: Method to Design Delays by Software for Long Times by Means of Mathematical Models and Methods, to Obtain the Algorithm with Exact Times”, in New Perspectives in Software Engineering, Lecture Notes in Networks and Systems, Springer, vol. 576, pp.272-285, 2023, DOI: 10.1007/978-3-031-20322-0_19.

[6] A. G. Arreola, J. M. Morán Loza, A. C. Rodríguez and P. V. Nuñez, "Method to design delay by software for short times,using models and mathematical methods,to obtain the algorithm with exact times," 2022 11th International Conference On Software Process Improvement (CIMPS), Acapulco, Guerrero, Mexico, 2022, pp. 158-163, doi: 10.1109/CIMPS57786.2022.10035699.

[7] Microchip Technology, *AVR Instruction Set Manual*, DS40002198A, 2020.

[8] R.L. Graham, D.E. Knuth, O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*. 2nd ed., Addison-Wesley, pp.67-70, 1994.

[9] Arduino Development Team, *Arduino Language Reference*, 2025. [Online]. Available: https://www.arduino.cc/reference/en/

[10] AVR-LibC Development Team, AVR-LibC User Manual, 2025. [Online]. Available: https://www.nongnu.org/avr-libc/ or https://github.com/avrdudes/avr-libc/