Worst-Case Execution Time Analysis of an Arduino-based Real-Time System in a CAN bus

Ammar Merazga
Artificial Intelligence and Autonomous Things Laboratory, University of Oum El Bouaghi, Algeria
merazga.ammar@univ-oeb.dz (corresponding author)

Djamel Rahem
Laboratory of Electrical Engineering and Automatic (LGEA), University of Oum El Bouaghi, Algeria
rahem_djamel@yahoo.fr

Fateh Moulahcene
Artificial Intelligence and Autonomous Things Laboratory, University of Oum El Bouaghi, Algeria
micro30.sysfateh@gmail.com

Djemai Kebbal
Traces stands for Research Group on Architecture and Compilation for Embedded Systems, University Paul Sabatier, France
djemai.kebbal@iut-tarbes.fr

Rostom Khelaf
Department of Electronics, Telecommunications and Spatial Telecommunications Engineering, University of Oum El Bouaghi, Algeria
khalef.rostom@umc.edu.dz

Imed Benacer
Artificial Intelligence and Autonomous Things Laboratory, University of Oum El Bouaghi, Algeria
benacerimad@gmail.com

Received: 17 March 2025 | Revised: 8 April 2025 and 13 April 2025 | Accepted: 19 April 2025

 DOI: https://doi.org/10.48084/etasr.10990

ABSTRACT

Real-time systems need communication networks, as they often operate across multiple physical nodes. CAN-BUS is a common field bus used in these systems. Such systems require a time-based analysis to meet key deadlines and ensure system safety. This study designs and implements a distributed embedded motor control system using FreeRTOS over CAN-BUS for real-time operation. A prototype was built with low-cost components such as Arduino, L298N, and an MCP2515 module. A WCET analysis was performed on the system. The system has two CAN nodes connected to a PC via PCAN-USB for testing and analyzing using Busmaster software. The first CAN node controls the DC motor speed using a real-time PID controller, and the other manages the motor speed through CAN. The experimental results of the PID controller showed a low steady-state error of less than 0.3%. As the speed increases, there is less overshoot. The settling time (Ts) is also short, proving the stability of the system. Performance was verified by comparing it with the signals from the Busmaster software. The WCET analysis used the Bound-T tool on the AVR2560 microprocessor (16 MHz without cache or pipeline). The calculated WCETs for three tasks were 1228.88 µs, 210.19 µs, and 2786.25 µs. This work verifies the schedulability of tasks for applications on a FreeRTOS real-time embedded platform. Bound-T is an open-source tool for WCET static analysis that has shown strong potential and can be used to perform precise and reliable temporal analyses.

Keywords-Arduino; CAN-BUS; DC motor; FreeRTOS; MCP2515; PID controller; real-time; WCET analysis

I. INTRODUCTION

Real-time communication is characterized by data processing and transmission within strictly defined and guaranteed timeframes. It can be implemented using two main architectures: centralized or distributed. In a distributed real-time communication system, numerous entities interact across a network while adhering to strict temporal constraints. The correctness of a real-time system depends not only on the accuracy of the computed results but also on the time at which they are produced. Hard real-time systems are those that must consistently meet their deadlines. A key parameter to ensure adherence to temporal constraints is WCET [1], which is defined as the longest execution time expected when the program runs on its target hardware. Knowing WCET is essential in real-time systems, as tasks need to be completed within strict time limits to guarantee the safe operation of the system [2, 3]. The Controller Area Network (CAN) bus plays a central role in industrial and automotive systems by providing reliable communication, bounded latency, and an arbitration mechanism that guarantees bus access. The design of such systems relies on Worst-Case Response Time (WCRT) analysis for messages [1, 4] and WCET analysis for tasks in each node to ensure that time constraints are met. Static timing analysis methods, which estimate WCET without prior execution, can perform this analysis. Most of the existing literature on the topic of WCET analysis in real-time systems predominantly relies on benchmark-based case studies or experimental evaluations. This study aimed to perform a WCET analysis of a low-cost prototype using real-world tasks from a CAN node implemented in a FreeRTOS/Arduino IDE environment. The objectives were: (i) The design and implementation of a real-time DC motor control system through a CAN bus using FreeRTOS, (ii) Develop a PID controller for motor speed regulation, and (iii) Perform a WCET computation for tasks in the node controlling the motor.

A. The CAN Bus

CAN is a serial communication bus, created by Robert Bosch in 1986 at the Society of Automotive Engineers (SAE), and is a standard feature in almost all new European passenger vehicles, reflecting its widespread acceptance throughout the global automotive community [5]. CAN functions as an asynchronous multimaster serial bus, supporting speeds of up to 1 Mbps and its frames carry payloads of up to 8 bytes. Broadcast communication enables any node to function as a master and initiate communication. Only the nodes relevant to the message process it, while others disregard it. CAN relies on bitwise arbitration, where each message is prioritized and determined by its identifier in the message frame (ID field). The message with the lowest identifier value, meaning the highest priority, wins the arbitration and continues its transmission. CAN uses fixed-priority non-preemptive scheduling [6, 7]. Busmaster was used to collect data from the CAN network, which is open-source software that simulates, analyzes, and tests in-vehicle data bus systems such as CAN. This software monitors the CAN bus, logs data, extracts parameters for database storage, and allows users to send custom data to the CAN bus for testing and analysis [8, 9]. This study incorporates a CAN bus as an embedded communication system and uses the CAN Bus Analyzer tool for diagnostics and validation testing.

B. WCET Analysis

The design of embedded systems for real-time applications is a challenging endeavor. Their execution time is a crucial factor in building a reliable real-time system. In such systems, the longest execution time of a program, called WCET, must not exceed the specified time for the task to satisfy the system's timing requirements. Determining the WCET of a program with accuracy is a major challenge, as execution time is not always constant or consistent but varies due to input variability and software/hardware interactions. The primary goal of static analysis is to estimate the WCET of a given task without executing it on the target hardware. The task code is analyzed by examining its control flow graph. The goal is to identify the longest execution path, often using annotations. These control flow data are then combined with an abstract model of the target hardware to obtain an upper bound for the WCET. Static analysis tools estimate this bound for real-time embedded systems and do not rely on program execution, unlike dynamic methods. Static analysis evaluates source or binary code and computes worst-case execution scenarios without requiring target hardware [10, 11]. Among system tools, Bound-T stands out because it deeply analyzes a task's control flow graph and determines the longest execution paths. It also accounts for hardware characteristics, such as caches, pipelines, and memory access times [12]. This study demonstrates the practical application of Bound-T static WCET analysis for Arduino-based applications running on Atmel AVR microcontroller platforms.

C. Related Works

Estimating the WCET of a program remains a persistent challenge. WCET analysis relies on specialized tools such as AIT, Bound-T, Chronos, etc., and this calculation is performed for a specific target processor. In [1], a real-time control system was designed and implemented for an EV prototype. This system was based on a CAN network with four real-time embedded nodes. WCET analysis focused on the messages transmitted over the CAN bus, and algorithms were implemented to calculate the Worst Case Response Time (WCRT) for all messages in the communication system. However, WCET analysis was not performed for CAN node applications. Developers must perform temporal analyses to verify the schedulability of system tasks. In [13] a single-path code approach was presented for real-time quadcopter control. This method significantly simplified the WCET analysis by relying on a single measurement. Execution times on the Patmos processor (80 MHz) were compared to those of other platforms, such as ARM Cortex-A53 and Atmel AVR (16 MHz). The results showed that the single-path version on Patmos achieved WCET performance comparable to that of a superscalar ARM core. This study validated the practical effectiveness of a single-path code for critical applications.

The Bound-T WCET analysis tool was ported to the H8/3292 microprocessor to allow students to perform WCET analysis for real-time systems education. A real-time laboratory framework has been developed, with the Mind platform as the target environment. Bound-T allows the calculation of the

WCET for student application code, for operating system calls, and the generation of illustrative flow graphs of the code [14]. The Platin [15] tool is an open-source framework for WCET analysis across multiple processors, including Patmos, RISC-V, ARM, and AVR. The ATmega1284p [16] implementation of the tool takes advantage of two essential features of the AVR instruction set architecture: deterministic instruction timing and no cache memory, allowing accurate estimation of WCET bounds. Experimental results with the count_negative benchmark from TACLeBench [17] have verified Platin's effectiveness in reducing overestimation. All of the tests on various platforms (Patmos and RISC-V) also demonstrate the framework's applicability in the presence of different hardware constraints. Existing works on WCET analysis cover specialized tools such as Platin and Bound-T for specific processors [1, 14], simplified approaches (e.g., single-path [13]), and multiplatform frameworks (e.g., Platin [15]). This study uses Bound-T for static WCET analysis of real-time CAN tasks on a FreeRTOS/Arduino architecture, while the studies in [1, 13] did not evaluate complete embedded applications.

II. MATERIALS AND METHODS

A. Proposed CAN-Based Control System

The proposed system is built on the CAN bus architecture, utilizing the Arduino Mega and the MCP2525 CAN controller [18] to establish communication between two distinct nodes. The first node, referred to as the speed control node, comprises an Arduino Mega board responsible for driving a 12V DC gear motor [1] and integrates a DC motor with a gearbox to deliver high torque at reduced speed. The second node, called the motor control node, consists of an Arduino Mega board integrated with a variable resistor module. The MCP2515 module enables CAN communication between nodes [19, 20]. This real-time system distributes tasks across two embedded nodes. The first node sets the motor speed using a potentiometer and sends data messages over the CAN bus. The other node receives the target speed and adjusts the motor speed using a real-time PID controller. A PID controller is a feedback loop control mechanism that calculates an error value as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms to minimize error [21, 22]. The system was connected to a PC via PCAN-USB [23] to test and analyze using Busmaster [8]. The CAN network uses a bus topology with 120 Ω termination resistors placed at both ends of the network to mitigate signal reflections. Figure 1 illustrates a schematic of the CAN system.

B. Hardware Components

The system uses a two-node CAN network. The required hardware components were selected to be low-cost. The following subsections provide detailed descriptions of the components and methods used in the two CAN nodes.

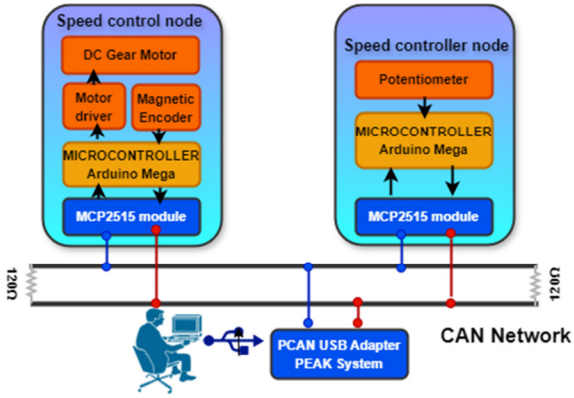

Fig. 1. Block diagram of the proposed system.

1) CAN Node for Speed Control

As shown in Figure 2, the CAN node combines a Microchip MCP2515 controller, a potentiometer, and an Arduino Mega board to control speed. It converts analog signals to RPM, transmits RPM as CAN messages, and receives speed data from the motor control node. This module was implemented on a FreeRTOS-based platform [24].

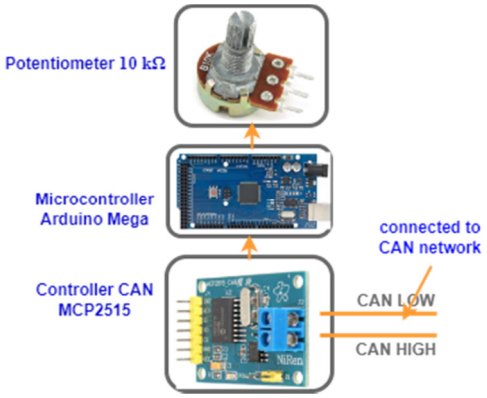

Fig. 2. Schematic diagram of the speed controller node.

2) CAN Node for Motor Control

The goal was to develop a real-time embedded system based on Arduino in the CAN network nodes. This approach allows us to analyze the application's WCET on its microcontroller. The prototype node controls a DC motor's speed using an Arduino board, a 12V motor, a high-power motor driver module to drive DC and Stepper motors (L298N driver) [25], and a Microchip CAN controller. The Arduino sends a PWM signal to control the motor's direction and speed. A PID controller was also implemented in a FreeRTOS-based real-time environment, as shown in Figure 3.

As shown in Figure 4, the DC motor used is the GB37Y3530 12V DC gear motor with a metal gearbox and a 43.8:1 ratio, along with an integrated quadrature encoder. The motor shaft rotates with a resolution of 16 CPR, equivalent to 700 CPR for the gearbox output shaft [1, 21].

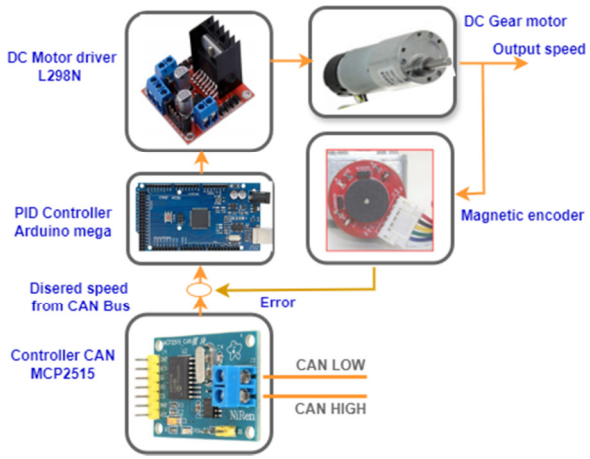

Fig. 3. Schematic of the PID controller for the motor control node.

Fig. 4. 12V DC geared motor/ model JGB37-3530.

TABLE I. SIX COLOR-CODED WIRES OF THE MOTOR

Wire color	Red	Black	Green	Blue	Yellow	White
Function	VCC Motor power 12 v	GND Encoder ground	Encoder signal B output	Encoder power supply	Encoder signal A output	GND Motor ground

PID Controller of DC motor: The angular velocity of the DC motor is calculated with the following two formulas:

$$Angular\ velocity = (ticks \times 2\pi)/(CPR \times Te) \quad (1)$$

$$RPM = (ticks \times 60)/(CPR \times Te)) \text{ e} \quad (2)$$

During rotation, the encoder produces pulses (ticks) over a time interval Te. At the same time, Counts Per Revolution (CPR) denotes the number of encoder pulses per motor shaft revolution, with Te indicating the sample duration. Closed-loop control adjusts a system's input according to its measured output to decrease the error. In a PID controller, a technique employed to optimize this process based on the error, its integral, and its derivative, as shown in the formula below:

$$PID = Kp \times err + Ki \times Integral_{er}$$
$$+Kd(err - prev) \quad (3)$$

where $err = target_{speed} - current_{speed}$, $Integral_{err} = integral_{err} + err$, and $prev = err$. Kp, Ki, and Kd are constants in the PID controller. Equations (1), (2), and (3) were used to implement the PID controller for tuning the speed of the motor in the motor speed control node [1, 21, 25]. A FreeRTOS-based multitasking application was implemented on Arduino for a motor controller node, consisting of three tasks: (i) *wheelSpeedControlTask()* to regulate motor speed, (ii) *sendCANTask()* to transmit messages on the CAN bus, and (iii) *receiveCANTask()* to receive CAN data. The *wheelSpeedControlTask()* algorithm uses a PID controller to tune the speed, running every 100 ms to calculate speed and error using (1), (2), and (3). Adjustments to Kp, Ki, and Kd are needed for real-time stability.

```
Algorithm 1: wheelSpeedControlTask() – PID
Controller for DC Motor Speed Regulation
on FreeRTOS.
1. void wheelSpeedControlTask()
#  Initialization
2.    xFrequency←100ms; currentSpeed←0;
      vSupply←12v
#     Motor velocity between 0... 255
3.    pretime←0; dutyC←20; sumEr←0;
      deltaEr←0;
4.    Kp←6.4;Ki←0.089;Kd←0.0077;
      xFrequency←100 ms
5.    While (1)
6.      read desRPM from xQueueReceive
7.      H-Bridge L298N Activation
8.      analogWrite(MA_Forward,
        duty_cycle);
9.      Vavr ←(dutyC/255.00)*Vsupply
#       Elapsed time and computing RPM
10.     t←millis()-pretime;
11.     RPM←(compteur1/700.0)*60000)/t
12.     er←desRPM -RPM
13.     sumEr←sumEr+(er*t)
14.     sumEr←abs(sumEr)
15.     if(sumEr>2)sumEr←2 endif
16.     deltaEr ← (er -lastEr)
17.     Vavr1←KP*er + KI*(sumEr) +
        KD*(deltaEr)
18.     Vavr1← abs(Vavr1)
19.     if (Vavr1>12) Vavr1←12 endif
20.     if (error>0)
21.       dutyC←dutyC +(Vavr1/Vsupply)*255
22.       if (dutyC>255)dutyC←255 endif
23.     else
24.       if (error<0)
25.         dutyC←dutyC-
            (Vavr1/Vsupply)*255
26.         if (dutyC>0) dutyC1←dutyC
27.         else
28.           dutyC←
              dutyC1(Vavr1/Vsupply)*255
29.         endif
30.       endif
31.     endif
# Reinitializing time and other variables
32:     time1←0; t←0; sumEr←0;
        deltaEr←0;
        compteur1←0 ; pretime←millis();
        currentSpeed←RPM
# Post item (current speed) on a queue
33:     xQueue(currentSpeedQueue,
```

```
       &currentSpeedValue)
34:    delay of 100ms
35:  end while
36:  return 0
```

Algorithm 2: sendCANTask () – FreeRTOS Task for Sending Measured Speed over the CAN Bus.

```
1. void sendCANTask()
# Initialization:
2.   xFrequency<-100 ms;
3.   struct can_frame canMsgTx;
4.   While (1)
# currentSpeedQueue  queue receives DC
# motor speed values (RPM) to be
# transmitted over the CAN bus
5.     if (currentSpeedQueue is not empty)
# Building the CAN Message for
  Transmission
6.       canMsgTx.ID<- 0x021
7.       canMsgTx.DLC<- 4
# Convert Float Value to Bytes
8.       memcpy(adcToBytes,&adcVReceived,
         4);
9.       for (k=0; k<4; k++)
10.        canMsgTx.data[k]←
           adcValueToBytes[k]
11.      endfor
12.      xSemaphoreTake(
         xSemaphore,portMAX_DELAY);
13.      mcp2515.sendMessage(
         &canMsgTx)
14.      xSemaphoreGive(xSemaphore)
15.      vTaskDelay(xFrequency)
16.    endif
17.  end while
18:  return 0
```

Algorithm 3: receiveCANTask() – FreeRTOS Task for receiving target speed over CAN Bus

```
1. void receiveCANTask()
# Initialization
2.   xFrequency←100 ms; float targetSpeed
3.   struct can_frame Rx;
4.   While (1)
# Read Messages Sent on the CAN Bus
5.     if(mcp2515.readMessage(&Rx)==
       MCP2515::ERROR_OK)
# Retrieve Desired Speed from Message's ID
# 0x021
6.       if(Rx.ID == 0x021)
# Convert Bytes to Value
7.         for(k=0; k<4; k++)
8.           speedV[k]<-Rx.data[k]
9.         endfor
10.        memcpy(&targetSpeed, speedV,4);
11.        vTaskDelay(xFrequency)
12.      endif
13     endif
14.  end while
15:  return 0
```

C. WCET Analysis for Atmel AVR Target Processor

Arduino boards are not perfect for real-time systems, but they are low-cost for rapid prototyping. An Arduino microcontroller may build an embedded management system by arranging sensors, actuators, and network interfaces. An AVR ATmega2560 was used to study the WCET of real-time applications.

1) Atmel AVR Timing

The Atmel AVR2560 is an 8-bit microcontroller with 32 general-purpose registers, 135 instructions, and a maximum frequency of 16 MHz. It has 256 KBytes of Flash, 8 KBytes of SRAM, 54 digital I/O pins, 16 analog input channels, and several timers, making it appropriate for real-time embedded applications. The AVR2560 has no cache and a two-stage instruction pipeline, executing most instructions in a single cycle at 16 MHz with a 62 ns cycle time [26].

2) Bound-T WCET Analysis Tool

Bound-T WCET analyzer is a commercially available static analysis tool developed by Tedium Ltd. for real-time software development. Its primary function is to calculate an upper bound on the WCET, and it can also determine an upper bound on stack usage. Bound-T is primarily designed for smaller 8- and 16-bit microcontrollers, but it supports 32-bit processors.

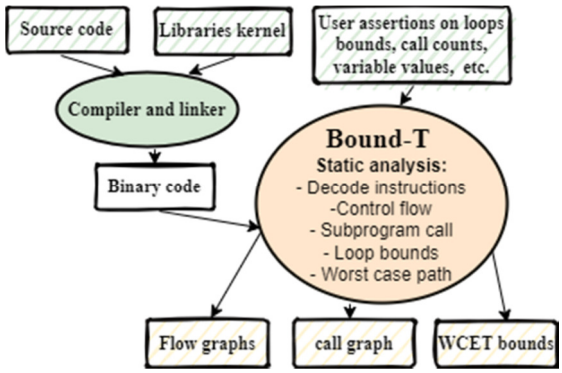

Fig. 5. The Bound-T WCET analysis tool.

The Bound-T execution time and stack usage analyzer is used with various target processors such as Atmel AVR ATmega2560, ARM7 TDMI, Intel 8051 series, and SPARC7/ERC32. As shown in Figure 5, it directly processes binary executables, decoding instructions and automatically generating CFGs and call graphs from the executable. It uses Presburger analysis to calculate loop bounds, resolve dynamic jumps, and calculate stack usage bounds. The Implicit Path Enumeration Technique (IPET) computes the WCET bounds, with timing data on nodes and/or edges within the CFG. Bound-T offers WCET estimates for individual functions and the complete program [12].

III. COMPUTING WCET OF REAL-TIME TASKS

The WCET of the motor control CAN node application was analyzed. The sketch of this implementation was composed of three FreeRTOS tasks, along with the *setup()* function and specific library inclusion instructions, such as FreeRTOS.h, SPI.h, MCP2560.h, semphr.h, and queue.h, and the initialization of global variables. The three implemented FreeRTOS tasks are as follows:

- *wheelSpeedControlTask()*: This task implements the PID controller to regulate motor speed (see Algorithm 1).
- *sendCANTask()*: This task transmits the current speed over the CAN bus. Details were provided in Algorithm 2.
- *receiveCANTask()*: A FreeRTOS task to receive the desired speed via the CAN bus (see Algorithm 3).

The steps for Bound-T WCET analysis are as follows. Generate an executable .elf file from the Arduino sketch for the Atmel AVR2560. The compilation and link files are retrieved, linked to library function calls in the source code file, and used to write assertions. The retrieved files are: SPI.cpp, SPI.h, WInterrupts.c, can.h, heap_3.c, delay.h,, task.h, variantHooks.cpp, -lib1funcs.S, list.c, timers.h, main.cpp, mcp2515.cpp, mcp2515.h, pins_arduino.h, port.c, task.c, wiring.c, portmacro.h, projdefs.h, queue.c, queue.h, semphr.h, stddef.h, stdint.h, timers.c, wiringdigital.c, and wiring_private.h. Bound-T needs loop and function call bounds to compute WCET bounds for the three tasks. The following assertion files were created to achieve this: analogWrite.ass, libs.ass, list.ass, MCP2515.ass, memcpy_4.ass,queue.ass, tasks.ass, receiveCAN.ass, sendCAN.ass, SPI.ass, turnOffPWM.ass, and wheelSpeedControl.ass. The content of the sendCAN.ass assertion file is as follows:

```
Assertions for sendCANTask:
subprogram sendCANTask
   loop
    on line 631 in "mcp2515.cpp"
    --This loop scans the TX buffers to
      find an idle one.
    --There are three such buffers.
      repeats 3 times;
   end loop;
   eternal loop that calls
     "<artificial>|xTaskDelayUntil"
     --This is the eternal job loop, but
     there is another eternal
     --loop (in mcp2515.cpp)in-lined into
     sendCANTask.
     repeats 1 time;
   end loop;
end "sendCANTask";
```

To invoke Bound-T for WCET analysis of the *wheelSpeedControlTask()* function under these assertions, use the following command: *boundt_avr –ATxmega128a1 -assert wheelSpeedControl.ass MCP2515.ass analogWrite.ass queue.ass SPI.ass tasks.ass CAN_NETWORK.v4.ino.elf wheelSpeedControlTask*

The command execution was automated using Bound-T scripts to analyze one task. The script files are ana.sh, ana-assert.sh, receiveCAN.sh, sendCAN.sh and wheelSpeedControl.sh. The content of wheelSpeedControl.sh is as follows.

```
#!/bin/bash
# Analyse wheelSpeedControlTask
./ana-assert.sh                          \
   wheelSpeedControlTask                 \
   -warn no_too_many_sources             \
   -assert wheelSpeedControl.ass         \
   -assert memcpy_4.ass                  \
    $*
```

Bound-T adds three options to generate call graphs and control flow. These graphs are saved in a .dot file. The dot program from the GraphViz package converts these graphs to other formats, such as PDF or PNG [12].

IV. EXPERIMENTAL RESULTS AND DISCUSSIONS

A prototype real-time system was built with two CAN nodes using FreeRTOS to carry out a static WCET analysis. A PCAN-USB adapter was utilized to establish a connection to the CAN network. This adapter was compatible with standard CAN 2.0A/B protocols and functions with CAN network management software, such as Busmaster, for monitoring, debugging, and data analysis. The system used Busmaster to watch and record signals of the CAN network to compare them to signals of the PID controller. Figure 6 shows the proposed system.

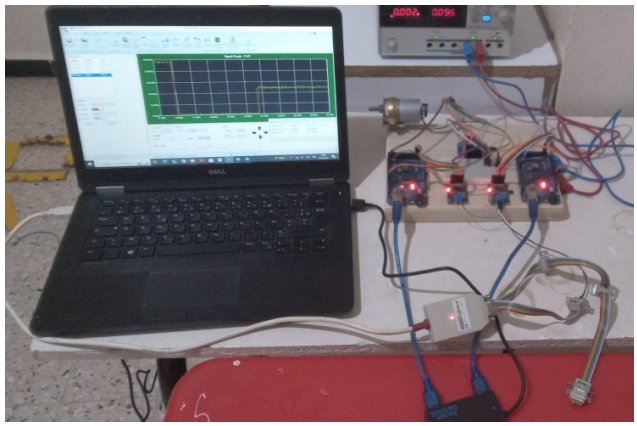

Fig. 6. Experimental setup of the proposed system.

A. Real-Time PID Controller for DC Motor Speed Regulation

Algorithm 3 implements the real-time PID controller as part of the FreeRTOS task in wheelSpeedControlTask. This controller will adaptively correct and adjust the speed of the DC motor to the desired setpoint received over the CAN bus. The PID gains were determined using the trial and error method. The proportional gain Kp was initially increased from 0.01 to 6.4 to enhance the rising time and diminish overshoot. After optimizing this parameter value, the focus was on correcting the steady-state error (Ess) by progressively

adjusting the integral gain Ki. Initially set at 0.001, the latter was increased to 0.0086 to minimize the stationary error. Finally, Ki was reduced to an optimal value of 0.0076 to limit oscillations and improve system stability. Three target speeds were used: 60, 120, and 180 RPM. After many experiments with the process and fine-tuning the PID parameters, the best gains were found to be $Kp = 0.64$, $Ki = 0.089$, and $Kd = 0.0077$. The resulting system showed very slight oscillations and one of the best stabilities, as shown in Figure 7.

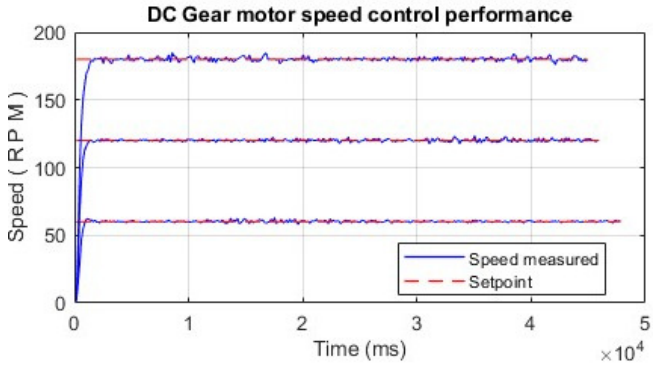

Fig. 7. DC motor with PID controller at 60, 120, and 180 RPM.

As shown in Table II, higher target speeds increase both the temporal complexity of the system and the rise time. Unusually, the stabilization time at 60 RPM shows substantial variation, diverging from the patterns seen at different speeds. Despite all these variations, the steady-state error of the system in all cases is fairly low and remains around (<0.3%) indicating the robustness of the system. The best case is that the maximum overshoot reduces with increasing speed. This means that there is less variance in the system response, leading to improved stability. Compared to the results of the position regulation from [21] (which imposes on a higher degree setpoint) with ESS<0.85 and [1] with ESS<0.65, this system performs much better with ESS<0.3, indicating a good acceptable quality of control.

TABLE II. PID CONTROLLER WITH GAINS K_p = 6.4, K_i = 0.089, K_d = 0.0077 AT DIFFERENT TARGET SPEEDS

Controller Gain	Target speed (rpm)	Tr(ms)	Ts(sec)	Ess(%)	Mp(%)
$Kp = 6.4$	60	441	17.5	0.2	5.17
$Ki = 0.089$	120	558	0.9	0.3	2.97
$Kd = 0.0077$	180	735	1.05	0.13	2.84

Figure 8 shows input values varying from 60 to 180 RPM with a reverse transition. Red curves are for the system outputs while blue curves are for the setpoints. Results analysis shows that the system maintains a stable performance over responsiveness. Even further, maximum overshoot remains very low, and there is practically a minimal steady-state error, further affirming the PID controller's extreme effectiveness.

The node controlling the motor transmits the measured speed over the CAN bus. In turn, the speed control node also sends a target speed message on the same bus. The CAN bus is interfaced with the BUSMASTER software via the PEAK USB interface. The graph in Figure 9 depicts the evolution of two signals, the target speed and the desired speed of a DC gear motor. This visualization is similar to the one in Figure 8 and makes it easier to perform a direct comparison of how the system performs [9, 27].

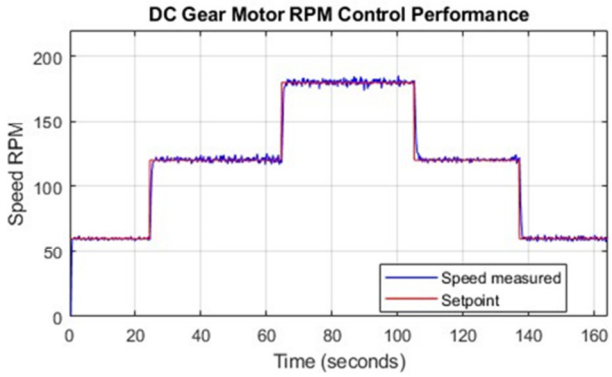

Fig. 8. System response to speed (60-180 RPM) and reverse transitions.

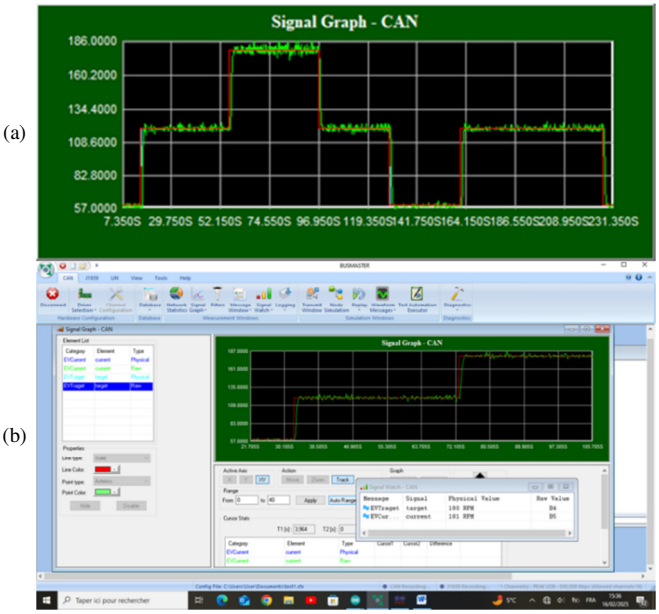

Fig. 9. CAN message signals of target speed (60-180 RPM) and measured speed acquired from the CAN network using Busmaster.

B. WCET Analysis Results with Bound-T

Bound-T was employed for the WCET static analysis, adding assertions to help it find bounds on the loops. Scripts were used to automate all Bound-T commands, allowing the computation of WCET for the three tasks, the generation of summary tables detailing the execution times of various called library functions, and the construction of their call graphs. To calculate the WCET bound for one job of each task (one iteration of the eternal task loop), each task was analyzed twice: first with an assertion that the loop repeats once, then with an assertion that the loop repeats twice, and then the difference between the WCET bounds for one iteration and two iterations. It was difficult to understand the looping logic in the library routines for floating-point calculations, such as

__addsf3, *__subsf3*, *__mulsf3*, and *__divsf3*, as such routines are often written in assembly languages and use tricky manual optimizations. The source code for the libs helped in this task.

1) WCET of receiveCANTask()

The data in Table III present the WCET analysis results for the *receiveCANTask()* task obtained using Bound-T for a single iteration. Figure 10 shows the task's call graph, analyzed for two iterations. As shown in Table III and Figure 10, the total execution time is as follows: 2952+199+95+30+16+71=3363 cycles.

TABLE III. SUBPROGRAMS EXECUTION TIME (CLOCK CYCLES) IN THE WORST-CASE SCENARIO OF *receiveCANTask*()

Total time	Self-time	Num calls	Time per call		Subprogram
			Min	Max	
3363	71	1	3363	3363	*receiveCANTask*
2952	77	1	2952	2952	*xTaskDelayUntil*
2090	1574	1	2090	2090	*xTaskResumeAll*
900	900	2	450	450	*vPortYield*
325	113	1	325	325	*prvAddCurrentTaskToDelayedList*
199	199	1	199	199	*MCP2515.startSPI*
145	145	1	145	145	*vListInsert*
95	95	1	95	95	*MCP2515.endSPI*
67	67	1	67	67	*uxListRemove*
66	66	2	33	33	*prvResetNextTaskUnblockTime*
30	30	2	15	15	*SPIClass.transfer*
16	16	1	16	16	*xTaskGetTickCount*
10	10	1	10	10	*vTaskSuspendAll*

TABLE IV. SUBPROGRAMS EXECUTION TIME (CLOCK CYCLES) IN THE WORST-CASE SCENARIO OF *sendCANTask*()

Total time	Self time	Num calls	Time per call		Subprogram
			Min	Max	
19662	389	1	19662	19662	*sendCANTask*
10356	402	1	10356	10356	*xQueueReceive.constprop.3*
8360	6296	4	2090	2090	*xTaskResumeAll*
4050	4050	9	450	450	*vPortYield*
3028	364	2	1514	1514	*prvUnlockQueue*
2952	77	1	2952	2952	*xTaskDelayUntil*
2886	2886	13	222	222	*xTaskRemoveFromEventLis*
1516	160	4	379	379	*MCP2515.readRegister*
1300	452	4	325	325	*prvAddCurrentTaskToDelayedList*
1194	1194	6	199	199	*MCP2515.startSPI*
870	870	6	145	145	*vListInsert*
755	316	1	755	755	*MCP2515.setRegisters*
480	480	32	15	15	*SPIClass.transfer*
401	142	1	401	401	*MCP2515.modifyRegister*
380	380	4	95	95	*MCP2515.endSPI*
358	341	1	358	358	*__fixunssfsi*
268	268	4	67	67	*uxListRemove*
264	264	8	33	33	*prvResetNextTaskUnblockTime*
134	134	2	67	67	*xTaskCheckForTimeOut*
88	88	2	44	44	*memcpy*
40	40	4	10	10	*vTaskSuspendAll*
36	36	2	18	18	*prvIsQueueEmpty*
17	17	1	17	17	*__fp_splitA*
16	16	1	16	16	*xTaskGetTickCount*

2) WCET of sendCANTask()

The WCET for one iteration of the *sendCANTask*() function is 19,662 cycles. This time is distributed among various subprograms, as shown in Table IV and Figure 11. The total WCET is decomposed based on the execution time of the called subprograms as follows: 16+10.356+44+650+2.952+ 2,090+10+450+358+1.516+401+755+389=19662 cycles.

3) WCET of wheelSpeedControlTask()

The WCET for one iteration of this function is 44,580 cycles. As shown in Table V and Figure 12, the detailed breakdown of the WCET is as follows: 16+267+378+132+900+2952+10542+200+5+100+5+6+3+196 +147+28731(selftime)=44580 cycles.

TABLE V. SUBPROGRAMS EXECUTION TIME(CLOCK CYCLE) IN THE WORST-CASE SCENARIO OF *wheelSpeedControlTask*()

Total Time	Self Time	Num Calls	Time Per Call		Subprogram
			Min	Max	
44580	28731	1	44580	44580	*wheelSpeedControlTask*
10542	601	1	10542	10542	*xQueueGenericSend.constpr*
6270	4722	3	2090	2090	*xTaskResumeAll*
4050	4050	9	450	450	*vPortYield*
3028	364	2	1514	1514	*prvUnlockQueue*
2952	77	1	2952	2952	*xTaskDelayUntil*
2886	2886	13	222	222	*xTaskRemoveFromEventLis*
975	339	3	325	325	*prvAddCurrentTaskToDelayedList*
725	725	5	145	145	*vListInsert*
378	20	1	378	378	*__fixsfsi*
358	341	1	358	358	*__fixunssfsi*
268	268	4	67	67	*uxListRemove*
267	267	3	89	89	*digitalWrite*
259	259	7	37	37	*__fp_cmp*
200	200	2	100	100	*__floatunsisf*
198	198	6	33	33	*prvResetNextTaskUnblockTime*
196	48	4	49	49	*__gtsf2*
147	36	3	49	49	*__lesf2*
134	134	2	67	67	*xTaskCheckForTimeOut*
132	132	2	66	66	*pinMode*
100	100	1	100	100	*__floatsisf*
30	30	3	10	10	*vTaskSuspendAll*
17	17	1	17	17	*__fp_splitA*
16	16	1	16	16	*xTaskGetTickCount*
6	6	6	1	1	*__mulsf3*
5	5	5	1	1	*__subsf3*
5	5	5	1	1	*__divsf3*
3	3	3	1	1	*__addsf3*

The static WCET analysis allowed computing the upper bounds of the WCET on the AVR2560 processor (running at 16 MHz without cache or pipeline) for the three critical tasks of the PID controller CAN node. Table VI presents the obtained results.

TABLE VI. WCET OF PID CONTROLLER CAN NODE TASKS

Task	Clock cycles	Time (µs)
receiveCANtASK	3363	210.19
sendCANTask	19662	1228.88
wheelSpeedControlTask	44580	2786.25

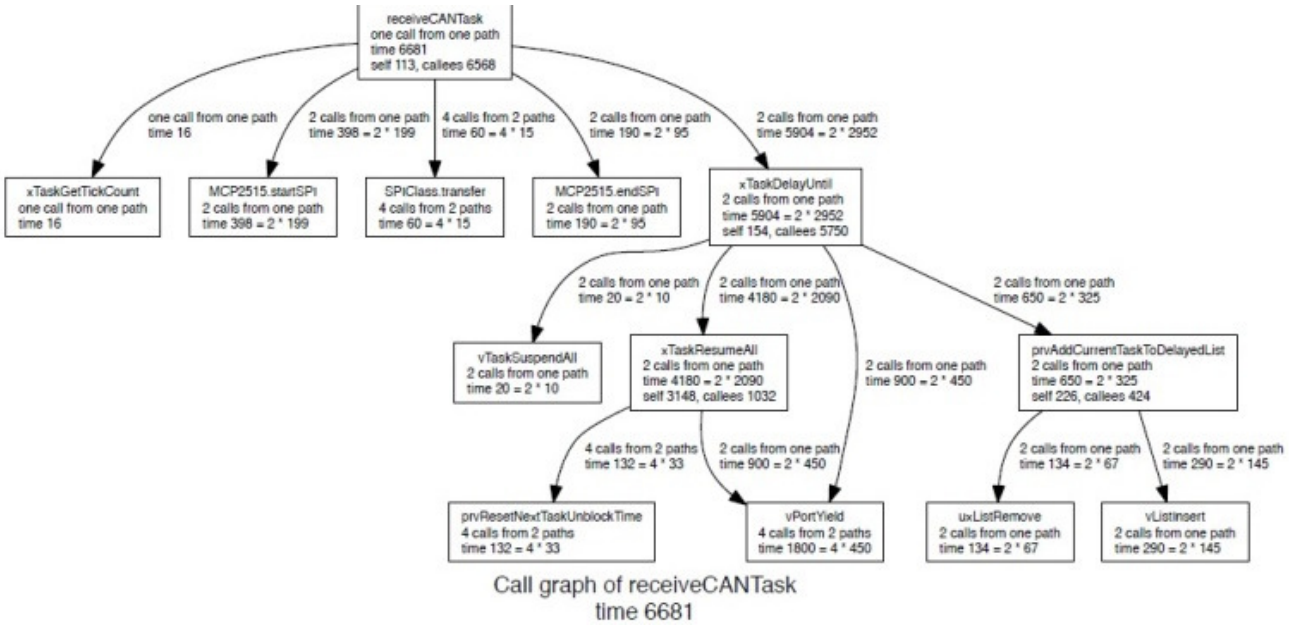

Fig. 10. Call graph of *receiveCANTask*.

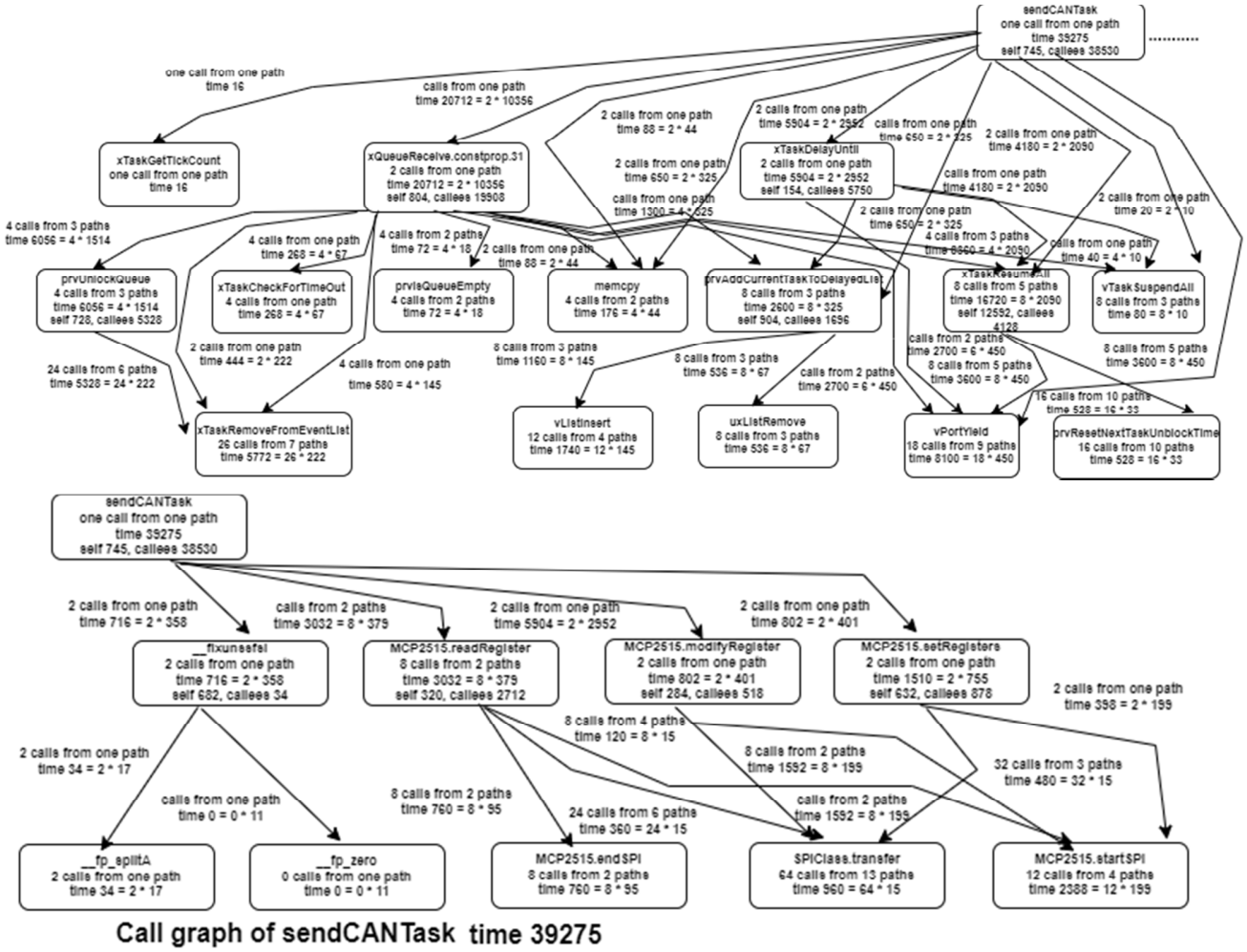

Fig. 11. Call graph of *sendCANTask*.

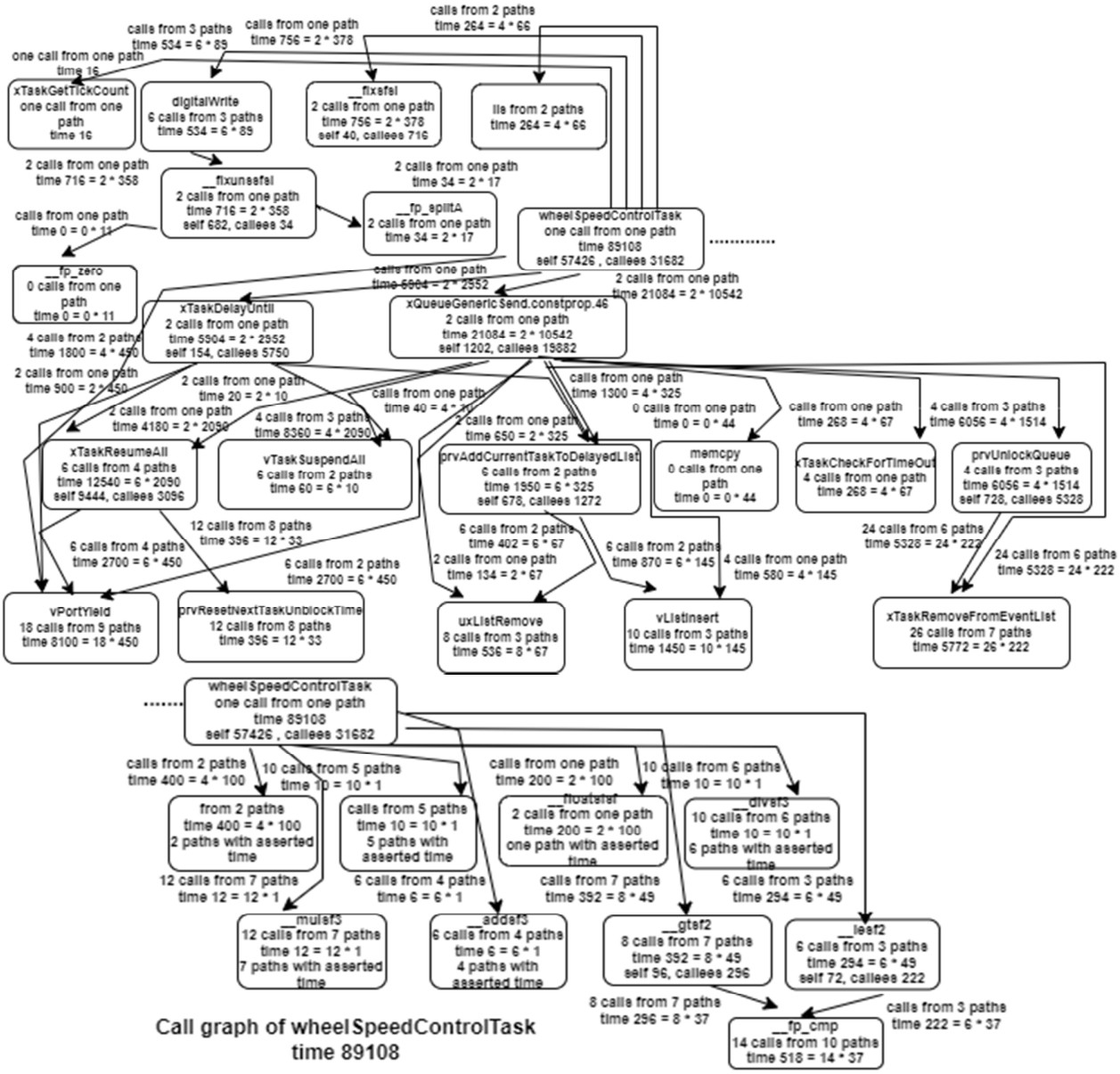

Fig. 12. Call graph of *wheelSpeedControl.*

C. Validation

Measurements can provide support for the validation of static analysis methods. The fundamental way for measuring run time using software instrumentation consists of injecting code at pre-recorded locations in the program, generally at the entry and exit points of functions or within loops [3]. This was applied to three FreeRTOS tasks, namely *sendTask*, *receiveTask*, and *wheelSpeedControlTask*, where FreeRTOS timestamps (*xTaskGetTickCount()*) are placed at the beginning and the end of each task's main loop. This measurement allows a very fine quantification of the runtime for each task while being minimally invasive to the flow of execution, as shown in Figure 13. To compare static timing analysis tools (Bound-T vs. Platin) for AVR platforms, two key dimensions require evaluation: (i) timing accuracy and (ii) application context. Table VII shows that Bound-T uses a task-level method that works well in FreeRTOS settings, while Platin uses a detailed instruction-level model. The comparative evaluation of Bound-T and Platin for WCET estimation on AVR platforms reveals complementary methodological approaches. Bound-T's task-level conservative analysis shows greater pessimism at 20-30%, while Platin's cycle-accurate modeling reduces this margin to 6-10% for algorithmic tasks. For other tasks, both tools exhibit comparable pessimism. This is in contrast to Bound-T's better performance on complex interrupt-driven systems compared to Platin's accuracy for bare-metal code analysis. This very strongly encourages exploiting a hybrid approach that can mold an optimal timing analysis for safety-critical AVR applications by the strengths of these tools.

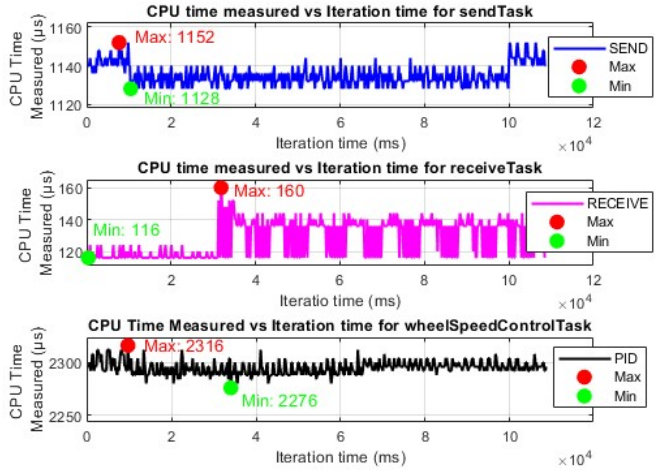

Fig. 13. Runtime measurement of FreeRTOS tasks in a CAN-based motor control node.

TABLE VII. COMPARATIVE ANALYSIS OF WCET TOOLS FOR AVR MICROCONTROLLERS: BOUND-T VS. PLATIN

Aspect	Bound-T/AVR	Platin/AVR
Target hardware	ATmega2560 (Atmel AVR (8-bit)	ATmega1284p (Atmel AVR (8-bit)
Timing model	Path-based worst-case	Cycle-accurate
Cache handling	N/A (No cache)	Explicitly excluded)
Analysis scope	Task-level WCET	Instruction-level WCET
Benchmarks/task application	Real-world tasks from CAN node	TACLeBench
Language	Arduino (C++ with libraries)	Standard C

TABLE VIII. COMPARATIVE EVALUATION OF BOUND-T AND PLATIN FOR WCET ESTIMATION

Bound-T/AVR			
Task	**WCET Bound**	**Measured**	**pessimism**
receiveCANTask	210.19 µs	160 µs	31.37%
sendCANTask	1228.88 µs	1152 µs	6.67%
wheelSpeedControlTask	2786.25 µs	2316 µs	20.3%
Platin/AVR			
count_negative(*TACLeBench benchmark*)	24009 cycles	22560 cycles	6%

V. CONCLUSION

This study designed and implemented a real-time embedded system based on FreeRTOS over CAN-BUS. The system consists of a two-node CAN network, with one node acting as a controller and the other managing a DC motor. The proposed system uses low-cost components, including an Arduino Mega, an MCP2515 CAN controller, an L298N motor driver, and a DC motor, on a FreeRTOS-based Arduino platform. A PID controller was implemented to regulate the speed of the DC gear motor within a real-time task. The results showed a low steady-state error of less than 0.3%, the settling time (Ts) was short, and there was less overshoot as speed increased, proving that the system is stable. The performance of the proposed system was verified by comparing it with Busmaster software signals. The AVR2560 processor operates at 16MHz, has a RISC architecture, and does not include a cache or a pipeline. The optimized compiler inlines function calls, helping to optimize most of the code. This analysis required source code and assertion files for loop bounds. The WCET results for the three tasks were 1228.88 µs, 210.19 µs, and 2786.25 µs, showing the effectiveness of Bound-T in setting an upper limit on task duration for processors such as AVR2560. Future work includes analyzing the stack usage of embedded programs to determine the worst-case stack usage for application tasks, find stack overflows or provide formal proof of their absence, and verify the schedulability of these tasks on this target platform. Bound-T's WCET analysis was conservative by (i) treating inline functions conservatively and (ii) providing floating point emulation (15% overestimation). However, the limitations point to precision trade-offs of static analysis in real-time systems, while providing temporal safety. The WCET analysis with Bound-T proved to be minimally conservative because (i) its handling of inline functions remains cautious and (ii) its floating-point emulation introduces approximations. However, its limited precision does not undermine its effectiveness as a powerful static analysis tool for critical systems.

ACKNOWLEDGMENT

This research was supported by the assistance of Mr. Niklas Holsti from Tidorum Ltd. His expertise and contributions were invaluable in this work.

REFERENCES

[1] M. Ammar, R. Djamel, M. Fateh, and K. Djemai, "Analyzing real-time communication: Arduino-based Controller Area Network (CAN) in electric vehicle," *Studies in Engineering and Exact Sciences*, vol. 5, no. 2, Dec. 2024, Art. no. e11618, https://doi.org/10.54021/seesv5n2-637.

[2] M. Li, K. Xiao, Y. Zhou, and D. Huang, "WCET Analysis Based on Micro-Architecture Modeling for Embedded System Security," *Applied Sciences*, vol. 14, no. 16, Jan. 2024, Art. no. 7277, https://doi.org/10.3390/app14167277.

[3] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, Apr. 2008, https://doi.org/10.1145/1347375.1347389.

[4] A. Berisa *et al.*, "Comparative Evaluation of Various Generations of Controller Area Network Based on Timing Analysis," in *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sinaia, Romania, Sep. 2023, pp. 1–8, https://doi.org/10.1109/ETFA54631.2023.10275549.

[5] "CAN knowledge." https://www.can-cia.org/can-knowledge.

[6] R. Cantoro, S. Sartoni, and M. S. Reorda, "In-field Functional Test of CAN Bus Controllers," in *2020 IEEE 38th VLSI Test Symposium (VTS)*, San Diego, CA, USA, Apr. 2020, pp. 1–6, https://doi.org/10.1109/VTS48691.2020.9107628.

[7] H. M. Boland, M. I. Burgett, A. J. Etienne, and R. M. Stwalley Iii, "An Overview of CAN-BUS Development, Utilization, and Future Potential in Serial Network Messaging for Off-Road Mobile Equipment," in *Technology in Agriculture*, F. Ahmad and M. Sultan, Eds. IntechOpen, 2021.

[8] "Vision Systems GmbH - Data Communication,Network Device Server,Industrial PC-Systems und Panel PCs - Norderstedt." https://www.visionsystems.de/news20121112.htm.

[9] G. S. Sumith, R. Bhatt, L. Shrinivasan, Y. Bagri, and S. V. Shrinivas, "Digital Dashboard for Electric Vehicles," in *2022 IEEE 4th International Conference on Cybernetics, Cognition and Machine Learning Applications (ICCCMLA)*, Goa, India, Oct. 2022, pp. 413–418, https://doi.org/10.1109/ICCCMLA56841.2022.9989314.

[10] R. Kirner and P. Puschner, "Classification of WCET Analysis Techniques," in *Eighth IEEE International Symposium on Object-*

Oriented Real-Time Distributed Computing (ISORC'05), Seattle, WA, USA, 2005, pp. 190–199, https://doi.org/10.1109/ISORC.2005.19.

[11] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper, "Applying Static WCET Analysis to Automotive Communication Software," in *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, Palma de Mallorca, Spain, 2005, pp. 249–258, https://doi.org/10.1109/ECRTS.2005.7.

[12] "Bound-T time and stack analyser." http://www.bound-t.com/.

[13] M. Platzer and P. Puschner, "A Real-Time Application with Fully Predictable Task Timing," in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, Nashville, TN, USA, May 2020, pp. 43–46, https://doi.org/10.1109/ISORC49007.2020.00016.

[14] S. Petersson, A. Ermedahl, A. Pettersson, D. Sundmark, and N. Holsti, "Using a WCET Analysis Tool in Real-Time Systems Education," *OASIcs, Volume 1, WCET 2005*, vol. 1, pp. 29–32, 2007, https://doi.org/10.4230/OASICS.WCET.2005.812.

[15] E. J. Maroun *et al.*, "The Platin Multi-Target Worst-Case Analysis Tool," *OASIcs, Volume 121, WCET 2024*, vol. 121, 2024, Art. no. 2, https://doi.org/10.4230/OASICS.WCET.2024.2.

[16] *AVR® Instruction Set Manual*. Microchip Technology Inc., 2021.

[17] H. Falk *et al.*, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research," *OASIcs, Volume 55, WCET 2016*, vol. 55, 2016, https://doi.org/10.4230/OASICS.WCET.2016.2.

[18] "MCP2515," *Microchip Technology*. https://www.microchip.com/en-us/product/mcp2515.

[19] Y. Wang, "Design of Electric Drive System of Electric Vehicle Based on CAN Bus," *Journal of Physics: Conference Series*, vol. 1982, no. 1, Jul. 2021, Art. no. 012131, https://doi.org/10.1088/1742-6596/1982/1/012131.

[20] A. A. Sase, Y. K. Bhateshvar, and K. C. Vora, "Electric Vehicle Control System by using Controller Area Network Communication," *International Journal of Engineering Sciences*, vol. 15, no. 2, 2022, https://doi.org/10.36224/ijes.150205.

[21] F. Moulahcene, R. Khalef, I. Benacer, A. Merazga, H. Laib, and S. Hanfoug, "Real-Time Speed Control of DC Gear-Motor Using PID Controller and Sliding Mode Control for Robotics," in *2024 1st International Conference on Electrical, Computer, Telecommunication and Energy Technologies (ECTE-Tech)*, Oum El Bouaghi, Algeria, Dec. 2024, pp. 1–6, https://doi.org/10.1109/ECTE-Tech62477.2024.10851156.

[22] A. W. Nasir, I. Kasireddy, and A. K. Singh, "Real Time Speed Control of a DC Motor Based on its Integer and Non-Integer Models Using PWM Signal," *Engineering, Technology & Applied Science Research*, vol. 7, no. 5, pp. 1974–1979, Oct. 2017, https://doi.org/10.48084/etasr.1292.

[23] "PCAN-USB: PEAK-System." https://www.peak-system.com/PCAN-USB.199.0.html?L=1.

[24] "FreeRTOS™ - FreeRTOS™." https://freertos.org.

[25] P. Peerzada, W. H. Larik, and A. A. Mahar, "DC Motor Speed Control Through Arduino and L298N Motor Driver Using PID Controller," *International Journal of Electrical Engineering & Emerging Technology*, vol. 4, no. 2, pp. 21–24, Dec. 2021.

[26] S. Chakraborty and P. S. Aithal, "Conveyor Belt Speed Control Through CAN BUS in CoppeliaSim using Arduino Mega2560," *International Journal of Case Studies in Business, IT, and Education*, pp. 194–201, Apr. 2022, https://doi.org/10.47992/IJCSBE.2581.6942.0159.

[27] H. Kashif, G. Bahig, and S. Hammad, "CAN bus analyzer and emulator," in *2009 4th International Design and Test Workshop (IDT)*, Riyadh, Saudi Arabia, Nov. 2009, pp. 1–4, https://doi.org/10.1109/IDT.2009.5404142.