CHALMERS
UNIVERSITY OF TECHNOLOGY

Embedded Control Firmware Optimization for Power Electronics

Master's thesis in Embedded Electronic System Design

Zhuoer Shao

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025

Master's thesis 2025

Embedded Control Firmware Optimization for Power Electronics

Zhuoer Shao

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2025

Embedded Control Firmware
Optimization for Power Electronics
Zhuoer Shao

Supervisor: Lena Peterson, Department of Microtechnology and Nanoscience
Company advisor: Neetigya Abichandani, Volvo Cars
Examiner: Per Larsson-Edefors, Department of Microtechnology and Nanoscience

Master's Thesis 2025
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2025

Embedded Control Firmware
Optimization for Power Electronics
Zhuoer Shao
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

Modern power converters face quicker input and load changes. With higher switching frequency and smaller inductors or capacitors, there is less stored energy to smooth disturbances. If the control reacts slowly, voltage or current overshoots or undershoots will take longer to settle, resulting in energy waste and potential damage to the device. Therefore, a faster response is needed in the power electronics system.

Embedded control firmware plays a key role in improving closed-loop speed and system stability. In high-frequency DC-DC converters and automotive power electronics, firmware execution performance directly affects control accuracy, energy efficiency, and system robustness. In this thesis, we compare three automotive-grade MCUs—TI F29H85x, TI AM263x, and Infineon AURIX TC4x—under a unified closed-loop control framework. By dividing the control loop into stages such as ADC sampling, PID calculation, and PWM output, and by analyzing differences in interrupt systems, CPU architecture, peripheral interconnect, and compiler optimization, we systematically show their impact on execution delay. Delay is measured using GPIO toggling and interrupt timestamps, and platform-specific optimizations (such as DMA acceleration, early interrupt mode, memory mapping, compiler tuning, and CDSP/PPU offloading) are applied to explore the shortest possible execution time. Results show that all three MCUs achieved significant improvements over their baselines, with F29H85x reaching 710 ns, AM263x 793 ns, and TC4x 750 ns.

The contribution of this project is not only to compare real-time performance across MCUs, but also to propose a unified cross-platform analysis method. By linking experimental results with structural differences, we show how interrupt paths, CPU pipelines, and peripheral interconnects determine real-time performance. This approach goes beyond single-platform studies, providing a systematic framework for analysis. It also offers practical guidance for MCU selection and firmware optimization in industrial applications.

Keywords: embedded control firmware; real-time performance; automotive microcontroller; DC-DC converter

Acknowledgements

First, I would like to express my gratitude to my supervisor, Lena Peterson, for her attentive guidance and valuable advice throughout this research. Her support has not only provided clear direction for my academic work but has also greatly inspired my approach to research and problem-solving.
I am also deeply grateful to my examiner, Per Larsson-Edefors, for his patient guidance and suggestions on scientific writing and research perspectives, which have greatly improved the presentation of my work.
My sincere thanks go to the Power Conversion team at Volvo Cars for providing the research topic and experimental platform for this thesis. In particular, I would like to thank Neetigya Abichandani and Rao Narendar for their assistance in helping me adapt to working in a real industrial environment and for deepening my understanding of the application of embedded control systems in the automotive field.
Lastly, I am truly grateful to my family for their unwavering understanding and support. Their encouragement gave me the courage to embark on my journey at Chalmers and to persevere to this stage.

Zhuoer Shao, Gothenburg, November 2025

Contents

1

Introduction

With the rapid improvement of power electronics, efficient and reliable control systems have become essential in various applications, including renewable energy, electric vehicles, industrial automation, and smart grids [1]. As these systems demand higher power efficiency, lower energy losses, and faster dynamic responses, the role of closed-loop control has become increasingly critical. In power electronic systems, the response time of closed-loop control systems (such as DC-DC and AC-DC converters) directly impacts system stability and real-time performance [2]. Long response time can lead to overshoot, oscillations, or instability in high-frequency converters, particularly in emerging technologies such as SiC and GaN-based power devices [3]. To mitigate these issues, improving the speed of the control loop becomes essential. A faster control loop response also enables the use of higher switching frequencies to improve transient performance and power density. Therefore, optimizing the response time of control loops can better accommodate dynamic input and output requirements, ensuring stable and efficient system operation.

In safety-critical applications such as electric vehicles, microcontroller units (MCUs) with integrated hardware functionalities and excellent real-time performance are widely used for closed-loop system design in the powertrain systems, advanced driver assistance systems (ADASs) [4]. Modern MCUs offer advanced digital control techniques, which enable better precision, adaptive tuning, and integration of safety mechanisms [5]. In these systems, the control firmware includes key control algorithms and coordinates hardware peripherals, enabling efficient system operation. However, implementing efficient firmware must balance execution speed, energy efficiency, and computational complexity, while also considering real-world constraints such as electromagnetic interference (EMI) and thermal management [6]. Moreover, advancements in functional safety standards such as automotive safety integrity level (ASIL) impose additional constraints on MCU-based controllers [7], requiring robust fault detection and redundancy mechanisms in critical applications. Therefore, choosing MCUs with suitable architectures and peripheral features and applying execution-time optimizations can significantly reduce response time and improve the stability of the control loop.

In our project, we compare several automotive-grade MCUs under a unified control loop to investigate how MCU selection and firmware optimization affect execution speed. Furthermore, we link these findings to energy efficiency, system robustness,

and compliance with industrial standards, providing not only practical guidance for MCU selection but also contributing to the development of more sustainable and high-performance embedded power control systems.

1.1 Related Work

In high-frequency power electronic systems, real-time MCUs have been widely applied in closed-loop control tasks. For instance, the TI C2000 series MCUs are often combined with MATLAB/Simulink to quickly build and verify real-time control loops [8]. Such studies demonstrate the practicality and widespread use of MCUs in industry, but they mainly emphasize feasibility validation and toolchain integration rather than deeper performance tuning.

At the same time, traditional closed-loop designs often struggle to meet strict response time requirements due to computational delays and hardware limitations. This highlights the importance of not only using real-time MCUs, but also optimizing control firmware algorithms and leveraging advanced microcontroller architectures to further improve speed, precision, and stability [9]. In recent research [10], the importance of firmware design is emphasized in achieving fast response times. To address the limitations of traditional methods, such as parabolic current control (PCC), which suffers from slow convergence and high computational delays, Zhang's team proposed a single-step current control (SSCC) strategy. SSCC achieves rapid current convergence within a single PWM cycle through dual current sampling and dynamic duty cycle adjustment. Implemented on a microcontroller, the SSCC algorithm reduced response times to less than 1 µs and demonstrated improved stability under dynamic load and input variations.

Similarly, some research [11] explored the role of advanced microcontroller hardware in improving real-time performance. Stolyarov et al. proposed optimizing current loop performance using a control law accelerator (CLA) in microcontrollers. By offloading critical tasks, such as control calculations and pulse width modulation (PWM) duty cycle adjustments, from the central processor unit (CPU) to the CLA, the response time was reduced significantly. This study highlighted the importance of hardware architecture and peripheral integration in high-performance control loops. Dual optimization approaches also provide efficient methods for algorithms and peripherals selection [12]. On the firmware side, adaptive synchronous rectifier (SR) driving logic and algorithm optimization reduced CPU utilization from 42% to less than 6%. By leveraging hardware acceleration for ripple detection and combining it with optimized firmware logic to update PWM, the study achieved a cost-effective solution.

The above methods focus on firmware optimization and hardware acceleration, providing complementary strategies for improving control performance. Building on these studies, although many have focused on firmware optimization and functional safety, most of them concentrate on a single MCU platform or general methods, lack-

ing a systematic comparison of how different MCU architectures, peripheral designs, and interrupt systems affect the execution performance of closed-loop control.

1.2 Purpose and Goal

Our project combines the two perspectives above to evaluate performance across various MCU platforms, and takes a step further by analyzing how the underlying hardware structures of different MCUs impact real-time performance. Specifically, we examine peripheral design like analog to digital converter (ADC) and PWM, CPU architecture (pipeline, cache, floating point unit (FPU), hardware accelerators), and interrupt systems, and study their impact on execution time in closed-loop control tasks. By building a unified control loop and measurement method, we can carry out a cross-platform comparison of automotive-grade MCUs from different vendors, and explore their shortest achievable execution times using platform-specific optimization mechanisms and compiler settings. The specific goals of this study include:

- To implement the same closed-loop control structure on different automotive-grade MCUs, ensuring a fair baseline for comparison.
- To measure the baseline performance of each MCU without optimization, providing an evaluation of their native execution speed.
- To apply platform-specific optimizations and compiler optimizations to explore the shortest achievable execution time.
- To compare baseline and optimized results across platforms, highlighting how peripheral design, CPU architecture, memory system, and development tools affect control loop performance.

Since the project involves the use of multiple MCUs, to achieve the goals above, this project faces several key challenges and tasks to be addressed.

- Comparing various MCU architectures and configurations to study the impact of specific features on system performance.
- Implementing control firmware to achieve a balance between algorithmic complexity and hardware-specific constraints, memory usage, and execution time.

This work provides not only practical guidance for MCU selection in industrial applications but also academic insights. By comparing the execution performance of different MCU architectures under a unified framework, we reveal how CPU structure, peripheral design, and interrupt mechanisms influence the sampling, computation, and output stages of a control loop.

This cross-platform analysis breaks the limitation of performance studies focused only on a single platform and offers a more systematic way to understand the relationship between hardware features and task requirements. At the same time, this approach lays the foundation for future exploration of hardware-software co-optimization in more complex control tasks, such as multi-loop control or AI-assisted control.

2

Technical Background

This chapter provides the necessary technical background for the rest of the project. It first introduces the basic concepts and common topologies of DC-DC converters, and explains the control modes often used in high-frequency power systems. Then, it describes the structure of embedded closed-loop control systems. These are important for understanding how the control loop works and where performance limitations may appear. Finally, it provides an overview of the three MCU platforms commonly used in the automotive field—F29H85x, AM263x, and AURIX TC4x—to compare their main features and evaluate their suitability for high-frequency control applications.

2.1 Fundamentals of DC-DC Converter Systems

A DC-DC converter is a circuit that changes one DC voltage to another DC voltage. It is widely used in power systems. The converter works by turning switches on and off, and moving energy between components like inductors and capacitors. This process can step the voltage up, down, or change its direction [13]. There are many types of converter designs and control methods based on different needs. We will explain some common topologies and basic control ideas.

2.1.1 Common Topologies of DC-DC Converters

Buck, boost, and buck-boost are three of the most common converter types [14]. As shown in Fig. 2.1, for these basic topologies, the inductor and capacitor work together to smoothen the energy flow and reduce output voltage ripple, and by controlling the turning on and off the switching device, the energy flow through the inductor can be adjusted, which helps to regulate the output voltage and keep it stable.

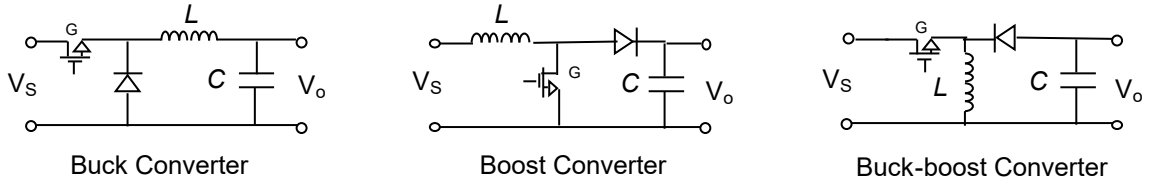

Figure 2.1: Different DC/DC topologies

Buck Converter

A buck converter is used to lower the input voltage to a smaller output voltage. It works by turning a switch on and off quickly. When the switch is on, energy is stored in an inductor. When the switch is off, the inductor releases the energy to the output [15].

The voltage conversion of a buck converter can be described by (2.1), where V_{in} is the input voltage, V_{out} is the output voltage, and D is the duty cycle, which is the ratio of the switch-on time to the full switching period.

$$V_{\text{out}} = D \cdot V_{\text{in}} \tag{2.1}$$

Boost Converter

A boost converter increases the input voltage to a higher level. When the switch is on, energy is stored in the inductor. When the switch turns off, the inductor releases the stored energy and adds it to the input, giving a higher output voltage. Today, boost converters are widely used in both small portable devices and large power systems due to their simplicity, good efficiency, and easy control [16].

The output voltage of a boost converter is given by (2.2). In this case, V_{out} is always higher than V_{in}, as long as the duty cycle D is between 0 and 1.

$$V_{\text{out}} = \frac{V_{\text{in}}}{1 - D} \tag{2.2}$$

Buck-Boost Converter

A buck-boost converter can either increase or decrease the input voltage. The output voltage depends on the duty cycle of the switching signal. This type also inverts the voltage, meaning the output polarity is opposite to the input. It is useful when the input voltage may be higher or lower than the desired output [17]. In (2.3), we can see that D is the duty cycle, and the negative sign means the output voltage is inverted compared to the input.

$$V_{\text{out}} = \frac{-D}{1 - D} \cdot V_{\text{in}} \tag{2.3}$$

Although buck, boost, and buck-boost are the most basic DC-DC converter topologies, engineers often build more complex designs based on them to meet real-world needs like higher efficiency, wider input/output range, or better control accuracy. For example, advanced systems may use multi-phase converters, bidirectional converters, or isolated topologies such as flyback, forward, or half-bridge, which are better suited for complex power systems [18].

2.1.2 Control Modes for DC-DC Converters

The control method also changes based on the application. Besides the commonly used voltage mode control (VMC) and current mode control (CMC), other modes like constant voltage (CV), constant current (CC), combined CV/CC, or peak current mode control (PCMC) are also widely used. These modes can be adjusted in real time using embedded controllers to improve system response, stability, and reduce electromagnetic noise [19, 20, 21, 22].

- **Voltage Mode Control:** The controller checks and compares the error between output voltage and reference voltage, then compares with a ramp wave (or triangle wave) to decide duty cycle.
- **Current Mode Control:** The controller checks and compares output current with reference current, and may also use an external voltage loop to set the reference current before sampling the inductor current.
- **Constant Voltage/Constant Current:** This means keeping voltage or current constant separately. The output voltage or current is measured and compared with a set value, the controller changes the duty cycle to hold the set value.
- **Constant Voltage and Constant Current:** This is a combination of two modes. At the beginning, the system works in CC. When the output voltage reaches a certain point, it switches to CV.
- **Peak Current Mode Control:** The inductor current is monitored during each cycle. The switch turns off as soon as the current reaches a set peak value.

The choice of control method has a direct effect on the overall performance of the system. Different control strategies can affect accuracy, response speed, stability, and electromagnetic interference, which is especially important in automotive and industrial systems. By using different control modes that can change in real time, embedded systems can keep good performance while becoming more flexible, reliable, and better suited for complex application needs [23].

2.2 Closed-Loop Control Systems Structure

In a DC-DC converter system, the output voltage usually needs to be carefully adjusted to match a given target value, so that the following load can work stably [24]. This is done by using the output voltage as a feedback signal, applying a proportional–integral–derivative (PID) controller to reduce the error, and then using PWM signals to control the on-time of the power switch, which is a typical voltage control mode. In this way, a typical closed-loop voltage control system is built. To achieve this, a sensor is often used to measure the feedback signal, which helps the system track the target value and keep the output accurate over time.

As shown in Fig. 2.2, in a typical DC-DC converter control system, the left part represents typical digital implementation in the MCU, while the right part represents

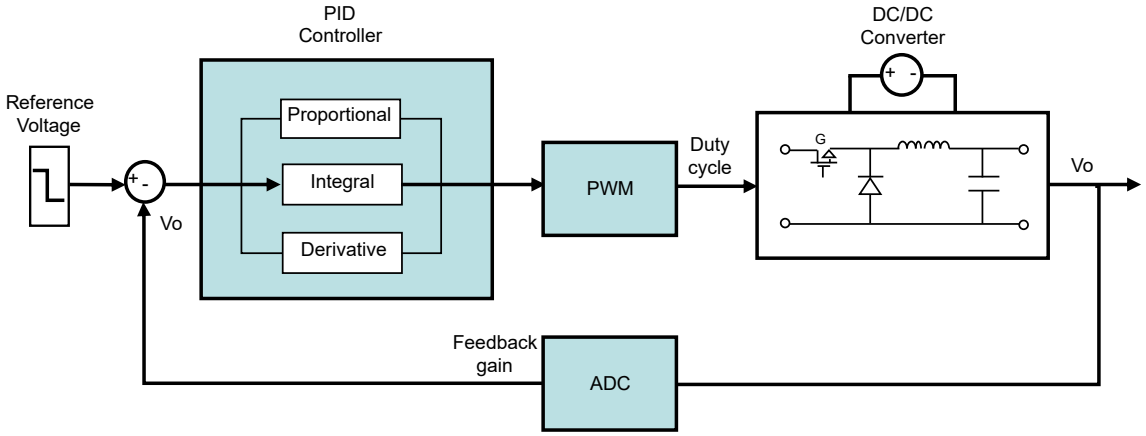

Figure 2.2: Closed-loop system in DC/DC conversion

the physical circuit. Output voltage is measured and converted into a digital signal by the ADC. This value is then sent back to the controller and compared with the reference voltage to calculate the error. The PID controller processes this error using proportional, integral, and derivative terms. Based on the result, it adjusts the duty cycle of the PWM signal, which controls the on-time of the power switch like the metal oxide semiconductor field effect transistor (MOSFET). This change affects the output voltage of the converter and helps keep it stable.

2.2.1 Principles of Closed-Loop Control Systems

In control systems, open-loop and closed-loop control are two fundamental types. Open-loop control means that the controller sends signals to the system without receiving any feedback from the output. In this kind of system, the controller does not adjust its action based on the actual result. It simply follows the given input and controls the device accordingly, like a one-way operation [25].

Because of the open-loop system's simple structure, easy setup, and stable operation, open-loop control is often used in situations where high accuracy is not needed or the environment does not change much. On the other hand, closed-loop control systems with automatic adjustment capability are more favored in practical industrial and embedded applications. A closed-loop control system, also called a feedback control system, is a type of automatic control system where the forward path and the feedback path form a closed loop. In this system, part or all of the output signal is measured by sensors and sent back to the input. This received output is then compared with the reference value to create an error signal. The controller uses this error to adjust the system in real time, so that the output gets closer to the target value. Most closed-loop systems use negative feedback, which helps improve control accuracy and response speed. This type of system is well-suited for complex situations that need high control performance [26].

2.2.2 Control Algorithm

In closed-loop control systems, the core task of the controller is to calculate an appropriate control signal based on the difference between the desired value and the actual output. This process is known as feedback control and is realized through various control algorithms. These algorithms define how the system reacts to the error signal over time. Among them, proportional–integral (PI), proportional–derivative (PD), and PID are the most common and practical solutions in industrial and embedded applications [27]. Selecting the right algorithm depends on the system's requirements for speed, accuracy, robustness, and available hardware resources.

PI control combines two parts. The proportional part reacts to the current error and gives an immediate response, while the integral part accumulates past errors to eliminate long-term bias. However, because it lacks a derivative component, PI control may respond more slowly to sudden changes. PD control is suitable for systems that require quick reaction and reduced overshoot because of the derivative component, which can predict future trends and improve system responsiveness. However, since it does not include the integral term, PD control cannot fully eliminate steady-state error. PID control uses three parts: proportional, integral, and derivative. When these three actions are set properly, the system can respond quickly and also reduce errors. The proportional part is the base control, the integral part helps remove steady errors, and the derivative part helps react faster. If the proportional value is too low, the system reacts slowly. If it is too high, it can cause large changes or make the system unstable. Too much integral action may cause overshoot, and the system takes longer to settle. The derivative part can help smooth the output, but it is easily affected by noise [28]. So, to get good control results, the three parts must be balanced carefully.

2.2.3 ADC-Based Feedback Acquisition

In most microcontroller unit (MCU)s, the ADC module is usually a successive approximation register (SAR)-ADC, this type of ADC module is shown in Fig. 2.3. This type of ADC works by using a step-by-step method to convert an analog signal into a digital value [29]. During the process, the SAR controls a digital to analog converter (DAC) to generate a reference voltage, which is then compared with the input analog signal, one bit at a time. The comparison starts from the highest bit and moves down, getting closer to the actual signal value with each step. This method provides a good balance between accuracy and speed, making it suitable for embedded systems that require fast and precise sampling.

In a typical data conversion process, an ADC goes through four main steps: sampling, holding, quantization, and encoding [30]. The first step is sampling, which means taking values from a continuous analog signal at regular time intervals. These sample points show how the original signal changes over time. Right after sampling, since the ADC needs some time to finish the next steps, the sampled value must be held steady. A holding circuit freezes the voltage for a short time, so the conversion

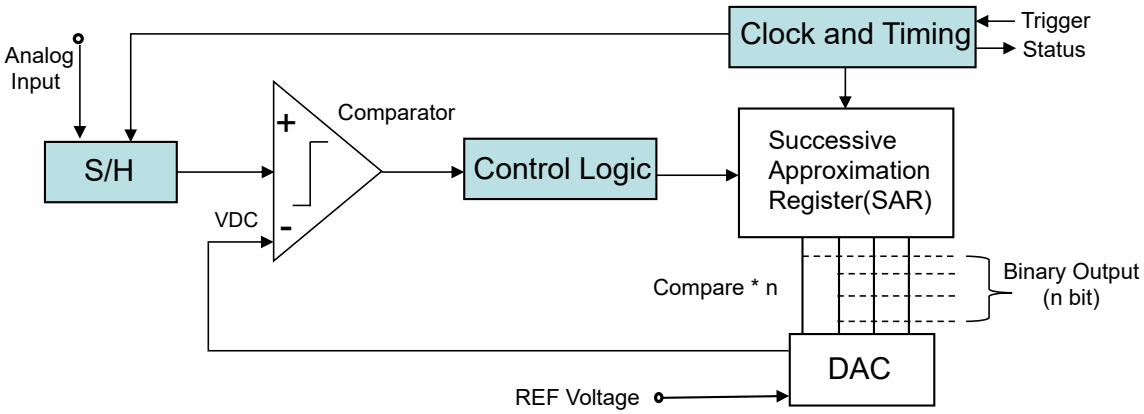

Figure 2.3: Basic SAR-ADC structure

can be done correctly. Even after holding, the signal is still an analog value. But digital systems can only work with a limited number of levels. The ADC matches the input voltage to the nearest level. This is like filing the analog value into a fixed slot. In a SAR-ADC, this is done step by step by comparing the input voltage with the DAC output and setting each bit to 1 or 0. The final result is a digital level closest to the input. The last step is encoding. The selected level is turned into a binary number so that digital systems can read and use it. For example, a 12-bit ADC gives a number from 0 to 4095 (2^{12} - 1), showing the size of the input signal. This whole process changes the signal from continuous to discrete in both time and value, which is the main goal of analog-to-digital conversion. Next, we will introduce several common types of SAR-ADCs in different MCUs.

ADC Architecture in TI MCUs

The ADC module in many Texas Instruments (TI)'s MCUs is a multi-channel, high-speed, and flexible SAR-ADC. It is built with several main parts: the input side uses an analog multiplexer to select channels, and works with a sample-and-hold (S/H) circuit that has a configurable sampling window to set the sampling time. The core converter is a 12-bit or 16-bit SAR engine, and supports multiple converter cores for parallel sampling [31]. Each conversion is controlled by a start of conversion (SOC) unit, where users can set the input channel and trigger source (such as enhanced pulse width modulation (ePWM), general purpose input/output (GPIO), or software). The result is stored in a register which can be read by the CPU or direct memory access (DMA). The interrupt module supports different types of interrupts, like end of conversion (EOC). In addition, the post-processing block (PPB) allows hardware-level offset correction, limit checking, and over-threshold event detection, which helps reduce the CPU load. The event logic can generate event signals based on user-defined conditions (such as when the result reaches a certain threshold), which can trigger interrupts or other module actions. This improves system responsiveness and enables smarter control.

In addition to the main sampling, conversion, and event processing structure, the TI ADC module also includes a SOC arbitration and control unit. This unit manages

trigger requests from multiple SOCs and ensures each conversion task is handled in order. The interrupt module is divided into several independent interrupt blocks, allowing users to assign different interrupt channels based on the SOC or event source, which improves flexibility in interrupt handling. The TI ADC has the multi-trigger support mechanism, where each SOC unit can be set up with different trigger conditions. This makes the system better suited for complex control tasks.

TMADC Architecture in Infineon MCUs

Infineon time multiplexed analog to digital converter (TMADC) is a high-performance ADC based on the SAR architecture in Fig. 2.4. The module supports multiple analog input channels, each connected to a dedicated S/H unit. A connection matrix dynamically selects the target input channel and connects it to the converter core. The module also supports external analog multiplexers, which can be controlled through the EMUX interface to expand input capability [32].

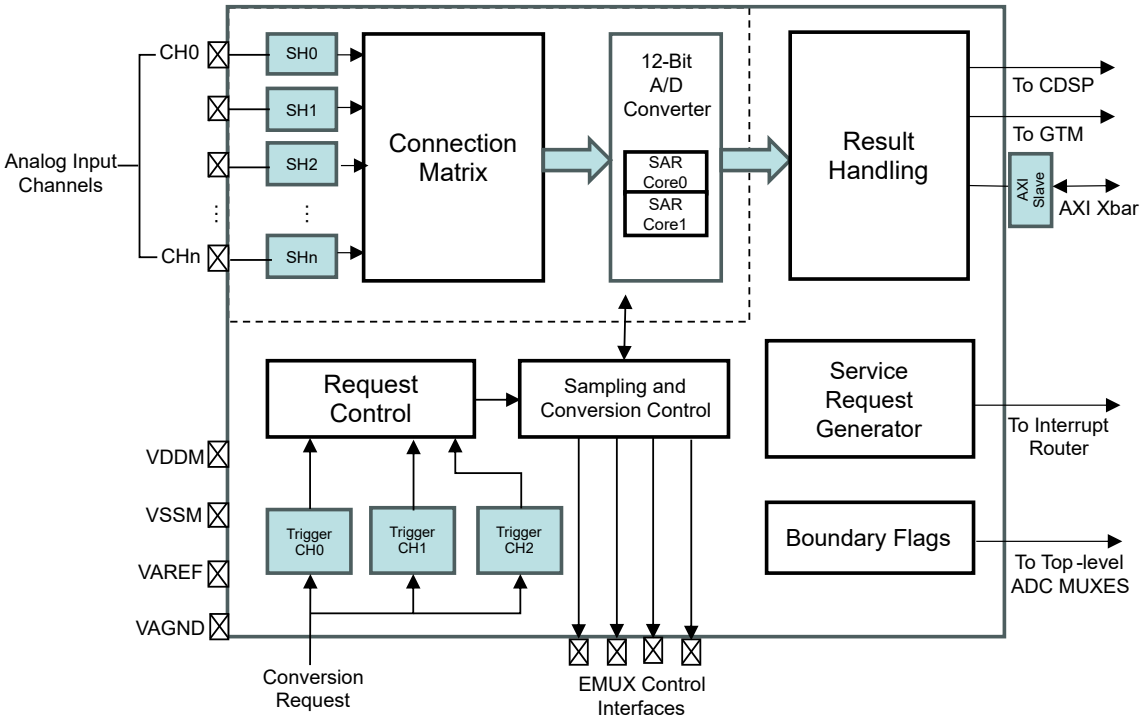

Figure 2.4: Infineon TMADC block design - Simplified (Source: Infineon Technology). [32]. Adapted with permission.

TMADC includes SAR core0 and core1 to enable parallel sampling. The sampling process is managed by the sampling and conversion control unit, which ensures correct timing between the S/H circuit and the ADC cores. Conversion requests are handled by the request control logic, which receives and prioritizes requests from different trigger sources. TMADC can be triggered by events from generic timer module (GTM), timers, or software. After conversion, the result is processed by the result handling unit and passed to downstream modules through different interfaces. TMADC supports direct data transfer to the converter digital signal

processor (CDSP) for real-time signal processing, feedback to GTM for closed-loop control, or transfer through the advanced extensible interface (AXI) crossbar to system memory. The module can also notify the main processor via the interrupt router (IR). The service request generator sends an interrupt or DMA request after each conversion to ensure fast and efficient data handling. In addition, TMADC includes boundary flag logic to check if the input value goes beyond a defined range. It supports both upper and lower threshold detection. In addition, the service request generator can send requests to both the interrupt router and DMA.

2.2.4 PWM-Based Mechanism

PWM is a method of generating a periodic digital signal with an adjustable duty cycle, like Fig. 2.5. By changing the ratio of high time to the total period, PWM signals serve two main purposes. First, they are commonly used as output signals for controlling power devices, such as MOSFETs, by adjusting the duty cycle. Second, when set in a fixed frequency and duty cycle, the PWM can act as a precise timing signal to trigger other components, such as the ADCs, enabling synchronized sampling. In the following, we will introduce two general PWM generation modules.

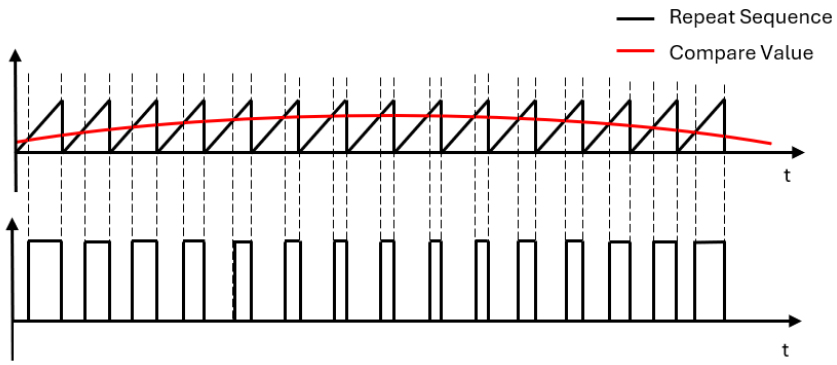

Figure 2.5: PWM duty cycle modulation with changing compare value

ePWM Architecture in TI MCUs

The ePWM module is a flexible timer peripheral commonly found in TI [31]. It can generate high-resolution PWM signals and supports configurable frequency, duty cycle and phase. The ePWM module also supports synchronized triggering with other peripherals and can operate in up-count, down-count, or up-down count modes. The ePWM requires some calculations by using an internal time-base counter (TBCTR) and a period register to create a periodic waveform. The frequency of PWM is shown below. f_{SYSCLK} is the system clock frequency, HSPCLKDIV and CLKDIV are clock dividers set by the user. For the f_{TBCLK}, by setting these clock dividers, the user can control the base counting speed of the PWM.

$$f_{\mathrm{TBCLK}} = \frac{f_{\mathrm{SYSCLK}}}{\mathrm{HSPCLKDIV} \cdot \mathrm{CLKDIV}} \tag{2.4}$$

Next, the PWM output frequency f_{PWM} and duty cycle is given by:

$$f_{\text{PWM}} = \frac{f_{\text{TBCLK}}}{\text{TBPRD}} \tag{2.5}$$

$$\text{Duty Cycle} = \frac{\text{CompareValue}}{\text{TBPRD}} \cdot 100\% \tag{2.6}$$

Here, time base period register (TBPRD) sets the maximum value of the counter. When the counter reaches this value (or returns to zero), one PWM cycle is complete. So, a higher TBPRD value results in a lower PWM frequency. The duty cycle is determined by the ratio between the CompareValue user set and the period register TBPRD. When the counter reaches the compare value, the PWM output toggles, forming a pulse with controllable width.

GTM Architecture in Infineon MCUs

For the PWM generation, most MCUs in Infineon use the GTM. The GTM is a timer core architecture developed by Bosch in Germany. It includes a modular framework with several sub-modules, each with different functions [33]. These sub-modules can be combined in a flexible way to create a complex timer system. This system can support different application areas, as well as different types of tasks within the same application.

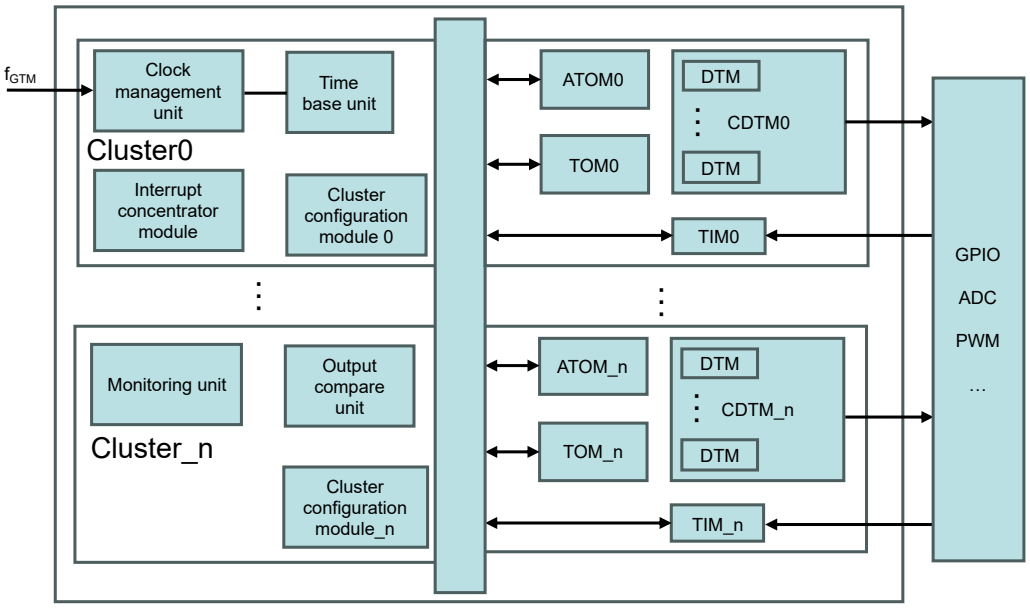

Figure 2.6: GTM structure (Source: Infineon Technology). [32]. Adapted with permission.

In the GTM, as shown in Fig. 2.6, the internal time output module (TOM) and ATOM (advanced TOM) are powerful components. By setting the clock source, period, and duty cycle, we can generate multi-channel PWM signals. These signals

can be used not only to drive power devices but also to trigger ADC sampling, enabling precise timing synchronization. In addition to basic PWM generation, the GTM architecture also includes modules such as the dead time module (DTM) and combined dead time module (CDTM), which help insert dead time between complementary signals—important for preventing short circuits in power stages. It also contains time input module (TIM) that can capture external signal timing and support event measurements. Compared to traditional timers, the GTM allows more flexible and accurate control with register-level configuration. This makes it widely used in automotive MCUs, supporting high-precision PWM and complex timing logic.

2.3 Embedded Platform Architecture

In the design of DC-DC converter control systems, selecting the right MCU requires careful consideration of several key factors. In addition to the advanced ADC and PWM module we discussed above, these also include real-time response, safety features, and software development support. A balanced choice ensures the control system runs efficiently and reliably under various working conditions. Here, we will summarize the specific MCU's requirements for real-time power electronics systems.

- **Real-Time Response:** In DC-DC control systems, voltage or current feedback must often be processed within microseconds. Therefore, the MCU should have low interrupt latency, fast instruction execution, and a high-bandwidth bus system to ensure real-time control. Especially under high switching frequency or fast load changes, quick response is essential for maintaining system stability and precision.
- **Peripheral Configuration:** High-performance peripherals are critical for precise closed-loop control. The ADC should offer high resolution and fast sampling, with support for multi-channel synchronous sampling, hardware triggering, oversampling, and digital post-processing. The PWM module, which drives power switches, should support features like dead-time insertion, symmetric/asymmetric output, fault handling, and sync triggering. These help improve resolution, reduce EMI, and simplify external circuit design.
- **Functional Safety Support**: In industrial and automotive power systems, safety is a key requirement [34]. The MCU should support safety features like lockstep dual-core architecture, error correction code (ECC) checks, fault detection, and injection. It should also help meet standards such as ISO 26262. These functions ensure fault isolation and recovery, improving system reliability under abnormal conditions.
- **Architecture and Development:** Besides performance, the development tools and software ecosystem are also important. A good MCU platform should provide reliable compiler support, debugging tools, driver libraries, and reference projects. At the same time, factors like cost, supply stability, and compatibility with existing systems must be considered to ensure feasibility and long-term maintenance.

By evaluating these important requirements, we can better understand what makes an MCU suitable for power control applications. In the following, we will introduce several MCU platforms that meet these needs, each offering different strengths in performance, peripheral integration, and so on.

2.3.1 Overview of F29H85x

The F29H85x belongs to the C2000 series of real-time MCUs, developed for power electronic systems that need low latency and high scalability.This device is especially suitable for applications with high power density and fast switching, including those that use GaN or SiC technology to increase energy efficiency [35].

The F29H85x has three C29x digital signal processor (DSP) cores, each running at up to 200 MHz. The operations can be run directly from the on-chip Flash or RAM. It also supports trigonometric instructions to speed up common real-time control tasks. To ensure system robustness, the chip includes features like CPU lockstep mode and ECC-protected connections. These help detect system faults with very low CPU load. The built-in safety and security unit (SSU) includes an memory protection unit (MPU) that can change memory access rules based on the running task. This helps improve safety without making the software more complex. At the core of the F29H85x is the C29x CPU, which adopts a very long instruction word (VLIW) architecture with a protected multi-stage pipeline. It can execute up to eight instructions in parallel, supported by 64 general-purpose registers and dedicated status registers for managing execution state and interrupts. The modular CPU design includes a main execution unit, a protected stack unit for handling function calls and interrupts, and internal buses that coordinate memory and peripheral access. This architecture enables fast instruction decoding, address generation, and data transfer, making it well-suited for real-time control, signal processing, and feedback computation tasks [36].

The F29H85x also includes high-performance peripherals, such as several 12-bit and 16-bit ADC modules with many input channels. These ADCs support oversampling and digital post-processing. The chip also has many high-resolution PWM outputs to help improve control accuracy and system safety. The F29H85x provides four fixed types of interrupt lines: RESET, NMI (non-maskable interrupts), RTINT (maskable real-time interrupts), and INT (maskable and disableable interrupts). It also includes a peripheral interrupt priority and expansion (PIPE) module for extra priority control and arbitration. The system supports interrupt nesting in hardware. During nesting, the CPU automatically saves and restores the system context, allowing very short interrupt response time.

2.3.2 Overview of AM263x

The AM263x is a microcontroller in the Sitara Arm family, designed for industrial and automotive embedded systems that require high-performance real-time processing. It supports up to four Arm Cortex-R5F cores, each running at 400 MHz. Each

R5F core can operate in either lockstep or dual-core mode, allowing different levels of functional safety based on application needs [37].

In terms of robustness, each core is equipped with a dedicated MPU and internal diagnostics that monitor voltage, temperature, and clock signals during runtime. The R5F cores are organized into a cluster that shares tightly coupled memory (TCM) and static random access memory (SRAM), which helps reduce the need for external memory. ECC is widely applied across internal memory, peripherals, and interconnects to improve reliability. The device also integrates a hardware security manager (HSM) that provides fine-grained access control, helping developers meet the demands of safety-critical systems. The AM263x provides a wide range of on-chip peripherals to support high-precision sensing and control tasks. For analog input, the device includes five 12-bit SAR-ADCs, each capable of sampling up to 4 MSPS. These ADCs support oversampling and digital post-processing. For actuation, each ePWM module supports single or dual-channel outputs and includes extended high resolution pulse width modulation (HRPWM) support, making it suitable for complex power control and motor drive applications.

The R5F subsystem in the AM263x is based on a dual-core Arm Cortex-R5F architecture, supporting both dual-core and lockstep configurations [38]. In dual-core mode, the two cores operate independently and can handle different tasks, organized in 4-way sets and protected by ECC. The AM263x's interrupt, for each R5F core, uses a vectored interrupt manager (VIM) module; this hardware vectored interrupt connection in ARM Cortex R5F is like Fig. 2.7. Each interrupt can be set as interrupt request (IRQ) or fast interrupt request (FIQ). By default, the software uses a table lookup and dispatch to call the user's interrupt service routine (ISR), which adds some delay to the response. This interrupt system does not support full hardware nesting, but it can support limited nesting using IRQ/FIQ separation and software-based handling. This architecture reflects a common pattern in ARM-based systems. Many ARM platforms use a centralized interrupt controller along with a software-managed vector table to dispatch ISRs.

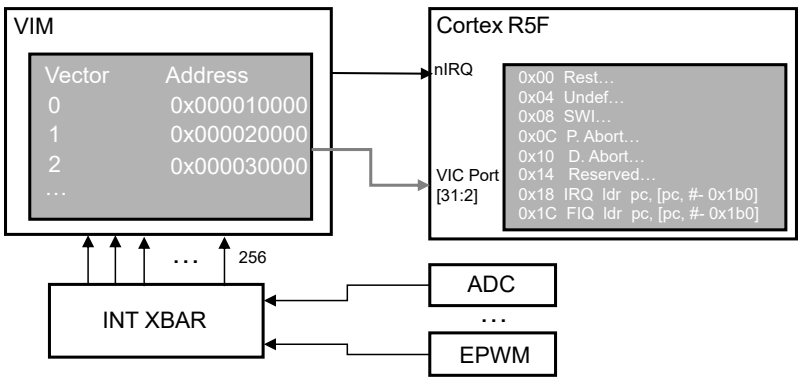

Figure 2.7: Interrupt architecture of AM263x [39]

2.3.3 Overview of AURIX TC4x

The AURIX TC4x series is a new generation of 32-bit microcontrollers from Infineon, designed for real-time control in automotive and industrial applications [40]. As shown in Fig. 2.8, it is based on the proven TriCore architecture and supports up to six CPU cores, each running at up to 500 MHz. The TC4x family includes hardware safety features that meet ASIL-D standards, making it suitable for systems that require high reliability, such as electric vehicles, ADAS, and chassis control.

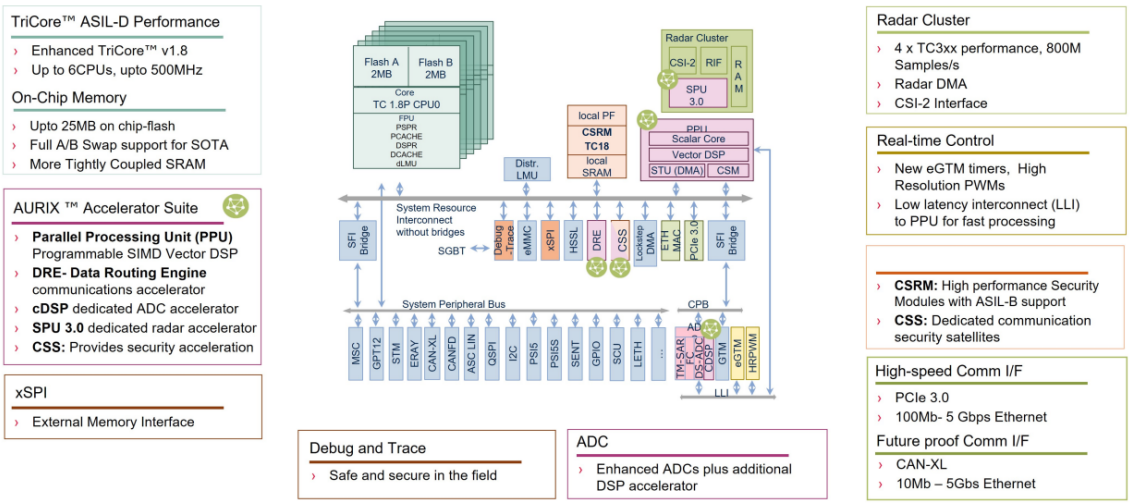

Figure 2.8: Functional block diagram of AURIX TC4x (Source: Infineon Technology). [40]. Reprinted with permission.

TC4x integrates a wide range of advanced modules to improve performance and flexibility. These include enhanced ADCs for accurate signal capture, and the enhanced generic timer module (eGTM) for high-resolution PWM control. The low-latency interconnect (LLI) helps these modules work together quickly. The series also includes programmable hardware accelerators like the parallel processing unit (PPU) and CDSP, which can offload complex control or signal processing tasks from the main CPU. High-speed communication is supported through PCIe, Ethernet, CAN-XL, and other modern interfaces.

At the center of the TC4x is the TriCore 1.8 processor. This 32-bit CPU combines the real-time control of a microcontroller, the math ability of a DSP, and the efficient instruction flow of a reduced instruction set computer (RISC) core. It supports both 16-bit and 32-bit instructions, fast interrupt handling, zero-overhead loops, and SIMD operations. The core also includes dual multiply-accumulate (MAC) units, and optional units like a FPU and memory protection. These features make it well-suited for fast control loops and safety-critical applications.

The interrupt system of the AURIX TC4x is similar to that of the AM263x, its structure is shown in Fig. 2.9. Peripheral modules send interrupt requests to the IR through service request node (SRN). The IR arbitrates and routes the requests based

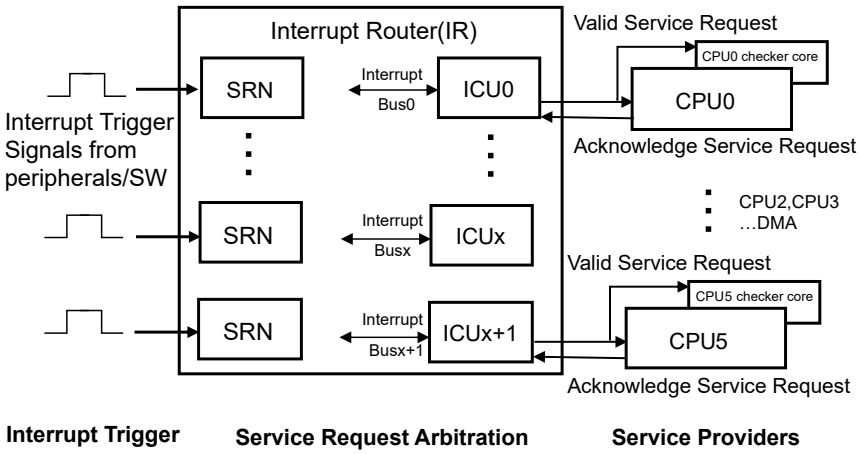

Figure 2.9: Interrupt architecture of AURIX TC4x (Source: Infineon Technology). [32]. Adapted with permission.

on the priority, enable status, and type of service (TOS) configured in the service request control (SRC) register. Once the CPU receives the interrupt, it jumps to the corresponding address in the interrupt vector table according to the interrupt priority. This mechanism supports fast hardware-level vector jumping.

3

Methods

This chapter focuses on the control system design approach for realizing good real-time performance. We propose a general method to analyze and improve system performance. By breaking down the control process into different types of delays, we will build an interrupt-driven framework and explore how to apply the same control logic across different hardware platforms. The optimization strategies cover multiple aspects, including peripheral settings, data transfer, interrupt handling, memory usage, and computation acceleration. Finally, we will use a unified testing method to evaluate how well these strategies perform and transfer across platforms.

3.1 Control Structure and Workflow

The control structure will follow a typical closed-loop model: sampling, calculation, and output. The system works by reading the ADC input, computing the control output, and updating the PWM signal within a fixed cycle.

In this project, we will use an interrupt-driven loop that runs once per PWM cycle. The PWM is 100 kHz, so we trigger the control ISR every 10 µs. This lets us update the duty cycle each switching period and still leave time for ADC conversion and control code, giving a fast and stable loop. For ADCs, there is a basic trade-off between resolution and sampling rate. A higher resolution means more precise quantization but requires more time for each conversion. Conversely, increasing the sampling rate shortens the available conversion time, which can increase noise and reduce accuracy. The ADC sampling time is set to 80 ns. At 80 ns we get a good balance, the reading is close to the true value while easily meeting the 10 µs period.

The control algorithm remains the same across all platforms. While the PWM update mechanism differs between MCUs, we will keep the initial configuration consistent, including counting mode, duty cycle, and PWM frequency. Since the PWM value is continuously updated by the PID controller during runtime, these initial settings are sufficient to eliminate the impact of platform-specific differences. Precise matching is not critical, as the control loop dynamics quickly override the initial values.

3.2 Performance Analysis and Optimization

To reduce delay in the system, we will analyze the control cycle step by step. This includes sampling delay, processing delay, and response delay. We will also examine how these delays interact and propagate through the system, identifying key bottlenecks in execution.

The optimization work will focus on several directions:

- Adjusting peripheral configurations such as ADC sampling time and PWM update behavior to improve timing characteristics and reduce uncertainty.
- Using automatic data transfer methods, like DMA, to offload routine memory operations from the CPU, minimizing processing delay.
- Improving the interrupt structure to shorten response latency and enable faster entry into the control loop after hardware events.
- Optimizing memory usage and compiler settings to accelerate control logic execution by minimizing access latency and maximizing instruction throughput.
- Utilizing on-chip hardware accelerators, where available, to offload compute-intensive tasks such as PID calculations, improving performance while reducing CPU workload.

All methods aim to enhance real-time responsiveness without sacrificing control accuracy or system reliability. These optimizations provide a general framework that can be adapted to different microcontroller architectures.

3.3 Cross-Platform Evaluation

To verify the portability and real-time performance of the control system on different hardware platforms, we will implement the same control logic on three mainstream MCUs in the automotive field: TI F29H85x, TI AM263x, and Infineon AURIX TC4x.

We use a GPIO-based timing method to capture the delays of different stages in the control loop. While the core algorithm remains the same, these platforms differ significantly in architecture, peripheral design, instruction set, and interrupt handling. We will focus on adapting a unified control structure—including ADC sampling, PID calculation, and PWM update—to different hardware environments. The same measurement method will be used to evaluate response latency on each platform, ensuring a fair comparison. In addition, we will apply all the proposed optimization strategies to each platform. These strategies aim to match each MCU's characteristics, to reduce control latency as much as possible while keeping control accuracy.

Through this cross-platform evaluation, we will reveal how architectural differences affect control performance. We will also summarize a set of optimization ideas suitable for migrating control systems between platforms, and identify key factors and common strategies that are important in real-time control loop development.

4

Design

This chapter describes the design and implementation process of the control system. It first explains the setup of the development environment, including the hardware platforms and measurement equipment used. Then, it introduces the design of the baseline control loop, showing how the peripherals were configured and how control was implemented using interrupt-driven methods. After that, it analyzes the sources of system delay and the methods used to measure them. Finally, it presents several optimization strategies, including improvements in ADC sampling, DMA implementation, interrupt handling, compiler settings, and MCU-unique customizations, to further reduce delay and improve control performance.

4.1 Development Environment Setup

To successfully implement the closed-loop control system in our project, this section introduces the hardware connections for the three MCU platforms, the setup of the development environment, and the external equipment used for measurement.

4.1.1 Hardware Platform Configuration

Our project is based on three MCU platforms: F29H85x, AM263x, and AURIX TC4x. Each setup includes the required control board, adapter board, debugger, and cables. These can support the full control loop, as well as debug ability. There are some differences between the three platforms in peripheral support, connection interfaces, memory allocation, and hardware setup. These differences provide the basis for later comparison and performance optimization.

F29H85x Hardware Connection

The F29H85x hardware platform used in this project includes the core control board, interface adapter board, and debugging equipment, as shown in Fig. 4.1. A brief introduction and connection overview are provided below [41].

- **F29H85x control system on chip (SOM)**: This is the core processing board that contains the F29H85x MCU. It handles all key control tasks.
- **High-Speed Edge Card (HSEC)-180 Adapter Board**: This adapter converts the controlSOM interface to a standard 180-pin HSEC format, enabling compatibility with legacy C2000 evaluation platforms.

- **Docking Station**: Acts as the baseboard for the control SOM. It exposes key MCU signals through pin headers and provides a breadboard area for rapid prototyping. Power can be supplied via USB or a 5V DC jack.
- **XDS110 Isolated Debug Probe**: Used for program flashing and real-time debugging. It supports both isolated and non-isolated connections and provides JTAG, cJTAG, SWD, and UART interfaces. Additional I/O and DAC channels are available for advanced debug features.
- **USB Cables**: One for the power supply to the SOM baseboard, and one for connecting the debug probe to a host PC.

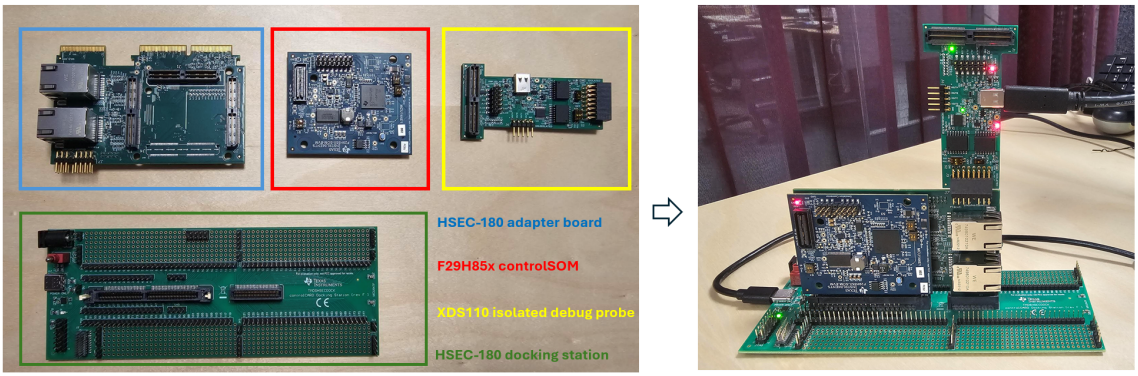

Figure 4.1: F29H85x hardware connection

AM263x Hardware Connection

To evaluate and develop the embedded control system on the AM263x platform, the hardware connection is shown in Fig. 4.2, and the following components are required [42]:

- **AM263x controlCARD**: This is the main microcontroller module based on the AM263x series. It integrates the Arm Cortex-R5F cores and the necessary on-chip peripherals for real-time control tasks. It serves as the processing and control unit of the system.
- **HSEC Dock and Breakout Board**: This board is used to connect the AM263x controlCARD through a 180-pin HSEC connector. It breaks out the internal system on chip (SoC) signals to headers for easy access. It supports hardware interfaces like CAN, LIN, JTAG, TRACE, and GPMC, and includes a built-in 64Mb PSRAM for memory expansion during development and testing.
- **USB Cables**: One USB cable is used to supply power to the board through the USB interface if no external 5V supply is used. The second USB cable connects the onboard XDS110 debug interface to the PC, providing support for JTAG/cJTAG debugging, flash programming, and UART communication.

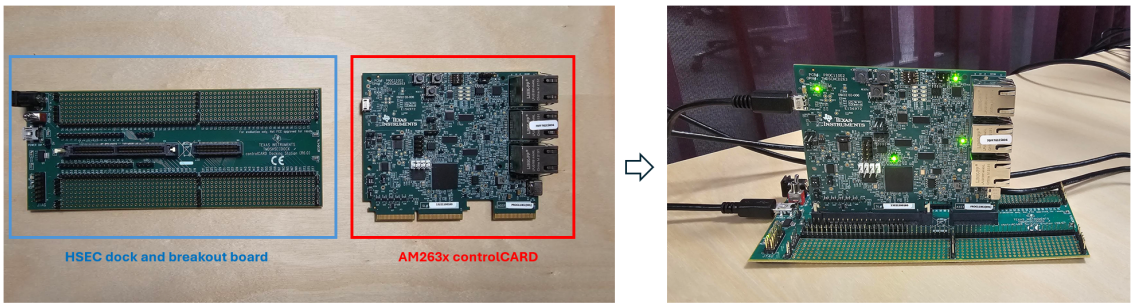

Figure 4.2: AM263x hardware connection

AURIX TC4x Hardware Connection

To evaluate the TC4x platform, we use the AURIX TC4x Lite Kit with the connection in Fig. 4.3, which integrates the core MCU, peripheral interfaces, and debug functionalities on a single compact board. The hardware setup includes:

- **TC4x Lite Kit Board**: Unlike the F29H85x and AM263x platforms which rely on HSEC-based interconnections, the Lite Kit consolidates all necessary components—including power supply, peripheral interfaces, and debug via a single PCB.
- **USB Cables**: The board requires only one USB Type-C cable to support both power delivery and debugging access. This single-cable configuration simplifies setup and avoids additional adapter or docking modules.

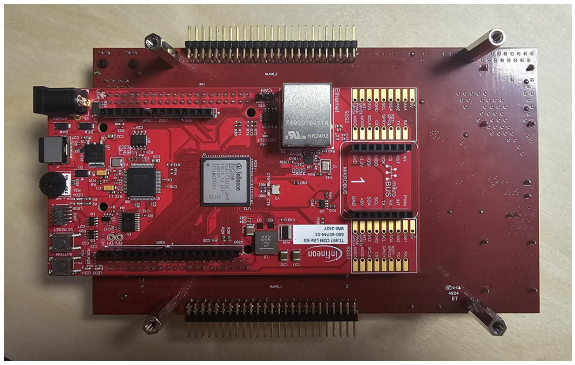

Figure 4.3: TC4x hardware connection

4.1.2 Measurement and Testing Instruments

In our project, two laboratory instruments are used to test the real-time performance and output response of the closed-loop control system. The oscilloscope is used to capture key waveforms in the control system, including GPIO level changes and PWM output signals. These waveforms help us measure sampling delay, control loop period, and dynamic response. In this process, GPIO toggling shows the execution time, and the oscilloscope helps display the timing relation between multiple signals, which makes it useful for time measurement in our tests.

The signal generator is mainly used to provide controlled analog input signals to the system, such as a target voltage or disturbance signal. This helps to activate the control loop and observe how it responds. In real applications, the signal sent to the PID controller usually comes from the feedback voltage of the power converter. This voltage is regulated by the PWM, processed through the power stage, and then sampled by the ADC, as shown earlier in Fig. 2.2, forming a complete closed-loop control system.

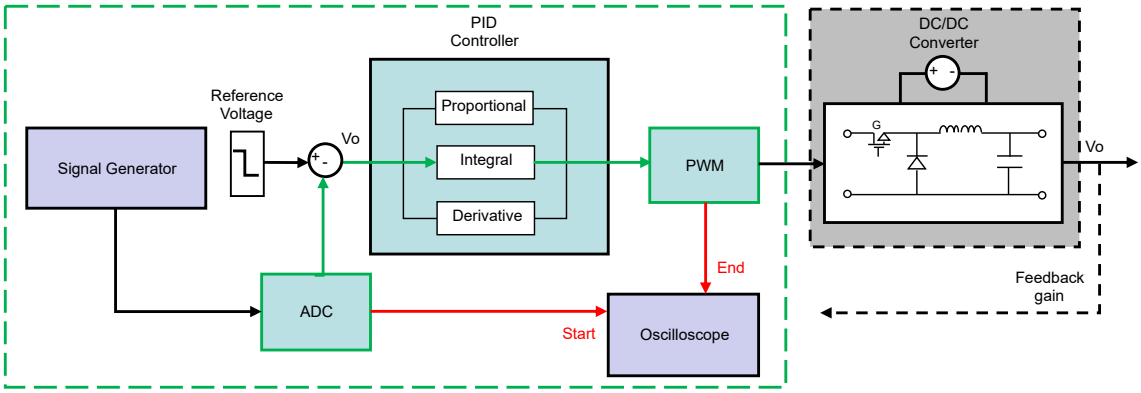

Figure 4.4: Experimental setup for control loop timing measurement

However, in our project, the goal is to measure the execution time from end-to-end on different MCUs, that is, the time from ADC sampling to PWM update. To achieve this, we disconnect the actual DC-DC converter, as shown on the left side of Fig. 4.4, and use a signal generator to replace its output. The signal generator provides a stable and adjustable analog voltage, simulating the converter's feedback behavior under controlled conditions. This analog signal is directly connected to the ADC, which triggers the control loop as if a real converter were present.

4.2 Baseline Control Loop Design

The baseline control loop in this project includes the basic functions needed for closed-loop control, such as sampling, computing, and actuation. This section focuses on how the same system is built using different peripherals and control structures, and how structural differences between peripherals lead to different implementation methods. These explanations help reveal the basic system performance before any optimization, and to some extent, show how peripheral design affects execution performance under different architectures, which can help us find out how the optimization can be designed.

4.2.1 ADC Module Configuration

In our control loop implementation, the ADC module is an important part used for sampling the input signal. Its settings directly affect how fast and how accurately the control system works. The main configuration points include the clock, resolution, signal mode, and trigger source. In our previous section, we have introduced

the structure of ADC modules from TI and Infineon. Here, we will focus on the configuration and final implementation aspects.

TI ADC Module Setting

First, we start from the TI's ADC configuration for F29H85x and AM263x. The ADC clock controls how fast the ADC works. A lower clock divider gives a faster sampling speed, which means the EOC signal and the interrupt flag will be generated earlier. This helps reduce the total loop time. However, if the clock is too fast, the ADC or control code may not have enough time to complete each operation, leading to noise or small conversion errors. It also increases switching loss and power use. As a result, the overall accuracy and efficiency may drop. So, the clock setting should balance speed and system reliability.

$$f_{\text{ADC}} = \frac{f_{\text{SYSCLK}}}{\text{Prescaler}} \tag{4.1}$$

Second, the resolution of the ADC decides how detailed the digital result is. Higher resolution gives smaller quantization error and more accurate data, but usually takes more time for each conversion. To trade off the accuracy and execution time, we will choose 12-bit ADC for our ADC implementation and select ADC clock prescale to 6, and keep the sampling time to 80 ns. Then we can find the timing information by selecting our prescale in Table 4.1 provided by TI.

Table 4.1: TI ADC module timing parameters in 12-bit mode [31]

ADCCLK Prescale		**STSCLK CYCLE**		
ADCCTL2 PRESCALE	**Prescale Ratio**	**t_EOC**	**t_LAT**	**t_INT**
0	1	11	13	11
2	2	21	23	21
3	2.5	26	28	26
4	3	31	34	31
5	3.5	36	39	36
6	4	41	44	41

This project uses a single-ended sampling mode. In this mode, each input signal is compared to a fixed reference voltage (usually 0V, known as V_{REFLO}). For example, if we connect an analog signal, the ADC will compare it to 0V and then convert the difference into a digital number, and we can calculate the raw value, which is the unprocessed digital code directly output from the ADC, by (4.2) with the resolution n. This applies to the SAR-based ADC module discussed earlier.

$$V_{\text{digital}} = \frac{V_{\text{in}} - V_{\text{REFLO}}}{V_{\text{REF}}} \cdot (2^n - 1) \tag{4.2}$$

In TI platforms, ADC conversion is controlled using SOC units. Each SOC defines which input channel to sample, when to trigger the sampling, and how long to hold

the signal. These include channel selection, trigger source, and acquisition window, measured in ADC clock cycles, which sets how long the signal is held for conversion. If too short, accuracy may suffer.

The acquisition time T_{acq} can be calculated using (4.3), where N is the sample window for the acquisition window, and T_{SYSCLK} is the system clock frequency used by the ADC. The formula is:

$$T_{\text{acq}} = N \cdot T_{\text{SYSCLK}} = \frac{N}{f_{\text{SYSCLK}}} \tag{4.3}$$

This duration corresponds to the time during which the S+H circuit holds the input voltage. After that, the ADC starts the conversion process. Once the EOC period is completed, the conversion is considered finished. If ADC interrupts are enabled, an interrupt will be triggered at the same time after the conversion ends. The latch time refers to the moment when the converted value is written into the corresponding result register.

Therefore, the total sampling time of the ADC consists of the sum of the acquisition time (S+H) and the latch time. While the acquisition time can be calculated using (4.3), the durations of the EOC, interrupt, and value latched are platform-dependent and can be found in Table 4.1. In the basic test of this project, only one SOC is used to sample a single input channel. However, if more channels are needed, additional SOC units can be configured to support multiple or alternating samples.

TMADC Module Setting

Unlike TI's ADC, which performs sampling and conversion immediately after receiving a trigger signal, TMADC from Infineon TC4x separates the sampling and conversion. During the sampling stage, the TMADC tracks the analog signal on the selected channel and only begins the conversion process once a conversion request or trigger is received. TMADC supports two conversion modes: one-shot and continuous auto-scan. One-shot conversion is trigger-based, while continuous mode operates through automatic polling. Both modes can be applied to either a single channel or a group of channels. In one-shot conversions, when a trigger event occurs, the selected channel is placed into an arbitration queue. The SAR core then selects the next higher-indexed channel from the previously converted one, following a round-robin arbitration scheme. The whole process is in Fig. 4.5. This mechanism ensures that fast channels do not block the conversion of relatively slower channels. However, for single-channel applications, this scheduling has less impact.

On the other hand, when it comes to calculating the total time of the ADC process, TMADC lacks exact data on conversion time due to commercial or confidentiality reasons. However, the sampling time can still be configured by setting the corresponding register with a desired value. Although TMADC has a minimum sampling time to ensure accuracy, this is not a major limitation. Since the ADC module from

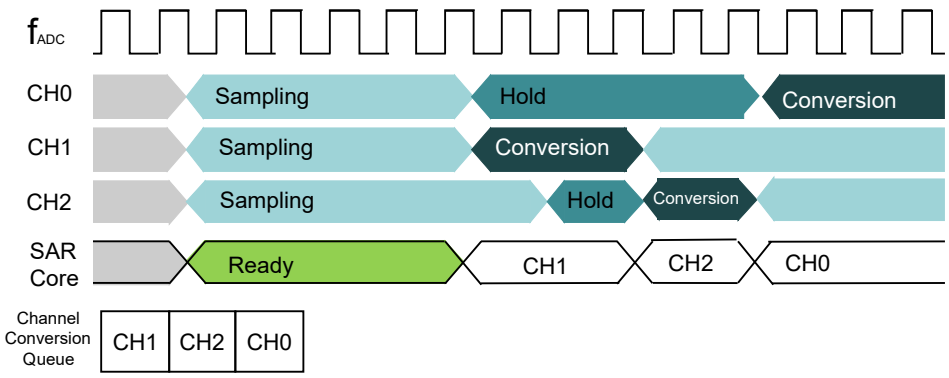

Figure 4.5: TMADC module sampling diagram (Source: Infineon Technology). [32]. Adapted with permission.

TI also allows the sampling time to be calculated through (4.3), it is not difficult to align the sampling durations of both platforms while still meeting their respective requirements. Therefore, in the baseline configuration stage, we set the sampling time to 80 ns, enable interrupt generation upon conversion completion at the same frequency, and use single-channel sampling on both platforms, and so on. These consistent settings help ensure that only one variable is changed during comparison.

4.2.2 Control Algorithm Implementation

The PID controller continuously calculates the error between the reference value and the feedback signal (measured by the ADC, can be calculated by (4.2)), and generates a control signal to adjust the system output. In this project, the PID controller is used to calculate the PWM duty cycle needed to regulate the output voltage of the DC-DC converter.

The standard continuous-time PID formula is:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)d\tau + K_d \cdot \frac{d}{dt}e(t) \tag{4.4}$$

In a digital implementation, we usually use a discrete version:

$$u[k] = K_p \cdot e[k] + K_i \cdot \sum_{i=0}^{k} e[i] \cdot T + K_d \cdot \frac{e[k] - e[k-1]}{T} \tag{4.5}$$

where $e[k]$ is the error at the current step, equal to reference minus feedback, T is the control cycle time, Kp, Ki, Kd mean proportional, integral, and derivative gains. To ensure system stability and avoid extreme output, the integral term is limited to prevent windup. The final control output is clipped to stay within the valid PWM range. $u[k]$ will represent the control output for the later calculation. After PID adjustment, the output $u[k]$ is mapped to a specific duty cycle through normalization like (4.6).

$$\text{Control Output} = \frac{\text{u[k]} - U_{\text{MIN}}}{U_{\text{MAX}} - U_{\text{MIN}}} \tag{4.6}$$

Here, U_{MAX} and U_{MIN} represent the lower and upper bounds of the usable control output range. The normalized control output will fall within [-1, 1], which can then be mapped to the PWM-relevant peripheral from 0 to 1.

4.2.3 PWM Module Configuration

After getting the duty cycle calculated and normalized by the PID controller, the corresponding PWM waveform can be generated directly through the PWM relevant module and changing the PWM duty cycle to simulate how a DC-DC converter adjusts its output voltage, completing the execution of the closed-loop control cycle. On the TI platforms, F29H85x and AM263x, the PWM signal is generated using the ePWM module, while on the Infineon TC4x platform, the GTM is used instead. In addition to generating the output waveform, these PWM-relevant modules also serve as trigger sources for ADC sampling, ensuring precise synchronization between sampling and control. In the following, we first focus on how to configure the PWM modules as ADC trigger sources.

ADC Trigger Implementation

Although the configuration methods differ between TI's MCUs and Infineon's, the basic principle of setting a trigger source is the same. PWM trigger ADC sampling depends on the three counting modes. In up-count mode, the interrupt is usually triggered when the counter reaches the top of the period. In down-count mode, the interrupt is triggered when the counter returns to zero. In up-down count mode, the interrupt can be triggered at the middle point during both the up and down counting phases. We can see these three count modes in Fig. 4.6, and for our configuration, we use the up-count mode.

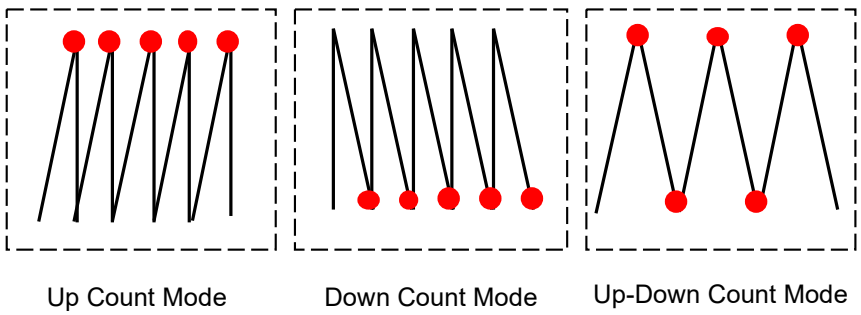

Figure 4.6: Three common PWM count modes

For the TI F29H85x and AM263x platforms, the ePWM module is used. In the actual configuration, the ePWM is set to generate a SOC event when the counter counts up to a defined compare value, as shown in Fig. 4.7. This ensures that the ADC samples at a fixed point in each PWM cycle, which gives the controller

consistent and predictable feedback. To help with debugging and performance measurement, the SOC event is also sent to a GPIO pin. When the trigger happens, the GPIO level changes, and changes again when the counter resets. This makes it easier to observe the sampling time and system delay on the oscilloscope. On the other hand, by setting the time base clock and high-speed clock of the ePWM module, we can control the PWM frequency according to (2.5) and (2.4). This allows us to trigger ADC sampling at the desired frequency.

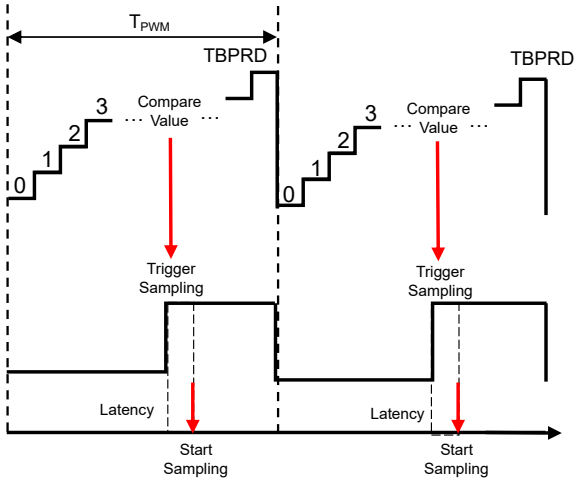

Figure 4.7: How does ePWM trigger the ADC sampling

Similarly, on Infineon's TC4x platform, the GTM is used as the trigger source (and also for PWM output). In common setups, the trigger is usually generated by the TOM sub-modules inside the GTM as shown in Fig. 4.8. The GTM includes a common time base unit (TBU), which provides a unified time reference for the TOM sub-module. With this time base, events across different sub-modules can be precisely synchronized and timestamped, ensuring consistent timing for critical control actions.

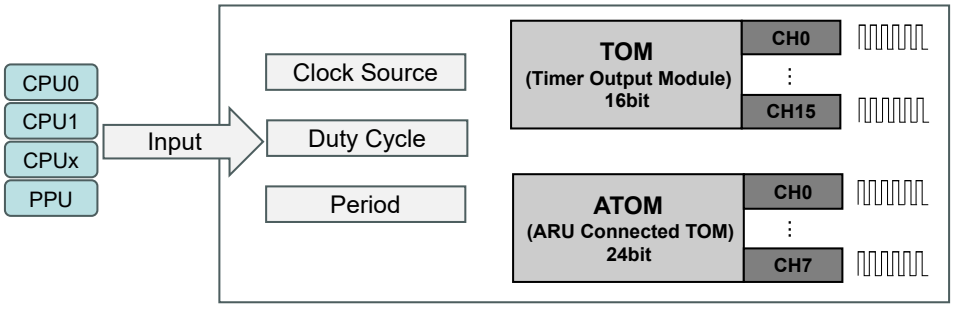

Figure 4.8: PWM generation by GTM (Source: Infineon Technology). [43]. Adapted with permission.

For the TOM configuration, the process is also more straightforward. After enabling the GTM and its clock, the specific TOM channel is directly configured with parameters such as frequency, duty cycle, and output pin. Then, the TOM channel

output (i.e., the PWM signal) is mapped as the trigger source for the ADC, establishing a hardware trigger connection. Each time the TOM PWM reaches the defined condition (such as a specific edge or counter value), a sampling trigger event is automatically generated to initiate TMADC sampling.

PWM Output Implementation

Configuring the PWM module to adjust its duty cycle based on the controller output is very similar to setting it as a trigger source. The key steps are still setting the clock source, defining the period and duty cycle, and selecting the output signal.

On TI's F29H85x and AM263x platforms, we set the PWM frequency to 10 kHz, choose an initial duty cycle of 50%, select the system clock with suitable dividers, and use the up-count mode to generate the signal. During runtime, the controller gives an output value as shown in (4.6). To map this to a 0% to 100% duty cycle, we normalize the output and multiply it by the PWM period register (TBPRD). The result is the new compare value in (4.7), which we write into the PWM register. In this way, the ePWM module updates its duty cycle in real time based on the controller output and responds to changes in the system.

$$\text{Compare Value} = \text{Normalized Control Output} \cdot \text{TBPRD} \tag{4.7}$$

For the PWM output configuration using the GTM module on Infineon's TC4x platform, the process is almost the same as when configuring it as a trigger source. The duty cycle is calculated by normalizing the controller output and writing the result directly into the related register. To make a fair performance comparison, all three MCUs use the same PWM configuration: the same frequency, initial duty cycle, and output mode.

4.2.4 Interrupt-Driven Control Execution

In this project, the control loop is implemented using an interrupt-driven approach. Once the ADC completes a conversion, it automatically triggers an interrupt. The system then jumps to the ISR, where the control algorithm is executed and the PWM output is updated, forming a complete control cycle.

This mechanism offers better real-time performance and timing stability. The sampling time of the ADC is precisely controlled by a hardware timing module (such as ePWM or GTM), ensuring consistent sampling points in every control cycle and leading to more stable feedback signals. The interrupt mechanism makes sure to immediately process once sampling is done, reducing delay and minimizing timing jitter.

In our project, the interrupt-based control flow includes the following steps:

- **PWM triggers ADC sampling:** The ePWM/GTM module is configured to generate a trigger signal when the timer reaches a predefined compare value.

This ensures that the ADC samples at the same point in every PWM cycle and happens at a fixed frequency, improving data stability and predictability.

- **ADC triggers an interrupt after conversion:** When the ADC finishes the sampling and gets a digital result, it sets an interrupt flag. The system detects this and enters the related ISR.
- **Control algorithm runs inside the ISR:** Inside the ISR, the system reads the ADC result and compares it to the target value. It then runs the PID control calculation and gets the new control output.
- **Update PWM output:** The system uses the control result to change the PWM duty cycle by updating the compare value. This controls the on-time of the power switch in the DC-DC converter and helps adjust the output voltage.

This interrupt-driven control method has a simple structure, fast response, and improved timing stability for closed-loop systems. Since the entire control logic runs within the ISR, the system benefits from lower delay and reduced susceptibility to interference. As a baseline implementation, it provides a solid foundation for comparing the performance differences introduced by various MCUs' peripherals, CPUs, and overall architectures. It also sets the stage for subsequent delay analysis and optimization.

4.3 System Delay Analysis

In addition to the baseline implementation, minimizing overall system delay is crucial for achieving fast, stable, and accurate control performance. This total delay arises from the coordinated operation of several system components, including ADC sampling and conversion delay, control algorithm execution time, interrupt response time, and memory access delay. To systematically analyze their impact on the control period, this section breaks down each component, quantifies its contribution to the total delay, and identifies potential performance bottlenecks, providing a foundation for further system-level optimizations.

4.3.1 ADC Delay

As previously discussed, the ADC module mainly performs two steps: sampling and conversion. A longer sampling time can provide more accurate results, but it also increases the delay. After sampling, the ADC converts the analog signal into a digital value, which also takes time. In control system design, it is important to balance sampling accuracy and time cost.

Each MCU usually has a minimum sampling window to ensure accuracy. Although our three MCUs have different limits on minimum sampling time, this minimum sampling time often represents an ideal condition—hardware components are assumed to be free from aging or other issues, and software must be carefully tuned to meet tight timing constraints. This makes it a high-risk configuration in practice. Therefore, we adopted a more conservative approach by setting the same sampling

time longer than each minimum sampling time in subsequent tests, which is 80 ns for each MCU. This not only avoids potential risks to sampling accuracy, but also, since the conversion time of the TC4x is not explicitly known, helps prevent introducing additional variables that could compromise the fairness and accuracy of the final comparison. For the conversion part, MCUs like the Infineon TC4x series do not offer specific conversion time values. This lack of data is one of the current limitations of our study, but the overall ADC delay can still be measured.

Therefore, under the same ADC clock frequency, sampling window length, and 12-bit resolution, measuring the actual delay from ADC trigger to conversion completion can compare ADC performance across platforms.

4.3.2 Interrupt Response Delay

After the ADC completes the conversion, an interrupt is triggered according to the system configuration. However, there is a delay between the interrupt request and the moment the CPU actually enters the ISR. Normally, once the ADC raises an interrupt request, the CPU or interrupt controller first checks and detects the event. If multiple interrupts occur at the same time, a priority arbitration is performed to select the appropriate one, especially in systems with nested or grouped interrupts.

Once the interrupt is accepted, the CPU must pause the current task and save the context, including general-purpose registers, program counter, and status flags, to ensure the system can resume correctly after the ISR. Then, the program jumps to the corresponding interrupt vector and begins executing the interrupt handling routine. The delay introduced in this process depends not only on the complexity of the interrupt configuration the system needs, but also on the CPU architecture and performance, factors such as pipeline depth, cache behavior, and interrupt vector handling contribute to the total delay. This stage is therefore a key part of control loop timing analysis and may differ between our three MCU platforms.

4.3.3 ISR Execution Delay

In our closed-loop control system, once the CPU starts to handle the ISR, it performs three main tasks: reading the latest ADC sampling result, executing the PID control algorithm, and converting the control output into a PWM duty cycle to complete the modulation process.

The delay in this process consists of data access delay and computation execution delay. For data access, the ISR involves reading the ADC result and updating the PWM registers. These read/write operations from peripherals or memory can lead to differences in execution speed across different MCUs. As for the computation itself, the execution speed of the control algorithm is closely related to factors such as the presence of the FPU, the CPU clock frequency, the instruction execution, and the instruction issue capability. In addition, the memory location where the

code is executed also affects its performance; for example, functions stored in Flash generally suffer from slower access times compared to those in RAM. Similarly, the choice of compile mode impacts execution speed. the aggressive optimizations provide inlining and loop unrolling, which help reduce instruction count and lead to faster execution. These factors together define the baseline execution time of the ISR and highlight the architectural differences between MCUs.

4.3.4 Measurement Methods

For the timing measurement of code execution time, there are three common methods. The first is to use the MCU's built-in timer modules or operating system timing functions to read timestamps at the beginning and end of the code and calculate the time difference. The second is to use performance analysis tools integrated into IDEs, which can automatically collect function runtime, cycle counts, or event frequencies in debug mode.

However, both methods introduce extra variables when comparing different MCUs: timer frequency, width, and precision vary across platforms, and the overhead of calling time functions is also different. Similarly, IDE-based profiling depends on the toolchain, debugger speed, and connection type. Because of these inconsistencies, we choose to measure execution time using the third way, GPIO toggling, even though this method also introduces some delay as shown in Fig. 4.9. The method we used in our design is GPIO toggle, which has already been mentioned before. However, changing the state of a GPIO usually needs several write operations to its control registers. The time it takes to finish these operations depends on the MCU's system clock speed, bus delay, and how the instructions are processed. Since GPIO control often works through accessing special IO registers, there may also be delays caused by bus traffic or wait cycles during access.

Table 4.2: General-purpose output switching characteristics [31]

PARAMETER			MIN MAX	units
t_rise	Rise time, GPIO switching low to high	All GPIOs	8	ns
t_fall	Fall time, GPIO switching high to low	All GPIOs	8	ns
t_freq	Toggling frequency, GPIO pins		50	MHz

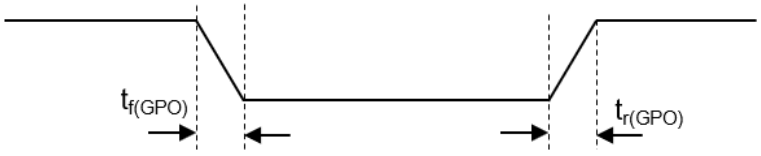

Figure 4.9: Measurement delay from general-purpose output

This kind of delay sometimes is not fixed, for example, the hardware datasheet from F29H85x says the rise time and fall time are both 8ns, as shown in Table 4.2, this

is only the best-case value in ideal conditions. In real situations, things like PCB layout, load from connected components, or even how the probe touches the board can make the waveform edges unclear; different compiler optimization levels also introduce differences in delay. This will reduce the accuracy of time measurements. To reduce this kind of error in our project, we first let the GPIO toggle once by itself, then subtract the delay measured in this test from later measurements. This helps cancel out the error and make our results more accurate.

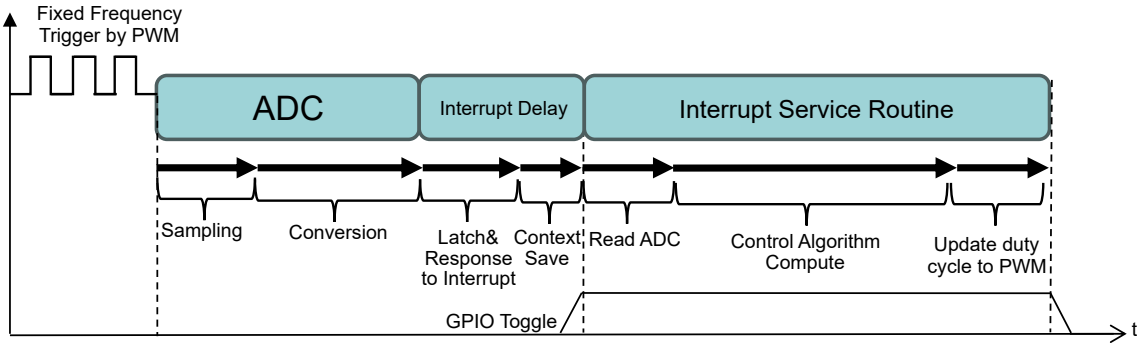

Figure 4.10: Closed-loop control system delay analysis

Based on the above discussion of the delay from each stage, the delay analysis is also shown in Fig. 4.10. In our measurement, the PWM trigger marks the start, and the GPIO toggle shows only the execution inside the ISR. The delay between the trigger and ISR entry can be estimated as a whole, but we set the sampling time, and the conversion time is taken from the user manual; with both known, the interrupt delay can be derived. Inside the ISR, each step can be measured with extra toggles. This method allows us to estimate the time cost of each stage and provides useful references for later optimization.

4.4 Optimization Strategies

Baseline comparison can only show the raw performance of each platform in the most basic setup. However, MCU vendors have different default settings, hardware features are not used, and the setup is different from real applications. Therefore, it cannot represent the best or real performance of the platform. This is why we analyze the delay to make full use of each MCU's strengths and optimize execution time.

Based on the delay analysis in section 4.3, this chapter presents a set of optimization methods to improve the overall performance of the control loop. The goal is to reduce delay in key paths, increase execution speed, and raise the control frequency as much as possible. The optimization strategies are discussed for three platforms. Each part focuses on the available peripheral resources, module settings, and special features of the MCU to design targeted improvements. The methods include ADC setting adjustment, DMA usage, interrupt delay reduction, use of TCM (or similar fast memory), hardware accelerators, and compiler optimization options. For each MCU, the possible resources and design choices are carefully analyzed.

4.4.1 Optimization on F29H85x

To improve control performance on the F29H85x platform, multiple optimization methods were tested. These include early interrupt, RTDMA, RTINT, TMU, and RAM execution. Each method targets a specific stage in the control loop, and helps reduce delay or CPU load depending on the optimization type [44, 45].

Early Interrupt mode Implementation

In the ADC configuration of the F29H85x, TI provides a unique interrupt trigger strategy called early mode. This mode allows the ADC to send an interrupt signal before the conversion is fully completed. The main idea is that once the ADC finishes sampling, it can notify the CPU early that the result will soon be ready. This helps reduce the CPU's waiting time, improves the interrupt response speed, and allows the CPU to prepare for entering the ISR earlier. This feature is useful for high-speed control systems that require fast response.

However, the early mode also has potential risks. If the CPU or DMA tries to read the ADC result before it is written into the register, it may read the previous cycle's value, leading to incorrect data and a race condition. This issue is harder to detect when using DMA for automatic data transfer, but for F29H85x, this issue will be solved by the unique hardware design that we will show later in the RTDMA part.

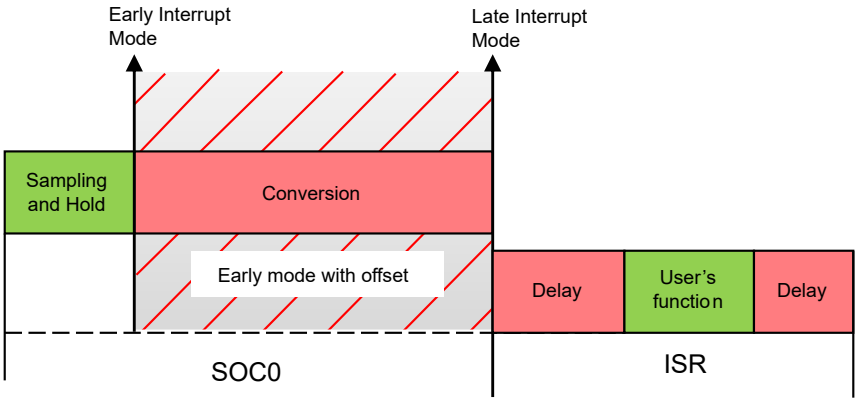

Figure 4.11: ADC sampling offset setting

In addition to that, TI also provides a common related feature: in early mode, users can set an interrupt offset like in Fig. 4.11. This means delaying the read access by a fixed number of sample cycles after the interrupt, to make sure the result has been written. This mechanism combines the fast response of early mode with safe data access, avoiding read errors while keeping high performance.

Therefore, the combination of early mode and interrupt offset offers a good balance between speed and reliability for TI's ADC in high-speed closed-loop control systems. In our project, we use this method on the F29H85x ADC module to reduce interrupt delay.

RTINT Optimization

Based on the previous introduction of the F29H85x interrupt system, real time interrupt (RTINT) has a clear advantage over the normal INT interrupt in terms of ISR entry delay.

This is mainly because the context save and restore for RTINT is done by hardware automatically. Unlike INT, which needs several software instructions to push and pop registers, RTINT uses hardware to complete this process, which reduces the time needed to enter and exit the interrupt. In addition, the PIPE module in F29H85x supports dynamic priority arbitration. RTINT can be set to a higher priority. When multiple interrupts happen at the same time, the PIPE module checks and selects the highest priority interrupt every clock cycle, and sends that vector to the CPU. The RTINT signal goes through its own independent path to the CPU, avoiding the shared and delayed INT path. This path is designed in hardware for fast response and helps reduce the time between trigger and ISR entry.

Because of this, RTINT offers a fast, flexible, and reliable interrupt handling method. It is a unique optimization feature of the F29H85x platform, especially useful for real-time control tasks.

DMA Implementation

On the F29H85x platform, to reduce control loop response delay and lighten the CPU workload, this project uses the real-time direct memory access (RTDMA) module to build a high-speed data transfer path. The parts most relevant to the control loop include the trigger source configuration, channel register settings, interrupt handling, and memory access control.

RTDMA works based on event-triggered mechanisms. We configure the ADC-conversion-complete signal as the DMA trigger source. This way, every time the ADC finishes one conversion, RTDMA immediately starts transferring the data. The DMA channel registers are set to read the result from the ADC result register and write it into a register address used by the PID controller. In our experiment, only one sample is transferred each time, and after each transfer, an interrupt is automatically triggered to enter the ISR and run control logic. This whole process avoids any CPU involvement in data transfer, which improves efficiency. After the transfer is complete, RTDMA sends a signal to the interrupt controller, which then triggers the interrupt. In the ISR, the ADC result has already been transferred, performs the PID control calculation, and updates the PWM output, completing one control loop. During this process, the built-in MPU of RTDMA ensures that only valid memory regions are accessed, and the arbiter guarantees stable channel scheduling.

Another important feature in the RTDMA configuration on the F29H85x platform is the *tDMA* trigger mechanism. It is specifically designed to ensure that the DMA

transfer only starts after the ADC result has been written to the result register. Even when the ADC is configured in early mode, which allows an interrupt to trigger before the conversion result is ready, the *tDMA* trigger still waits until the data is valid before initiating the DMA transfer. This prevents race conditions or incorrect transfers and provides reliable data synchronization between the ADC and DMA in fast control systems.

TMU Assistant

Without changing the control structure, we can still reduce the execution time in the control path by using the built-in hardware instructions. F29H85x has a built-in trigonometric math unit (TMU) that can accelerate math functions like sine and cosine. When the compiler flag `-ffast-math` is enabled, the compiler replaces the normal function calls with TMU instructions. This skips the slower software functions and removes the need for call and return steps, making the calculation faster. In addition, the FPU is enabled by default and supports fast hardware-level float operations like add, subtract, multiply, and divide. This helps keep the floating-point calculation delay low and ensures that the time spent in ISR remains as short as possible.

4.4.2 Optimization on AM263x

To reduce the total control loop delay on the AM263x platform, several optimization techniques were tested, including early interrupt, DMA transfer, fast interrupt routing, memory placement, and compiler tuning. These methods targeted different stages of the loop and helped improve execution speed step by step.

Early Interrupt Mode Implementation

On the AM263x platform, the ADC module is architecturally the same as on the F29H85x, since both come from TI. As long as the basic configuration, such as input channel, sampling time, and trigger source, is aligned, some optimization strategies can be directly applied to both platforms. For example, the early mode with the result offset technique can also be used on AM263x to reduce delay and improve response time. However, this also brings similar challenges, such as the risk of data being transferred by DMA before it is ready. These issues will be further discussed and addressed in the DMA sections.

VIM Optimization

The AM263x is based on the ARM Cortex-R5F core, and its interrupt path is more complex. By default, TI's software framework adopts a shared interrupt dispatcher architecture. All interrupt signals are first routed to a central handler, which then identifies and invokes the corresponding user-defined ISR. While this approach provides flexibility and modularity, it introduces extra software-layer delay due to indirect function lookup and context management.

In addition, this interrupt response involves multiple steps: saving the processor state, adjusting stack alignment, clearing the interrupt flag in the VIM, and enabling nested interrupts before finally calling the user ISR. After the ISR is executed, the system must reverse these steps—acknowledging the interrupt, restoring the stack and registers, and resuming the main program. These layers of software processing increase delay compared to hardware-managed mechanisms like those on the F29H85x.

To reduce interrupt delay on the AM263x, we will use several optimization methods offered by TI for the ARM R5 architecture:

- **Dedicated ISR Functions:** Instead of using a shared dispatcher, each interrupt can be registered with a dedicated entry function. This allows the CPU to jump directly to the ISR, avoiding look-up and indirect calls, and reducing response time.
- **Naked ISR Functions:** Makes the function have no default context handling. Then manually saves and restores only the necessary registers, reducing overhead.
- **Vectored Interrupt Controller (VIC):** The VIM supports direct vectoring by passing the ISR address to the CPU through the VIC port. Instead of always branching to address 0x18, the CPU can jump directly to the correct ISR based on interrupt priority.
- **Interrupt Flag Clearing:** To avoid self-triggering and race conditions, the interrupt source must be manually cleared before re-enabling global interrupts. This ensures the ISR is not immediately re-entered.

Through the combined optimizations above, we can achieve low-latency interrupts, greatly improving the system's real-time control performance and execution efficiency.

DMA Implementation

On the AM263x platform, the DMA used is the enhanced direct memory access (EDMA) module. The EDMA consists of two main parts: the channel controller and the transfer controller. The channel controller handles events from peripherals, such as ADC conversion complete, manages transfer requests, and sets up the *PaRAM* parameter table. The transfer controller performs the actual data read and write tasks, moving data from the source address to the destination address.

EDMA supports up to 64 DMA channels and offers features like chained transfers, automatic address updates, and multiple sync modes. However, compared to the RTDMA on the F29H85x platform, the EDMA configuration on AM263x faces main problem: from the official hardware issue (errata i2355 [46]), which may cause the DMA to start transferring before the new ADC result is written, leading to reading old data. Since the EDMA cannot be set to wait for ADC conversion to finish like on the F29H85x, this project adds an empty transfer channel to avoid the problem.

After ADC sampling is done, the empty channel runs a dummy transfer first to refresh the data state. This ensures that the actual transfer channel reads the correct value.

Optimization with TCM

On the AM263x platform, TCM is a low-latency, high-bandwidth memory directly connected to the CPU. It is designed to support real-time systems that need fast and predictable access. Unlike Flash or regular RAM, TCM does not pass through the system bus or cache (such as L1 or L2), which helps reduce access delay and avoid cache miss. Instructions or data stored in TCM can be fetched in just one or two clock cycles. This makes TCM ideal for time-critical tasks like interrupt handling and control logic.

When running in single-core or lockstep mode, each core of the AM263x can use the full 128KB TCM. TCM is divided into two parts: instruction tightly coupled memory (ITCM) and data tightly coupled memory (DTCM). We can assign specific functions and variables to TCM using linker scripts or attribute macros. In this project, we place the interrupt functions and control algorithms in ITCM to shorten execution time. Meanwhile, important data such as ADC results and control outputs are stored in DTCM to ensure fast access and stable operation. This setup improves the real-time performance and reliability of the control system.

4.4.3 Optimization on AURIX TC4x

The AURIX TC4x platform is based on a more complete default setup, and because it is a newer product, many modules have not yet been fully explored. Therefore, the known optimization methods are relatively fewer, but we will still evaluate MCU's unique hardware modules to find possible optimization solutions. To improve the control performance, several hardware features were tested, including SDMA for automatic data transfer, CDSP for offloading floating-point control, and PPU for standalone execution. Each approach provides different advantages in speed and structure.

DMA Implementation

Similarly, the AURIX TC4x platform is equipped with its own DMA module in Fig. 4.12, known as the system direct memory access (SDMA). Inside the SDMA, each SDMA channel has a corresponding transaction control set (TCS), which stores the source and destination addresses, transfer length, data width, and other configuration parameters. After the TMADC completes a sampling operation, it sends a hardware request to the SDMA to initiate a transfer. This request is routed through the IR to the SDMA channel request and arbitration module. The SDMA then performs priority-based arbitration across all pending channels and assigns the task to an available move engine.

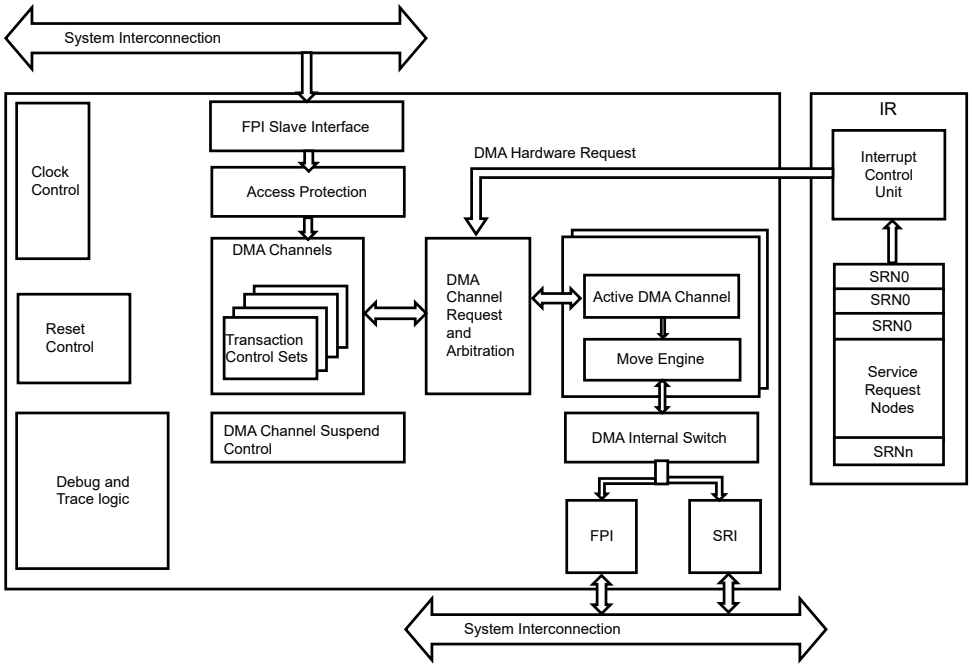

Figure 4.12: SDMA structure (Source: Infineon Technology). [32]. Adapted with permission.

The move engine accesses internal memory or peripherals through the internal switch and system interconnect, executing the transfer independently of the CPU. Once the data movement is complete, the SDMA generates an interrupt via the channel completion interrupt interface to the IR.

The overall configuration is the same as the DMA usage on the other two MCUs: the ADC end-of-conversion event is used to trigger the DMA transfer, and the DMA completion then triggers an ISR, where subsequent control operations are performed to form a closed-loop system. The safety issue we discussed before can be solved by the ADC setting here. The TMADC in AURIX TC4x can be set to *wait-for-read*, which means that before the result register is read by the CPU or DMA, no new sampling result will be written, to avoid new data from overwriting unread data.

CDSP Implementation

CDSP is an on-chip digital signal processing unit used in Infineon's AURIX TC4x series. It is designed to replace external DSPs. The CDSP structure shown in Fig. 4.13, the core is based on the ARC EM5D architecture, with a 3-stage pipeline, instruction closely coupled memory (ICCM), data closely coupled memory (DCCM), and features like timestamping, interrupt control, and FIFO buffering. It can reach up to 5202 DMIPS and supports up to 18 independent CDSP units.

CDSP can process input signals directly from delta-sigma analog-to-digital converter (DSADC), external modulator (EXMOD), TMADC, carrier modulator generator (CARMAG), or general-purpose registers. It is suitable for fast control and real-

time signal processing tasks. The results are stored in RES0 to RES2 registers or in the DCCM. RES0 also has a 4-level FIFO buffer to support continuous output. CDSP is a separate module and is controlled by the TriCore CPU, which manages its start, reset, and operation.

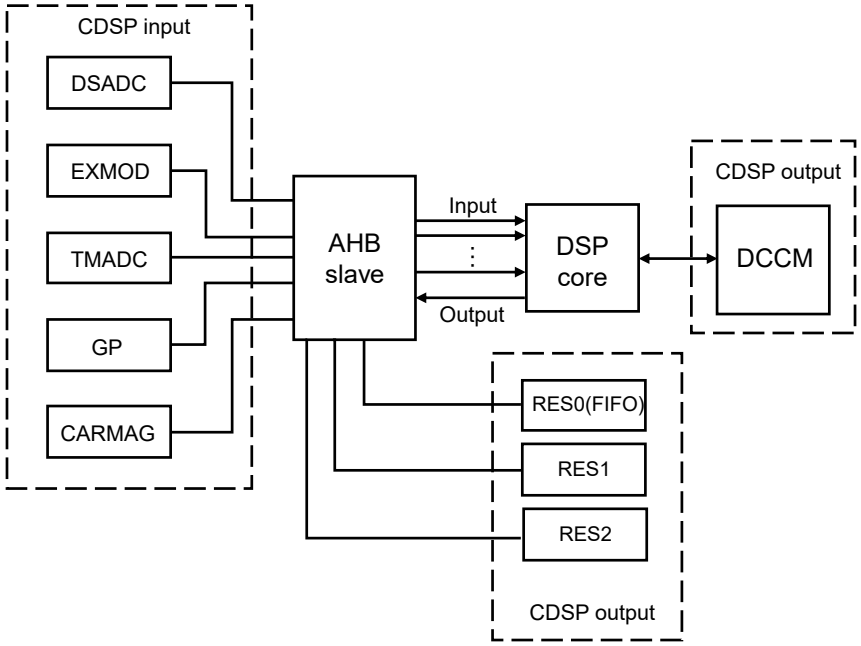

Figure 4.13: CDSP structure [47]

After the CDSP is added, the control loop design changes: the CDSP offloads the control algorithm calculations to improve execution speed and reduce CPU load. The process starts with CDSP configuration. We use the *CDSP Framework* to convert the control algorithm and its parameters into machine code and store them in ICCM and DCCM. This conversion means compiling the C code into executable code and data that the ARC EM5D core can run.

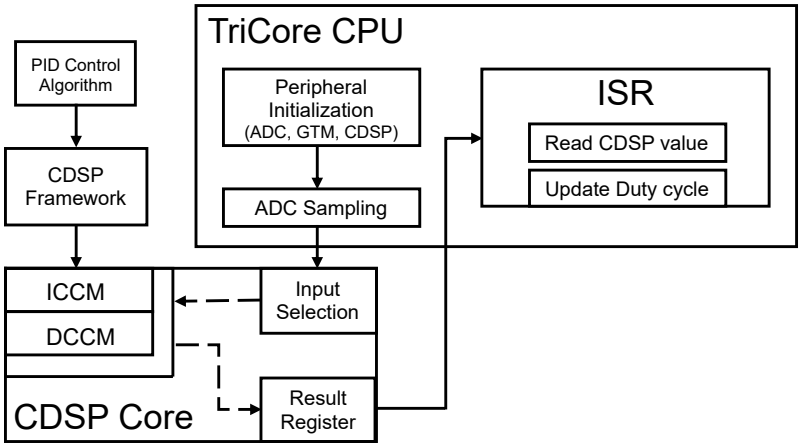

Figure 4.14: CDSP workflow in closed-loop control system

After the configuration, the CDSP is connected to a specific TMADC channel, and the trigger source is set. When the CPU starts running, once the TMADC finishes sampling, it will automatically trigger the CDSP. The CDSP then reads the ADC result and runs the control algorithm inside. After the calculation, the CDSP sends an interrupt. In the ISR, the CPU reads the output from the CDSP and uses it as the control result. This value is then used to set the PWM duty cycle through the GTM module, completing the full control loop, the whole workflow can be seen in Fig. 4.14. The previously mentioned C29x DSP core used in the F29H85x series is itself a processor architecture optimized for time-sensitive tasks. In contrast, the CDSP in the TC4x acts as a separate sub-processor that offloads control computations, enabling a clearer modular design and parallel processing. This architectural difference reflects two mainstream strategies for control optimization and highlights the scalability advantage of CDSP in multi-core and high-concurrency applications.

PPU Implementation

To further improve the execution efficiency of control algorithms and reduce the load on the main core, the TC4x platform provides another optional co-processor called the PPU. The PPU combines a scalar core and a vector core, and supports single instruction multiple data (SIMD) and VLIW architectures. It is designed for high-performance parallel computing and is well-suited for complex, computation-heavy control tasks.

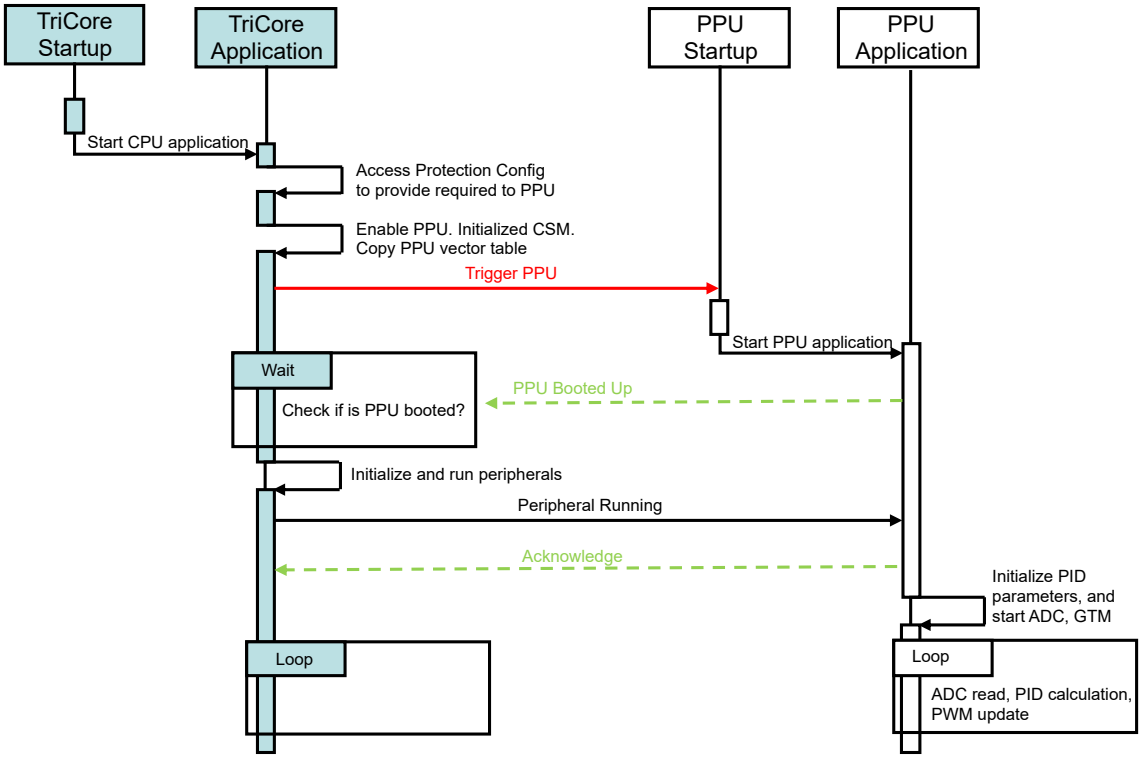

Figure 4.15: PPU workflow in closed-loop control system

The PPU includes a level-1 cache for fast access to input and output data. In addition, the PPU can communicate directly with peripherals. This means it can independently perform data sampling, algorithm processing, and result output—without needing help from the main TriCore CPU—achieving full offload of the control process. In the use of PPU as shown in Fig. 4.15, we first need to start the main core application (TriCore) and prepare for cooperation with the PPU. In the CPU, access permissions are configured to allow the PPU to access memory and required peripherals. Then, the PPU is triggered to start.

After power-up, the PPU completes CSM initialization and self-check, and sets a special flag to synchronize with the CPU. Once the PPU application starts, it initializes GTM, TMADC, and PID parameters and then runs the closed-loop control. Another possible method is to let the CPU initialize the peripherals first and allow the PPU to access them later. After the CPU and PPU confirm the peripheral status through synchronization flags, the PPU starts the real-time control task. The closed-loop part still follows the same steps as before: reading ADC (TMADC) sampling results, executing the PID control algorithm, converting the output to PWM duty cycle, and updating the PWM output through the GTM module.

4.4.4 Compiler Optimization

Compiler optimizations help improve the speed and size of the program without changing its behavior. They are especially useful in real-time embedded systems where performance and code size are both critical. These optimizations can be divided into two types: manual settings and automatic compiler behavior based on optimization level.

Manual optimization settings are settings we can choose and apply ourselves based on the different MCUs:

- **Running Code in RAM:** F29H85x runs code from Flash by default. But Flash has a slower access speed than RAM, which can delay interrupt handling. By running critical code from RAM, we reduce access delay and improve execution speed.
- **Release Mode:** In AM263x, compared to debug mode, release mode enables more compiler optimizations. It removes debug symbols and applies code simplification, which reduces instruction count and improves speed.
- **Link Time Optimization:** Link time optimization (LTO) performs extra optimizations at the linking stage. It can remove unused functions across files and optimize function calls across modules. This is helpful in control paths that involve many small function calls.
- **FPU Support:** The MCUs we used in our project include FPU, and compilers usually enable it by default. The FPU handles floating-point operations like add, subtract, multiply, and divide using hardware. This gives faster and more stable performance for control tasks like PID and reduces time spent in the ISR.

The compiler also applies some optimizations automatically depending on the chosen optimization level. Common optimization techniques include:

- **Function inlining:** Small and frequently used functions are replaced with their actual code. This removes the overhead of function calls and speeds up execution, which is useful for tight control loops like PID.
- **Dead code elimination:** Code that is never used or never executed is removed. This reduces the program size and helps save memory and CPU resources, especially important on microcontrollers with limited space.
- **Loop optimization:** The compiler may unroll loops, simplify loop conditions, or replace slow operations with faster ones (e.g., bit shifts instead of multiplication). These changes reduce the number of CPU cycles per iteration.
- **Constant folding and propagation:** If the value of an expression can be calculated during compilation, the compiler replaces it with a constant. This reduces runtime calculations and makes the code more efficient.
- **Register allocation:** Frequently accessed variables are stored in CPU registers instead of RAM. Since register access is much faster than memory access, this helps reduce execution delay.

These optimizations are performed by the compiler after translating the source code into an internal representation. The extent of optimization depends on the selected compiler flag:

- `-O0`: No optimization. Used for debugging and development.
- `-O1`: Basic optimizations that do not affect debugging much.
- `-O2`: Good balance between performance and code size. Enables most useful optimizations.
- `-O3`: Enables all high-level optimizations focused on performance, including aggressive inlining and loop transformations.
- `-Os`: Optimizes for small code size, which is useful when memory space is limited.

In this project, we selected `-O3` because it can maximize the execution speed of the code, which is especially suitable for evaluating the performance of different MCUs. However, the `-O3` optimization level may sometimes introduce potential issues, such as debugging difficulties due to function inlining, timing problems caused by aggressive optimizations, or unexpected behavior from variable optimizations. To avoid these issues, we added the `volatile` keyword to key variables to prevent incorrect optimization and ensured that the system remained stable and reliable after applying the optimizations.

5

Results

In this part, we first explain how we arrange our results from experiments, and show the baseline performance of three MCUs in a control loop, including the time spent on sampling, conversion, ISR entry, and each step inside the ISR, such as reading, calculation, and PWM update. Then, we apply both general and platform-specific optimizations to each MCU and compare the results before and after optimization. In the end, we combine all proven optimizations to show the best possible execution time for each MCU. Note that common optimizations used on all three MCUs will be skipped here if they do not lead to clear improvements.

5.1 Structure of the Result Tables

This section explains how we correct the measured data and describes the structure of the result tables and the meaning of each component.

5.1.1 GPIO Delay Correction

As discussed in section 4.3.4, our measurement method will introduces some errors, but we can reduce errors from hardware with simple steps.

Table 5.1: GPIO delay for different MCU platforms

Platform	GPIO Delay (ns)
F29H85x CPU	10
AM263x CPU	25
TC4x CPU	60
TC4x CDSP	50
TC4x PPU	60

As shown in Table 5.1, we measured the GPIO toggle delay—the fixed delay used to mark the start and stop on the oscilloscope. For any later data measured with this method, we subtract this delay for the corresponding platform.

5.1.2 Result Table Explanation

For all ensuing result tables, each table group steps by stage, lists each operation with its delay, and gives a stage subtotal for that stage. The explanations listed:

- **Stage:** ADC module covers from the start of sampling to the moment the interrupt is triggered; the ISR shows the work inside the interrupt context.
- **Operation:** In our project, it maps to a set of instructions/functions, to make a fair MCU-to-MCU comparison and reflect our optimization works.
- **Delay (ns):** The time of a specific step. It comes from the oscilloscope readings and from values inferred based on the stage subtotal.
- **Stage Subtotal (ns):** The total time within the same Stage, taken from the oscilloscope measurement.

Introducing the optimization design may change the table layout, for example, a stage will add new hardware blocks. But the rows stay in execution-time order, so it should still be easy to read. The explanations of the basic operation items are as follows:

- **S+H:** Time for the ADC to sample the input and hold the voltage during conversion.
- **Conversion:** Time for the ADC to do the analog-to-digital conversion.
- **Interrupt delay:** Time from conversion finish to the CPU getting the interrupt and entering the ISR.
- **Result read:** Time to read the conversion result from the result register in the ISR.
- **PID control:** Time for the PID control calculation in the ISR.
- **PWM update:** Time to write the new control value to the PWM in the ISR.
- **Clear flag:** Time to clear the interrupt flag.
- **Path interaction:** Time change caused by overlap or resource contention on the real instruction path.

The **Cross-stage path interaction** shows the time change caused by the real instruction path across stages. Its meaning is the same as **Path interaction** in the Operation list. We will discuss it in section 6.1.3 later.

End-to-End total delay (ns) is the total time of the whole control loop. It comes directly from the oscilloscope and is the most important result.

5.2 Baseline Performance Comparison

This section measures the baseline control loop delay of each platform before any optimization. The test results for the three platforms are shown below.

5.2.1 Baseline Control Delay Measurement of F29H85x

The ADC sampling time is 80 ns, and the conversion time is 205 ns, giving a total of 566 ns for the ADC part, including the interrupt delay is 281 ns. Inside the ISR, the steps include reading the ADC result (20 ns), PID calculation (470 ns), PWM

update (175 ns), and clearing the interrupt flag (45 ns), making the total ISR time 726 ns. The total delay of the control loop is 1270 ns. The result is organized in Table 5.2.

Table 5.2: Baseline control loop implementation of F29H85x

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	566
	Conversion	205	
	Interrupt delay	281	
ISR	Result read	20	726
	PID control	470	
	PWM update	175	
	Clear flag	45	
	Path interaction	16	
Cross-stage path interaction		-22	-22
End-to-End total delay (ns)		1270	

5.2.2 Baseline Control Delay Measurement of AM263x

On the AM263x platform, Table 5.3 shows the baseline time for the AM263x. The ADC module takes 1044 ns in total; the interrupt delay is 759 ns, which is slightly higher than the previous platforms. Inside the ISR, the steps include reading the result (85 ns), PID control (195 ns), PWM update (150 ns), and clearing the flag (95 ns), adding up to 535 ns. The total control loop delay is 1584 ns.

Table 5.3: Baseline control loop implementation of AM263x

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	1044
	Conversion	205	
	Interrupt delay	759	
ISR	Result read	85	535
	PID control	195	
	PWM update	150	
	Clear flag	95	
	Path interaction	10	
Cross-stage path interaction		5	5
End-to-End total delay (ns)		1584	

5.2.3 Baseline Control Delay Measurement of AURIX TC4x

On the AURIX TC4x platform, the ADC sample-and-hold time is 80 ns. Since the exact conversion time is not available from Infineon, it cannot be directly shown.

However, the total time from sampling to entering the ISR is observed to be 458 ns. Inside the ISR, the steps include reading the ADC result (80 ns), PID calculation (60 ns), and PWM update (140 ns), making a total of 380 ns. The overall delay of the control loop is 828 ns; all time costs in the baseline implementation for TC4x can be seen in Table 5.4.

Table 5.4: Baseline control loop implementation of AURIX TC4x

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	458
	Conversion	N/A	
	Interrupt delay	N/A	
ISR	Result read	80	380
	PID control	60	
	PWM update	140	
	Path interaction	100	
Cross-stage path interaction		-10	-10
End-to-End total delay (ns)		828	

5.2.4 Baseline Performance Comparison

We evaluated the data from three tables for the MCUs and summarized them in Fig. 5.1. We separate the execution into three parts: the first is the ADC module, which includes the sampling, conversion time, and the delay to entering the ISR, then is the PID algorithm, which shows the time taken by our control algorithm, and the final one, is the register access, which includes the read operation to result registers from ADC and PWM relevant modules. After that, we do the comparison to the entire control-loop execution time.

The F29H85x platform has a short ADC delay (566 ns) and a medium interrupt delay (280 ns). However, the PID calculation inside the ISR takes the longest time (470 ns) among the three. The total control loop delay is 1265 ns, which is at a medium level. The AM263x platform has the longest ADC delay (1044 ns), mainly because the interrupt delay is very high (759 ns). This makes the ISR start slowly, even though the PID and PWM steps inside the ISR are faster (195 ns and 150 ns). The total delay is 1584 ns, the highest of all three. The main problem of AM263x is in the way it accesses peripherals and handles interrupts. The TC4x platform has the shortest total delay, only 828 ns. It enters the ISR quickly, and the PID calculation inside the ISR is very fast (only 60 ns). The ADC time is 458 ns, also better than the other two platforms.

We can see the entire control loop execution time for three MCUs in Fig. 5.2. All the baseline results reveal that each platform has its unique performance bottleneck. These differences highlight the need for targeted optimization. In the following

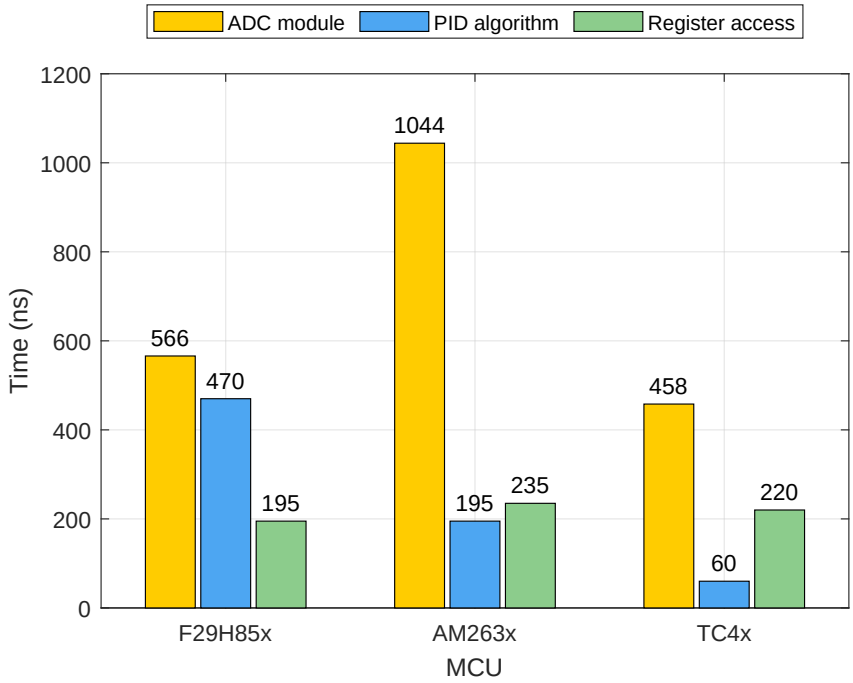

Figure 5.1: Baseline control loop execution time comparison across three main stages

sections, several strategies are explored to reduce control loop delay and improve system responsiveness.

5.3 F29H85x Optimized Performance

To improve control performance, several optimization methods were tested on the F29H85x platform. The following results show how early interrupt, RTDMA, RTINT, TMU, and RAM mapping affect the delay in each stage of the control loop.

5.3.1 Early Mode Optimization

After enabling the early interrupt mode, the interrupt is triggered before the ADC conversion is fully completed. This method does not change the ISR execution flow but significantly reduces the delay between sampling completion and interrupt entry. Test results show that the ADC stage duration was reduced from 566 ns to 359 ns, and the overall control loop delay decreased from 1.27 µs to 1.09 µs, achieving an improvement of about 14%. This successfully decreases the waiting time between ADC conversion and interrupt response and the result is shown in Table 5.5.

In this table, when only the early mode is used, there is no risk of the ADC reading outdated data, because the time saved by skipping the conversion is much shorter than the time it takes to enter the ISR and read the ADC. However, when the interrupt delay is further reduced, this risk can still happen.

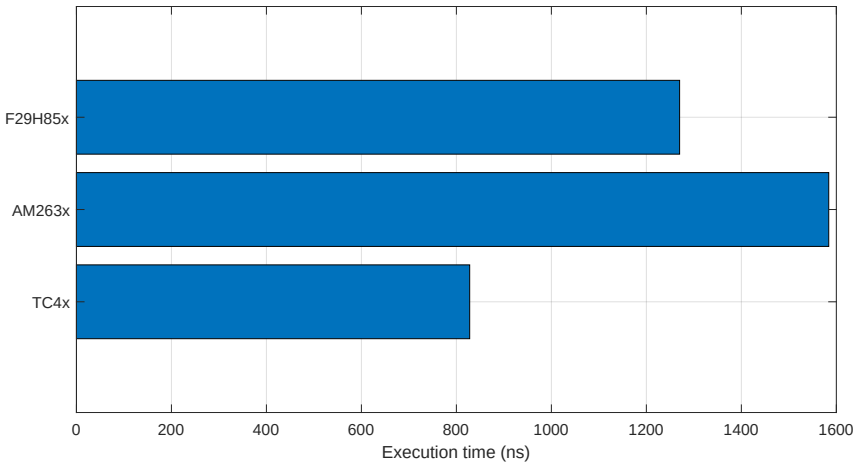

Figure 5.2: Baseline control loop execution time comparison in total

Table 5.5: Control loop implementation of F29H85x in early mode

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	359
	Conversion	(205)	
	Interrupt delay	279	
ISR	Result read	20	726
	PID control	470	
	PWM update	175	
	Clear flag	45	
	Path interaction	16	
Cross-stage path interaction		5	5
End-to-End total delay (ns)		1090	

5.3.2 RTINT Optimization

After enabling the RTINT, the system uses a dedicated interrupt entry and a shorter response path, allowing the interrupt controller to jump to the ISR more quickly after detecting an interrupt request. As shown in Table 5.6, this mechanism reduces the interrupt response delay from 280 ns to 155 ns, a reduction of about 45%.

The advantage of RTINT is that it optimizes the response path purely through hardware, without needing to modify the control logic. This makes it suitable for scenarios where faster interrupt handling is required but structural changes to the ISR are not desired. As a result, the total control loop delay is reduced from 1.27 µs to 1.155 µs, achieving an improvement of about 9%.

5.3.3 DMA Optimization

After introducing the RTDMA, the ADC result is automatically transferred after conversion and used to trigger the ISR. This means the CPU does not need to read

Table 5.6: Control loop implementation of F29H85x in RTINT

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	455
	Conversion	205	
	Interrupt delay	170	
ISR	Result read	30	715
	PID control	475	
	PWM update	155	
	Clear flag	45	
	Path interaction	10	
Cross-stage path interaction		-15	-15
End-to-End total delay (ns)		1155	

the ADC data manually, which reduces its workload. As a result, the ISR no longer needs time to read the ADC value.

Table 5.7: Control loop implementation of F29H85x with RTDMA

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	620
	Conversion	205	
DMA	Wait-for-trigger	15	
	Transfer and interrupt delay	320	
ISR	Result read	0	685
	PID control	495	
	PWM update	170	
	Clear flag	50	
	Path interaction	-58	
Cross-stage path interaction		-42	-42
End-to-End total delay (ns)		1347	

However, in Table 5.7, this also brings a clear trade-off: the DMA process adds extra time, 15 ns to wait for the trigger due to *tDMA* mechanism, 320 ns to transfer the data, and the interrupt delay. So, the delay between ADC conversion and ISR entry becomes longer, with a total of 335 ns for the DMA part. Overall, the total loop delay is 1347 ns, which is slightly higher than other optimization methods. But since the CPU no longer handles the data transfer, the ISR becomes simpler, the system's response is more stable, and more CPU time is available for other tasks. Such a small increase does not affect the practical value of using DMA in real applications, and we will discuss this further in the following sections.

5.3.4 TMU Optimization

After enabling the built-in TMU module on the F29H85x in Table 5.8, the compiler automatically replaces standard math library function calls with TMU instructions. This skips the slower software implementation and speeds up floating-point computations. Without TMU, the PID calculation is performed by the purely CPU core and takes 470 ns; with TMU enabled, it only takes 225 ns—nearly a 50% reduction. As a result, the total control loop delay is reduced from 1270 ns to 1050 ns, giving an overall improvement of 17% on execution time.

Table 5.8: Control loop implementation of F29H85x with TMU

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	575
	Conversion	205	
	Interrupt delay	290	
ISR	Result read	30	475
	PID control	225	
	PWM update	170	
	Clear flag	45	
	Path interaction	5	
Cross-stage path interaction		0	0
End-to-End total delay (ns)		1050	

5.3.5 Memory Optimization

When the control loop code is placed in RAM, since RAM is faster to access than Flash, this reduces the time spent in the ISR, especially the PID part, which drops from 470 ns to 375 ns. The total ISR time goes down from 726 ns to 625 ns. As a result, the full control loop delay is reduced from 1270 ns to 1177 ns, about 7% faster, which is summarized in Table 5.9.

5.4 AM263x Optimized Performance

On the AM263x platform, different improvements were applied, including early mode, EDMA, VIM interrupt controller, TCM memory, and compiler optimizations like Release and LTO. The results below show how each method changes the control loop delay and improves system performance.

5.4.1 Early Mode Optimization

AM263x also supports the early mode. As shown in Table 5.10, the total ADC stage time is reduced from 1044 ns to 884 ns, saving about 160 ns. Since the ISR is entered earlier, the total control loop delay drops from 1584 nµs to 1374 ns, giving an overall performance improvement of around 13%.

Table 5.9: Control loop implementation of F29H85x in RAM

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	555
	Conversion	205	
	Interrupt delay	270	
ISR	Result read	30	625
	PID control	375	
	PWM update	160	
	Clear flag	45	
	Path interaction	15	
Cross-stage path interaction		-3	-3
End-to-End total delay (ns)		1177	

Table 5.10: Control loop implementation of AM263x in early mode

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	884
	Conversion	(205)	
	Interrupt delay	764	
ISR	Result read	85	530
	PID control	185	
	PWM update	150	
	Clear flag	95	
	Path interaction	15	
Cross-stage path interaction		-40	-40
End-to-End total delay (ns)		1374	

5.4.2 Interrupt Optimization

For the VIM architecture in AM263x, through direct vector jumps, hardware arbitration, and bare-metal interrupt support, the interrupt controller can jump to the target ISR more quickly, avoiding extra overhead in the interrupt dispatch process. As shown in Table 5.11, the interrupt delay is significantly reduced to 318 ns (compared to 759 ns in the baseline, a reduction of 441 ns), meaning the interrupt response path is greatly accelerated. The total control loop delay decreases from 1584 ns to 1134 ns, resulting in an overall improvement of about 28%.

5.4.3 DMA Optimization

After enabling EDMA on AM263x, the ADC result is moved to memory through an EDMA chain of empty to actual to self-link, so the main core no longer needs to read the data manually, reducing CPU load. However, this method significantly increases the total control loop delay to 3456 ns. The main reason is the hardware

Table 5.11: Control loop implementation of AM263x with VIM

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	603
	Conversion	205	
	Interrupt delay	318	
ISR	Result read	85	530
	PID control	185	
	PWM update	150	
	Clear flag	95	
	Path interaction	15	
Cross-stage path interaction		-1	-1
End-to-End total delay (ns)		1134	

design limitation of EDMA, which requires extra design to make the control loop work. As a result, the actual DMA transfer takes a long time (2459 ns), and the cache invalidate must be called after entering the ISR (costing 310 ns) to make sure the data is correct. The ISR cannot start until the DMA transfer is fully finished.

Table 5.12: Control loop implementation of AM263x with EDMA

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	2744
	Conversion	205	
DMA	Empty transfer	2459	
	Actual transfer		
	Self-link		
ISR	Cache invadate	310	685
	Result read	0	
	PID control	245	
	PWM update	155	
	Path interaction	-25	
Cross-stage path interaction		27	27
End-to-End total delay (ns)		3456	

The result is shown in Table 5.12; therefore, this method is not recommended for us, especially for using DMA with ADC. These drawbacks mentioned above outweigh the benefits in time-sensitive control applications.

5.4.4 Memory Optimization

After enabling TCM, the ISR execution efficiency of the AM263x is significantly improved, especially in the PID control and PWM update stages, which is shown

in Table 5.13. By mapping critical code and data to the low-latency TCM, the processing speed is accelerated. As a result, the total ISR time is reduced from 535 ns to 480 ns, and the overall control loop delay decreases from 1584 ns to 1464 ns, achieving about 8% improvement.

Table 5.13: Control loop implementation of AM263x with TCM

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	983
	Conversion	205	
	Interrupt delay	698	
ISR	Result read	85	480
	PID control	185	
	PWM update	150	
	Clear flag	95	
	Path interaction	-35	
Cross-stage path interaction		1	1
End-to-End total delay (ns)		1464	

5.4.5 Compiler Optimization

After enabling release mode on the AM263x in Table 5.14, the ADC stage was reduced from 1044 ns to 793 ns, and the interrupt response time also dropped significantly from 759 ns to 508 ns. The total control loop delay decreased to 1309 ns, shortened by about 275 ns, resulting in a 17% improvement. This was mainly due to the release build optimizations, which improved memory scheduling and instruction paths during the sampling process.

Table 5.14: Control loop implementation of AM263x in release mode

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	793
	Conversion	205	
	Interrupt delay	508	
ISR	Result read	85	515
	PID control	190	
	PWM update	150	
	Clear flag	95	
	Path interaction	-5	
Cross-stage path interaction		-99	-99
End-to-End total delay (ns)		1209	

After enabling link time optimization (LTO) in Table 5.15, the total control loop delay on the AM263x platform decreased to 1529 ns, with a reduction of 55 ns,

bringing obvious benefits to this MCU. Most of the improvement happened in the ISR stage. The PID control time dropped from 195 ns to 160 ns, the interrupt delay also became slightly shorter, while the ADC module time stayed the same.

Table 5.15: Control loop implementation of AM263x with LTO

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	1024
	Conversion	205	
	Interrupt delay	739	
ISR	Result read	85	515
	PID control	160	
	PWM update	150	
	Clear flag	100	
	Path interaction	20	
Cross-stage path interaction		-10	-10
End-to-End total delay (ns)		1529	

5.5 AURIX TC4x Optimized Performance

For the TC4x platform, the TriCore setup uses different optimization ideas like offloading by implementing two types of subsystems, which are CDSP and PPU. The results compare the control loop delay in each case and show how the structure and speed are affected.

5.5.1 DMA Optimization

Regularly, by using the SDMA module to automatically transfer the ADC conversion result and trigger the interrupt, the system no longer needs to read the ADC data. This also improves the efficiency of the ISR, reducing its time from 380 ns (baseline) to 260 ns. However, due to the extra time needed to start and complete the DMA transfer, the DMA process takes 593 ns in total. As a result in Table 5.16, the total control loop delay slightly increases to 865 ns, which is 42 ns (about 5%) more than the baseline. In view of the similar behavior observed in the first two MCUs, this outcome is not unexpected. This same as the F29H85x; a small increase can be ignored in real applications.

5.5.2 CDSP Optimization

The local memory structure of CDSP (DCCM and ICCM) and its instruction set make it more efficient when handling floating-point control logic like PID. By offloading the PID task to the CDSP, the main core does not need to take part in the calculation. According to Table 5.17, the total delay of the CDSP setup is 750 ns, which is 78 ns (about 9.4%) less than the baseline. Since the CDSP also acts as

Table 5.16: Control loop implementation of TC4x with SDMA

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	598
	Conversion	N/A	
DMA	Transfer and interrupt delay	N/A	
ISR	Result read	0	260
	PID control	60	
	PWM update	150	
	Path interaction	50	
Cross-stage path interaction		7	7
End-to-End total delay (ns)		865	

the ISR trigger source, the 510 ns delay from ADC sampling to ISR entry includes ADC sampling, conversion, CDSP triggering, PID execution, and CDSP triggering the ISR after finishing. In contrast, the baseline's 453 ns only includes sampling, conversion, and triggering the ISR. The extra 48 ns shows the time used by CDSP to run the PID. So we can see that the PID calculation is clearly faster (from 60 ns to 48 ns), and the ISR also becomes simpler, with execution time reduced from 380 ns to 240 ns.

Table 5.17: Control loop implementation of TC4x with CDSP

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	510
	Conversion	N/A	
CDSP	PID control	48	
	Interrupt delay	N/A	
ISR	Result read	97	240
	PWM update	115	
	Path interaction	28	
Cross-stage path interaction		0	0
End-to-End total delay (ns)		750	

Offloading PID to CDSP improves both speed and structure. Total delay is reduced by 9.4%, and the ISR becomes lighter, making the system more efficient and parallel.

5.5.3 PPU Optimization

The PPU can run control tasks independently from the main core and works closely with peripherals like GTM and ADC. In this setup, because the PPU interrupt settings need support from the `speed_hal` library, we did not complete a full closed-loop where the ADC directly triggers an ISR inside the PPU. However, since we

mainly focus on measuring the timing of each part of the control loop, running all instructions in a loop inside the PPU by polling is also a valid way. The PPU takes over the control path, fully skips the main core, and directly handles peripheral interaction. The total control delay is 464.5 ns, which is reflected in Table 5.18.

Table 5.18: Control loop implementation of TC4x with PPU

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	N/A
	Conversion	N/A	
	Interrupt delay	N/A	
ISR (Dummy)	Result read	120	465
	PID control	220	
	PWM update	160	
	Path interaction	-35	
Cross-stage path interaction		N/A	N/A
End-to-End total delay (ns)		N/A	

5.6 Cross-Platform Evaluation

In this section, we apply the optimization strategies discussed earlier to all MCU platforms and carry out a comprehensive comparison. We first discuss the trade-offs of using DMA, as it may affect interrupt delay while offering potential benefits in CPU load reduction. Based on this analysis, we then compare the actual control loop execution times across platforms under their final optimized configurations. Finally, to eliminate the influence of different clock frequencies, we evaluate the control efficiency of each platform in terms of execution cycles and relative performance, providing a fair and in-depth cross-platform analysis.

5.6.1 Trade-off for DMA Optimization

While keeping compiler optimizations and code optimizations unchanged, the self-comparison results show that when DMA is enabled, the total execution time becomes longer for F29H85x, AM263x, and AURIX TC4x. This is mainly because the interrupt is triggered only after the DMA transfer is completed, which introduces extra delay in the ISR entry.

However, there are several methods to make the DMA benefits. One common method is to use DMA to transfer data in the background, while still letting the ADC module trigger the interrupt. This creates a form of parallel operation. Unfortunately, as shown in Table 5.7, the DMA transfer takes about 270 ns. If we compare this with Table 5.2, where the interrupt is triggered directly by the ADC and enters the ISR, the time is already very close, even without any optimization. Once optimizations are applied, it becomes very likely that in the ISR tries to read

the ADC value, the DMA has not yet moved the new result to the target address. Although we can still avoid it by setting an offset for the ADC, during code execution, it is difficult to know exactly how much offset to set. Also, this setting only works for the current situation and is not generally applicable. So it's not practical in our experiment setup.

Still, the DMA-based approach brings clear benefits. Since data is transferred without CPU involvement, it reduces CPU load and improves overall system efficiency. More importantly, in our setup, DMA is only used to move ADC results in a cycle. But in practice, it can be applied to many parts of the control path, such as updating PWM duty cycles, since the MCU supports multiple DMA channels. For the cross-platform comparison in the next sections, we will compare all MCUs with the final implementation without DMA to ensure that we show the most extreme performances and fairness. But we won't deny that the advantages of using DMA in system performance and scalability are significant; we will prefer to use the DMA-enabled version for the real application. This will be discussed in detail in the discussion section.

5.6.2 Execution Time Comparison for Final Optimization

In the final version of the F29H85x control loop, after applying all the effective optimizations, the total delay was reduced from the original 1270 ns to 655 ns in Table 5.19, a reduction of about 48%. The sampling and conversion time of the ADC module dropped to 225 ns, much lower than the original 560 ns, decreased to the early interrupt feature that allows earlier entry into the ISR. The interrupt delay was reduced from 280 ns to 145 ns with the help of the RTINT optimization, which offers a faster interrupt response path. Inside the ISR, the PID control time dropped to 150 ns, down from 470 ns, which is a 68% improvement. This was mainly achieved by using the TMU for hardware acceleration and running code in RAM, which reduces instruction fetch delay.

Table 5.19: Aggressive optimized implementation for F29H85x

Stage	**Operation**	**Delay (ns)**	**Stage Subtotal (ns)**
ADC module	S+H	80	225
	Conversion	(205)	
	Interrupt delay	145	
ISR	Result read	30	405
	PID control	150	
	PWM update	170	
	Clear flag	45	
	Path interaction	10	
Cross-stage path interaction		25	25
End-to-End total delay (ns)		655	

Although this extreme optimization setup minimized the total control loop execution time, we noticed that it introduced a risk of reading outdated data. Since we used early mode, the interrupt is triggered right after the S+H window ends. According to the data in Table 4.1, after the 75 ns sampling time, it takes 220 ns for the ADC result to be stored in the result register. However, after all optimizations were applied, the fast interrupt from RTINT allowed the system to enter the ISR in only 145 ns. This means the ISR could be triggered before the new ADC result was stored, leading to reading the value from the previous cycle. Luckily, as explained in Fig. 4.11, the early mode includes an offset feature. By adding a delay, we can ensure that when the ISR starts, the correct and current ADC value has already been stored in the result register. With this safer optimization method, an extra 75 ns delay was added, making the total control loop time 710 ns, which is shown in Fig. 5.20.

Table 5.20: Final optimized implementation for F29H85x

Stage	**Operation**	**Delay (ns)**	**Stage Subtotal (ns)**
ADC module	S+H	80	300
	Conversion	(205)	
	Interrupt delay	145(+75)	
ISR	Result read	30	405
	PID control	150	
	PWM update	170	
	Clear flag	45	
	Path interaction	10	
Cross-stage path interaction		5	5
End-to-End total delay (ns)		710	

In the final optimized version of the AM263x control loop, the total delay is reduced from the original 1584 ns to 793 ns in Table 5.21, a reduction of about 50%. The ADC sampling and conversion stage is optimized to 398 ns, also thanks to the early mode technique that allows the system to enter the ISR before the conversion is fully completed. The interrupt response delay is reduced from 759 ns to 318 ns, benefiting from the fast jump path enabled by the VIM. The ISR execution stage is also significantly accelerated—PID control time drops from 195 ns to 80 ns. These improvements are mainly due to TCM acceleration and enhanced execution efficiency brought by LTO compiler optimization.

For the AURIX TC4x platform, a standard optimization flow is still under development. We also introduced the CDSP as a new optimization method. Although the PPU was only used to run a CPU polling control loop, it still helps to reflect the time spent inside the ISR. The maximum execution time of these setups has already been discussed in section 5.4, so it will not be repeated here.

Among the five platforms (with CDSP and PPU on TC4x counted as separate

Table 5.21: Final optimized implementation for AM263x

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80	398
	Conversion	(205)	
	Interrupt delay	318	
ISR	Result read	60	395
	PID control	80	
	PWM update	145	
	Clear flag	100	
	Path interaction	10	
Cross-stage path interaction		0	0
End-to-End total delay (ns)		793	

ones), the delay from signal acquisition to entering the ISR shows clear differences. F29H85x has the shortest time at 220 ns with an extra 75 ns for data safety, followed by AM263x at 318 ns. On AURIX TC4x for both cores, since Infineon does not show the ADC conversion time clearly in the public source, the interrupt delay cannot be calculated. But for the time between ADC sampling starts to enter the ISR is clear, F29H85x with 295 ns, AM263x with 398 ns, for the TriCore takes 453 ns, and the CDSP mode takes 505 ns, because it needs to finish processing before triggering the interrupt. The PPU mode does not use interrupt triggering, so this part has no data.

Inside the ISR, the read time ranges from 30 ns on F29H85x to 120 ns on PPU. PID control time is 150 ns for F29H85x, 80 ns for AM263x, 60 ns for TriCore, 48 ns for CDSP, and 270 ns for PPU. PWM update time is between 100 ns and 170 ns across all platforms. For total control loop time, F29H85x takes 705 ns, AM263x takes 793 ns, CDSP takes 746 ns, TriCore takes 823 ns, and PPU takes 464.5 ns (excluding ADC and interrupt parts). Also, only the TI platforms list the time for clearing interrupt flags: 45 ns on F29H85x and 100 ns on AM263x. TC4x platforms do not require this step due to their architecture.

All the results can be found in Table 5.20, 5.21 and 5.4, and can be summarized in Fig. 5.3. In this figure, as before, we divide the entire closed-loop system into three parts for discussion for different CPUs, which have the same workflow.

However, due to the TC4x's CDSP and PPU — one using a different hardware path and the other not building a complete interrupt-based loop — we only compare the execution time of the ADC module, PID algorithm, and register access when running on the CPU for all three MCUs. It can be seen that in the ADC module, the two TI MCUs have more room for optimization, so their performance is slightly better than the AURIX TC4x. In algorithm execution, both the AM263x and AURIX TC4x show strong performance, clearly better than the F29H85x, even when the total execution time is not the best. For register access, as discussed earlier, the

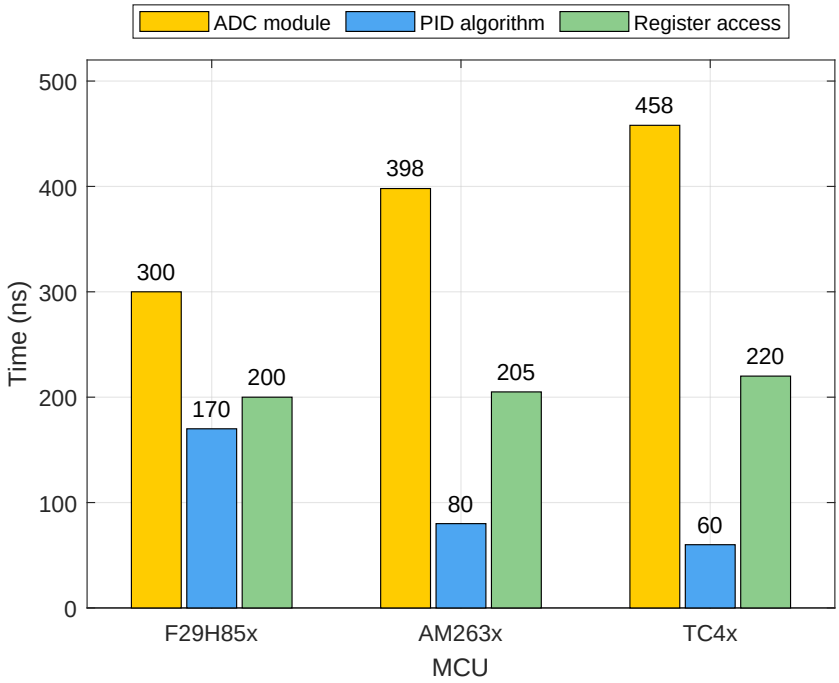

Figure 5.3: Final optimized control loop execution time comparison across three main stages

use of DMA can remove this overhead in a suitable design. Both the F29H85x and AURIX TC4x have safe and efficient DMA mechanisms, while the AM263x may require a more complex design.

For the total execution time based on Fig. 5.4, the F29H85x requires the least time, followed by the AURIX TC4x with CDSP, then the AM263x with a similar result. The TC4x's CPU performs slightly slowly, and its PPU did not show complete and strong performance under the conditions of our experiments.

5.6.3 Efficiency Evaluation for Final Optimization

The above tables show the time distribution of each stage in the control loop in detail, it does not reflect the cycle efficiency of each platform under different clock frequencies, nor does it reveal the overall control cost caused by architectural differences. Therefore, Table 5.22 adds this perspective by introducing three key metrics: execution cycles, performance ratio, and effective MHz per core (eMHz/Core).

The execution cycles represent the number of clock cycles required for a platform to complete one full control loop at its current frequency. This reflects how efficiently the internal architecture handles timing delays.

$$\text{Execution Cycles} = \frac{\text{Control Loop Time (ns)}}{1000/\text{Frequency (MHz)}}$$

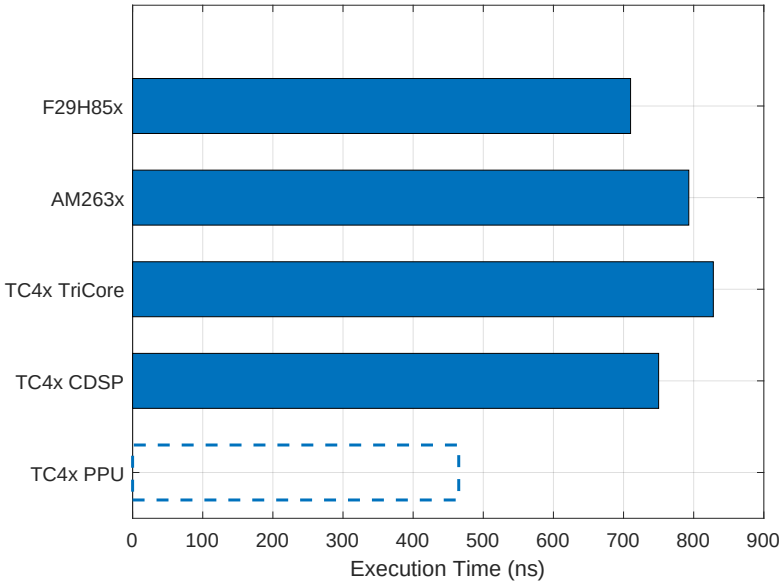

Figure 5.4: Final optimized control loop execution time comparison in total

The performance ratio is used to compare how long different platforms take to complete the same control task. F29H85x is used as the baseline (141 cycles), and the ratio shows how efficient other platforms are for the same task.

$$\text{Performance Ratio} = \frac{\text{Reference Excution Cycle (F29H85x)}}{\text{Target Excution Cycle}}$$

The effective MHz per Cycle (eMHz/Core) removes the direct impact of frequency and helps evaluate how much frequency is needed for each unit of performance. This metric is useful to compare platforms with different clock speeds and check for architectural overhead or unused potential.

$$\text{eMHz/Core} = \text{CPU Frequency (MHz)} \cdot \text{Performance Ratio}$$

Some key insights can also be drawn from the table. In terms of platform efficiency, the F29H85x, although running at only 200 MHz, completes a full control loop in just 141 cycles, making it the most efficient among all platforms. It is therefore set as the reference (Performance Ratio = 1). This shows that its architecture design, RTINT interrupt mechanism, and VLIW parallel execution offer strong advantages in control tasks. In contrast, AM263x and TC4x, even with higher clock speeds of 400 MHz and 500/160 MHz respectively, still require more cycles to finish the same task. The PPU uses a polling structure and lacks a complete sampling path, so it is not included in the full-cycle comparison.

From the perspective of control efficiency, clock frequency is not the key factor in performance. Even though AM263x and TC4x platforms run at higher frequencies, they require more cycles to complete the same control task compared to F29H85x.

Table 5.22: Efficiency result of final optimized MCUs

MCU	CPU	CPU type	CPU frequency	Optimization methods	Cycles	Performance Ratio	eMHz/Core
F29H85x	C29	9-stage pipeline VLIW (up to 8 instructions)	200 MHz	Early mode, RTINT, TMU, (DMA)	141	1	200
AM263x	Cortex-R5F	8-stage pipeline, limited dual-issue, branch prediction	400 MHz	Early mode, VIM INT, TMU, TCM	317	0.44	176
TC4x TriCore	TriCore 1.8	3 × 4-stage superscalar pipeline, branch prediction	500 MHz	(DMA)	432	0.32	160
TC4x CDSP	TriCore 1.8/ ARC EM5D DSP core	-	500 MHz/ 160 MHz	CDSP	373	0.37	185

This shows that architectural design matters more than just frequency.

To sum up, we have presented the performance and efficiency results for each platform across the full control loop. While two metrics were evaluated, our main focus is on execution speed and how it can be further improved. Therefore, in the next section, we will analyze the key execution paths in each platform to uncover the architectural reasons behind these speed differences and explore suitable optimization strategies for each MCU.

6

Discussion

This chapter aims to explain the structural reasons behind the performance results. We are not only looking at the execution time itself, but also trying to understand how each step in the control flow is affected by the system architecture and peripheral design. This helps to show the real differences in how each platform performs in practice. In addition, to show the best possible performance of each platform, our tests used some extreme settings in earlier chapters. These settings helped highlight differences in architecture, but they may not match common real-application use cases. Instead, such extreme settings might overlook or hide configurations that are more practical or better suited for real applications. So, in this chapter, we will first look at the structural causes of timing differences and then discuss whether these extreme settings are reasonable or not.

6.1 Results Analysis

The results section only shows the measured times and execution cycles for each MCU under different test conditions. Here, we will look deeper into the reasons behind these results and try to understand how the internal structure of each MCU affects its performance and efficiency by tracing it back to architectural mechanisms. We will follow the full process, starting from ADC sampling and conversion, then how the interrupt is triggered, and finally what happens inside the ISR. By looking at each step, we can explain why each MCU takes more or less time, and how different design choices lead to these differences.

6.1.1 Experimental Setup

In fact, the experimental setup was deliberately conservative and strict, with the main goal of ensuring real-time data accuracy and safety, especially in how the interrupt sources were configured. Unlike typical industrial applications that use continuous sampling and timer-based interrupts, our system was designed to sample at a fixed rate and trigger interrupts only after the previous control task was completed. This setup ensures that the data read is always up to date.

For example, on the F29H85x platform, we added an extra offset delay to slow down the interrupt response, avoiding the risk of reading old data before the ADC result was written. Similarly, DMA was not included in the final optimized setup, because

in such a short control cycle, it could introduce extra delay. Overall, instead of simulating practically relevant trade-offs between speed and efficiency, we aimed to push each MCU to its performance limit under strict safety conditions. This allowed us to identify the most effective architectural optimizations for improving execution speed while maintaining data integrity.

6.1.2 ADC Delay and Interrupt Path

To fairly compare the ADC behavior across platforms, all MCUs were configured with a sampling time of 80 ns. Although their hardware capabilities differ, this setting helps focus the comparison on structural differences in conversion efficiency, interrupt triggering, and ISR response performance.

Sampling and Conversion window

The minimum sampling windows for the F29H85x and AM263x are 75 ns and 80 ns, respectively, showing similar analog sampling capabilities. The TMADC in TC4x, however, supports a minimum sampling window as short as 12.5 ns, demonstrating higher analog bandwidth and faster sampling potential. While we set its sampling time to 80 ns in our experiment to keep the conditions consistent (balancing precision and eliminating extra variables), the underlying performance advantage remains evident for high-frequency control applications.

In real embedded control systems, ADCs usually trigger interrupts after sampling and conversion are fully completed to ensure data accuracy. However, in our experiment, to evaluate the interrupt response capabilities under extreme conditions, both the F29H85x and AM263x enabled the early interrupt mode. Although this mode may bring data risks in real applications, it removes the delay from conversion time, making it easier to maximize the performance of the interrupt architecture. With early mode, up to 205 ns of waiting time can be saved, which is why these two platforms show the shortest response paths in the final result graph.

Different Interrupt Path

In our project, since the exact conversion time of the AURIX TC4x is not available, we assume that under the same sampling time, the conversion times of all platforms are roughly similar. Based on this assumption, the differences in the ADC module can approximately reflect the differences in interrupt delay.

The results show that the F29H85x, with its PIPE hardware arbitration and dedicated RTINT stack, not only supports automatic context saving and interrupt nesting but also uses a fixed vector jump mechanism, creating the shortest and most efficient interrupt path. In contrast, the AM263x interrupt process needs to pass through an IR and MUX, where multiple arbitration steps make the path longer and add delay. The TC4x uses a centralized IR and SRN, handled by the interrupt service provider (ISP) and its interrupt control unit (ICU). Each ISP has its own

control unit, which supports flexible priority and parallel interrupt management, but in simple closed-loop tasks, this general-purpose design may not bring the lowest delay. Further analysis shows that more routing and arbitration layers result in a longer interrupt path; if context saving uses a dedicated hardware stack, the response is faster; and if the vector jump is fixed and direct, it avoids extra overhead. These structural differences directly cause the difference in interrupt delay among the three MCUs. It should be noted that the actual conversion time of the TC4x is unknown. If it is longer than TI's 205 ns, then under our experiment setup, the TC4x could perform better; if shorter, it may indicate that its interrupt path delay is relatively larger.

Therefore, the clear conclusion is that the F29H85x has a more efficient interrupt path than the AM263x, while the TC4x, although lacking exact data, still shows a well-designed interrupt mechanism. Overall, this method of analyzing structural differences based on measured results cannot cover all delay factors, but it helps reveal the key impact of interrupt structures on real-time performance.

6.1.3 Peripheral Access and Floating-point Calculation

The ISR includes several key steps, among them the read and update operations, which both involve accessing peripheral modules. So, their efficiency depends not only on instruction execution ability, but also on the bus structure, how peripherals are connected, and how well the access path is optimized. The following analysis will explore these aspects to understand how each platform handles ISR execution at the hardware level.

System Interconnection

Instructions in ISR, like reading ADC results or updating PWM registers, rely heavily on the internal peripheral interconnect of the MCU. In the F29H85x, all control-related peripherals are connected to a shared bus called VBUS32. When the CPU accesses ADC or PWM in the ISR, it must go through this common bus. While this design helps with centralized peripheral management, it may cause access delays due to bus contention and arbitration. The AM263x uses a multi-layer interconnect structure based on the common bus architecture (CBA). Peripherals like ADC and PWM are connected through several VBUSP layers. Although TCM allows fast single-cycle access to local variables and instructions, it does not change the fact that peripheral access must still go through the system interconnect. In the TC4x, all peripherals are accessed through a unified memory-mapped address space. The system also includes a special LLI, which is designed to shorten the data path between the CPU and key peripherals.

Overall, all three MCUs show good performance in peripheral access and can meet the real-time needs of control loops. However, it is worth noting that the TC4x still has room for further system-level optimization. As a result, its special LLI already brings noticeable benefits at this stage. In comparison, the interconnect structures'

performances of the F29H85x and AM263x are more mature and have been well optimized, meaning their performance is closer to the upper limit, while TC4x still has potential for improvement.

Instruction Setting and Calculation

In PID computation, each platform shows its architectural strengths that directly affect instruction processing and calculation speed:

- **F29H85x:** This uses the C29x CPU, which can run multiple 64-bit operations per cycle and has a 128-bit instruction bus for high parallel throughput. Its built-in FPU and TMU improve the speed of floating-point and nonlinear operations, making it suitable for heavy control calculations.
- **AM263x:** This is based on the Arm Cortex-R5F core, with a single-precision FPU (VFPv3) and a dual-issue pipeline. This shortens the instruction path and speeds up execution. The FPU handles multiplication and integration steps used in PID control efficiently.
- **AURIX TC4x:** This uses the TriCore 1.8 core with a double-precision FPU. It runs arithmetic operations like addition, multiplication, and division efficiently in control algorithms. The short pipeline of TriCore helps fast response in real-time loops. The CDSP module maps the control algorithm and data to local high-speed memory, and with its own FPU and independent execution path, it achieves much higher speed than the main core for our custom PID algorithm.

From the results, these design differences explain part of the performance gap between platforms. In our tests, the CDSP of TC4x reached the best execution speed, while the TriCore main core and AM263x showed similar performance, and both were faster than the F29H85x in tasks with more floating-point operations. However, since our tests used small data sets and single-point calculations, the full potential of parallelism and vectorization was not fully reflected. The reason why the PPU did not reach high performance will be explained in section 6.2.1.

It is important to note that although we measured execution time for each stage on different platforms, such comparisons are not scientifically rigorous, just like evaluating a sports car's performance based on a 10-meter sprint. We will provide a more comprehensive discussion of this issue in the following subsection.

System-Level Execution Path Analysis

When evaluating the performance of a control loop, measuring the time of each step, such as ADC reading, PID calculation, and PWM update, separately may not truly reflect the actual response delay of the system. This kind of step-by-step analysis overlooks the overlapping and coupling that exist in real control paths.

First, toggling a GPIO is often used to measure the delay of an operation, but it can also add extra overhead. Then, different platforms have different pipeline depths

and scheduling mechanisms. Some computations may run in parallel at the hardware level, such as the dual-issue pipeline of the AM263x or the wide-bus parallel data paths of the F29H85x. This means instruction execution is not always done in a simple, linear order. Second, data dependencies between instructions and cache behavior can change the execution path dynamically, which affects the timing. Also, the delay of peripherals like ADC and PWM is not always fixed; it can vary with system load or interconnect arbitration. In addition, the triggering and execution of an ISR do not only depend on the CPU but are also affected by things like the interrupt vector setup, compiler optimization, and stack configuration.

Therefore, to truly understand and compare the performance of different platforms, we need to look at the full control path and consider hardware structure, system load, and software context together. Although step-by-step measurements are still useful for identifying bottlenecks or tuning specific functions, focusing on the overall execution time gives a more meaningful and realistic view of system performance.

6.2 Trade-offs of Architectural Optimizations

To improve the efficiency and response of the control system, we tried to use some hardware accelerators and automatic data transfer mechanisms. In theory, these technologies can reduce the load on the main core and speed up data processing. However, our experiments showed that although these methods have architectural advantages, they may actually introduce extra delay and overhead under our test conditions. In the following sections, we will explain the reasons for the delays based on the working principles and evaluate whether these methods are worth using.

6.2.1 Evaluation and Potential of PPU

On the AURIX TC4x platform, the PPU is designed as a high-performance co-processor to reduce the workload of the main core. It uses a SIMD architecture, which allows it to perform the same operation on multiple data items in a single instruction cycle. This greatly improves the efficiency of batch data processing. The PPU has an independent execution path and built-in hardware acceleration. It can bypass the main core, directly access main memory, read input data, perform calculations, and write back the results. This design is especially suitable for tasks with strong periodicity and large data volumes, such as image pre-processing, filtering, feature extraction, and AI inference [48]. In these scenarios, the PPU can handle complex and intensive computations without interrupting the real-time work of the main core, improving system-level parallelism and responsiveness.

Although the PPU has strong floating-point computing power, it did not show ideal performance in our real-time end-to-end control loop tests. One reason is that the PPU and TriCore core must cooperate through shared memory and interrupt mechanisms. This adds data transfer and task scheduling overhead. In short, frequent and small control tasks may not benefit from the PPU's acceleration, as the over-

head may be larger than the gain. In addition, our test only processes single data points at a time, with very little data per loop. Such tasks have low parallelism and cannot make full use of the PPU's vector features. Also, the PPU has longer communication paths to peripherals compared to the main CPU, which uses LLI. The PPU needs to go through more steps to access peripherals, causing longer response times. In short-period control tasks, this makes the PPU slower than the main core, which can directly read or write control registers. These results do not mean that the PPU is weak—it simply means that its strengths do not match tasks that need very fast responses and very fine-grained control, like our tests.

Still, the PPU has strong potential in control systems. Its biggest advantage is true hardware-level parallelism. Unlike traditional MCUs that rely on interrupts or task switching to simulate parallelism, the PPU has its own execution unit and instruction flow. It can run complex algorithms while the main core handles real-time tasks. Also, since it can directly access main memory, the PPU can handle data from peripherals. With DMA or TMADC, the PPU could do signal preprocessing, frequency domain analysis, and more. This would reduce the main core's load and shorten the full control loop. In the future, if control tasks become more complex—such as MIMO control or multiple filter paths—the PPU's parallel computing ability may show more value.

6.2.2 Evaluation and Potential of DMA

When using DMA to optimize the control loop, we observed that although it avoids manually reading data in the ISR, using DMA completion as the interrupt trigger delays the ISR response, since the system must wait for the transfer to complete. This approach simplifies the ISR logic but sacrifices some real-time performance. For example, while the delay introduced by DMA was minor on F29H85x and AURIX TC4x, the AM263x case clearly showed increased total control loop delay. In contrast, early interrupt modes (if supported) can offer faster response by triggering the ISR before the DMA or even ADC conversion completes. However, this carries the risk of reading outdated or invalid data. Therefore, while not the fastest, DMA often provides better data integrity, making it more reliable under certain conditions.

Moreover, the speed limitation of DMA was only observed under our highly conservative experimental setup. In the following section, we will show that even under such conservative conditions, DMA still offers notable advantages, along with a common optimization approach that ensures data safety.

DMA Performance in High-Throughput Scenarios

The conclusion that DMA often results in slower performance does not apply to all situations. As the number of sampling channels and samples per channel increases, the advantage of DMA becomes clearer. In our test case with two ADC channels, each performing six consecutive conversions, we compared two implementations: with and without DMA. The results show that although DMA slightly increases the

ADC stage time (due to transfer operations), it greatly reduces the ISR execution time by removing the manual read step.

Table 6.1: 2-channel 6-SOCs closed-loop system

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80*6	1842
	Conversion	205*6	
	Interrupt delay	212	
ISR	Result read	100	535
	PID control	225	
	PWM update	150	
	Path interaction	60	
Interrupt delay		125	125
ISR	Result read	85	495
	PID control	225	
	PWM update	150	
	Path interaction	50	
Cross-stage path interaction		3	3
End-to-End total delay (ns)		3000	

As a result in Table 6.1 and 6.2, respectively, present the timing composition of the two-channel closed-loop system without DMA and the same system with DMA. The total control loop delay becomes even shorter by using DMA. The measured data shows that under this multi-channel continuous sampling condition, the version with DMA had a delay of 2.95 µs, while the version without DMA had a delay of 3 µs. This indicates that in high-throughput applications, the structural efficiency of DMA gradually outweighs its initial interrupt delay, leading to better system responsiveness. The value of DMA is not only in saving CPU load or simplifying logic, but also in its structural performance advantage for tasks with many channels or frequent sampling.

The Ping-Pong Buffers

In real applications, this kind of multi-channel continuous sampling is more common than the one-sample-at-a-time approach we used for testing different platform performances. To improve efficiency, real systems often use continuous sampling and continuous DMA transfer. In such cases, the ISR typically reads results directly from shared memory without waiting for the DMA to complete. If the control algorithm takes too long, the DMA may overwrite old data before the system finishes processing it.

To avoid this problem, a ping-pong buffer is very important. This double-buffer mechanism ensures that each set of sampled data is fully processed, preventing data loss or mismatch. We introduced a ping-pong buffer mechanism on the F29H85x

Table 6.2: 2-channel 6-SOCs closed-loop system with DMA

Stage	Operation	Delay (ns)	Stage Subtotal (ns)
ADC module	S+H	80*6	1960
	Conversion	205*6	
DMA	Wait-for-trigger	15	
	Transfer and interrupt delay	235	
ISR	Result read	0	435
	PID control	230	
	PWM update	175	
	Path interaction	55	
Interrupt delay		115	115
ISR	Result read	0	430
	PID control	230	
	PWM update	150	
	Path interaction	50	
Cross-stage path interaction		10	10
End-to-End total delay (ns)		2950	

platform in Fig. 6.1. This structure uses two alternating buffers (Buffer0 and Buffer1). After each ADC sampling, the DMA writes the result into the currently active buffer (e.g., Buffer0), then enters the ISR. Inside the ISR, the system processes the data in Buffer0 and switches the DMA target address to Buffer1 for the next round. After finishing calculations and updates, the ISR exits. When ePWM triggers the ADC again, the DMA writes into Buffer1 (because it was switched in the last ISR), and ISR will process Buffer1's data, then switch back to Buffer0. This way, sampling and processing can happen in parallel, improving system throughput and ensuring data integrity and stability.

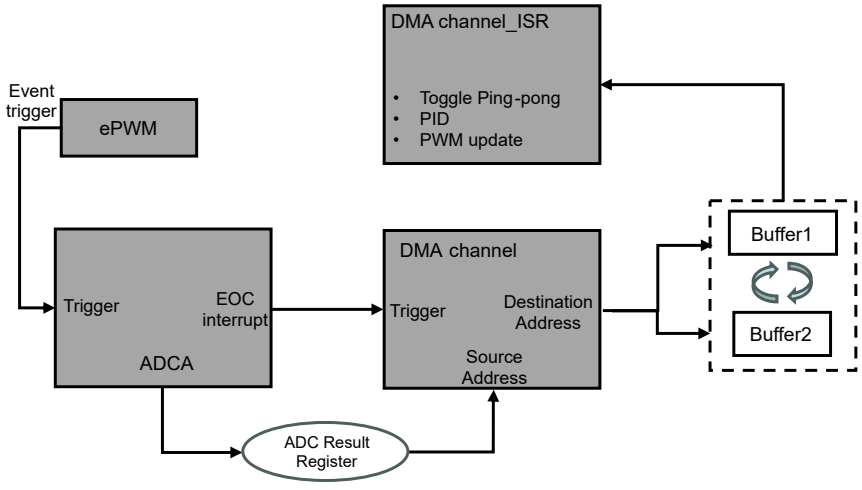

Figure 6.1: Control loop design with ping-pong buffers

Table 6.3 summarizes the timing result of the F29H85x control loop when using RTDMA with a ping-pong buffer. It shows how DMA transfer, buffer switching, and ISR execution contribute to an overall end-to-end delay. After introducing RTDMA and the ping-pong Buffer, the total control loop delay on the F29H85x increased from 1.27 µs (baseline) to 1.913 µs, about a 50% rise. The delay in the ADC stage grew from 560 ns to 630 ns, mainly due to the added DMA trigger wait and transfer time (together around 350 ns). In the ISR stage, the total time increased from 726 ns to 1.277 µs, mostly because of the 600 ns buffer switching. Meanwhile, the manual read step was fully removed, and the PID control and PWM update times stayed nearly the same. Overall, although the CPU workload was reduced, the added DMA logic and buffer handling caused a longer total delay.

Table 6.3: Control loop design of F29H85x with RTDMA in ping-pong buffer

Stage	**Operation**	**Delay (ns)**	**Stage Subtotal (ns)**
ADC module	S+H	80	630
	Conversion	205	
DMA	Wait-for-trigger	15	
	Transfer and interrupt delay	330	
ISR	Buffer switch	600	1277
	Result read	0	
	PID control	460	
	PWM update	175	
	Clear flag	40	
	Path interaction	-25	
Cross-stage path interaction		6	6
End-to-End total delay (ns)		1913	

From a result view, this method trades some real-time performance for better data integrity and system robustness. The ping-pong buffer helps avoid overwriting unprocessed data when new ADC values are written by DMA, which is important in continuous sampling or complex control systems. Compared to traditional fast-response designs, using DMA offers a more stable balance between speed and safety, showing a typical trade-off between real-time response and reliability in control systems.

7

Conclusion

This study compared three automotive-grade MCUs—F29H85x, AM263x, and AURIX TC4x—under a unified closed-loop control system. The results show that while all three platforms can handle basic control tasks, their performance and optimization potential differ because of architectural differences. The F29H85x achieves the fastest overall execution time thanks to its efficient interrupt path and bus design. The AM263x shows strength in floating-point operations, but its more complex interrupt and peripheral access paths add extra delay. The AURIX TC4x already shows good baseline performance, and its hardware accelerators (CDSP and PPU) offer greater potential for further optimization.

The contribution of this study goes beyond performance comparison. It introduces a cross-platform analysis method: by breaking the control loop into sampling, computation, and output stages, and linking them with interrupt structures, CPU architectures, peripheral designs, and compiler optimizations, we can clearly see how different MCU designs affect real-time performance. This approach goes further than single-platform studies and provides a more general way to understand the relation between hardware features and task requirements. In addition, the study shows that hardware advantages depend strongly on the type of task. In short-cycle, fine-grained control tasks, some accelerators with theoretical benefits may add overhead instead. But in more complex cases, such as multi-loop control, signal pre-processing, or AI-assisted control, these accelerators may bring much more value. This means optimization is not "one-size-fits-all" but must be chosen based on control period, data size, and task complexity.

In conclusion, this work not only provides practical guidance for MCU selection in industrial applications, but also introduces a way to study MCU performance by considering both the needs of the task and the hardware design. The evaluation method and data presented here form a foundation for future research on more complex embedded control tasks, and give useful guidance for building low-latency, high-efficiency control systems in automotive and industrial fields.

Bibliography

[1] B. K. Bose, "Power electronics and motor drives recent progress and perspective," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 2, pp. 581–588, 2009.

[2] N. Mohan, T. M. Undeland, and W. P. Robbins, "Power electronics, converters, applications and design," *Microelectronics Journal*, vol. 28, no. 1, 1995.

[3] J. Millán, P. Godignon, X. Perpiñà, A. Pérez-Tomás, and J. Rebollo, "A survey of wide bandgap power semiconductor devices," *IEEE Transactions on Power Electronics*, vol. 29, no. 5, pp. 2155–2163, 2014.

[4] H.-P. Li and Y.-w. Li, "The research of electric vehicle's MCU system based on ISO26262," in *2017 2nd Asia-Pacific Conference on Intelligent Robot Systems (ACIRS)*, 2017, pp. 336–340.

[5] M. Rossi, N. Toscani, M. Mauri, and F. C. Dezza, *Introduction to Microcontroller Programming for Power Electronics Control Applications: Coding with MATLAB® and Simulink®*, 1st ed. Boca Raton: CRC Press, 2021.

[6] A. Boglietti, A. Cavagnino, D. Staton, M. Shanel, M. Mueller, and C. Mejuto, "Evolution and modern approaches for thermal analysis of electrical machines," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 3, pp. 871–882, 2009.

[7] International Organization for Standardization (ISO), *ISO 26262: Road Vehicles – Functional Safety*, Std., 2018, iSO 26262:2018, International Organization for Standardization, Geneva, Switzerland.

[8] M. A. Rahman, A. A. Abushaiba, and A. M. Elrajoubi, "Integration of C2000 microcontrollers with MATLAB Simulink embedded coder: A real-time control application," in *2024 7th International Conference on Electrical Engineering and Green Energy (CEEGE)*, 2024, pp. 131–136.

[9] M. Barr and A. Massa, *Programming Embedded Systems: With C and GNU Development Tools*, 2nd ed. O'Reilly Media, 2006.

[10] L. Zhang, H. Ma, R. Born, X. Zhao, and J.-S. Lai, "Single-step current control for voltage source inverters with fast transient response and high convergence speed," *IEEE Transactions on Power Electronics*, vol. 32, no. 11, pp. 8823–8832, 2017.

[11] E. Stolyarov, A. Anuchin, M. Lashkevich, D. Aliamkin, S. Grishin, and A. Zharkov, "Using a control law accelerator for current loop performance enhancement," in *2020 27th International Workshop on Electric Drives: MPEI Department of Electric Drives 90th Anniversary (IWED)*, 2020, pp. 1–4.

[12] C. Fei, Q. Li, and F. C. Lee, "Digital implementation of adaptive synchronous rectifier (SR) driving scheme for high-frequency LLC converters with microcontroller," *IEEE Transactions on Power Electronics*, vol. 33, no. 6, pp. 5351–5361, 2018.

[13] K. Tytelmaier, O. Husev, O. Veligorskyi, and R. Yershov, "A review of non-isolated bidirectional DC-DC converters for energy storage systems," in *2016 II International Young Scientists Forum on Applied Physics and Engineering (YSF)*, 2016, pp. 22–28.

[14] L. Schmitz, D. C. Martins, and R. F. Coelho, "Comprehensive conception of high step-up DC–DC converters with coupled inductor and voltage multipliers techniques," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 6, pp. 2140–2151, 2020.

[15] M. Gildersleeve, H. Forghani-zadeh, and G. Rincon-Mora, "A comprehensive power analysis and a highly efficient, mode-hopping DC-DC converter," in *Proceedings. IEEE Asia-Pacific Conference on ASIC,*, 2002, pp. 153–156.

[16] M. Forouzesh, Y. P. Siwakoti, S. A. Gorji, F. Blaabjerg, and B. Lehman, "Step-up DC–DC converters: A comprehensive review of voltage-boosting techniques, topologies, and applications," *IEEE Transactions on Power Electronics*, vol. 32, no. 12, pp. 9143–9178, 2017.

[17] A. Chakraborty, A. Khaligh, A. Emadi, and A. Pfaelzer, "Digital combination of buck and boost converters to control a positive buck-boost converter," in *2006 37th IEEE Power Electronics Specialists Conference*, 2006, pp. 1–6.

[18] Z. Zhang, Y. Huang, Y. Wu, and R. Wang, "Review of bidirectional DC-DC converter topologies for hybrid energy storage systems," *Energy Storage and Saving*, vol. 2, p. 100010, 2022.

[19] S. Arora and M. Singh, "Reduction of switching transients in CC/CV mode of electric vehicles battery charging," in *5th IET International Conference on Clean Energy and Technology (CEAT2018)*, 2018, pp. 1–6.

[20] S. Basak, "Implementation of real-time digital control for phase shifted full-bridge rectifier using texas instruments' ARM Cortex-R5F core based MCU," in *2024 IEEE International Communications Energy Conference (INTELEC)*, 2024, pp. 1–5.

[21] U. Nayak, J. Chakraborty, and A. K. Pati, "A critical review on different DC-DC converter and charging methods for EV application," in *2024 10th International Conference on Electrical Energy Systems (ICEES)*, 2024, pp. 1–6.

[22] D. He and R. Nelms, "Current-mode control of a DC-DC converter using a microcontroller: implementation issues," in *The 4th International Power Electronics and Motion Control Conference, 2004. IPEMC 2004.*, vol. 2, 2004, pp. 538–543 Vol.2.

[23] P. Singh, A. Singh, and A. Arora, "A comprehensive study on the closed loop performance of a zeta converter," in *2023 11th National Power Electronics Conference (NPEC)*, 2023, pp. 1–6.

[24] S. S. K. Kenny, R. Naayagi, S. S. Lee, and C. Shuyu, "Three phase VSI control system rapid prototyping with TI C2000 MCU and PLECS coder," in *2021 5th International Conference on Green Energy and Applications (ICGEA)*, 2021, pp. 42–46.

[25] K. Kamalesh, P. S. P. Reddy, and J. S. Vinodhini, "Open loop and closed loop comparision for single phase cyclo converter (PID controller)," in *2016 Second International Conference on Science Technology Engineering and Management (ICONSTEM)*, 2016, pp. 481–484.

[26] M. N. Alam, N. Bashar, S. Sarker, S. A. Lopa, and T. Ahmed, "Comparative study of the open-loop boost converter and the closed-loop PID controlled boost converters," in *2023 International Conference on Electrical, Computer and Communication Engineering (ECCE)*, 2023, pp. 1–6.

[27] K. J. Åström and T. Hägglund, *Advanced PID Control*. ISA - The Instrumentation, Systems and Automation Society, 2006.

[28] D. Rivera, "PID control: New identification and design methods - [book review]," *IEEE Control Systems Magazine*, vol. 26, no. 1, pp. 95–97, 2006.

[29] C.-C. Liu, S.-J. Chang, G.-Y. Huang, and Y.-Z. Lin, "A 10-bit 50-ms/s SAR ADC with a monotonic capacitor switching procedure," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 4, pp. 731–740, 2010.

[30] Analog Devices, Inc., "Chapter 20: Data Converters," https://wiki.analog.com/university/courses/electronics/text/chapter-20, n.d., accessed: 2025-07-18.

[31] Texas Instruments, *F29H85x and F29P58x Real-Time Microcontrollers Technical Reference Manual*, Texas Instruments, Nov. 2024, document No. SPRUJ79, Accessed: 2025-07-18. [Online]. Available: https://www.ti.com.cn/cn/lit/ug/spruj79/spruj79.pdf

[32] Infineon Technologies AG, *AURIX™ TC4Dx Microcontroller User's Manual*, 2024, available at https://www.infineon.com/assets/row/public/documents/10/44/infineon-infineon-aurix-tc4dx-um-v1.1-public-en-us.pdf.

[33] S. Vaishnav, C. Murarishetty, and G. Jayakrishna, "Timer validation of automotive microcontroller using motor application," in *2017 2nd IEEE International Conference on Intelligent Transportation Engineering (ICITE)*, 2017, pp. 72–77.

[34] R. R. Sabbella and M. Arunachalam, "Functional safety development of motor control unit for electric vehicles," in *2019 IEEE Transportation Electrification Conference (ITEC-India)*, 2019, pp. 1–6.

[35] Texas Instruments, *F29H850TU-Q1, F29H859TU-Q1 Technical Reference Manual*, 2024, revision A, November 2024. [Online]. Available: https://www.ti.com/lit/pdf/sprsp93

[36] ——, *C29x CPU Reference Guide*, March 2025, literature Number: SPRUIY2A, Revision A. [Online]. Available: https://www.ti.com/lit/pdf/SPRUIY2

[37] ——, *AM263x Sitara Microcontrollers Technical Reference Manual*, October 2024, revision H. [Online]. Available: https://www.ti.com/lit/pdf/SPRUJ17

[38] Arm Ltd., *Cortex-R5 Processor: Technical Reference Manual (DDI0460D)*, 2012, revision D. [Online]. Available: https://developer.arm.com/documentation/ddi0460/d

[39] Texas Instruments, *Optimizing Applications with MCU SDK*, https://software-dl.ti.com/mcu-plus-sdk/esd/AM263X/09_00_00_35/exports/docs/api_guide_am263x/OPTIMIZING_APPLICATIONS_WITH_MCU_SDK.html, 2024, accessed July 21, 2025.

[40] Infineon Technologies AG, "Aurix™ TC4x microcontroller (promotional page)," https://www.infineon.cn/promo/aurix-tc4x, 2025, accessed July 21, 2025.

[41] Texas Instruments, *F29H85X-SOM-EVM ControlSOM Evaluation Board User's Guide*, December 2024, revision B. [Online]. Available: https://www.ti.com/lit/pdf/SPRUJE4

[42] ——, *AM263x Control Card Quick Start Guide*, https://www.ti.com/lit/an/sprade4/sprade4.pdf, 2023, accessed July 21, 2025.

[43] Infineon Technologies AG, *AURIX® Generic Timer Module Training*, Infineon Technologies AG, 2020, accessed: 2025-07-24. [Online]. Available: https://www.infineon.com/assets/row/public/documents/10/56/infineon-aurix-generic-timer-module-training-en.pdf

[44] Texas Instruments, *Application Software Optimization on the C29 CPU*, 2024, available at https://www.ti.com/lit/ug/sprujg0a/sprujg0a.pdf.

[45] ——, *F29H85x Flash API User's Guide*, 2024, available at https://www.ti.com.cn/cn/lit/ug/spruje7a/spruje7a.pdf.

[46] ——, *AM263x Sitara™ Microcontroller Silicon Revision 1.0A, 1.1*, https://www.ti.com/lit/er/sprz488e/sprz488e.pdf, 2024, accessed July 21, 2025.

[47] Infineon Technologies AG, *AURIX™ TC4x Converter-Digital-Signal-Processors (cDSP) Analog-to-Digital Converter*, https://www.infineon.com/dgdl/Infineon-AURIX_TC4x_Converter_DSP_Analog-to-Digital_Converter_V1.0.pdf-Training-v01_00-EN.pdf, 2024, accessed July 21, 2025.

[48] M. Hassan and K. Walluszik, "AURIX™ TC4xx - parallel processing unit (PPU) for multi-layer-perceptron (MLP) based direction of arrival estimation," in *AmE 2022 - Automotive meets Electronics; 13. GMM-Symposium*, 2022, pp. 1–6.