Delft University of Technology
Master of Science Thesis in Computer and Embedded Systems Engineering

Embedded Firmware Debugging and Telemetry

Ojasvi Kumar

TU Delft Delft University of Technology

Embedded Firmware Debugging and Telemetry

Master of Science Thesis in Computer and Embedded Systems Engineering

Embedded Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Ojasvi Kumar
okumar@student.tudelft.nl

22nd June 2024

Author
Ojasvi Kumar
Title
Embedded Firmware Debugging and Telemetry
MSc Presentation Date
26th June 2024

Graduation Committee

Georgi Gaydadjiev	Delft University of Technology
Przemyslaw Pawelczak	Delft University of Technology

Abstract

In the rapidly evolving realm of embedded systems, debugging is a critical component in the lifecycle of embedded firmware development, especially given the high demands of performance in real-time systems. Debugging embedded systems is inherently challenging and time-consuming due to limited internal system observability. Many modern processors, specifically those based on the ARM architecture, are equipped with debugging features such as Coresight components and debug registers to aid the debugging process. However, most debugging solutions rely on external hardware, which is often removed or disabled in a production system. Furthermore, existing software/hardware debug platforms do not support co-debug capabilities for multicore and multiprocessor debugging. This research aims to tackle the challenges developers face due to limited resources, strict real-time constraints, and the complex nature of debugging in embedded systems. The focus is on enhancing code efficiency and ensuring system responsiveness within stringent timing constraints. This thesis introduces a software library solution for debugging and profiling firmware, leveraging trace infrastructure and debug registers for multicore and multiprocessor ARM-based SoCs, facilitating comprehensive system analysis across ARM R5, M4, and M33 series processors. The library supports performance measurement, profiling, hardware breakpoints and watchpoints, synchronous breakpoints in multicore systems, and inter-processor communication. Furthermore, integrating Embedded Trace Macrocell and Micro Trace Buffer allows for detailed tracing across all processors, significantly enhancing system observability and debugging efficiency. Notably, the library is designed to be user-friendly, hardware-agnostic, and easily expandable to other ARMv7, and ARMv8 series processors. Comparative results documented in this study highlight the library's low overhead and latency. This thesis advances technical capabilities in debugging embedded systems and strategically enhances design decisions and process optimizations, ultimately contributing to more robust and efficient embedded solutions.

"If debugging is the process of removing software bugs, then programming must be the process of putting them in." – Edsger Dijkstra

Preface

The motivation for this thesis arose from an internship proposal at Marvell Technology, where they focus on high-performance embedded firmware for proprietary SoCs. Marvell was seeking innovative solutions to address the challenges associated with debugging embedded firmware. Having worked in the development of embedded systems for the past five years, I found these challenges resonated deeply with my own experiences. I was eager to explore these issues further and develop solutions to streamline and expedite the debugging process. This motivation stems from the realization that debugging often consumes more time than the actual development of code. Therefore, any improvement in this area could significantly enhance efficiency and productivity in the field of embedded systems development. This thesis is my endeavor to contribute to this exciting and essential aspect of embedded systems.

I would like to express my profound gratitude to Arnaud Gouder de Beauregard and Bart van Drongelen for providing me with the opportunity to collaborate with Marvell. Their proposed assignment was instrumental in shaping this research work and the solution proposed in this thesis.

My deepest appreciation goes to my supervisors, Georgi Gaydadjiev and Bart van Drongelen. Their invaluable advice, unwavering guidance, and numerous insightful sessions have been pivotal in steering the course of this thesis.

I am also immensely grateful to Przemyslaw Pawelczak for his willingness to serve as a committee member. His contribution of time and expertise in reviewing and assessing this work is highly appreciated. I hope he finds this thesis both engaging and enlightening.

Lastly, I sincerely thank the team at Marvell for their constructive feedback on my thesis. I earnestly hope this work will prove beneficial in future projects. This acknowledgment is a small token of my gratitude for their invaluable input.

Ojasvi Kumar

Delft, The Netherlands
25th June 2024

Contents

Chapter 1

Introduction

Embedded systems are integral to a myriad of applications, from simple household devices to complex automotive and aerospace controls. Each application demands rigorous standards for performance, reliability, and safety. However, the very nature of these systems, characterized by their tight coupling with hardware, limited resources, and real-time operational requirements, presents substantial challenges in firmware development, particularly in the areas of debugging and profiling [33].

Debugging embedded systems is notoriously difficult due to their constrained resources and limited visibility into running processes[53, 59]. Traditional debugging methods often rely heavily on external hardware, such as Joint Test Action Group(JTAG [31]) or in-circuit emulators which, while useful during development, are not viable in the final product stages [37]. This poses a significant problem in environments where systems must be debugged and optimized post-deployment without intrusive hardware [59]. Moreover, the existing platforms lack support for co-debugging in multicore and multiprocessor systems, further complicating the debugging process [64, 65].

The ARMv7 and ARMv8 architectures, which power many modern embedded systems, provide built-in debugging features [62, 43]. However, the potential of these features is not fully harnessed in many existing tools, particularly in the context of multicore and multiprocessor configurations.

This thesis aims to address these shortcomings by introducing a novel software library solution for debugging and profiling firmware in ARMv7 and ARMv8-based systems on Chip (SoCs). The proposed solution leverages the Trace Infrastructure and debug registers available in these systems, providing a comprehensive system analysis across various ARMv7 series processors.

The library is designed to be user-friendly, hardware-agnostic, and easily expandable, offering features such as performance measurement, profiling, hardware breakpoints and watchpoints, synchronous breakpoints in multicore systems, and inter-processor communication. The library significantly enhances system observability and debugging efficiency by integrating the Embedded Trace Macrocell (ETM [55]) and Micro Trace Buffer (MTB [46]).

Figure 1.1, as presented in the research paper "ARMV8 debug and trace architectures [62]" depicts the evolution of ARM architecture, highlighting the progression from early cores like ARM7TDMI, which supported basic ETM™ Trace and had a stopped clock for debugging[62], through to more advanced

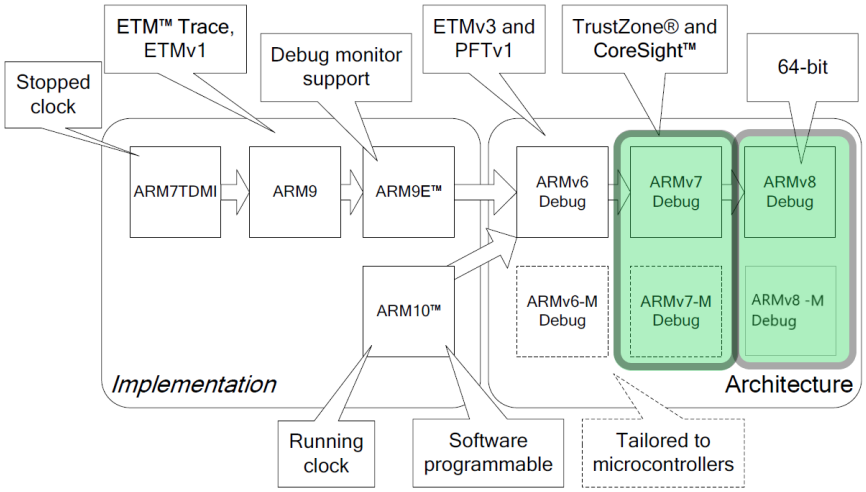

Figure 1.1: **Evolution of ARM debug Architecture(from left to right) [62] and Software Library's particular emphasis on the and ARMv7 and ARMv7-M Debug**

cores like ARM9 and ARM9E™ with enhanced debug monitor support and advanced Embedded Trace Macrocell ETMv3 [55] features. ARM10™ further improved flexibility with running clocks and software programmability of clocks. In the architecture section, ARMv6 introduced foundational debug features, while ARMv7 and ARMv8 cores, highlighted in green, showcased significant advancements such as TrustZone and CoreSight™ technology, providing robust security and comprehensive system visibility [62].

The focus of the debug library on ARMv7, ARMv7-M, and ARMv8-M cores is driven by their widespread adoption and versatility in many real-time and embedded applications. These cores are integral to various consumer electronics, industrial automation, and other embedded systems, making them a crucial target for a robust debugging tool. ARMv7, ARMv8, and ARMv7-M provide advanced debugging features, such as Data and Watchpoint Trace Unit (DWT) to apply watchpoints, Flash Patch and Breakpoint Unit (FPB) for the support of breakpoints, Performance Measurement Unit (PMU), and CoreSight™, enabling efficient diagnosis and resolution of system issues while optimizing for performance and power efficiency in resource-constrained environments.

While the scope of the library proposed in this is currently limited to only ARMv7, ARMv7-M, and ARMv8-M cores, with minor modifications, the library can be extended to support other ARM cores having similar debugging features, ensuring its utility across a broader range of platforms and future-proofing it against advancements in ARM technology. This targeted approach allows for a deep and effective implementation of a working system while maintaining the flexibility to extend the support when needed.

1.1 Problem Statement

While effective during the development phase, external hardware-based debugging tools often cannot be integrated into the final product due to cost, space, security, and power constraints. This leads to significant challenges in maintaining observability, particularly after deployment, thereby complicating maintenance and debugging.

The need for a software-based debugging approach that effectively utilizes ARM's debugging capabilities is evident, especially one that can adapt to the architecture's evolution, enhance the system's observability in the production phase, and support a wide range of ARM processors.

1.2 Thesis Objectives

This thesis aims to address the challenges through the following objectives:

- To design and implement a versatile, software-only debugging library tailored for ARM-based multicore and multiprocessor SoCs. This library will utilize ARM's built-in Trace Infrastructure and debug registers to enable detailed observation and control over system operations without the need for external hardware.
- To ensure that the debugging library is hardware-agnostic, capable of supporting a wide array of ARM processors including the AM R5, M4, and M33 series, and easily extendable to other ARMv7 and ARMv8 series processors. This goal aims to future-proof the tool against the rapid evolution of embedded processor technologies.
- To optimize the debugging library and to ensure it introduces minimal overhead and maintains low latency, making it suitable for use in real-time embedded systems where performance is critical.
- To conduct comprehensive testing and analysis against existing hardware-based debugging methods to validate the effectiveness, efficiency, and advantages of the software-based approach. This will include detailed performance measurements, profiling capabilities, and real-world usability.
- To leverage the insights gained from the advanced debugging features to aid in design decisions, bottleneck identification, and performance optimization, ultimately contributing to the enhancement of the overall reliability and efficiency of embedded systems.

1.3 Thesis Structure

The structure of this report is as follows. Chapter 2 provides an in-depth background, discussing the fundamental concepts, existing technologies, and the challenges faced in embedded system debugging. Chapter 3 delves into the architecture and system design, outlining the core components, their interactions, and the overall design of the embedded debug library. Chapter 4 focuses on the technical implementation, detailing the sequence diagrams, API functionalities, and the practical steps involved in integrating the library with various

systems. Chapter 5 transitions into porting on hardware, where the library is adapted to specific development kits, showcasing the setup, configuration, and validation processes. Chapter 6 presents the evaluation and results, thoroughly analyzing the library's performance, effectiveness, and outcomes of various tests conducted on different hardware platforms. Chapter 7 concludes the thesis by summarizing the key findings, contributions, and overall impact of the research. Finally, Chapter 8 discusses the limitations of the current implementation and proposes directions for future work.

Chapter 2

Background

Debugging plays a pivotal role in the development lifecycle of embedded firmware, especially in the context of high-performance systems [37, 60]. It serves as the linchpin for ensuring the reliability, functionality, and performance of the firmware. The intricate nature of embedded systems, often constrained by limited resources and real-time requirements, underscores the significance of effective debugging methodologies [57]. Identifying and rectifying errors in the code ensures the proper functioning of the firmware and contributes to the embedded system's overall efficiency [49]. Debugging is the key to unveiling latent issues, optimizing code for resource utilization, and fine-tuning algorithms to meet stringent performance criteria. It facilitates the discovery of bottlenecks, enables swift resolution of issues, and streamlines the development process, ultimately creating robust, high-performance embedded systems.

2.1 Introduction to Debugging

Embedded firmware development encompasses various software issues across environments such as Real-Time Operating Systems (RTOS), bare metal applications, and systems with advanced multicore architectures. This section explores various types of bugs typical in these setups and highlights differences in their emergence during development versus production phases. Some of the common bugs are as follows:

2.1.1 Bare Metal Bugs

Bare metal bugs refer to the issues that arise when programming at the hardware level without an operating system's abstraction layer. Some of these bugs are as follows:

- **Interrupt Handling Bugs:** Issues occur when interrupts are not managed correctly, leading to missed interrupts or incorrect interrupt service routines (ISR) handling [53].
- **Resource Allocation Errors:** Without an OS managing hardware resources, manual errors in allocation can lead to conflicts or inefficient resource use [37].

- **Timing Errors:** Mistakes in timing calculations can result in failures to meet hardware interaction timings or clock synchronization problems [57].

2.1.2 RTOS-Specific Bugs

This section discusses specific bugs that can occur in a Real-Time Operating System (RTOS). Some of the common bugs are as follows:

- **Task Synchronization Bugs:** These arise when tasks do not synchronize properly through semaphores or mutexes, leading to race conditions or deadlocks[33, 53].
- **Priority Inversion:** Occurs when a higher priority task is indirectly preempted by a lower priority task holding a shared resource, due to intermediate priority tasks taking CPU time.
- **Memory Corruption:** Happens due to improper management of shared resources or buffer overflows, common in multitasking environments. [20]

2.1.3 Concurrency Bugs

Concurrency bugs are errors that occur when multiple processes are executing concurrently. They include:

- **Race Conditions:** Occur when the outcome depends on the non-deterministic ordering of events, such as accessing shared data without proper locking[50].
- **Deadlocks:** Arise when two or more processes block each other by holding a resource the other needs [65].
- **Livelocks:** Situations where tasks continuously change states in response to other tasks without making progress [45].
- **Heisenbugs:** These errors seem to disappear or alter when one attempts to study them, often due to timing or environmental changes[67, 20]. Attaching a debugger disturbs the timing of a parallel program, easily masking errors. [22]

2.1.4 Multicore and Multiprocessor Bugs

Multicore and multiprocessor bugs are issues that arise when programming systems with multiple cores or processors. They include:

- **Cache Coherency Issues:** These problems occur when multiple processors have inconsistent views of shared data, leading to incorrect program execution [42].
- **Thread Migration Issues:** Bugs can happen when threads are moved between cores, leading to inefficiencies or incorrect behavior if not managed correctly.
- **Synchronization Challenges:** Advanced synchronization techniques are required to manage access to shared memory across multiple cores, often leading to complex bugs[50, 18].

2.2 Components and Requirements of a Typical Debugging System

A fully-fledged debugging system encompasses a range of tools and features to facilitate efficient and comprehensive debugging of software applications. The requirements for such a system can vary based on the complexity of the software, the target platform, and the development environment. Components provide the necessary tools and capabilities to interact deeply with the hardware and software of the target system, while requirements ensure that these tools perform optimally under various constraints and scenarios. This comprehensive approach ensures that the debugging system can handle the complexity of modern embedded systems, providing developers with powerful and intuitive tools to identify, diagnose, and resolve issues swiftly and effectively. Here are the key components and requirements for a robust debugging system:

2.2.1 Components

Components refer to the elements of the debugging system. These are essential for establishing the framework within which debugging activities are performed.

Low-Level Debugging

The system provides essential hardware interface support, including Joint Test Action Group(JTAG [31]) and Serial-Wire Debug (SWD [54]), which are pivotal for direct and intricate hardware interactions necessary for precise control and observation of system states[44]. These interfaces enable low-level access to the processor, allowing developers to set breakpoints, perform single-step debugging, and inspect or modify memory and registers[59, 37]. Real-time debugging capabilities are also incorporated, allowing developers to perform diagnostics and make adjustments without halting the system, crucial in environments where system downtime can lead to significant disruptions or data loss such as debug monitor mode[31]. Watchpoints, which monitor specific memory locations, are also supported, enabling the detection of unexpected changes in critical data areas, further aiding in the diagnosis of complex issues [52].

Trace and Logging

Tracing is capturing data that illustrates how the components in a design are operating, executing, and performing for example Instruction Trace generates information about the instruction execution of a core [62]. Advanced trace capabilities, such as those provided by the Embedded Trace Macrocell (ETM), offer detailed and comprehensive analysis capabilities with minimal impact on system performance[48]. This is crucial for tracking down elusive bugs that manifest under specific conditions [59]. The integration with logging systems ensures that runtime data is continuously collected, providing valuable insights during debugging and retrospective analysis sessions.

Multicore Debugging

With support for multicore processors, the system addresses the increased complexity and concurrency issues found in contemporary embedded systems [50]. This component is complemented by provisions for additional debugging hardware integration, facilitating enhanced analysis and diagnostics in highly parallel operational settings [34]. The ability to set synchronous breakpoints across multiple cores allows for coordinated debugging, ensuring that the interactions between different processors can be thoroughly examined and understood [64]. Mechanisms to manage a range of issues such as thread interaction, control of timers, semaphores, and mutexes, IPC message passing, event handling among different cores should also be a part of the system [57].

Performance Optimization

The system includes performance monitoring tools, such as Performance Measurement Units (PMUs), which play a critical role in identifying performance bottlenecks and optimizing software execution [13, 45]. Profiling tools are integrated to provide detailed insights into the execution flow and resource usage, helping developers optimize their code effectively [19]. The system also interfaces with memory to monitor and manage usage, ensuring that performance metrics reflect real operational conditions accurately.

2.2.2 Requirements

Requirements reflect the functional needs and conditions that the debugging system must meet to be effective in various environments and scenarios.

1. **Resource-Constrained Environments:** The debugging system is designed to maintain minimal resource overhead to ensure that it can operate effectively within the tight constraints typical of embedded environments such as having minimum latency and memory footprint[59]. Efficient memory inspection capabilities allow for quick checks and modifications of system memory without significant performance degradation, an essential feature in systems with limited memory capacity.

2. **Interface Limitations:** To manage the challenges posed by low-bandwidth interfaces, the system employs data encoding and compression techniques to maximize the efficiency of data transmission. This includes buffering and flow control strategies that optimize the use of interface bandwidth, ensuring effective communication even under restricted conditions.

3. **Integration with Design Decisions:** The debugging system provides critical insights that inform design decisions regarding the configuration of CPUs and their clock speeds within an SoC [53, 47]. This strategic guidance is vital for optimizing both system performance and power consumption.

4. **Ease of Use:** A user-friendly interface is a key feature of the debugging system, making it accessible and easy to operate for firmware developers [31]. This simplifies the debugging process, enabling efficient and effective troubleshooting without requiring extensive training.

5. **Security Considerations:** Security features are integrated into the debugging system to protect sensitive data and maintain the integrity of the debugging process[44]. This is particularly crucial in environments dealing with confidential or safety-critical data [66].

6. **Documentation and Training:** Comprehensive documentation and robust training resources are provided with the debugging system. This ensures that all users can fully leverage the debugging tools' capabilities, enhancing the debugging process's effectiveness and reducing the learning curve for new users.

7. **Compatibility with SoCs:** The debugging system is designed to be compatible with a wide range of System-on-Chips consisting of ARMv7 and ARMv8 processors. This ensures broad applicability across different platforms, making it a versatile tool in various hardware environments.

2.3 Current Solutions in the Field

The landscape of debugging solutions in the market is diverse and continually evolving, driven by the increasing complexity of embedded systems and the need for efficient fault detection and resolution. These solutions range from traditional debugging tools, such as JTAG and SWD debuggers, to advanced telemetry systems that provide real-time insights into system performance [52]. Additionally, software-based solutions like GDB and Open-OCD offer powerful features for code-level debugging. This comprehensive exploration delves into the key categories of debugging solutions: hardware-focused, software-focused, and simulation debugging, providing insights into their functionalities, use cases, and essential interactions [16]. However, each of these solutions comes with its own set of advantages and limitations, necessitating a careful evaluation based on specific debugging requirements. This section will delve into the details of these current market solutions, providing a comprehensive overview of their functionalities, use cases, and comparative analysis.

2.3.1 Hardware-Focused Debugging Tools

Hardware-focused debugging tools are essential for developers requiring direct interaction with the system's physical components. These tools enable deep access to the system's core at a hardware level, facilitating real-time diagnostics and modifications critical in tightly controlled or high-stakes environments. The most common hardware-focused debugging tools currently used in the industry are as follows:

JTAG (Joint Test Action Group)

JTAG is a standardized interface for on-chip debugging that enables access to various debugging and testing features. JTAG is a vital hardware interface for on-chip debugging, providing low-level access for inspecting and modifying memory, halting the processor, and single-stepping through code, which is indispensable for boundary scan testing and real-time debugging [44, 43]. Despite its comprehensive capabilities, JTAG requires additional hardware support on the

chip and may encounter bandwidth limitations in high-performance systems. Implementing JTAG involves designing the chip with JTAG support, connecting a JTAG debugger hardware such as SEGGER J-Link or ARM DSTREAM, and utilizing debugging software like GDB or IDEs such as Keil or IAR.

SWD (Serial Wire Debug)

SWD is designed as a two-wire, lower pin-count alternative to JTAG [54]. Its reduced pin count simplifies hardware design but is limited to Cortex-M microcontrollers. To implement SWD, developers need to integrate SWD support in the chip design and connect with debuggers like Segger J-Link or CMSIS-DAP compliant debuggers.

ARM DSTREAM

The DSTREAM unit, an integral part of the ARM development environment, includes a debugger, performance analyzer, and system profiler. It is designed to support multicore debugging, trace analysis, and performance optimization for ARM-based systems. The DSTREAM unit, when integrated with the DS-5 Debugger and ETM (Embedded Trace Macrocell) support, provides an advanced debugging environment [45]. One of its key advantages is its ability to support multicore debugging and trace analysis. However, it does require the DS-5 Debugger for advanced trace and debugging capabilities. Additionally, being a proprietary tool, it may incur potential licensing costs. To fully implement the DSTREAM unit support, it needs to be integrated into the development environment. Following this, the DSTREAM unit should be connected to the ARM processors to facilitate advanced debugging and trace analysis. This setup provides a comprehensive solution for debugging and performance optimization of ARM-based systems.

CORESIGHT Debug

Figure 2.1 illustrates a CoreSight-based debugging architecture for ARM systems, showing the interaction between various components involved in the debugging process. PC-based debugging tool interfaces with the target system using JTAG or Serial Wire (SW). This connection feeds into a Debug Access Port, which serves as the gateway for all debugging communications.

Within the target system, the ARM processor includes built-in debug logic that connects to the CoreSight capture system via trace components like ETM (Embedded Trace Macrocell)[45] and trigger units [44]. Embedded Trace Macrocell (ETM) offers detailed analysis of program execution through instruction and data trace capabilities, providing high-level insights into software execution and performance analysis without significantly impacting system performance [62, 65]. The CoreSight capture system collects trace data and is directly linked to a Trace Port Analyzer, which sends the captured data back to the PC-based debugging tool for analysis.

ETM's implementation, however, requires Coresight hardware support in the chip and external hardware for decoding trace data at high speeds [3]. Tools like ARM's DS-5 Debugger and Lauterbach TRACE32 are commonly used to enhance ETM's capabilities, providing advanced trace and debugging features.

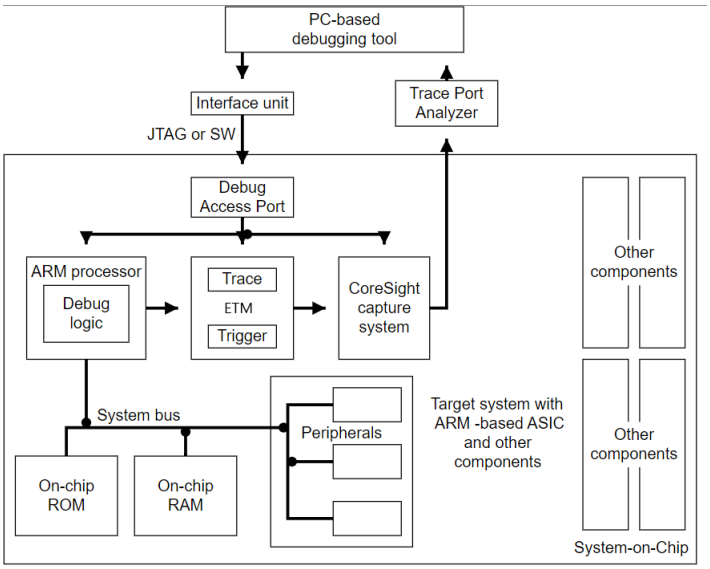

Figure 2.1: **Coresight Debug Architecture and Program Flow [4]**

2.3.2 Software-Focused Debugging Tools

Software-focused debugging tools provide powerful capabilities for source-level debugging and are particularly beneficial in environments where hardware access is constrained or non-intrusive methods are preferred. The most common software-focused debugging tools currently used in the industry are as follows:

GDB (GNU Debugger)

GDB is a cornerstone in the software debugging landscape, supporting various ARM architectures and offering extensive debugging capabilities across both bare metal and operating system environments [34]. While GDB can function independently, it often utilizes JTAG or SWD interfaces for low-level hardware interaction and is typically used with a GDB server for remote debugging over a network.

OpenOCD (Open On-Chip Debugger)

OpenOCD supports a range of on-chip debugging protocols, including JTAG and SWD, and is noted for its versatility and open-source nature[33, 24]. Although powerful, OpenOCD requires careful configuration and setup. It is frequently used alongside GDB to provide a comprehensive debugging environment, particularly for ARM processors.

Logging System

A logging system involves systematically recording events, messages, or data during the execution of a program [55]. This recorded information is stored in log files or some memory location, transferred over some interface, and can

include variable values, function calls, and custom messages. Logging systems are crucial for capturing detailed runtime data, enabling post-mortem analysis, and aiding in root cause discovery. These systems are highly customizable and function independently of hardware, focusing instead on the strategic insertion of logging statements within the software to capture critical diagnostic information. If well-architected, logging creates a small and deterministic intrusion. But, depending on the architecture, logging might also completely change the system's behavior [55].

2.3.3 Simulation Debugging

Simulation debugging involves using software tools to emulate hardware or an entire system, allowing developers to execute and analyze their code in a controlled environment [20]. This method is highly beneficial for early stages of development when physical hardware might not be available, for complex systems where setting up real-world scenarios is impractical, or in educational settings where cost and accessibility are factors[22].

Simulation debugging offers several advantages such as it is usually non-intrusive, unlike hardware probes and debuggers that might alter the system's behavior. It provides complete control over the execution environment, including the ability to pause, inspect, and modify the state at any point. Simulated environments can be deterministic, which simplifies the reproduction of bugs. Features such as reverse debugging [21, 17] and deterministic replay allow developers to step back in time to understand how the system reached a certain state and to rewind the program's state to any previous point in its execution history, respectively. These features are invaluable for examining the sequence of events leading to a bug. Reverse debugging also enables developers to inspect variables, memory, and processor states at any point in the execution timeline, offering insights into the cause-and-effect relationships that led to a fault[22].

However, simulation debugging also has its disadvantages. The accuracy of the simulation model can lead to scenarios where bugs are missed because they don't manifest in the simulation or, conversely, where issues appear in the simulation that wouldn't occur in reality[20, 12]. Simulation often runs significantly slower than real-time, especially in full-system emulation. This performance overhead can make debugging time-consuming, particularly for large applications or systems that require real-time or near-real-time performance[63, 32, 19]. It can also obscure timing-related issues that only appear under real operational loads. Setting up a simulation environment, especially for complex systems, can be a daunting task requiring significant computational resources [42]. Simulated environments might not be able to fully emulate the nuances of interaction with the physical world, such as sensor inputs interference. Despite these challenges, simulation debugging remains a critical component of the software development process. The most widely used simulation software packages include:

QEMU

QEMU is a versatile open-source machine emulator and virtualizer that can run operating systems and programs made for one machine on a different machine. It operates in two modes: full system emulation and user-level emulation. QEMU's dynamic translation feature ensures efficient execution, and it provides

a rich set of device models, making it versatile for various simulation needs [12]. However, achieving high-speed execution can sometimes come at the cost of reduced accuracy in simulating hardware behavior. While QEMU supports a wide range of devices and architectures, its ability to simulate specific peripherals or proprietary hardware accurately might be limited [20].

Simics

Simics is a full-system simulator by Intel that allows the creation of a virtual platform of an entire system, including processors, devices, and network components [42]. It enables developers to simulate any target hardware, facilitating debugging, testing, and system analysis [2]. Simics excels in system-level simulation with high fidelity and advanced features like reverse debugging. However, as a commercial product, accessing Simics can involve significant costs. While offering detailed simulation capabilities, the execution speed in Simics can be slower compared to running on actual hardware, which might affect its suitability for performance-critical debugging scenarios.

2.3.4 Comparison of Current Solutions

Components	JTAG	SWD	ARM DSTREAM	Coresight Debug	GDB	OpenOCD	Logging System	QEMU	Simics
Breakpoint	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Synchronous Breakpoint	Limited	No	Yes	Yes	No	No	No	No	No
Core Dump	No	No	Yes	No	Yes	Yes	Yes	Yes	Yes
Watchpoint	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Dependency on External Hardware	Yes	Yes	Yes	Yes	Limited	Limited	No	No	No
Performance Measurement Unit (PMU)	No	No	Yes	Yes	Limited	Limited	No	No	No
Multicore Support	Limited	No	Yes	Yes	Limited	Limited	No	Yes	Yes
Trace Buffer Support	Limited	No	Yes	No	No	No	Yes	No	No
Reverse debugging	No	No	No	Limited	Yes	No	No	Yes	Yes

Table 2.1: **Table comparing the key features of various debugging solutions, including traditional hardware-focused tools, software-focused tools, and simulation debugging tools**

Table 2.1 compares key features across various debugging solutions, encompassing traditional hardware-focused tools, software-focused tools, and simulation debugging tools. Each solution is evaluated based on its capability to handle crucial debugging tasks such as breakpoints, synchronous breakpoints, core dumps, watchpoints, dependency on external hardware, performance measurement units (PMUs), multicore support, and trace buffer support.

JTAG and SWD are prominent hardware-focused debugging tools that provide extensive support for breakpoints, and watchpoints. Both JTAG and SWD require external hardware, which can be a limiting factor in some environments. JTAG offers limited multicore support and no support for performance measurement units (PMUs), whereas SWD does not support multicore systems as well as PMUs.

ARM DSTREAM and Coresight ETM stand out for their advanced capabilities in multicore systems and performance measurement. These tools also support core dumps, and trace buffers, making them suitable for high-performance debugging tasks. However, ARM DSTREAM and Coresight ETM are reliant on proprietary hardware and may involve additional costs.

GDB and OpenOCD are software-focused debugging tools that provide extensive capabilities for breakpoints and watchpoints having a limited dependency on external hardware. Both tools offer limited support for performance measurement and multicore systems, which may restrict their effectiveness in complex debugging scenarios.

Logging systems focus on capturing detailed runtime data for post-mortem analysis and do not rely on hardware, making them highly customizable and suitable for software-based diagnostics. However, they lack real-time debugging capabilities and do not support breakpoints, synchronous breakpoints, or watchpoints.

QEMU and Simics are powerful simulation debugging tools that offer a non-intrusive environment for comprehensive system emulation. They provide core dump capabilities and support for multicore systems, with Simics additionally offering reverse debugging features. However, simulation debugging lacks hardware debugging capabilities.

In summary, each debugging solution presents a unique set of strengths and limitations. Hardware-focused tools like JTAG and SWD provide deep hardware access but require external hardware. Advanced tools like ARM DSTREAM and Coresight ETM offer robust performance measurement and multicore support but are more complex and costly. Software-based tools like GDB and OpenOCD are versatile and hardware-independent but limited in advanced performance features. Simulation tools like QEMU and Simics offer comprehensive system emulation but lack direct hardware debugging capabilities.

2.4 Challenges with Current Methods

Debugging embedded systems, particularly those based on complex architectures like multicore or multiprocessor setups running an RTOS or bare metal configurations, poses several significant challenges. The limitations of current debugging tools and methods are often exacerbated by the inherent properties of embedded environments, including real-time operation, resource constraints, and the necessity for non-intrusive debugging techniques. Here are some of the key challenges faced:

Limited System Observability

One of the primary challenges in debugging embedded systems is the limited system observability. Many traditional debugging tools require halting the system to inspect the state or implement breakpoints, which is not feasible in real-time systems where stopping might disrupt critical operations. This intrusiveness can lead to significant operational disruptions. Additionally, these tools often provide insufficient insights into the internal state of the firmware, interactions between threads or processes, and the handling of asynchronous events.

Complexity in Multicore/Multiprocessor Systems

Debugging tools often struggle to handle the inherent complexity of multicore and multiprocessor systems. Concurrency issues such as race conditions and deadlocks are particularly challenging to address, as they occur across multiple

cores or processors. Synchronization of data collection across different execution threads without causing performance bottlenecks or data corruption requires sophisticated mechanisms. [36] These complexities necessitate advanced debugging tools that can manage concurrent operations efficiently and provide accurate diagnostic information without introducing significant overhead.

Real-Time Constraints

Embedded systems often operate under stringent real-time constraints, where the timing accuracy of debugging tools is critical. Tools that introduce additional latency can alter the system's behavior, leading to the "observer effect" where the act of debugging changes the system's performance [35]. Additionally, there is a need for real-time data analysis and feedback, which many traditional debugging tools do not support. The inability to perform real-time analysis limits the effectiveness of these tools in diagnosing and resolving issues in systems that require continuous and precise operation [53].

Dependence on External Hardware

Many powerful debugging features require additional hardware like JTAG or trace probes, which may not be available in the production or testing environment. The dependency on such hardware adds to the cost and complexity of the debugging process [62]. Moreover, debugging capabilities that rely on hardware are often stripped in production to save cost and space, reducing the ability to perform post-deployment diagnostics. This dependence on external hardware limits the flexibility and scalability of debugging tools, making it difficult to perform effective diagnostics and maintenance in deployed systems.

2.5 Motivation for a Comprehensive Debugging Library

Given the challenges associated with current debugging methods for embedded systems, there is a clear and pressing need for an advanced, efficient, and flexible debugging solution such embedded debug library. Traditional debugging tools, such as JTAG and SWD, although powerful, often require halting the system to inspect its state, which is impractical for real-time systems. Additionally, they rely heavily on external hardware, which may not be available in production environments and can be removed for security reasons or to save costs and space, severely limiting post-deployment diagnostics. Current software-based tools like GDB and OpenOCD, while providing source-level debugging, often fall short in handling the complexities of multicore and multiprocessor environments. They also struggle with providing real-time analysis and feedback, essential for systems operating under stringent timing constraints. Furthermore, the existing tools lack sufficient observability of the system's internal state and interactions between processes, making it difficult to diagnose and resolve issues efficiently. For more insight Coresight technology is present, but it's not used to its potential because of its dependency on hardware such as DSTREAM.

The embedded debug library presented in this thesis will address the above limitations by integrating a comprehensive suite of tools designed to provide

deep insights into the system's operations without the need for external hardware. By leveraging advanced features such as synchronous breakpoints, interprocessor communication, and performance measurement units (PMUs), the software library will ensure minimal resource overhead and support real-time debugging, crucial for maintaining system performance and reliability. The inclusion of core dump capabilities and watchpoints will further enhance the ability to capture and analyze system states accurately, facilitating quick identification and resolution of bugs. Moreover, the library's user-friendly interface and seamless integration with existing development environments will reduce the complexity of the debugging process, making it accessible even to developers with limited expertise in hardware-level debugging. This combination of advanced features and ease of use will position the proposed library as an essential tool for developers, enabling them to navigate the complexities of modern embedded systems efficiently and effectively.

Chapter 3

Architecture and System Design

This chapter outlines the architecture and system design of the proposed software library for debugging and profiling ARM-based multicore and multiprocessor system-on-chips (SoCs). The design aims to leverage the ARM architecture's existing capabilities such as Trace Infrastructure and debug registers, enhancing them with a software layer that provides extensive debugging functionalities without external hardware dependencies. This chapter details the conceptual framework, design considerations, and architectural components of the library, providing a comprehensive understanding of how the library addresses the challenges associated with embedded system debugging.

Figure 3.1 illustrates the Host PC interaction with the Target SoC and a high-level System Design of the SoC's debugging components already present and supported by Hardware.

The Host PC serves as the development and debugging station, equipped with essential tools such as an Integrated Development Environment (IDE), compiler, JTAG, SWD, GDB, and a DS-STREAM trace decoder. These tools interface with the target device via debugging protocols (JTAG or SWD) through an Interface Unit, allowing developers to set breakpoints, monitor system states, and trace program execution in real time. This setup provides a powerful and user-friendly environment for embedded system development and debugging.

The Target Device SoC comprises several key components that work together to facilitate debugging. The Debug Access Port (DAP) acts as the central interface for external hardware debug access through the APB bus, connecting to various debug components within the SoC. The DAP serves as the gateway for all debugging communications between the Host PC and the SoC, ensuring seamless data exchange. Figure 3.1 highlights three specific ARM Cortex cores, each with dedicated debug support: the M4 Debug setup includes the Flash and Breakpoint Unit (FPB), Debug and Watchpoint Trace Unit (DWT), Embedded Trace Macrocell (ETM), and a ROM Table, enabling detailed monitoring and control of the Cortex-M4 core; the R5 Debug setup features CoreSight debug, Performance Measurement Unit (PMU), ETM, and a ROM Table, providing comprehensive debugging and performance monitoring capabilities for the Cortex-R5 core; the M33 Debug setup includes FPB, DWT, ETM, and a ROM

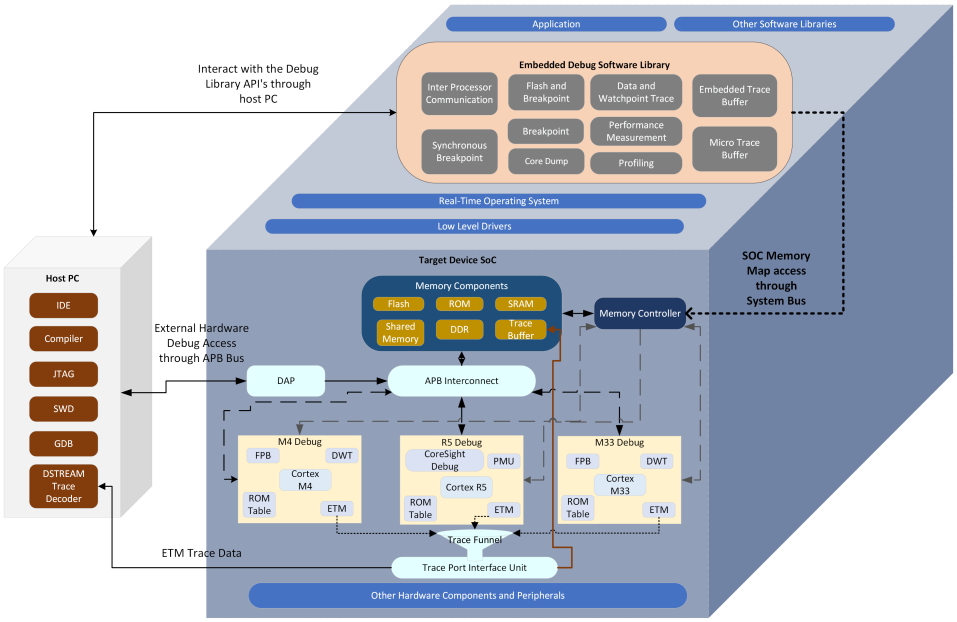

Figure 3.1: **Architecture of Debugging in Target SoC illustrating the interaction and components of Embedded Software Debugging Library with the ARM cores and hardware components**

Table, similar to the M4, but tailored for the Cortex-M33 core with additional security features like TrustZone. As mentioned in 1 the scope of this library is limited to these cores but can be easily tailored to support similar ARM Cores.

The SoC also includes various memory components such as Flash, ROM, SRAM, Shared Memory, DDR, and Trace Buffer, all managed by the Memory Controller. The SoC memory map is accessed through the system bus, allowing for efficient data management and storage during debugging sessions.

The core of this architecture is the Embedded Debug Software Library, which integrates seamlessly with the hardware components to provide a comprehensive debugging solution.

3.1 Proposed Architecture

Figure 3.1 also illustrates the comprehensive architecture of the Embedded Debug Software Library, showcasing its integration with ARM Cortex cores within a System-on-Chip (SoC) environment.

The library consists of several key modules. The Core Components include functionalities for Breakpoint, Synchronous Breakpoint, Core Dump, and Watchpoint. These components allow developers to set breakpoints, halt execution across multiple processors simultaneously, capture system states, and monitor specific memory locations for changes. The Flash Patch and Breakpoint Unit (FPB) manages breakpoints in flash memory, enabling precise control over code execution and facilitating debugging of code stored in non-volatile memory. The Debug and Watchpoint Trace Unit (DWT) monitors data access and provides

detailed trace information, helping to identify and resolve data-related issues within the system.

Performance monitoring is managed through the Performance Measurement Unit (PMU) and profiling tools, which track various performance metrics and help optimize system performance by identifying bottlenecks. Trace and logging functionalities utilize the Embedded Trace Buffer (ETB) and Micro Trace Buffer (MTB) to capture detailed execution traces, providing insights into the system's behavior over time. These traces are invaluable for diagnosing elusive bugs and understanding the flow of execution. The Inter Processor Communication (IPC) module manages communication between multiple processors, ensuring synchronized operations and efficient data exchange in multicore systems.

The Embedded Debug Software Library is designed to be highly functional even without the continuous presence of the Host PC. The library components can operate independently, saving profiling data, performance measurement data, and ETM trace data in dedicated memory locations. This data can be extracted later for postmortem analysis, allowing for comprehensive diagnostics and optimization after the system has experienced an issue.

The following sections detail the main components of the Embedded Debug Software Library, explaining their functionalities and how each of the hardware components present in SoC are leveraged :

3.2 Targeted ARM Cores

This section supports the various ARM cores and their components that are supported by the Software Debug. As discussed in Chapter 1, the Embedded Debug Library primarily focuses on ARMv7, ARMv7-M, and ARMv8-M processors, specifically targeting the R5, M4, and M33 cores. This emphasis is due to the library's thorough testing and adaptation of hardware featuring these cores. These processors are widely used in various embedded and real-time applications, making them an ideal focus for the library. Despite this targeted approach, the library is designed with flexibility in mind and can be readily extended to support other ARM cores with similar components, requiring only minor modifications. This adaptability ensures that the library can evolve to meet the needs of a broader range of applications and hardware configurations.

ARM Cortex-R5 Architecture

Figure 3.2 illustrates the ARM Cortex-R5 architecture [7], based on the ARMv7-R architecture. The Cortex-R5 is designed for high-performance real-time applications and includes features like ECC (Error Correction Code) for both memory and bus, multiple TCMs (Tightly Coupled Memories), and various cache options (D cache, I cache). It also integrates a robust memory protection unit (MPU) and a floating-point unit (FPU). For debugging, it includes CoreSight multicore debug and trace capabilities, making it suitable for complex, real-time embedded systems.

ARM Cortex-M4 Architecture

Figure 3.3 depicts the architecture of the ARM Cortex-M4 core [6], which falls under the ARMv7-M architecture. The Cortex-M4 features a range of compon-

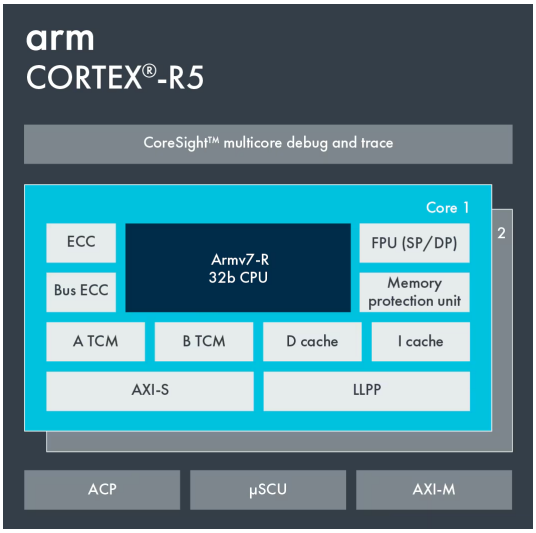

Figure 3.2: **ARM Cortex-R5 Architecture with supported modules [7]**

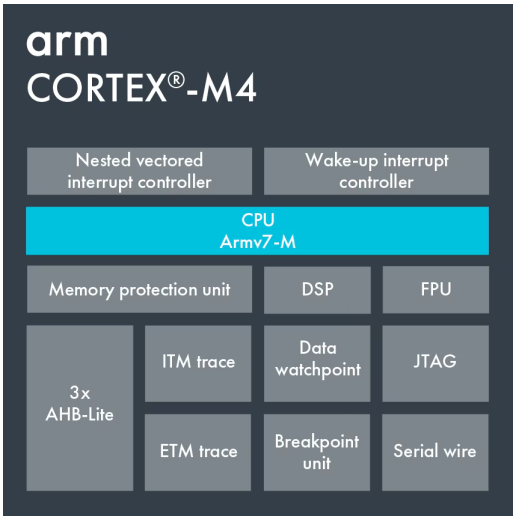

Figure 3.3: **ARM Cortex-M4 Architecture with supported modules[6]**

ents essential for embedded systems, including a CPU based on the Armv7-M architecture, a nested vectored interrupt controller (NVIC), and a wake-up interrupt controller. It also includes a Memory Protection Unit (MPU), a digital signal processor (DSP), and a floating-point unit (FPU). For debugging and tracing, it supports ITM trace, ETM trace, data watchpoints, and breakpoint units, alongside interfaces such as JTAG and serial wire [11].

ARM Cortex-M33 Architecture

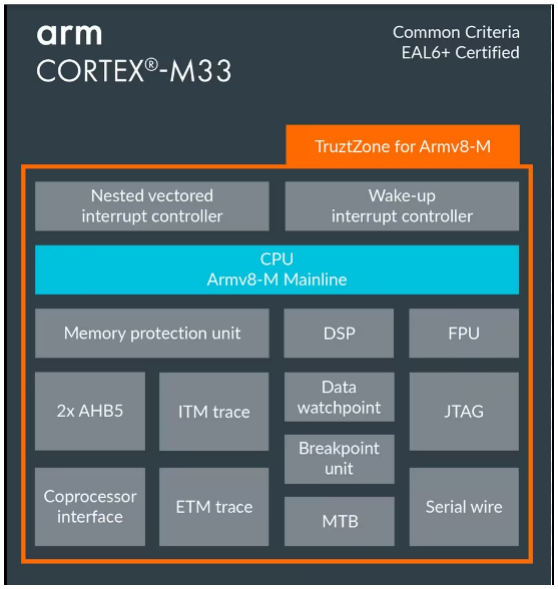

Figure 3.4: **ARM Cortex-M33 Architecture with supported modules [5]**

Figure 3.4 showcases the ARM Cortex-M33 architecture [5], an ARMv8-M processor. The Cortex-M33 builds on the M4 with additional features like TrustZone for enhanced security. It retains similar components such as the NVIC, wake-up interrupt controller, MPU, DSP, and FPU, while introducing a coprocessor interface and enhanced trace capabilities including MTB (Micro Trace Buffer). This architecture supports advanced debugging features such as ITM trace, ETM trace, data watchpoints, breakpoint units, and JTAG and serial wire interfaces.

The specific debugging components supported by each core are explained in the further sections.

3.3 Breakpoints and Watchpoints

Breakpoints allow developers to halt program execution at specific points, enabling them to inspect the system state, examine variable values, and understand the control flow of the application [66]. Watchpoints, on the other hand, monitor specific data addresses for read, write, or read/write operations [62].

3.3.1 Debug Register Interface - Cortex-R5

Offset (hex)	Register number	Access	Mnemonic	Description
0x000	c0	R	DBGDIDR	*CP14 c0, Debug ID Register*
0x18	c6	RW	DBGWFAR	*Watchpoint Fault Address Register*
0x01C	c7	RW	DBGVCR	*Vector Catch Register*
0x028	c10	RW	DBGDSCCR	*Debug State Cache Control Register.*
0x080	c32	RW	DBGDTRRXext	*Data Transfer Register.*
0x084	c33	W	DBGITR	*Instruction Transfer Register.*
0x088	c34	RW	DBGDSCRext	*CP14 c1, Debug Status and Control Register.*
0x08C	c35	RW	DBGDTRTXext	*Data Transfer Register.*
0x090	c36	W	DBGDRCR	*Debug Run Control Register.*
0x100-0x11C	c64-c71	RW	DBGBVR	*Breakpoint Value Registers.*
0x140-0x15C	c80-c87	RW	DBGBCR	*Breakpoint Control Registers.*
0x180-0x19C	c96-c103	RW	DBGWVR	*Watchpoint Value Registers.*
0x1C0-0x1DC	c112-c119	RW	DBGWCR	*Watchpoint Control Registers.*

Table 3.1: **Debug Memory-Mapped Registers for Cortex-R5 [40]**

The debug registers in ARM Cortex-R5 occupy a 4KB region of the memory map, and their base address must be aligned to a 4KB boundary in physical memory. To access the debug memory map of the Cortex-R5, one must use the memory-mapped registers accessible through the Advanced Peripheral Bus (APB) slave interface. The debug memory-mapped registers are detailed in table 3.1. These registers include various control and status registers essential for debugging tasks, such as the Debug Identification Register (DBGDIDR), Breakpoint Value Registers (DBGBVR), Breakpoint Control Registers (DBGBCR), Watchpoint Value Registers (DBGWVR), and Watchpoint Control Registers (DBGWCR) [40].

The Cortex-R5 processor contains an EmbeddedICE logic unit that supports up to eight breakpoints and eight watchpoints. The exact number of breakpoints and watchpoints available is configured during the implementation of the processor and can be verified by reading the Debug ID Register (DBGDIDR). These breakpoints and watchpoints can be programmed through their respective value and control registers.

3.3.2 Hardware Breakpoints

Hardware breakpoints are integrated into the chipset being used. At the silicon level, these breakpoints function as comparators that evaluate the instruction being fetched against an instruction configured in a peripheral register. When a match is detected, the hardware triggers a debug event, either halting the core or generating an exception. There is a limited number of hardware breakpoints available on any given chip.

For ARM Cortex-R5, hardware breakpoints are achieved by setting the values on DBGBVR and DBGBCR as shown in table 2.1 and figure 3.5.

Based on the Reference Manual of R5 [40] each Breakpoint Value Register (DBGBVR) is associated with a corresponding Breakpoint Control Register (DBGBCR) 3.5. Specifically, DBGBCRy is the control register for DBGBVRy. Together, a Breakpoint Value Register and its associated Control Register form a Breakpoint Register Pair (BRP), such as DBGBVR0 paired with DBGBCR0 to create BRP0, up to DBGBVR7 and DBGBCR7 forming BRP7. The value in a DBGBVR can correspond to an instruction address or a context ID, allowing breakpoints to be set based on an instruction address, a context ID value, or

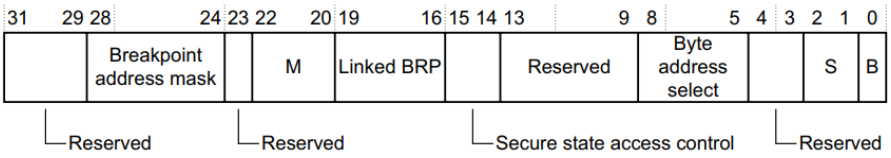

Figure 3.5: **DBGBCR Registers bit assignments [40]**

a combination of both. When breakpoints are set using an instruction address and context ID pair, two BRPs must be linked. A debug event is triggered only when both the instruction address and the context ID match simultaneously.

For ARM Cortex-M MCUs, hardware breakpoint functionality is exposed via the Flash Patch and Breakpoint Unit (FPB) and is explained in section 3.8

3.3.3 Software Breakpoints

Software breakpoints, in contrast, are managed by the debugger itself. They operate by modifying the code to be executed, replacing it with an instruction that triggers a debug event. This is often achieved by inserting a breakpoint instruction or, if not supported, by injecting an instruction that causes a fault to halt the core [66]. Software breakpoints can theoretically provide an unlimited number of breakpoints. However, they come with their own set of challenges:

- Not all code regions are patchable, such as Read-Only Memory (ROM).
- If the debugger crashes, the code may be left in an altered state, with the patched instruction remaining instead of the original code.

To apply a software breakpoint in code using assembly, you can insert a breakpoint instruction (bkpt) directly into the program.

1. **Inserting the Breakpoint:** In the application code, insert the software breakpoint using the assembly instruction asm("bkpt #0");. This instruction is used to trigger a breakpoint exception when the processor encounters it during execution.
2. **Execution of Breakpoint Instruction:** When the processor executes the bkpt instruction, it generates a breakpoint exception. This instruction causes the processor to halt normal execution and transfer control to the exception handler.
3. **Triggering the Exception Handler:** The processor looks up the address of the breakpoint exception handler in the vector table and jumps to this address. The exception handler is a predefined function that gets executed in response to the breakpoint exception.

3.3.4 Watchpoints

Watchpoints are embedded within the chipset used for debugging purposes. At the silicon level, these watchpoints operate as comparators that monitor specific

memory addresses against values configured in dedicated peripheral registers. When a match is detected, indicating that a particular memory address has been accessed or modified as specified, the hardware triggers a debug event, either halting the core or generating an exception. Similar to hardware breakpoints, there is usually a limited number of hardware watchpoints available on any given chip, which can lead to constraints when setting multiple watchpoints during a debugging session.

For ARM Cortex-R5, hardware watchpoints are achieved by setting the values on DBGBVR and DBGBCR as shown in table 2.1 and figure 3.6.

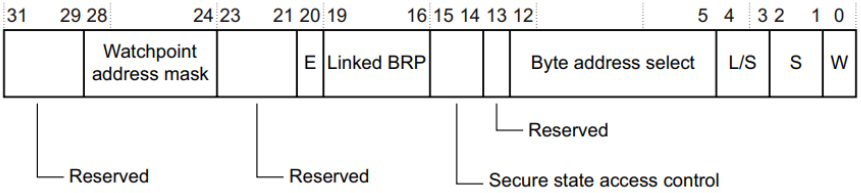

Figure 3.6: **DBGWCR Registers bit assignments [40]**

Based on the Reference Manual of R5 [40], each Watchpoint Value Register (DBGWVR) is associated with a corresponding Watchpoint Control Register (DBGWCR), with DBGWCRy being the control register for DBGWVRy. Together, a DBGWVR and its associated DBGWCR form a Watchpoint Register Pair (WRP), such as DBGWVR0 paired with DBGWCR0 to create WRP0, up to DBGWVR7 and DBGWCR7 forming WRP7. The value contained in a DBGWVR corresponds to a data address and can be configured to trigger on either a data address alone or a data address paired with a context ID. When setting a watchpoint on a data address and context ID pair, the WRP must be linked with a Breakpoint Register Pair (BRP) that has context ID comparison capability. A debug event is triggered when both the data address and the context ID match simultaneously, enabling precise monitoring and debugging of memory accesses in the system.

For ARM Cortex-M MCUs, watchpoint functionality is exposed via the Data and Watchpoint Trace Unit (DWT) and is explained in section 3.7

3.4 Inter Processor Communication(IPC)

Figure 3.7 illustrates the inter-processor communication (IPC) mechanism using mailboxes between two cores, Core 1 and Core 2 [58]. Each core can send and receive messages through dedicated hardware registers, known as mailboxes, which are mapped to shared memory. Each core runs applications that send and receive messages through a local queue (Rpmsg as referred in 3.7). The process begins with Core 1's application sending a message via Rpmsg_send(). This message is written to a virtual ring buffer (VRing-0) and stored in the mailbox associated with Core 1. The mailbox then triggers an interrupt in Core 2, indicating that a message has been received [51]. Core 2's application retrieves the message from VRing-0 via Rpmsg_recv(), processes it, and sends a response through its local queue. This response is written to VRing-1 and

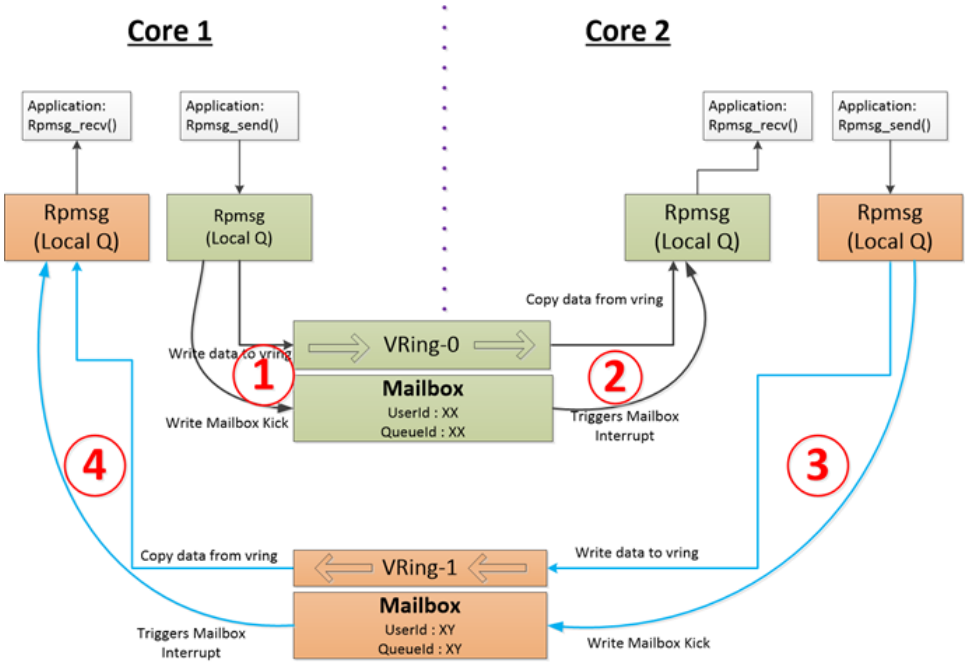

Figure 3.7: **Inter Processor Communication Flow using Mailbox Mechanism provided by Texas Instruments**[58]

stored in the mailbox for Core 2, which then triggers an interrupt in Core 1. Core 1 reads the response from VRing-1, completing the communication cycle. This IPC mechanism is crucial for efficient data exchange between processors in a multi-core system, ensuring synchronization and message integrity [51].

The IPC mechanism depicted in figure 3.7 is based on the Texas Instruments (TI) Software Development Kit (SDK). The hardware kit used to test the embedded debug library is also from TI, which includes drivers for this IPC mechanism. Although the library is tailored for TI's mailbox mechanism, it is designed to be adaptable with minor modifications for any System-on-Chip (SoC) that employs a similar mailbox-based IPC mechanism. This flexibility ensures the library's applicability across various platforms, enhancing its utility in diverse embedded system environments.

3.5 Synchronous Breakpoint

In embedded systems, particularly those involving multicore processors, achieving synchronized debugging can be crucial for identifying and resolving complex issues that span multiple cores. Synchronous breakpoints are a sophisticated debugging technique designed to ensure that all relevant processor cores halt execution simultaneously when a breakpoint is triggered on any one of them. This coordination is achieved by integrating the functionalities of both breakpoint handling and inter-processor communication (IPC).

Multicore processors often run parallel tasks that interact with each other. These interactions can lead to intricate bugs, such as race conditions and dead-

locks, which are challenging to diagnose if only one core is halted. Synchronous breakpoints ensure that when a breakpoint is hit on one core, all other relevant cores are also halted simultaneously. This coordinated halting allows developers to inspect the state of the entire system at the same point in time, providing a comprehensive view of the interactions and helping to pinpoint the root cause of complex bugs. When debugging multicore systems, capturing a consistent system state snapshot is vital. If only one core is halted, other cores might continue executing, potentially altering the system state and making it difficult to reproduce and analyze bugs. Synchronous breakpoints freeze all specified cores at the "same" moment, preserving the state of the system. This consistency is crucial for accurate debugging and for understanding how different parts of the system interact and affect each other.

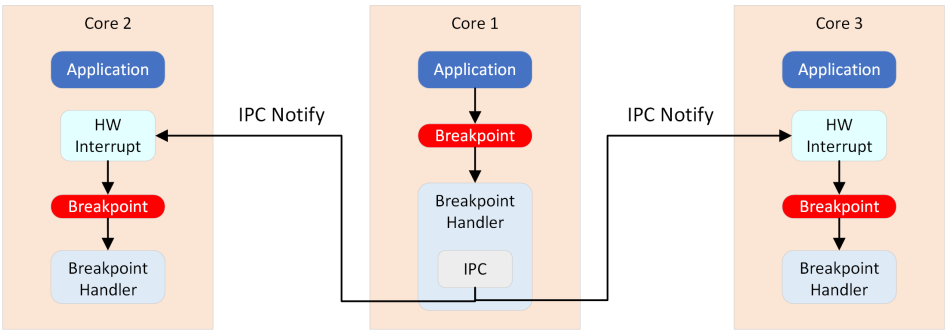

Figure 3.8: **Synchronous Breakpoint Architecture using Inter Processor Communication IPC Notify API**

The process begins when a breakpoint is triggered on one of the processor cores, in this example referred to as Core1. This breakpoint could be either a hardware or software breakpoint, configured to halt execution at a specific point in the application code. When the application on Core1 hits this breakpoint, the Core2 breakpoint handler will be invoked. The primary role of this handler is to notify the other processor cores about the breakpoint event.

To accomplish this notification, the breakpoint handler on Core1 uses the IPC Notify API, a lightweight and low-latency communication mechanism. The handler sends an IPC Notify message through the mailbox mechanism to the other cores in the system, such as Core2 and Core3. This notification includes a message ID that indicates the occurrence of a breakpoint event.

Upon receiving the IPC Notify message, each of the other cores' mailboxes triggers an interrupt, which invokes their respective breakpoint handlers. These handlers then execute the same breakpoint routine, causing Core2 and Core3 to halt execution at their corresponding points in the code. This synchronized halting ensures that all cores stop consistently, which is essential for accurately analyzing the system's behavior and diagnosing issues involving interactions between the cores.

3.6 Core Dump

A core dump is a crucial feature in embedded software debugging that involves capturing the memory and register state of a microcontroller at a specific point in time, typically when a system crash or fault occurs. This snapshot includes the contents of the program counter, stack pointer, general-purpose registers, and relevant portions of memory, providing developers with invaluable insights into the system's state at the moment of failure. Analyzing a core dump allows for post-mortem debugging, helping identify the root cause of the crash and facilitating the development of more robust and reliable software.

The current scope of the library for Core Dump functionality is limited to Cortex-M cores. This limitation is due to the extensive number of registers in the Cortex-R5 core, including co-processors and memory-mapped registers, which require significant time and effort to implement and thoroughly test, which wasn't there due to deadlines. Consequently, the Core Dump functionality for the R5 core has not been included in this implementation but can be implemented by dumping the necessary registers for Cortex-R5.

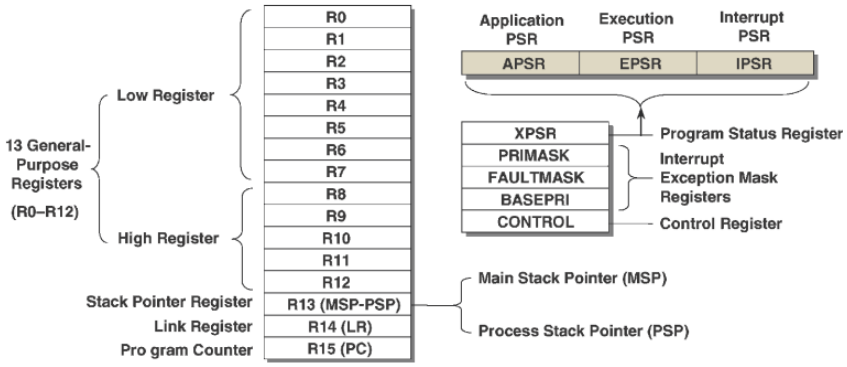

Figure 3.9: **A Block Diagram of 21 registers in the Cortex-M4 Core [11]**

Figure 3.9 illustrates an ARM Cortex-M microcontroller's critical registers and status registers, which are essential for performing a core dump. In the context of embedded software debugging, a core dump involves capturing the state of the system at the moment of a fault or crash. This image highlights the 13 general-purpose registers (R0-R12), the stack pointer register (R13), the link register (R14), and the program counter (R15). Additionally, it shows the program status registers (APSR, EPSR, IPSR) and other control registers such as XPSR, PRIMASK, FAULTMASK, BASEPRI, and CONTROL [11, 41].

When a fault occurs, the values stored in these registers provide a snapshot of the system's state, allowing developers to understand the exact conditions that led to the fault. The stack pointer (R13) indicates the current stack location, while the link register (R14) holds the return address for function calls. The program counter (R15) points to the next instruction to be executed. The program status registers (PSR) indicate the current state of the processor, including application, execution, and interrupt states. Together, these registers

provide comprehensive information necessary for post-mortem analysis, facilitating effective debugging and resolution of issues in the embedded software.

3.7 Data Watchpoint and Trace Unit (DWT)

The Data Watchpoint and Trace (DWT) unit in Cortex-M series microcontrollers provides essential functionality for debugging, particularly when applying watchpoints. The DWT unit is a part of the CoreSight debug architecture and offers hardware support for monitoring data accesses without significantly impacting system performance.

When a watchpoint is set using the DWT, the unit leverages hardware comparators to monitor memory addresses during data operations continuously. Specifically, the DWT contains several watchpoint comparators that can be configured to trigger on specific memory addresses for read, write, or read/write accesses. Each comparator can be programmed with an address value and an access type. As the processor executes instructions, any data access is compared in real-time against these configured values.

When the address of a data access matches the value set in a DWT comparator, the comparator generates a debug event. This event can halt the processor, allowing the debugger to take control and inspect the system's state when the watchpoint condition is met. This capability is crucial for identifying issues such as unintended modifications to critical variables, stack overflows, or unauthorized reads from sensitive memory regions.

The hardware implementation ensures that the comparison operation occurs in parallel with the normal execution flow, introducing no additional execution overhead unless a watchpoint condition is met. This efficiency is vital in real-time systems where maintaining performance is critical.

The primary registers involved in setting up a watchpoint are the DWT Comparator Registers (DWT_COMPn), the DWT Mask Registers (DWT_MASKn), and the DWT Function Registers (DWT_FUNCTIONn). Each comparator register (DWT_COMPn) holds the address value that will be monitored for data access. The mask register (DWT_MASKn) associated with each comparator specifies the address range to be monitored, allowing for flexibility in the size of the monitored region. This is particularly useful for setting watchpoints on entire data structures or memory blocks.

The DWT_CTRL 3.10 register tells how many hardware watchpoints are supported. Notably,

- **NUMCOMP:** The number of watchpoint comparators implemented or 0 in the case the DWT is not included in the hardware.

The primary registers involved in setting up a watchpoint are the DWT Comparator Registers (DWT_COMPn), the DWT Mask Registers (DWT_MASKn), and the DWT Function Registers (DWT_FUNCTIONn) 3.11. Each comparator register (DWT_COMPn) holds the address value that will be monitored for data access. The mask register (DWT_MASKn) associated with each comparator specifies the address range to be monitored, allowing for flexibility in the size of the monitored region. This is particularly useful for setting watchpoints on entire data structures or memory blocks [39].

The DWT_CTRL register bit assignments are:

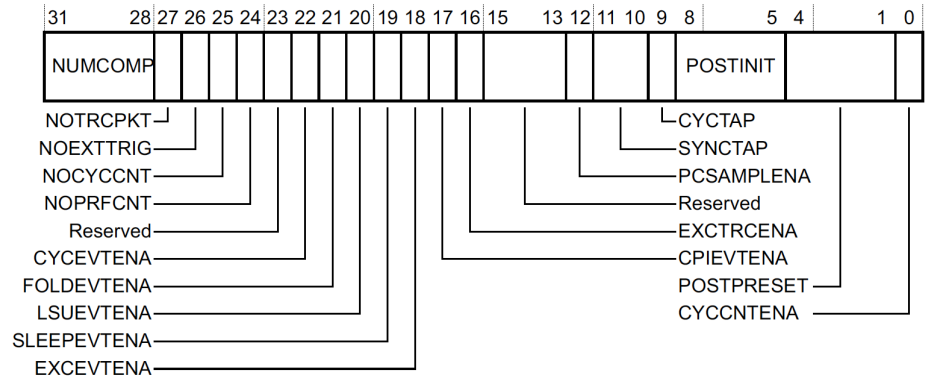

Figure 3.10: **DWT Control Register, DWT_CTRL, 0xE0001000 [39]**

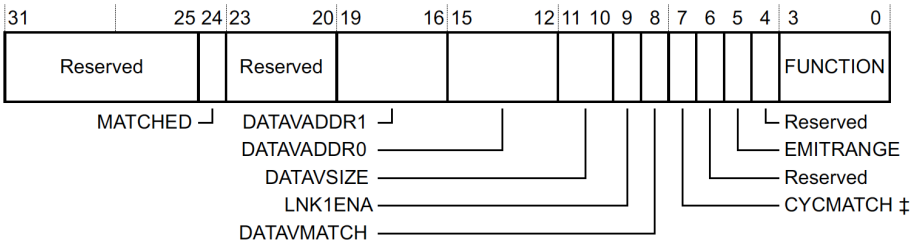

Figure 3.11: **Comparator Function Register DWT_FUNCTIONn, 0xE0001028 + 16n**

The function register (DWT_FUNCTIONn) determines the type of access that will trigger the watchpoint. This register can be configured to detect read accesses, write accesses, or both, providing detailed control over the conditions that generate debug events. For example, setting the DWT_FUNCTIONn register to monitor write operations enables the detection of any modifications to the specified memory address.

3.8 Flash Patch and Breakpoint Unit (FPB)

The Flash Patch and Breakpoint (FPB) unit in Cortex-M series microcontrollers is a critical component for debugging, specifically designed to facilitate the setting of breakpoints in both flash and RAM. The FPB unit provides hardware support for up to six instruction comparators and two literal comparators, allowing developers to set hardware breakpoints efficiently without altering the normal execution flow of the program.

When a breakpoint is set using the FPB unit, the process typically involves configuring the FPB control and comparator registers. Each instruction comparator in the FPB unit can be programmed with an address in the flash memory where the breakpoint should occur. When the processor fetches an instruction from this address, the FPB unit compares the address with the values stored in its comparators. If a match is found, the FPB unit generates a breakpoint exception, halting the processor and transferring control to the debugger. This hardware-based approach ensures that breakpoints can be set and managed with minimal overhead and maximum reliability.

The FPB unit also supports literal comparators, which are used to monitor access to specific data values in the memory. This is particularly useful for debugging read operations on constants and other literal values stored in the flash memory. By using comparators, the FPB unit can trigger breakpoints on data reads, providing a powerful tool for monitoring data access patterns.

The configuration of the FPB unit involves several key registers. The FP_CTRL register controls the overall operation of the FPB unit, enabling or disabling its functionality. The FP_REMAP register allows the remapping of flash addresses to RAM, facilitating the redirection of instruction execution for debugging purposes. Each comparator is configured through FP_COMP registers, which hold the addresses to be monitored and control bits that specify the type of comparison (instruction or literal).

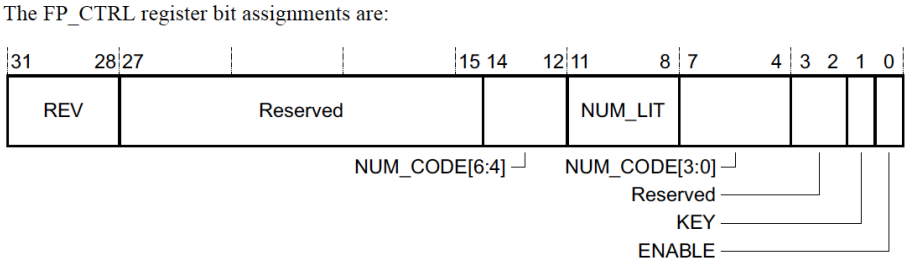

Figure 3.12: **Flash Patch Control Register, FP_CTRL, 0xE0002000 [39]**

The FP_CTRL 3.12 tells how many hardware breakpoints are supported and enables the FPB.

3.9 Hadware support for Profiling

In the context of the embedded software debugging library discussed in this thesis, profiling involves the systematic measurement and analysis of various performance metrics to understand the behavior and efficiency of the firmware.

For the ARM Cortex-M series, profiling can be achieved using the Data Watchpoint and Trace (DWT). The DWT unit provides hardware support for performance monitoring by counting CPU cycles, capturing data watchpoints, and generating events based on specific conditions. These events can include memory accesses, exceptions, and sleep cycles, which are invaluable for gaining insights into the system's performance.

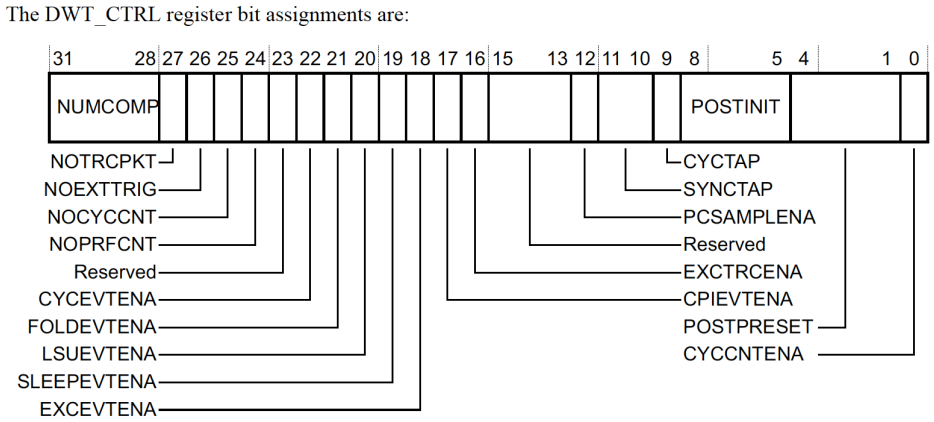

Figure 3.13: **DWT Control Register, DWT_CTRL, 0xE0001000 [39]**

The DWT_CTRL 3.13 register is a 32-bit register that controls various features of the DWT unit. For profiling purposes, the most relevant bits are:

- **CYCCNTENA (Bit 0):** This bit enables the cycle counter. When this bit is set to 1, the cycle counter starts counting the number of cycles the CPU has executed. This is essential for measuring the performance of code segments.
- **EXCEVTENA (Bit 18):** Enables events for the cycle count when exceptions occur. This bit is useful if you want to profile the impact of exceptions on performance.
- **SLEEPEVTENA (Bit 19):** Enables events for the cycle count when the CPU enters sleep mode. This is not required for simple cycle counting but can be useful for power profiling.
- **CYCEVTENA (Bit 22):** Enables events for the cycle counter overflow. This can be useful to detect when the cycle counter overflows, especially in long-running profiling sessions.

3.10 Performance Measurement Unit (PMU)

The Performance Measurement Unit (PMU) in the ARM Cortex-R5 processor is a sophisticated hardware feature designed to facilitate the monitoring and analysis of system performance. The PMU provides a set of counters and control registers that allow developers to track various performance metrics, such as the number of executed instructions, cache hits and misses, and branch predictions. This capability is crucial for optimizing system performance, as it enables detailed profiling and identification of performance bottlenecks.

The performance monitoring registers are as follows:

- Performance Monitor Control Register, PMCR
- Count Enable Set Register, PMCNTENSET
- Count Enable Clear Register, PMCNTENCLR
- Overflow Flag Status Register, PMOVSR
- Software Increment Register, PMSWINC
- Performance Counter Selection Register, PMSELR
- Cycle Count Register, PMCCNTR
- Event Type Selection Register, PMEVTYPERx
- Event Count Registers, PMEVCNTRx
- User Enable Register, PMUSERENR
- Interrupt Enable Set Register, PMINTENSET
- Interrupt Enable Clear Register, PMINTENCLR

The Performance Measurement Unit (PMU) in the ARM Cortex-R5 processor includes three event counting registers, one cycle counting register, and 12 CP15 registers for controlling and interrogating the counters. These performance monitoring registers are accessible in Privileged mode, with the User Enable (PMUSERENR) register allowing access to all but the Interrupt Enable Set (PMINTENSET) and Interrupt Enable Clear (PMINTENCLR) registers in User mode. The three event counters are read and written through the same CP15 register, with the Performance Counter Selection (PMSELR) register determining which counter is accessed. Each counter has its Event Selection register, also accessed via a single CP15 register. Control registers allow enabling or disabling individual event counters, and reading or resetting their overflow flags. Counters can assert an interrupt request output, nPMUIRQm, on overflow. In the Debug halt state, the PMU does not count events, events are not visible on the ETM interface, and the Cycle Count (PMCCNTR) register is halted. The PMU counts events only when non-invasive debug is enabled (with DBGENm or NIDENm asserted), except the PMCCNTR register, which is always enabled unless the DP bit of the PMCR register is set [40].

The Count Enable Set Register(PMCNTENSET) Enables the Event Count Registers. To access the PMCNTENSET Register, read or write CP15 with:

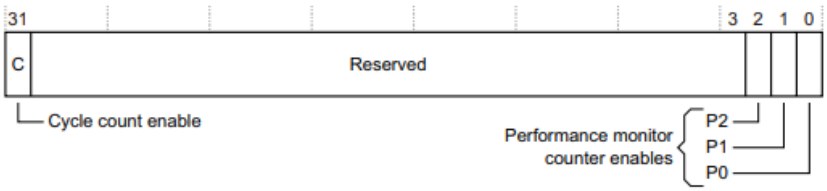

Figure 3.14: **PMU Count Enable Set Register [40]**

- `MRC p15, 0, <Rd>, c9, c12, 1` ; Read PMCNTENSET Register
- `MCR p15, 0, <Rd>, c9, c12, 1` ; Write PMCNTENSET Register

Similarly, other PMU registers can be read and written to.

3.11 Embedded Trace Buffer (ETB)

CoreSight provides features that allow for continuous collection of system information for later off-line analysis. Execution trace macrocells (ETM) exist for use with processors, software can be instrumented with dedicated trace generation, and some peripherals can generate performance monitoring trace streams.

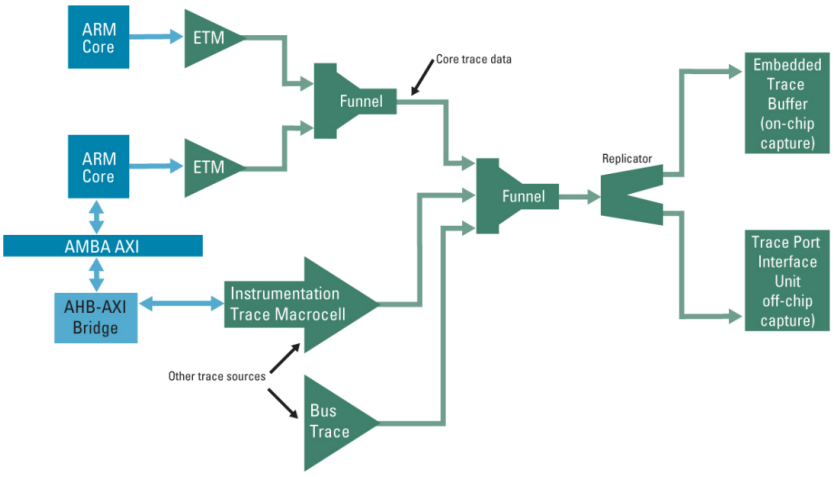

Figure 3.15: **Multiple Trace Sources Combined with a CoreSight Trace Funnel and sending data to ETB [9]**

The Embedded Trace Buffer (ETB) is a vital on-chip debugging and tracing component in many advanced ARM microcontrollers. The ETB captures and stores trace data generated by the Embedded Trace Macrocell or other trace sources, recording detailed information about the program execution, including instruction addresses and data accesses. By analyzing this data, developers can

identify performance bottlenecks, trace faults, and debug complex issues that might be difficult to reproduce in a non-embedded environment. One of the key advantages of the ETB is that it allows for continuous tracing without the need for large external trace memory, as the buffer size is fixed and on-chip.

When the ETB reaches its capacity, it typically overwrites the oldest data, maintaining a circular buffer of the most recent execution history [9]. This approach ensures that the most relevant and recent execution information is always available, making it an efficient solution for ongoing performance monitoring. The system bridge mode supports interrupt and event notification capabilities that support integration with device-level CPUs and/or DMAs to support a variety of use cases, including, e.g., relocation of trace data to DDR and off-chip export over USB [30].

Offset	Name	Type	Reset	Description
0x004	RDP	RO	0x00000000	*ETB RAM Depth Register*
0x00C	STS	RO	0x00000008	*ETB Status Register*
0x010	RRD	RO	0x00000000	*ETB RAM Read Data Register*
0x014	RRP	RW	0x00000000	*ETB RAM Read Pointer Register*
0x018	RWP	RW	0x00000000	*ETB RAM Write Pointer Register*
0x01C	TRG	RW	0x00000000	*ETB Trigger Counter Register*
0x020	CTL	RW	0x00000000	*ETB Control Register*
0x024	RWD	WO	0x00000000	*ETB RAM Write Data Register*

Table 3.2: **ETB registers in offset order from the Base Address [9]**

The Embedded Trace Buffer (ETB) in ARM microcontrollers is managed through a series of specialized registers that control its operation and provide access to the stored trace data [9]. These registers include the ETB Control Register (ETB_CTL), the ETB Status Register (ETB_STATUS), the ETB Data Register (ETB_DATA), the ETB RAM Write Pointer Register (ETB_RWP), the ETB RAM Read Pointer Register (ETB_RRP), and the ETB RAM Depth Register (ETB_RDP). The ETB Control Register (ETB_CTL) is used to enable or disable the ETB and to configure various operational parameters. The ETB Status Register (ETB_STATUS) provides status information, such as whether the buffer is full or empty, and whether the ETB is currently active.

The ETB Data Register (ETB_DATA) is used to read the trace data stored in the buffer. This register provides sequential access to the data captured during program execution, allowing developers to reconstruct the execution flow. The RAM Write Pointer Register (ETB_RWP) indicates the current write position in the ETB, effectively pointing to where the next trace entry will be stored. Conversely, the RAM Read Pointer Register (ETB_RRP) indicates the current read position, showing which trace entry will be read next. Together, these pointer registers facilitate the management of the circular buffer, ensuring that the most recent trace data is accessible.

The RAM Depth Register (ETB_RDP) specifies the total size of the trace buffer, allowing developers to configure the ETB according to the available on-chip memory and the needs of their application. By appropriately setting and reading these registers, developers can control the ETB's behavior, monitor its status, and retrieve trace data efficiently.

3.12 Micro Trace Buffer(MTB)

The Micro Trace Buffer (MTB [46]) provides a lightweight and efficient method for tracing program execution without significantly impacting system performance. Unlike traditional trace systems that rely on external hardware and can be intrusive, the MTB leverages on-chip circular buffer memory to record program flow, making it ideal for resource-constrained environments. The MTB collects information on non-sequential Program Counter changes to a dedicated area of SRAM whereas the ETB collects full Instruction Trace including data memory access along with time stamps.

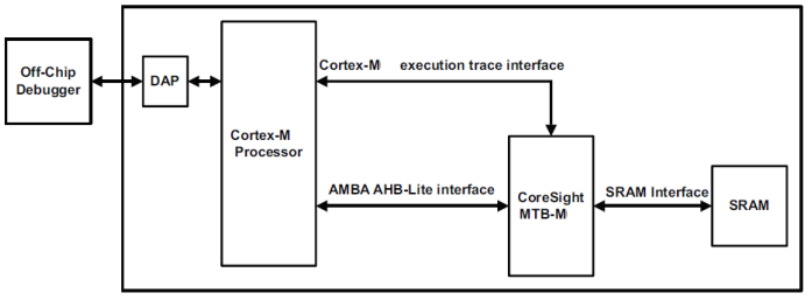

Figure 3.16: **The main interfaces on the MTB [46]**

The MTB works by capturing instruction addresses as the program executes, storing these addresses in a configurable circular buffer in SRAM. When the buffer fills up, it overwrites the oldest data, ensuring continuous logging with minimal memory usage. This data can then be used to reconstruct the program's execution path, allowing developers to analyze the system's behavior, identify performance bottlenecks, and debug complex issues that are hard to reproduce.

The execution trace packet consists of a pair of 32-bit words that the MTB generates when it detects the processor PC value changes non-sequentially(i.e. a branch is taken or an exception occurs). A non-sequential PC change can occur during branch instructions or exception entry.

The first 4 bytes of the packet are where the code was prior to branching (the "source address") and the next 4 bytes are the address which was branched to (the "destination address").

Figure 3.17 shows the MTB execution trace packet format when it's stored in the internal SRAM.

Note: the SRAM here mentioned is the internal SRAM around address 0x2000_0000, so MTB will share the same address range as the processor core. The MTB trace buffer range should not be used by application code to store global variables, heap or stack by changing the linker script and dedicating memory in SRAM for MTB.

ARM Cortex-M33 design may have an optional MTB integrated. Table 3.3 shows the registers that need to be examined and updated for enabling MTB on arm cortex M-33.

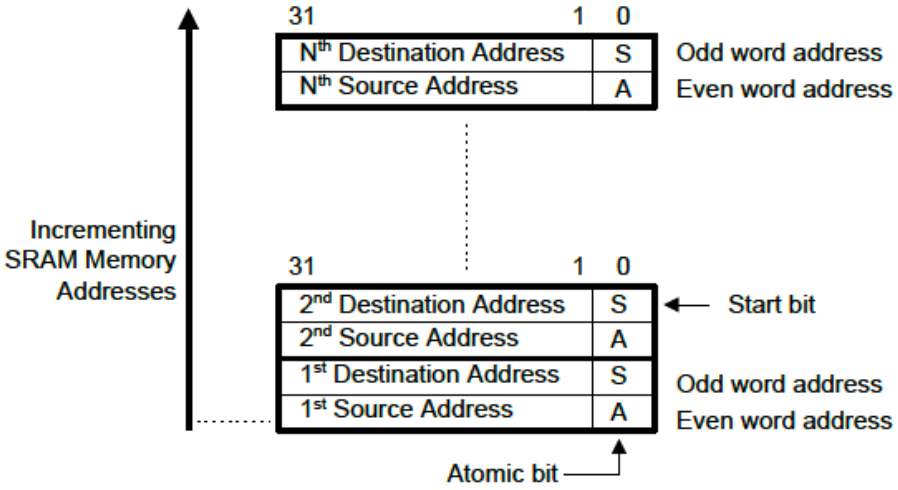

Figure 3.17: **MTB Trace Packet Format [46]**

Address	Name	Description
0xE0043000	MTB_POSITION	Holds current MTB write pointer offset
0xE0043004	MTB_MASTER	Used to enable the MTB and configure its size
0xE0043008	MTB_FLOW	Controls MTB behavior when full (default is to wrap and overwrite)
0xE004300C	MTB_BASE	Read only, reveals SRAM address where MTB packets are stored

Table 3.3: **Registers for MTB implementation on the Cortex-M33 (MTB-M33) [38]**

Chapter 4

Technical Implementation

The Technical Implementation chapter delves into the practical aspects of the embedded software debugging library, focusing on the sequence diagrams, APIs, and their functionalities. This chapter, in continuation to chapter Architecture and System design 3, provides a comprehensive overview of how each component of the library is implemented, offering insights into the intricate mechanisms that enable efficient debugging and performance analysis in ARM Cortex-M and Cortex-R5 series microcontrollers.

The sequence diagrams illustrate the step-by-step execution flow of various debugging operations, such as setting breakpoints, watchpoints, and profiling. Additionally, the chapter details the API functions provided by the debugging library. Each API is thoroughly explained, covering its purpose, usage, and the underlying operations it performs.

4.1 Breakpoint and Watchpoint

This section provides a detailed look into how breakpoints and watchpoints are implemented within the debugging library, offering a granular level of control for debugging tasks.

4.1.1 Hardware Breakpoint

To apply a hardware breakpoint on the Cortex-R5, the following sequence of steps is used, which is illustrated by figure 4.1. First, the breakpoint being set is disabled by writing 0x0 to the debug register. Next, the address is written to the Debug Base Address Value Register (DBGBVR). The byte address select value is then determined based on the instruction set architecture (ISA) being used. If the ISA is Thumb, the byte address select value is set to 3 shifted by the result of the address and 2. If the ISA is ARM, the byte address select value is set to 15. Finally, the mask and control register are written to enable the breakpoint, combining the byte address select value with the control settings.

```
1 void SetSimpleBreakpoint(int break_num, uint32_t address);
```

Listing 4.1: **Setting a Simple Breakpoint on Cortex-R5**

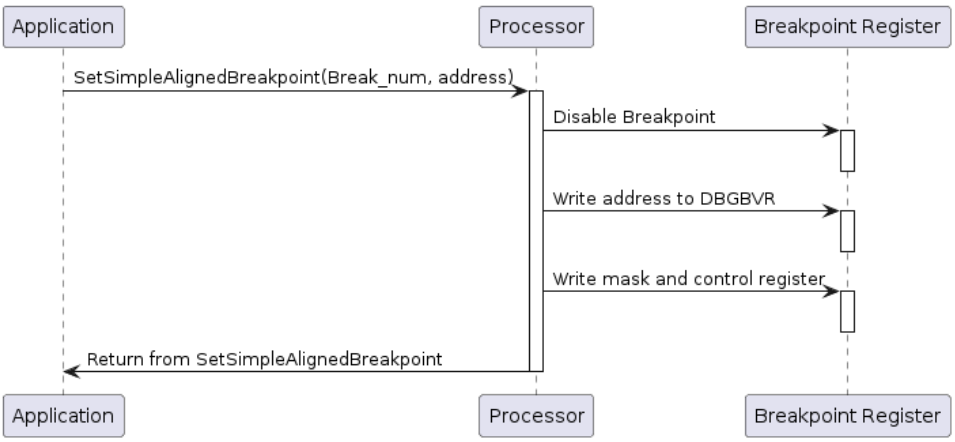

Figure 4.1: **Sequence Diagram for Hardware Breakpoint in R5 Core**

SetSimpleBreakpoint()

This function demonstrates how to set a breakpoint at a specified address

4.1.2 Software Breakpoint

To apply a software breakpoint using assembly, use the following sequence 4.2. This example shows how to set a software breakpoint and define the corresponding exception handler:

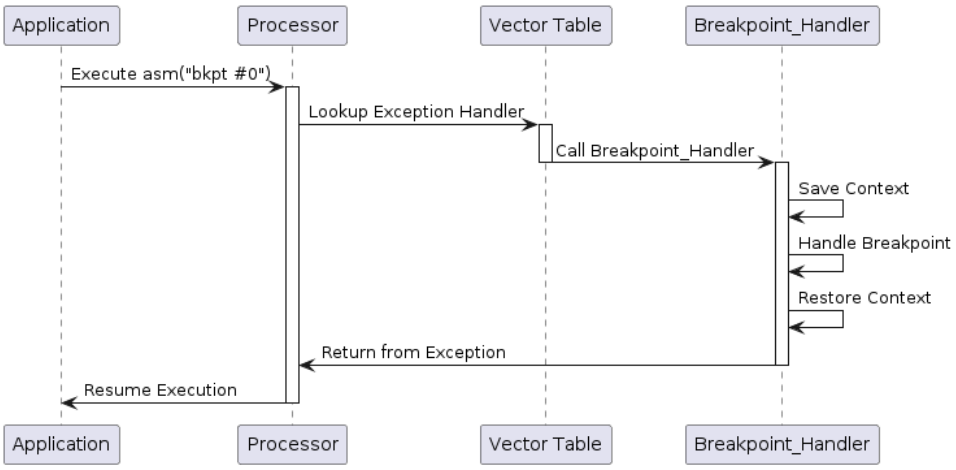

Figure 4.2: **Sequence Diagram for Software Breakpoint in ARM Core**

Exception Handler Code: In the exception handler, you can save the current context (such as register values), perform necessary debugging actions (like logging the state or inspecting variables), and then restore the context. Finally, the handler can return control back to the point where the breakpoint was triggered, allowing the program to continue execution.

4.1.3 Watchpoint

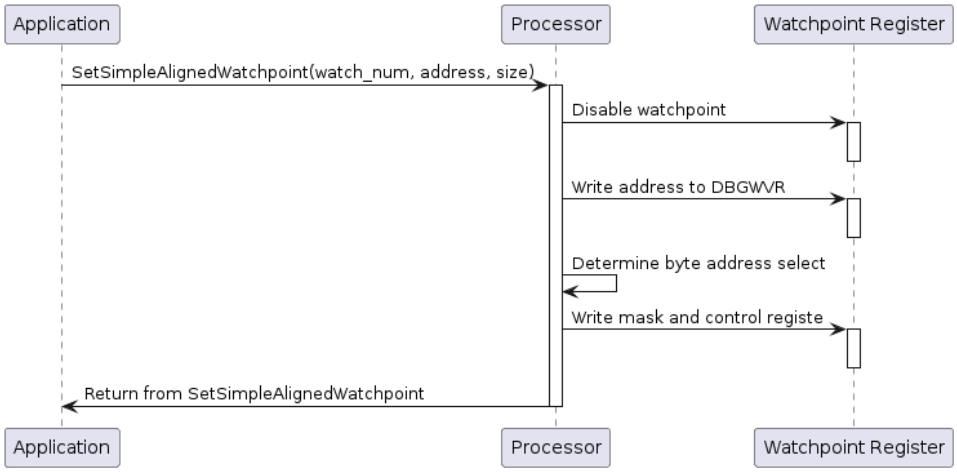

Figure 4.3: **Sequence Diagram for Hardware Watchpoint in R5 Core**

To apply a hardware watchpoint on the Cortex-R5, the function 'SetSimpleAlignedWatchpoint' 4.3 follows a sequence of steps to configure the necessary debug registers. First, the watchpoint being set is disabled by writing '0' to the corresponding Watchpoint Control Register (DBGWCR). Next, the desired address is written to the Watchpoint Value Register (DBGWVR). The byte address select value is then determined based on the size of the watchpoint, using a switch statement to set the appropriate value for 1, 2, 4, or 8 bytes. Finally, the watchpoint is enabled by writing the control bits and the byte address select value to the Watchpoint Control Register (DBGWCR). This configuration allows the hardware watchpoint to monitor specific memory addresses for read/write operations, triggering a debug event when the specified conditions are met, thereby facilitating efficient debugging and memory monitoring.

```
1 void SetSimpleAlignedWatchpoint(int watch_num, uint32_t address,
      int size)
2 }
```

Listing 4.2: **Setting a Simple Aligned Watchpoint on Cortex-R5**

SetSimpleAlignedWatchpoint()

This function demonstrates how to set a watchpoint at a specified address.

4.2 Inter Processor Communication(IPC)

Texas Instruments (TI) provides two primary APIs for inter-processor communication (IPC) between CPUs: IPC RP Message and IPC Notify. The IPC RP Message API allows CPUs to send messages as packet buffers to a logical endpoint on another CPU. These packet buffers reside in shared memory accessible by both CPUs. When a packet is placed in shared memory, the sending

CPU triggers a hardware interrupt to notify the receiving CPU of the new packet. The message packet size varies: with Linux on one end, the size is fixed at 512 bytes, while with RTOS/NORTOS on both ends, the minimum size is 4 bytes and the maximum recommended size is 512 bytes. Larger packets require more shared memory. Logical endpoints can be defined up to RPMESSAGE_MAX_LOCAL_ENDPT count. On the other hand, IPC Notify is a simpler mechanism where a CPU interrupts another CPU using a low-level hardware interrupt, sending a 28-bit message ID. This approach is highly efficient and low-latency but less flexible than RP Message. IPC Notify supports up to IPC_NOTIFY_CLIENT_ID_MAX logical endpoints or client IDs.

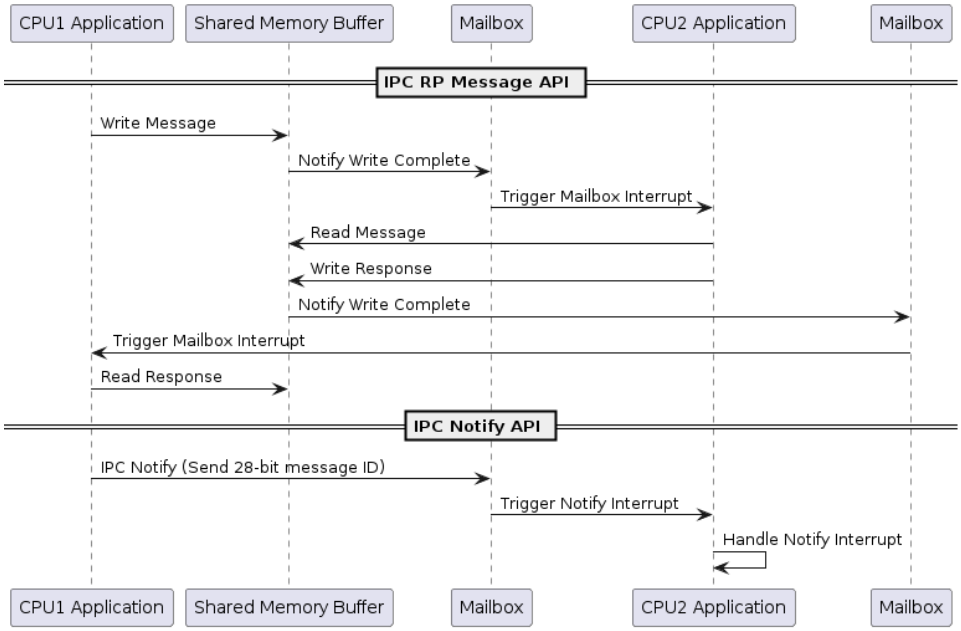

Figure 4.4: **Inter Processor Communication API's Flow and Sequence**

The sequence diagram above 4.4 illustrates the inter-processor communication (IPC) mechanism using both the IPC RP Message API and the IPC Notify API, focusing on shared memory buffers between CPU1 and CPU2:

```
int32_t IpcNotify_init(const IpcNotify_Params * params)
int32_t RPMessage_init (const RPMessage_Params *params)
}
```

Listing 4.3: **API's for IPC**

RPMessage_init()

1. The application on CPU1 writes a message to the shared memory buffer.
2. The shared memory buffer notifies the mailbox on CPU1 that the write operation is complete.
3. The mailbox on CPU1 triggers an interrupt on CPU2, notifying the application on CPU2.

4. The application on CPU2 reads the message from the shared memory buffer.

5. After processing, the application on CPU2 writes a response to the shared memory buffer.

6. The shared memory buffer notifies the mailbox on CPU2 that the write operation is complete.

7. The mailbox on CPU2 triggers an interrupt on CPU1, notifying the application on CPU1.

8. The application on CPU1 reads the response from the shared memory buffer, completing the communication cycle.

IpcNotify_init()

1. The application on CPU1 sends a notification using the IPC Notify API, including a 28-bit message ID.

2. The mailbox on CPU1 triggers a notify interrupt on CPU2.

3. The application on CPU2 handles the notify interrupt, processing the received message ID.

4.3 Synchronous Breakpoint

The synchronous breakpoint mechanism effectively combines the features of the traditional breakpoint handling section and the IPC section. Breakpoints provide the capability to halt execution at precise locations within the application code, while IPC Notify ensures that this halting is communicated across multiple cores efficiently and with minimal delay. By leveraging both these functionalities, synchronous breakpoints enable coordinated debugging, making it easier to identify and resolve issues that could be missed if only one core is halted.

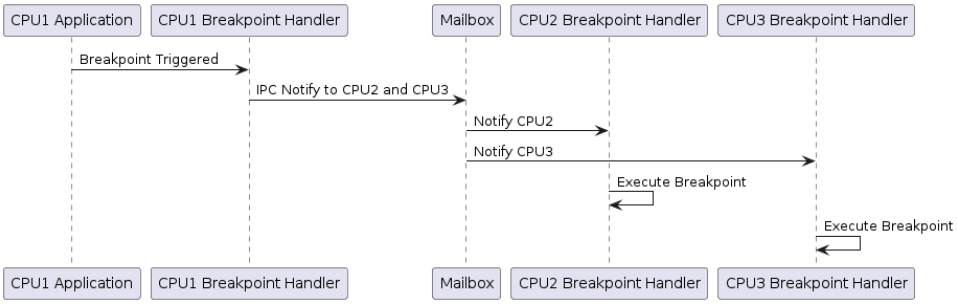

Figure 4.5: **Synchronous Breakpoint Flow and Sequence using Inter Processor Communication API's**

In the context of a synchronous breakpoint mechanism as shown in sequence diagram 4.5, the following sequence of actions occurs:

1. The application on CPU1 triggers a breakpoint, which invokes the breakpoint handler.

2. The breakpoint handler on CPU1 sends an IPC Notify message to the mailbox to notify other cores (CPU2 and CPU3) about the breakpoint.

3. The mailbox forwards the IPC Notify message to the corresponding handlers on CPU2 and CPU3.

4. Upon receiving the IPC Notify message, the breakpoint handlers on CPU2 and CPU3 execute the breakpoint, causing each core to halt and enter their respective breakpoint handlers.

This process ensures that all cores are synchronized and halt execution at the specified breakpoint, allowing for coordinated debugging across multiple cores.

4.4 Core Dump

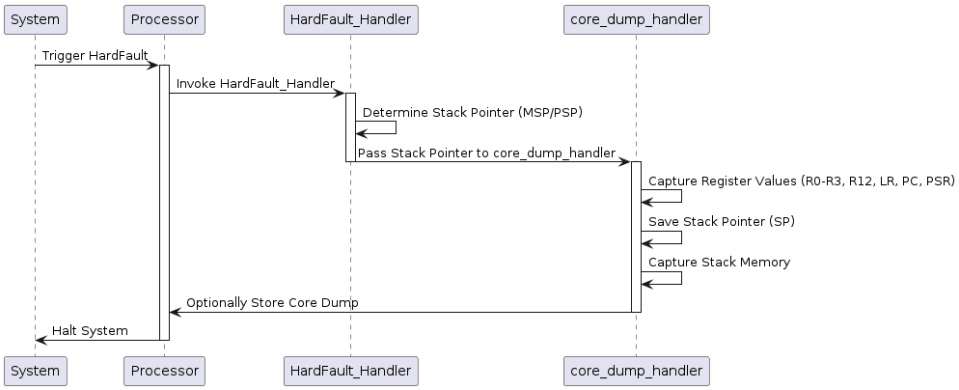

Figure 4.6: **Sequence Diagram for storing Core Dump in ARM Cortex-M**

The sequence diagram 4.6 demonstrates the process of handling a hard fault in an ARM Cortex-M microcontroller by capturing a core dump. This process involves several key steps, which are illustrated in the sequence diagram above.

1. **HardFault Occurrence:** When a hard fault occurs in the system, the processor automatically invokes the HardFault_Handler function.

2. **Determine Stack Pointer:** Within the HardFault_Handler, assembly instructions are executed to determine which stack pointer (Main Stack Pointer (MSP) or Process Stack Pointer (PSP)) was in use at the time of the fault. This is done using the TST and ITE instructions to test the EXC_RETURN value in the Link Register (LR) and select the appropriate stack pointer.

3. **Invoke Core Dump Handler:** The determined stack pointer is then passed to the core_dump_handler function, which captures the core dump.

4. **Capture Register Values:** The core_dump_handler function extracts the values of the general-purpose registers (R0, R1, R2, R3, R12), the Link Register (LR), the Program Counter (PC), and the Program Status Register (PSR) from the stack.

5. **Save Stack Pointer:** The current value of the stack pointer is saved.

6. **Capture Stack Memory:** The contents of the stack are captured and stored in the core_dump structure for further analysis.

7. **Store Core Dump:** Optionally, the core dump data can be stored in an external memory or signaled to a debugger for post-mortem analysis.

8. **Halt System:** Finally, the system is halted to prevent any further execution, allowing for a thorough analysis of the captured core dump data.

4.5 Debug and Watchpoint Trace Unit(DWT)

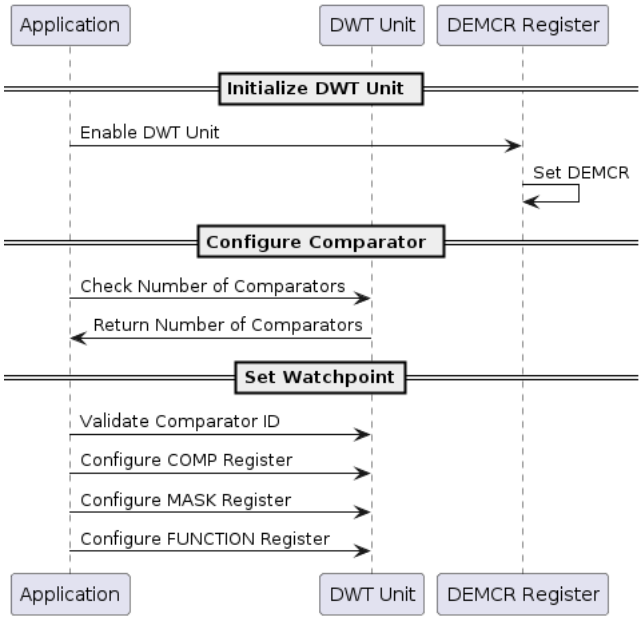

Figure 4.7: **Sequence Diagram for applying a Watchpoint using DWT in ARM Cortex-M**

The sequence diagram 4.7 depicts the process of setting a watchpoint in an ARM Cortex-M series microcontroller using the provided code. The process begins with the application enabling the Data Watchpoint and Trace (DWT) unit by setting the appropriate bit in the Debug Exception and Monitor Control Register (DEMCR). Next, the application checks the number of available comparators in the DWT unit by reading the CTRL register and extracting the number of comparators. After validating the specified comparator ID against

the number of available comparators, the application proceeds to configure the comparator. This involves setting the COMP register with the address to be monitored, configuring the MASK register to specify the address range, and finally setting the FUNCTION register to define the type of data access (read, write, or read/write) that should trigger the watchpoint. The configuration of the FUNCTION register enables the comparator, to complete the process of setting a watchpoint.

```
1 void dwt_dump(void);
2 void dwt_reset(void);
3 void dwt_install_watchpoint(int comp_id, uint32_t func, uint32_t
      comp, uint32_t mask);
```

Listing 4.4: **Data Watchpoint and Trace (DWT) API Functions**

dwt_dump()

This function is used to dump the current configuration of the DWT unit, including all comparators and their settings. It logs the state of the DWT control register and each comparator's function, comparison value, and mask.

dwt_reset()

This function resets all DWT comparators by setting their function, comparison value, and mask registers to zero. This effectively disables all watchpoints.

dwt_install_watchpoint()

This function installs a watchpoint on a specified comparator by configuring its function, comparison value, and mask registers. It first enables the DWT unit by setting the appropriate bit in the DEMCR register.

4.6 Flash Patch and Breakpoint Unit (FPB)

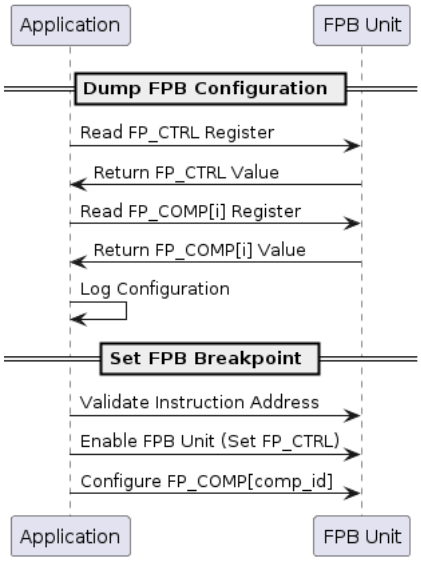

Figure 4.8: **Sequence Diagram for applying a Breakpoint using FPB in ARM Cortex-M**

The sequence diagram 4.8 illustrates the process of configuring a breakpoint using the Flash Patch and Breakpoint (FPB) unit in an ARM Cortex-M microcontroller. Initially, the application interacts with the FPB unit to dump the current breakpoint configuration. This process involves reading the FP_CTRL register to determine the control and status of the FPB unit, including the number of available hardware breakpoints and whether the FPB unit is enabled. The application then iterates through each comparator, reading the FP_COMP registers to retrieve the configuration details such as whether the comparator is enabled and the address it monitors. These details are logged for debugging purposes, providing a comprehensive snapshot of the FPB unit's current state.

Subsequently, the process of setting a new breakpoint is initiated. The application validates the instruction address to ensure it is within the permissible range. It then enables the FPB unit by setting the appropriate bits in the FP_CTRL register. After enabling the FPB unit, the application configures the specified comparator by writing the instruction address and control bits to the FP_COMP register. This configuration enables the comparator, allowing it to monitor the specified address and trigger a breakpoint when the address is accessed during program execution.

```
1 void fpb_dump_breakpoint_config(void);
2 bool fpb_set_breakpoint(size_t comp_id, uint32_t instr_addr);
```

Listing 4.5: **Flash Patch and Breakpoint (FPB) API Functions**

fpb_dump_breakpoint_config()

This function dumps the current configuration of the Flash Patch and Breakpoint (FPB) unit. It logs the state of the FPB control register and all configured comparators, including their enabled status and the addresses they monitor.

fpb_set_breakpoint()

This function configures a breakpoint on a specified comparator by setting its comparison value in the FP_COMP register. It first enables the FPB unit by setting the appropriate bits in the FP_CTRL register.

4.7 Profiling

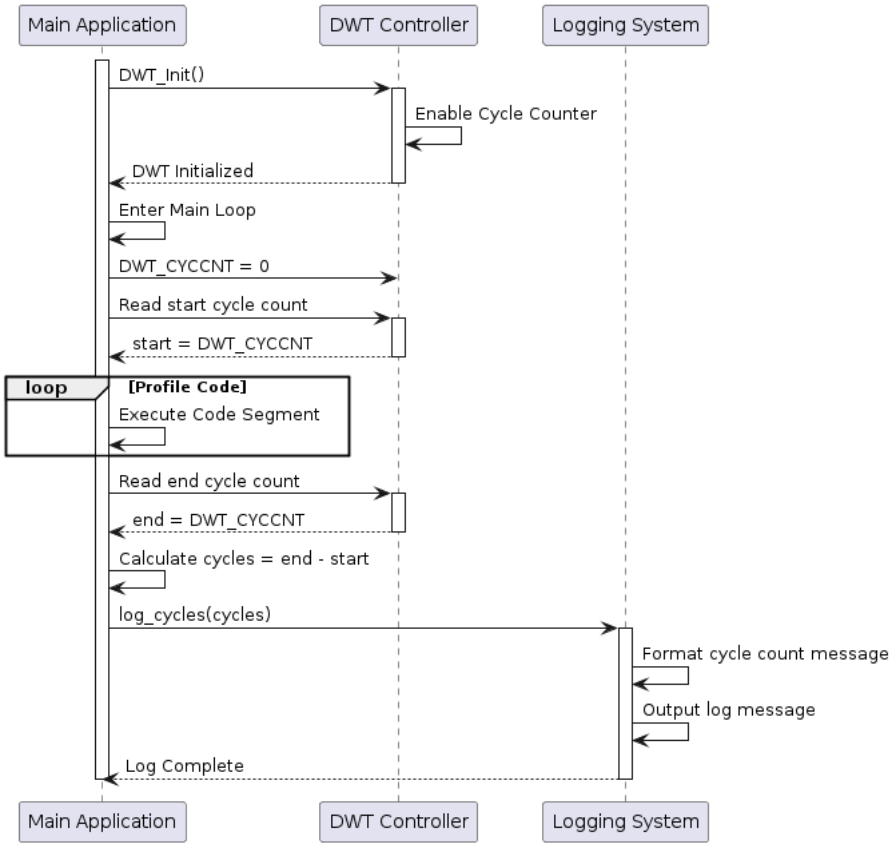

Figure 4.9: **Sequence Diagram for Profiling a Function using Cycle Counter in DWT in ARM Cortex-M**

The sequence diagram 4.9 illustrates the process of profiling code execution on an ARM Cortex-M microcontroller using the DWT (Data Watchpoint and Trace) cycle counter and logging the cycle count. This process begins with the user starting the application, which triggers the DWT_Init() function to

enable the cycle counter by setting the appropriate bits in the DWT control registers. After initialization, the main application enters a loop where it profiles a specific code segment. At the beginning of each iteration, the cycle counter (DWT_CYCCNT) is reset to zero to ensure consistent measurements. The application then reads the start cycle count, executes the code segment to be profiled, and reads the end cycle count upon completion. The elapsed cycles are calculated by subtracting the start count from the end count. This cycle count is logged using a function that formats the message and outputs it, crucial for recording profiling data for later analysis.

```
1 void enable_cycle_counter(void)
2 uint32_t read_cycle_counter(void)
```

Listing 4.6: **Enabling the DWT Cycle Counter**

enable_cycle_counter()

The enable_cycle_counter function enables the cycle counter of the Data Watchpoint and Trace (DWT) unit in the ARM Cortex-M microcontroller. It does this by setting the CYCCNTENA bit in the DWT_CTRL register. This action allows the cycle counter to begin counting the number of CPU cycles, which is fundamental for profiling purposes. By enabling the cycle counter, developers can measure the execution time of specific code segments in terms of CPU cycles.

read_cycle_counter()

The read_cycle_counter function reads and returns the current value of the cycle counter from the DWT_CYCCNT register. This value indicates the number of CPU cycles that have elapsed since the counter was last reset or enabled. By capturing the cycle count at the start and end of a code segment, developers can calculate the total number of cycles consumed by that segment.

4.8 Performance Measurement Unit(PMU)

The sequence diagram 4.10 illustrates the interaction between the main application, the PMU (Performance Measurement Unit) driver, and the PMU hardware in an ARM Cortex-R5 environment, focusing on profiling code execution. This process begins with the user starting the application, which triggers the PMU_init(cfg) function. During initialization, the PMU driver sets up the profile object, configures the PMU registers, enables user mode access, configures event and cycle counters, disables counter overflow interrupts, resets counters, and enables all counters. Once initialized, the application enters a loop that profiles specific code segments.

In each iteration, the profiling starts by calling PMU_profileStart(name), which resets the counters and reads the initial counter values. The main application then executes the code segment to be profiled. After execution, PMU_profile End(name) is called to read the final counter values and calculate the elapsed counts, which are essential for performance analysis. The profiling data is logged by calling PMU_profilePrintEntry(name).

This sequence is repeated for each code segment being profiled, allowing for continuous performance monitoring. When the user stops the application, the

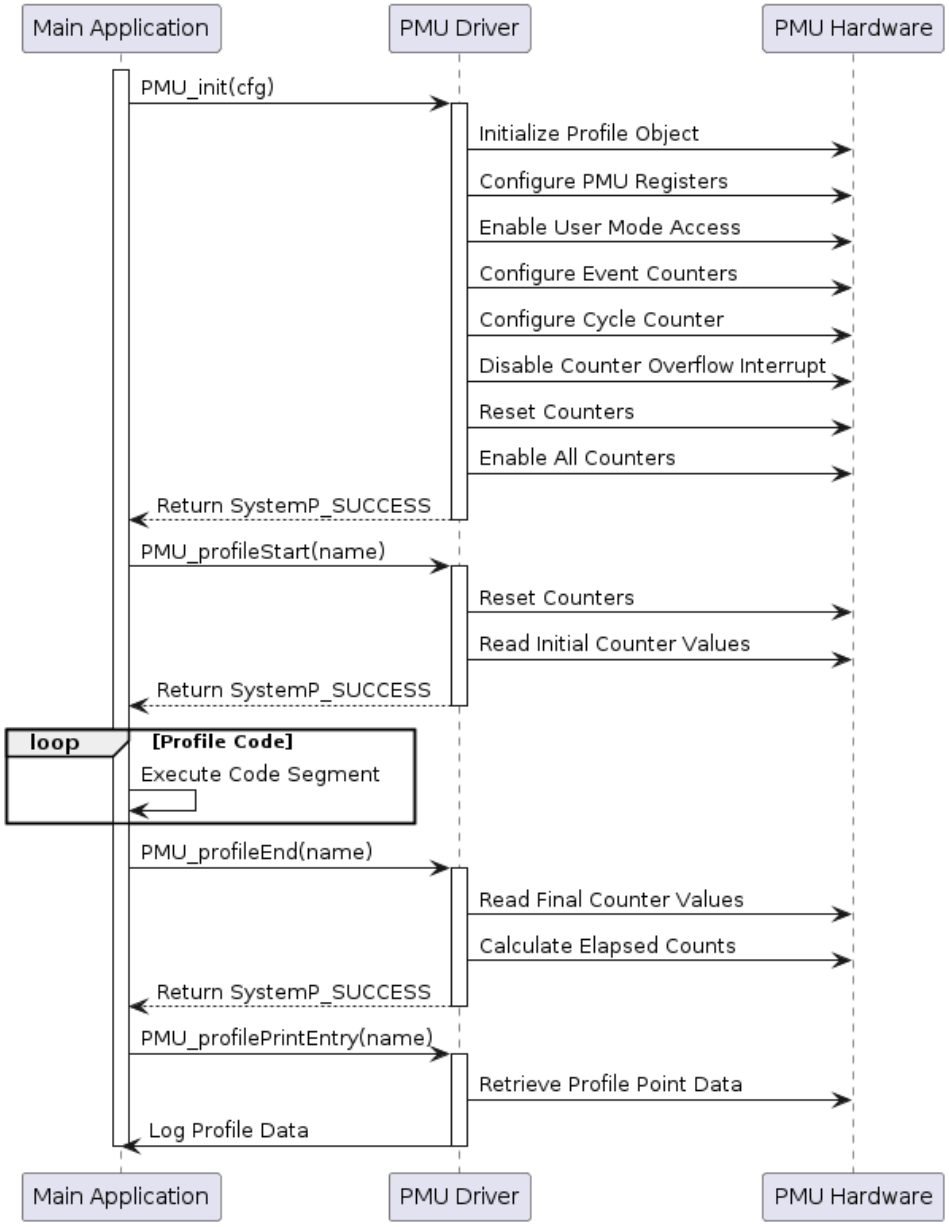

Figure 4.10: **Performance Measurement Unit Sequence Diagram**

main application calls PMU_disableAllCounters(numCounters) to disable all and specific counters.

```
int32_t PMU_init(PMU_Config *cfg)
int32_t PMU_profileStart(const char *name)
int32_t PMU_profileEnd(const char *name)
void PMU_profilePrintEntry(const char *name)
void PMU_profilePrint(void);
```

Listing 4.7: **API's for PMU**

PMU_init()

This function initializes the Performance Measurement Unit (PMU) with the specified configuration settings. It sets up the profile object, configures the PMU registers, enables user mode access, configures the event counters and the cycle counter, disables counter overflow interrupts, resets the counters, and enables all counters. This function ensures that the PMU is properly set up for profiling code execution.

PMU_profileStart()

This function starts profiling for a profile point. It resets the PMU counters and reads the initial counter values. This function is called before executing the code segment to be profiled, marking the beginning of the profiling interval.

PMU_profileEnd()

This function ends profiling for a named profile point. It reads the final counter values and calculates the elapsed counts by subtracting the initial values recorded by PMU_profileStart(). This function is called after executing the code segment to be profiled, marking the end of the profiling interval.

PMU_profilePrintEntry()

This function prints the profiling data for a specific named profile point. It retrieves the profile point data from the profile object and logs the cycle count and event counts. This function allows for detailed analysis of a particular profile point.

PMU_profilePrint()

This function prints all profiling data stored in the profile object. It iterates through all recorded profile points and logs their cycle counts and event counts. This function provides a comprehensive overview of the profiling data collected during the execution of the application.

4.9 Embedded Trace Buffer(ETB)

The sequence diagram depicts 4.11the process of initializing, enabling, using, disabling, retrieving data from, and closing the Embedded Trace Buffer (ETB)

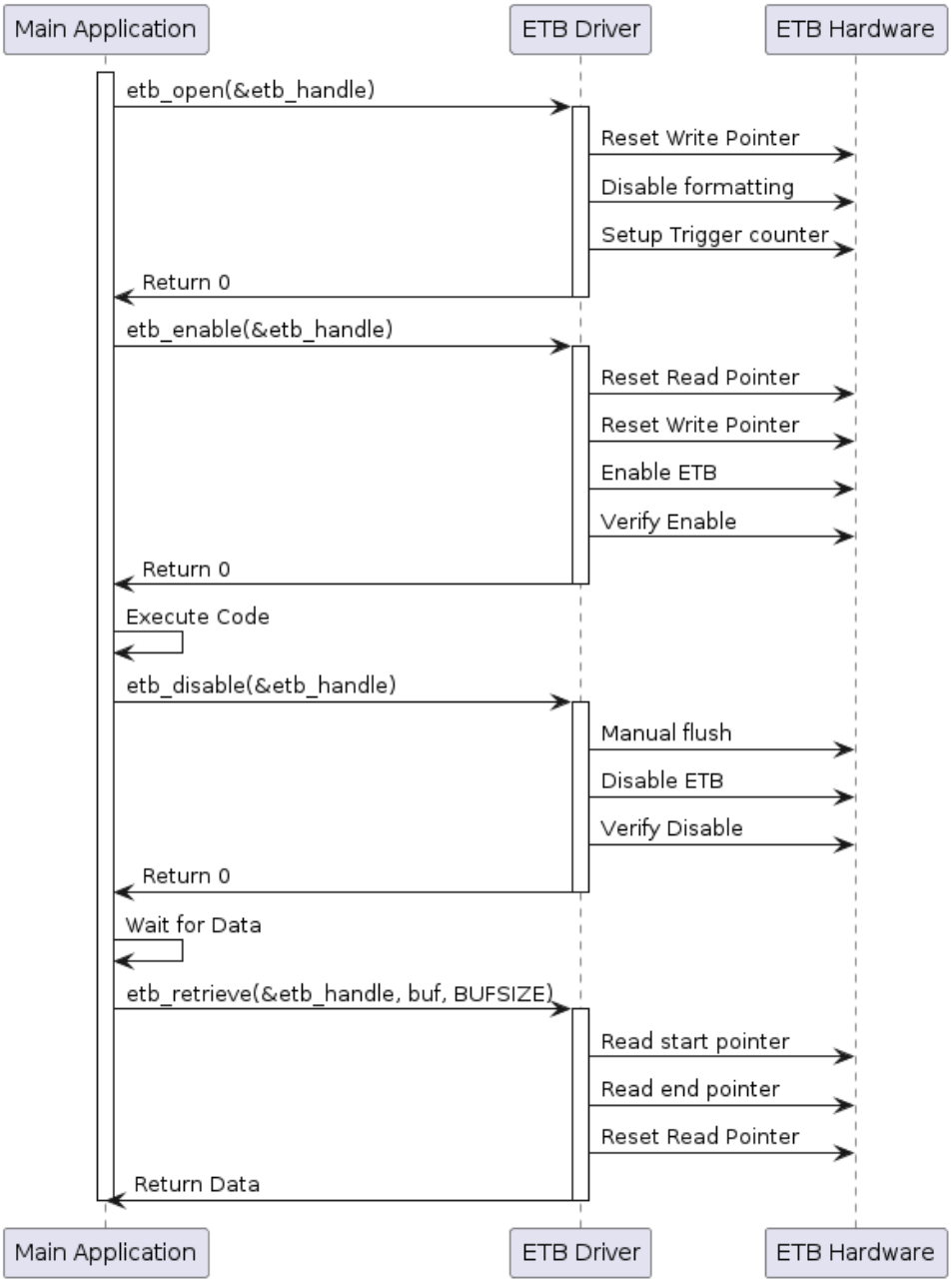

Figure 4.11: **Embedded Trace Buffer Sequence Diagram**

within an embedded system, showcasing the interactions between the main application, the ETB driver, and the ETB hardware. Initially, the user configures the ETB by resetting the write pointer, disabling formatting, and setting up the trigger counter. This sets up the ETB for tracing, and the function.

The ETB driver resets both the read and write pointers and writes to the control register to enable the ETB. It then verifies the ETB's enable status by reading the control register and checking the appropriate bit. Upon successful enabling, the main application proceeds to execute the code, with the ETB capturing trace data during this period.

After code execution, the main application stops the ETB. The ETB driver performs a manual flush, disables the ETB via the control register, and verifies the disable status. The application then enters a loop to retrieve the trace data. This function reads the start and end pointers to determine the available data, resets the read pointer, and transfers the trace data into the provided buffer, returning the size of the retrieved data.

This sequence highlights the comprehensive workflow for managing ETB operations, emphasizing the key function calls and hardware register interactions essential for trace data collection, which is critical for debugging and optimizing embedded systems.

```
1 int etb_open(struct etb_handle_t *etb_handle);
2 void etb_close(struct etb_handle_t *etb_handle);
3 int etb_enable(struct etb_handle_t *etb_handle);
4 int etb_disable(struct etb_handle_t *etb_handle);
5 int etb_status(struct etb_handle_t *etb_handle);
6 ssize_t etb_retrieve(struct etb_handle_t *etb_handle, void *buf,
      size_t bufsize);
```

Listing 4.8: **API's for ETB**

etb_open()

The etb_open function initializes the Embedded Trace Buffer (ETB) and prepares it for operation. This function maps the ETB base address, unlocks the ETB hardware, resets the write pointer (ETB_RWP), disables the formatting (ETB_FFCR), and sets up the trigger counter (ETB_TRIG). Upon successful configuration, the function returns 0. If the initialization fails, it returns an error, which is handled by the application to abort further operations.

etb_close()

The etb_close function cleans up and releases resources associated with the ETB. It unmaps the ETB base address and sets the handle to NULL, effectively closing the ETB. This function ensures that all resources are properly freed and that the ETB is left in a clean state after use.

etb_enable()

The etb_enable function enables the ETB to start capturing trace data. It unlocks the ETB hardware, resets both the read pointer (ETB_RRP) and the write pointer (ETB_RWP), and then writes to the control register (ETB_CTL) to enable the ETB. The function includes a verification step where it reads

back the ETB_CTL register to ensure that the ETB is enabled. If the ETB is successfully enabled, the function returns 0; otherwise, it returns an error.

etb_disable()

The etb_disable function stops the ETB from capturing trace data. It unlocks the ETB hardware, performs a manual flush by setting a bit in the formatting control register (ETB_FFCR), and disables the ETB by writing to the control register (ETB_CTL). The function verifies the disablement by reading back the ETB_CTL register to ensure that the ETB is no longer active. Upon successful disablement, the function returns 0.

etb_status()

The etb_status function is designed to read and display the current status of the Embedded Trace Buffer (ETB). This function provides valuable information about the state and configuration of the ETB.

etb_retrieve()

The etb_retrieve function reads the trace data captured by the ETB into a provided buffer. It unlocks the ETB hardware, reads the start pointer (ETB_RRP) and the end pointer (ETB_RWP) to determine the amount of data available, resets the read pointer (ETB_RRP), and then reads the trace data into the buffer. The size of the retrieved data is returned. If the size is less than 0, it indicates an error in data retrieval.

4.10 Micro Trace Buffer(MTB)

The sequence diagram 4.12 depicts the interaction between the main application, the MTB driver, and the MTB hardware during the enabling and disabling of the Micro Trace Buffer. Initially, trigger the mtb_enable(mtb_size) function to enable MTB tracing. The MTB driver first validates the mtb_size to ensure it is at least 16 bytes and has a power of 2. If the size is valid, it scrubs the MTB SRAM by setting all bytes to zero, making it easier to identify which parts have been written to.

Next, the MTB driver disables the MTB by calling mtb_disable(), ensuring it can be safely reconfigured. This involves setting the MASTER register to disable tracing. The position counter is then reset to zero, and the MASTER register is configured to enable tracing with the specified buffer size.

After the setup, the main application executes its code, during which the MTB continuously traces the program execution, writing the addresses to the circular buffer. Finally, when tracing needs to be stopped, the main application calls mtb_disable() again to turn off the MTB by clearing the enable bit in the MASTER register.

```
1 int mtb_enable(size_t mtb_size)
2 int mtb_disable(void)
```

Listing 4.9: **API's for MTB**

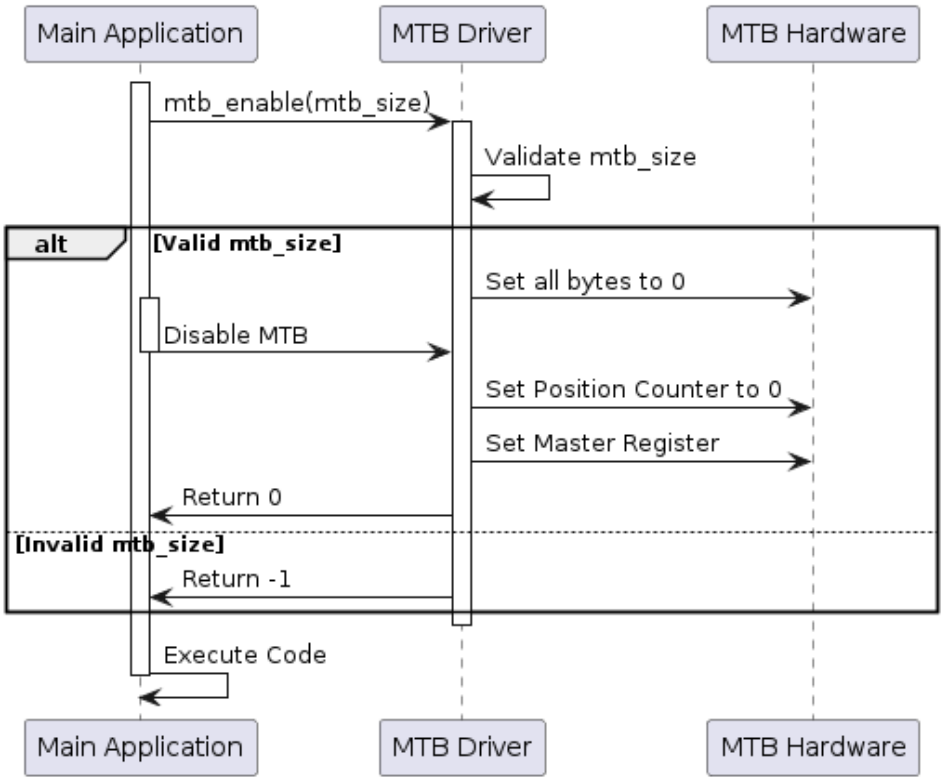

Figure 4.12: **Micro Trace Buffer Sequence Diagram**

mtb_enable()

The mtb_enable function is designed to initialize and start the MTB with a specified buffer size. This involves setting the most significant bit to enable the trace and setting the lower bits to define the size of the buffer.

mtb_disable()

The mtb_disable function, on the other hand, stops the tracing by clearing the enable bit in the MASTER register. This action halts the recording of execution addresses into the trace buffer, effectively pausing the MTB's tracing capability.

Chapter 5

Porting on Hardware

This chapter details the process of porting the embedded software debugging library to various hardware development kits. The objective is to validate and demonstrate the functionality of the debugging library across different platforms, ensuring its versatility and robustness. The selected hardware platforms for this porting process include the Texas Instrument's TMDS64EVM, Adafruit Grand Central M4 Express featuring the SAMD51, and the Renesas DA14695-00HQDEVKT-U. Each of these development kits offers unique features and capabilities, making them ideal candidates for testing the various components of the debugging library.

Each section below will provide an overview of the development kits, describe the setup process, a base code as a part of the application running on each core, and validate the functionality of each component of the software debug library on these platforms. A base code is implemented on each hardware platform to effectively test and validate the debugging library. By incorporating real-world application scenarios, the base code simulates typical use cases that developers might encounter. This includes inter-processor communication, handling interrupts, performing periodic tasks, and more. Testing the debugging library in these real-world scenarios helps ensure that it will perform reliably in actual development environments. This systematic approach ensures that the library is versatile, capable of operating effectively across different hardware environments.

5.1 Texas Instrument's TMDS64EVM

The TMDS64EVM development kit [28] is equipped with an AM64x System-on-Chip (SoC) [29], which integrates a versatile combination of processing cores and advanced debugging features as shown in figure 5.1, making it an ideal platform for embedded system development and debugging. The AM64x SoC includes two ARM Cortex-A53 cores, four ARM Cortex-R5F cores, a single ARM Cortex-M4 core, and two PRU-ICSSG cores, providing a robust and flexible architecture for a wide range of applications [27]. This kit supports real-time processing, industrial communication, and decentralized motor drives, enhanced by three RJ45 Ethernet ports, a high-speed expansion connector, and onboard power measurement capabilities.

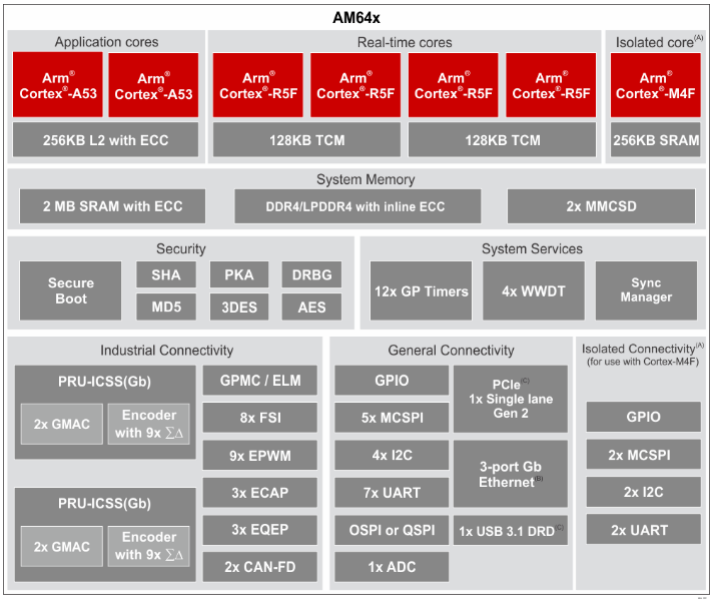

Figure 5.1: **Texas Instruments TMDS64EVM-AM64x Cores and Components Architecture**

For software development, the Processor-SDK-AM64X offers a unified platform with mainline Linux, Long-Term Stable (LTS) kernel support, FreeRTOS, and No-RTOS options for ARM Cortex-R5F, and a variety of integrated demos and examples. Debugging is facilitated through the XDS110 on-board emulator, which supports automatic selection between on-board and external emulation, along with a 20-pin JTAG connection. The kit also features comprehensive memory options, including 2GB DDR4, 16GB eMMC Flash, and multiple EEPROM types. The inclusion of debug interfaces like UART to USB, multiple I2C ports, and user-configurable push buttons ensures detailed monitoring and troubleshooting capabilities, aligning perfectly with the objectives of the embedded software debugging library detailed in this report.

5.1.1 Base Code

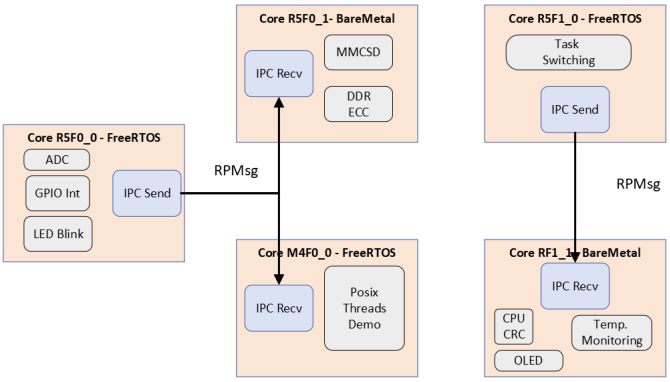

Figure 5.2: **TMDS64EVM-AM64x Base Code System Design**

The provided system design in figure 5.2 outlines the base code running on the evaluation module, showcasing a mix of FreeRTOS and bare-metal implementations to simulate a comprehensive range of functionalities typically used in the industry. This approach thoroughly validates the software debug library, covering diverse scenarios encountered in real-world applications. Each core is tasked with different functionalities, integrating both real-time operating system (RTOS) environments and bare-metal operations based on the Technical Reference Manual [30]. For instance, Core R5F0_0 runs FreeRTOS handling ADC, GPIO interrupts, and LED blinking, while Core R5F0_1 operates in a bare-metal environment focusing on MMCSD and DDR ECC. Similarly, Core R5F1_0, also running FreeRTOS, handles task switching, and Core M4F0_0 and Core RF1_1 operate with IPC communication for various tasks. The subsequent subsections will elaborate on the detailed implementation and validation of the code running on each core. This setup ensures the software debug library is rigorously tested across a broad spectrum of real-time and bare-metal applications, affirming its robustness and versatility.

The following sections illustrate the flow of applications executing on each core programmed in the base code:

R5F0_0-FreeRTOS

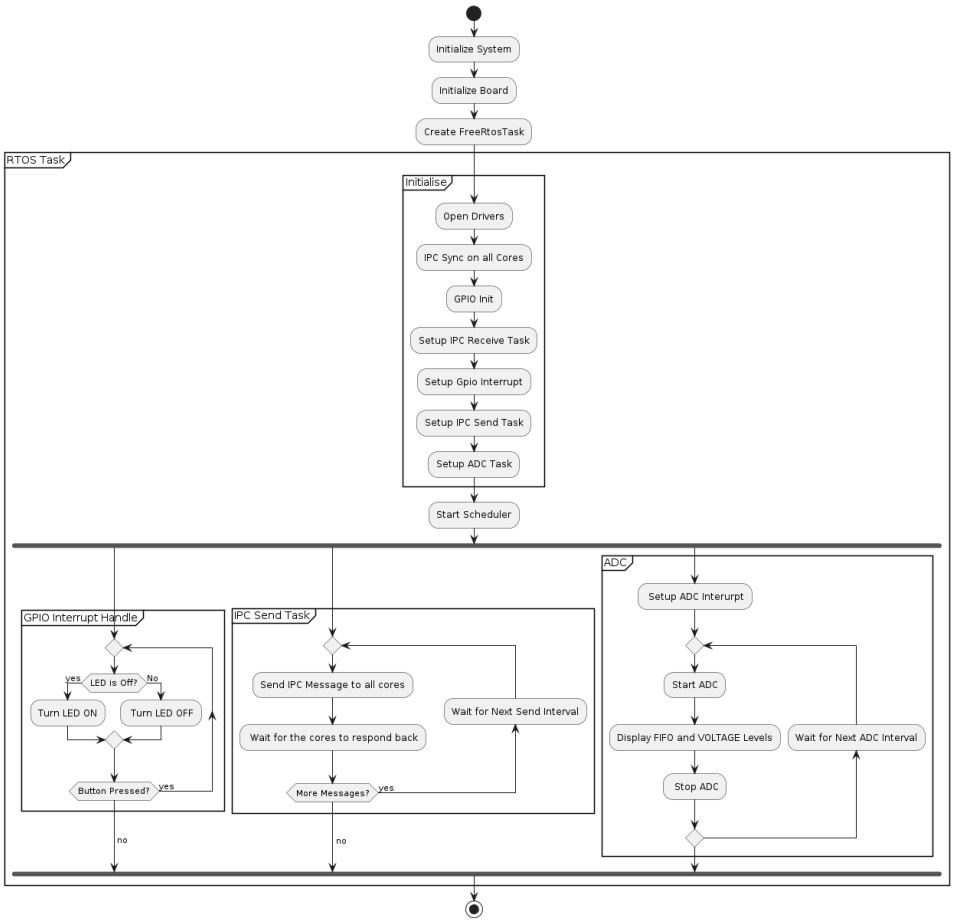

Figure 5.3: **TMDS64EVM-AM64x Core R50_0 FreeRTOS Application**

The R5F0_0 5.3 core runs a FreeRTOS-based application demonstrating inter-processor communication (IPC), GPIO interrupt handling, and ADC single-shot conversions. The main CPU (R5F0_0) sends multi-byte messages to remote CPUs (R5F0_1 and M4F0_0) using RP Message APIs. A GPIO pin is configured to generate an interrupt on a rising edge, toggling an LED and printing the button press count. The ADC is configured to convert all eight input channels in single-shot mode, averaging 16 samples per conversion. Results are stored in FIF0 and displayed on the console. A medium-priority task manages the ADC conversions, ensuring efficient handling without interrupting higher-priority tasks. This application showcases the multi-core communication and peripheral handling of TMDS64EVM in a real-time environment.

R5F0_1-BareMetal

The R5F0_1 5.4 core operates a bare metal application that includes inter-processor communication (IPC), ECC error handling for DDR, and eMMC Flash Raw IO operations. The CPU (R5F0_0) sends multi-byte messages to remote CPUs using RP Message APIs. The R5F0_1 core, acting as a remote CPU, echoes the messages received by the main CPU. The ECC error handling section simulates both single-bit (1b) and double-bit (2b) ECC errors in DDR memory. The application modifies values at specific DDR addresses to introduce errors and attempts to trigger error-handling mechanisms. Single-bit errors are detected and corrected, while double-bit errors are detected and logged. Also, the application performs basic read and write operations to eMMC Flash memory. Known data is written to a specific offset in the eMMC and then read back. The read data is compared with the written data to verify correctness.

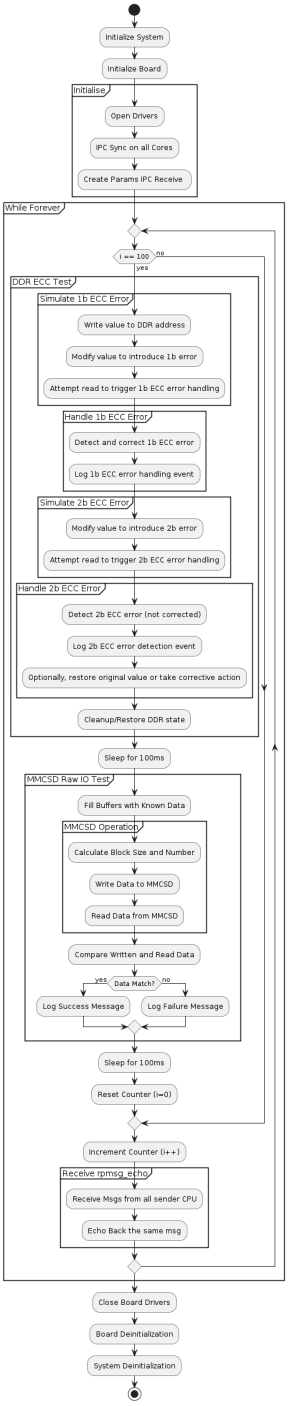

Figure 5.4: **TMDS64EVM-AM64x Core R50_1 BareMetal Application**

R5F1_0-FreeRTOS

The R5F1_0 5.5 core runs a FreeRTOS application that demonstrates semaphore-based task switching through a ping-pong mechanism and inter-processor communication (IPC) for message exchange. The main CPU, R5F1_1, sends multi-byte messages to the remote CPU, R5F1_0, using the RP Message APIs. The R5F1_0 core, acting as a remote CPU, receives the messages and echoes them back to the main CPU. A task is created to handle incoming messages from the remote cores and send them back, ensuring continuous communication and validation of the IPC mechanism. Also, two semaphores are created in this application, and two tasks, ping and pong, are established. These tasks signal each other using semaphores and task notifications to simulate a task-switching environment. The ping task is triggered by a hardware interrupt service routine (ISR), demonstrating how tasks can be signaled from ISRs in a real-time system. The tasks alternate execution by giving and taking semaphores and the task delay API is used to manage task timing and execution flow.

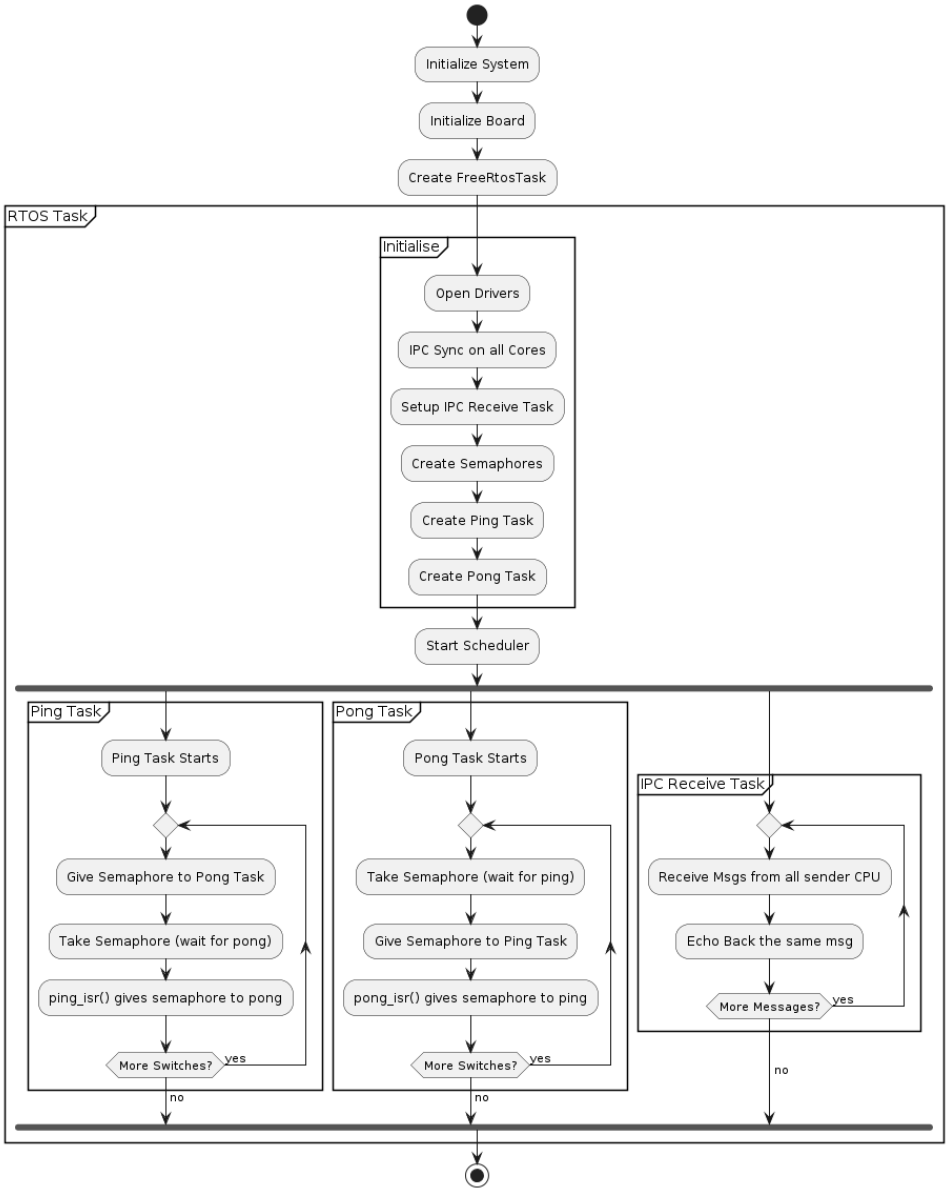

Figure 5.5: **TMDS64EVM-AM64x Core R51_0 FreeRTOS Application**

R5F1_1-BareMetal

The R5F1_1 5.6 core runs a bare-metal application that showcases various functionalities such as I2C communication for temperature reading, OLED display, CRC computation, and inter-processor communication (IPC) for message exchange. The application demonstrates how to interface with an I2C temperature sensor and an OLED display. The system initializes the I2C communication, reads temperature data from the sensor, and displays the temperature on the OLED screen. This process involves taking multiple samples, printing the data to the console, and updating the OLED display continuously. The CRC (Cyclic Redundancy Check) functionality is implemented to verify data integrity. The application configures the CRC parameters, initializes the memory with reference data, and calculates the CRC signature for a data frame stored in memory. The computed CRC signature is then compared with a pre-calculated reference value to ensure data correctness. The CPU, R5F1_1, sends multi-byte messages to remote CPUs using the RP Message APIs. The remote CPUs receive these messages and echo them back to the main CPU.

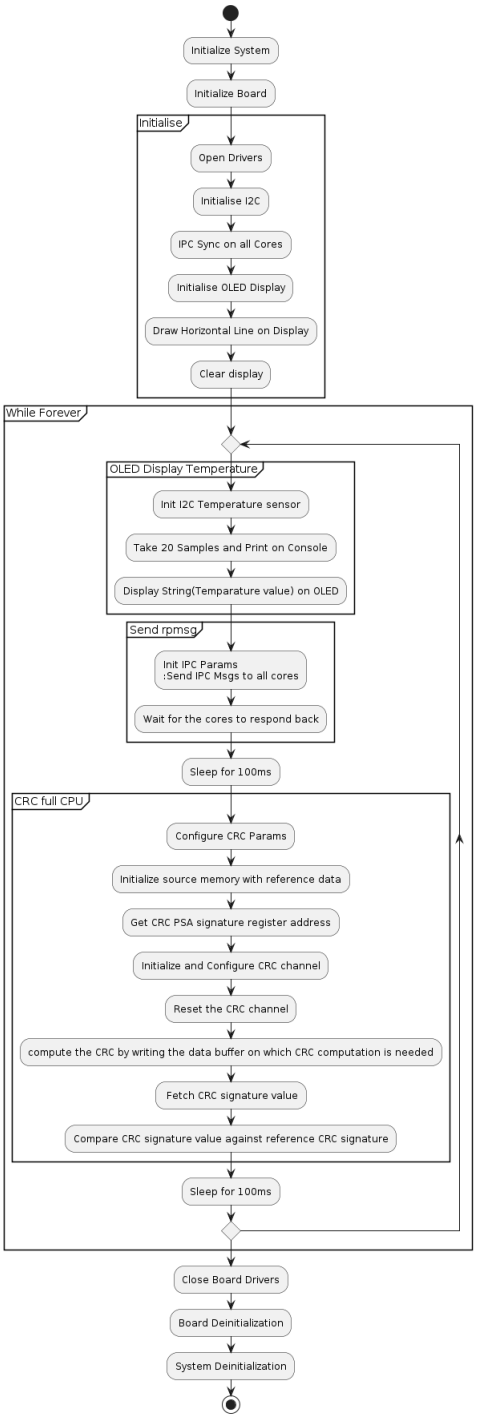

Figure 5.6: **TMDS64EVM-AM64x Core R51_1 BareMetal Application**

M4F0_0-FreeRTOS

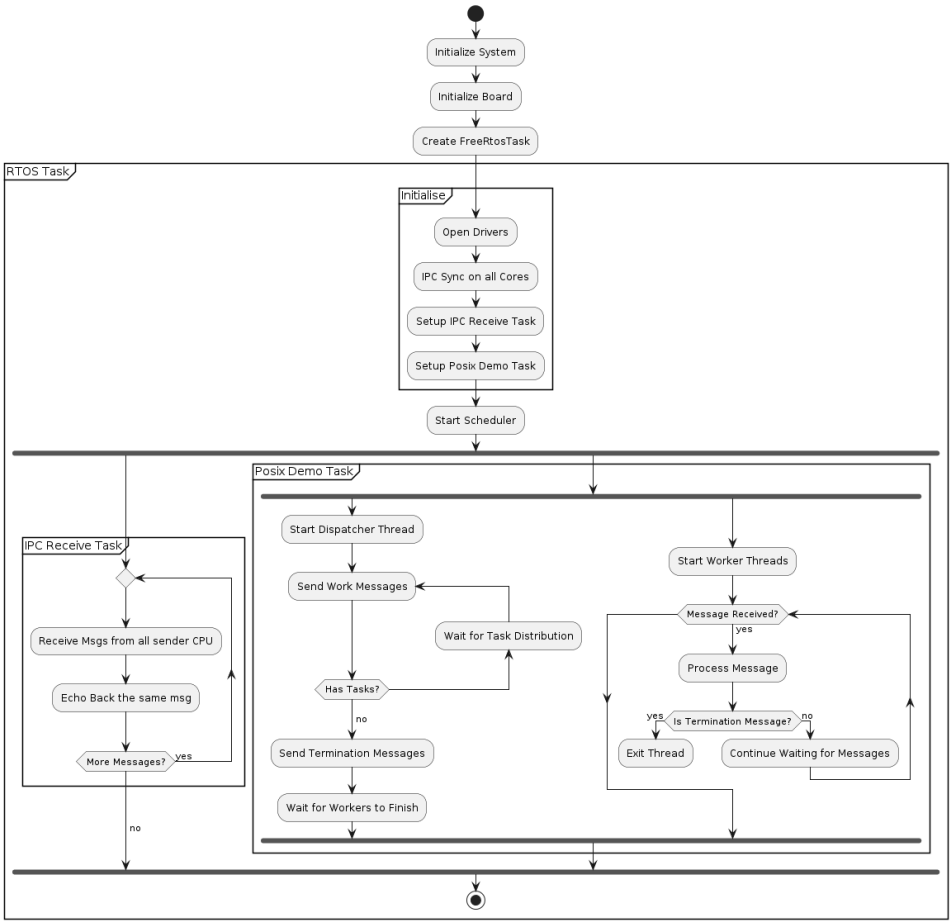

Figure 5.7: **TMDS64EVM-AM64x Core M40_0 FreeRTOS Application**

The M4F core runs a FreeRTOS-based application that demonstrates inter-processor communication using RPMsg and the utilization of POSIX APIs over FreeRTOS. The main CPU, R5F0_0, sends messages to remote CPUs, including R5F0_1 and M4F0_0, using the RPMsg APIs. The remote CPUs then echo back the same messages to the main CPU. This functionality is encapsulated within a task that handles receiving messages from remote cores and sending back the same messages. The application demonstrates the use of POSIX APIs with FreeRTOS running underneath the POSIX layer. The code includes the creation of POSIX threads and message queues. The main tasks involved are:

- Creating Worker Message Queues and Worker Threads: The application initializes multiple message queues and worker threads. Each worker thread waits for messages in its respective queue.
- Creating a Dispatcher Thread: A dispatcher thread sends messages to the

worker message queues. It sends work messages to each queue, prompting the worker threads to process these messages.

- Task Distribution and Termination: After distributing tasks for a certain number of iterations, the dispatcher thread sends exit messages to the worker threads. Upon receiving an exit message, the worker threads complete their tasks and terminate. The main thread waits for all the threads to finish their execution before concluding the application.

5.1.2 Validation of Embedded Debug Library Components

Components	Cores	Validation method
Hardware Breakpoint	R5	Verified through JTAG that the processor halts
Hardware Watchpoint	R5	Verified through JTAG that the processor halts
Inter Processor Communication	R5, M4	Verified through Logs
Synchronous Breakpoint	R5, M4	Verified through Logs and JTAG
Core Dump	M4	Verified through Logs Comparison with CPU registers in IDE
Data Watchpoint and Trace Unit(DWT)	M4	Verified the Unit functionality through JTAG
Flash Patch and Breakpoint Unit(FPB)	M4	Verified the Unit functionality through JTAG
Performance Measurement Unit (PMU)	R5	Verified through Logs
Profiling	M4	Verified through comparing cycle count with delay function
Micro Trace Buffer		No Support in Hardware
Embedded Trace Buffer		No Support in Hardware

Table 5.1: **Validation of Embedded Debug Software Library Components on TMDS64EVM**

The table 5.1 summarizes the validation methods used for various components of the Embedded Debug Software Library on the TMDS64EVM platform, which includes both FreeRTOS and Baremetal configurations. Each component is listed alongside the cores it was tested on and the validation method employed.

Hardware Breakpoints and Hardware Watchpoints were validated on the R5 core using JTAG to confirm that the processor halts as expected. Inter Processor Communication (IPC) was tested on both the R5 and M4 cores, with successful validation through log outputs. The Synchronous Breakpoint functionality, which combines breakpoints and IPC, was also validated on the R5 and M4 cores using both logs and JTAG.

For the Core Dump functionality on the M4 core, logs were compared with CPU registers in the Integrated Development Environment (IDE) to ensure accuracy. The Data Watchpoint and Trace Unit (DWT) and Flash Patch and Breakpoint Unit (FPB) on the M4 core were validated by verifying their functionalities through JTAG. The Performance Measurement Unit (PMU) on the R5 core was validated through logs.

Profiling on the M4 core was confirmed by comparing the cycle count with a known delay function. However, the Micro Trace Buffer (MTB) and Embedded Trace Buffer (ETB) could not be validated as it is not supported in the hardware.

This comprehensive validation ensures that the debug library components function correctly across different cores and configurations on the Texas Instruments' TMDS64EVM platform.

5.2 Adafruit Grand Central M4 - ATSAMD51

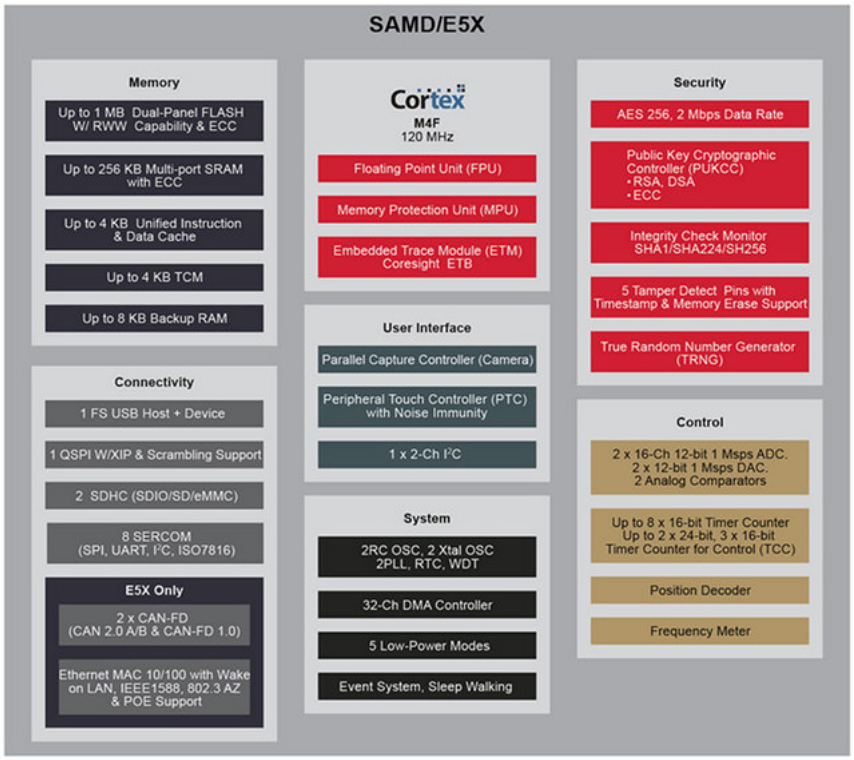

Figure 5.8: **Adafruit Grand Central M4-SAMD51 Cores and Components Architecture [26]**

The Adafruit Grand Central M4 [1] Express featuring the SAMD51 5.9 is a powerful development board tailored for complex embedded applications. At its core, it houses a 120MHz ARM Cortex-M4 processor with floating point support and Cortex-M4 DSP instructions, ensuring robust performance and versatility. The board adopts the Arduino Mega form factor, providing many pins for extensive functionality.

For debugging, the Grand Central M4 includes native USB support, allowing it to operate as a serial device without requiring an additional USB-to-serial converter. Key specifications and core functionality include 1MB of flash memory, 256KB of RAM, 70 GPIO pins, dual 1MSPS DACs, dual 1MSPS ADCs, and eight hardware SERCOMs configurable as I2C, SPI, or UART. Additionally, the integrated crypto engines provide enhanced security features.

5.2.1 Base Code

The Base code aims to enable and verify the functionality of the Embedded Trace Buffer (ETB) in the Adafruit Grand Central M4 featuring the SAMD51 microcontroller [25]. This is accomplished by initializing the system, as per the

SAMD51 datasheet [25] setting specific control registers, and reading memory regions to ensure the ETB is correctly configured.

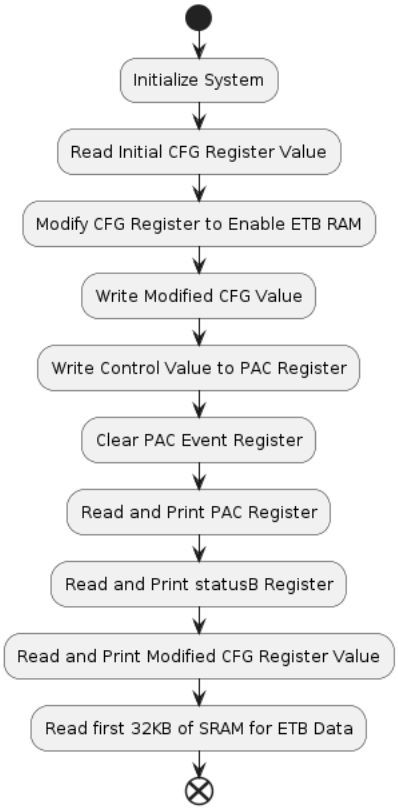

Figure 5.9: **Adafruit Grand Central M4-SAMD51 Flow to Enable ETB**

The processor reads the current configuration (CFG) register value, modifies it to set the ETBRAMEN bit (bit 4), and writes the updated value back to the CFG register. This step is essential as it enables the ETB RAM, allowing it to function correctly. In addition to modifying the CFG register, the processor also writes control values to the Peripheral Access Controller (PAC) registers to configure access control. This includes setting a specific key and peripheral ID in the PAC register and clearing the PAC event register.

For further verification, the processor reads and prints the statusB register value and the modified CFG register value to ensure that the changes were successfully applied.

The Base code also includes a section for reading and printing 32KB of SRAM. This is done by iterating over the SRAM addresses, reading each value, and printing it to the serial monitor.

Moreover, the base code has a loop function that contains a simple LED blinking sequence. It turns the LED connected to pin 13 on and off with a one-second delay in between. This part of the code demonstrates basic functionality and is verified by decoding the ETB data.

5.2.2 Validation of Embedded Debug Software Library Components on Adafruit - M4

Components	Cores	Validation method
Hardware Breakpoint		Verified in M4 through FPB
Hardware Watchpoint		Verified in M4 through DWT
Inter Processor Communication		Single core
Synchronous Breakpoint		Single core
Core Dump		Can't be verified due to absence of Debugger Support in Hardware
Data Watchpoint and Trace Unit(DWT)		Can't be verified due to absence of Debugger Support in Hardware
Flash Patch and Breakpoint Unit(FPB)		Can't be verified due to absence of Debugger Support in Hardware
Performance Measurement Unit (PMU)		No Support in Hardware
Profiling	M4	Verified through comparing cycle count with delay function
Micro Trace Buffer		No Support in Hardware
Embedded Trace Buffer	M4	Verified through decoding ETB data for a simple loop

Table 5.2: **Validation of Embedded Debug Software Library Components on Adafruit M4**

Table 5.2 provides an overview of the validation methods used for different components of the Embedded Debug Software Library on the Adafruit Grand Central M4. The validation focuses primarily on the Cortex-M4 core, reflecting the core capabilities and limitations of the hardware.

Hardware Breakpoints and Hardware Watchpoints were verified on the M4 core using the Flash Patch and Breakpoint (FPB) unit and the Data Watchpoint and Trace (DWT) unit, respectively. Inter Processor Communication (IPC) and Synchronous Breakpoints were not validated due to the single-core nature of the M4 setup.

The Core Dump feature could not be verified due to the absence of debugger support in the hardware. Similarly, the Data Watchpoint and Trace Unit (DWT) and Flash Patch and Breakpoint Unit (FPB) functionalities could not be fully validated for the same reason. The Performance Measurement Unit (PMU) and Micro Trace Buffer (MTB) features are not supported by the hardware.

Profiling was successfully validated by comparing the cycle count with a known delay function, ensuring accurate performance measurement. The Embedded Trace Buffer (ETB) was verified by decoding ETB data for a simple loop, confirming the trace capabilities of the library on the M4 core.

5.3 Renesas DA14695-00HQDEVKT-U

The DA14695 USB Kit [15] from Renesas provides a compact and cost-effective platform for software development and debugging for the DA14695 Bluetooth Smart SoC [14]. The board integrates a variety of features that facilitate easy connectivity and development, including JTAG and UART interfaces for programming and debugging, a QSPI flash memory for data storage, and various GPIOs accessible through MikroBUS slots as shown in figure 5.10. The kit is designed to connect directly to a PC USB port, offering power options through USB, an onboard LDO, or an external battery. Key debugging capabilities are enhanced by a dedicated Segger processor that manages the USB-to-JTAG/UART functions, allowing seamless communication between the development environment and the SoC.

The DA14695 SoC [14] itself is a highly integrated solution combining an ARM

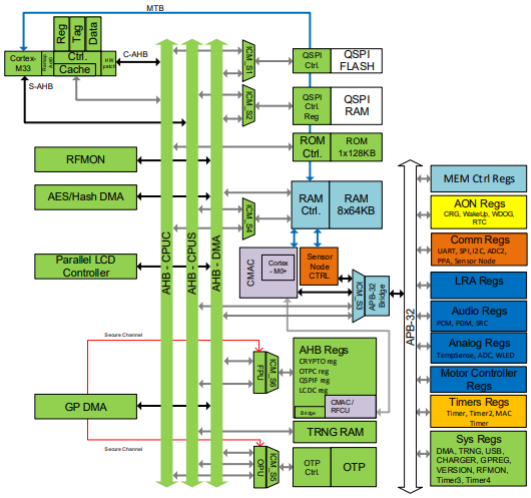

Figure 5.10: **Renesas DA14695 - Cores and Components Architecture [14]**

Renesas

Cortex-M33 CPU, power management, memory, and a configurable Bluetooth Low Energy MAC engine with a radio transceiver. It supports both 32 MHz and 32.768 kHz crystals, enabling precise timing for various operations. The kit includes various features for enhanced development and debugging, such as reset circuits, user-controllable LEDs, and robust power management systems.

5.3.1 Base Code

The MTB stores the addresses of executed instructions in a circular buffer, which can be useful for post-mortem analysis after a crash or unexpected behavior. The MTB position register indicates the current write position in the buffer. For validating the functionality of Micro Trace Buffer the base code flow 5.11 is kept very simple. It initializes the system, configures a GPIO pin to drive an LED, and blinks the LED on and off in a loop.

5.3.2 Validation of Embedded Debug Software Library Components on Renesas DA14695

The table 5.3 details the validation methods employed for different components of the Embedded Debug Software Library on the Renesas DA14695, focusing primarily on the Cortex-M33 core. Each component is evaluated on FreeRTOS as well as BareMetal configuration.

Hardware Breakpoints and Hardware Watchpoints were verified on the M33 core using the Flash Patch and Breakpoint (FPB) unit and the Data Watchpoint and Trace (DWT) unit, respectively. These verifications were performed using JTAG to ensure the CPU halts as expected.

Inter Processor Communication (IPC) and Synchronous Breakpoints could not be validated on the Renesas DA14695 due to its single-core nature, which

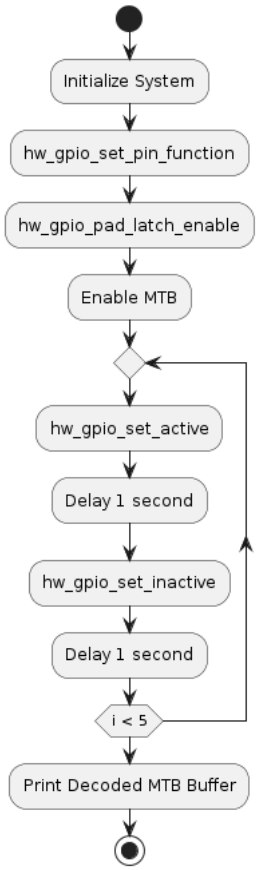

Figure 5.11: **Renesas DA14695 - SAMD51 Micro Trace Buffer Flow**

Components	Cores	Validation method
Hardware Breakpoint		Verified in M33 through FPB
Hardware Watchpoint		Verified in M33 through DWT
Inter Processor Communication		Single core
Synchronous Breakpoint		Single core
Core Dump	M33	Verified through Logs Comparison with CPU registers in IDE
Data Watchpoint and Trace Unit(DWT)	M33	Verified through JTAG that the CPU halts
Flash Patch and Breakpoint Unit(FPB)	M33	Verified through JTAG that the CPU halts
Performance Measurement Unit (PMU)		No Support in Hardware
Profiling	M33	Verified through comparing cycle count with delay function
Micro Trace Buffer	M33	Verified through decoding MTB data and matching with assembly
Embedded Trace Buffer		No Support in Hardware

Table 5.3: **Validation of Embedded Debug Software Library Components on Renesas DA14695**

does not support these multi-core synchronization features.

The Core Dump feature was successfully verified by comparing logs with CPU register values obtained from the IDE, ensuring accurate capture of the processor state during faults. The Data Watchpoint and Trace Unit (DWT) and Flash Patch and Breakpoint Unit (FPB) functionalities were also validated through JTAG, confirming their operational integrity.

The Performance Measurement Unit (PMU) is not supported by the hardware, hence it could not be validated. Profiling was successfully validated by comparing cycle counts with a known delay function, ensuring precise performance measurement.

The Micro Trace Buffer (MTB) was verified by decoding MTB data and matching it with assembly instructions, confirming its tracing capabilities. However, the Embedded Trace Buffer (ETB) is not supported by the hardware, and hence it could not be validated.

Chapter 6

Results and Evaluation

The Results and Evaluation chapter comprehensively analyzes the Embedded Debug Software Library's functionality, performance metrics, and use cases. This chapter is structured to validate the library's various components and evaluate its performance across multiple dimensions. Each section and subsection will detail the methodologies used for validation and measurement, as well as present the results obtained from these evaluations.

6.1 Functionality Validation

The Functionality Validation section focuses on verifying that each component of the Embedded Debug Software Library performs as expected.

6.1.1 Breakpoints and Watchpoints

For R5 Cores the Breakpoints and Watchpoints are validated through JTAG and the IDE. The validation is done by checking that the CPU is halted whenever a Breakpoint or Watchpoint occurs and the address of the instruction is also validated.

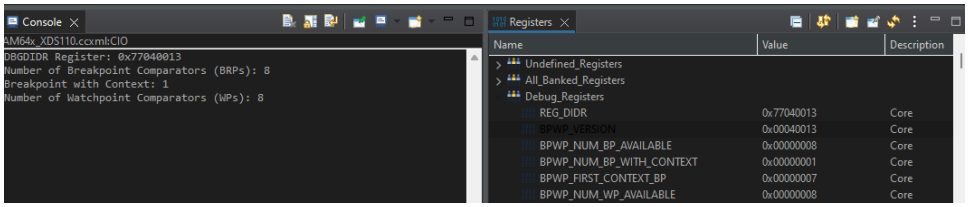

Figure 6.1: **TMDS64EVM -R4 Debug Register Interface**

Figure 6.1 illustrates the validation of the debug register interface on the Cortex-R5 processor by comparing the console output generated by a custom program with the values displayed in the debugger interface. On the left side, the console output shows the DBGDIDR register value as 0x77040013, indicating the processor's debug capabilities. The parsed output reveals that the

processor supports 8 breakpoint comparators (BRPs), 1 of which supports context matching, and 8 watchpoint comparators (WPs). On the right side, the debugger interface displays the contents of various debug registers, confirming the values seen in the console output. The REG_DIDR value matches the DBGDIDR value from the console, verifying the accuracy of the register read operation. Additionally, other registers such as BPWP_NUM_BP_AVAILABLE, BPWP_NUM_BP_WITH_CONTEXT, and BPWP_NUM_WP_AVAILABLE confirm the number of available breakpoints and watchpoints, as well as the context support. The number of breakpoints and watchpoints are also validated through the Technical Reference Manual of TMDS64EVM.

6.1.2 Performance Measurement Unit(PMU)

The functionality of the PMU is tested to ensure it accurately measures performance metrics such as cycle counts, and other critical parameters.

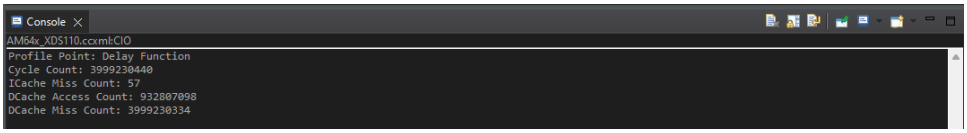

Figure 6.2: **TMDS64EVM -R4 PMU Console**

The total CPU cycles taken for a delay function of 5 seconds on R5 are shown in figure 6.2. To verify the accuracy of profiling, total execution time of the delay function can be calculated. Total code execution time = (1/CPU Frequency) x Total Cycles where the CPU frequency of R5 is 800MHz.

Therefore the **Total code execution time = (1/800 MHz) * 3999230440 = 4.9990 seconds**.

As shown in figure 6.2 the PMU also displays different bus events such as ICACHE MISS, DCACHE MISS, etc.

6.1.3 Inter-Processor Communication

This subsection verifies the IPC mechanisms, particularly focusing on the RP Message APIs. Tests are conducted to ensure reliable message exchange between different CPUs, with logs used to confirm correct operation.

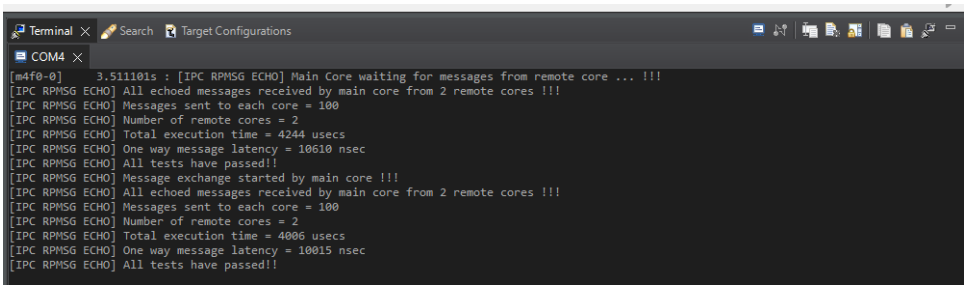

Figure 6.3: **TMDS64EVM - Inter-Processor Communication Output**

Figure 6.3 shows the latency of IPC RPMsg for 100 messages. Once the 100 messages sent by a core to remote cores are echoed back then only the tests are passed.

6.1.4 Synchronous Breakpoint

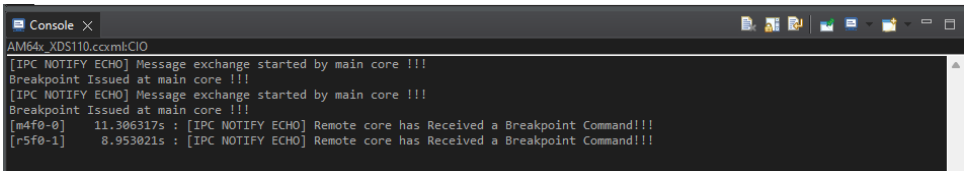

Figure 6.4: **TMDS64EVM - Synchronous Breakpoints Output**

Figure 6.4 demonstrates the console output for validating the synchronous breakpoint functionality using IPC Notify on the TMDS64EVM. The logs indicate the initiation of message exchanges by the main core, followed by the issuance of a breakpoint command. Further down, the logs from the remote cores (M4F0-0 and R5F0-1) indicate the successful reception of the breakpoint command with timestamps, validating the synchronous nature of the breakpoint.

6.1.5 Data and Watchpoint Trace (DWT)

For Cortex - M Cores the Watchpoints are validated through JTAG and the IDE. The validation is done by checking that the CPU is halted whenever a Watchpoint occurs and the address of the instruction is also validated.

Figure 6.5 presents the console output for the DWT validation. The console output begins by showing the DWT control register value, indicating that the DWT is enabled, and it provides the number of comparators available, which is four (NUMCOMP=0x4). For each comparator, the output displays the configuration registers: FUNCTION, COMP, and MASK. The number of watchpoints is also validated through the Technical Reference Manual of TMDS64EVM. For Comparator 0, the COMP register is set to the address of a variable to watch, in-

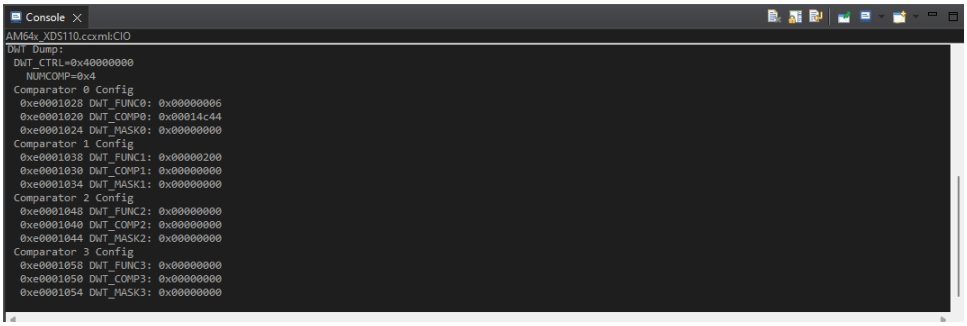

Figure 6.5: **TMDS64EVM - M4 DWT Dump after applying a watchpoint**

dicating an active watchpoint configuration. The other comparators (1 through 3) are shown with zeroed configurations, meaning they are not actively used.

6.1.6 Flash Patch and Breakpoint (FPB)

For Cortex - M Cores the Breakpoints are validated through JTAG and the IDE. The validation is done by checking that the CPU is halted whenever a breakpoint occurs and the address of the instruction is also validated.

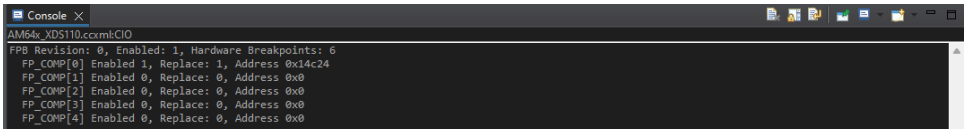

Figure 6.6: **TMDS64EVM - M4 FPB Dump after applying a Breakpoint**

Figure 6.6 shows the console output for the FPB validation. The console displays the FPB configuration, revealing the revision number, whether the FPB is enabled, and the number of hardware breakpoints available, which is six in this case. The number of breakpoints is also validated through the Technical Reference Manual of TMDS64EVM. The console output further details the state of each FPB comparator. FPB Comparator 0 is enabled and is set to address of a variable. The other comparators (1 through 4) are shown as disabled with no addresses configured. This output indicates that the FPB unit is functioning correctly, allowing breakpoints to be set and managed via the comparators.

6.1.7 Profiling

Profiling functionality is validated by comparing cycle counts obtained from the profiling API with known delay functions. This ensures the accuracy of the profiling data.

The total CPU cycles taken for a delay function of 5 seconds on M4 are shown in figure 6.7. To verify the accuracy of profiling total execution time of the delay

Figure 6.7: **TMDS64EVM -M4 Profiling Cycles Console**

function can be calculated. Total code execution time = (1/CPU Frequency) x Total Cycles where the CPU frequency of M4 is 400MHz.

Therefore the **Total code execution time = (1/400 MHz) * 1999762133 = 4.9994 seconds**.

Similar validation is done for M33 as well.

6.1.8 Micro Trace Buffer (MTB)

The MTB's operation is tested by configuring it to store execution traces and verifying the trace data against the expected execution flow. This validates the MTB's integration with the library.

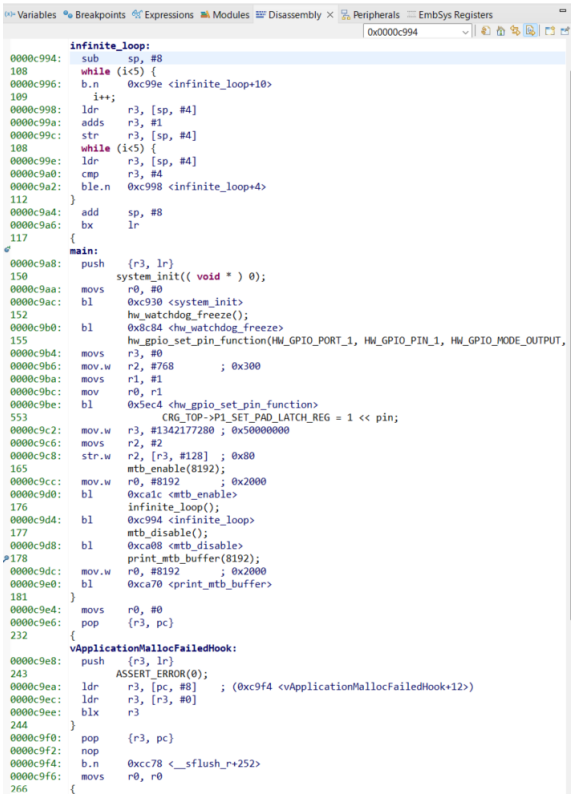

Figure 6.8: **Renesas DA14695 - SAMD51 Disassembly**

Figure 6.8 shows the Disassembly of a simple code running on M33 on Renesas DA14695 - SAMD51 development code.

```
MTB Trace Buffer Contents:

Trace Packet :
  S: 0000CA50 (Flag A: 0, Branch)
  Begin Trace Session
  D: 0000C9D4 (Flag S: 1)

Trace Packet :
  S: 0000C9D4 (Flag A: 0, Branch)
  D: 0000C994 (Flag S: 0)

Trace Packet :
  S: 0000C996 (Flag A: 0, Branch)
  D: 0000C99E (Flag S: 0)

Trace Packet :
  S: 0000C9A6 (Flag A: 0, Branch)
  D: 0000C9D8 (Flag S: 0)

Trace Packet :
  S: 0000C9D8 (Flag A: 0, Branch)
  D: 0000CA08 (Flag S: 0)
```

Figure 6.9: **Renesas DA14695 - SAMD51 MTB Buffer Decoded Data**

Figure 6.9 shows the output of a basic MTB decoder implemented to extract Micro Trace Buffer Data and display the instructions. It can be observed by comparing the figures 6.8 and 6.9 that the instructions extracted from the MTB are the same as those of the disassembly validating the functionality of Traces stored in Micro Trace Buffer.

6.1.9 Embedded Trace Buffer (ETB)

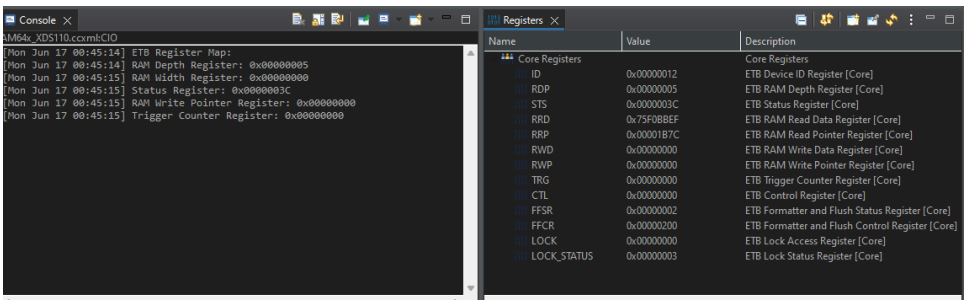

Figure 6.10: **TMDS64EVM - Embedded Trace Buffer Output**

Figure 6.10 illustrates the Embedded Trace Buffer (ETB) validation process on the TMDS64EVM platform. The left side displays the console output after reading the ETB memory map, which consists of a 4KB register space. The register values are printed after performing regional address translation on the R5 core. The console output includes key ETB registers such as the RAM Depth Register, RAM Width Register, Status Register, RAM Write Pointer Register, and Trigger Counter Register.

The XDS110 debug probe output on the right side shows the ETB registers as read from the DebugCell SOC ETB bus. This includes detailed register values such as the Device ID Register, RAM Depth Register, Status Register, and various pointer registers (Read Pointer, Write Pointer). The consistency

between the console output and the debug probe output confirms that the ETB is correctly mapped and accessible.

This validation process verifies the ETB functionality by ensuring that the ETB registers are correctly read and mapped. Additionally, while the library includes functionality to read data from the ETB buffer, decoding this data is currently outside the scope due to its complexity and the time required but can be done using hardware trace decoders such as [61]. Nonetheless, the ability to access and read the ETB registers establishes a foundation for future work in decoding and utilizing ETB data for detailed performance analysis and debugging. This validation underscores the library's capability to interface with core debug components effectively, reinforcing its utility in embedded system development and debugging.

6.2 Performance Metrics

IPC performance

Main Core	Remote Core	Clock Remote Core(MHz)	Average Message Latency (ns)
r5f0-0	m4f0-0	400	1490
r5f0-0	r5f0-1	800	707
r5f0-0	r5f1-0	800	766
r5f0-0	r5f1-1	800	848

Table 6.1: **Average one-way message latency for Inter-Processor Communication for 10,000 messages**

Table 6.1 illustrates the average one-way message latency for Inter-Processor Communication (IPC) across different core configurations. The table presents the main core, the remote core, the clock speed of the remote core, and the observed average message latency in nanoseconds for 10,000 messages. This data is crucial for understanding the performance and efficiency of IPC mechanisms in multi-core embedded systems.

The latency values highlight the time taken for a message to travel from the main core to the remote core and be acknowledged. For instance, the message latency between the main core (r5f0-0) and a remote core running at 400 MHz (m4f0-0) is approximately 1490 nanoseconds. In contrast, communication between cores running at 800 MHz (r5f0-0 to r5f0-1, r5f0-0 to r5f1-0, and r5f0-0 to r5f1-1) exhibits lower latencies, ranging from 707 to 848 nanoseconds. The difference in latencies is due to the clock of the M4F0_0 Core as well as the sequence in which the messages are sent to the cores, for instance, the message is first sent to r5f0_1 core hence it shows the lowest latency.

Several factors, including core clock speeds and the synchronization mechanisms employed, influence the performance of IPC. In the current setup, semaphores are used to manage the synchronization between cores. The system waits for IPC messages to be echoed back before sending new messages, contributing to the observed latencies. This approach, while ensuring message integrity and synchronization, introduces additional wait times, which can be optimized to enhance performance.

Implementing a more efficient synchronization mechanism or reducing semaphore dependency could lower the IPC latency. For example, leveraging non-

blocking synchronization techniques or optimizing semaphore handling can lead to faster message exchanges. Furthermore, the current setup serves as a form of stress test, demonstrating the system's ability to handle high-frequency IPC messages and maintain stability under load.

Synchronous Breakpoint performance

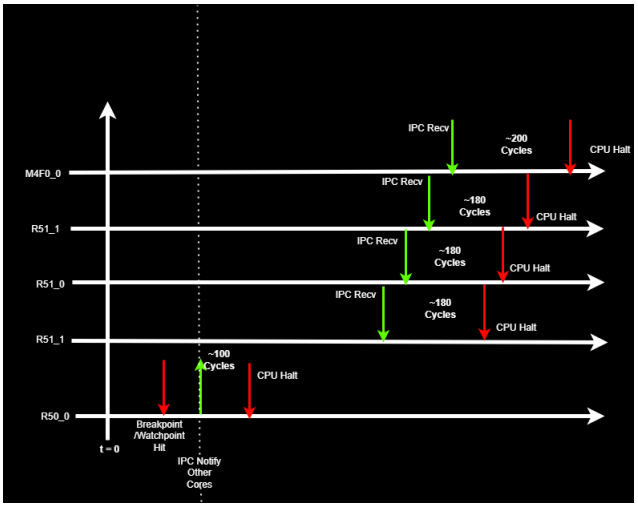

Figure 6.11: **TMDS64EVM - Synchronous Breakpoint Performance**

Figure 6.11 illustrates the output for the synchronous breakpoint mechanism in a multi-core system, highlighting the latency involved in this process. The graph shows the sequence of events across different cores (M4F0_0, R5F1_1, R5F1_0, and R5F0_0), beginning from the initial breakpoint/watchpoint hit on the R5F0_0 core. Upon hitting the breakpoint, the R5F0_0 core sends an IPC notification to all other cores, initiating the synchronization process. The time taken from the IPC interrupt reception to entering the breakpoint handler on each core is measured in cycles, with R5F0_0 taking approximately 100 cycles to halt the CPU, and other cores (R5F1_1, R5F1_0, and M4F0_0) taking around 180 to 200 cycles.

This sequence illustrates that the synchronous breakpoint latency is the sum of the IPC latency and the latency for each core to enter its breakpoint handler. The IPC latencies, provided in Table 6.1, show the average one-way message latency for IPC notify across different cores. The synchronous breakpoint latency can be reduced by optimizing the IPC latency, as it is a significant component of the total latency.

Compared to current synchronous breakpoint solutions available in the market, such as those implemented with GDB or JTAG, this method offers unique advantages and trade-offs. While the latency of the proposed method might be higher as compared to multicore debugging through ARM - DStream or the approach for debugging in a single session GDB presented in [34], it does not require any additional hardware, leveraging the existing IPC mechanism in multi-core, multi-processor systems. Also in [34], the latency of synchronous

breakpoints is not mentioned but it will be a considerable amount as scripting is done in the GDB session to send breakpoint commands to each core and is dependent on the debugger as well. So leveraging the IPC reduces the overall cost and complexity, especially in production environments where external debuggers are often removed. Another alternate solution is to have separate JTAG port interfaces for each core, which adds to the additional complexity of chip designing and cost as well [36, 23].

Furthermore, this method provides greater flexibility and additional capabilities. For instance, during the breakpoint handlers, it is possible to read ETB data or perform a core dump, actions that are typically not feasible with external debuggers. External debuggers also tend to be expensive and are not always practical for use in production systems. This synchronous breakpoint mechanism thus provides a more integrated and cost-effective solution for debugging in complex multi-core systems, enabling advanced diagnostics and debugging capabilities directly within the system.

Breakpoints and Watchpoints performance

Latency is a critical metric in embedded systems debugging [35], reflecting the time delay between the triggering of an event, such as hitting a breakpoint or watchpoint, and the actual halting of the CPU. Measuring latency is essential for understanding the performance impact of debugging tools on the system. In this context, a latency test was conducted to compare the performance of breakpoints and watchpoints implemented using the Debug Watchpoint and Trace (DWT) unit and the Flash Patch and Breakpoint (FPB) unit against those implemented via JTAG.

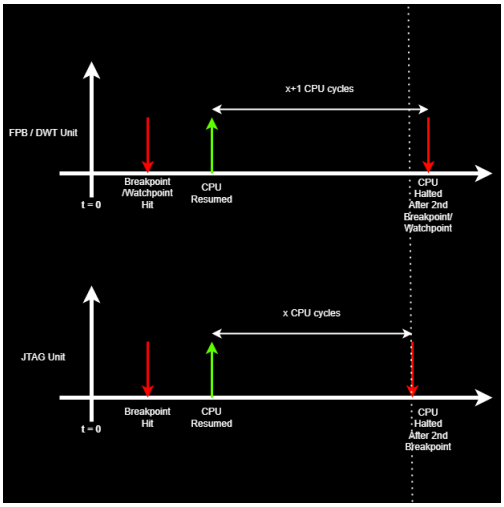

Figure 6.12: **TMDS64EVM - Breakpoint and Watchpoint Performance**

The timing diagram 6.12 illustrates the sequence of events and the corresponding CPU cycles for both scenarios. The upper part of the diagram represents the DWT/FPB unit, while the lower part depicts the JTAG unit. The procedure involved setting a breakpoint and a watchpoint on a variable in the code

and measuring the number of CPU cycles from the moment the breakpoint or watchpoint is hit until the CPU is halted. This comparison involved observing the performance using both the DWT/FPB units and the JTAG interface.

The CPU halts almost instantaneously when a breakpoint or watchpoint is hit using the DWT/FPB unit. The average latency observed is consistent, with minor variations of 1-2 CPU cycles. On the other hand, the latency using the JTAG unit depends significantly on the speed of the JTAG interface. While the performance is generally stable, it can vary depending on the JTAG speed and system load.

The slight variation of 1-2 CPU cycles observed with the DWT/FPB units is negligible and does not impact the actual performance of the code. This variation could be attributed to minor differences in the internal handling of debug events by the core, but these differences are minimal. The DWT and FPB units are core-based, meaning their operation is tightly integrated with the CPU's execution flow, ensuring low-latency responses when breakpoints or watchpoints are hit. In contrast, JTAG-based debugging involves an external interface, where the communication speed between the JTAG probe and the core can influence the latency. This dependency can introduce additional variability in the latency measurements.

The latency tests demonstrate that using DWT and FPB units for breakpoints and watchpoints provides reliable and low-latency performance, with only minor variations that are negligible in practical scenarios. However, this does not significantly affect the overall debugging performance, as the primary use of these units is during the debugging phase rather than in the actual operational phase of the embedded system.

Profiling Performance

When profiling using the Data Watchpoint and Trace (DWT) unit's cycle counter on the Cortex-M4, it was observed that the cycle count obtained from the DWT counter is, on average, 20 cycles higher than the count reported by the JTAG debugger. Given a 400 MHz clock speed, this difference translates to an overhead of merely 50 nanoseconds, which is negligible in most practical scenarios.

This slight discrepancy can be attributed to the method used to read the DWT counter in the code. Specifically, the process of storing the start and end values of the DWT counter involves additional instructions, which introduce extra cycles. These instructions are necessary for capturing the counter values but contribute to the observed overhead.

Despite this minor overhead, the impact on the overall system performance is minimal. The DWT unit operates within the core and provides a highly efficient means of profiling, with the introduced overhead being practically insignificant for most real-time applications.

PMU Performance

The Performance Monitoring Unit (PMU) in the R5 processor core exhibits significantly faster cycle counting performance compared to the cycle counter available through the JTAG debugger. Specifically, the PMU counter records around 1000 fewer cycles for the same operations, running at a clock speed of 800 MHz for the R5 core.

This observed difference, despite the PMU's implementation of start and stop functions, can be attributed to several factors. The PMU is hardware-embedded and operates at the core level i.e. it accesses the co-processor registers, enabling it to capture performance metrics with minimal latency and overhead. In contrast, the JTAG cycle counter relies on external interfacing and communication protocols as it doesn't have direct access to co-processor registers, introducing additional delays that contribute to the higher cycle count. The efficiency of the PMU allows it to monitor performance metrics more precisely and quickly.

Also, the overhead of PMU is 0, hence PMU is a non-invasive debug component that doesn't affect the CPU performance which is also mentioned in Ninja- a tool for malware analysis using trace features [45].

Beyond basic cycle counting, the PMU offers extensive event-tracking capabilities that provide deeper insights into system performance. These include:

- Instruction Cache Miss (ICACHE Miss): Tracks the number of instruction cache misses, providing data crucial for optimizing instruction fetch operations.
- Data Cache Miss (DCACHE Miss): Monitors the number of data cache misses, which is essential for improving data access efficiency.
- Branch Mispredictions: Counts instances where the CPU's branch prediction mechanism fails, helping to fine-tune branch handling strategies.
- Load/Store Unit Stalls: Records stalls in the load/store unit, identifying bottlenecks in-memory operations.
- Pipeline Stalls: Tracks stalls in the instruction pipeline, offering insights into potential areas for pipeline optimization.

These capabilities extend far beyond what is typically available through JTAG, which primarily offers basic cycle counting and limited debug functionalities. The PMU's detailed event tracking is invaluable for performance tuning and optimization, providing a comprehensive view of the system's operational efficiency.

ETB Performance

The Embedded Trace Buffer (ETB) is an essential feature for capturing detailed execution traces in embedded systems, providing significant insights for debugging and performance optimization. While the overhead introduced by the ETB is minimal (0 extra cycles), a comprehensive evaluation has not been extensively performed due to the complexity involved in scripting and decoding the compressed Embedded Trace Macrocell (ETM) data.

In typical usage, the ETB captures trace data with negligible impact on system performance because it's aided by hardware. This minimal overhead is a considerable advantage, allowing continuous monitoring of the system without significantly affecting its operation. However, the actual overhead may include factors such as increased power consumption and potential latency introduced by the trace compression and buffering mechanisms. These factors, while present, are generally outweighed by the substantial benefits provided by the ETB.

Moreover, the performance results after enabling the ETM match with the experiment results of a Tracing tool, Ninja [45] which mentions that the overhead introduced by ETM is less than 0.1%. Hence ETM is also a non-invasive debug feature.

The benefits of using the ETB are manifold. It enables developers to capture detailed execution traces, which are invaluable for post-mortem analysis in case of system crashes or unexpected behavior. This trace data can help identify the root causes of issues, understand the sequence of executed instructions, and develop more effective corrective measures.

Although the ETB doesn't support real-time analysis of traces like the ARM D-Stream debugger, which costs around 3000 euros and requires extra trace pins for sending out trace data, it remains highly valuable in certain scenarios. The ETB has a limited capacity based on the size of the allocated buffer, but it is extremely useful when a debugger can't be used, such as in production environments. It can be utilized for post-mortem analysis using the same ETM trace stored. Additionally, the ETB supports a variety of use cases, including the relocation of trace data to DDR and off-chip export over USB, enhancing its utility beyond immediate debugging needs.

MTB Performance

When profiling a known function with the MTB enabled, it was observed that the function executed with zero additional cycles compared to when the MTB was disabled. This indicates that the overhead of using the MTB for profiling is effectively zero because it's aided by hardware and hence tracing is handled by the processor itself.

The absence of overhead is a significant advantage, as it ensures that the tracing does not interfere with the normal execution of the code. This characteristic is particularly beneficial for real-time systems where maintaining precise timing and performance is crucial. By enabling the MTB, developers can capture detailed execution traces without affecting the system's behavior, thereby obtaining accurate performance data.

Storing and reading MTB data can be immensely useful for post-mortem analysis. In the event of a system crash or unexpected behavior, the trace data stored in the MTB can provide a detailed record of the instructions executed leading up to the event. This information can help developers identify the root cause of the issue, understand the sequence of events, and implement corrective measures. The ability to perform such detailed analysis without introducing runtime overhead makes the MTB an indispensable tool for debugging and optimizing embedded systems.

A known Function takes 0 extra cycles when executing with Micro Trace Buffer enabled. Hence the overhead of MTB is 0, therefore MTB is another non-invasive debug feature.

6.2.1 Memory Footprint

The images present memory usage snapshots for the R5 and M4 cores, both with and without the embedded software debug library included. These snapshots provide insights into how the debug library impacts the memory footprint.

Each image details the memory sections and their usage, highlighting important segments like .text, .data, .bss, and stack memory.

Measuring the size of the software debug library is crucial as it impacts the overall memory footprint of the application. This, in turn, affects the performance and resource utilization of the system. In resource-constrained environments, efficient memory usage is critical to ensure that there is sufficient space for both application code and debug functionalities without exceeding the available memory.

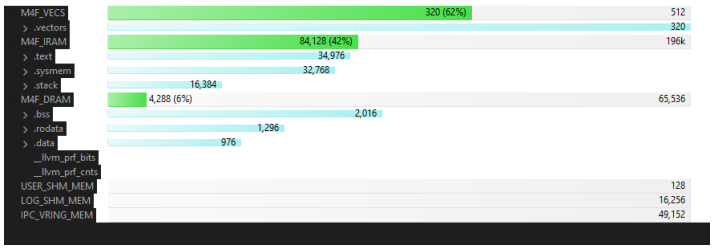

Figure 6.13: **TMDS64EVM - M4 Memory Allocation without Debug Library**

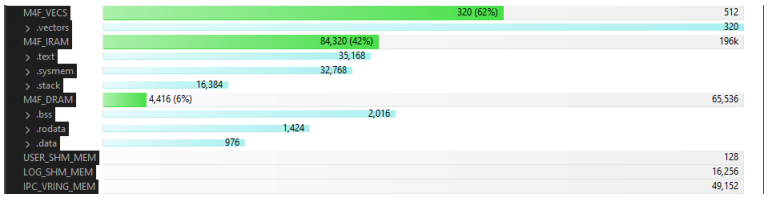

Figure 6.14: **TMDS64EVM - M4 Memory Allocation with Debug Library**

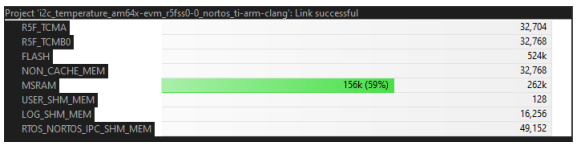

Figure 6.15: **TMDS64EVM - R5 Memory Allocation without Debug Library**

Figure 6.16: **TMDS64EVM - R5 Memory Allocation with Debug Library**

text	data	bss	dec	filename
815	0	0	815	SetWatchpoint.o (ex libemdebug.a)
472	0	0	472	Profiling.o (ex libemdebug.a)
244	0	0	244	HwBreakpointArmR5.o (ex libemdebug.a)
570	0	0	570	HwBreakpoint.o (ex libemdebug.a)
228	0	0	228	Mtb.o (ex libemdebug.a)
2,195	0	3,340	5535	Pmu.o (ex libemdebug.a)
336	0	0	336	ArmR5Pmu.o (ex libemdebug.a)
396	0	0	396	Debug_Register_R5.o (ex libemdebug.a)
268	0	0	268	HwWatchpointArmR5.o (ex libemdebug.a)
1,172	8	0	1180	etb.o (ex libemdebug.a)

Table 6.2: **Sizes of the Embedded Debug Library Components in Bytes**

The table 6.2 presents the size of various components of the embedded debug library, with each component's text, data, bss, and dec values listed. Analyzing the results, it can be observed that the total size of the library is relatively compact, indicating its efficiency. For the R5 cores, key components include SetWatchpoint.o (815 bytes), HwBreakpointArmR5.o (244 bytes), Pmu.o (5535 bytes), ArmR5Pmu.o (336 bytes), Debug_Register_R5.o (396 bytes), and HwWatchpointArmR5.o (268 bytes). The M cores utilize components like Profiling.o (472 bytes), HwBreakpoint.o (570 bytes), Mtb.o (228 bytes), and etb.o (1180 bytes).

The results demonstrate that the PMU component for the R5 cores is the largest, primarily due to its extensive functionality in performance measurement and its access to many co-processor registers. On the other hand, components such as HwBreakpointArmR5.o and HwWatchpointArmR5.o are notably smaller, reflecting their specialized but limited scope. For the M cores, the components exhibit a balanced distribution in size, with none exceeding 1180 bytes.

Most of the code runs in SRAM in R5 and DRAM in M4, so the latencies and size can be further reduced if the code base is set to TCM in R5 and SRAM in M4. Additionally, as this library is a proof of concept, there is significant potential for optimization. Through code refinement and efficiency improvements, the library size can be further reduced, enhancing its suitability for resource-constrained environments.

Since no software debug framework is available for ARM Cores, comparing it with the Software-driven Debug Framework for RISC-V [37] which utilizes one external interrupt, and 1.04KB of memory covering only stop interrupt and de-

bug handler, this library maintains a competitive edge in terms of size, features and efficiency. Also, other frameworks like the OpenOCD project [24] exhibit larger footprints due to their extensive feature sets and broad compatibility requirements. By focusing on core debugging functionalities and maintaining a streamlined codebase, this library offers a lightweight alternative suitable for embedded systems where memory and performance are critical considerations.

Also, debugging tools performing the same tasks can range anywhere between €2900 to $17900 [18]. Whereas, the library is independent of any hardware tool and is relatively simple with a minimum learning curve.

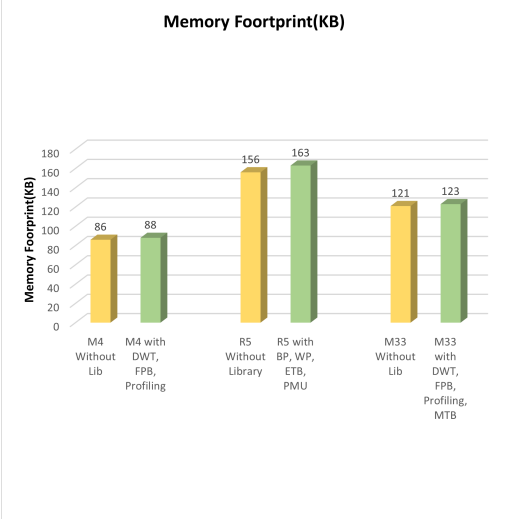

Figure 6.17: **Memory Footprint with and without Debug Library Components on various Cores**

The bar graph 6.17 illustrates the increase in memory footprint when the debug library components are included for various cores. The code sizes in the RAM are of the base codes explained in previous chapters. The inclusion of the debug library demonstrates a minimal increase in memory usage, showcasing its efficiency and low overhead as well as the flexibility to choose different components for different Cores as per the requirements.

6.3 Use Cases

The embedded software debug library offers a range of functionalities that go beyond the capabilities of traditional JTAG debuggers. These additional features provide critical advantages in both development and production environments, particularly where JTAG is not available. Here are some key use cases:

Watchpoints and Breakpoints in Non-RAM Regions:

Most JTAG debuggers can only set breakpoints and watchpoints in RAM. However, this library allows setting watchpoints and breakpoints in other memory

regions, including Flash and peripheral registers, enabling more comprehensive debugging.

Synchronous Breakpoints:

The library supports synchronous breakpoints across multiple cores without requiring additional hardware. This is especially useful for debugging multicore systems where coordinated halts are necessary. Unlike JTAG-based solutions, this can be done without external dependencies.

Post-Mortem Analysis Using ETB:

When a fault occurs, the ETB can transfer trace data to DDR. This data can be analyzed later to understand the sequence of events leading to the fault. This capability is crucial in production environments where JTAG is not available, allowing for effective debugging and issue resolution after the fact, particularly with bugs that are non-deterministic and difficult to reproduce

Profiling and Performance Analysis:

Using the DWT and PMU, the library provides detailed profiling and performance analysis, including cycle counts and cache misses. This level of detail is not typically available with standard JTAG debuggers and is invaluable for optimizing performance and identifying bottlenecks.

Comprehensive ETM Trace:

The ETM trace provides insights into various complex issues that can be difficult to diagnose otherwise [10]. These include:

- Pointer Problems: Identifying invalid pointer dereferences.
- Illegal Instructions and Data Aborts: Tracing how the system reaches fault vectors due to misaligned writes or invalid operations.
- Code Overwrites: Detecting unexpected writes to Flash or peripheral registers.
- Corrupted Stacks and Out of Bounds Data: Diagnosing stack corruption and buffer overflows.
- Uninitialized Variables and Arrays: Identifying the use of uninitialized memory.
- Stack Overflows: Analyzing the causes of stack growth beyond expected limits.
- System Timing Problems: Understanding timing issues within the system.
- Profile Analysis and Code Coverage: Obtaining detailed execution profiles and code coverage metrics, is essential for thorough testing and validation.

Production-Ready Debugging:

In production environments, the library's ability to operate without JTAG makes it indispensable. Features such as transferring ETB data to non-volatile memory upon faults ensure that debugging information is preserved and can be analyzed post-mortem, facilitating robust fault analysis and system reliability.

Versatile Integration:

The library can be used with various debugging tools and interfaces:

- GDB Integration: It can be used with GDB through Python scripting, allowing for powerful and customizable debugging workflows.
- JTAG Support: While it extends beyond the limitations of JTAG, it is still compatible with JTAG for environments where JTAG is available.
- Command Line Interface: The library can be operated via command line, providing a flexible and scriptable interface for debugging tasks.
- Independent Operation: It can function independently, making it suitable for embedded systems that do not have access to external debugging tools.

By providing these advanced debugging capabilities, the embedded software debug library not only enhances the development process but also ensures that systems can be effectively monitored and debugged in production, leading to higher reliability and performance.

Chapter 7

Conclusions

The development of an embedded debugging and profiling library for ARM-based systems presented in this thesis marks a meaningful contribution to the field of embedded system diagnostics. This library addresses critical challenges in debugging multicore and multiprocessor environments, particularly in scenarios where traditional hardware-based debugging tools are inadequate. The primary objective was to design and implement a versatile, software-only debugging library tailored for ARM-based multicore and multiprocessor SoCs. This goal was successfully achieved by leveraging ARM's built-in Trace Infrastructure and debug registers, enabling detailed observation and control over system operations without the need for external hardware.

The library was meticulously designed to be hardware-agnostic, supporting a wide array of ARM processors including the ARM R5, M4, and M33 series. It is also easily extendable to other ARMv7 and ARMv8 series processors, ensuring its future-proofing against the rapid evolution of processor technologies.

In terms of performance, the library was optimized to introduce minimal overhead and maintain low latency, making it suitable for use in real-time embedded systems where performance is paramount. This objective was validated through comprehensive testing and analysis against existing hardware-based debugging methods. The results demonstrated the library's capability to provide detailed performance measurements, profiling, hardware breakpoints and watchpoints, synchronous breakpoints, and inter-processor communication.

The integration of Embedded Trace Macrocell (ETM) and Micro Trace Buffer (MTB) within the library enriches its ability to capture and analyze trace data, facilitating thorough system analysis and post-mortem diagnostics. This advanced feature set aids in design decisions, bottleneck identification, and performance optimization, ultimately enhancing the overall reliability and efficiency of embedded systems.

While the current scope of the library is limited to ARM7 and ARM8 processors, with testing primarily conducted on R5, M4, and M33 cores, the groundwork has been laid for future expansions. Potential future work includes extending support to other processor architectures like RISC-V, implementing step-through debugging, and enhancing CLI and GDB scripting capabilities. Developing automated testing frameworks and real-time trace analysis tools would further improve the robustness and efficiency of the library.

In conclusion, this thesis successfully fulfills its objectives by delivering a

versatile, efficient, and comprehensive software library for embedded system debugging. It stands as a significant contribution to the field, offering practical solutions for scenarios where traditional hardware-based debugging tools fall short. This library represents a crucial step forward in providing robust debugging capabilities for high-performance real-time embedded systems.

Chapter 8

Limitations and Future Work

This chapter delves into the limitations of the current implementation of the embedded software debugging library and proposes recommendations for future work. The objective is to provide a comprehensive overview of the challenges encountered and potential enhancements that could significantly improve the functionality, performance, and versatility of the library. By addressing these limitations and incorporating the recommended future work, the embedded software debugging library can be further optimized to meet the diverse needs of modern embedded systems, ensuring robust and efficient debugging across various platforms and applications. This chapter aims to guide future developments and innovations that will extend the library's capabilities and application scope.

8.1 Dedicated memory for ETB and MTB:

One significant limitation of the library is the handling of ETB (Embedded Trace Buffer) and MTB (Micro Trace Buffer) traces. These traces are not decoded in real-time, necessitating dedicated memory space within the design to store them. Since ETB and MTB are stored on hardware, they require dedicated trace buffers, and the size of these buffers is limited. Consequently, only recent traces can be stored, which might not capture the entire execution history needed for comprehensive analysis. The extraction and post-mortem analysis of these buffers are crucial but add complexity and delay to the debugging process. However, the ETB supports the relocation of trace data to DDR and off-chip export over USB, enhancing its utility beyond immediate debugging needs.

8.2 Limited Scope and Testing:

- Current Scope: The library is designed for ARM 7 and ARM 8 processors, specifically targeting ARM Cortex-R5, Cortex-M4, and Cortex-M33 cores.
- Proof of Concept: Presently, the library operates as a proof of concept. Extensive testing and validation have been conducted only on the aforementioned cores.

- Future Work: To broaden its scope, the library should be tested on a wider range of ARM processors. Automated testing frameworks should be developed and implemented to ensure consistent validation across various hardware platforms.

8.3 Step-Through Debugging :

- Current Limitation: The library does not yet support step-through debugging, a critical feature for detailed code analysis and debugging.
- Future Work: Implementing step-through debugging will enhance the library's usability by allowing developers to inspect code execution line-by-line. This feature would significantly aid in identifying and resolving bugs more efficiently.

8.4 Expansion to RISC-V Processors:

- Current Limitation: The library is currently limited to ARM architecture.
- Future Work: Expanding the library to support RISC-V processors would increase its applicability and adoption as done in [37]. Given the growing popularity of RISC-V, integrating support for this architecture would future-proof the library and broaden its user base.

8.5 ETB and MTB Trace Decoding:

- Current Limitation: The library collects Embedded Trace Buffer (ETB) and Micro Trace Buffer (MTB) data but does not decode these traces in real time. This necessitates additional memory space for storing traces, which are then extracted and analyzed later.
- Future Work: Implementing real-time decoding of ETB and MTB traces would enhance the library's efficiency and reduce the memory footprint using scripts as mentioned in [8]. This improvement would allow immediate analysis and debugging, streamlining the debugging process.

8.6 CLI, GUI and GDB Scripting:

- Current Limitation: Command Line Interface (CLI) or Graphic User Interface (GUI) such as in [56]and scripting for GDB are not yet implemented, limiting the flexibility and automation capabilities of the library.
- Future Work: Developing a robust CLI and integrating scripting support for GDB would provide users with powerful tools for automated debugging and custom script execution. This enhancement would make the library more versatile and user-friendly, particularly for advanced debugging scenarios.

Bibliography

[1] Adafruit grand central m4. https://learn.adafruit.com/adafruit-grand-central.

[2] Lars Albertsson. Holistic debugging-enabling instruction set simulation for software quality assurance. Technical report, 2006.

[3] Seyed Mohammad Ali Zeinolabedin, Johannes Partzsch, and Christian Mayr. Analyzing arm coresight etmv4.x data trace stream with a real-time hardware accelerator. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1606–1609, 2021.

[4] ARM Ltd. CoreSight Architecture, 2021. https://developer.arm.com/documentation/ihi0035/b/Introduction/About-the-Program-Trace-Macrocell/The-debug-environment.

[5] ARM Ltd. Cortex-M33 Architecture, 2021. https://developer.arm.com/Processors/Cortex-M33.

[6] ARM Ltd. Cortex-M4 Architecture, 2021. https://developer.arm.com/Processors/Cortex-M4.

[7] ARM Ltd. Cortex-R5 Architecture, 2021. https://developer.arm.com/Processors/Cortex-R5.

[8] ARM Ltd. ETB Decode, 2021. https://developer.arm.com/documentation/ka005450/latest/.

[9] ARM Ltd. ETM Architecture Version, 2021. https://developer.arm.com/documentation/ddi0242/b/functional-description/data-formatter/etm-architecture-version.

[10] ARM Ltd. ETM Trace Benefits, 2021. https://learn.arm.com/learning-paths/microcontrollers/uv_debug/5_etm_trace/.

[11] Ying Bai. *ARM® Microcontroller Architectures*, pages 13–82. 2016.

[12] Xiaoxiao Bian. Implement a virtual development platform based on QEMU. In *Proceedings - 2017 International Conference on Green Informatics, ICGI 2017*, pages 93–97. Institute of Electrical and Electronics Engineers Inc., 11 2017.

[13] Helmar Burkhart and Roland Millen. Performance-Measurement Tools in a Multiprocessor Environment. Technical Report 5, 1989.

[14] Renesas Electronics Corporation. Da1469x datasheet, 2021. https://www.renesas.com/us/en/document/dst/da1469x-datasheet?r=1606281.

[15] Renesas Electronics Corporation. Smartbond da14695 bluetooth low energy 5.2 usb development kit, 2021. https://www.renesas.com/us/en/products/wireless-connectivity/bluetooth-low-energy/da14695-00hqdevkt-u-smartbond-da14695-bluetooth-low-energy-52-usb-development-kit.

[16] Naim Dahnoun. *Single and multicore debugging*, pages 280–323. 2018.

[17] Bi Xiang Dai. Simulation implementation of embedded protection control platform. In *2019 IEEE Innovative Smart Grid Technologies - Asia (ISGT Asia)*, pages 1995–1998, 2019.

[18] Mihai Daraban, Cosmina Corches, Raul Fizesan, and Gabriel Chindris. Real-time embedded framework debugger. pages 36–39, 10 2022.

[19] Thibault Delavallée, Philippe Manet, Hans Vandierendonck, and Jean-Didier Legat. Embedding functional simulators in compilers for debugging and profiling. In *2011 Faible Tension Faible Consommation (FTFC)*, pages 55–58, 2011.

[20] Pavel Dovgalyuk. Deterministic replay of system's execution with multi-target QEMU simulator for dynamic analysis and reverse debugging. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 553–556, 2012.

[21] Pavel Dovgalyuk, Denis Dmitriev, and Vladimir Makarov. PROCEEDING OF THE 18TH CONFERENCE OF FRUCT ASSOCIATION Platform-Independent Reverse Debugging of the Virtual Machines. Technical report.

[22] Jakob Engblom. A REVIEW OF REVERSE DEBUGGING. Technical report.

[23] Xiaolei Hu, Yu Jin, and Zhaolin Li. A parallel jtag-based debugging and selection scheme for multi-core digital signal processors. In *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*, pages 527–530, 2018.

[24] Hubert Högl, Dominic Rath, Hubert Hoegl@fh-Augsburg, Dominic De, Fachhochschule De, and Augsburg. Open on-chip debugger–openocd–. 01 2007.

[25] Microchip Technology Inc. Sam d5x/e5x family data sheet, 2017.

[26] Microchip Technology Inc. Adafruit grand central m4 diagram, 2022. https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/32-bit-mcus/sam-32-bit-mcus/sam-e/_jcr_content/root/responsivegrid/container_2034953330_60347119/image.coreimg.jpeg/1646110556056/170619-mc32-diag-samd-e5x-7x5-no-title.jpeg.

[27] Texas Instruments. Am64x datasheet, 2023. https://www.ti.com/lit/gpn/am6442.

[28] Texas Instruments. Tmds64evm overview, 2023. https://www.ti.com/tool/TMDS64EVM.

[29] Texas Instruments. Tmds64evm sdk, 2023.

[30] Texas Instruments. Tmds64evm technical reference manual, 2023. https://www.ti.com/lit/pdf/spruim2.

[31] Chengyan Jiang and Siyuan Wu. Design of a general embedded software debug system. In *2010 2nd International Conference on Computer Engineering and Technology*, volume 3, pages V3–467–V3–470, 2010.

[32] Myoungsoo Jung, Jie Zhang, Ahmed Abulila, Miryeong Kwon, Narges Shahidi, John Shalf, Nam Sung Kim, and Mahmut Kandemir. SimpleSSD: Modeling Solid State Drives for Holistic System Simulation. *IEEE Computer Architecture Letters*, 17(1):37–41, 1 2018.

[33] Karl UC Koscher San Diego, Tadayoshi Kohno, and David Molnar Microsoft. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. Technical report.

[34] Santosh Kumar, GP Potdar, Pravin Game, Pict Pune, and GS Lab Pune. APPROACH FOR DEBUGGING IN A SINGLE SESSION GDB. Technical report.

[35] J. Langer, K. Koppenberger, C. Sulzbachner, and T. Nestler. Debug-tool for embedded real time systems. In *EUROCON 2005 - The International Conference on "Computer as a Tool"*, volume 1, pages 599–602, 2005.

[36] R. Leatherman and N. Stollon. An embedding debugging architecture for socs. *IEEE Potentials*, 24(1):12–16, 2005.

[37] Jimin Lee, Jae Min Kim, Junho Huh, and Jungwoo Kim. Software-driven Debug Framework for Embedded RISC-V, that Transparently Emulates the Industry Standard Debug Framework. In *Digest of Technical Papers - IEEE International Conference on Consumer Electronics*, volume 2023-January. Institute of Electrical and Electronics Engineers Inc., 2023.

[38] ARM Ltd. Arm cortex-m33 technical reference manual, 2023. https://developer.arm.com/documentation/100230/0100.

[39] ARM Ltd. Arm cortex-m4 technical reference manual, 2023. https://developer.arm.com/documentation/100166/0001/.

[40] ARM Ltd. Arm cortex-r5 technical reference manual, 2023. https://developer.arm.com/documentation/ddi0460/d/.

[41] ARM Ltd. Armv7 technical reference manual, 2023. https://developer.arm.com/documentation/ddi0406/cd/.

[42] Peter Magnusson and Bengt Werner. Efficient Memory Simulation in SIMICS. Technical report.

[43] Zhenyu Ning, Chenxu Wang, Yinhua Chen, Fengwei Zhang, and Jiannong Cao. Revisiting ARM Debugging Features: Nailgun and its Defense. *IEEE Transactions on Dependable and Secure Computing*, 20(1):574–589, 1 2023.

[44] Zhenyu Ning and Fengwei Zhang. Understanding the Security of ARM Debugging Features. Technical report.

[45] Zhenyu Ning and Fengwei Zhang. Hardware-assisted transparent tracing and debugging on arm. *IEEE Transactions on Information Forensics and Security*, 14(6):1595–1609, 2019.

[46] NXP Semiconductors. Micro Trace Buffer, 2021. `https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/kinetis%40tkb/752/1/Micro%20trace%20buffer.pdf`.

[47] Hyeongbae Park, Jingzhe Xu, Jeong-Hoon Ji, Jusung Park, and Gyun Woo. Design methodology for on-chip-based processor debugger. *Design Automation for Embedded Systems*, 19, 04 2014.

[48] M. Pena-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, Y. Morilla, and P. Martin-Holgado. Online error detection through trace infrastructure in ARM microprocessors. *IEEE Transactions on Nuclear Science*, 66(7):1457–1464, 7 2019.

[49] Vasiliy Pinkevich and Alexey Platunov. Method for testing and debugging flow formal specification in full-stack embedded systems designs. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4, 2020.

[50] M Krish Ponamgi, Wenwey Hseush, and Gail E Kaiser. Debugging Multithreaded Programs with MPD. Technical report.

[51] Y T Poornima, Suresh Reddy Kalathuru, and Prerana Gupta Poddar. Mailbox based inter-processor communication in soc. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, pages 1033–1037, 2017.

[52] Mondipalle Prathyusha and C.V. Ravi Kumar. A survey paper on debugging tools and frameworks for debugging real time industrial problems and scenerios. In *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, pages 1–4, 2019.

[53] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design Test of Computers*, 18(6):23–33, 2001.

[54] Bhanu Singh and Sharath Patil. Single wire debug interface. In *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 814–817, 2020.

[55] Aaron Spear, Markus Levy, and Mathieu Desnoyers. Using tracing to solve the multicore system debug problem. *Computer*, 45(12):60–64, 2012.

[56] Alan P Su, Jiff Kuo, Kuen-Jong Lee, Ing-Jer Huang, Guo-An Jian, Cheng-An Chien, Jiun-In Guo, and Chien-Hung Chen. Multi-core software/hardware co-debug platform with arm coresight™, on-chip test architecture and axi/ahb bus monitor. In *Proceedings of 2011 International Symposium on VLSI Design, Automation and Test*, pages 1–6, 2011.

[57] G. Tanyeri, T. Messiter, and Paul Beckett. Framework-based debugging for embedded systems. pages 424–454, 01 2014.

[58] Texas Instruments. IPC Architecture, 2021. `https://software-dl.ti.com/processor-sdk-rtos/esd/docs/latest/rtos/index_device_drv.html`.

[59] Bart Vermeulen. Functional Debug Techniques for Embedded Systems. Technical report.

[60] Bart Vermeulen and Kees Goossens. A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor socs. In *2009 International Symposium on VLSI Design, Automation and Test*, pages 183–186, 2009.

[61] Matthew Edwin Weingarten, Nora Hossle, and Timothy Roscoe. High throughput hardware accelerated coresight trace decoding. In *2024 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2024.

[62] Michael Williams. ARMV8 DEBUG AND TRACE ARCHITECTURES. Technical report, 2012.

[63] Jin Xue, Renhai Chen, and Zili Shao. SoftSSD: Software-defined SSD Development Platform for Rapid Flash Firmware Prototyping. In *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors*, volume 2022-October, pages 602–609. Institute of Electrical and Electronics Engineers Inc., 2022.

[64] Baek Youngsik') and Jin Sungil. IEEE TENCON '93 / Bcijnn SOFTWARE ABORT AND MULTIPROCESSOR DEBUGGING. Technical report.

[65] Seyed Mohammad Ali Zeinolabedin, Johannes Partzsch, and Christian Mayr. Real-Time Hardware Implementation of ARM CoreSight Trace Decoder. *IEEE Design and Test*, 38(1):69–77, 2 2021.

[66] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. Using hardware features for increased debugging transparency. In *Proceedings - IEEE Symposium on Security and Privacy*, volume 2015-July, pages 55–69. Institute of Electrical and Electronics Engineers Inc., 7 2015.

[67] Feng Zhu and Yiping Yao. A bug locating method for the debugging of parallel discrete event simulation. In *Proceedings - 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation, PADS 2012*, pages 81–83, 2012.

Chapter 9

List of Abbreviations

Abbreviation	Full Form
ADC	Analog-to-Digital Converter
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ARM	Advanced RISC Machine
AXI	Advanced eXtensible Interface
BP	Breakpoint
CPU	Central Processing Unit
DRAM	Dynamic Random-Access Memory
DWT	Data Watchpoint and Trace
ETB	Embedded Trace Buffer
ETM	Embedded Trace Macrocell
FPB	Flash Patch and Breakpoint
GDB	GNU Debugger
GPIO	General-Purpose Input/Output
HW	Hardware
IPC	Inter-Processor Communication
IRAM	Instruction Random-Access Memory
ITM	Instrumentation Trace Macrocell
JTAG	Joint Test Action Group
MTB	Micro Trace Buffer
PMU	Performance Measurement Unit
ROM	Read-Only Memory
SRAM	Static Random-Access Memory
SWD	Serial Wire Debug
TCM	Tightly Coupled Memory
TPIU	Trace Port Interface Unit
TRM	Technical Reference Manual
UART	Universal Asynchronous Receiver-Transmitter

Table 9.1: **List of Abbreviations**