Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Electron Devices

Navaneeth Paliath

IMPLEMENTATION OF A SECURE DEBUG MECHANISM FOR AN AUTOMOTIVE ECU

SUPERVISORS

Internal Supervisor

Dr. Gusztáv Hantos
BME, Budapest

External Supervisor

Juha Mäki-Asiala
Elektrobit, Finland

BUDAPEST, 2023

Table of contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Definition
ARM (Company name)	Advanced RISC Machine
AUTOSAR	AUTomitive Open Software ARchitecture
CLI	Command Line Interface
DLT	Diagnostic Log and Trace
ECC	Elliptic Curve Cryptography
ECU	Electronic Control Unit
GUI	Graphical User Interface
HSM	Hardware Security Module
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JTAG	Joint Test Action Group
LR	Link Register
LSB	Least Significant Bit
MSB	Most Significant Bit
NVIC	Nested Vectored Interrupt Controller
OEM	Original Equipment Manufacturer
PA	Power Architecture
PC	Program Counter
RISC	Reduced Instruction Set Computing
RSA	Rivest-Shamir-Adleman
SHE	Secure Hardware Extension
SP	Stack Pointer
SWD	Serial Wire Debug
TPM	Trusted Platform Module
USB	Universal Serial Bus
xx	Microcontroller version (Eg: 36)

STUDENT DECLARATION

I, **Navaneeth Paliath**, the undersigned, hereby declare that the present MSc thesis work has been prepared by myself and without any unauthorized help or assistance. Only the specified sources (references, tools, etc.) were used. All parts taken from other sources word by word, or after rephrasing but with identical meaning, were unambiguously identified with explicit reference to the sources utilized.

I authorize the Faculty of Electrical Engineering and Informatics of the Budapest University of Technology and Economics to publish the principal data of the thesis work (author's name, title, abstracts in English and in a second language, year of preparation, supervisor's name, etc.) in a searchable, public, electronic and online database and to publish the full text of the thesis work on the internal network of the university (this may include access by authenticated outside users). I declare that the submitted hardcopy of the thesis work and its electronic version are identical.

Full text of thesis works classified upon the decision of the Dean will be published after a period of three years.

Budapest, 1 July 2023

...…….
Navaneeth Paliath

Summary

Automotive Electronic Control Units (ECUs) are safety critical embedded systems that are required to make important decisions based on several inputs, within a very short amount of time. Collecting information from multiple sensors and other physical units to process and make decisions in such short time is achieved through complex software packages. Complex and tightly integrated software are always accompanied by the difficulty to isolate and debug failures quickly, upon occurrence. The current industry standard is to set up hardware debug facilities and employ powerful debugging tools to instrument the ECU binaries line by line. While this opens a large window for deeper analysis, they are bound by the limitation of being difficult to set up and time consuming to analyze. In addition, reproduction of the failure scenario is necessary which is in turn challenging as the inputs and environment must be replicated. With increased demand for automotive ECUs, the fast development, prototyping, and testing are ever more important to ECU software and hardware vendors as it is to OEMs. Hence there exists a need to have a light debugging framework that can capture system errors during execution and test, to aid in quicker analysis of failure root cause. The thesis work involves the analysis of such frameworks for embedded controllers with limited resources followed by the implementation of a secure logging and crash dump mechanism for a specific component of the automotive cybersecurity software stack, the Hardware Security Module (HSM).

1 Introduction

Engine control was the first automotive application for an embedded system. It was known as an ECU or Engine Control Unit computer (now ECU broadly refers to Electronic Control Units). The first ECU, which was designed to perform electronic fuel injection inside an engine, originally appeared in a Volkswagen car in 1968. [1] Starting from one ECU for controlling an engine function, modern embedded systems find their use in a multitude of automotive functions such as airbags, collision avoidance, telematics, traction control, in-car entertainment systems, climate control, anti-lock braking systems, etc. This push for electronics in cars came as a direct result of a demand for increased automotive safety. As the number of vehicles on the road grew a measure was required to ensure that rising vehicle numbers do not pose a threat to drivers, pedestrians, and others. [2] This was the beginning of automotive safety and standards. There are primarily two safety divisions in automotive industries, Passive and Active safety. Passive safety systems focus on reducing the impact of an accident after the accident has taken place. For example, airbags fall under passive safety as they are deployed immediately after an accident has occurred and help prevent injuries to drivers. Another example is seatbelts. Alternatively, active safety systems focus on avoiding safety hazards or try reducing the impact of an accident before it has taken place. Examples of active safety systems include Antilock braking systems, emergency braking, traction control, and so on. Not all the functions under active safety focus on preventing accidents but can be a mix of comfort and safety features.

Modern automotive Electronic Control Units are a powerhouse of information and almost every function in an automobile is routed via such controllers. By accessing the information from these controllers, it would be possible to control various vehicle functionalities thus making such systems a target for attacks. With regards to these concerns, the Automotive Open Software Architecture (AUTOSAR) group introduced some security modules into the automotive framework which can safeguard critical data within the ECU and prevent external attacks. In addition, with the advent of connected cars and features such as FOTA (Flash Over-the-air), these controllers have the possibility of interacting with the external world over wireless communication protocols, which have further widened the possibilities of cyber-attacks and information breaches.

Secure hardware components within an automotive ECU – Secure Hardware Extension (SHE), Trusted Platform Module (TPM), and Hardware Security Module (HSM) – represent the security pillars that help store critical data such as passwords, prevent unauthorized access to such data and help accelerate the cryptographic algorithms employed in the component by providing a controlled and isolated environment which ensures the confidentiality of the algorithms and cryptographic keys. [3] They are the entry point for any software within the system and act as verification mechanisms before the software is allowed to access other functional units. Furthermore, these hardware components have the capability to employ special cryptographic accelerators to improve the cryptographic operation such as encryption and decryption thereby enabling faster processing and reducing the load on the application processor. With respect to their individual functionalities, TPM is a hardware chip on the motherboard that stores the cryptographic keys and passwords required for authentication, HSM is a physical device that generates and manages these cryptographic keys while providing cryptographic processing capabilities, and SHE is a security concept that shifts the control of cryptographic keys from software to hardware (with less capability for asymmetric cryptography) thereby reducing the possibility of software attack. [4]

1.1 Motivation and aim

The automotive security package developed at Elektrobit houses a Hardware Security Module for cryptographic processing and key management. The HSM is a part of the crypto stack within the AUTOSAR which is tasked with authentication and key management activities through the implementation of several cryptographic algorithms and security protocols, isolating the software module from the external world. With regards to the hardware, HSM software is flashed on an automotive controller in which the CPU where the application software runs forms the Host system, and a separate processor core with its own peripherals from the HSM system. The security operations begin with the access of the HSM functionalities through an external Application Programming Interface (API) or referred to as proxyAPI, invoked by the host system. The security module reserves access to all the memory locations in the ECU which includes both the host memory and HSM memory. Implementing the layered security functionalities generate a complex environment with millions of lines of code that makes trying to uncover functional issues an extremely difficult task.

This complex software system with layers of functional code put forth a difficult environment to quickly debug a system failure. For example, each time a customer reported failure is to be analyzed, the approach has been to resort to failure reproduction on available HSM systems and tracing the failure through intense software analysis. While this is time-consuming, it also does not guarantee that the failure scenario can be captured as the inputs and the failure environment may be difficult to replicate. Hence there was the need to have a viable mechanism to capture such system failures and assist in easy debugging at least at a primary level. The aim of the thesis work was to establish an optimized and secure method to debug the HSM. This was broadly achieved in two parts, one as a logging framework which could securely capture software events or data, store, and decode them as needed, and the second by implementing a kernel crash dump capture mechanism to ensure system information is not lost also during a software crash. The requirements put forth are further detailed out in the design section of the thesis document.

The programming language used for the development of the solution was C which is ideal for embedded system middleware development as it uses fewer resources due to its small footprint and provides optimized instructions while enabling close control over the hardware. As the primary requirement for the real-time controllers are to finish execution well within the predetermined process time in the CPU, both these factors are crucial for automotive embedded controllers as they help achieve the memory and performance requirements such systems put forth. It is also highly portable and standardized making it easy to write code that can run on different hardware.

1.2 Author's contribution

The author worked on the design, analysis, and implementation of a secure debugging solution for the HSM as part of the Zentur (Security) Team at Elektrobit Automotive Finland Oy, under the supervision of the team's Senior Expert Juha Mäki-Asiala. The author was also responsible for developing a crash dump capture mechanism that would aid the developers in understanding the potential reason for software crashes by analyzing the software parameters during their occurrence. It also involved the development of an easy-to-use decoding method for the logged information and to identify an open-source crash dump retrieval mechanism. As the AUTOSAR framework encompasses several software modules which possess functionalities for data logging and

retrieval, there was also the need to analyze certain modules within the automotive framework to check its compatibility and possible adoption into the HSM code bank.

Respecting the company policy and with regards to responsible disclosure of information, some details relating to functional code, hardware, system traces, and detailed component description cannot be disclosed in this public document. But the author and the supervisors believe that the information provided is sufficient to capture the idea of embedded instrumentation, its overall architecture with respect to the HSM use case and the importance of achieving the same in a complex system.

2 Background

Embedded software development has received a lot of attention over the past years and a lot of focus has been on making the development process efficient. Several tools, most of them open source, are now available which can help achieve software logging and error tracing features. The first step towards developing a log and crash dump methodology is to learn about the available resources and analyze the essential elements in such systems.

2.1 Linux kernel logging

Several logging methods exist for Linux based embedded systems to retrieve the run-time information that would help evaluate the broad functionality of the software system. While being useful during the development phase it is also very helpful in the deployment phase where the existence of a 'system event and data' capture framework can help identify unexpected errors, while aiding to resolve them. But many a time, such information is generated using logging statements inserted within the functional code that spits out log messages as string data into a file. As studied in [5] the analysis of log data can aid in the diagnosis of security vulnerabilities, configuration issues, and critical system failures in a way leading to the overall evaluation of the module reliability. But the insertion of such logging statements into the software code in an ad-hoc manner represents poor logging practices and are the primary cause for data leakage in many embedded systems. [6] Also as pointed out in [6] verbose logging shall never be part of the compiled application but instead be only available as part of development work.

Studying the change of logging statements starting from its introduction into a code system and into its various software releases can help us identify the way developers make use of the logging data and see how such logging functions evolved over time. Publications such as [7] made a comprehensive list of studies on Linux kernel related logging and then tried to understand the practices in such systems. Based on the open-source data collected for their study they conclude that though the density of logging statements varies across kernel subsystems, the software driver and filesystem implementations are generally the most logged subsystems with close to one log statement for every 27 lines of code. The authors further point out that modifications to logging code is made to improve log output quality and precision, which sometimes

comes as an afterthought. Based on the analysis of 900 change sets that target improvements of log code, the reasons for these changes were identified as unclear messages, logging incorrect variables and redundant log data.

In addition to the fact that common logging techniques involve the generation of messages that indicate the start or completion of certain tasks, they are targeted at applications that have enough memory for large log data storage and those which are not stringent on execution time. On the other hand, most logging methods targeting embedded systems store the log data to Flash memory by writing data as a file (such as FatFS). Since logs are created frequently, this takes a hit on the execution time. To better understand the embedded system logging methodologies and their general structure, various logging tools were studied in detail. Some of these tools that differ with respect to the underlying platform, targeted programming language, popularity in the industry and so on are selected and presented:

2.1.1 syslog

As a standard logging protocol for Unix based operating systems, syslog facilitates the transfer of log messages to a central server or log file for application and system processes. Its detailed information capture possibility and pervasiveness allow for system analysis and diagnostic tasks like performance checks, analyzing security threats, and software crashes. Due to its simplicity and flexible nature, it is widely accepted for Unix-like systems. [8] Syslog includes a logging library or module that needs to be integrated into the application software which is to be traced, to help generate the log messages and transfer them to the output location. This forms a client-server architecture when operating over a network. The log originator passes information in the form such as text, value, and/or log level to a syslog function which would then create the logs in a specified format. The message sent from the originator is added with other information such as process ID, IP address or device ID and timestamp when processing. The interaction with syslog built-in libraries occurs through the inclusion of the syslog header file in the application software followed by the use of *openlog()*, *syslog()* and *closelog()* library functions. A syslog daemon (background process or service) also known as `syslogd` acts as the central system to process and route logging information. It runs in the background awaiting messages on a specified port and follows a defined logging structure as specified by the syslog protocol. Communication with the syslog daemon and

the application code occurs via the syslog library. [9] There are various syslog-based tools that provide additional features for software analysis like `rsyslog` or `syslog-ng`.

2.1.2 barectf

Targeted at C/C++ applications on bare metal or embedded systems, the barectf (bare metal and Common Trace Format) logging tool is a lightweight open-source library designed with flexibility and scalability in mind, well suited for general-purpose applications. It records information regarding program execution in a specialized binary format that can be extracted by an external tool (such as Babeltrace 2) to aid in system analysis. It reads data specified in a system configuration file to generate a metadata file (consisting of parameters such as datatypes, data size, comments and so on) and a source code file (ANSI C type) which must be compiled with the other application files to create the executable. The implementations in this source file can be accessed from the application code, after including the respective header files, to produce the CTF log data streams during program execution. Regarding the use of system resources, barectf can be employed to log events and trace points in a relatively efficient and optimized manner. As it uses a simple Domain Specific Language (DSL) to implement the data model it remains relatively easy to create and modify. With python extensions, CLI based access and support for a wide variety of log formats including text outputs such as JSON and YAML, integrating barectf to the embedded development projects has become an attractive option.

The data flow in barectf as represented in Figure 2.1 begins with the yaml configuration file where the users can specify various parameters such as data types and structures to be logged. The main barectf generator tool takes the yaml configuration file as input and generates the necessary source code and header files, "barectf.c" and "barectf.h" respectively, based on the specified configuration. These files contain the functions required for initializing the logging system, creating records, and writing to an output stream. The output stream can include a file or network socket and is generally defined in the "platform.h" file. Integrating the generated code into the project and calling the functions provided by the barectf API enables the developers to log data from their application files. Compiling the generated files and the application files creates the executable which will create the CTF data streams during execution and the metadata file which will contain additional log information. [10]

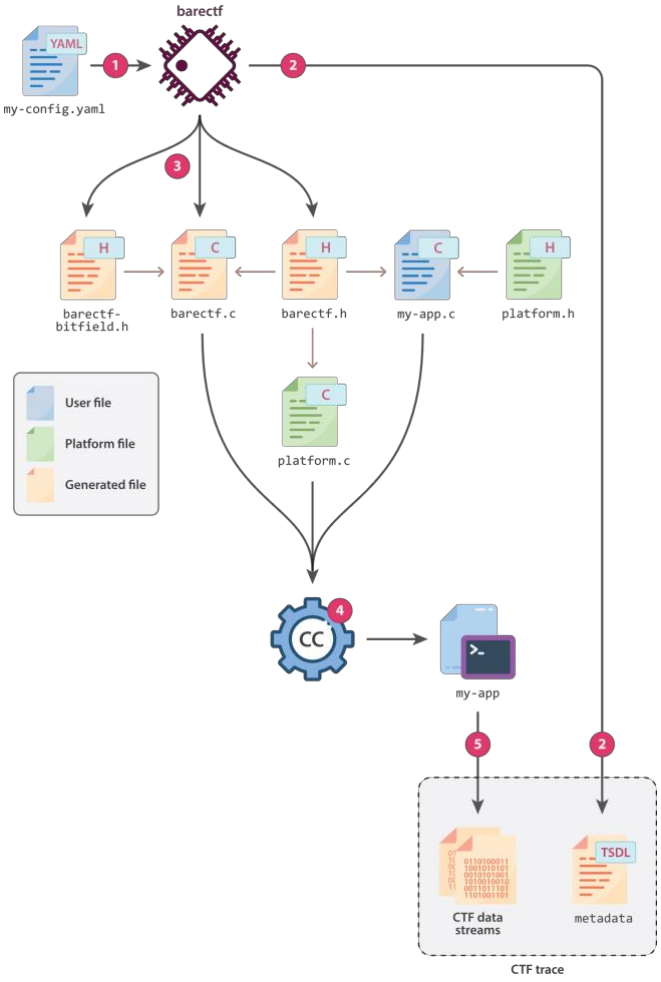


Figure 2.1. Representation of barectf data flow

2.1.3 LTTng

LTTng (Linux Trace Tool next generation) is an open-source tracing framework built upon its predecessor the LTT project, adding support for tracing both kernel space (events at kernel level) and user space (application-level tracing) in Linux systems. Using LTTng the target system is instrumented, and trace data is collected during execution to identify events that led to a system behavior/failure. [11] User-space tracing offers the possibility of instrumenting and tracing application-level events, function calls, or operations while kernel-space tracing involves interrupts, system calls, and other kernel-specific activities.

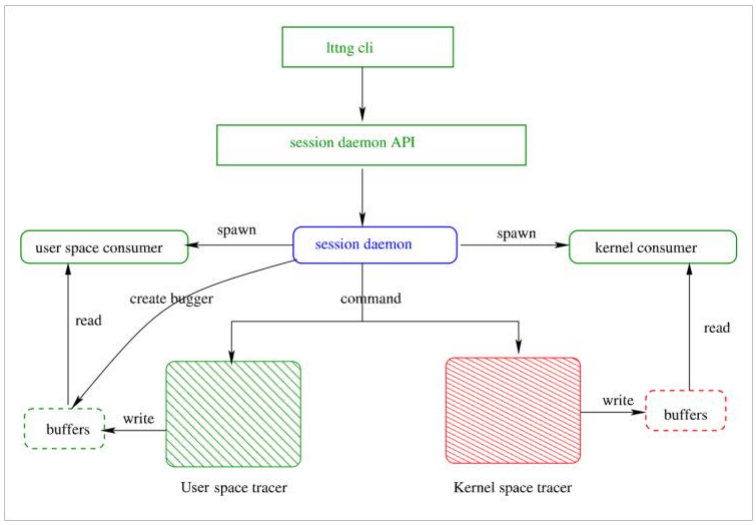


Figure 2.2. Program flow in LTTng

The code flow in LTTng as represented in Figure 2.2 starts by invoking a session daemon (a background process at user session level) that acts as a meeting point for all tracing components. Communication between the user and session daemon occurs via the CLI and includes the creation of sessions, enabling events, the command to start tracing etc. Static trace points need to be added to the application for user-space tracing while kernel-tracking is conducted by pre-instrumenting the kernel at the source level (modifying kernel source code to add hooks or instrumentation points). But this also comes as an overhead that kernel modifications are required to enable tracing which may introduce compatibility issues. [12] Above all its configuration and use can be fairly complex, and the effort spent may not be justified with respect to the features available.

2.1.4 Log4cxx

Log4cxx is a powerful open-source logging framework licensed under Apache License and is designed primarily for C++ applications. Though not directly targeted for C language, it can give insight into how logging frameworks are developed for other environments. Log4cxx provides features such as multiple log level support, log message formatting, driving output to the console, writing to file or even remote destinations. Like other tools, it also requires the placement of log statements throughout the application code but allows dynamic log-level changes and other adjustments at runtime through various logging configurations. It is built based on the Apache log4j (Java targeted logging framework) that uses Apache Portable Runtime (APR), which can create and maintain software libraries that provides an API for developers to use with consistent

behavior regardless of the underlying platform code. The framework extends support for tracing C-based applications with similar log4cxx C APIs.

The log4cxx architecture is composed of three main elements: loggers, appenders, and layout. These elements together achieve the requirements of a logging system. Loggers are part of the framework which is responsible for generating the log messages by providing the interface for application-level code to invoke them with the relevant data. An important feature available here is that loggers can be configured to inherit their parent logger settings, making it easier for log placement and creating the possibility of controlling a branch of log statements from one central location. Appenders decide the location where log messages are appended or sent to like a file, remote location, or console. It is also possible to print the log outputs to multiple locations if needed. And finally, layout determines the structure of the log message, that is how they are represented at the output stage such as pattern layout (using conversion pattern like %r [%t] %%-5p %c - %m%n), HTML table, XML, etc. With such configuration possibilities and features, though it facilitates effective debugging and application monitoring capabilities, it becomes complex to learn and apply the framework effectively. [13]

2.2 Linux kernel crash dump

Any software program, despite best efforts from developer, can crash due to several factors including input combinations that may be unaccounted for, an unseen execution environment or plain software errors that slipped the test and verification stages. In such a scenario the kernel of the operating system, which handles the interaction between the hardware and software components, can store some important information available to it to facilitate the investigation of the software crash. This is known as the core-dump or crash-dump as the kernel is dumping the data available at the hardware level such as the register contents into RAM, NVM or Flash memory. This task is usually triggered automatically by the bootloader or system firmware for any predefined system events which can be on the software side (such as kernel panic or application crash) or on the hardware side (such as hardware malfunctions or power failure). Core dump contents indicate a snapshot of the software program at the occurrence of the crash and analyzing it can help identify the part of code that was under execution when the crash occurred. Hence the existence of the core dump feature is a necessity for efficient system debugging.

Like the kernel log, the core dump must also be limited in size to easily fit inside the available system memory, while storing sufficient information for identifying the source of program crash. Information in the core dump generally include controller register contents which can point out the state of the system during the crash, contents of the RAM which can indicate if there was buffer overflow or leaks, call stack that will refer to the function which caused the crash, information about the operating system and application such as version or variants.

Capturing system information when software crash occurs is achieved through the effective use of exception (software/hardware malfunctions that lead to undefined system behavior) handler functions in embedded systems. Though manufacturers implement exception capture and exception priority setting for some predefined exception types, the duty of handling exceptions and saving relevant data resides with the developers. This is when tools that help access system data such as function call sequence, microcontroller register data or timer data become important. This is usually achieved with the help of debugging tools specific to the environment and hardware. Some of the tools that enable system analysis, which can also be extended to study software crashes are discussed:

- **GDB** (primarily software debugger): GNU Debugger (GDB) is a command line tool developed as free software to aid effective debugging and software program analysis primarily targeting C and C++ code. It provides extensive system tracing and altering capabilities during execution for the user, including unrestricted access to functions and internal variables. It also offers possibilities such as remote debugging over a network. [14] The crash analysis is made possible through its memory examination feature which allows developers to inspect the contents of a memory region such as stack or heap data. Debugging a hardware platform with GCC is possible using debug interfaces such as JTAG or SWD (see section 3.1.2) and additional configuration tools. Another popular debugger employed on C/C++ based applications is the LLDB debugger made available as part of the LLVM toolchain for ARM. It also provides a CLI based analysis for both the kernel space and user space.
- **Segger J-link** (hardware debugger): A popular hardware debugging probe used for tracing and debugging microcontrollers to analyze software and firmware components in the system. It also requires physical connection to the

supported controller for crash analysis using debug interfaces. It has several attractive features such as real-time trace which allows developers to study function calls and interrupts, high speed data transfer facilities through USB or WiFi, support across various OS platforms and integration with popular IDEs like Eclipse. Hardware debugging goes beyond software analysis in that it can allow developer to inspect hardware states, real-time register updates, hardware registers and timing data. [15]

- **Trace32** (software and hardware debugger): Software and hardware (via debug probe) debugging platform that supports a variety of controllers including Infineon TriCore 32-bit AURIX automotive microcontroller and achieves advanced debugging capabilities using debug interfaces connected to hardware (JTAG or SWD). [16]. It has a feature-rich software environment for the analysis of target systems and enables deep real-time analysis of Infineon hardware, but it requires a paid license.

2.3 AUTOSAR and its logging modules

Automotive Open Software Architecture (AUTOSAR) introduces a standardized development format for automotive software and electronic parts by laying out functional specifications and defined application interfaces that must be adhered to by all parties involved in the development. This heavily reduces the research and development cost for these basic modules and reduces the dependency of the Original Equipment Manufacturer (OEM) on a single part supplier, as the products from different vendors can be easily integrated to the main system due to their pre-defined interfaces. [17] This helps reduce monopoly among equipment suppliers. The AUTOSAR has a layered structure which provides a mechanism for software and hardware independence. There are three main divisions present in the architecture as shown in Figure 2.3 [18]:

- Application layer: It forms the topmost layer of the architecture and is hardware independent. It is an encapsulation of the application software that are implemented in the ECU. This is where the main algorithm for an ECU function is implemented, for example anti-lock braking or traction control.

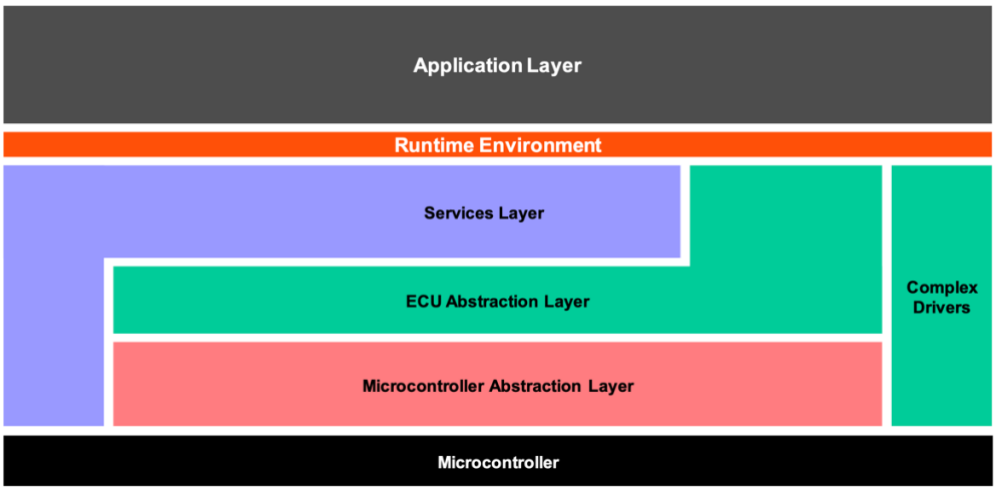


Figure 2.3. AUTOSAR layered structure

- Runtime Environment (RTE): It is a software abstraction layer that enables hardware-independent development of Software Components (SWC) in the application layer without specific knowledge of the hardware used. It forms an interface for information exchange between the top and the bottom layers, the application layer and basic software layer respectively.
- Basic software (BSW): This hardware-dependent layer is closest to the microcontroller and is further divided as Services layer, ECU abstraction and Microcontroller abstraction.

A better overview of the layered structure is represented in Figure 2.4. [18] As is visible from the figure the Basic software layer includes several functionalities necessary for communication and control of the hardware with the SWCs where the main features are implemented. The Hardware Security Module functions which form the focus of this thesis, are implemented as part of the Crypto stack in AUTOSAR which again is subdivided into the following three layers:

- Crypto Services: includes for example the Crypto Service Manager which handles the cryptographic jobs, the Key Manager which manages the verification and storage of certificates and the Intrusion Detection System Manager (IDsM) which is responsible for handling security events reported by upper layers such as from a SWC or BSW layer.

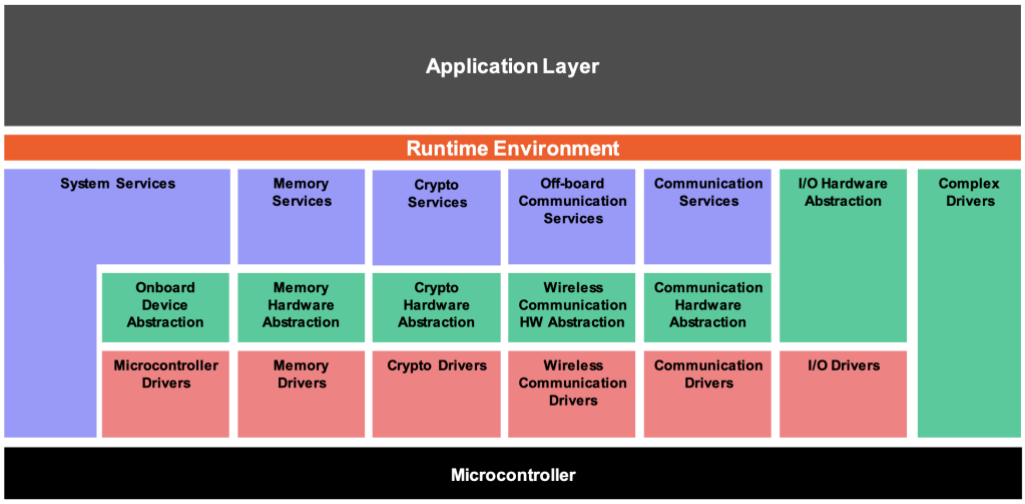


Figure 2.4. Detailed Basic Software layers

- Crypto Hardware Abstraction: employed to hide or abstract the physical location details of the encryption, decryption or hashing algorithms known as the cryptographic primitives. This helps to ensure that the cryptographic operations are performed securely despite the underlying hardware or software functionalities.
- Crypto Drivers: software programs that aid upper layers to communicate with the on-chip cryptographic devices like HSM or SHE

Being a complex software system the AUTOSAR architecture also defines methods for finding and reporting errors. It can be achieved using software modules that help route errors generated during software execution and create detailed logs necessary for analysis. There exists a main module which is indented for this purpose as described in section 2.3.1 but there are also alternative methods within AUTOSAR which could help achieve partial logging functionality using certain parts of their module implementation.

2.3.1 Diagnostic Log and Trace Module

Diagnostic Log and Trace (DLT) is a standardized protocol implemented as a module part of the communication service layer in AUTOSAR architecture (refer Figure 2.4) that targets to collects log messages from various ECUs, process, store and send the log data via communication bus. It is the main module for logging in the automotive standard and was originally implemented for collecting errors and log data from SWCs in the application layer. It is a complex module with detailed information to be included

as part of each data packet, like the ECU ID, data counters, timestamp with nanosecond resolution etc. It is also required to provide several configuration possibilities such as verbose and non-verbose logging, variable data length, data compression feature etc. as defined by AUTOSAR Log and Trace protocol specification. It also operates in a client-server architecture.

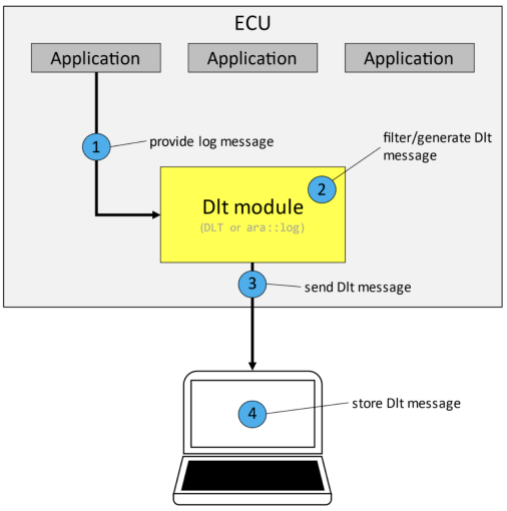


Figure 2.5. The flow of control when logging with DLT

A basic logging method with DLT is shown in Figure 2.5 [19] where the SWC (application) sends the log message that includes details such as ECU ID, Application ID, and so on embedded as a header in the data packet along with the main data to log. The DLT module generates the log message based on the pre-set configuration and sends the data via the communication bus to an external client (or internal modules like NVRAM Manager) that collects and stores it to a defined storage location such as external memory. It also supports other modes of operation such as tracing mode (real-time capture of function calls, variable values etc. rather than condition/even based as in logging) which provides a comprehensive view of system behavior and runtime configuration allowing dynamic DLT settings changes while in operation, with the help of an external client. [19]

2.4 Hardware Platform: Infineon TriCore AURIX

Implementing data retrieval mechanism such as log and crash dumps require good understanding of the processor architecture and functionalities. Security software stack from Elektrobit is implemented into commercially available automotive controllers with built-in HSM. A few examples of such controllers include Infineon AURIX TC2, AURIX TC3 and STMicroelectronics SPC platforms. Taking into consideration the short duration

of the thesis work and that the latest development within the team focused on the TC3xx hardware (where xx is some version number), the same was selected for the initial analysis and implementation of the log and crash dump module. As the underlying architecture of HSM is similar for the TC2xx in comparison to TC3xx, the logging feature could be easily extended to TC2xx variant with very few modifications. For SPC hardware, a brief comparison is drawn in section 3 to the processor architecture to outline the changes that may be needed. The following section describes the necessary background information only for the targeted hardware platform.

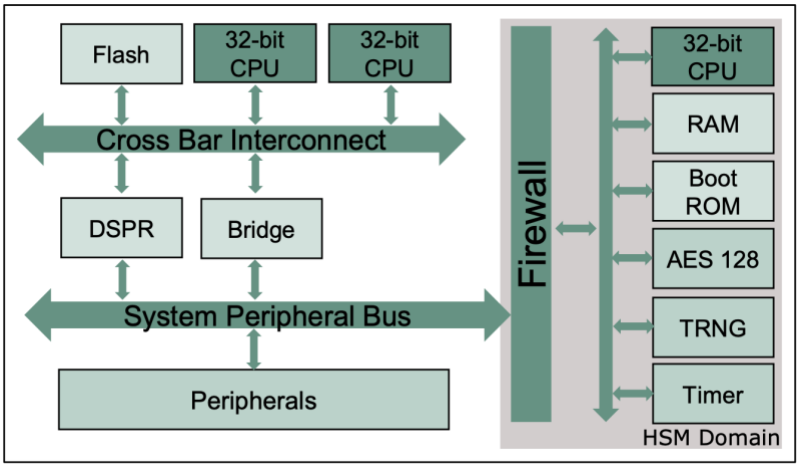


Figure 2.6. Block diagram of HSM on Infineon AURIX TC3xx

AURIX family of microcontrollers are called TriCore as it integrates 3 independent processor cores on a single chip and implements a sophisticated architecture combining DSP and microcontroller RISC architectures. But in TC3xx hardware, the HSM is built on a dedicated ARM Coretx-M3 core and includes its own peripherals for secure and efficient functioning. Other application software run on separate TriCore architecture CPUs with their own peripherals to form the Host system. To isolate and safeguard critical data, the HSM memory is strictly prohibited access from all components external to HSM, including the Host processor. On the contrary, HSM has access to all memory and peripherals on the platform including the Host memory. Figure 2.6 shows the block diagram of TC3xx hardware to highlight the separation between HSM and Host system through the firewall. Like RAM, the flash memory also follows a separation between Host and HSM to provide a secure external storage location for cryptographic keys and facilitates fast data writing and retrieval. [20] The next section provides more details on the HSM core.

2.4.1 HSM core: ARM Cortex-M3

The ARM Cortex-M3 processor is delivered as a synthesizable RTL and the chip manufacturer Infineon has the responsibility to configure it to produce a hard macrocell while integrating RAM and required peripherals. On the code level, the processor offers two options: Privileged code which has access to all processor resources and Unprivileged code which is restricted from accessing certain resources in the processor. The processor also has two modes of operation which are Thread mode and Handler mode. Thread mode is entered after a processor reset or when the system returns from an exception. This forms the normal operation phase. Code which is executed during Thread mode can be Privileged or Unprivileged. Alternatively, the processor enters Handler mode while it handles an exception. The code executed during this mode is always Privileged to allow access to any system information that may be required for exception handling. The processor also supports two operating states, the Thumb instruction state, and the Debug state. Thumb instructions are the 16-bit length instruction set which forms a smaller part of the commonly used ARM instructions which are 32-bit sized. Thumb instruction state is the normal processor execution state where 16-bit or 32-bit halfword aligned Thumb instructions are executed. During Debug state the processor is at a halt which aids in debugging the system with breakpoints. [21]

ARM Cortex-M processors have 16 CPU registers which are each 32-bit in size. Their names and functionalities are as follows:

- General purpose registers: R0 to R12 – used to store generic data important for program execution. Out of these 13 registers, the registers R0 to R7 are freely accessible for all instructions that specifies general purpose registers while the registers R8 to R12 are accessible to 32-bit instructions but not 16-bit instructions.
- Stack pointer: R13 – contains the 32-bit address where the stack begins in memory, or it can be said to point to the top of the stack. It is word aligned (4 bytes) as it ignores writes to bits [1:0].
- Program counter (PC): R15 – contains the address of the next instruction to be executed. Instructions are always aligned to word or half-word, hence the LSB of the PC is always 0.

- Link register: R14 – it takes the return address from the PC when Branch and Link (BL) or Branch and Link with Exchange (BLX) instructions are executed. Simply put, it contains the return address of the last executed function in the code. It is employed to return the control to normal execution after exception handling.
- Program Status Register: XPSR – special purpose register used to convey the program status.

The Cortex-M implements two stacks, the Main stack pointed by `SP_main` (MSP) and the Process stack pointed by `SP_process` (PSP). Out of all the registers only the Stack Pointer is banked meaning the address of the SP can change based on the controller mode. The Handler mode always uses `SP_main` while the Thread mode can be configured to handle `SP_main` or `SP_process` which is selectable through the SPSEL bit of the special purpose CONTROL register. [21]

2.4.2 Exception handling in ARM-Cortex M3

Exceptions are defined as events that interrupt the normal program flow of control, usually from within the processor, for example memory access violation errors or software-generated exceptions. They are used to report critical errors in the system and for error handling. Interrupts on the other hand are events (that also interrupt program flow) generated outside the processor, from external sources, for example interrupts from hardware devices like timers. Upon generation of an exception based on the processed code the processor follows a sequence of steps starting from execution entry, exception handling, and exception return. The ARM processors include what are known as exception vectors which are a series of fixed memory locations that contain the addresses of the exception handler code blocks in the memory. The following exceptions are defined in ARMv7-M architecture [22]:

- **Reset**: A form of exception that terminates the current execution of the program and is unrecoverable. When de-asserted the program starts execution from an initial fixed point. It has the highest priority among all exceptions and is fixed to value -3.

- **Supervisor Call (SVC)**: Used to request access for operating system functions in User mode, that is when the user code requires system privilege access for actions such as file I/O [23]
- **Fault**: Cortex-M feature that detects illegal memory access and program behavior. They are classified as [24]:
    - *HardFault*: It is the default case exception and can be triggered when the handling of the exception itself faces an error or when other mechanisms are unable to process the exception. It has the next highest priority (fixed to '-1') after Non-Maskable Interrupts.
    - *MemManage*: Under the regions defined in Memory Management Unit, this exception checks for memory access errors. It has configurable priority.
    - *BusFault*: Identifies memory access errors during scenarios such as data read/write, instruction fetch, interrupt vector fetch, or register stacking on interrupt. The priority can be configured by the programmer. It is one of the frequently occurring exceptions during embedded development and technically includes the largest subset of faults in the system. It is subdivided into two types: Synchronous (Precise) – exceptions that occur immediately after bus transfer; and Asynchronous (Imprecise) – happens due to write buffering in processor design which causes the system pipeline to proceed to the execution of the next instruction before the bus response is received.
    - *UsageFault*: Identifies undefined instruction executions, load/store multiple operations accessing unaligned memory, divide-by-zero, and other unaligned memory accesses. It also has configurable priority.
- **Interrupt**: Generally used by other system components (including the software running on another processor in the system) to communicate critical information with the processor. The communication occurs with a defined format that includes an identifier, interrupt priority, and a memory address that indicates the entry point of the Interrupt Service Routine (ISR)

or the exception handler. [22] NMI has the second highest priority (of value '-2') in the system.

ARM Cortex-M architecture implements an Exception/Interrupt Vector Table (IVT) which is a defined location in the memory which contains the starting addresses of the code to be executed for various types of exception/interrupts the processor can handle, such as memory exceptions, timer interrupt and so on. When an exception is detected, the processor goes to the address of the corresponding exception vector location in memory and determines the address location of the exception handling code. The Exception/Interrupt Vector Table is generally programmed to the NVM during manufacturing and gets copied to a fixed location (as described by the manufacturer) in RAM during boot. It is usually copied to the beginning of the processor memory address space and remains read-only. The address of the EVT is stored in an Exception Vector Register (VTOR) which is referenced when an exception occurs. The exception handlers pointed to by the EVT are usually located in the firmware or OS of the controller. In Cortex-M3 the prioritization and handling of exceptions are performed through the Nested Vectored Interrupt Controller (NVIC) which achieves low-latency exception and interrupt handling, provides System Control Registers, and controls power management. [21]

2.4.3 Timer in ARM-Cortex M3

The ARM Cortex-M3 processor provides a 24-bit count-down SysTick timer which decrements (or ticks) at a defined rate from a preloaded value in its registers. The timer has 4 registers available in the ARM core memory dedicated to storing values for its use and all of them require privileged access. [25] These are:

- SysTick Control and Status Register (SYST_CSR): configures the functionality of the timer like enabling, clock source selection etc.
- SysTick Reload Value Register (SYST_RVR): specifies the value to be reloaded to timer after it has counted down to 0. This becomes the start value for the timer in the new cycle.
- SysTick Current Value Register (SYST_CVR): holds the present value of the timer which is in operation. The SYST_RVR value is copied into this register when the timer reaches 0.

- SysTick Calibration Value Register (SYST_CALIB): defines the calibration properties of the timer.

2.5 Development environment

Developing any embedded software module requires good development tools and hardware interfaces. The following sections give a brief outline of the tools used for the development of the HSM logging and crash dump module and their respective benefits for this thesis work:

2.5.1 Compilers

GNU Compiler Collection (GCC) is one of the most popular free open-source compilers for C language-based embedded software development owing to its support for a wide range of CPU architectures, efficiency, compact output, and code optimization possibilities. It is currently based on the C standard ISO/IES 9899:2018 which was published in 2018 but prepared in 2017 (hence called the C17 standard). GCC can generate executable code on variety of platforms including Windows, Linux, macOS, and other embedded OS, and support different CPU architectures like x86, ARM, PowerPC etc. It also provides the possibility to debug the compiled code by including debugging information in the binary. The reason for the success of GCC is attributed to its structure that separates the machine-dependent program parts from the machine-independent part. Due to the listed benefits, its strong development community, and its popularity in the industry this compiler was the first choice for HSM development. [26]

2.5.2 Make build system

Make is a build automation tool that orchestrates the compilation process by selectively including and defining dependencies between various files using preset commands and then invokes the compiler (E.g.: GCC) to build the executable file. It is a popular build tool for development based on languages such as C, C++, and Java though it can be used for compiling any programming language file. The rules and dependencies are specified in a plain text file named `Makefile` which contains the name of files to be built which are called targets, the files to be included as dependencies for the target and the instructions for building the target system. Executing the command reads this defined `Makefile` and checks each target and its associated dependencies to determine whether they need to be rebuilt. If one of the dependency files has been modified recently

compared to the target, or if the target itself does not exist, then the associated `Makefile` commands are executed to rebuild the target. However, no commands are executed if the target is already up to date. This has the advantage that incremental builds are possible which reduces build time especially for projects with large number of files and dependencies. It also offers the option to perform parallel builds for targets that do not have interdependencies, which is useful for multicore systems. [27]

2.5.3 Operating system

The HSM hardware runs on Nanook (Nano Operating System Kernel) which is an efficient and lightweight preemptive multitask scheduler designed for 8 to 32-bit microcontrollers. It is an Elektrobit proprietary OS ideal for resource-constrained applications because it focuses on achieving functionalities like low memory footprint, efficient task scheduling, and fast boot times by sacrificing some extended features offered by other OS. Due to this it has a predictable and deterministic nature making it ideal for real-time applications such as automotive controllers. The Nano OS uses a make file build system to simplify cross-platform compilation known as Nano Build. [28] Being a lightweight implementation it does not feature any crash dump storage methods nor come with any logging framework like syslog pre-implemented. This introduces the need for the HSM log and crash dump module.

2.5.4 Code editors

Code editors provide developers with an environment (often interactive) to write and edit code which enables easier programming using autocompletion helpers, syntax checks, syntax highlights and so on. This project involved the use of first the Eclipse code editor for development in C due to its highly optimized C/C++ indexing feature and file search mechanism, and second the Visual Studio Code for Python-based development due to good extension availability that adds a whole set of features and the option to run the code on the terminal within the editor application.

2.5.5 Trace32

As described in section 2.2 the Trace32 tool is a hardware and software debugger with support for the Infineon Tri-core microcontrollers. It offers debug functionalities, analysis tools, and interfaces to hardware as software components. It has a GUI with all these features well integrated making it easy and effective in line-by-line debugging.

When used along with debug probes (these facilitate the communication and control to hardware) and connected to target hardware via debug interfaces like JTAG, they can be used for real-time hardware-assisted debugging. Lauterbach Power Debug Interface is the common hardware available with Trace32 having debug interfaces built in and comes with the connector cable for the specific hardware as shown in Figure 2.7. [16] The tracing tool runs on the personal computer or workstation and communicates with the debug hardware via USB.

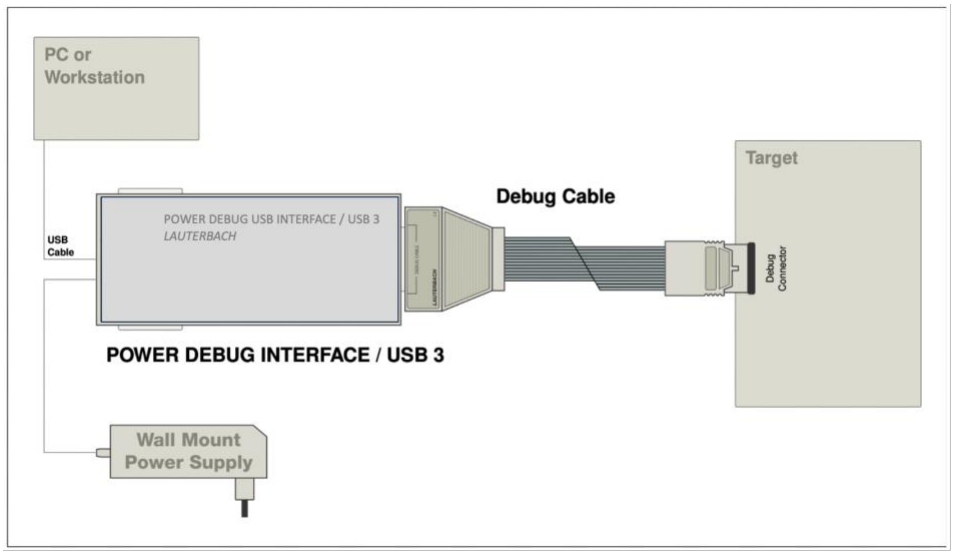


Figure 2.7. Debug setup when using Trace32

2.5.6 JTAG standard

The Joint Test Action Group (JTAG) standard is the name used for the IEEE 1149.1 standard which is an efficient protocol designed for testing and debugging of integrated circuits and logic devices on a Printed Circuit Board (PCB). The IEEE standard is titled the Standard Test Access Port and Boundary Scan Architecture for Test Access Ports (TAP). JTAG supports two main functionalities, one being boundary scan testing, a method used to check and verify physical connections between various components on a PCB and the second being debug access, a method used to gain access to the internals of a chip thereby making its system resources and functions available to the user and modifiable. [29] The debugging functionality is achieved by laying down a specification towards having a connector/interface on board which allows external devices to access ICs on the PCB. For ARM microcontrollers there exist two types of JTAG debug interfaces, ARM JTAG 20-pin connector and 10-pin connector, each useful for specific applications. Detailed pinout and characteristics are documented in [30].

2.5.7 Other tools and interfaces

The following tools were necessary and important for the development of the log and crash dump module, but they were used with limited scope:

- Git: a popular distributed version control system used for effective collaboration during software development process. It keeps track of file changes while providing decentralized architecture allowing developers to create copies (branches) of a project to work simultaneously but without interdependency. All development files in HSM are stored in Git repositories. Whenever changes are to be made, a copy of it is retrieved (checked out), modifications are made, and new changes are merged to the main branch (master) after code review (via pull requests).
- Cygwin: the target hardware is a Unix-based operating system while the workstations that the developers use run on Windows OS, the Cygwin tool is used to emulate a Unix-like environment and provide CLI based interactions allowing to run Linux and Unix applications.
- W&T USB redirector: with a limited number of hardware devices and the difficulty in physically accessing the target hardware each time for testing, a remote access to the hardware setup is preferred. The W&T USB redirector tool developed by Wiesemann & Theis GmbH allows redirecting USB devices over a network utilizing the client-server architecture (requires client and server-side software installations).

3 Design methodology

The following sections discuss the need to develop a log and crash dump framework from ground up, the design steps and the design considerations involved.

3.1 Advantages of the new debug solution

Using a logging tool integrated into the software provides an advantage over software or hardware debugging tools as they reduce dependency on interfaces such as JTAG for communicating with the hardware and extracting system information for each debug session. Also, the analysis time is reduced with a logging tool as the system information is readily available instead of traversing the source code with a JTAG based debugger to determine the action at each instruction. Moreover, there is the possibility to capture faults during system execution than through fault reproduction when the customer reports a failure scenario. This is particularly important as it enables sporadic faults or complex environment-based faults to be captured which were not detected during testing.

As shown in section 2.1, several powerful and feature-rich logging tools are available which can provide ways to capture and trace complex embedded software. But since each of them target a particular type of application or is indented for specific hardware, they are often limited by certain elements in their design and usage when applied to new software environments. While they may still achieve the goal, their adoption into the software system could potentially have more drawbacks than advantages.

For example, syslog is made available as part of the Unix-like operating system and hence may reduce the integration efforts to achieve the data logging setup. But it lacks a standardized logging structure. This introduces a limitation that once the logs are planted into the application it becomes tightly coupled to the OS in a way that upgrading the OS, which may come with new syslog formats (due to its non-standardized nature), would demand effort in updating all the previously used log function calls. [31] Also, it has limited scalability as large number of logs can cause high strain on the logging infrastructure. Similarly, the disadvantage of barectf is that it requires manual configuration and setting up of log format into the YAML configuration file which is time-consuming and prone to misconfigurations. LTTng and Log4cxx are feature-rich,

but their integration and configuration of parameters is a complex process and the learning curve for the effective use of these tools can be steep. DLT is a powerful tool for automotive software development but requires considerable effort to integrate and maintain the framework. Above all, most of these tools have several features that are unnecessary for specialized embedded software modules like HSM with a narrow field of interaction, target real-time operation, and run on controllers with limited resources.

For instance, most of these tools are designed to support the logging of different datatypes such as signed and unsigned integer values, floating point values, null-terminated string, and so on over a range of data sizes. But under HSM software all the implementation-related data are unsigned integer values and all of them are smaller than 32-bits in size. Additionally, it is not advisable from a security perspective to store string values as log messages in a system-safeguarding module such as HSM because sensitive information may be leaked accidentally into unsecured memory locations due to improper use of the log messages by developers (as discussed in section 2.1) or by passing critical information such as cryptographic keys into the log data.

Feature	DLT	barectf	LTTng
Target	High level logging	Low level logging	High level
Integration and configuration effort	High	Medium	High
Usage and popularity	Widely used (automotive)	Less usage (general purpose)	Widely used (Linux based systems)
Standardized	Yes (AUTOSAR)	No	No
Built-in security	Encryption and authentication	None	None
Resource usage	Moderate	Low (static config)	High (dynamic config)
Flexibility	Low	High	High
Payload representation	Binary format	CTF format (modifiable)	Binary format
Data compression	Available	Not available	Available
Graphical output	Available (DLT viewer)	Not available	Available (Trace Compass)

Table 3.1. Comparison of popular embedded software log tools

Moreover, many of these logging tools do not offer a standalone mechanism to read and interpret the system data for analysis during a software crash. They either require additional tools for crash dump creation or for symbolizing these values. Table 3.1 shows a comparison drawn between popular embedded software logging tools selected based on their applicability to HSM software systems. As indicated, each of them has qualities that may be ideal for certain embedded applications. But considering the integration and configuration effort, learning curve, resource wastage from unused features and the need

to achieve secure logging with limited effort, it was established that an implementation from scratch targeting HSM would serve the purpose better. It could also include an easy method to store and process system data during crash. Thus, the target was to achieve an efficient and lightweight crash dump and logging framework that derives positive traits from the listed mechanisms but built specifically for the use case at hand.

To summarize, the following are the advantages of the proposed log-based debugging method with respect to the existing methods:

- Achieves faster analysis, reduced hardware dependency, the possibility to better capture runtime faults, and ease of use in comparison to JTAG-based software and hardware debuggers.
- Implements an easy-to-use, lightweight, optimized, crash data capture integrated, secure debugging with respect to existing logging methods.

3.2 Design steps

With respect to the aforementioned target, the following steps were identified as necessary to implement the HSM log and crash dump framework:

3.2.1 Choosing log components and format

The first step is to identify what data need to be captured to debug an embedded software and then define the logging format that establishes a way to structure the log.

3.2.1.1 Log components

Absolutely necessary and good to have (optional) components for the logging framework are:

- Log-level: it indicates the priority of the log such as failure, error, warning, information, etc.
- Log-data: it is the information to be captured indicating the status of the system at the logged instant. It represents a value.
- Keyword: it is the keyword that produces the log data that is captured for system analysis. It represents a variable or function name.
- File name: indicates the file where the log is created from; this helps the user to debug the system faster.

- Line number: indicates the line in the file where the log was called.
- Contextual information (optional): some information at a limited scale can be captured to provide more context to log data, but without compromising security. For example, a timestamp, short comment, occurrence counter, etc.

As for the crash dump data, no special format is defined or is necessary as it only involves capturing the real time system data and displaying it to the tool users. Since these values already have a predefined size and structure, no encoding is necessary.

3.2.1.2 Log format: Tag-Length-Value encoding

It is important that the above-mentioned components of the log are captured and grouped in a specified format which is maintained across the entire system. This would make it readable and easier to process. To structure the log components the Tag-Length-Value (TLV) encoding scheme was selected as it is easier to construct and decode and is a popular encoding scheme which could mean wider decoding tool availability if needed.

Tag-Length-Value (also known as Type-Length-Value) is a type of encoding mechanism where the entire data is constructed in the specified format starting with a tag field, which can be used as a label for identification, followed by a length field that generally represents the length of the value part and finally the value field itself which contains the actual information. Several variations of the encoding scheme exist like the value field can itself contain a TLV, meaning several TLVs at various levels are encapsulated to form a large TLV, or the length part may not exist, and so on.

The TLV design may be achieved in two ways which are suitable in respective applications. One way was to define a structure for the TLV where the individual components, the tag, length, and value fields, can be defined within the structure. This is helpful in that it allocates sufficient memory space for each of the fields within the 32-bit memory organization of the controller RAM. For example, if the tag field is defined with an unsigned integer of 8 bits (`uint8`) while the length is defined as unsigned integer of 16 bits (`uint16`) then they are grouped together automatically in the memory. This is useful as it reduced the memory size of the code but has the limitation that the writing data to the memory location would then need to be processed via the structure object which will consume more time. The second method is to avoid the use of structure and instead define the fields separately. But in this case, for writing the data to the memory each of the fields must be converted to 32-bit integers through bit manipulation and then

added together to obtain the TLV encoded data. While this is faster it has the disadvantage that each of the fields will have to be adjusted to 32-bit values thus consuming more space in memory. Considering that the HSM has high performance requirements, the decision was to reduce logging execution time in exchange for slightly larger memory usage.

3.2.2 Identifying log interface

An interface must be available that the modules within HSM could use to create the log messages in the defined format and store them in a given location. This can be a simple function that collects the data passed from the respective module and transfers it to the processing algorithm that consolidates the data into TLV format.

3.2.3 Creating crash dump capture

System data must be captured during a software crash to help in crash analysis, but it begins with the task of identifying that a software crash has occurred in the first place and then retrieving data from those parts that are important for analysis. Automotive embedded controllers are complex systems with several peripherals and can contain a lot of information. Identifying system data that are relevant for analysis is an important task so that resources are not wasted in this data extraction process and unwanted information is not presented to the developers. Identification of a crash is already made possible in most modern microcontrollers with the help of exception handlers.

When an exception is detected in an ARM Cortex-M3 processor, to prevent the current state of the processor from being lost and to return to normal execution after exception handling, the processor always saves some important data such as CPU register values before proceeding to the exception handling code. The method by which this data is saved can vary across various processor architectures. For some ARM processors, various system registers can change based on the mode that the processor is currently in. These are known as banked registers and a banked register is strictly mapped to a user mode. When the mode changes, the banked register from the new mode replaces the current register. For example, registers R13, R14, and SPSR are banked in ARMv7-A processor architecture. During abort mode, the registers R13, R14, and SPSR are replaced by the registers R13_abt, R14_abt, and SPSR_abt. For such systems, care must be taken to ensure that the correct value of the register is being read, for example during logging or to report the instruction that causes a crash. But this is not the case for every processor. In ARM Cortex M3 architecture only the stack pointer which is represented in the register

R13 is banked while the following register contents are pushed to the stack in this exact order before exception handling: XPSR, Return address, LR, R12, R3, R2, R1, and R0. [21] Hence, extending this exception handling feature would help achieve the crash time data capture as it not only detects crashes but also offers a common platform to retrieve the system data for any exception.

3.2.4 Creating storage method

The next step is to identify and create a mechanism to store the generated log considering the system resources. This storage mechanism must be easy to write and read from and should achieve efficient data storage without causing buffer overflows.

3.2.4.1 Ring buffer storage

Defining a fixed number of memory locations for log storage is efficiently achieved using a ring buffer. Ring buffer implements the insertion and deletion of data in a circular fashion upon exceeding the defined buffer size and therefore is a better approach compared to linear memory for systems with limited memory. This is because they are well equipped to fit large amounts of data in the available memory locations through repeated data addition and removal while being less prone to buffer overflows in comparison to linear memory. As linear memory requires element shifting when incorporating data insertion or deletions, they cannot guarantee a fixed processing time for each execution. This is a major drawback especially for time-critical systems such as automotive ECUs and the limitation would only scale with repeated calls, such as with logging applications, which are called several times in one task cycle. They also offer a thread-safe operation (thread is a unit of execution within a process) where one thread can write data to the buffer while the other could read out the data from the buffer there by achieving inter-thread communication.

3.2.4.2 Secure log and trace

The next task is to decide where the Ring buffer must be located. The data that is passed from the logging functions need to be stored in a location that is easily but securely accessible. HSM memory would be a good choice to store the logged information as it would be secure and quickly accessible thereby avoiding overheads from frequent writes to memory location. But the following points need to be considered:

- Available HSM memory is smaller in comparison to the Host memory size.

- HSM requires sufficient resources to perform its complex cryptographic tasks and not retaining enough resources can have serious security implications.
- As HSM has access to Host memory, there already exists a way to write data from HSM to Host RAM via the communication module with minimal overheads, memory write locks, and error checks.
- Since HSM crashes also need to be analyzed, retrieving data from the HSM RAM would not be straightforward during crash. Having logs saved outside HSM can be helpful to prevent log data from being lost or corrupted.

After careful analysis of the above information, it was concluded that storage to Host memory would be the ideal design choice for the log and crash dump module. This would reduce the risk of affecting HSM performance and log data loss from software crashes. But on the other hand, it introduces an important factor regarding overall system security because Host memory is not a secured memory location. Thus, writing log data to Host memory could be a potential risk. Based on this, to avoid important information leakage the log memory should not contain any character data or plain text encoded information. This meant that even the log file name as mentioned in section 3.1.1 should not be written directly to the log storage location. Hence a secure log and trace solution must be introduced that can write a numerical parameter for file name when logging but can identify the actual file name where the logging was made when tracing the log.

3.2.5 Defining output method

After figuring out a way to store the log messages the next step is to retrieve them and to identify how the log should be displayed to the user. As mentioned in section 3.1.1.2 the usage of TLV would benefit in that it is easy to decode. Once the memory dump is made available the recreation of the log data is possible by first structuring the binary data in the right form and then processing it based on the byte positions defined in TLV encoding. Developing a decoding tool using Python was the choice considering the online community and resources, ease of availability of libraries to perform a complex task, good IDE availability etc.

3.2.6 Adding configurability

Finally, the framework should provide configuration options to increase efficiency and to better tailor it to the software build. Some log components in the log

interface should have the option to be skipped if needed, adding extra context information in the log should be a target for the processing function and not developer, the filtering of the log based on the log type should be possible and so on. This means that the framework should be flexible enough to some extent and make its use easy.

3.3 Design considerations

Automotive applications have stringent time constraints and their bill of material cost further limits the possibility of using more powerful hardware. To reduce the effect of the logging module on system resources and to ease its adoption into the software module the following design considerations were laid:

- It shall be possible to switch off logging and crash dump features: the framework shall have an enable and disable option which shall act as the master switch to turn off the entire functionality.
- Three log levels shall be supported, and it shall be selectable: the framework shall include 3 log-levels – Error, Warning, and Information. It must be possible to choose which of these shall be logged during compilation.
- Design must take into consideration the resource usage.
- Design must take into consideration the system security: no string data shall be stored in an unsecured location in system memory.
- For secure feature activation and data retrieval, a security certificate must be implemented for the module.
- The order of the logs must be identifiable: the position of the log in the log storage location must be based on its occurrence or there shall be a way to understand the order in which the log data were added to the log storage.

3.4 Design changes: ARM vs Power Architecture

Two prominent architectures need to be discussed in this section with regard to the two microcontroller cores used in the automotive security application mentioned here. The HSM development as pointed out revolves mainly around the Infineon AURIX platform with the ARM Cortex-M core which is implemented on ARM architecture and the STMicroelectronics SPC platform with the e200z core that follows the Power Architecture (referenced here as PA). The PA was originally developed through a

collaboration between Apple-IBM-Motorola. While both architectures are based on the Reduced Instruction Set Computing (RISC) design focusing on efficiency and code density, PA offers a larger range of performance levels from being designed for High Performance Computing (HPC), servers, automotive and complex embedded applications. [32] ARM architecture generally targets consumer electronics due to its power efficiency and smaller footprint. The Instruction Set Architecture (ISA) also differ between the two with each of them having their own advantages. ARM with 16/32-bit thumb instruction set provides parallel processing with efficient operations such as one cycle multiply and accumulate (MLA operation) while PA with 32-bit instructions offers advanced floating-point operations like decimal arithmetic. In addition, due to their popularity, tool development around both these architectures have evolved with tools such as GCC or Trace32 discussed here supporting both architectures. ARM cores achieve tighter integration with standard peripherals providing quicker interrupt handling and deeper debugging capabilities. But this limits the modification possibility for the embedded platform developers which then becomes an advantage when using PA. Ultimately the decision of selecting the type of core and in turn the microcontroller (platform) depends on several factors such as the application, requirements, capabilities of the hardware platform itself, cost, etc., and usually lies in the hands of the OEM.

The ARM architecture and PA have significant differences and so does the cores that implement them. To understand the design changes for log and crash dump module between the hardware, they will need to be compared. For better comparison to be made a specific version of these cores will need to be considered, for example, ARM Cortex-M3 from AURIX TC39 automotive controller and e200z4 from STMicroelectronics SPC58 automotive controller. Though both the cores follow 32-bit addressing they have several differences, but for this thesis work only their register configurations and exception handling are important. The Cortex-M3 has 16 32-bit registers (R0-R15) with 13 of them being general-purpose registers and the rest special registers. [21] The e200z4 core has 128 registers in total out of which 32 of them are 64-bit general purpose registers (R0-R31) and rest special registers, making it ideal for complex data processing. [33]

However, this difference in registers would not require major modification in the crash dump capture logic since both cores store important system data like LR and PC to stack before exception handling, making it possible for extraction. As for the logging module, these hardware differences do not pose any design issues at all as the framework

is entirely at the code level. Certainly, there are differences such as in clock speed, timers used for logging, the memory address usage, and so on but they are all at the code level and the basic architecture for the log and crash dump module remains the same. Hence, the overall design like the log components, TLV format, logging interface, ring buffer storage, etc. remains the same for PA as considered for ARM in the TC3xx platform.

4 Implementation and Result

Achieving the targeted functionality required the development of independent states such as log storage, file generation, interface creation, etc. defined in the design section and building upon them to reach the final implementation. The implementation can be broadly divided into two stages. One is the development of the log and crash dump module in the HSM side (encoding and decoding) and the second is the implementation of the security certificate and data interfaces on the ProxyAPI side. The overall structure of the log and crash dump framework that was implemented is represented in Figure 4.1 and more details are laid out in further subsections.

The files that are used for the development of the log and crash dump functionality are implemented in a separate component named HSMLogger in the HSM build repository. The Host first defines the size of the log buffer needed by specifying the number of log elements required for the system. This data is sent to the HSM via the ProxyAPI which acts as an interface between the Host and HSM hardware. As each log element requires 16 bytes of data, the ring buffer size in bytes will be the number of elements set by the Host multiplied by 16. The ring buffer parameters get defined during HSM initialization and exist in HSM memory, but the buffer elements are stored in the Host memory by mapping the address of the memory location to the ring buffer variable.

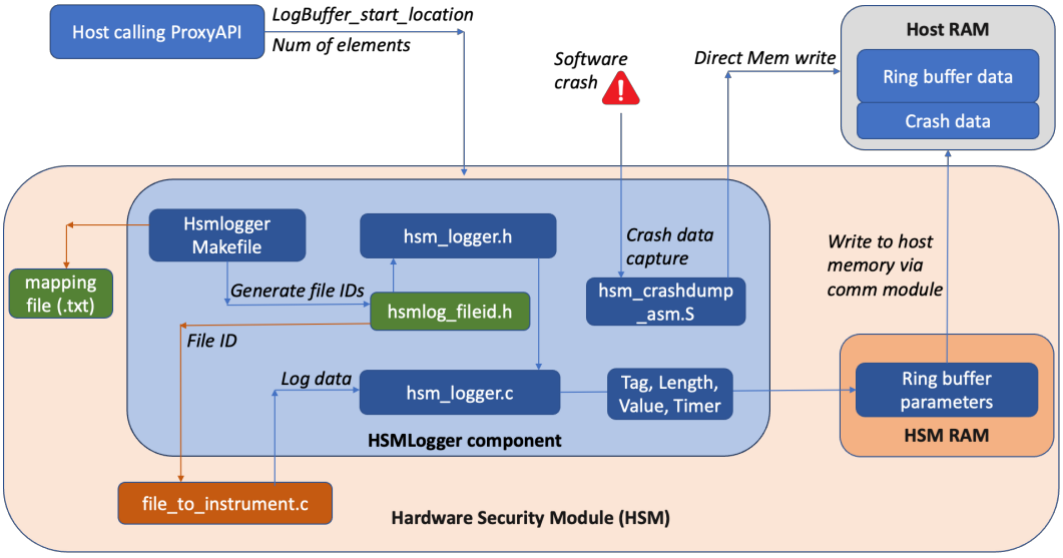


Figure 4.1. The overall structure of HSM logging framework

During compilation, the `Makefile` from the HSMLogger component searches through each source file to capture the entire line of code that calls the logging API and stores it to a mapping file (`<build_variant>_map.txt`) in the build folder. The make file then triggers the generation of a `hsmlog_fileid.h` header file where all the source files (from a selected part of the repository) get defined with a unique numerical identifier. The main functions that implement the logging system are part of the `hsm_logger.c` file and its corresponding declarations in `hsm_logger.h` file. The logging function requires three inputs from the file that wishes to log some data (`file_to_instrument.c`) which are: an identifier for the filename, the log level, and the data to be logged. The `file_to_instrument.c` can use its own FileID macro defined in the generated header file `hsmlog_fileid.h` as the identifier and pass the log level along with the data to log. The HSMLogger function then processes the data to structure it as TLV, adds additional information, and logs it to the Host memory using the memory write services offered by the communication module. The crash data is captured by the assembly code defined in `hsm_crashdump_asm.S` and it writes directly to the memory after the allocated space for the log data. This write should be direct and not via comm as there are no function executions after a software crash and the comm module in HSM may not function properly due to the crash. The following sections describe in detail the main functional blocks of the implementation.

4.1 Log creation and storage

The log and crash dump module in HSM broadly achieves two tasks, one is the data encoding process and the second is the data decoding process from the memory dump. For the first part, the aim is to capture and format the log data as needed, store it in a defined location, and ensure efficient handling of new log data. This is achieved by:

4.1.1 Implementing a secure log tracing method

The initial stage of implementing the logging module for the HSM middleware was to begin with the build environment itself. Typical to the recommended file structuring, most of the functional code resides in C source files while its corresponding declarations and variable definitions are recorded in the header files which are then included in the source file. The source files are compiled to generate a binary file using the Make build system which is common in Unix-based applications. The initial target

was to develop the tag for the logging data (to enable us to trace the log) which is referred to here as the FileID. One method that was explored was to use the built-in preprocessor macro __FILE__ that is supported by many of the C compilers, to return the file name as a character pointer. But this had the drawback that it would jeopardize the system security. This is because __FILE__ returns the name of the file where it is called as a string in memory (series of characters) which would make it directly readable through ASCII conversion, making the system information available to an attacker.

Thus, it was required to implement a method through which the developer could identify the file from where the log is returned but an external agent accessing the memory cannot. This was achieved by defining a compile-time constant for each of the files available in the build system using the #define macro. But to perform this task manually has overheads such that tracking every file addition/removal in a branch is difficult, defining a numerical value to each of the files in the branch would be time-consuming and manual definitions are prone to errors for example by assigning same value to multiple files. To resolve this a `Makefile` adaption was introduced where using shell commands all the '*.c' and '*.h' files present in the branch were associated with a numerical value. This definition itself was placed inside of a header file which is also included into the build system for compilation. The `Makefile` changes were implemented in a robust way which ensures that the code could be used for any repository or module with minimal updates.

In addition, the `Makefile` was also configured to generate a mapping file (saved to a build folder outside the HSMLogger component) that includes a consolidated list of all the file names that access the logging interface. The map file also documents the line number, the keyword passed and the API that was used for the call. This file is important because it helps the logging framework to function independently of the source code. Even when new changes are made to the source code in the repository, having the mapping file archived will help the developers to identify the logs in the code that were present at the time the project was compiled. This mapping file is also used as input for the decoding tool to determine the right file names and line numbers even when the actual project source code may have changed.

The following code snippet from the logging function `Makefile` represents the main part of code that generates the `hsm_logger_fileid.h` and mapping file. The C_FILES stores the path of all the *.c files in the search path '../' (meaning the previous

directory), excluding the ‘tests’ and ‘build’ folders. These identified C files are searched line by line for the keyword ‘HSMLOG_CAPTURE’ which is the logging function that the files-to-instrument could call to access the logging functionalities. Once found, the parameters in that line are read out using the numbers specified in the print command and the data is saved to the path which is stored in the variable FILE_PATH_MAP which forms the mapping file. After certain other modifications to the names in C_FILES, such as replacing the backslash character with three underscores (for easy identification), reading each line and assigning a unique numerical value to them, and adding the header file include-protection switch, each file name is taken and prefixed with the keyword ‘HSMLOG_’. This is finally written to the file represented by the variable FILE_PATH_FILEID by adding the ‘#define’ macro thus forming the generated file containing the FileIDs.

```
# Shell commands to generate mapping file
C_FILES := $(shell find ../ -type f -name "*.c" ! -path '*/build/*' ! -path
'*/tests/*' )
$(shell grep -n 'HSMLOG_CAPTURE' $(C_FILES) | awk -F'[(),]' '{print $$1,
",", $$4, ",", $$5}' > $(FILE_PATH_MAP))

...

# Main shell command to write FileIDs
$(shell $(foreach word,$(C_FILES),echo "#define HSMLOG_$(word) $(call
IDX,$(C_FILES),$(word))" \
>> $(FILE_PATH_FIELID) && echo "" >> $(FILE_PATH_FIELID);) )
```

The generated output of a sample mapping file is shown in Figure 4.2 and the generated FileIDs can be seen from Figure 4.3. The mapping file is not part of the compilation but can be archived along with the build executable files and is available only to authorized users such as the customer and developers.

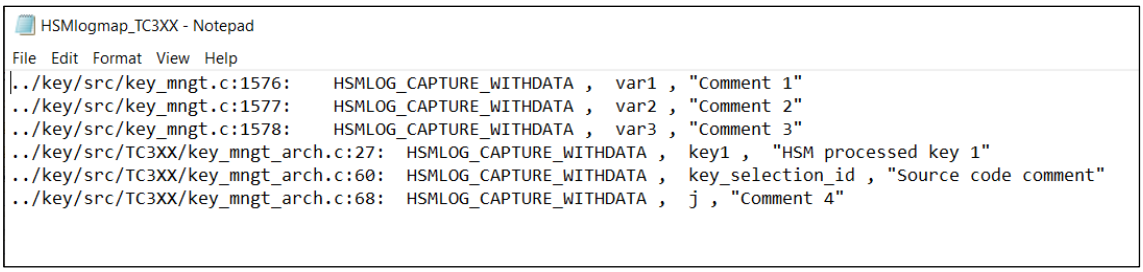


Figure 4.2. Contents of the generated mapping file

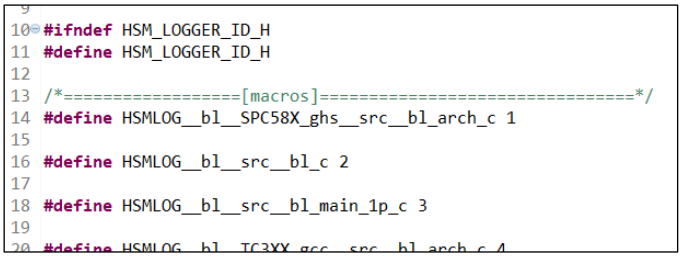


Figure 4.3. Generated file containing the FileIDs

4.1.2 Developing the encoding scheme

Tag-Length-Value encoded data is important for secure debugging as pointed out in section 3. The design was to have 32 bits designated to the Tag part which will contain the necessary information to identify the portion of the code where the logging occurred. 32 bits are further mapped for the Length section, where the first 16 bits specifies the occurrence of the logged data within the memory and the remaining 16 bits capture the length of the data passed. But the design was later updated to output the number of bits in the actual data instead of bytes to provide better detail. This creates the possibility for better scrutiny of the information part by comparing the bit value encoded in length to the actual one available at the time of decoding to ensure memory corruption has not taken place.

The TLV implementation for the HSM logging framework was established as follows, where one block of log data utilizes 16 bytes. The aim was to make the logged data easy to understand and identify for the developers while keeping the data secure. Hence, the use of character data for logging would not be ideal as the data would be easily readable for an attacker. Instead, an identification tag could be employed for the files during encoding. This is represented as File ID in Table 4.1. Designed memory organization of Tag in TLV which is a 12-bit (from MSB) data because the number of functional files in the repository would never exceed 0xFFF or 4095. The adjacent 16-bits are used to capture the line number of the logging function call and two bytes of data are mapped to it as the number of lines would never exceed 0xFFFF. The LSB would hold the log level for the logging function call which can be ‘Error’: signifies the capture of a functional issue, ‘Warning’: warns the possibility of an incorrect system behavior, and ‘Information’: which conveys the log data for tracking software flow. 4 bits are used for the log level to leave enough room for the implementation of other logging levels in case needed.

Bits 28-31	Bits 24-27	Bits 20-23	Bits 16-19	Bits 12-15	Bits 8-11	Bits 4-7	Bits 0-3
File ID	File ID	File ID	Line no:	Line no:	Line no:	Line no:	Log level

Table 4.1. Designed memory organization of Tag in TLV

Bits 28-31	Bits 24-27	Bits 20-23	Bits 16-19	Bits 12-15	Bits 8-11	Bits 4-7	Bits 0-3
Count	Count	Count	Count	Length	Length	Length	Length

Table 4.2. Designed memory organization of Length in TLV

The Length part of the TLV consists of a counter value that keeps track of the total number of log data that was captured in the system or indicates the actual order of occurrence of a log data with respect to others in memory. 16 bits of data is allocated for the counter value and another 16 bits are allocated to hold the length of the data passed into the log. Here the length in number of bits were used instead of bytes as it creates the possibility for better scrutiny (due to better resolution) of the value part by comparing the encoded bit length to the actual one available at the time of decoding, to in turn ensure that memory corruption has not taken place. These together form the 32-bit length part of the TLV as depicted in Table 4.2. The value part of the TLV is composed of any 32-bit data that can be passed to the logging function which would aid in understanding the system state based on its value. Hence any variables or function return can be passed to the logging function that might help to debug a system fault. In addition to the data, another 32 bits are reserved for the TLV block to capture the elapsed time in the system since HSM initialization. This would not only indicate the order of occurrence of the log data (as would the counter value), but it could also help identify the part of code where the program experiences delays or long execution time and if the program control has crashed. All together one block of TLV structure that corresponds to the data of one log element as represented in Table 4.3 is composed of four 32-bit memories or 16 bytes in total.

Byte 1-4	Byte 5-8	Byte 9-12	Byte 13-16
Tag	Length	Value	Timer

Table 4.3. Overall TLV structure for one log element

TLV data is created by adding the respective data values in the tag, length, and value fields after converting them to 32-bit values through binary rotation using the left shift operator ‘<<’, as it pads zeros on the right. The shifts required correspond to the position of the last bit of each of the fields in the TLV group. As the FileID corresponds

to the first 12 bits from the MSB, it is left shifted by 20 bits to form the 32-bit integer for addition. The line number occupying the next 16 bits is hence left shifted only by 4 bits and the two 32-bit integers formed are added with the one-bit log-level value to form the tag part.

The log levels as shown below are implemented as an enumeration to assign a numerical value, maintain order, and thereby the priority among the used levels. A compile time switch was created based on each of these log levels to enable the developers to control the type of logs they would like to capture, for example only the logs defined as error log-level will log when the corresponding switch is activated.

```
typedef enum
{
  Error,       /* HIGHEST PRIO: Errors during runtime */
  Warning,     /* Problems that inhibit proper functioning */
  Information  /* LOWEST PRIO */
}log_level;
```

The creation of the tag and length elements are achieved with simple bit manipulation as mentioned in section 3. The data part of the log does not require any modification as it would take up to 32 bits. The timer value is retrieved using a newly developed function that fetches the timer register content, just before writing the log to the ring buffer. The following code snippet represents the part where the same is achieved. The 'HSMLOG_' are macro variables that contain the bit position of each TLV element as a numerical value so that the bits can be shifted by the correct amount.

```
tag = (FileNameId << HSMLOG_FILEID_POS) + (LineNo << HSMLOG_LINENO_POS) +
(Level << HSMLOG_LOGLEVEL_POS);

length = (((LogRingBuffer_g.globalerrorcount) << HSMLOG_COUNTER_POS) +
passeddatalength);

hwtimer = Timer_GetTimerValue(TIMER_0);
```

4.1.3 Implementing log storage: Ring buffer

Considering that a large number of executions calls may be made to the logging function leading to a large amount of log data and that the logging framework must remain light, a circular buffer was used for log storage. It is a popular choice in real-time systems and audio processing where a large amount of data write occurs. It can continuously write data to a defined memory space and wraps it around to the beginning of the allocated memory upon reaching the end, overwriting the earlier data without the need to make

deletions. This saves time especially in applications with large number of write cycles. The circular buffer follows a First-In-First-Out (FIFO) strategy which ensures that the most recent data is always available. This can be an advantage during situations such as a software crash where the latest data in the buffer may point to the most recent function calls that may have caused the crash.

A series of design iterations were performed for the ring buffer implementation until the final version was reached. The overall structure of the ring buffer was defined using a structure that can group dissimilar datatypes together, in this case, integers of different sizes. Head, tail, and size are the control parameters of the ring buffer and the buffer data pointer points to the starting address of the ring buffer in memory. At first, an array of defined sizes corresponding to the required buffer size was address mapped to the buffer data pointer. The data write to memory was also implemented as a straightforward write to the defined address location by directly using the assignment operator. This loaded the logged elements to the HSM memory as the array was defined in HSM address space. Hence the buffer data remained inaccessible to Host.

As the next step, the buffer implementation was shifted to the Host memory by referencing the address of a specified free Host RAM location to the ring buffer data pointer. But for such a scenario the direct write to the Host memory from the HSM side using the assignment operator can cause memory access and write errors due to synchronization issues. Thus, the write to buffer was performed using an inbuilt function that was part of the HSM comm module (represented here by an alternate name write_to_host). The `HsmLogger_PushData` function is called to update the TLV fields discussed in section 4.1.2 to the ring buffer. The ring buffer start address is already stored to the global variable `Host_Ringbuf_Startaddr_g` from which each succeeding address location can be accessed by incrementing the variable value. The `Buffer->tail` corresponds to the position where the next element will be inserted. When the buffer is full, the tail will reset to value 0 and the next write would happen again at the start of the ring buffer (position 0).

```
uint8 HsmLogger_PushData(struct RingBuffer *Buffer, uint32 Data)
{
   uint32 *data_addr = (uint32 *) Data;
   write_to_host((Host_Ringbuf_Startaddr_g+Buffer->tail), &data_addr, 4);
   Buffer->tail += 1;
   if (Buffer->tail == Buffer->size)
   {
     Buffer->tail = 0;
```

```
    }
    return 0;
}
```

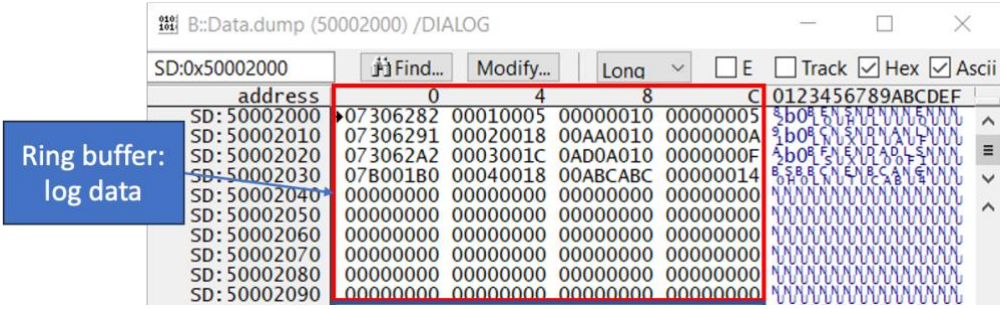


Figure 4.4. Logged ring buffer data shown using Trace32

The in-built function in the TC3xx hardware platform writes 8 bits of data at a time to the defined address location securely by checking the processor states, pending errors, and in a synchronized manner. Figure 4.4 shows a 10-element ring buffer from the Host memory taken while debugging using the Trace32 tool. The four rows of four 32-bit memory elements are the logged data. Each address group is 4 bytes in size and as mentioned previously, each log element consumes 16 bytes of memory. The log values are stored at predefined locations pointed by `Host_Ringbuf_Startaddr_g` (which was configured here as SD:0x50002000) and can be modified from the code. For the initial test, the memory location was hardcoded to a Host memory address, but it was modified to take the value sent from the Host once the proxyAPI was implemented. The ring buffer parameters like the head, tail, and size values are stored in the HSM memory as the buffer is first initialized within the HSM.

The ultimate goal however concerning log storage would be to have the data exported to the external non-volatile memory such as Flash memory. As the test team had already planned an implementation for synchronized write to external memory, to export the hardware trace outputs to Flash, the aim was to integrate the same with the HSM log and crash dump module after completion. This was planned as future work and is not explored as part of the thesis.

4.1.4 Method to capture crash data

Software crashes are captured as system exceptions in the hardware platform. The ARM processor already contains built-in methods to identify and handle exceptions as discussed in section 2.4.2. During exception handling some of the register contents are pushed to the stack but not retained as described in section 2.4.2. In addition, general-

purpose register contents are also not stored before exception handling. Hence, a crash dump capture function was developed at an assembly-level to integrate with the operating system code where the exception handler call is implemented (references the exception handling code in EVT).

In the HSM operating system code, the exception handler call is implemented as a wrapper function using assembly level macros. So, to save the crash data, a new wrapper function NK_GENERATEEXWRAPPER_CD was created using macros that push the LR to stack, calls the `hsmcrashdump_capture` function to save all the register contents and then calls the exception handler as shown below. Note that to prevent the original register value from getting corrupted, the LR is pushed to stack, used in the crash dump capture function, and popped from stack upon function return.

```
.macro NK_GENERATEEXWRAPPER_CD xyz
  .extern nk_\xyz\()Handler
FUNC nk_ExcHnd\xyz

#ifdef ENABLE_HSM_CRASHDUMP
  push {lr}
  bl hsmcrashdump_capture
  pop {lr}
#endif

...call to exception handler...

ENDFUNC nk_ExcHnd\xyz
.endm
```

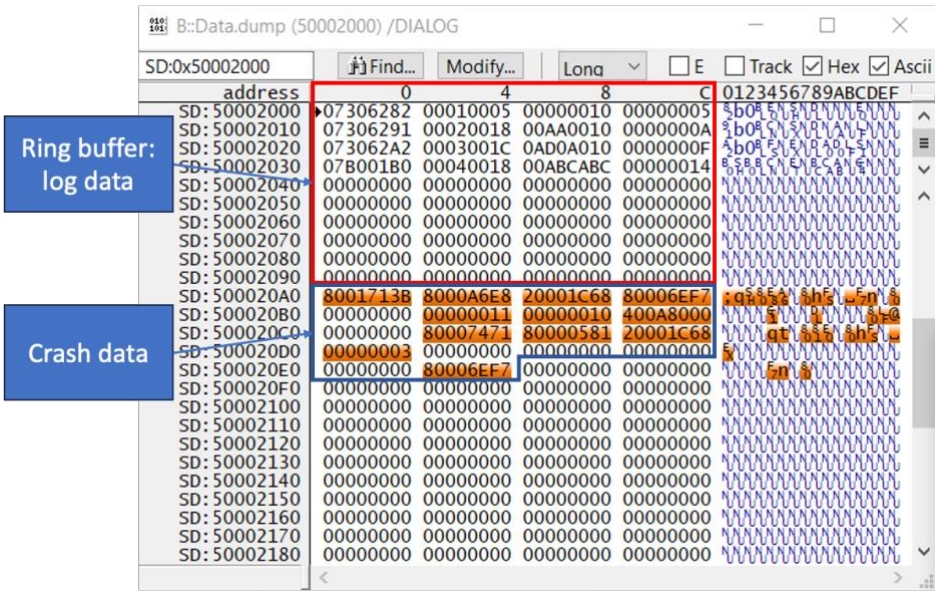


Figure 4.5. Log and crash dump data representation in memory

As a fixed number of registers are captured, the crash dump always has a fixed size and is programmed to be saved to the memory location following the ring buffer as shown in Figure 4.5. Thus, in the `hsmcrashdump_capture` assembly function, by accessing the start address of the ring buffer (`Host_Ringbuf_Startaddr_g`) and the buffer size, the required general-purpose registers and special registers retrieved from the stack (ARM Coretx-M3 behavior as mentioned in section 2.4.2) can be stored to Host memory. During a crash, 18 32-bit register values are captured always which makes the crash dump size to be 72 bytes in total. The captured register values are older LR, PC, SP values, Return address (earlier LR), and registers R0-R13. For the analysis only 4 out of the 18 registers are prominent. The 18 values are configured as a preliminary step and more register contents can be captured if needed as the storage mechanism implementation is robust enough to handle any ring buffer and crash dump size. Code from the `hsmcrashdump_capture` assembly function showing how the address to store data is passed and how data can be retrieved from the stack is as follows:

```
/* Store address of ring buffer as a value into register */
ldr r0, =Host_Ringbuf_Startaddr_g   /* Get global variable address */
ldr r0, [r0]                        /* Get value at the address */

/* Get LR value and write to address stored in r0 */
ldr r1, [sp, #36]    /* LR value at exception as per CortexM3 */
stmia r0!, {r1}      /* Store to address pointed by r0 */
```

4.1.5 Configuring hardware timer for timestamp

Timer is required to capture the exact time at which the log was created by the logging module relative to the start time of the HSM hardware. This helps the developer to understand which logs were created before others in the ring buffer and to see if there has been an unnecessary halt at some part of the code. In the HSM hardware, there exists the possibility to use different timers. In addition to the ARM Cortex-M3 processor SysTick timer as described in section 2.4.3 the AURIX TC3xx controller platform provides two 16-bit timers for the HSM. Unlike the SysTick timer, these count upward from a value stored in a reload register until it overflows. Upon overflow it triggers an overflow interrupt to the microcontroller, reloads the value defined in the reload register and starts counting upward from the new value. Details of the registers and address location cannot be provided as the reference used was a confidential document.

The ARM Cortex-M3 timer was already defined and being used in the HSM software system from the operating system code. It has a predefined 100µs time period

implementation in the system. But the controller timers were not yet used in HSM, and they offer the possibility to vary the time period of the count by manipulating the bits in the control register. Setting specific bit values in the control register to 00, 01, 10 or 11 will divide the input clock frequency by 4, 16, 64 or 512 respectively. The input clock frequency of the system in 100MHz, so by using the scaling factors we can attain a time-period between 0.04μs to 5.12μs, which provides a much higher resolution for the log timestamp than what is achievable by the configured SysTick timer. By setting the overflow flag of timer one as the clock for the second timer, the two 16-bit microcontroller timers offer the possibility of being used in a cascaded form to obtain a single 32-bit timer. But for long execution times and high resolution the registers would easily fill up, eventually providing not much benefit for the log system. Thus, the implementation was adapted to be able to choose the timer required before the compilation.

4.2 Log extraction and display

A Python-based application was implemented to decode the TLV-encoded HSM logs and display a human-readable form of the ring buffer data elements. A GUI was created using the `tkinter` Python library and the following were configured as input to the decoder tool:

- Path to exported memory dump: this text file containing the contents of the ring buffer and crash data is the primary source for the decoding tool.
- Path to the memory map file in the build folder: as this generated file contains important information regarding the log function calls, this is an important source for the decoding process.
- Path to `hsm_logger_fileid.h` file: this is needed to extract the name of the file where the log was called, based on the FileID from the memory dump.
- Path to the executable file in build folder (required only when crash dump decode is activated): the *.elf file is necessary for the crash dump decode where the line number and code corresponding to a given instruction address is to be found.

In addition, the number of log elements that can be logged in the buffer is also provided as input to the tool based on which the total size of the ring buffer is

automatically calculated. The log or crash dump decode can be separately selected and the decoded output may be saved as text file in addition to being displayed in the output window. Figure 4.6 shows an example of the decoder tool execution.

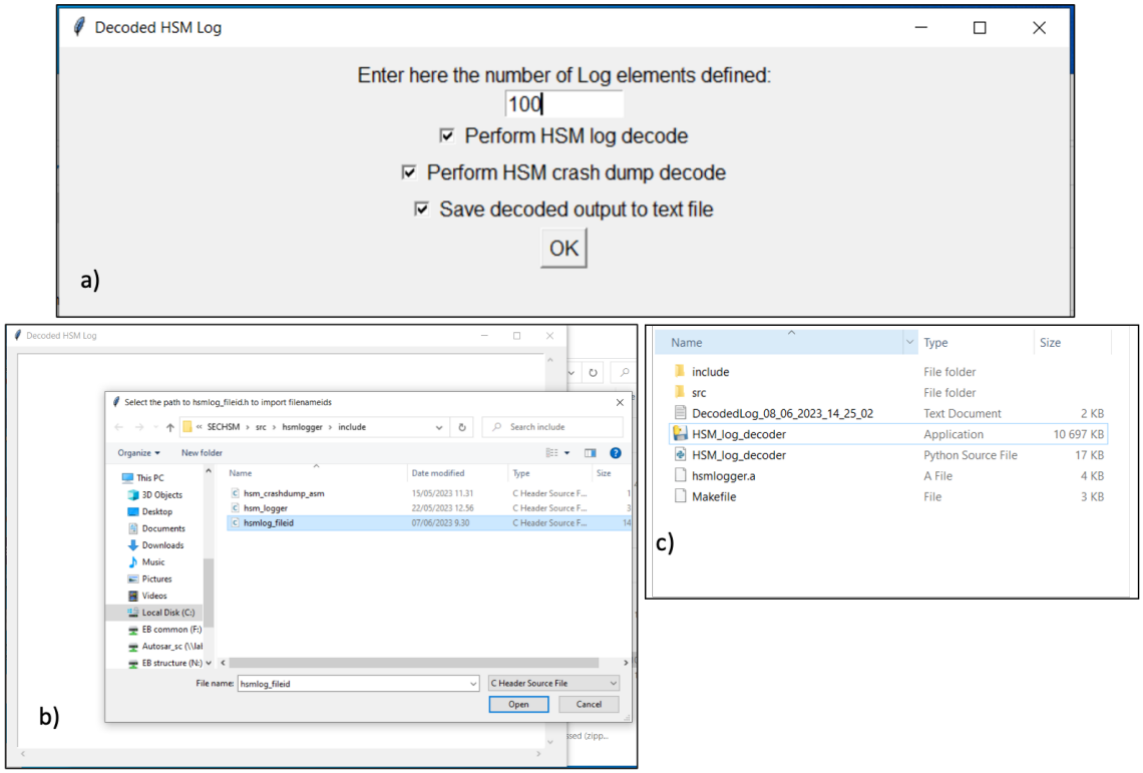


Figure 4.6. HSM decode tool: a) GUI window to choose decode configurations, b) Selection of required files as decoder inputs, and c) file view showing decoder exe and generated txt output

4.2.1 Obtaining the memory dump

As specified in the previous section, the Host system defines the memory location at which the ring buffer should start storing the logs. Two global variables (`Host_Ringbuf_Startaddr_g` and `Host_Ringbuf_Endaddr_g`) were defined in the HSM code for the start and end address of this ring buffer (also including the crash dump data), and they were updated with the values that get passed from the Host during execution. After system execution as the memory gets filled with the log data, the global variables can be used as reference to retrieve the log data. This is possible by using the Trace32 command to export data between a set of memory addresses:

```
Data.SAVE.AsciiHex C:\HSMlog_dump.txt Data.Long(Host_Ringbuf_Startaddr_g) --
Data.Long(Host_Ringbuf_Endaddr_g)
```

The memory dump created with the above command for a crashed software having a 10-element ring buffer is shown in Figure 4.7. The number of register values collected during a crash is fixed to 18 and is stored below the ring buffer.

```
 $A2000,
82 62 30 07 05 00 01 00 10 00 00 00 04 00 00 00
91 62 30 07 18 00 02 00 10 00 AA 00 09 00 00 00
A0 62 30 07 1C 00 03 00 10 A0 D0 0A 0E 00 00 00
B0 01 B0 07 18 00 04 00 BC CA AB 00 13 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
23 71 01 80 E8 A6 00 80 60 1C 00 20 F7 6E 00 80
00 00 00 00 11 00 00 00 10 00 00 00 00 80 0A 40
00 00 00 00 71 74 00 80 81 05 00 80 60 1C 00 20
03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 F7 6E 00 80 00
$S1572
```

Figure 4.7. An example memory dump exported as text file

4.2.2 Implementing Python based decoding tool: Log data

The following are the key elements in the decoding tool for the logging part, but the initial data processing and the output step are common between both the log and crash data decoding segments:

- Processing raw data from memory dump: data exported from the Trace32 tool saved as a text file is not aligned in a directly readable format as shown in Figure 4.8. First the exported file is read line by line until the defined number of log elements. The hex data in the dump is then captured four bytes at a time and swapped to form the correct data value.

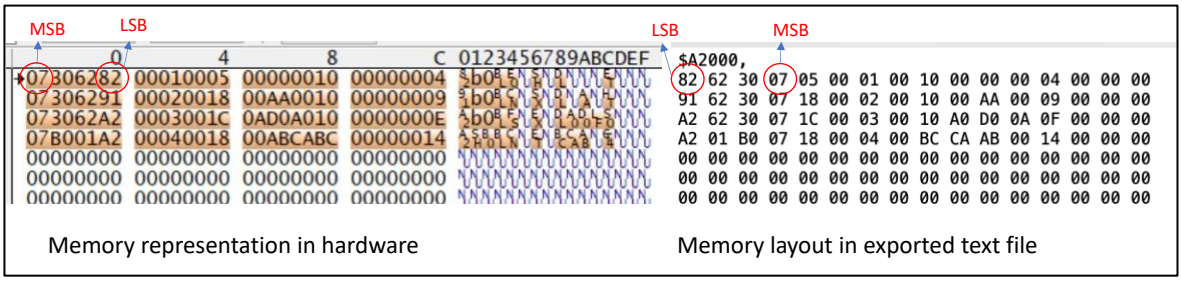


Figure 4.8. Byte order shift during memory export using Trace32

This is achieved by passing the data read from the file as a string to the formatting function, reversing and then swapping adjacent bytes:

```
def format_hex_data(stringval):
```

```
    val = list(stringval)
    val.reverse()
    for i in range(0, len(val), 2):
        val[i+1], val[i] = val[i], val[i+1]
    return ''.join(val)
```

- Capturing required data elements: once the data is aligned as required, the values at required bit positions are captured and separated into the TLV elements discussed in section 4.1.2. This provides a numerical value for each of the elements we would like to decode. For example, '07300682' gets decomposed as Log-level = '0x2', Line_no = '0x0068', FileID = '0x073' as per the encoding scheme shown in Table 4.1. The following code snippet shows how the data string is decomposed to required elements.

```
# Capture elements from data string using known bit ranges
timervalue.append(create_timestamp((data[j])[0:8]))
passeddata.append(hex(int((data[j])[8:16],16)))
logcount.append((int((data[j])[16:20],16)))
passeddatalength.append((int((data[j])[20:24],16)))
```

- Converting captured numerical value to required output: Once the individual elements are captured, the numerical value is compared, and the text data is derived for certain elements. For example, Log-level = '0x2' is converted to Log-level = 'Information' and the FileID = '0x073' or decimal value 115 is searched inside the `hsm_logger_fileid.h` file to recover the actual file name.

- Extract from mapping file: By searching for the filename and line number in the mapping file, the keyword corresponding to the data passed and the documented comment are extracted.

- Formatting time to timestamp: The logged time in microseconds from the timer is converted to a timestamp relative to the start time of the system. The logs shown here are taken by running HSM testcases and so the timer had to be manually started. Thus, the first log has a very small timer value close to 0. But in the main execution phase, the timer can be configured to start within the HSM initialization task which is not configured in the test cases.

- Generate the output: Once all the elements are captured, they are displayed on the output window using the print statement. This is achieved by using a `stdout` redirector that redirects the print statements to the `tkinter` window as well as a file that is created if the 'save to text file' option was

enabled during the initial decode configuration setup window. The decoded log output is shown in Figure 4.9.

4.2.3 Implementing Python based decoding tool: Crash data

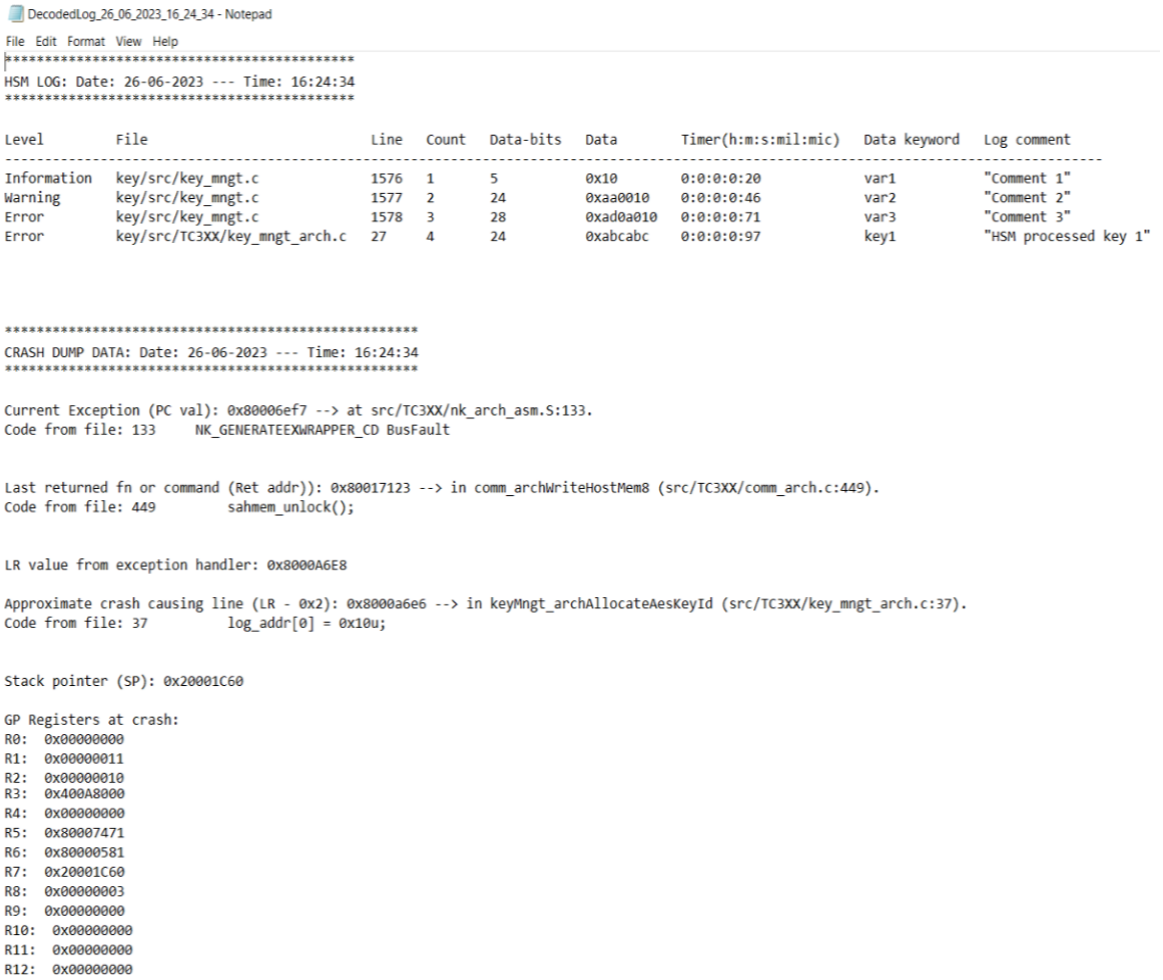
The data processing and output steps remain the same for the crash dump decode segment as discussed in previous section. The important elements in the decoding tool with respect to the crash dump data decoding are:

- Retrieving register contents: Once the data alignment is corrected the respective register contents can be retrieved directly because the order in which the register values are written to the log is known beforehand (from the crash dump creation implementation).
- Using GDB to extract important information: Using the register contents from PC at the time of the crash, we can determine the function that caused the software crash by using an open-source debugger application such as GDB. For this we also require the executable *.elf file that was provided as input to the tool. The initial task would be to invoke the GDB subprocess via Python for which a library named `pygdbmi` was used that facilitates executing commands and receiving response from GDB subprocess. But as the hardware platform requires the ARM Coretx-M specific version of GDB named the `arm-none-eabi-gdb`, the `pygdbmi` library code that invokes the subprocess was modified to make a call to the `arm-none-eabi-gdb` tool.
- Running GDB commands: After running the GDB subprocess, the 'file' command is used to load the executable file to the debugger. The `pygdbmi` library provides a function `gdbmi.write(<command>)` to write and execute a command in GDB and it also returns the response from the GDB subprocess for the executed command as a Python dictionary datatype (key-values pairs). From the response data, the actual output string will be available from the value mapped to the key name 'payload'. Next using the 'list * <address>' command we can retrieve the function under which the instruction address is present, the line number, and the code at that line. The Python function that achieves the same is shown below.

```
def decode_crashdump(elf_file,addr):
    if __name__ == '__main__':
```

```
# create gdb controller instance
    gdbmi = GdbController()
    # load the ELF file
    elf_file = "file " + elf_file
    response = gdbmi.write(elf_file)
    new_command = "list *" + str(addr)
    response = gdbmi.write(new_command)
    line = str(response[7]["payload"])
    text = str(response[1]["payload"]).replace("is", "-->")
    val = text + "Code from file: " + line
    return val
else:
    return "No output"
```

- Organizing the output: The three most important pieces of information from the crash dump are the current PC value, which would be from within the exception handler function, thereby telling us exactly which exception has occurred (for example `BusFault`), the last return address (previous LR value) which would indicate the function that last successfully returned and the current LR value (previous PC value) which would correspond to the next instruction to be executed after the instruction that caused the software crash. The LR value is subtracted by 0x2 (size of instruction) because when the crash causing instruction itself is an assembly code, then the next instruction gets captured as the LR value. Using the return address and the LR value we can generate the call stack data which shows the function flow before the software crash. All the general-purpose registers and SP are directly written to variables. Finally, the decode crash data is redirected to the output window using print statements and saved also to the file if the corresponding option is enabled. The final output that combines the log and crash dump decode is shown in Figure 4.9. In the figure only 4 logs are captured as the system had crashed when processing one of the HSM functions. The decoded crash dump shows the function that caused the software crash.



DecodedLog_26_06_2023_16_24_34 - Notepad

File Edit Format View Help

```
*******************************************
HSM LOG: Date: 26-06-2023 --- Time: 16:24:34
*******************************************

Level        File                          Line   Count   Data-bits   Data       Timer(h:m:s:mil:mic)   Data keyword   Log comment
-------------------------------------------------------------------------------------------------------------------------------------
Information  key/src/key_mngt.c            1576   1       5           0x10       0:0:0:0:20             var1           "Comment 1"
Warning      key/src/key_mngt.c            1577   2       24          0xaa0010   0:0:0:0:46             var2           "Comment 2"
Error        key/src/key_mngt.c            1578   3       28          0xad0a010  0:0:0:0:71             var3           "Comment 3"
Error        key/src/TC3XX/key_mngt_arch.c 27     4       24          0xabcabc   0:0:0:0:97             key1           "HSM processed key 1"



*****************************************************
CRASH DUMP DATA: Date: 26-06-2023 --- Time: 16:24:34
*****************************************************

Current Exception (PC val): 0x80006ef7 --> at src/TC3XX/nk_arch_asm.S:133.
Code from file: 133     NK_GENERATEEXWRAPPER_CD BusFault


Last returned fn or command (Ret addr)): 0x80017123 --> in comm_archWriteHostMem8 (src/TC3XX/comm_arch.c:449).
Code from file: 449         sahmem_unlock();


LR value from exception handler: 0x8000A6E8

Approximate crash causing line (LR - 0x2): 0x8000a6e6 --> in keyMngt_archAllocateAesKeyId (src/TC3XX/key_mngt_arch.c:37).
Code from file: 37          log_addr[0] = 0x10u;


Stack pointer (SP): 0x20001C60

GP Registers at crash:
R0:  0x00000000
R1:  0x00000011
R2:  0x00000010
R3:  0x400A8000
R4:  0x00000000
R5:  0x80007471
R6:  0x80000581
R7:  0x20001C60
R8:  0x00000003
R9:  0x00000000
R10:  0x00000000
R11:  0x00000000
R12:  0x00000000
```

Figure 4.9. Decoded output for a crashed software run

4.3 Implementing security certificate

Features such as logging and other system monitoring features should be controllable so that they are activated only on command. To ensure security, the design requirement was to have a possibility for secure activation and deactivation of HSM logging functionalities. During the initial stage, this was controlled using simple compile time switches. But at the full software level, there shall be a possibility for the Host system or a trusted external agent to dynamically activate HSM functionalities like log and crash dump capture. This was achieved by implementing a security certificate that can be used as an authentication or handshaking mechanism between HSM and Host for activating the required features.

Cryptographic algorithms are broadly divided into two, symmetric and asymmetric. Symmetric cryptographic algorithms use a single shared key for the encryption and decryption of data while asymmetric cryptographic algorithms use a

public key for encryption and a private key for decryption. Asymmetric cryptography is most suited for authorization due to higher security even though its use can be more resource intensive. But as the security certificate likely targets one-time use, for the activation of features in HSM, the asymmetric cryptography algorithm was the better choice here.

A security certificate was designed as a collection of various information corresponding to the software and hardware system, that can be used to evaluate the authenticity of the requesting module or device. It was defined as a structure composed of the following information: version of its flashed software, the unique ID (UID) of the ECU where the module is running, an integer variable where each bit value can be configured to control the activation of an HSM functionality, the key slot where the cryptographic public key is present (keyID) and a cryptographic signature generated using the keys and the other parameters mentioned. To authenticate, the requesting component must generate a signature based on the public and private keys available to it and the same is verified by comparing it to the generated signature at the HSM side.

```
typedef struct
{
     uint32 cert_version;
     uint32 cert_featureselector;
     uint32 cert_keyID;
     uint8  cert_uid[16];
     uint8  cert_signature[32];
}CryptoJobLoggingCertificate;
```

The UID capture, public key generation, signature creation and so on are achieved using inbuilt asymmetric cryptographic functions that were implemented by the cybersecurity team using the ECC or RSA algorithms. Their use is strongly related to cybersecurity functions and their details are out of scope of this thesis work. The process by which the authentication occurs is:

- The external module accesses its software version, UID, and the value to be configured for the feature selector. It then derives its public and private keys. With the above data and its keys, it generates a signature and writes it into the `cert_signature` field. At this step, the certificate is said to be signed.
- The module then sends this certificate (as a structure object) to HSM via the proxyAPI interface. HSM accesses this certificate and uses its own private and public keys upon the received data (version, UID, feature-selector and public

key from the key slot pointed by keyID) to create a signature. If the generated signature matches the one from the certificate, the authentication is approved.

- Once approved, the bit fields in the `cert_featureselector` are checked and the respective feature is activated or deactivated. If the authentication fails, the request is discarded. For the log and crash dump feature, if the LSB of the approved `cert_featureselector` variable is 1 then the feature is activated.

To achieve the transfer of the security certificate from the Host modules to HSM a proxyAPI interface was implemented. This was achieved by utilizing an existing cryptography structure object from the communication module and attaching the newly created certificate object `CryptoJobLoggingCertificate` under it. A pointer to the communication structure object is made available on the HSM side and from it the respective logging certificate structure can be derived. The exact same method was followed to implement a proxy interface to send the ring buffer address from Host to HSM. The overall architecture of this implementation is depicted in Figure 4.10.

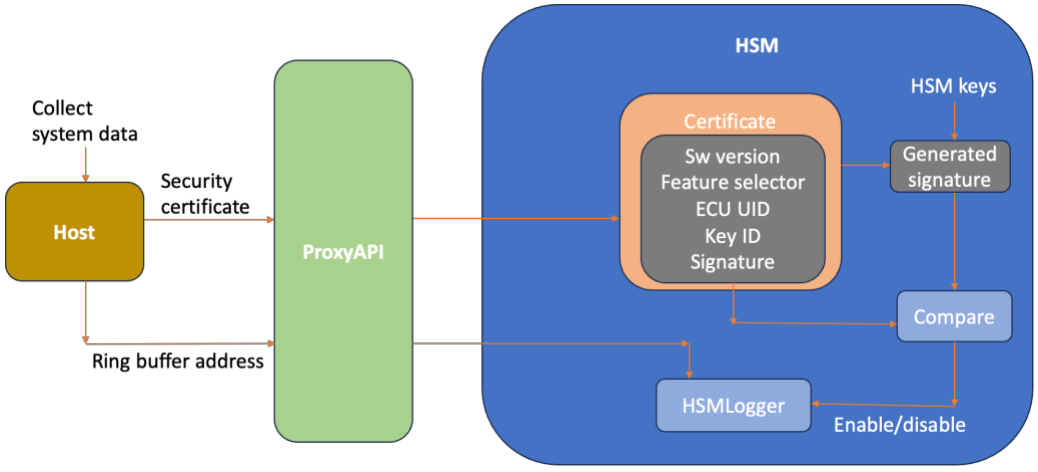


Figure 4.10. The overall structure of certificate and interface implementation

4.4 Verification steps

The verification of the encoding and decoding sections of the implementation was made using the Trace32 debugging tool and with the help of the log outputs from the decoder tool. But as the source code changes were involved in two main parts of the software system, one in the HSM code and the other in the proxyAPI or interface code, the implementation was also verified with tests in respective environments.

4.4.1 Log and crash data verification using Trace32

Basic inspection was first made by compiling the software and ensuring the required files were generated and included for compilation. Over multiple checks the generated file contents and the use of activation switches to control the logger functions were verified. The next phase of verifications involved the thorough inspection of the logged data in the ring buffer locations with respect to the data provided as input in the instrumented source code files. This was made possible using Trace32 debugger tool for the AURIX TC3xx hardware where each address location in the HSM and Host RAM can be checked. The test setup involved the control of hardware via JTAG remotely connected to the developer PC via the USB port using the WuT USB redirector application. The Trace32 tool provides an option to view the memory dump for the Host and the HSM hardware, as well as to locate the exact memory address for each of the local or global variables in the software (after initialization). With the help of the tool and its GUI functions the capture of FileID, line number, correct encoding of each element in the logging process, write to ring buffer, its address location, the overflow process etc. were manually checked. An example of the verification is shown in Figure 4.11.

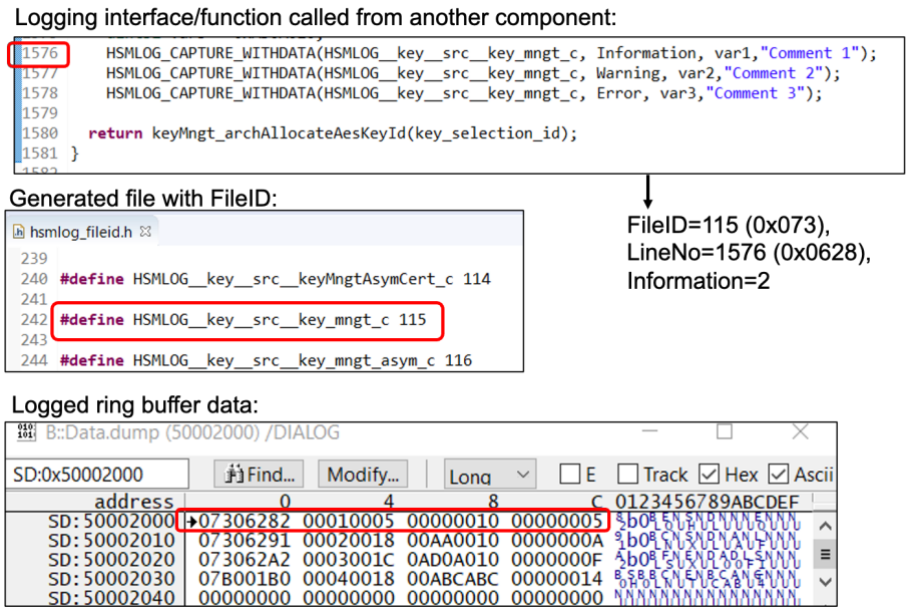


Figure 4.11. Verification of logged data using Trace32

As for the crash dump different types of known software crashes were created from the software to check if the same were captured in the log. Then the captured data in the ring buffer were verified against the real-time data in the hardware registers at the instant of the crash. The software crashes created were:

- Bus Fault: by defining an address location that is not available to HSM and then trying to write a value to it. This was achieved either by C code as:

```
volatile uint32 *log_addr = (uint32 *) 0x400a8000;  //Invalid address
log_addr[0] = 0x10;
```

The `BusFault` caused by the above example is shown in Figure 4.9. Another method that can be used is to run assembly code directly in C files using the `asm` command from compiler. `BusFault` created this way by accessing an undefined memory location is:

```
asm volatile(
   ".word 0xFFFFFFFF\n"
   "bx lr\n"
);
```

- SVCall: Supervisor Call command can be made in ARM processors to run predefined software services or functions. Making a call with value 0 using the assembly command `__asm__ volatile("svc 0")` inside of a source code file will return an SVC exception due to invalid command.
- Usage Fault: by making the software run into an abrupt termination simply by calling the `__builtin_trap()` function available from the GCC compiler.

Verification of the crash data is shown in Figure 4.12.

Finally for the decoding tool verification was done against different possible input combinations and commands. For different correct memory dump exports, the decoded outputs were verified against the data in the memory dump through manual comparisons. The output from the crash dump decode was also verified in this manner. During implementation, various exception handling methods were created as part of the Python code with try and except commands to capture empty inputs passed, abrupt termination of the program, out-of-bound inputs provided and so on. Each of these was manually tested. Also, the decoder tool was tested against wrong inputs such as giving incorrect generated files, wrong executable files, memory dump with empty crash data or memory dump with incorrectly aligned memory ranges, and so on. Each of the incorrect input cases were verified against various exception capture methods created.

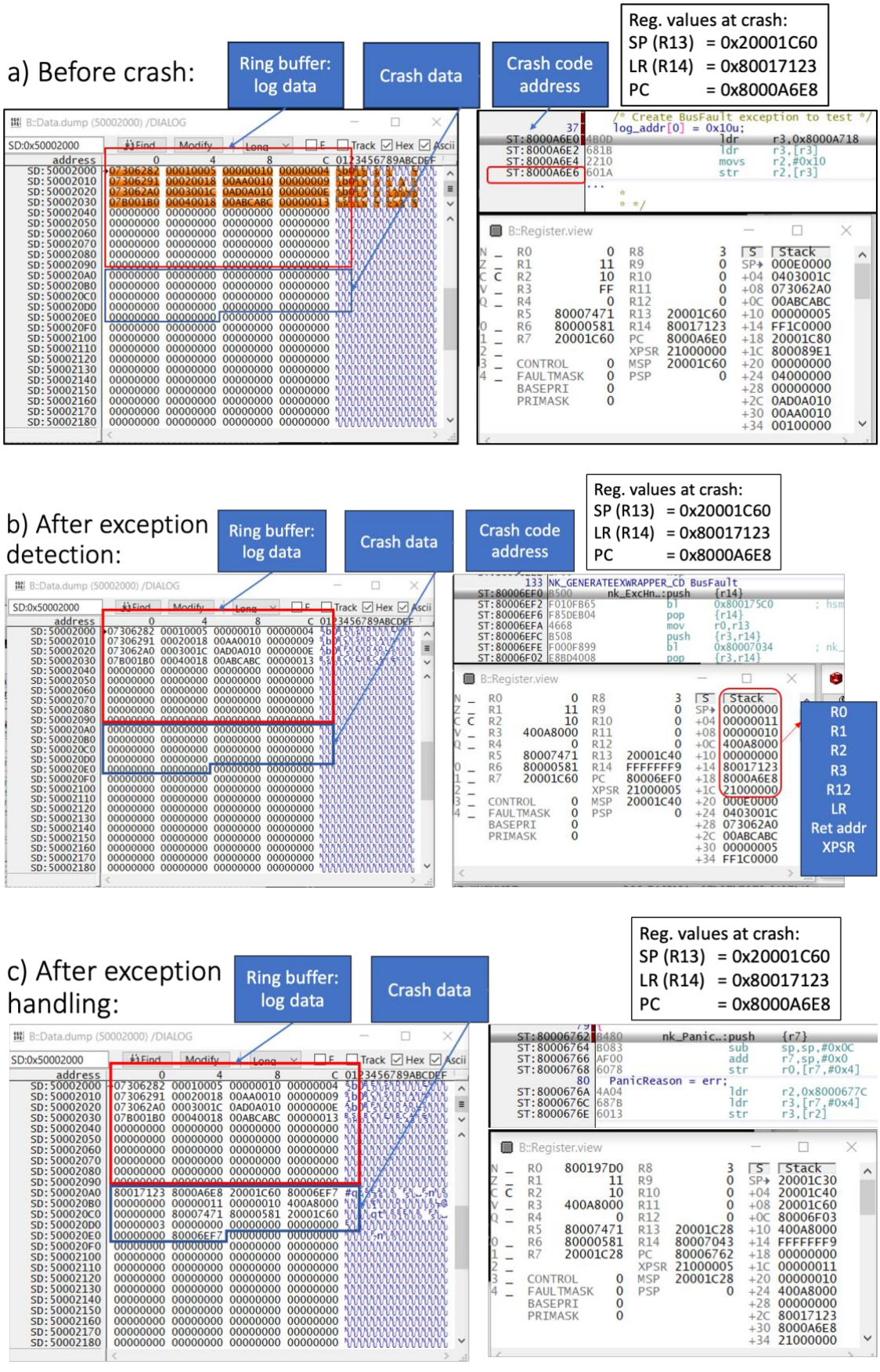


Figure 4.12. Verification of crash dump using Trace32

4.4.2 Security certificate and proxy interface verification using tests

The security certificate implementation was tested using the functional test method available within the software environment. A test case was created on the ProxyAPI side to generate the security certificate using available functions from HSM. A certificate structure was created as mentioned in section 4.3 and values for each of the structure variables like keyID, UID, etc. were derived using the inbuilt HSM functions. The software version was reserved for future use as there are no available functions now to derive the version number. So, a dummy value of 0 was provided for it. The function selector was configured to value 0x00000001 as only the LSB is defined for use now. A signature was generated using the collected data and written to the certificate structure.

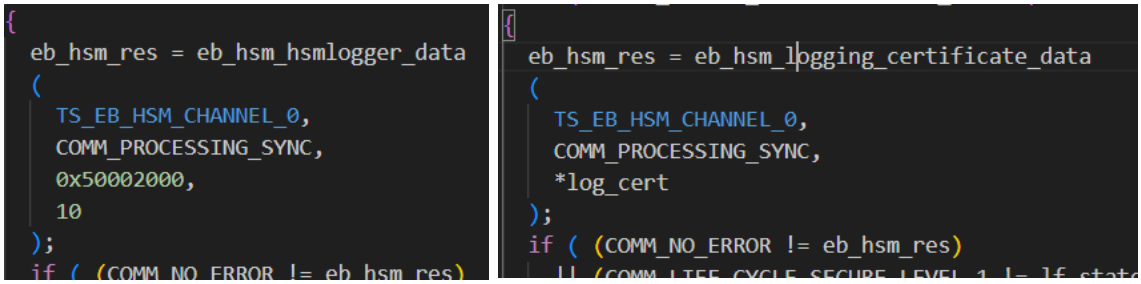


Figure 4.13. ProxyAPIs that send buffer data and security certificate

Next a proxyAPI interface was created as shown in Figure 4.13 to send the certificate from the Crypto driver side to HSM via the comm module. On the HSM side, another function was created to collect the certificate and extract its parameters. With the extracted data HSM generates a signature and verifies it with the incoming signature. After successful authentication, the high bit value of the function selector activated the HSM logging function and the logging process was established. The same method was used to verify the transfer of the ring buffer address from the Host to HSM by creating a different proxyAPI to send the data as a pointer to HSM side where it was read into a global variable.

4.4.3 Unit test and resource analysis

All the implementations performed in the C code were examined for proper execution by running code coverage and unit tests. Code coverage ensures that each of the loops or condition statements are covered during the test run. This confirms that each part of the code is executable and there are no infinite loops or statements blocks that cannot be entered. This was achieved by compiling for coverage build from the

`Makefile` and running a testcase that includes calls to each block of the HSMLogger functions. Unit tests ensure that all positive and negative cases of a function implementation are performed. They were developed by making mock calls to the logging interface with predefined values for FileID, data, log level, etc., reading out data from specific parts of the memory and ensuring that the logged data from the memory match the values passed from the test case. Passing invalid values were checked against the generation of value 0 in the log indicating that the passed data is incorrect or out of bounds. The result is shown in Figure 4.14 where the dummy function alone is not covered as it is empty and only executed when the logging function itself is not active. The Python code for the decoder tool and the assembly code for the crash dump capture cannot be tested from the coverage or unit tests and can only be inspected through manual verification as mentioned in section 4.4.1.

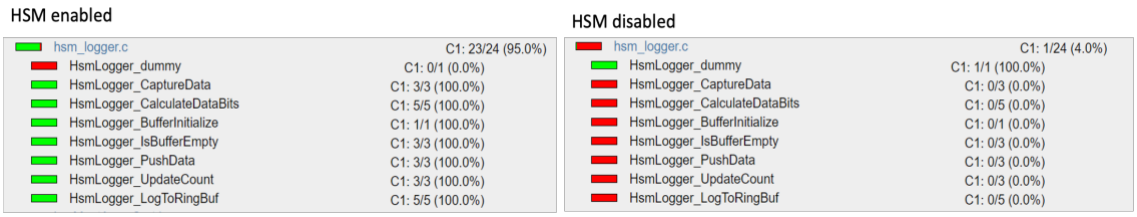
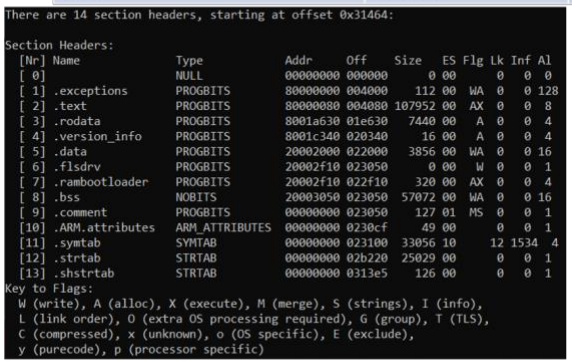


Figure 4.14. Test coverage for HSM logger

The mapping file generated from the compilation was used to analyze the effect of the code changes on the system resources. In the implementation, a dummy function was used to replace the log interface calls when the logging function was not enabled. Thus, the comparison of the *.map file was performed between a HSMLogger enabled build, a disabled build, and build from a fresh branch without any HSMLogger changes at all. The comparison as shown in Figure 4.15 indicates that the *.text* section corresponding to the code size increased by 110 bytes which means that the mentioned number of bytes would be needed additionally from flash storage to accommodate the added code. The '*.data'* section of the map file which indicates the RAM usage for initialized data increased by 16 bytes while the '*.bss'* section corresponding to the uninitialized data remained the same. With a flash size of 128kB and RAM of 96kB this addition in resource usage is minimal and hence lightweight. The compilation time also saw an increase with the logger function enabled due to the make file system evaluating and generating the required files. The CPU utilization for the build calculated using the Windows performance monitor showed an increase from 2.6% to 3.4% with the logging

function activated. In addition, the timing analysis of the logging function was performed using a built-in process time calculation function via the unit test code. It showed an average of 5 microseconds to log one element which is small compared to similar automotive logging methods such as DLT which may take up to tens of microseconds. Finally, the logging function was integrated into multiple HSM components including cryptographic functions, and their coverage and unit tests were performed. The passed results indicated that the logging system did not have a functional impact on the integrated components.

Feature	Clean branch	HSMLogger disabled	HSMLogger enabled
.text	107952	108016	108640
.data	3856	3872	3872
.bss	57072	57072	57072



```
There are 14 section headers, starting at offset 0x31464:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000      0 00      0   0  0
  [ 1] .exceptions       PROGBITS        80000000 004000    112 00  WA  0   0 128
  [ 2] .text             PROGBITS        80000080 004080 107952 00  AX  0   0  8
  [ 3] .rodata           PROGBITS        8001a630 01e630   7440 00   A  0   0  4
  [ 4] .version_info     PROGBITS        8001c340 020340     16 00   A  0   0  4
  [ 5] .data             PROGBITS        20002000 022000   3856 00  WA  0   0 16
  [ 6] .flsdrv           PROGBITS        20002f10 023050      0 00   W  0   0  1
  [ 7] .rambootloader    PROGBITS        20002f10 022f10    320 00  AX  0   0  4
  [ 8] .bss              NOBITS          20003050 023050  57072 00  WA  0   0 16
  [ 9] .comment          PROGBITS        00000000 023050    127 01  MS  0   0  1
  [10] .ARM.attributes   ARM_ATTRIBUTES  00000000 0230cf     49 00      0   0  1
  [11] .symtab           SYMTAB          00000000 023100  33056 10     12 1534  4
  [12] .strtab           STRTAB          00000000 02b220  25029 00      0   0  1
  [13] .shstrtab         STRTAB          00000000 0313e5    126 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (purecode), p (processor specific)
```

Clean branch build without HSMLogger changes

```
There are 14 section headers, starting at offset 0x31748:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000      0 00      0   0  0
  [ 1] .exceptions       PROGBITS        80000000 004000    112 00  WA  0   0 128
  [ 2] .text             PROGBITS        80000080 004080 108640 00  AX  0   0  8
  [ 3] .rodata           PROGBITS        8001a8e0 01e8e0   7440 00   A  0   0  4
  [ 4] .version_info     PROGBITS        8001c5f0 0205f0     16 00   A  0   0  4
  [ 5] .data             PROGBITS        20002000 022000   3872 00  WA  0   0 16
  [ 6] .flsdrv           PROGBITS        20002f20 023060      0 00   W  0   0  1
  [ 7] .rambootloader    PROGBITS        20002f20 022f20    320 00  AX  0   0  4
  [ 8] .bss              NOBITS          20003060 023060  57072 00  WA  0   0 16
  [ 9] .comment          PROGBITS        00000000 023060    127 01  MS  0   0  1
  [10] .ARM.attributes   ARM_ATTRIBUTES  00000000 0230df     49 00      0   0  1
  [11] .symtab           SYMTAB          00000000 023110  33488 10     12 1548  4
  [12] .strtab           STRTAB          00000000 02b3e0  25319 00      0   0  1
  [13] .shstrtab         STRTAB          00000000 0316c7    126 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (purecode), p (processor specific)
```

Build with HSMLogger changes and set enabled

Figure 4.15. Comparison of map file data

5 Conclusion and future work

With the implementation of the HSM log and crash dump module, the thesis work successfully demonstrates a framework to securely debug AUTOSAR software modules in an automotive ECU and achieve faster analysis. The design discussed here is lightweight, robust and have limited dependency on the underlying hardware. It can be applied to any software module on an embedded system to create a simple and secure logging mechanism.

The system indeed has some limitations such as the extent of analysis is limited with a logging solution, introducing log statements can be cumbersome if not handled well and it introduces overheads in build time. But it offers a possibility to move towards hardware-independent debugging and more importantly the capture of errors on the field. Also, the current implementation focuses on writing the logs outside of HSM to ensure that the data is not lost during an HSM crash. But as it is still part of the volatile memory it has a dependency on the JTAG interface and debugger software such as Trace32 to export the memory contents to the developer workstation. Thus, the ideal method would be to save the logs to the system flash memory from which it can be exported as needed. This implementation to develop an interface to write to external flash memory was an ongoing activity from the HSM test team and hence was not included in the thesis work. Once implemented the aim would be to write the ring buffer data in intervals to the flash memory using this available interface. Further future work includes the development of a communication method that avoids interfaces such as JTAG to reduce hardware dependencies further, requiring authentication to allow memory dump export and the study of methods to increase the standardization within the framework.

The thesis work would not have been possible without the support from the internal supervisor Dr. Gusztáv Hantos, the external supervisor Juha Mäki-Asiala, the managers and cybersecurity team members at Elektrobit Automotive Finland, and other professors at Budapest University of Technology and Economics, Hungary. The author would like to wholeheartedly thank them for their guidance.

References

[1] Steckelberg, Danilo. Development of an internal combustion engine fuel map model based on on- board acquisition. 2016.

[2] Science Direct, Automotive Electronics, Trends, and Challenges, accessed 10 February 2023, https://www.sciencedirect.com/topics/engineering/automotive-electronics (Second edition, 2019).

[3] D. Kim, Y. Jeon and J. Kim, A secure channel establishment method on a hardware security module, 2014 International Conference on Information and Communication Technology Convergence (ICTC), Busan, Korea (South), 2014, pp. 555-556, doi: 10.1109/ICTC.2014.6983209.

[4] Subrahmanyam Namdikam, SHE Key Update Protocol, How to Persist SHE Key in MICROSAR and Functional Flow, version 1.01.00, accessed 5 March 2023, https://support.vector.com/sys_attachment.do?sys_id=534d25eb87548590b9f233770cbb3550

[5] El-Masri D, Petrillo F, Gu´eh´eneuc YG, Hamou-Lhadj A, Bouziane A (2020) A systematic literature review on automated log abstraction techniques. Information and Software Technology 122:106276.

[6] Zhou, Rui & Hamdaqa, Mohammad & Cai, Haipeng & Hamou-Lhadj, Abdelwahab. (2020). MobiLogLeak: A Preliminary Study on Data Leakage Caused by Poor Logging Practices. 577-581. 10.1109/SANER48275.2020.9054831.

[7] Patel, Keyur & Faccin, João & Hamou-Lhadj, Abdelwahab & Nunes, Ingrid. (2022). The Sense of Logging in the Linux Kernel. 10.48550/arXiv.2208.06640.

[8] Zhou, Junwei et al. DeepSyslog: Deep Anomaly Detection on Syslog Using Sentence Embedding and Metadata. IEEE transactions on information forensics and security 17 (2022): 1–1. Web.

[9] IBM Documentation, syslog() – Send a message to the control log, accessed 10 April 2023, https://www.ibm.com/docs/en/zos/2.3.0?topic=functions-syslog-send-message-control-log

[10] barectf, barectf 3.0 documentation, accessed 5 April 2023, https://barectf.org/docs/barectf/3.0/index.html

[11] The LTTng Documentation, last update 15 June 2021, v2.13, accessed 6 April 2023, https://lttng.org/docs/v2.13/#doc-what-is-tracing

[12] Margheritta, Paul, and Michel R. Dagenais. LTTng-HSA: Bringing LTTng Tracing to HSA-based GPU Runtimes. Concurrency and computation 31.17 (2019): e5231–n/a. Web. Pages 973-977.

[13] Apache Log4cxx version 1.1.0, Loggers, Appenders and Layouts, accessed 10 April 2023, https://logging.apache.org/log4cxx/latest_stable/usage.html#layouts

[14] Debugging with GDB, Version 14.0.50.20230507-git, accessed 25 April 2023, https://sourceware.org/gdb/current/onlinedocs/gdb

[15] Segger, J-Link Debug Probes, accessed 1 May 2023, https://www.segger.com/products/debug-probes/j-link/models/j-link-base/#:~:text=J%2DLink%20BASE%20is%20a,production%20(flash%20programming)%20purposes

[16] Lauterbach Development Tools, Training – Debugger Basics, Release 09.2022, https://www2.lauterbach.com/pdf/training_debugger.pdf

[17] AUTOSAR, Standards, accessed 20 March 2023, https://www.autosar.org/standards

[18] AUTOSAR Layered Software Architecture, document ID 53, release R20-11, https://www.autosar.org/fileadmin/standards/classic/22-11/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[19] AUTOSAR, Log and Trace Protocol Specification, release R20-11, https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_LogAndTraceProtocol.pdf

[20] HSM, AURIX TC3xx Microcontroller Training, document version V1.0 2020-09, Infineon, https://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_Hardware_Security_Module_Quick-Training-v01_00-EN.pdf?fileId=5546d46274cf54d50174da4ebc3f2265

[21] Cortex-M3 Technical Reference Manual, Revision r2p0, https://developer.arm.com/documentation/ddi0337/latest/

[22] ARMv7-M Architecture Reference Manual, Issue E.e, Nineth release, https://developer.arm.com/documentation/ddi0403/latest/

[23] ARM Cortex-A Series Programmer's Guide for ARMv7-A, Version 4.0, https://developer.arm.com/documentation/den0013/d/Exception-Handling/Other-exception-handlers/SVC-exception-handling

[24] Using Cortex-M3/M4/M7 Fault Exceptions, MDK Tutorial, AN209 Summer 2017, V 5.0, https://developer.arm.com/documentation/kan209/latest/

[25] Cortex-M3 Device Generic User Guide, Version 1.0, https://developer.arm.com/documentation/dui0552/a/cortex-m3-peripherals/system-timer--systick

[26] Qian, Zhihong et al. "Research and Development of Compiler Based on GCC." Recent Advances in Computer Science and Information Engineering. Vol. 126. Germany: Springer Berlin / Heidelberg, 2012. 809–814. Web.

[27] GNU Make: A Program for Directed Compilation, document Version 4.4.1, https://www.gnu.org/software/make/manual/make.pdf

[28] Nano OS, nanosoft-net/nano-os, GitHub, accessed 25 April 2023, https://github.com/nanosoft-net/nano-os

[29] Lauterbach Development Tools, Training JTAG interface, Release 09.2022, https://repo.lauterbach.com/pdf/training_jtag.pdf

[30] Lauterbach Development Tools, Arm JTAG Interface Specifications, Release 09.2022, https://www2.lauterbach.com/pdf/app_arm_jtag.pdf

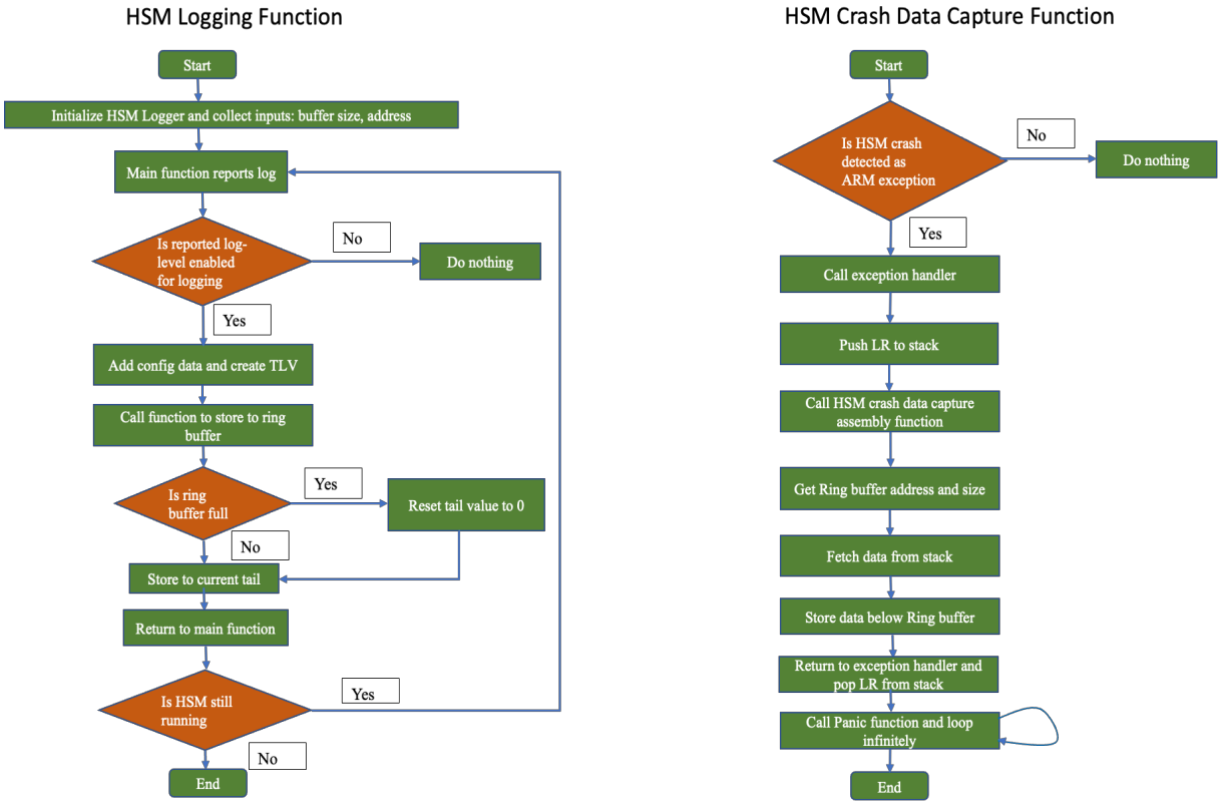
[31] Qiu, Tongqing et al. “What Happened in My Network: Mining Network Events from Router Syslogs.” Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement. ACM, 2010. 472–484. Web.

[32] S. Bunch, R. Hochsprung and T. Moore, "PowerPC Platform: a system architecture," COMPCON '96. Technologies for the Information Superhighway Digest of Papers, Santa Clara, CA, USA, 1996, pp. 140-144, doi: 10.1109/CMPCON.1996.501760.

[33] e200z4 Power Architecture Core Reference Manual, NXP, Rev. 0 10/2009, https://www.nxp.com/files-static/32bit/doc/ref_manual/e200z4RM.pdf

Annex

A simplified flowchart for the design of the logging and crash data capture functionalities are shown below.



The following Trace32 debug window shows how HSM data can be inspected based on the code being executed. The window on the right shows an example of writing to undefined memory locacation 0x400A8000 which then creates a BusFault exception.