Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Integration of iLLD drivers to Erika Enterprise RTOS on Infineon TC387

Bc. Danylo Begim

Supervisor: Ing. Michal Sojka, Ph.D.
Field of study: Cybernetics and Robotics
May 2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Begim** Jméno: **Danylo** Osobní číslo: **510862**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra řídicí techniky**

Studijní program: **Kybernetika a robotika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Integrace iLLD ovladačů do RTOS Erika Enterprise pro Infineon TC387

Název diplomové práce anglicky:

Integration of iLLD drivers to Erika Enterprise RTOS on Infineon TC387

Pokyny pro vypracování:

1. Seznamte se s RTOS Erika Enterprise 3 (EE) a mikrokontrolérem Infineon TC387.
2. Na základě pokynů vedoucího zprovozněte kompilaci a ukázkové aplikace EE ve vývojovém prostředí Aurix Studio.
3. Integrujte do EE ovladače iLLD, zejména podporu pro obsluhu přerušení. Zprovozněte minimálně GPIO (IfxPort), CAN FD a SPI ovladače. Pomocí SPI zprovozněte čtení a zápis na SD kartu se souborovým systémem.
4. Vytvořte ukázkové aplikace demonstrující výhody použití EE (multi-tasking) oproti „bare-metal“ a využívající zmíněné periferie. Dále implementuje benchmarky měřící výkonost či dobu odezvy jak EE, tak zmíněných periferií (odezva CANu, latence zpracování přerušení v OS, rychlost čtení z SD karty apod.).
5. Výsledky zdokumentujte.

Seznam doporučené literatury:

[1] Infineon AURIX TC3xx UserManual v2.0.0 2021-02
[2] Infineon AURIX TC38x UserManual v2.0.0 2021-02
[3] https://www.erika-enterprise.com/wiki/
[4] Rutrle T., Using an embedded QP solver for automotive applications, Master's thesis, ČVUT, 2022, https://dspace.cvut.cz/handle/10467/101688

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Michal Sojka, Ph.D. vestavěné systémy CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **05.02.2024** Termín odevzdání diplomové práce: **24.05.2024**

Platnost zadání diplomové práce:
do konce letního semestru 2024/2025

Ing. Michal Sojka, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání Podpis studenta

Acknowledgements

I would like to thank everybody who supported me while writing this thesis. Especially I would like to thank my parents and friends who helped me and made the completion of this thesis possible. Also, I would like to thank my supervisor for all the knowledge that I gained during the development of this diploma thesis. Finally, I thank Garrett Motion Inc. for making this project possible.

Declaration

I declare that this work was written independently. At the same time, I declare that all the sources used in the work are cited and listed according to the Methodical Instructions for Observing the Ethical Principles in the Preparation of University.

In Prague, May 2024

Abstract

The use of microcontrollers with Real-Time Operating Systems can lower the development cost of the application since the development team does not need to build everything from scratch. This diploma thesis shows how to compile Real-Time Operating System Erika Enterprise for Infineon TriCore TC387QP using Tasking compiler and Aurix Development Studio. The Erika Enterprise can run on many CPUs but lacks support for TriCore TC387QP and Infineon Low Level Driver library. To verify changes to Erika Enterprise, example applications for CAN and SPI communication were developed and tested.

Keywords: RTOS, Erika Enterprise, CAN, SPI, Infineon TriCore, iLLD, Embedded systems, FAT

Supervisor: Ing. Michal Sojka, Ph.D.
Jugoslávských partyzánů 1580/3, 160 00 Prague 6 - Dejvice

Abstrakt

Použití mikrokontrolérů s operačními systémy reálného času může snížit náklady na vývoj aplikace, protože vývojový tým nemusí stavět vše od začátku. Tato diplomová práce se věnuje tomu, jak zkompilovat operační systém reálného času Erika Enterprise pro Infineon TriCore TC387QP pomocí Tasking kompilátoru a Aurix Development Studio. Erika Enterprise ma podporu mnoha CPU, ale nemá podporu pro TriCore TC387QP a knihovnu Infineon Low Level Drivers. Pro ověření změn do Erika Enterprise byly vyvijeny a otestovány ukázkové aplikace s použitim sběrnic CAN a SPI.

Klíčová slova: RTOS, Erika Enterprise, CAN, SPI, Infineon TriCore, iLLD, Vestavěné systémy, FAT

Překlad názvu: Integrace iLLD ovladačů do RTOS Erika Enterprise pro Infineon TC387

Contents

May 24, 2024

Chapter 1
Introduction

The number of embedded systems in the automotive segment has significantly grown in recent decades. Cars that originally used to be just pieces of mechanical equipment are now an advanced electromechanical engineering art. Some features of modern cars, such as electronic stability program (ESP), torque splitting and parking assistant, just to name a few, require sophisticated mathematical computation algorithms, such as a quadratic program (QP) solver that is used for optimization purposes in control systems or model prediction control (MPC).

All of these changes are meant to provide customers with more efficient, more reliable and safer cars. This led to growing demand for the development of more advanced, multi-core microcontroller units (MCU), with advanced processors that are designed to be robust and comply with strict rules for safety-critical systems. To use this computational power more efficiently and make maintenance of the code easier, new real-time operating systems (RTOS), like Erika Enterprise have been developed to satisfy new industry needs. The use of real-time operating systems assures that in a well-designed system, all hard deadlines will be met. There are a lot more different reasons to use RTOS. For example, a better application structure or the concept of threaded execution, which allows the program to fill the busy-waiting gaps with the execution of another task. Also since Erika Enterprise is an OSEK/VDX type of RTOS. Migration to different hardware or different RTOS is easier due to the standardized API.

This thesis's main goal is to integrate iLLD drivers to Erika Enterprise and compile it with Tasking compiler using Aurix Development Studio. To do that, the code of RTOS should be changed to add support for TriCore TC387QP MCU and iLLD drivers. The thesis is also meant to provide the reader with enough information about how to use Erika RTOS with iLLD. For this purpose programming examples are developed. Also, examples are tested to verify the changes that were made to Erika Enterprise.

The hardware for this project is provided by *Garrett Motion Inc.*.

Chapter 2

Background

This chapter describes the background of the diploma thesis and should introduce the reader to the technologies that were used. Basic characteristics of the used MCU and the description of the MCU's modules that will be used in examples. Another important part of the thesis is Erika Enterprise. A short description of the RTOS, as well as some basic concepts of its configuration, are discussed in this chapter. The examples written as a part of this thesis use 2 types of communication protocols. One of them is CAN (Controller Area Network) and the second is SPI (Serial Peripheral Interface), which is used to communicate with an SD card. To store the data on the SD card, a file system called FAT (File allocation Table) is used.

The chapter describes the technologies used in a thesis from the higher level concepts like Erika Enterprise RTOS, and its configuration, moving to the lower level concepts like communication protocols and description of MCU modules.

2.1 ERIKA Enterprise RTOS

ERIKA Enterprise is an RTOS that is suitable for a wide range of microcontrollers, such as Karlay MPPA, AVR 8bit, ARM Cortex A5x, Intel x86-64 and Infineon TriCore just to name a few. RTOS became quite popular due to its AUTOSAR compatibility and hypervisor support[1].

Erika provides portable API implementation for different hardware architectures, configurable ROM and stack sharing to save RAM space. Resulting in a low and configurable footprint. The source code of the RTOS is available on the Evidence Srl Github repository[2] under GNU GPL. Thus Erika enterprise is the only open-source OSEK/VDX certified RTOS. That allows freely modify "MAKE" files or other parts of ERIKA to achieve needed goals. Even though Erika Enterprise has rudimentary support for the TriCore TC398 processor, it does not have support for TriCore TC387QP or iLLD integrated into it. The process of integration of iLLD drivers and TriCore TC387QP is discussed in section 3.3.1.

[1]`https://www.erika-enterprise.com/index.php/erika3/features.html`
[2]`https://github.com/evidence/erika3`

Erika allows users to configure different types of task schedulers. For example, user can define a static scheduler using *SCHEDULETABLE Object* or use a standard priority-based scheduler with different types of task queues that implement different time complexity of the scheduler itself.

Except for task scheduling another important part of any RTOS is interrupt handling. Erika provides users with two types of interrupt according to AUTOSAR standard [1]. Interrupt of category two can execute OS calls. More strictly said, interrupts of category 1 (ISR1) are not directly handled by the OS. Category 2 interrupts are handled by the OS and thus provide access to OS primitives [2]. In Erika, Category 2 interrupts (ISR2) are scheduled as a task with very high priority.

2.1.1 Configuration of ERIKA

This section provides users with a basic understanding of Erika Enterprise configuration. A more detailed description of the OIL configuration format can be found on Erika OIL Wiki page [4] or in the specification of OIL standard [3].

Since ERIKA implements the AUTOSAR OS and OSEK/VDX API specification. It does not allow the creation of the RTOS objects, e.g., tasks, timers, etc. dynamically at system run-time (with exceptions of Dynamic API functionality). Instead, the objects are created statically, at compile time. Configuration of objects is done in a specialized development environment RT-DRUID.

This environment is based on Eclipse IDE. And provide a set of plugins for users that implement a more comfortable way of writing configuration files for RTOS and additional debugging capabilities for embedded systems. Such as debugger support and memory analyzing tools.

To standardize configuration files among different implementations of OSEK like RTOS, the OSEK standard has introduced configuration file format OIL (OSEK implementation language). This format is used as a main configuration in ERIKA RTOS. Configuration of RTOS is written in a file conf.oil (just a configuration file later in a text). RT-Druid generates parts of Erika's source code based on the configuration in this file.

Configuration consists of so-called objects. Examples of such objects are tasks, resources, alarms etc. Each object has specific attributes (if the object has multiple attributes it can be called a section) such as priority for a Task object or interrupt source for an interrupt object.

Each configuration starts with the definition of the object called *CPU*. This object is used as a container for all other objects and does not have any specific attributes.

The next object that configuration should have is *OS object*. this object is used to define the ERIKA's global configuration as well as the compilation parameters. Compilation attributes are strings that can be used to specify some additional flags that the user might need to provide to the compiler. For example, level of optimisation, or custom flags that are used in make files.

Other important to mention attributes are OSEK/VDX-specific attributes. These attributes are described in OSEK standard [5]. An example of such an attribute is attribute *STATUS*, which specifies if OS will use a standard kernel or extended kernel.

Another important sections of OS object are *CPU_DATA*, *MCU_DATA* and *KERNEL_TYPE*. The first two specify which CPU is running the operating system, and what its parameters should be. Specifically *CPU_DATA* is used to specify CPU configuration such as compiler and clock-rate of CPU. Section *MCU_DATA* specifies the HW that OS is running on (connected pins, external oscillator frequency etc.). Core configuration can also include idle hook function or core-specific task options like *MULT_STACK*. Core specification in *KERNEL_TYPE* serves to provide a user with a more flexible kernel configuration such as the OSEK conformance class and scheduler that should be used.

The following code snippet provides the reader with an example configuration. It configures RTOS to run on a TriCore MCU and use just two cores. Code will be compiled using the TASKING compiler and the main core (core 0 or CPU0) is running on a 300MHz clock rate. The kernel conformance class is ECC2, which allows RTOS to store pending activations of a task. Each core of the CPU has its own idle hook and according to *EE_OPT*, the OS is compiled in debug mode.

```
CPU mySystem {

  OS myOs {

    EE_OPT    = "OSEE_DEBUG";
    EE_OPT    = "OSEE_ASSERT";
    EE_OPT    = "OS_EE_APPL_BUILD_DEBUG";
    EE_OPT    = "OS_EE_BUILD_DEBUG";
    STATUS    = EXTENDED;
    ERRORHOOK = TRUE;

    CPU_DATA = TRICORE
    {
      ID           = 0x0;
      COMPILER     = TASKING;
      CPU_CLOCK    = 200.0;
      IDLEHOOK     = TRUE {HOOKNAME = "idle_hook_core0";};
    };

    CPU_DATA = TRICORE
    {
      ID           = 0x1;
      IDLEHOOK    = TRUE {HOOKNAME = "idle_hook_core1";};
    };

    MCU_DATA     = TC39X {DERIVATIVE = "tc387qp";};
    KERNEL_TYPE  = OSEK {CLASS = ECC2;};

  };
```

2.1.2 Interrupt configuration in Erika

As it has been mentioned earlier Erika provides support for two types of interrupts. The first type of interrupt is referred to as ISR1 and it is serviced as a function call. The second type ISR2 is serviced as a task, that has to be scheduled, but within this type of interrupt OS primitives can be called. Erika Enterprise has basic support of TriCore's architecture interrupts.

The definition of interrupt in Erika Enterprise looks similar to this code.

```

ISR timer_isr_handler
{
    CPU_ID = 0x0;
    CATEGORY = 1;
    SOURCE   = "GPT12_GPT120_T3";
    HANDLER  = "timer_isr_handler";
    PRIORITY = 10;
};
```

The code in the snippet determines which CPU will serve this interrupt (in this case CPU0). The category of the interrupt is 1, according to AUTOSAR [1]. Which means that it is serviced with a function call. HANDLER specify the function name that will be called in case of interrupt in this case timer_isr_handler. Be aware that the following restriction applies on interrupt priority: ”all interrupts of Category 1 must have a higher or equal hardware priority compared with interrupts of Category 2. This limitation has been introduced to avoid various rescheduling problems appearing when an ISR2 interrupts a lower priority ISR1” [2].

The toughest part of interrupt configuration is to find out the source of the interrupt. Erika uses aliases to define the source of interrupts. All the names are located in the file **ee_tc_src.h**. The file also contains the aliases that are used for the interrupt sources in user manuals [6] and [7].

2.2 Controller Area Network (CAN)

This chapter is based on a description of CAN protocol in a CAN standard [8] and a guide for CAN FD [9]

Controller Area Network or shortly CAN belong to one of the most used data-communication protocols in the industry segment. Also, it is one of the most important protocols that has been used in the industry since it has been developed in the early 80‘s. Originally protocol was meant to be a solution for a growing number of copper wires that had been in cars at that time. The protocol allows small embedded devices to communicate with each other. CAN is a message-based protocol. It become so popular due to its reliability, robustness and efficiency.

CAN bus is often used for a safety critical application in the automotive industry. Such as engine control, anti-lock braking system (ABS), electronic stability program (ESP) and many other important parts of a modern car. The reliability of the CAN bus is achieved by the following factors:

- Message prioritization
- Error detecting and correction
- Differential signalling.

Let’s get through them one by one starting with message prioritization. Each frame has a so-called header. In general header in communication protocols is used to define protocol-specific information such as the priority of the message, length of data or other important flags such as time to live (TTL) or routing information in internet protocol (IP). According to CAN 2.0 specification, two headers are allowed on a bus. One of them is the standard header that was introduced in the original CAN bus protocol and allows devices on a bus to use 2048 priorities. Another one is called the ”extended header”. It consists of 29 bits and allows to use (2^{29}) priorities. Both headers can be used simultaneously on the same bus. CAN bus message header structures are shown in Figure 2.1.
Whenever the bus is free any unit (CAN controller) may start to transmit a message. If 2 or more units start transmitting messages at the same time, the bus access conflict should be resolved using the identifier. This process is called arbitration or arbitration phase. The conflict is resolved by performing bitwise AND operation between two headers. The arbitration phase is done in such a way that the message with a lower identifier wins arbitration over a higher identifier value. Meaning that the message with priority 0 has the highest priority.

This is because CAN is a dominant zero protocol, whenever any transmitter sends a dominant level, the bus will switch to that level no matter what is the level before. During the arbitration phase, every transmitter listens to the bus and compares the level of the transmitted bit with the level of the bus. If levels are not equal that means that the unit has lost an arbitration phase and must immediately withdraw without sending any other bits.

Error detection mechanism on CAN is implemented to prevent faulty devices to repeatedly sending error frames which will disturb communication.

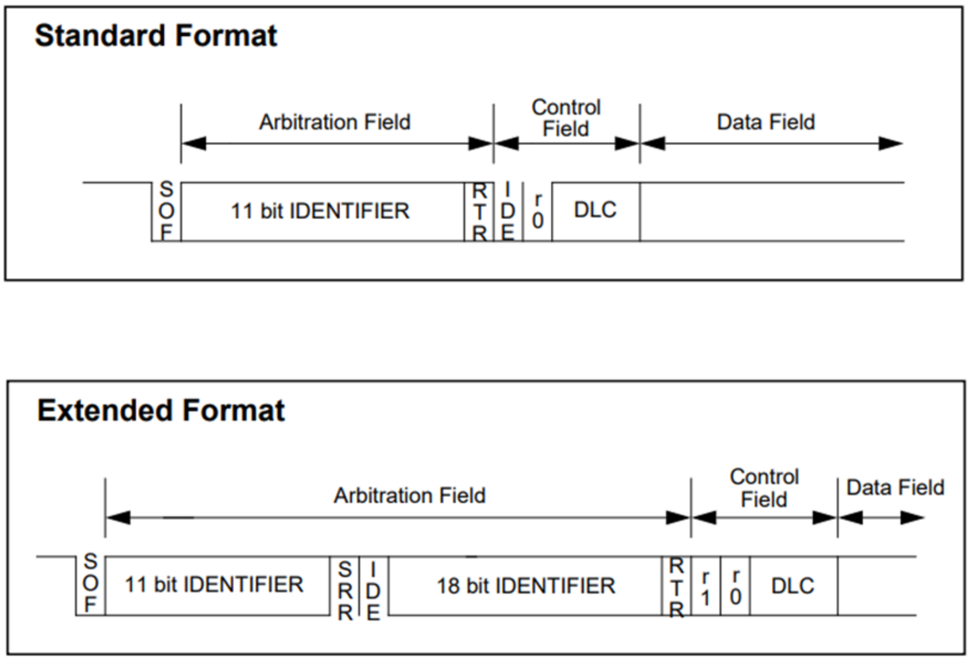


Figure 2.1: CAN bus data frames header types. Taken from [8].

The error frame consists of 6 bits of the dominant level. This frame also violates framing rules because according to the standard, each frame also has so-called “stuffed bits”. They are added to the message automatically by the CAN controller if 5 consecutive bits have the same level. If any controller detects an error frame, it increases the internal counter by 8 (if the message has been received/transmitted successfully it decreases by 1). Error frame can be sent by any unit on a bus if the CRC (cyclic redundancy code) value does not match the content of the message. During this time the bus is set to a recessive state, so the error frame can be detected. Counters of the controller can’t be reset by the program so the controller at fault state will no longer be able to interrupt communication until its error counter drops down. After reaching the specified level controller switches to one of the following modes:

- **Error active** – counter value 0 - 127. This is the default state of the controller. Able to transmit data and error frames
- **Error passive** – counter value 128 - 255. The controller can transmit data but is not able to raise an error flag. Instead, it raises a passive error flag (consecutive 6 bits of recessive state)
- **Bus off** – counter value 255. the controller is only able to listen to other devices and receive messages, however, he is not able to interrupt communication for other devices.

Last but not least important redundancy factor is differential signalling. Protocol has just two data wires named `CAN_H` and `CAN_L`. One wire contains the inverse signal of the opposite wire. This provides much better noise immunity and common-mode rejection. Allowing to use of longer cables and higher data rates.

2.2.1 CAN latency measurement

One of the goals of the diploma thesis is to test the implementation of the demo examples that were developed. To test CAN communication, the latency of the communication will be measured. However the latency is not a goal, but just a measure that can provide some evidence that ERIKA works as it is intended.

The latency of the can communication for 1 Mbit/s communication speed is mostly measured in tens or sometimes even hundreds of microseconds. Measuring such latency is a complicated task that requires integration into Linux Kernel drivers or specialized high-cost equipment. Due to this fact already existing solution *canping*[3] is used. This program has been developed by the Control Department of Czech Technical University

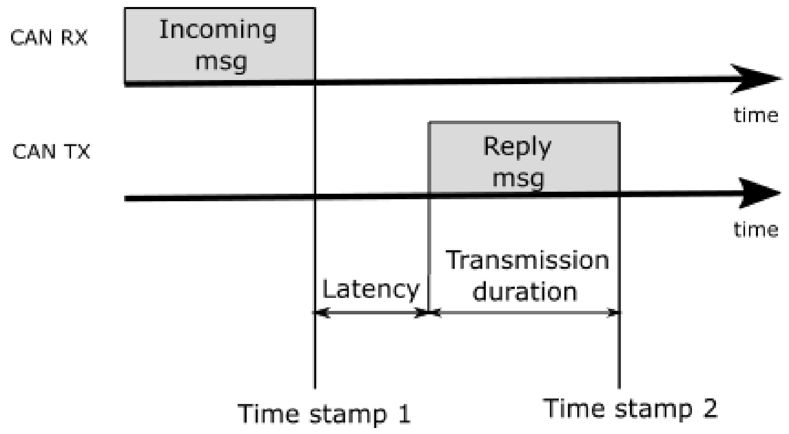


Figure 2.2: Definition of latency in CAN bus communication. Taken from [10].

Figure 2.2 shows the definition of communication latency used in this diploma thesis. To measure the latency program sends the message labeled on a picture as *Incoming msg*. And save a timestamp of a time when the message has been completely sent. Then the program waits until the new message is received, this event is labelled as *Reply msg*. When a new message is completely received program makes another timestamp and subtracts two values.

Since we know the communication speed and amount of bits sent (including bit stuffing bits) we can subtract this value from the measurement. After this, we will obtain a latency made by the device under test. Since the messages sent one-by-one arbitration of the CAN does not affect the test.
To use this program it should be installed and compiled according to the instruction in ”README.md” file.

[3]`https://sourceforge.net/p/ortcan/canping/ci/master/tree/`

2.3 SD card interface

The following section describes the concept of SD cards, and the basics of the communication process using SPI and File Allocation Table, or FAT in short.

2.3.1 SPI

The section is based on information from the source [11].

Single peripheral interface or SPI belong to the most popular communication protocols. It is widely used in embedded electronics and provides a simple interface for inter-controller communication. Some of the most popular use cases are communication with AD/DA converters, sensors or EEPROM memory. Protocol became so popular due to the simplicity of implementation. Almost every microcontroller nowadays has this communication protocol built-in. And even if MCU does not have the support of SPI, it is easy to make SW-defined SPI, sometimes also called *"Bit-Bang SPI"*. The protocol does not have any communication speed limitations. The only limitation is HW capabilities of the devices. Data are sent with each change of the clock cycle. The architecture of the protocol is based on so-called *"Master-Slave communication"*. That means that slave devices can send data only when the master controller requests it.

The hardware interface for this protocol is quite simple and does not require additional integrated circuits like for example CAN transceivers that are additionally installed for CAN bus communication.

For full-duplex communication between two microcontrollers, 4 wires are needed:

- **CS** (Chip select) – from master to specific slave to enable communication (negated)
- **SCLK** (Serial clock) – Clock signal provided by Master
- **MOSI** (Master Output Slave Input) – Data line from master to slave
- **MISO** (Master Input Slave Output) – Data line from slave to masters.

Since the protocol is a serial **bus**. Data lines and clock signals are shared between all connected devices. However, each device should have its own CS signal.

The clock signal is provided by the master and even in case of slave transmission, the master's clock signal is used. Clock signals can have inverted or non-inverted polarity. To avoid confusion first edge is referred to as the leading edge, and the second edge is called the trailing edge. This terminology allows to use same naming for both polarities. Also, protocol allows to set on which edge of the clock data will be sampled. This setting is commonly referred to as *"clock phase"*. An example of SPI communication is shown in

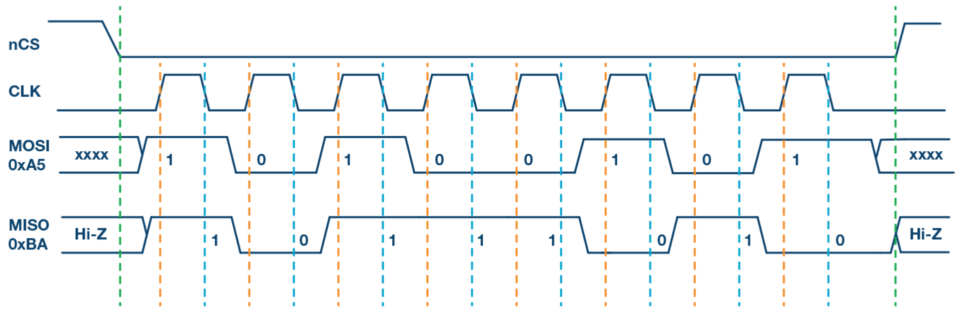


Figure 2.3: SPI communication example. Polarity is not inverted. Data sampled on leading edge. Source [11].

the picture below

TriCore MCU has SPI functionality built-in. With iLLD QSPI (Queued Serial Peripheral Interface) allows to use a similar concept that is used in CAN communication drivers, where messages are put into the memory from where it is accessed using DMA by the QSPI module. However, due to the limitations described in section 6.1 a bit-bang SW definition of the SPI is used. Implementation of which has been provided by *Garrett Motion Inc.*

2.3.2 SD card

The information for the following description of SD card is gathered from FatFs description of SD card [13] and [12].

The secure Digital Card or just SD card originally created as a response to a need for small and reliable data storage. Since its first introduction in 1999, it has become a very important part of embedded devices and industry overall. SD cards can use one of two protocols for communication either I2C or SPI. In this project, SPI communication is used.

SD cards have internal information registers that can be used for control of various SD card features. Some of the registers can be configured by the host controller. For example Operation Condition Register or OCR in short. These registers are configured by the file system or the driver of the SD card during the initialization process. Overall communication with an SD card is a master-to-slave communication. The host controller sends a request or command and the SD card responds with the data request or the result of the command. The SD card in SPI mode has the following list of pins:

- **DO** – Data output or MISO
- **SCLK** – CLK signal
- **DI** – Data input or MOSI
- **CS** – Chip select.

In SPI mode, the data direction on the signal lines is fixed (except for the initialization procedure) and the data is transferred in byte-oriented serial communication. The card is ready to receive a command frame when it drives DO (data output or MISO) high. Because the data transfer is driven by a serial clock generated by the host controller, the host controller must continue to read data, send a 0xFF and get the received byte, until a valid response is detected. The DI signal must be kept high during read transfer (send a 0xFF and get the received data). The response is sent back within command response time, 0 to 8 bytes for SDC, and 1 to 8 bytes for MMC. The CS signal must be driven high to low before sending a command frame and held it low during the transaction (command, response and data transfer if exist). The CRC feature is optional in SPI mode. The card does not check the CRC field in the command frame.

2.3.3 File Allocation Table

The information for this section is gathered from [13] and [14].

File allocation table or FAT in short. Originally was developed in 1977 and used on floppy disks. Then it became commonly used as a standard for Microsoft DOS machines and it is still commonly used for flash disks and some operating systems.

The file system has evolved over its lifespan and has a lot of variations. Each new version adds new features and support for higher-capacity disks. The oldest still commonly used version is FAT16, this format can store files up to 2 GB (without large file support). A newer version of the file system FAT32 can store files up to 4 GB with a maximum partition size of up to 2 TB. Also, an extended version of FAT or exFAT. Which is widely used as a default file system for many Linux distributions (Linux kernel version 5.7 or newer). The extended file allocation table (exFAT) allows to store the files up to 128 PB (petabyte) with the same maximum partition size.

Often FAT is criticized for wasting too much memory using the clusters. If the cluster size is 512 bytes and the file size is just 256 bytes, then the cluster where the file is located will be assumed to be already filled with the data and 256 bytes will be wasted. This problem is partially solved in newer versions of FAT, where the number of clusters is increased, which leads to a smaller size of the clusters and less memory wasted.

Information in FAT is stored in so-called allocation tables. Each table describes how the file is stored in a memory. The table consists of chained items. Each item has a link to the next cluster. The last item contains a special index instead of a link which means EOF (End of the File).

Each FAT system is made of a few regions:

- **Reserved region** to store bootloader and file system information.
- **FAT region** to store file allocation tables (typically two copies are used).

- **Root Directory region** (FAT12/16 only) to store information about files and directories
- **Data region** to store actual data.

The first sector is the so-called *Reserved sector*. This is from where the system is booted.

The reserved region of the system consists of two parts bootloader and file system information. FAT allows to boot the system from the medium where the information about the system is stored. This is the reason behind having a bootloader in a file system. In FAT16 size of the bootloader is 512 bytes.

The second region is the file allocation table. This section contains the data about how the file is stored in memory. Then follows the root directory region, which has information about the internal structure of the file systems. such as folders and files of the name. The last sector is the data region where the file data is stored.

2.4 Infineon TriCore TC387QP

The Infineon TriCore 32-bit family of MCUs has been on the automotive market for more than 20 years. This 32-bit RISC (reduced instruction set computer) based processor has been specially developed to work with real-time applications. It combines the computation power of DSP and real-time systems capabilities. It has been introduced by Infineon company as a concept called *Automotive unified processor* (AUDO in short). Processors became popular in the automotive and industrial segments due to their reliability and safety features such as calculation of CRC, HW watchdogs, and register write locks. In the automotive segment, these MCUs are used in mission-critical places such as engine control units (ECU) and other important safety parts of the car[4].

The family of processors has been developed for more than 20 years and consists of 4 generations of microcontrollers. In this diploma thesis, the MCU from the third-generation TriCore TC387QP is used. Key processor features according to Infineon web-page[5] are:

- 4 cores running at 300 MHz (with 2 additional checker cores delivering 2700 DMIPS)
- Up 10 MB flash with ECC protection
- Up to 1.5 MB SRAM with ECC protection
- 128x DMA channels

[4] `https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/`

[5] `https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/aurix-family-tc38xqp/`

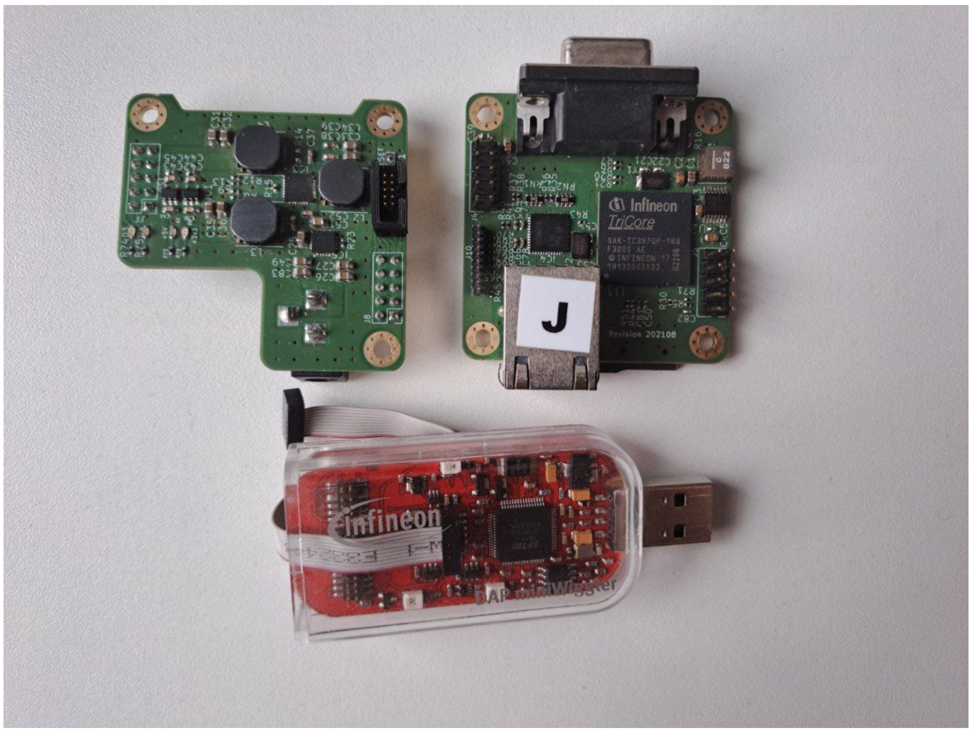


Figure 2.4: Microcontroller board provided by *Garrett Motion Inc.*

- 1 Gbit Ethernet, 12x CAN FD and 6x QSPI
- AUTOSAR 4.2 support[6]
- Developed and documented following ISO 26262/IEC61508 to support safety requirements up to ASIL-D/SIL3

Hardware for this diploma thesis is provided by *Garrett Motion Inc.* as a part of the internal research project. The provided hardware is based on an MCU TC387QP. The hardware has two PCB boards. The first one (bottom board shown top right in Figure 2.4) has a TriCore processor on it and circuitry for peripheral devices such as Ethernet, CAN or SD card slot. The board has an RJ-45 connector and a gigabit Ethernet PHY transceiver, micro SD-card slot and D-sub connector for CAN.

On the top board, the hardware has electrical components related to the power supply and connector for an OEM DAP debugger Infineon Miniwiggler v3 (shown in Figure 2.4). Also, the top desk of the microcontroller has an additional debug ”interface” which is 4 green LEDs. Two of them cannot be controlled by the processor and indicate that voltage circuits are functioning. The other two can be controlled by the processor and will be used for debugging purposes.

[6]AUTOSAR – (AUTomotive Open System ARchitecture) is a global partnership of leading companies in the automotive and software industry to develop and establish the standardized software framework and open E/E system architecture`https://www.autosar.org/`

2.4.1 CAN module

This chapter is based on a description of the CAN module in the user manual [7]

The MCMCAN (or simply CAN) module in TriCore implements Bosch `M_CAN` as CAN nodes. The `M_CAN` performs communication according to ISO 11898-1 according to ISO 11898-4. Module supports classical CAN and CAN FD. The `M_CAN` provides all features of time-triggered communication specified in ISO 11898-4, including event-synchronized time-triggered communication, global system time, and clock drift compensation. A general structure of the module is shown in the Figure 2.5

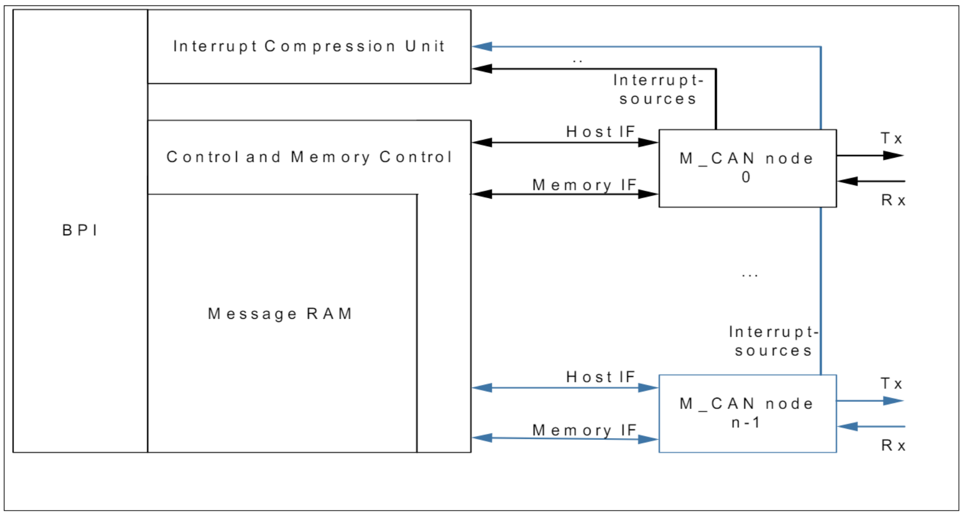


Figure 2.5: CAN module general structure in TC3xx family of MCU. Source [7].

TriCore TC387QP is equipped with 3 modules each as 4 nodes. CAN module provides not only an interface to the CAN bus communication but also implements some of the advanced features such as grouping of the interrupt that is done by the ICU (Interrupt Compression Unit), or module's RAM, which is connected to the common Bus Peripheral Interface (BPI). This memory is a common space for all the nodes to store incoming and outgoing messages.

The size of the memory is enough to fit 64 messages on receive and 32 messages on transmission. The module supports different types of queues for message storage. It is equipped with two FIFO buffers, a queue buffer and Dedicated Buffers. FIFO and Queue buffers can be configured to create a shared memory space from where messages can be accessed by both queues. For example, both FIFO buffers can be set to have one shared space, as well as queue and FIFO buffer can create a shared memory space.

The module provides numerous interrupts that can help with easier debugging and faster application implementation. For example, RX FIFOi Full interrupt (i is the number of FIFO i=0,1), can save a lot of time during implementation because the software does not have to check the state of the

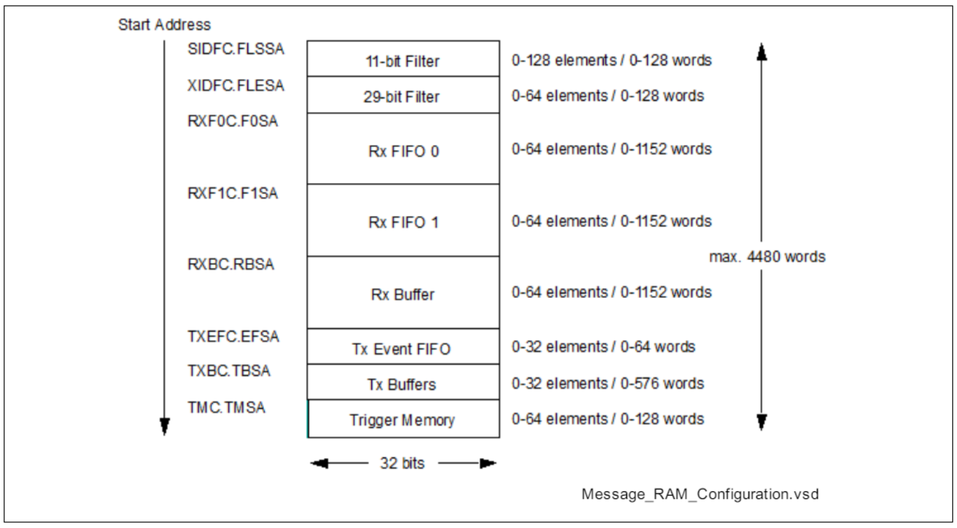

Start Address
SIDFC.FLSSA
XIDFC.FLESA
RXF0C.F0SA
RXF1C.F1SA
RXBC.RBSA
TXEFC.EFSA
TXBC.TBSA
TMC.TMSA
11-bit Filter
29-bit Filter
Rx FIFO 0
Rx FIFO 1
Rx Buffer
Tx Event FIFO
Tx Buffers
Trigger Memory
0-128 elements / 0-128 words
0-64 elements / 0-128 words
0-64 elements / 0-1152 words
0-64 elements / 0-1152 words
0-64 elements / 0-1152 words
0-32 elements / 0-64 words
0-32 elements / 0-576 words
0-64 elements / 0-128 words
max. 4480 words
32 bits
Message_RAM_Configuration.vsd

Figure 2.6: TriCore CAN module memory layout. Source [7].

hardware buffer. It is enough to check if the flag of this interrupt is active in a register. Each interrupt is connected to so-called *interrupt lines* which are used as an instrument that helps ICU to combine interrupts into groups. Each interrupt can be assigned to any core of the MCU.

Also, the module provides support for so-called *transmission events* that are described more in User Manual [7]. The key knowledge about transmission events is that. after the node has transmitted a message on the CAN bus, the Message ID and timestamp are stored in a Tx Event FIFO element. To link a Tx event to a Tx Event FIFO element, the Message Marker from the transmitted Tx Buffer is copied into the Tx Event FIFO element. The purpose of the Tx Event FIFO is to decouple handling transmit status information from transmit message handling i.e. a Tx Buffer holds only the message to be transmitted, while the transmit status is stored separately in the Tx Event FIFO. This has the advantage, especially when operating a dynamically managed transmit queue, that a Tx Buffer can be used for a new message immediately after successful transmission.

2.4.2 SPI module

The information about this module is gathered from [7]

The main purpose of the QSPI (Queued SPI) module is to provide synchronous serial communication with external devices using clock (CLK), data-in (MISO), data-out(MOSI) and slave-select(CS) signals. The focus of the module is set to fast and flexible communication: either point-to-point or master-to-many slaves communication. The term “Queue Support” is used in this context to describe the functionality implemented for comfortable switching of the timing configuration of the QSPI frames, depending on the

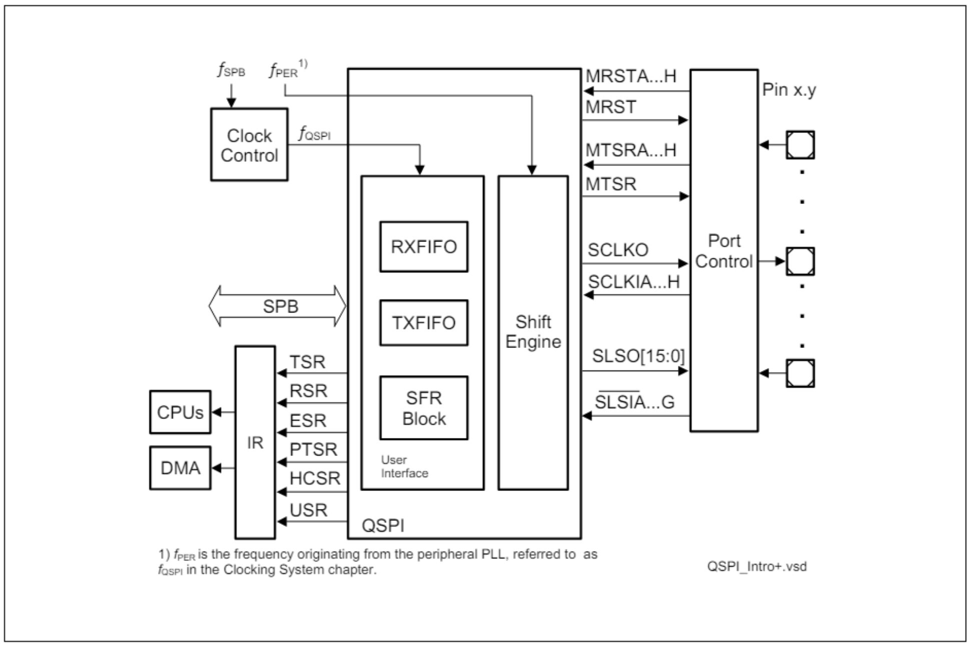


Figure 2.7: SPI module general structure. Taken from [7].

slave select signal which is to be activated. The main feature of the module is the possibility to take both the configuration and data to the transmission buffer and to track down which transmission buffer entry is configuration, and which data. The main features of the module are the following:

- Master and Slave Mode Operation
- Interoperability with SSC and USIC modules of Infineon microcontroller families, and with popular (Q)SPI interfaces of multiple suppliers
- QSPI supports control and data handling by the DMA controller
- Interrupt generation

Overall the structure of the QSPI module and the main features that it provides are very similar to the CAN module. It also has internal FIFO buffers for received and transmitted messages and even iLLD configuration in its ”philosophy” is very similar to CAN module configuration. It also has a configuration structure where every part of the module is initialized. The general structure of the QSPI module looks the following way.

2.4.3 Context Save Area

TriCore architecture supports some of the operating system features on a hardware level. One of the features is CSA registers (Context Save Area). This chapter is a short version of the description of the context save area in [15]. Erika Enterprise uses this feature to switch between tasks and interrupts much faster.

Most embedded and real-time control systems are designed according to a model in which interrupt handlers and software-managed tasks are each considered to be executing on their own ‘virtual’ microcontroller. That model is generally supported by the services of a Real-time Executive or Real-time Operating System (RTOS), layered on top of the features and capabilities of the underlying machine architecture. In the TriCore™ architecture, the RTOS layer can be very ‘thin’ and the hardware can efficiently handle much of the switching between one task and another. At the same time, the architecture allows for considerable flexibility in the tasking model used. System designers can choose the real-time executive and software design approach that best suits the needs of their application, with relatively few constraints imposed by the architecture. The mechanisms for low-overhead task switching and for function calling within the TriCore architecture are closely related. Contexts, when saved to memory, occupy 16-word blocks of storage, known as Context Save Areas (CSAs).

The context types are:

- Upper context: Consists of the upper address registers A[10] to A[15] and the upper data registers D[8] to D[15]. The upper context also includes PCXI and PSW. These registers are designated as non-volatile for purposes of function-calling (their contents are preserved across calls).
- Lower context: Consists of the lower address registers A[2] to A[7], the lower data registers D[0] to D[7], A[11] (Return Address) and PCX

The architecture uses linked lists of fixed-size Context Save Areas. A CSA is 16 words of memory storage, aligned on a 16-word boundary. Each CSA can hold exactly one upper or one lower context. CSAs are linked together through a Link Word. The Link Word includes two fields that link the given CSA to the next one in a chain. The fields are a 4-bit segment and a 16-bit offset. The segment number and offset are used to generate the Effective Address (EA) of the linked CSA (see architecture manual[15]). Incrementing the pointer offset value by one always increments the EA to the address that is 16-word locations above the previous one. The total usable range in each address segment for CSAs is 4 MBytes, resulting in storage space for 216 CSAs. The upper context is saved automatically as a result of an external interrupt, trap or function call. The lower context is saved explicitly through instructions.

Architecture switches context when one of the following happens:

- Interrupt or Trap
- CALL - Function Call
- BISR - Begin Interrupt Service Routine
- SVLCX - Save Lower Context
- STLCX - Store Lower Context

- STUCX - Store Upper Context.

The upper and lower contexts are saved in Context Save Areas (CSAs). Unused CSAs are linked together in the Free Context List (FCX). CSAs that contain saved upper or lower contexts are linked together in the Previous Context List (PCX). The contents of the FCX register always point to an available CSA in the Free Context List. That CSAs Link Word points to the next available CSA in the free context list. Before an upper or lower context is saved in the first available CSA, its Link Word is read, supplying a new value for the FCX. To the memory subsystem, context saving is therefore a read/modify/write operation. The new value of FCX, which points to the next available CSA, is available immediately for subsequent upper or lower context saves.

Chapter 3

Compiling Erika with Aurix Development Studio

This section is an overview of all the changes that have been done to Erika Enterprise so it can be compiled using the Tasking compiler under Aurix Development Studio.

The coding project creation process is described in [17] and this chapter is based on this technical report.
The latest version of Erika Enterprise 3 supports neither the TC387 microcontroller nor compilation with AURIX Development Studio, which contains a free version of the Tasking compiler. However, Erika supports a similar microcontroller TC39x as well as the Tasking compiler, but in its commercial variant. Erika applications are typically configured and compiled via a graphical IDE called RT-Druid. This tool is responsible for:

1. Generating parts of Erika source code based on the application configuration and
2. Compile both Erika and the application with the C compiler

However, due to the use of a free version of the Tasking compiler, this standard workflow cannot be used, because the compiler works when run within ADS (Aurix Development Studio) and does not work when run within RT-Druid. Therefore, we extend the workflow and use AURIX Studio in addition to RT-Druid. Specifically, the following changes have been made:

- Support for the TC387 microcontroller was added
- Support for executing the build under AURIX Studio
- Initial support for using iLLD drivers (this will need to be further extended)

The figure below explains the basic workflow with Erika. The left light blue rectangle represents standard Erika workflow. Due to the use of a free Tasking compiler, this standard workflow cannot be used, because the compiler does not run under RT-Druid. Therefore, we extend the workflow and use AURIX Studio in addition to RT-Druid.

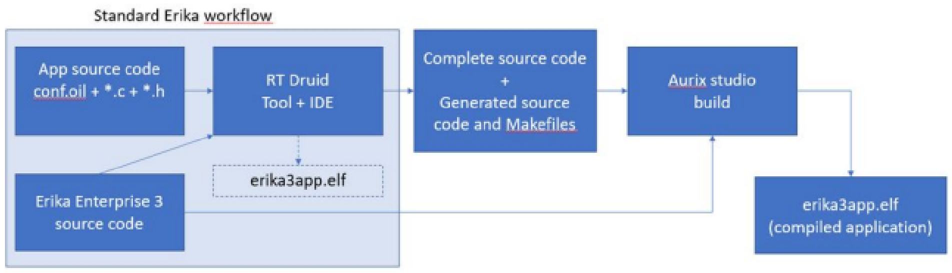

Standard Erika workflow
App source code conf.oil + *.c + *.h
RT Druid Tool + IDE
erika3app.elf
Erika Enterprise 3 source code
Complete source code + Generated source code and Makefiles
Aurix studio build
erika3app.elf (compiled application)

Figure 3.1: Development workflow. Source of the image [17].

3.1 Installation

This section explains basic settings that need to be done for example compilation.

1. Install AURIX Studio.
2. Download modified Erika sources from GitHub[1] and unpack them to some directory.
3. Follow the Quick start[2] guide on the Erika wiki.
   a. Install JRE
   b. Installation of Cygwin can be skipped. It will be replaced with AURIX Studio.
   c. Download and install RT-Druid by unpacking it to any directory..
   d. Run RT-Druid via eclipse.exe
   e. Configure RT-Druid Preferences (discussed later in this section)

Before attempting to compile the project following steps should be done in configuration. In RT-DRUID Preferences→Oil→Erika Enterprise enter the path to modified Erika sources via *Manual* edit box. Then select *Plugins* as a location of the RT-Druid extension pack (e.g. *eclipse/plugins/com.eu.evidence.ee3_3.0.1.20190524_gh65/ee_files/rtdruid.ext*).

Then open Oil→Generator Properties and set TriCore→TASKING CTC Compiler to the compiler path under Aurix Studio installation. This will be something similar to *C:/Infineon/AURIX-Studio-1.5.2/plugins/com.infineon.aurix.tools_1.5.2/build_system/tools/Compilers/Tasking_1.1r7.*. For other versions of Tasking compiler and Aurix Development Studio the path can be different.

After, copy *cygpath.exe* from the utils folder of the downloaded Erika Enterprise to the *tools/make* folder of the Tasking compiler (something similar to e.g. *C:/Infineon/AURIX-Studio-1.5.2/plugins/com.infineon.aurix.tools_1.5.2/build_systemtools/make*). The program *cygpath.exe* is available in the Cygwin environment and Erika build system calls it. Since we run

[1]`https://github.com/Darth-Bujar/erika3-tc38x`
[2]`https://www.erika-enterprise.com/wiki/index.php/Quick_start_guide`

the build without Cygwin, under AURIX Studio, we must provide a tool that provides similar functionality there. Note that for some targets, Erika can compile itself without Cygwin, but it doesn't work for the Tasking compiler.

3.2 Creating a project

To create a project that enables both iLLD drivers and Erika compilation under Tasking. The next steps should be performed:

1. Create an AURIX Studio project for the TC38xQP_A-Step device. In the following, we assume the project is named erika-test.
2. Create an RT-Druid project (fully called RT-Druid v3 Oil and C/C++ Project) in the folder "ee" created in the root folder of the project created in the first step (describe in more detail in the same chapter)
3. Specify the make files and location of the modified version of Erika Enterprise
4. Compile the configuration of Erika using RT-DRUD
5. Compile the rest of the application code using ADS.

Let's get through the steps in more detail one by one. The first step does not need any further explanations. However, the second step will be described in more detail.

In the New Project window of RT-DRUID select a project ***"RT-Druid v3 Oil and C/C++ Project"***, then "uncheck" *Use default location* box and specify the location of the file as follows. Open a location of the new ADS project created in step 1. Create a new folder named "ee" (or any other name). Select the "ee" folder as a location of the project so the resulting path looks similar to ***path_to_ads_project/erika_test/ee***. Note that the RT-Druid workspace should be at a different location than the AURIX Studio workspace.

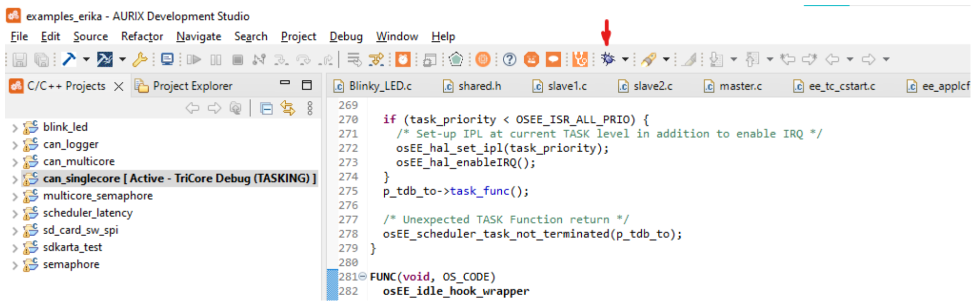
Click next and in the new opened window choose a project "TriCore → AURIX 2G → TASKING → MultiCore + Blink" and press ***Finish***. After the project was created RT-DRUID will open conf.oil file and automatically compile it. The following message will appear.

```
make doc
Cannot run program "make": Launching failed

Error: Program "make" not found in PATH
```

This is expected, as *cygwin* and its tool *make* are not installed. Instead, the project is compiled using Aurix Development Studio.

The third step is to make ADS compiling using Erika Enterprise files. To do this open properties of the project erika_test that has been created



```

  if (task_priority < OSEE_ISR_ALL_PRIO) {
    /* Set-up IPL at current TASK level in addition to enable IRQ */
    osEE_hal_set_ipl(task_priority);
    osEE_hal_enableIRQ();
  }
  p_tdb_to->task_func();

  /* Unexpected TASK Function return */
  osEE_scheduler_task_not_terminated(p_tdb_to);
}

FUNC(void, OS_CODE)
  osEE_idle_hook_wrapper
```

Figure 3.2: Location of the Debug Active Project button in ADS (located under red arrow)

in ADS in the first step. Then open C/C++ Build→"Generate MakeFiles Automatically" property should be changed to the location of Erika's makefiles in the project ***{workspace_loc:/erika-test}/ee/out***. After the application is ready to be compiled by the Tasking compiler using Aurix Development Studio. To compile the application and download it to the microcontroller click on ***Debug Active Project*** button in ADS.

Note that the modifications of Erika Enterprise files in the project will be, or can be overwritten by the RT-DRUID. To keep changes, modify Erika Enterprise files in the location of the modified version of Erika. Recompilation of Erika Enterprise configuration (means almost any changes to conf.oil file) can be done only in RT-DRUID or using shell script[3] that can be added to make file. That means that in case of any changes to the configuration of the Eika in conf.oil. Firstly RT-DRUID (or shell script) should recompile the configuration of Erika Enterprise and only after, a full compilation of the project in ADS can be done. So steps 4 and 5 should be followed in case of any changes to RTOS configuration.

To run any project that is already done in the way that has been described in this chapter. It is enough to import the project to both RT-DRUID and ADS.

3.3 Using iLLD drivers with Erika

iLLD drivers is a proprietary set of low level drivers developed by Infineon specifically for their TriCore microcontrollers. Erika does not have an implementation of iLLD built in. Under the hood, RTOS uses drivers that have been developed by Erika's developers themselves. That means that some of the functions can be accessed in both ways either from iLLD (for example using IfxScuCcu_getSourceFrequency) or by using Erika Enterprise function (osEE_tc_get_osc_freq). However this project concentrates on the integration of iLLD drivers, so only the iLLD's functions will be used to configure hardware.

[3]https://www.erika-enterprise.com/wiki/index.php/RT-Druid_command_line

3.3.1 Erika's source code changes

Since Erika officially does not support TriCore TC387QP and iLLD, a few changes have been made to Erika's source code. These changes follow the initial work that has been done by the supervisor of this project Ing. Michal Sojka, Ph.D. That includes changes to a build system so the project can be compiled using Aurix Development Studio.

Since the TC387 architecture is not much different from TC39x, only the linker script should be changed. The main difference is that TriCore TC39x has two more cores. To cut those two additional cores, macro *__PROC_TC38X__* has been added to Erika's linker script file (file extension type .lsl). This macro is used to distinguish two types of processors that share the same Erika's linker script file. At the top of the file ee_tc_tasking_flash.lsl the following code is added to determine which processor is compiled and which linker script file should be used:

```
#if defined(__PROC_TC39X__) || defined(__PROC_TC38X__)
#if defined(__PROC_TC38X__)
#include "tc38x.lsl"
#endif
#if defined(__PROC_TC39X__)
#include "tc39x.lsl"
#endif
```

Additionally in the same file, script sections that are related to cores 5 and 6 (which TC387 does not have) are put under condition of macro *__PROC_TC39X__* existence, so these parts of linker script are not compiled for TC387 processor.

Support for iLLD drivers is also added to the project. Since we are using Erika's MAKE files to compile the project, they should be extended with a path to the iLLD drivers header and source files.

This can be achieved by modifying *ee_arch_compiler_tasking_ctc.mk*. Only modules that have been used in this diploma thesis were added to the make file. The changes are separated into 4 commits in GitHub repository[4]. Each commit represents support for one of the modules that have been used during this project. Namely, modules CAN, QSPI, GPT and STM are added. These commits can be used as a template for future extensions of iLLD support. Without these changes, compilation will fail when compiling iLLD files, since Erika's MAKE file cannot find them.

Let's take a closer look at one of the commits that adds support for the GPT12 module. First of all the path to the header files should be included in the make file, mentioned above. This can be done by modifying variable *INCLUDE_PATH* contained in the file. The second step is to add the source files of the drivers (file extension .c) to the list of compilable sources. This can

[4]`https://github.com/Darth-Bujar/erika3-tc38x/`

be done by adding a new source to the end of list *OS_EE_APP_CFG_SRCS* in the same file. The resulting change should look similar to the following code snippet.

```

INCLUDE_PATH += $(call short_native_path,
 $(abspath $(wildcard
 ../../Libraries/iLLD/TC38A/Tricore/Gpt12/Std))
INCLUDE_PATH += $(call short_native_path,
 $(abspath $(wildcard
 ../../Libraries/iLLD/TC38A/Tricore/Gpt12/IncrEnc))

OS_EE_APP_CFG_SRCS += \
  ... \
  ... \
  ../Libraries/iLLD/TC38A/Tricore/Gpt12/Std/IfxGpt12.c
```

3.3.2 Startup initialization difference

Erika Enterprise has a slightly different startup initialization code than iLLD. One of the main differences is that Erika does not configure SCU (System Control Unit) and CCU (Clock Control Unit) registers in the way that iLLD does it. Due to this fact, a configuration of the clock should be done by the application of the MCU. The easiest way is to use the default (already existing in iLLD) configuration of these registers. The default configuration can be loaded into the variable of type *IfxScuCcu_Config* using the function named *IfxScuCcu_initConfig*, and the following function *IfxScuCcu_init* will load the configuration into the MCU's registers.

The mentioned workflow for SCUCCU registers initialization was not a part of the changes made to the Erika Enterprise code. The following code should be added before the initialization of the specific hardware that requires these registers to be configured, such as CAN module. In the case when more peripherals require it, code can be added to the "main" function before the start of the OS.

```
// SCU CCU configuration handler
IfxScuCcu_Config IfxScuCcu_sampleClockConfig;

// PLL & clock initialization
IfxScuCcu_initConfig(&IfxScuCcu_sampleClockConfig);
IfxScuCcu_init(&IfxScuCcu_sampleClockConfig);
```

Now let's move to the configuration of the PLL in the System PLL Configuration register (SYSPLLCON). According to the default setting mentioned above for SCU CCU registers, PLL0 with a frequency f_{PLL0} is used as a primary source for most of MCU's peripherals, including CPU clock frequency f_{CPUi} (where i is the number of cores i=0..3), and the frequency of the PLL is set to be 300 MHz. This is calculated using values P, N and K2[5] from the SCU CCU configuration register and the formula from:

$$f_{PLL0} = \frac{N * f_{osc0}}{P * K2} = \frac{(29+1) * 20}{(0+1) * (1+1)} = 300MHz \tag{3.1}$$

The frequency of the CPUi is set to be the value of the f_{SRI} divided by the value of the register *SRIDIV* in case of the default SCU CCU configuration from iLLD. The value of the register is set to zero for all cores. That means that each core of the MCU is running on the frequency $f_{cpu} = f_{SRI}$, where $f_{SRI} = f_{PLL0}$ according to the configuration of *CCUCON0.CLKSEL*.

[5]According to User Manual [6] values P, N and K2 in registers should be increased by one to get real value

Chapter 4
Basic Erika examples

This chapter describes two basic examples and Erika's basic configuration is then used with slight modifications in other demos developed as part of this thesis. All examples that were developed in this diploma thesis can be found on GitHub[1].

4.1 Scheduler example a GPIO

A small example has been done to verify the correctness of the scheduler work. The example consists of two tasks that are activated periodically by an *ALARM*.

Each task has its own *ALARM*. This Erika's object is used to periodically activate some event or task. More about it later in section 5.1.1.

When the task with low priority is running and the task with high priority is activated. The low-priority task should not be preempted if it does not have a parameter *SCHEDULABILITY* set to NON (non-preemptive).

The task with low priority prints "Y" to the console each time it gets activated and the task with high priority prints "X" to the console when running activated.

The following output was obtained by running the program in two variations. In the first experiment, low priority task has a parameter *SCHEDULABILITY* set to FULL. This means that this task can be preempted. In the second experiment, the same parameter is set to NON, so the low-priority task cannot be preempted. According to the output logs the scheduler works correctly. Each time the program switches between the tasks context of a task is switched using CSA discussed earlier in 2.4.3

```
# Experiment 1
Y
Y
X <- 500 ms alarm expires and activates HIGH_priority_task
Y
Y
```

[1] https://github.com/Darth-Bujar/examples_erika

```
# Experiment 2
Y
Y
Y <- 500 ms alarm expires and activates HIGH_priority_task,
                 but LOW_priority_task cannot be preempted
Y
Y
```

When the high-priority task is activated it will switch on the LED on the top PCB board of the hardware. The low-priority task switches off the LED each time it completes the busy wait cycle.

This is done using the iLLD drivers for GPIO (general-purpose input/output). It is enough to configure the mode of the GPIO and then set, or reset the state of the GPIO. The following snippet shows how to initialize and change the state of GPIO.

```
// Definition of GPIO port, pin
#define LED_D110 &MODULE_P20, 14

// Initialize GPIO
IfxPort_setPinModeOutput(LED_D110,
                IfxPort_OutputMode_pushPull,
                IfxPort_OutputIdx_general);

// Set GPIO
IfxPort_setPinHigh(LED_D110);

// Reset GPIO
IfxPort_setPinLow(LED_D110);
```

The source code for this example is located in a folder *basic_examples/scheduler*.

4.2 Semaphore example

Another small example has been done to verify that the semaphore works correctly on TC387. The program has 3 tasks, two of them are so-called consumer tasks and one is a producer task. The producer task has the lowest priority. When producer tasks post a semaphore a consumer task with higher priority should take a semaphore, thus it will be proven that the implementation of the semaphore works correctly..

Let's take a look at how a semaphore can be created in Erika RTOS. First of all, semaphores should be defined in conf.oil

```
USEEXTENSIONAPI = TRUE
{
  SEMAPHORE = DEFAULT { NAME = "S"; COUNT=0; };
};
```

Be aware that semaphores can be used only in extended task types. The code above defines a semaphore with the name ”S” and an initial count of 0. To access the semaphore, firstly it should be added into the .c file as an external variable of type SemType e.g. *extern SemType S;*. Then to get the semaphore, a function *WaitSem* is used, and to post a semaphore, use function *PostSem*. Both functions take a pointer to the semaphore as an argument.

All three tasks of the application are set to be activated periodically by an *ALARM* with a period of 3, 2 and 1 second. The lower the priority of the task the longer its ALARM period. Additionally, the alarm for middle-priority task (or low-priority consumers) has 10 ticks offset from other tasks, so the timing is ”asymmetric” and low-priority consumer may take the semaphore after high priority task or before it.

The result of the program run is written into the debug console.

```
Z posting one semaphore
X Waiting for semaphore
X get the semaphore

Y Waiting for semaphore
Z posting one semaphore
Y get the semaphore

X Waiting for semaphore
Y Waiting for semaphore
Z posting one semaphore
Y get the semaphore

Z posting one semaphore
X get the semaphore
```

In the console log, the task ”Z” is a producer task. Task ”X” is low priority consumer task and ”Y” is high priority consumer task. As we can see when the producer posts a semaphore and the high-priority consumer task is not already waiting for it, the low-priority consumer task will take the semaphore. If both of the consumer tasks are waiting for the semaphore then the task with higher priority will get the semaphore. That proves that the semaphore works as it is intended.

The source code for this example is located in a folder *basic_examples/semaphore*.

Chapter 5

CAN communication

To test if the iLLD can work with ERIKA RTOS, there is no better way than to write an application code and test it. Since TriCore is a multi-core MCU, two examples will be done. The first example is a single-core example and it was done as a lightweight implementation of CAN communication where the message is immediately processed after being received.

The second example is a multi-core example, that may have slightly worse performance in latency measurement. However, this application example should use a bit more sophisticated features of the CAN module, such as message filtering.

In general, both applications can be divided into the following list:

- Start-up setting of the HW.
- Interrupt handling by ERIKA RTOS
- Basic functionality of CAN (sending and receiving messages)
- CAN hardware buffers (multi-core example only)
- CAN hardware filtering (multi-core example only)

To verify the performance of the CAN communication program *canping* (mentioned in section 2.2.1) is used together with oscilloscope measurement. The general flowchart for both examples is shown in figure 5.1. Program has a ”reactive” behaviour so firstly CAN message should be sent to the MCU by some external source (USB-CAN converter connected to PC etc.). Than the message is accepted by the TriCore, and when the message is stored in the RAM of the CAN module, a new message interrupt flag is raised. After this, ERIKA should activate the ISR (Interrupt service) for this specific interrupt.

In ISR, the program will read data from the HW buffer and store them in the software buffer (or simple variable). After the message is stored program will start to process the stored messages. Both examples are set to reply to the received (original) message with a *reply message*. The CAN ID of the reply message is set to be the same format (standard or extended) as the CAN ID of the received message. The ID of the original message is increased by decimal 1 in a reply message. The same procedure is applied to data, but only the first byte of the received message is used. Thus the reply message always has only one byte of data. After the reply message has been generated it is added to the transmission queue transmission process is fully handled by the CAN module. Additionally, MCU sent a so-called *keep-alive message* that indicates that the controller has not been trapped into some exception and is still running as it is intended.

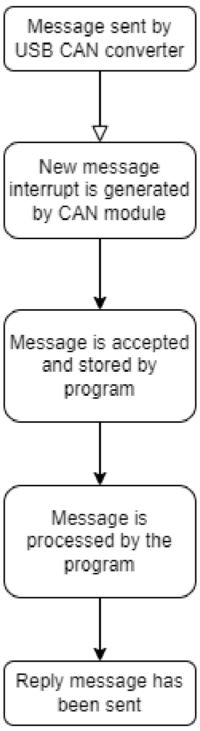

Message sent by USB CAN converter
New message interrupt is generated by CAN module
Message is accepted and stored by program
Message is processed by the program
Reply message has been sent

Figure 5.1: General sequence diagram for CAN examples

To provide users with a better understanding of what is happening in the application *"Debug mode"* has been added to the program. This mode can be activated by sending a CAN message to the MCU with a specific ID (0x1) and a value one in the first byte of the message. After this MCU will automatically start to write logs into the debugger serial interface.

To switch back to the normal mode microcontroller should receive a message with the same ID but a value of zero in the first byte. Be aware that this mode significantly reduces the performance of the microcontroller since serial writing to the debugger is done in a blocking way. Example output of the controller debug mode.

```
CAN driver initialization: Complete
RX CAN ID: 0x154 data: 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
TX: Success
RX CAN ID: 0x152 data: 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
TX: Success
```

5.1 Single-Core Example

This example was made to be a simple benchmark for measuring the latency of CAN communication using iLLD drivers and Erika Enterprise RTOS. This program uses interrupt of category 2, which can call RTOS primitives and it is scheduled as a task with a high priority. Withing this interrupt control should read the message that is stored in the RX HW buffer, process this message, and send a reply message. Additionally, the microcontroller is configured to send *"keep-alive"* messages that indicate that the controller is still alive and able to access CAN BUS. The message contains useful debug information. In the case of the single-core application, it is a counter of received messages. To send this message periodically, the program has an ALARM that activates a task once per second.

5.1.1 Erika configuration

The beginning of the configuration is very similar to the configuration discussed in section 2.1.1. However, the number of cores for the single-core example is reduced to 1. The exact configuration for the single-core project can be found in *`can_singlecore/ee/conf.oil`*.

Interrupt and task configuration

The program has one interrupt handler of category two (ISR2) that services the RX new message interrupt flag for FIFO 0, or *RXF0N*. The interrupt handler is set to be serviced by CPU0 with a priority of the interrupt set to 10. Be aware that the following restriction applies: all interrupts of Category 1 must have a higher or equal hardware priority compared with interrupts of Category 2. This limitation has been introduced to avoid various rescheduling problems [2]. Configured this way the interrupt will be triggered each time a new message appears in the FIFO0 buffer of the CAN module. If the interrupt's handler is not specified then the function with the same name as an interrupt (in this case `can_ISR_RX_handler`) is used instead. If the function with such a name is not found, compilation will fail. The exact definition of interrupt handler from the configuration file:

```
ISR can_ISR_RX_handler
{
    CPU_ID = 0x0;
    CATEGORY = 2;
    SOURCE = "CAN_CAN0_INT0";
    HANDLER  = "can_ISR_RX_handler_func";
    PRIORITY = 10;
};
```

The parameter *SOURCE* must be set to provide Erika with a definition of the hardware interrupt flag. The list of sources can be found in User Manual [7].

This structure definition has a similar meaning as calling the macro *IFX_INTERRUPT* that is used to assign a function to the ISR in iLLD.

The next part of the configuration is the task that sends the *keep-alive message*. The message should be sent periodically and serve as an indication that the controller did not get stuck in some trap or infinite loop. To trigger the task periodically, typically *ALARM* object is used, the definition of the alarm is discussed later in the next section 5.1.1.

The task definition is very simple in this case. No synchronization primitives such as *RESOURCE* or *EVENT* are used, and the task does not require a private stack. The task is defined as a *TASK* object with identifier *can_keep_alive_task*. Since there is no other task the priority is set to 1. The schedulability of the task is set to NON, which means that the task cannot be preempted.

```
TASK can_keep_alive_task
{
    CPU_ID = 0x0;
    PRIORITY = 1;
    ACTIVATION = 1;
    SCHEDULE = NON;
};
```

Alarms and counters

The *ALARM* object is a time-triggered object that can activate a task, *EVENT*, or set of events. The definition of such an object contains the source of the ticks, the tick offset, the period in ticks and a task or event to be activated. An alarm object can be used to trigger the tasks on a different core than the core it is assigned to. The source of the countdown can be any *COUNTER* object.

From Erika's OIL wiki [4], *COUNTER* object is the timing reference that is used by alarms. There are two main types of counters. The first one is a software counter and it is not connected to the HW timer, this type of counter is serviced during the system timer service call. *COUNTER* can be set to be connected to a hardware timer (TYPE = HARDWARE). If it is so, RT-DRUID generates an additional handler for this specific counter. The counter can be also equipped with priority and the parameter *SYSTEM_TIMER* which defines if the timer is serviced as an OSEK system timer or not. Also, a counter object has parameters that specify the maximum and minimum values of the counter, the number of ticks per base and the number of seconds per tick. The exact definition of the pair counter and alarm in this application look as follows:

```
COUNTER system_timer_master
{
    CPU_ID = 0x0;
    MINCYCLE = 1;
    MAXALLOWEDVALUE = 2147483647;
    TICKSPERBASE = 1;
    TYPE = HARDWARE
    {
        DEVICE = "STM_SR0";
        SYSTEM_TIMER = TRUE;
        PRIORITY = 2;
    };
    SECONDSPERTICK = 0.001;
};
ALARM alarm_1s
{
    COUNTER = system_timer_master;
    ACTION = ACTIVATETASK { TASK = can_keep_alive_task; };
    AUTOSTART = TRUE { ALARMTIME = 0; CYCLETIME = 1000; };
};
```

This configuration code uses a device `STM_SR0` as a source of ticks for *COUNTER*. STM (System Timer) is a TriCore module, that contains a few free-running 64-bit counters. Erika Enterprise supports `STM_SR0` and `STM_SR1` as system timer devices.

The timer increases its value once per millisecond as it is defined in *`SECONDSPERTICK`* until the maximum value of 2147483647 ticks is reached (31-bit maximum value). An alarm is set to activate the task with an identifier *`can_keep_alive_task`* which is a task that sends keep-alive messages. The task is triggered every 1000 ticks with each tick equal to 1 ms task will be activated each second.

5.1.2 CAN drivers – iLLD

As it has been mentioned above, the single-core example is done as a simple benchmark-oriented. The program should successfully receive the CAN message and then it will generate and send a reply message immediately without storing a message. Interrupt configuration should be done on both sides in Erika's configuration and CPU registers. For Erika, it is a definition of the interrupt in the configuration file (conf.oil), for CPU configuration is done through an already pre-made iLLD CAN module configuration structure that contains all needed parameters for the CAN module configuration.

CAN Node configuration

The first step is to select the CAN module. MCU that is used in this diploma thesis has 3 CAN modules and only *MODULE0* is used in all CAN examples. Each CAN module, as well as the CAN node, has a slightly different definition of available pins. To have a better view of the available pins for each module I would recommend checking the file located in *../iLLD/TC38A/TriCore/_PinMap/IfxCan_PinMap.c*. This file contains the list of all available pins for each CAN module and each CAN node.

According to the schematics of the HW provided by *Garrett Motion Inc.* pins P20.7 and P20.8 are connected to the D-sub connector that is used for CAN communication. So only CAN module 0 and its node 0 can be used.

After selecting the module and its node and verifying the pin assignment according to the schematics, a type of node should be set. In both single-core and multi-core CAN examples the same node is used for both directions of communication, so it is set to value *IfxCan_FrameType_transmitAndReceive*. This value is then used by iLLD to initialize buffers according to the selected direction of communication. In case when only one direction of the communication is chosen, only one buffer (RX or TX) will be initialized.

The next setting is the baud rate of the CAN. For all projects that have a CAN bus, the communication speed is set to be 1 Mbit/s standard speed and 4 Mbit/s for the data rate switching in CAN FD.

Configuration of the RX and TX buffers is done in the following way. The mode of the TX buffer is set to be a queue of the length 2 and the size of one element is `IfxCan_DataFieldSize_64`, which is the maximum size of the message in CAN FD communication. Received messages are set to be stored in the FIFO0 buffer. The FIFO mode is set to value *IfxCan_RxFifoMode_blocking*. That means that the module is not allowed to overwrite any already stored messages. If the buffer is full, then the RX FIFOi Full interrupt is generated and until the interrupt is serviced, the node will not accept any new messages to this buffer. For each lost message, interrupt RX FIFOi Message Lost (RXF0L) is generated and should be serviced. Another available option for the ”buffer full behaviour” is to overwrite an already stored message with a new one. However, this behaviour leads to the loss of messages which should not happen during normal communication. Furthermore, such a behaviour will lead to incorrect results for the communication latency tests.

The transmission buffer data field size is set to only 8 bytes and stored in a queue. The buffer size (amount of messages that it can hold) is defined by a *CAN_BUFFER_SIZE* macro (expands to 1) for both buffers.

Be aware that the saviour change of the TX/RX buffer should be made alongside a change of the start address of a particular buffer and buffer listed below.

The next part of the configuration is an interrupt configuration in iLLD (which means the configuration in MCU). It consists of 4 main parts. First of all, interrupts should be enabled in the list of interrupts in the communication structure. This setting will set a specific bit in the IE (Interrupt Enable) register of the CAN module according to the register description in the user manual [7].

After interrupt generation has been enabled, for the correct functioning interrupt should be provided with a valid description for the ICU (Interrupt Control Unit). This will cover the remaining 3 steps of the interrupt configuration. The type of service (a module or CPU to serve the interrupt), priority, and ICU interrupt line should be assigned to it.

The configuration for the ICU is done through the interrupt group names, rather than the name that has been listed in the IE register. For the ”conversion” between two variations of the interrupt naming, the picture in Appendix B is added. The same picture can be found in the user manual [7].

When interrupts are being generated two registers change their values. First register IR (Interrupt Register) represent the interrupts themselves. The second register ISREG (Interrupt Signalling Register) represents groups of interrupts. if the group of interrupts has at least one pending interrupt service request, then the bit in ISREG will be set to TRUE. For example, the interrupt Lost message used in the multi-core example is configured in group alrt (alerts). When the interrupt flag is active entire group is also active.

The entire configuration is to big to add it here as a code snippet, but it can be found in the file *can_singlecore/ee/can_control.c* variable name is *IfxCan_Can_NodeConfig canNodeConfig*.

iLLD with CAN module

After the configuration of the CAN module has been discussed let us move to the functions that service the CAN module. These functions can be split into 3 main categories.

- Module and Node initialization.
- Servicing the interrupts.
- Reading and sending CAN messages.

Firstly the module should be initialized. To initialize CAN on TriCore microcontroller code should first initialize the chosen CAN Module (in this case MODULE_CAN0). And only after that chosen CAN Node is initialized (IfxCan_NodeId_0). Initialization of the CAN module is done in the function *void can_init(void)*. Be aware if registers SCU and CCU are not set to enable the CAN clock, then the controller will stuck in an infinite loop during the CAN module initialization procedure more about it in section 3.3.2.

```
// CAN module configuration structure
IfxCan_Can_Config canConfig;
// CAN module configuration handler
IfxCan_Can canModule;

// CAN initialization
```

```
IfxCan_Can_initModuleConfig(&canConfig, &MODULE_CAN0);
IfxCan_Can_initModule(&canModule, &canConfig);
IfxCan_Can_initNode(&canNode, &can_node_config);
```

The next step is the interrupt service routines. To handle the interrupts that are generated by the HW iLLD has a straightforward interface. In the source code of the drivers, an enumerated structure is defined. This structure represents all the possible interrupt sources that can be generated by the CAN module. To service specific interrupt flags, a function named *IfxCan_Node_clearInterruptFlag* is used. After the execution of this function, the interrupt flag will be serviced and a new interrupt can be generated. So the flag should be served at the end of the interrupt function.

After receiving a new message interrupt the interrupt service routine should (or can) load the message from the HW buffer and move the acknowledgement index of a buffer (handled by iLLD). The reading from the HW buffer can be done by using the iLLD function *IfxCan_Can_readMessage*. The function should be provided with a node configuration (discussed in section 5.1.2, a pointer to the variable that stores CAN message header of type *IfxCan_Message* and a pointer to an array where data should be written.

An iLLD message header described in a data type *IfxCan_Message* contains a few parameters that specify from which buffer data should be read (FIFO0, FIFO1, Queue). In this example, FIFO0 is configured to be an RX buffer so the *readFromRxFifo0* parameter should be set to TRUE before reading the message.

After the message has been read from the HW buffer the interrupt handler processes the message and sends it as a reply message. In this case, a simple function that increments both CAN ID and data is used, but any more sophisticated function can be used instead.

To successfully send a CAN message using iLLD drivers it should be provided with a standard, for the CAN message header, parameters. Such as header type, CAN ID and data length. As well as the special parameters that are used inside of the CAN iLLD drivers.

Such parameters are the mode of the frame and the buffer where the message should be stored before being sent. Since the reply message always contains only one byte of data the frame mode is set to be *IfxCan_FrameMode_standard*. For messages that contain more data, iLLD can be configured to setting *IfxCan_FrameMode_fdLongAndFast*, for transmission with CAN FD and enable bitrate switching on RX and TX.

The last parameter before sending a message is a queue where the message should be stored. By default, if none of the buffers are specified, then the message will be stored in the dedicated buffer. But in this example, the CAN node is configured to have a queue instead of a dedicated buffer as a TX buffer. To store the message in a queue, parameter *storeInTxFifoQueue* should be set to TRUE. The next code snippet shows the simple example of the message being read from the HW buffer, modified and sent in a busy-wait loop.

```
IfxCan_Message rxMsgHdr;
IfxCan_Can_initMessage(&rxMsgHdr);
uint8 rxData[MAXIMUM_RX_CAN_FD_DATA_PAYLOAD];

/* Received message content should be read from RX FIFO 0 */
rxMsgHdr.readFromRxFifo0 = TRUE;

/* Read the received CAN message */
IfxCan_Can_readMessage(&canNode, &rxMsgHdr, (uint32*)rxData);

IfxCan_Message txMsgHdr;
IfxCan_Can_initMessage(&txMsgHdr);

/* Configure message header */
txMsgHdr.messageId        =  rxMsgHdr->messageId + 1;
txMsgHdr.messageIdLength  =  rxMsgHdr->messageIdLength;
txMsgHdr.frameMode        =  IfxCan_FrameMoLade_standard;
txMsgHdr.dataLengthCode   =  IfxCan_DataLengthCode_1;
txMsgHdr.storeInTxFifoQueue  = TRUE;

/* Send a message with busy-waiting */
do
{
    status = IfxCan_Can_sendMessage(&canNode, txMsgHdr,
                                    (uint32 *)rxData);
}
while(status == IfxCan_Status_notSentBusy);
```

At this point we have already discussed the configuration of Erika Enterprise and iLLD drivers for the single-core CAN example. The source code for this project is located in a folder *can_singlecore*.

5.2 Multi-Core Example

This example has been done as a demonstration that the hardware modules (in this case CAN module of MCU) can be accessed from multiple cores under RTOS. In this example, CAN messages are now received on CPU0 and sent using a task on CPU1. When messages are received at CPU0 they are stored in a ring buffer before being processed.

The example is based on the single-core example described in chapter 5.1. Let's quickly recapitulate what single-core application has:

- CAN communication using iLLD drivers
- Received message interrupt (RXF0N)
- Keep-alive message with a counter of received messages

- Debug mode

The application still follows the general flowchart shown in figure 5.1, however, some parts of the chart have changed and a more detailed description of the message ”path” is shown as a sequence diagram in figure 5.2.

In this example, the message after being read from the HW buffer, is stored in the SW buffer instead of processing it right away. Then periodically, a CPU1 task that service the SW buffer is triggered by ALARM, then the message is finally processed and a reply is sent.

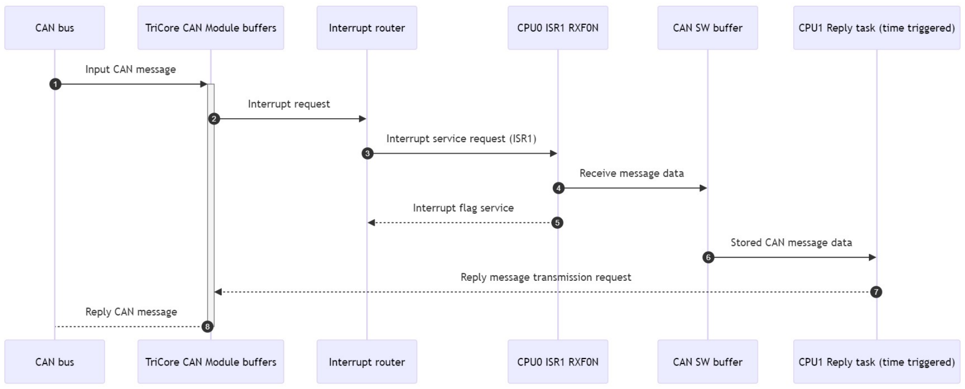


Figure 5.2: Multi-core application sequence diagram, detailed.

In comparison to the single-core example, this example utilizes more resources of the CAN module. The application uses bigger HW buffers, more interrupts and also HW filtering based on the ID of the CAN message. Also, the application now has an implementation of SW buffer, which is a circular buffer (or a ring buffer) guarded by a spinlock to allow multi-core access.

The following sections will describe how Erika Enterprise RTOS is configured and what changes to the source code of single-core example from section 5.1 are done.

The source code for this example is located in a folder *can_multicore/*.

5.2.1 Erika configuration

The core of the configuration is still the same as in the single-core example. But few changes have been made. The list of the changes can be described as follows:

- One more CPU has been added to the configuration.
- RXF0N interrupt category is changed to 1.
- New interrupt handlers definition has been added.
- A new task that services the SW CAN buffer is added.
- The Data length of the keep-alive message is increased.

Let us get through the changes step-by-step. The first change adds one more core to the conf.oil. This can be done by adding this code.

```
CPU_DATA = TRICORE
{
  ID = 0x1;
  IDLEHOOK = TRUE
  {
    HOOKNAME = "idle_hook_core1";
  };
};
```

It is very similar to the definition of the first core, but frequency and compiler are not specified. By default, they should be the same as for the CPU0.

The next change is new interrupt handlers that are servicing newly added interrupts. Resulting in having three interrupts in the application, including two new.

```
    ISR can_isr_fifo0_msg_lost {
         CPU_ID = 0x0;
         CATEGORY = 1;
         SOURCE = "CAN_CAN0_INT3";
         PRIORITY = 6;
    };

    ISR can_isr_tx_success {
         CPU_ID = 0x0;
         CATEGORY = 1;
         SOURCE = "CAN_CAN0_INT9";
         PRIORITY = 8;
    };
```

ISR named *can_isr_fifo0_msg_lost* is triggered each time when a message is lost for FIFO0. The message is considered lost when the CAN module does not have free resources to save the message. It can happen if the RX buffer is full.

Next is an interrupt triggered on the transmission success event. The transmission events were discussed in a section 2.4.1.

The sources of interrupts have been found according to a description in section 2.1.2

A new task has been added to service CAN messages stored in a SW buffer. The task checks whether the buffer has new messages and if it is so, then it picks these messages and processes them one-by-one. The task will continue to pick messages from a buffer until it is empty. The task is fired periodically by *ALARM* object (alarm is triggered each 10 ms). The size of both buffers (HW and SW) should be enough to handle all incoming messages within the period between task calls, see chapter 5.2.2. Since the *TASK* is running on CPU1 *ALARM* uses a definition *COUNTER* object for CPU1. However, it is possible to trigger a task with *ALARM* based on the *COUNTER* from another core.

```
TASK task_can_tx_msg_processing_cpu1
{
    CPU_ID = 0x1;
    SCHEDULE = FULL;
    ACTIVATION = 1;
    PRIORITY = 10;
};
```

5.2.2 CAN Driver – iLLD and SW buffer

The multi-core definition of the CAN configuration did not change much from the single-core example, two interrupts have been added and HW filters are configured.

The main change is that CPU1 can now access the module simultaneously with CPU0.

Also, *keep alive* message now hosts 2 more counters for new interrupts. Thus the new keep alive message contains all the needed information for debugging purposes. It sends data about how many messages have been received, transmitted and lost. Under normal conditions, the counter of lost messages should be zero.

CAN Node configuration

Since the core functionality is the same, the source code for this example is based on a single-core example. Thus the already mentioned in section 5.1 functions and their usage will not be described here.

Description of the changes to the new configuration of the CAN node can be described as follows. Two interrupts have been enabled in the configuration, and both of them are serviced on CPU0. Resulting in all CAN module interrupts being serviced by CPU0. CPU1 has only one interrupt assigned to it, which is COUNTER interrupt.

The configuration of the interrupts is very similar to the configuration of the interrupt Rx FIFO0 New Message. New interrupts have lower priority than RF0N interrupt. Because new interrupts serve only a debug role. All of the interrupts have category 1 (ISR1). Changes to CAN node configuration for multi-core example are shown in a list (interrupt name in brackets according to Appendix B):

- RX and TX buffer is set to 32
- 2 more interrupts are added so the full list is now:

- Rx FIFO0 New Message (RF0N) – triggered when new message arrived
- Transmission Completed (TC) – triggered when transmission has been completed
- Rx FIFO0 Message Lost (RF0L) – when the buffer has no space to add a new message.

Due to the blocking behaviour of the FIFO buffers, it can happen that interrupt flag RF0F (RX FIFO0 Full) will be raised. This interrupt is not handled by the application. Under the normal behaviour of the application, it should not happen since the execution time of the RF0N interrupt is less than the time that the CAN message needs to be fully sent and the size of the SW buffer can be configured to a much higher value than HW buffers. However, by default, it is just 32 messages.

Each interrupt has a specific counter value assigned to it. When some event is happening the counter is being incremented by one. The value of the counters is sent in *keep alive* message that in the multi-core example has 12 bytes corresponding to 3 values each 4 bytes long. When the counter reaches a limit of 32bit value it is reset to 0 by the application automatically.

Also in this example, the CAN node is configured to reject messages within the defined range of CAN ID. The filters are configured separately for extended CAN ID and standard CAN ID. The general flowchart of the HW filter looks as it is shown in figure 5.3.

The application is configured to reject all messages within the ID range from 0x2 to 0xAA (170 decimal). That means that after such a message is received the interrupt new message is not generated and the message is not added to the RX FIFO0 buffer. In the configuration structure of the node the section *filterConfig* is set to accept non-matching frames to RX FIFO0. It is important since interrupt RX FIFO1 New Message is not configured. The configuration of the filter is done using iLLD functions and both extended and standard IDs are configured the same way:

```
// Initialize the filter structure
IfxCan_Filter filter;

filter.number = 0;
filter.elementConfiguration =
            IfxCan_FilterElementConfiguration_rejectId;
filter.type = IfxCan_FilterType_range;
filter.id1 = from_id; // 0x2
filter.id2 = to_id;   // 0xAA

IfxCan_Can_setStandardFilter(&canNode, &filter);
IfxCan_Can_setExtendedFilter(&canNode, &filter);
```

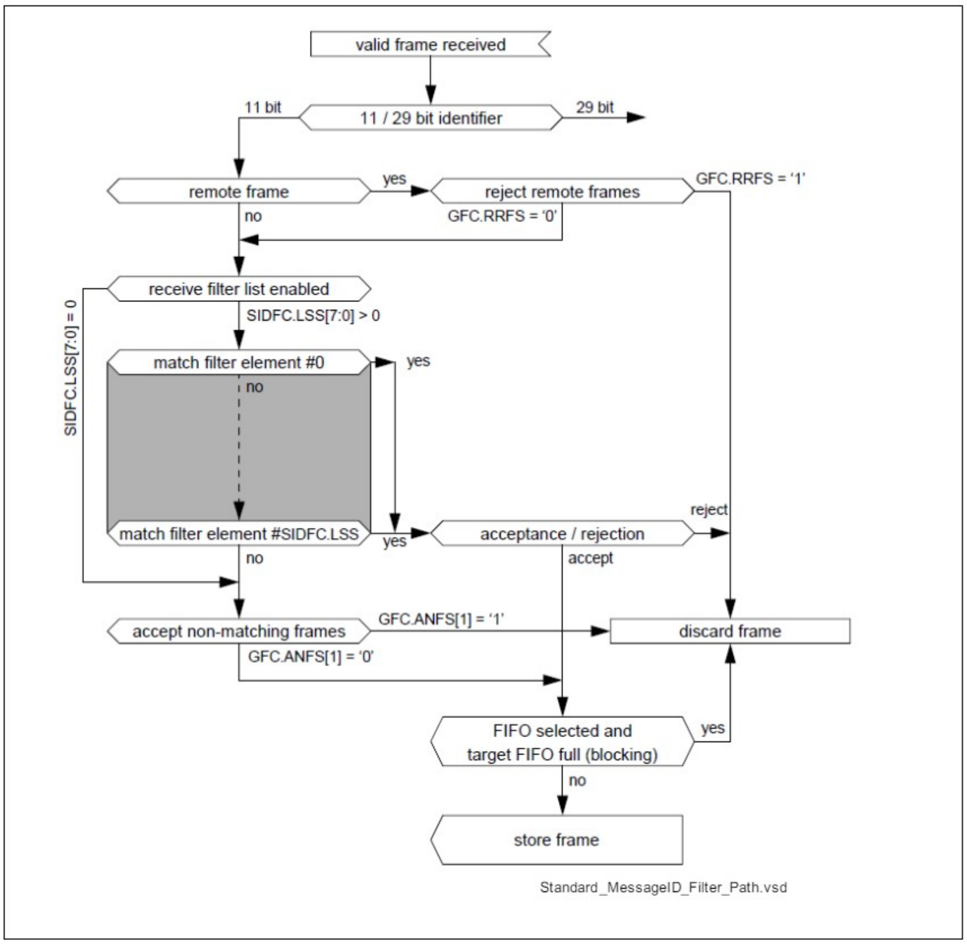


Figure 5.3: CAN hardware message filtering flowchart. Taken from TriCore User Manual Part 2 [7].

SW buffer implementation

The SW buffer is a commonly used technique when dealing with communication. The type of buffer used in this implementation is a so-called ring buffer. A ring buffer is such a buffer, where the end of the buffer is connected to the beginning of the buffer, as shown in Figure 5.4. Buffer has two indices first is where the data will be written (buffer_write_idx) and the second is from where data should be read (buffer_read_idx). When the write index is about to cross the length of the buffer it is reset to the initial value (buffer start). That allows messages or other data to be stored in some sort of queue where the first message to get in is the first message to get out (FIFO). It is the same structure as HW buffers of the CAN module use.

Since the application is multi-core and the buffer will be accessed at least from 2 places it should have a synchronization mechanism. It should be done so the reading and writing cannot be made from the different cores, or threads, at the same time. The synchronization mechanism is the simple

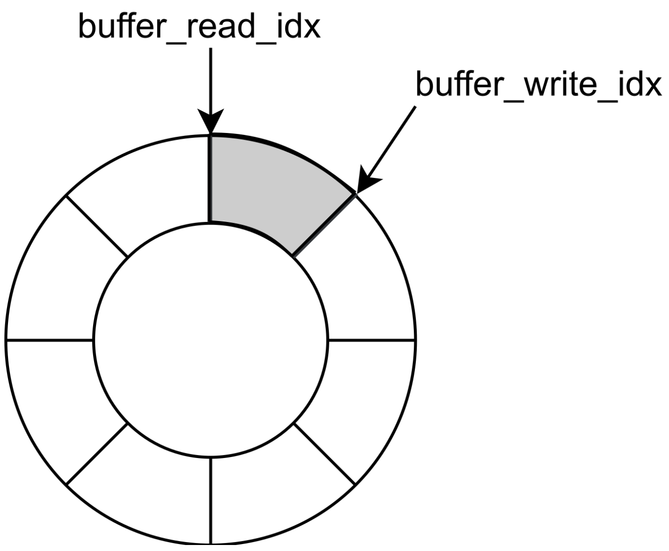


Figure 5.4: Ring buffer structure example.

implementation of a spinlock that iLLD already has.

Erika also has such a primitive, but Erika's implementation is far more complicated than iLLD's. Erika's implementation additionally performs multiple checks, when iLLD's implementation simply waits in a while loop for a specified amount of cycles until it gets a spinlock. If after a specified amount of cycles thread cannot access the resource (spinlock), a function should return a value that will indicate that. And the application should handle it.

To take a spinlock program calls function *IfxCpu_setSpinLock*, which has two parameters, first is the pointer to a spinlock, and second is the amount of cycles to wait. To reset spinlock function *IfxCpu_resetSpinLock* is used. During the time that thread has a spinlock, interrupts are disabled to allow faster execution of the critical part of the code. The code snippet that implements the process of acquiring spinlock and resetting the spinlock looks the following way:

```
 /** Lock spinlock and reenable interrupts
 */
static void _spinlock_lock(IfxCpu_spinLock *lock)
{
    IfxCpu_disableInterrupts();
    IfxCpu_setSpinLock(lock, MAX_32BIT_VAL);
}

/** Unlock spinlock and reenable interrupts
 */
static void _spinlock_unlock(IfxCpu_spinLock *lock)
```

```
{
    IfxCpu_resetSpinLock(lock);
    IfxCpu_enableInterrupts();
}
```

It is also important to mention that according to the TriCore architecture manual [16] instruction that is used for access the spinlock *cmpswap* requires the address to be aligned to 4 bytes, which means that the address should end either on 0x0 or 0x4. To ensure that the Tasking compiler will place the variable in proper memory space, a special macro ***IFX_ALLIGN(4)*** should be used in front of the data type of the variable. This explicitly says to the compiler that the variable should be aligned to 4 bytes. If it is not done, the processor will fail at the execution of this instruction. This is very hard to debug since the behaviour seems random.

Continue on SW buffer implementation, spinlock should be acquired by the shortest possible amount of time. Because while the thread has a spinlock interrupts are disabled, To achieve it, a spinlock is only locked during the moment when the buffer is accessing the read or write indices. An example of such a usage is a function that loads a message from a buffer to some external variable of type can_message.

```
boolean can_buffer_pick_message(can_message* message)
{
    boolean buffer_status = FALSE;

    /* Blocking spinlock */
    _spinlock_lock(&can_sw_buffer_index_lock);
    if (can_buffer_read_idx != can_buffer_write_idx)
    {
        *message = can_sw_rx_buffer[can_buffer_read_idx];
        buffer_status = TRUE;
    }
    _spinlock_unlock(&can_sw_buffer_index_lock);

    return buffer_status;
}
```

As it is shown in a code snippet and has already been described above. The spinlock is blocked only for the shortest possible amount of time and released immediately after the message is copied into the external variable. The function *can_buffer_write_message* is implemented according to the same principle.

The only change that is left to discuss is the task that service the SW buffer. The flowchart of the task is shown in figure 5.5. But in short, the task will be active until the TX buffer is not full and the SW buffer has new messages to process.

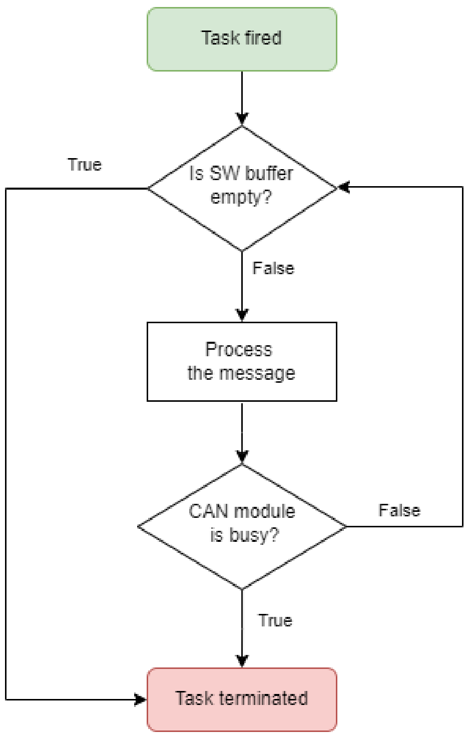


Figure 5.5: Flowchart of the task that process CAN messages in multi-core CAN example.

To implement this behaviour reading from the SW buffer has been separated into two steps. The message should be picked from a buffer (implemented in *can_buffer_pick_message*) and only if the message has been successfully sent, the read index of the buffer is moved (function *can_buffer_move_index*).

An important note is that TriCore architecture allows any core to access any address space in the memory. So the entire memory can be considered as shared.

5.3 Benchmark results

The subject of a benchmark is the latency of communication that has been described in chapter 2.2.1. Latency was measured using a program *canping* and an oscilloscope to verify the results obtained by *canping*. Figure 5.6 shows the definition of latency used in this tests.

It is important to mention that the latency itself is not a goal of a test. MCU with higher computational power, or even more simple code will have lower latency. The goal is to verify that the implementation of the example applications is robust and fast enough to handle a flood of CAN messages without their loss and changes that were done to Erika Enterprise are not causing the program to fail during runtime.

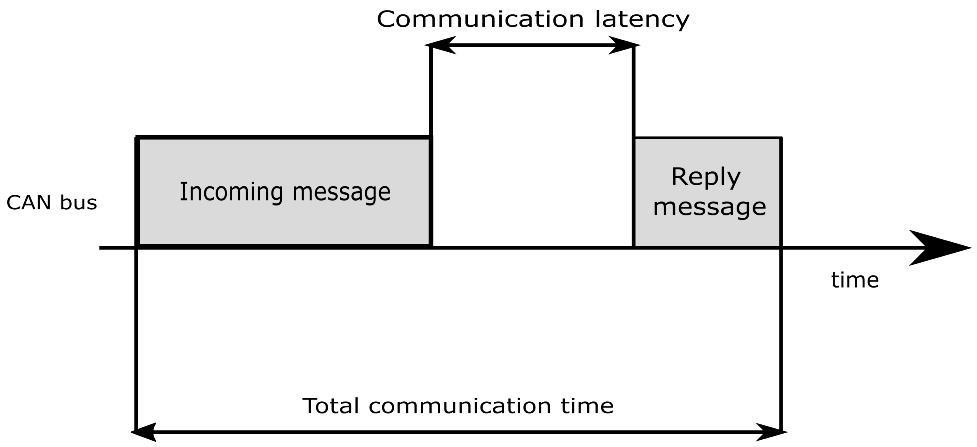


Figure 5.6: Definition of latency used for tests. Source [10].

Let us take a look at the test that was done using an oscilloscope. Figure 5.7 is an example of how the measurement looks like. The blue line is the CAN_H signal, and the red line is the state of the LED mounted on the hardware's top PCB board. The LED is switched on when the program enters ISR and switched off when the program leaves it.

The total communication time (between the start and end of communication) is around 175 μs. The time between vertical green lines is the latency of the communication, which is around 12 μs. This value also includes the "end of frame" (see section 2.2). After the message is sent the bus is switched to the recessive state (0 for CAN_H) for an amount of time equal to 5 bytes. With the communication speed set to 1 Mbit/s, this time is equal to 5 μs and can be subtracted from the latency.

The program spent just 3.5 μs in ISR. The same figure shows that the incoming message is longer than a reply message from the MCU. This is correct because the payload for incoming messages is set to be 8 bytes and the reply message always has only 1 byte of data. Also, as we can see in the same figure, the amount of time spent servicing interrupt and scheduling it (ISR2 is used) is much lower than the amount of time needed to send a CAN message. This means that the buffer will always be emptied before another message is received.

The next test is done using *canping*. The command that used to start canping is: "**sudo ./__compiled/bin/vca__canping -m 1 -d can0 -w 2 -vv -c 40000 -r**".

During the tests, different sizes of HW and SW buffers were tested. The number of messages sent is always a 40000. The payload is 8 bytes and messages are sent with extended identifiers (29 bits long). Program should sent another message as soon as the reply message is received (flag -w 0 in command).

The test result for the single-core example is shown in a table 5.1. Also note that the N in the table's column "Size of HW buffer" means, how many

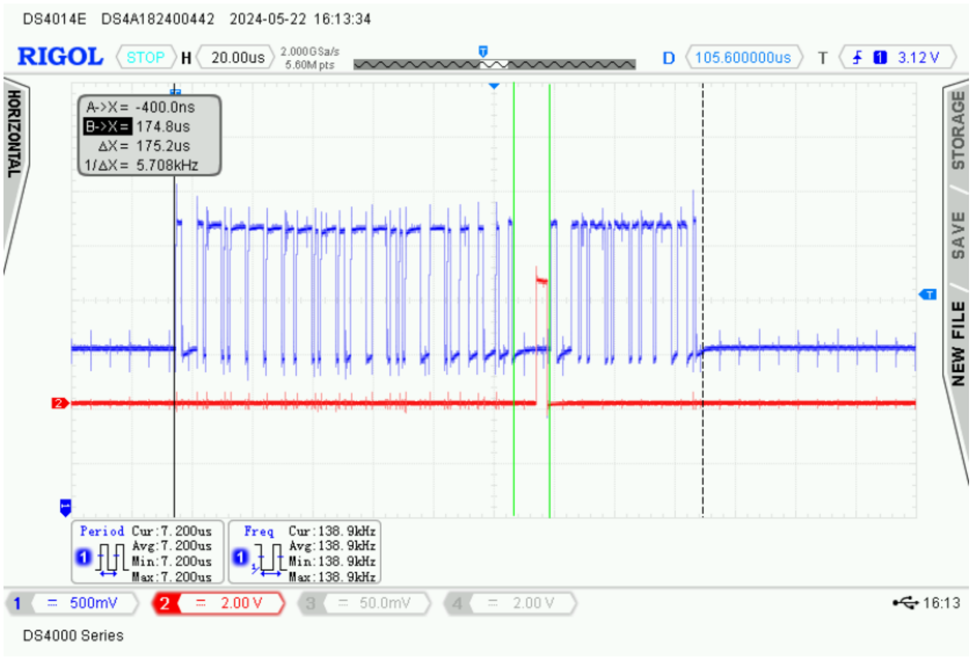


Figure 5.7: Oscilloscope measurement of CAN communication latency. The time between two green vertical beams is the latency of communication. The time between black vertical beams is the total communication time. The grid is 20 μs.

Size of HW buffer [N]	Mean latency [μs]	Standard deviation [μs]	loss [%]
1	379.84	35.57	0
4	379.61	34.59	0
10	381.87	39.10	0
32	373.39	28.09	0

Table 5.1: Single-core Latency test result

messages it can fit.

The latency values in table 5.1 are significantly higher than in a test done with an oscilloscope. This is because the UBS-CAN converter latency is much higher than it was expected. Also *canping* measure total communication time (definition in figure 5.6).

However, the results of the tests are still useful as they can show that the program CAN handle the flood of messages without their loss and that the program will not fail. Thus the implementation is robust enough for demonstration purposes. The slight difference in the latency value that seems to be related to HW buffer sizes, can be mostly explained as the variations in USB-CAN converter latency rather than the real difference made by the size of the HW buffer.

The following figure 5.8 shows the latency histogram of the measured data

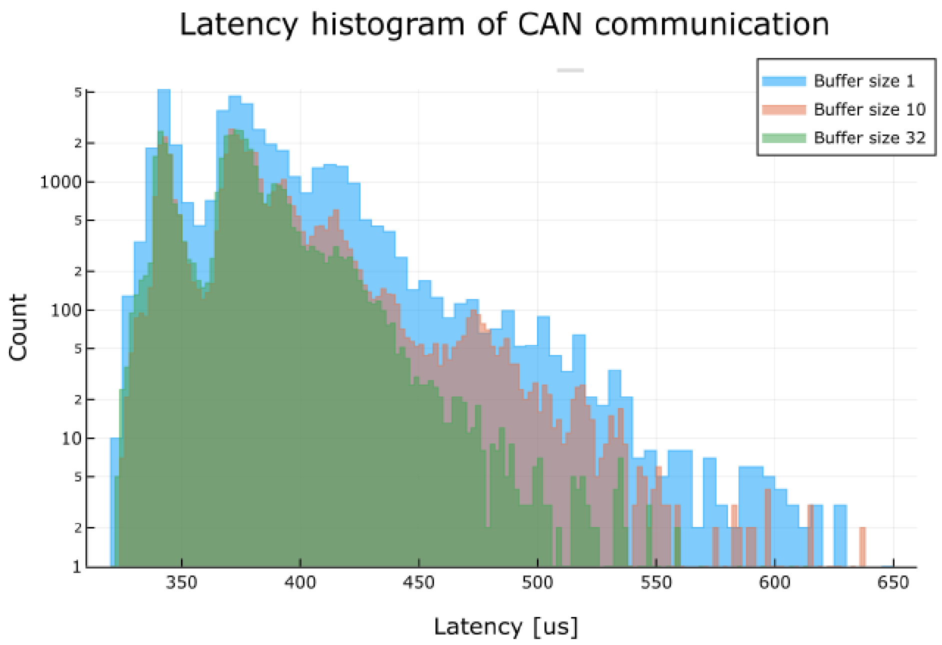


Figure 5.8: Histogram of latency for single-core application.

Task period [ms]	Size of SW buffer [N]	Size of HW buffer [N]	Mean latency [μs]	Standard deviation [μs]	loss [%]
1	512	32	949.61	142.56	0
5	512	32	4921.72	349.33	0
10	512	32	9921.05	345.76	0
1	512	1	950.33	142.27	0
1	32	32	948.24	147.29	0

Table 5.2: Multi-core Latency test result

for the single-core example with different sizes of the HW buffer.

One important fact that has been already mentioned is that enabling debug mode causes latency to grow significantly. The latency, in this case, is around 34690 μs, Which is more than 100 times bigger latency than with debug mode turned off and more than a thousand times more than a latency measured with an oscilloscope. This is happening due to the blocking behaviour of command *printf* that is used to write debug data to the console. The latency histogram for this scenario can be found in Appendix C.1

Now let's take a look at a multi-core example. This example can be used as a base for CAN communication in more sophisticated demo projects because this example has SW buffer implementation that allows to postpone the new message processing.

As we can see in the table 5.2. The application is still able to handle all the messages on a CAN bus without a loss. Even if the hardware buffer size is set to 1. However, it does not make sense to set a SW buffer to 1. The application will end up leaving messages in the HW buffer since the SW buffer is full after the first message is received, which can lead to the HW buffer being full and new messages being rejected. Also, the message processing task is done in such a way that it cannot read messages from the HW buffer. So after replying to one message contained in the SW buffer, the task will be terminated.

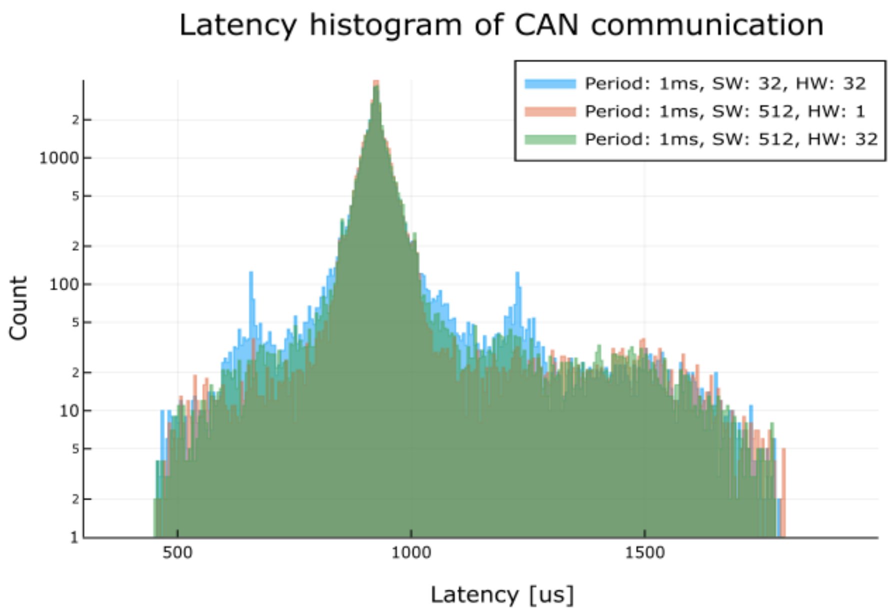


Figure 5.9: Histogram of latency for multi-core application with a period of task set to be 1 ms.

In the same table, we can see that the period of the task significantly affects the latency, which is expected. The controller will send a response only after the task is activated, and the larger the period of the task the higher is maximum latency. As was expected for a 1 ms period of the task the mean latency is around 1 ms, the same happens for 5 ms and 10 ms periods. The slight difference between 10 ms or any other period and the mean value of the same measurement can be explained by the fact that messages are sent by canping one after another and some of them arrive when the task is running, so they will be processed almost instantly. Some of them arrive when the task was just terminated, so they should wait until a new iteration begins. For example in the test where the task period was set to 10 ms the lowest latency is 4834 μs. Also, some messages will not be sent in the next iteration after they arrive if the size of the TX buffer is smaller than the number of received messages.

Additionally, the multi-core example has been tested with the reply task running in an infinite loop without delay. In this case, the latency of the multi-core example is almost the same as for the single-core.

The histogram in figure 5.9, shows that most of the messages are sent back (replied) at a time around 1 ms for the reply task period set to 1 ms.

Chapter 6

Serial Peripheral Interface (SPI)

This chapter explains the basics of using an implementation of FAT file system FatFs with SPI on TriCore MCU running Erika Enterprise.

Unfortunately, the HW provided for this diploma thesis has been developed for internal testing purposes, and it does not have the support of the QSPI interface yet. The pins are not connected to the SD card slot. But since SPI is a very simple protocol, *Garrett Motion Inc.* provided a solution for the problem which is a ”bit-bang” implementation of SPI.

6.1 SPI Drivers – iLLD & bit-bang

At the beginning of this chapter, we will discuss and explain a bit-bang implementation of SPI. After some basic code that can be used for communication with an SD card using an inbuilt TriCore module and iLLD will be described.

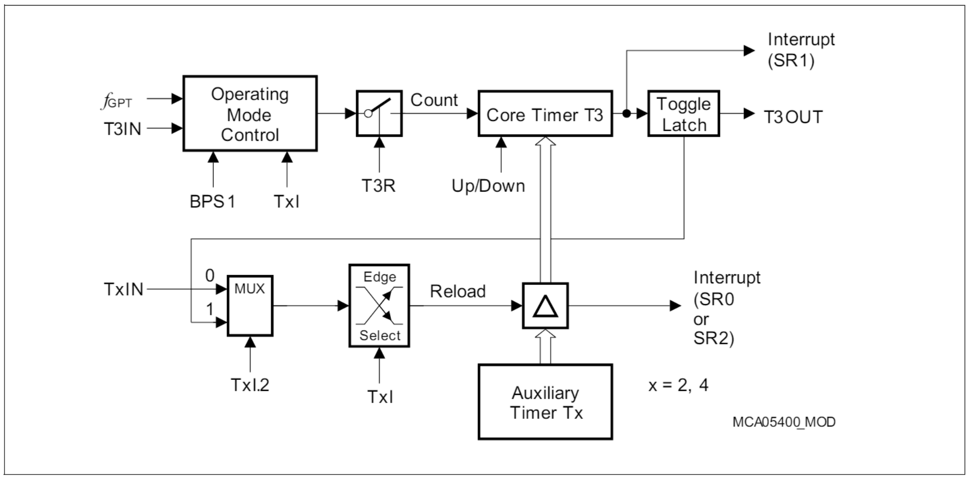


Figure 6.1: GPT reload mode general structure [7].

The SW implementation of SPI requires a controller to have some source of clock signal. In this case, a General Purpose Timer (GPT) is used to generate a clock signal and periodically check the state of the bus. Tricore has 6 GPT timers, however, not all of them have the same functionality. According to manual [7]. Only timers T2 and T4 can be used in the reload mod for timer T3.

As it has been described in section 2.3.1. SPI is a clock synchronous bus, so that means that the bus is sampled only on the clock edge. That significantly simplifies the implementation since the entire communication protocol can be done in one interrupt of timer.

First of all GPT module should be configured to generate interrupts with the desired communication speed. GPT T3 interrupt is generated every 500 ns, which means that the frequency of the CLK signal for SPI is 1 MHz. To generate one cycle of CLK signal two interrupts of GPT T3 are needed. One will set CLK to a HIGH state and the other will reset it to a LOW state. Configuration of the GPT timer can look like this:

```
/* Initialize the GPT12 module */
IfxGpt12_enableModule(&MODULE_GPT120);
IfxGpt12_setGpt1BlockPrescaler(&MODULE_GPT120,
            IfxGpt12_Gpt1BlockPrescaler_8);

/* Initialize the Timer T3 */
IfxGpt12_T3_setMode(&MODULE_GPT120, IfxGpt12_Mode_timer);
IfxGpt12_T3_setTimerDirection(&MODULE_GPT120,
                IfxGpt12_TimerDirection_down);
IfxGpt12_T3_setTimerPrescaler(&MODULE_GPT120
            IfxGpt12_TimerInputPrescaler_1);
IfxGpt12_T3_setTimerValue(&MODULE_GPT120, RELOAD_VALUE);

/* Initialize the Timer T2 */
IfxGpt12_T2_setMode(&MODULE_GPT120, IfxGpt12_Mode_reload);
IfxGpt12_T2_setReloadInputMode(&MODULE_GPT120,
                IfxGpt12_ReloadInputMode_bothEdgesTxOTL);
IfxGpt12_T2_setTimerValue(&MODULE_GPT120, RELOAD_VALUE);

/* Initialize the interrupt */
volatile Ifx_SRC_SRCR *src =
                            IfxGpt12_T3_getSrc(&MODULE_GPT120);
IfxSrc_init(src, ISR_PROVIDER_GPT12_TIMER,
                        ISR_PRIORITY_GPT12_TIMER);
IfxSrc_enable(src);
```

At the beginning of this code, the GPT module is initialized with prescaler 8 (resolution 250ns for each tick of the timer). After GPT timer T3 is configured to be in timer mode the direction of counting is set to *IfxGpt12_TimerDirection_down*. GPT T2 is used as a holder of reload value for timer T3. Each time T3 reaches zero its value is automatically reset to the value stored in timer T2. Timer value for both T2 and T3 defined in macro RELOAD_VALUE that extends to value 2. With each tick of a timer being generated every 250 ns the resulting period of a timer is 500 ns. After T3 is fully configured, T2 is configured to reload mode and reload value loaded into it.

The next step is the initialization of interrupts. The initialization of interrupts for the GPT module is different from the CAN module. Firstly the pointer to the value that describes GPT T3 is stored, and then the value of the registers SRPN (Service Request Priority Number) and TOS (Type of Service Control) is updated with a priority of interrupt and type of service (in this case CPU1) is stored into the registers of GPT T3.

Now let's move to the implementation of an interrupt service routine that will do all the jobs related to SPI communication. The pseudo-code for this routine:

```
if edge is rising then:
{
    bit_count++;
    set_clk();
    read_data_from_miso();
}

if edge is falling then
{
    if bit_count >= 8 then
    {
        reset_clk();
        byte_count++;
        bit_count = 0;

        if byte_count >= data_byte_length then
        {
            stop_t3();
        }
    }
}
else
{
    set_MOSI((tx_buff[byte_count] >> (7-bit_cnt)) & 0x01);
}
```

Let us get through this code. The code should keep track of two values. First is the number of bytes (byte_count), this is important since this value is used as an index for transmission data buffer (tx_buffer). The second index is the number of a bit in this byte (bit_count) that is now being transmitted or received. Since the value is sent bit-by-bit we should keep track of it.

When the program gets its first interrupt, cock edge is set to be falling, which means that the SPI communication is defined as CPHA = 0, or in other words sampled on the first edge of the CLK signal.

In the very first GPT interrupt we set the value of the MOSI signal to the

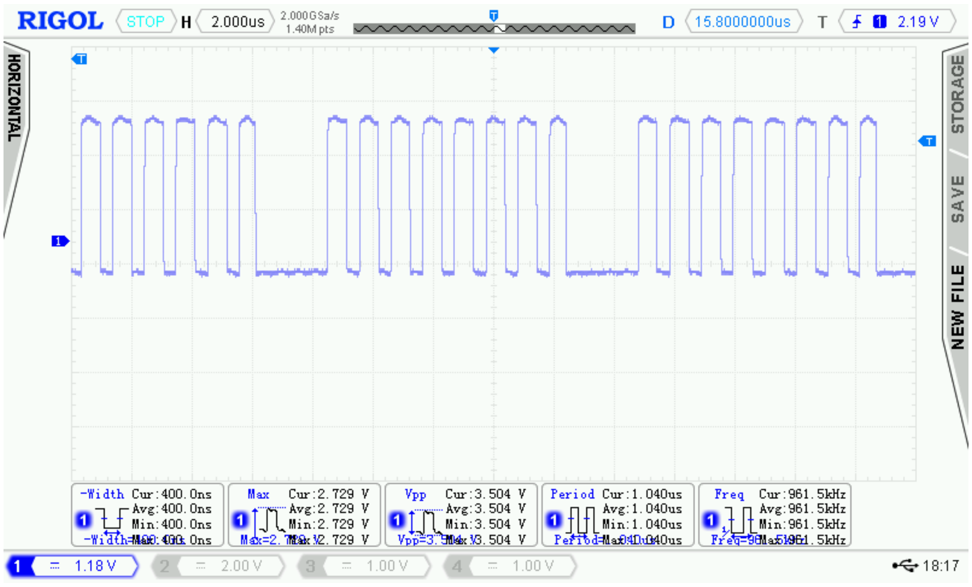


Figure 6.2: SPI CLK signal generated by GPT T3 measured on an oscilloscope.

value that is contained in the first bit of the first byte in the transmission buffer. Then when the next interrupt is received, the program sets the CLK signal to 1 and reads the value of signal MISO. In the next iteration bit index is moved and the next value is set as a value of MOSI. This continues until the end of the byte is reached. Then next byte from the transmission buffer is sent to the bus following the same procedure.

This continues until the end of the transmission buffer is reached. The same applies to the receiving procedure. The value of the MISO signal is captured on each Rising edge. The exact implementation can be found in the file *can_logger/ee/SPI_CPU.c*.

6.2 FAT Library

The information about FatFs usage is gathered from the Wiki of the file system [14]. The following section explains the code that is used for communication with the SD card. The code is initializing the SD card, opens a file and writes data to it. Then the file is closed and the volume is unmounted.

FatFS requires users to define basic communication interfaces. Each project has a different peripheral interface configuration and it is not the job of the file system to take control of it. Thus the interface for the storage system should be defined by the user. The file system has 3 main interfaces send a command, receive data and transmit data. In this diploma thesis SD card is used as the storage.

Communication with an SD card starts with its initialization. This is done by using the function *disk_initialize* from FatFs. To perform this operation clock signal should be set to slow mode (100 KHz). The program waits for 80 dummy clocks (data lanes are set to idle). After 80 clocks program sends a command CMD0 to the SD card which will change the state of the SD card to idle. After this, the program sends a command *SEND_IF_COND* to identify if the SD card type is SDv2. Then voltage range checks are performed. If this check passed then the SD card is considered to be initialized.

After the disc is initialized the volume of the disk should be mounted (e.g. selected). Since the disc has only one volume there are not many options to choose from. The mounting process is done by the command *f_mount*. Be aware the results of every FatFs operation should be checked to verify that the operation result is correct.

When volume is mounted, a file can be created or opened with commands *f_open*. The function should be provided with the name of the file to open (or create) and a list of flags. In the logger example following flags are used "FA_OPEN_APPEND | FA_WRITE | FA_READ". This set of flags indicates to the FatFs that the file should be opened if it exists and the new information should be appended to the end of the file. If a file does not exist then it should be created.

To perform a write operation to the file, the function *f_printf* is used. It is a handy way of creating formatted strings. The syntax is the same as for the C language function *printf*, but additionally, a pointer to the file is passed to the function. To use this function in the configuration file of the system option "FF_USE_STRFUNC" should be enabled.

When the write operation to a file is completed The file should be closed with *f_close* and unmounted if no other operations will be performed with this volume. To do this, function *f_mount* with the last argument equal 0 is used.

6.3 Example Application

The example application is an extension of the multi-core CAN example with SPI communication. Briefe recapitulation of multi-core CAN example properties:

- CAN communication using iLLD drivers
- HW message filtering
- 3 ISR1 interrupts from the CAN module (New message received, message lost, transmission completed)
- Keep-alive message with counters of received, transmitted and lost messages.
- CAN SW buffer
- Messages received on CPU0 and transmitted from CPU1

The application has been extended with one more SW buffer, this time for storing logs of transmitted or received messages. Implementation of the SW buffer for logs is the same, as for the SW buffer used in the CAN example. Functions have been renamed and a new spinlock is added.

The Erika Enterprise configuration has been extended with a handler for a GPT T3 interrupt (type ISR1) and a new task that handles the log writing process. A new task is done as an infinite loop that continuously checks the state of the log buffer. This task can be preempted by the CAN message reply task, so the controller is still able to respond to all messages with adequate latency.

The process of writing something over SPI communication to an external device is much longer than handling memory inside of microcontroller. The performance is also downgraded by the fact that SW implementation of interrupt is used instead of the QSPI module. That means that during write or read operations, the microcontroller will receive an interrupt every 500 ns. With the CPU running at 300 MHz (each cycle 33.33 ns), the controller will be almost constantly busy with handling the SPI communication. The task that services a log buffer is a copy of the task that services a CAN SW buffer in a multi-core example. The example of output logs stored on the SD card looks the following way:

```
CAN: DIR: 2 ID 518 DATA:  0x01 0x00 0x00 0x00 0x00 0x00
CAN: DIR: 2 ID 514 DATA:  0x00 0x00 0x00 0x00 0x00 0x00
CAN: DIR: 2 ID 518 DATA:  0x01 0x00 0x00 0x00 0x00 0x00
CAN: DIR: 2 ID 514 DATA:  0x00 0x00 0x00 0x00 0x00 0x00
CAN: DIR: 2 ID 518 DATA:  0x01 0x00 0x00 0x00 0x00 0x00
CAN: DIR: 2 ID 514 DATA:  0x00 0x00 0x00 0x00 0x00 0x00
```

Data are written to an SD card in a formatted way, very similar to the debug mode output of can single-core example. The only difference is the new column ”DIR” that specifies the direction of the communication. It is set by the parameter *log_type* in *log_item* structure. The value is an enum where 0 – EMPTY,1 –RX_EVENT, 2 – TX_EVENT and 3 – INFO_EVENT is reserved for additional usage (for example system activities like stack usage).

The source code for this example is located in a folder *can_logger*.

6.4 CPU frequency

This section describes how the frequency of the CPU is derived using registers and the schematics for the microcontroller. This information is then used in the section 6.5. The ”manual” computation of CPU frequency is needed because Erika does not set the frequency of the CPU for the TC387QP, even if the frequency is specified in the configuration file.

According to the general scheme Figure 6.3. The building blocks of the clocking system mentioned in user manual [6] are:

- Basic clock generation (Clock Source)
- Clock speed up-scaling (PLLs)
- Clock distribution (CCU)
- Individual clock configuration (Peripherals).

To determine what is the frequency of the CPU, the external crystal oscillator frequency should be known. According to the schematic of the MCU, the frequency of the oscillator is 20 MHz. This value is within allowed for the MCU range, from 16 MHz to 40 MHz. And since the external crystal is used, frequency is labelled as f_{osc0} in user manual [6]. Configuration of the oscillator in the Oscillator Circuit Control Register (OSCCON) is not discussed in this diploma thesis. There is no need to change the values in this register.

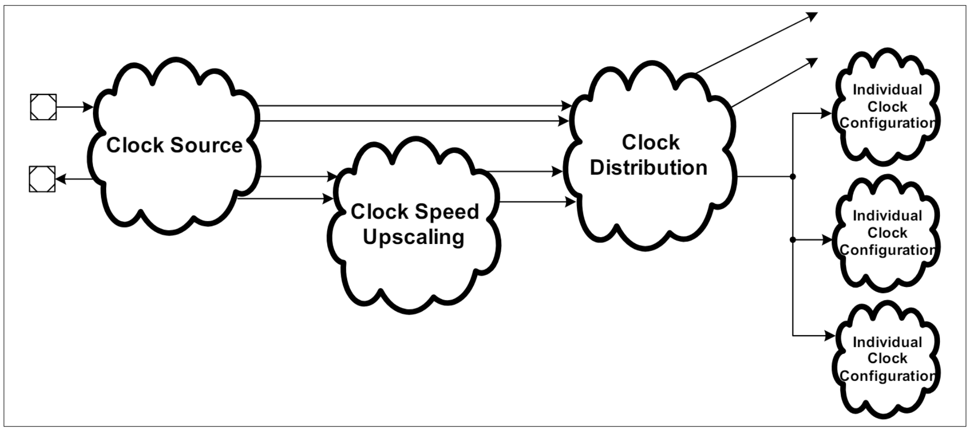

Clock Source
Clock Speed Upscaling
Clock Distribution
Individual Clock Configuration
Individual Clock Configuration
Individual Clock Configuration

Figure 6.3: TC3xx family general structure of the clocking system. Source [6].

6.5 Benchmark Results

Benchmark tests were done to find the data write bandwidth.

Firstly latency of the application has been measured. The length of data written to the SD cards has been set to 1 byte only, and then Performance Counters (more in architecture manual [15]) of the CPU were used to log the number of CPU cycles between the start of the writing function and its end. Since we know from section 3.3.2 that the CPU frequency is 300 MHz we can find the time elapsed between two events. Knowing the SW overhead in communication we can find data write bandwidth. Multiple tests with different data payloads were done.

Now with the results from the table 6.1 the data write bandwidth without SW overhead can be calculated. The overhead in this test is the additional

Data payload [byte]	Number of samples [N]	Mean [ms]	Standard deviation [ms]
1	1000	40.75	4.33
66	1000	45.03	4.80
128	1000	47.04	5.58
256 (100 times)	100	38.08	9.06

Table 6.1: SPI communication test results

time that is needed for SW to initialize the SD card, mount the file system and open the file. Additionally, the speed of the SD card plays an important role in this test.

The variable T is the total communication time with the SD card measured using Performance Counters. Calculated as the difference between the 1-byte test and the 128-byte test, then $T = 0.0063s$. Assuming that the 1-byte test is a pure SW overhead of communication. The value T then represents the time needed to transfer the data between the microcontroller and the SD card without the SW overhead. The number of bytes transferred for the time T is $N = 127$ since 1 byte of 128-byte payload is already included in T. Now let's find the write bandwidth for 128-byte payload B_{128}.

$$B_{128} = \frac{N}{T} = \frac{127}{0.0063} = 19725\ Byte/s = 19.73\ kB/s \tag{6.1}$$

So even though the CLK signal for SPI is set to be 1 MHz or 125 kB/s, due to the big overhead in communication, the real write bandwidth is only around 20.16 kB/s for a 128-byte payload. The SW overhead is around 41 milliseconds according to results in table 6.1. However the bigger is the number of bytes to be written in one access to the file, the higher is the write bandwidth. One file access means that the SD card was initialized only once and the file was opened only once during the communication.

For example, writing a sequence of 256-byte long data repeated 100 times within one file access (25 kB) takes only 300 ms. After subtracting the overhead of the communication time with the SD card ($T = 0.381s$). Then write bandwidth B_{256} for 256 byte payload is 73.53 kB/s, calculated using the equation 6.1.

$$B_{256} = \frac{25600}{0.340 * 1024} = 73.53\ kB/s \tag{6.2}$$

The value obtained from 6.2 is much closer to the maximum theoretical value than the value calculated for a 128-byte payload.

Table 6.1 also has the time needed to write one CAN log. The size of a log for an 8-byte CAN message with a 3-digit ID (for example ID 555) is 66 bytes including the newline character. The in this case is $T = 0.0045s$ or 4.5 ms. If CAN messages are sent with a lower period than 4.5 ms (plus a time to process them), messages will start to accumulate in the LOG buffer which will eventually lead to the buffer being locked for writing and the log will be lost.

The results are the following. During one access to the file, the application should write the most possible amount of data, due to the high overhead equal to 41 ms. However, the size of the data written should be reasonably high so it does not take too much space in the memory of the microcontroller.

Another limitation is the SW-defined SPI or ”bit-bang” SPI. When the communication will be switched to the QSPI module of the MCU the communication speed can be raised to much higher values, which will also positively affect data write bandwidth.

However, even facing the above-mentioned limitations the application works correctly as it is intended and the data are transmitted from the microcontroller to the SD card and the file can be opened on the PC afterwards.

Chapter 7
Conclusion

The main goal of this thesis was to add the support of TriCore TC387QP and iLLD drivers to Erika Enterprise and to develop examples that can demonstrate that the Erika Enterprise now works with iLLD and TC387QP.

Source files of Erika Enterprise were changed to provide support for TriCore TC38X MCU. Also, "MAKE" files of Erika Enterprise were changed to include a compilation of iLLD drivers.

To test CAN communication, with iLLD and Erika Enterprise, two examples are done. One example demonstrates the basics with a single core. The second, more complex, multi-core example demonstrates some of the multicore capabilities of TriCore MCU and Erika Enterprise such as accessing the same module from multiple cores. This example can be used as a base for other demo applications.

The implementation of the CAN examples has been tested. The latency for CAN communication for a single-core example is around 12 μs. The execution time of the interrupt service that receives an incoming message and ends a reply message is 3.5 μs.

Unfortunately, the QSPI module of the MCU has not been tested due to HW limitations of the development board. However, an example that implements SW-defined SPI using the GPT module has been provided by *Garrett Motion Inc.*. And on the base of this example, a simple CAN logger has been done.

As a test of the implementation, the data write bandwidth has been measured. The bigger the data payload to be transferred during one access to a file on the SD card, the higher is the data write bandwidth. SPI communication speed is set to 125 kB/s and the maximum data write bandwidth reached with a 256-byte payload is 73.53 KB/s.

All examples that were developed in this diploma thesis can be found in GitHub[1]. Changes that were made to Erika Enterprise can be found in another GitHub repository[2]

[1] `https://github.com/Darth-Bujar/examples_erika`
[2] `https://github.com/Darth-Bujar/erika3-tc38x/`

Bibliography

[1] AUTOSAR. Explanation of Interrupt Handling within AUTOSAR. Autosar. Available at: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_EXP_InterruptHandlingExplanation.pdf.

[2] ERIKA Enterprise Minimal API Manual (December 11, 2012). Available at: https://download.tuxfamily.org/erika/webdownload/manuals_pdf/ee_minimal_refman_1_1_3.pdf.

[3] OSEK (July 1, 2004) System Generation OIL: OSEK Implementation Language Version 2.5. Available at: https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/oil25.pdf.

[4] ERIKA3 OIL specification. Available at: https://www.erika-enterprise.com/wiki/.

[5] OSEK/VDX (2005): OSEK/VDX Operating System Specification 2.2.3. Available at https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf.

[6] Infineon AURIX™ TC3xx User Manual, Part 1. Available at: https://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_Part1-UserManual-v02_00-EN.pdf?fileId=5546d462712ef9b701717d3605221d96.

[7] Infineon AURIX™ TC3xx User Manual, Part 2. Available at: https://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_Part2-UserManual-v02_00-EN.pdf?fileId=5546d462712ef9b701717d35f8541d94.

[8] Robert Bosch, GmbH. (1991). CAN 2.0 specification. Available at: http://esd.cs.ucr.edu/webres/can20.pdf.

[9] Lennartsson, K. Comparing CAN FD with classical can. Kvaser. Available at: https://www.kvaser.com/wp-content/uploads/2016/10/comparing-can-fd-with-classical-can.pdf.

[10] M. Sojka, P. Píša and Z. Hanzálek, ”Performance Evaluation of Linux CAN-related system calls,” 2014 10th IEEE Workshop on Factory

Communication Systems (WFCS 2014), Toulouse, France, 2014, pp. 1-8, doi: 10.1109/WFCS.2014.6837608. keywords: Logic gates;Sockets;Linux;Kernel;Protocols;Hardware;Performance evaluation. Available at: https://ieeexplore.ieee.org/abstract/document/6837608.

[11] Piyu Dhaker. (2018, September). Introduction to SPI interface. Introduction to SPI Interface | Analog Devices. Available at: https://www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf.

[12] Wikipedia, The Free Encyclopedia. 12 May 2024, SD card. Available at: https://en.wikipedia.org/w/index.php?title=SD_card&oldid=1223534233

[13] File allocation table (2024) Wikipedia. Available at: https://en.wikipedia.org/wiki/File_Allocation_Table(Accessed:13May2024).

[14] ChaN. FatFs – Generic FAT Filesystem Module. Available at: http://elm-chan.org/fsw/ff/

[15] Infineon Technologies AG. TriCore™ TC1.6.2 core architecture manual Volume 1. Available at: https://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_Architecture_vol1-UserManual-v01_00-EN.pdf?fileId=5546d46276fb756a01771bc4c2e33bdd.

[16] Infineon Technologies AG. TriCore™ TC1.6.2 core architecture manual Volume 2. Available at: https://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_Architecture_vol2-UserManual-v01_00-EN.pdf?fileId=5546d46276fb756a01771bc4a6d73b70.

[17] Michal Sojka (October 18 2022). Project report, Erika Enterprise 3 for AURIX Studio and TC387.

Appendix A
List of Abbreviation

Symbol	Meaning
RTOS	Real-time operating system
QP	Quadratic Programming
CAN	Controller area network
MCU	Microcontroller unit
SPI	Serial Peripheral Interface
QSPI	Queued SPI
FAT	File Allocation Table
iLLD	Infineon Low Level Drivers
AUTOSAR	Automotive Open System Architecture
ASIL	Automotive Safety Integrity Level
CRC	Cyclic redundancy check or Cyclic redundancy Code
SD	Secure Digital
Erika or EE	Erika Enterprise RTOS
ADS	Aurix Development Studio
ROM	Read-only memory
RAM	Random access memory
OSEK	German:Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen. English: Open Systems and their Interfaces for the Electronics in Motor Vehicles
VDX	Vehicle Distributed eXecutive
OIL	OSEK Implementation Language
ISR1	Category 1 interrupts according to AUTOSAR[1]
ISR2	Category 2 interrupts according to AUTOSAR[1]
MCMCAN	New regeneration of MultiCAN+ module used in TriCore TC3xx family
ICU	Interrupt Compression Unit
SCU	System Control Unit
CCU	Clock Control Unit
GPT	General Purpose Timers

Appendix B

Interrupt mapping into groups

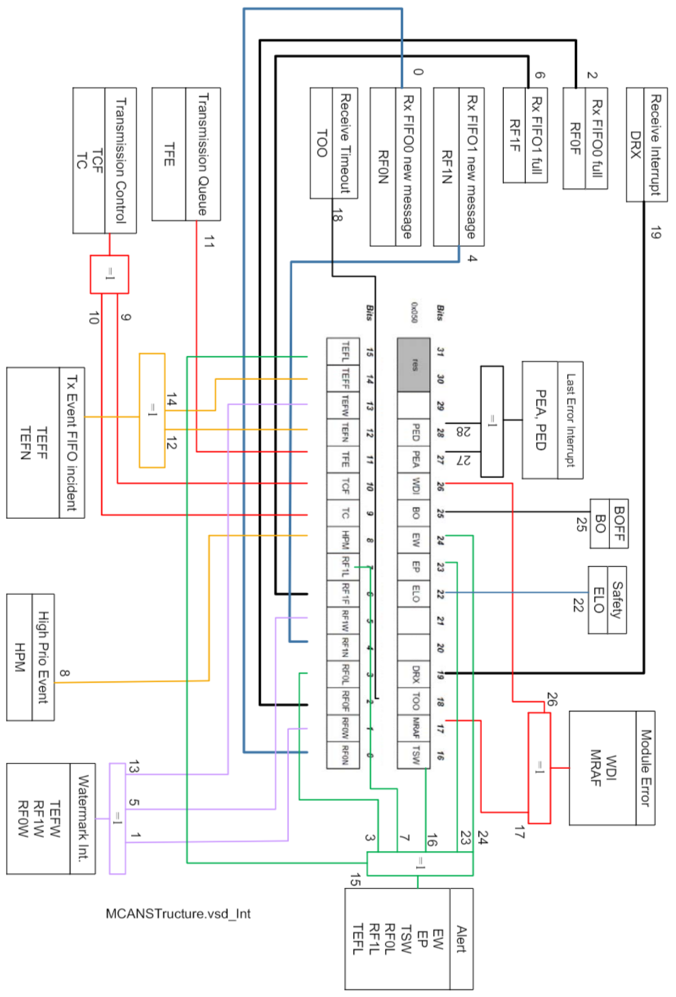


Figure B.1: Interrupt mapping to groups. Taken from [7].

Appendix C
Benchmark results

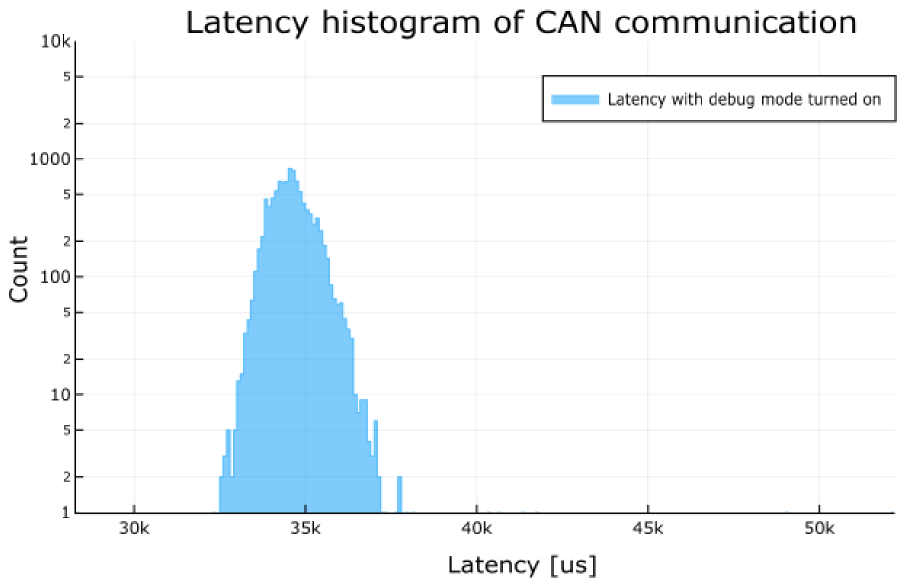


Figure C.1: Single-core CAN example latency communication with Debug mode turned on.

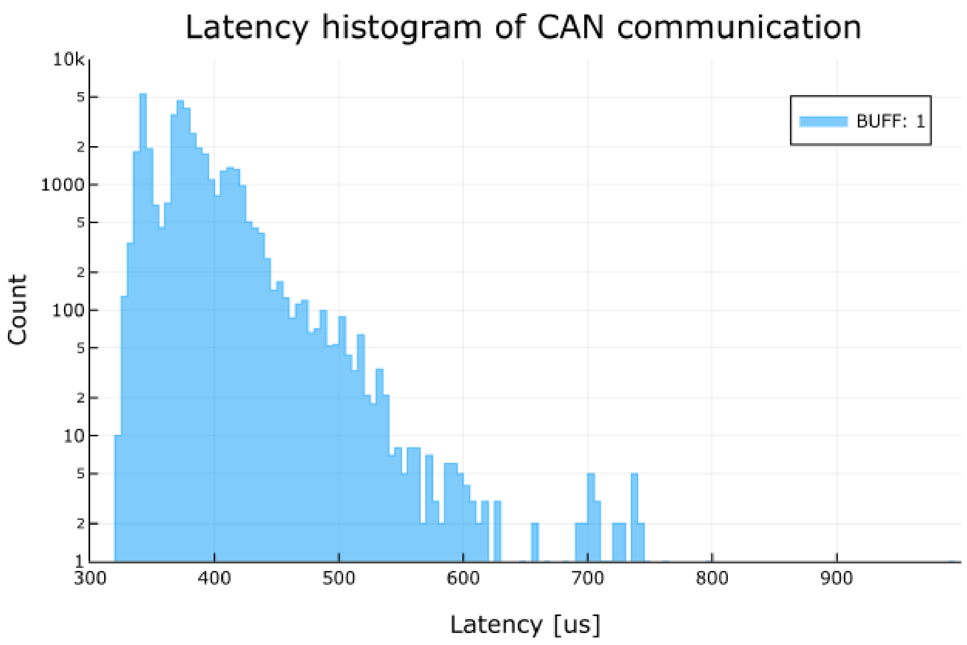


Figure C.2: Single core CAN example latency with a hardware buffer size set to 1 message

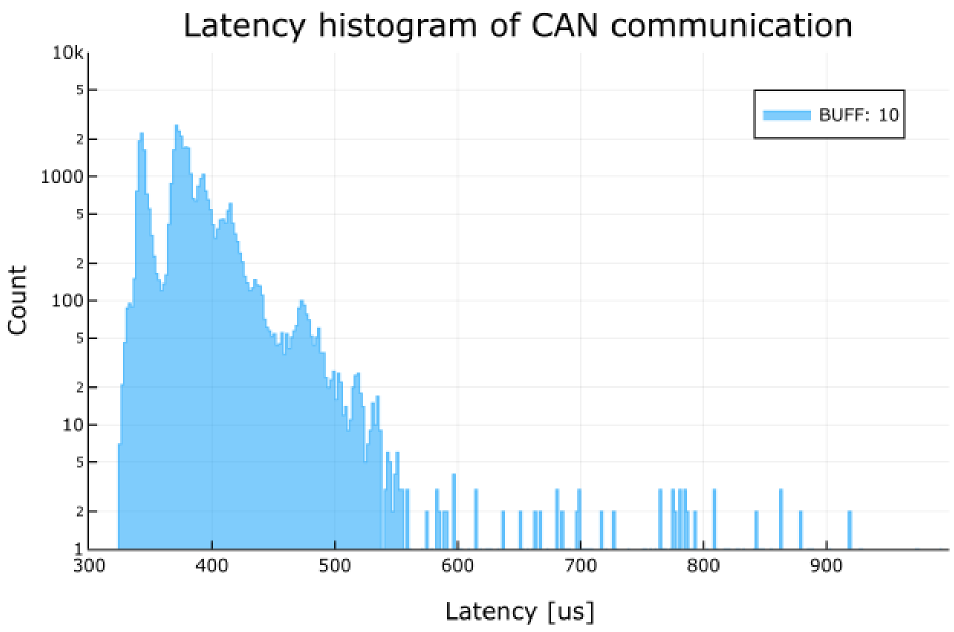


Figure C.3: Single core CAN example latency with hardware buffer size set to 10 messages

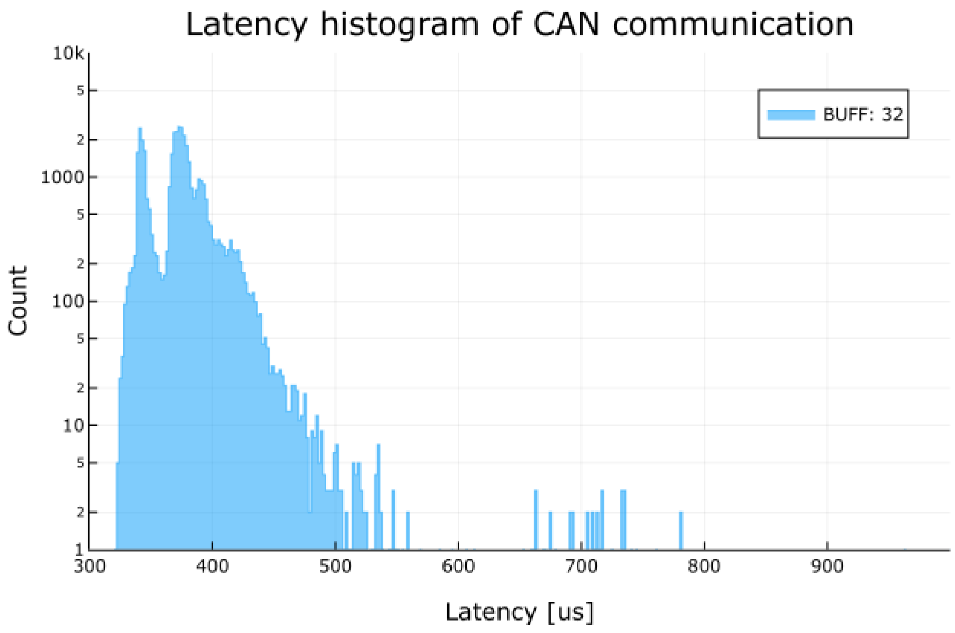


Figure C.4: Single core CAN example latency with hardware buffer size set to 32 messages

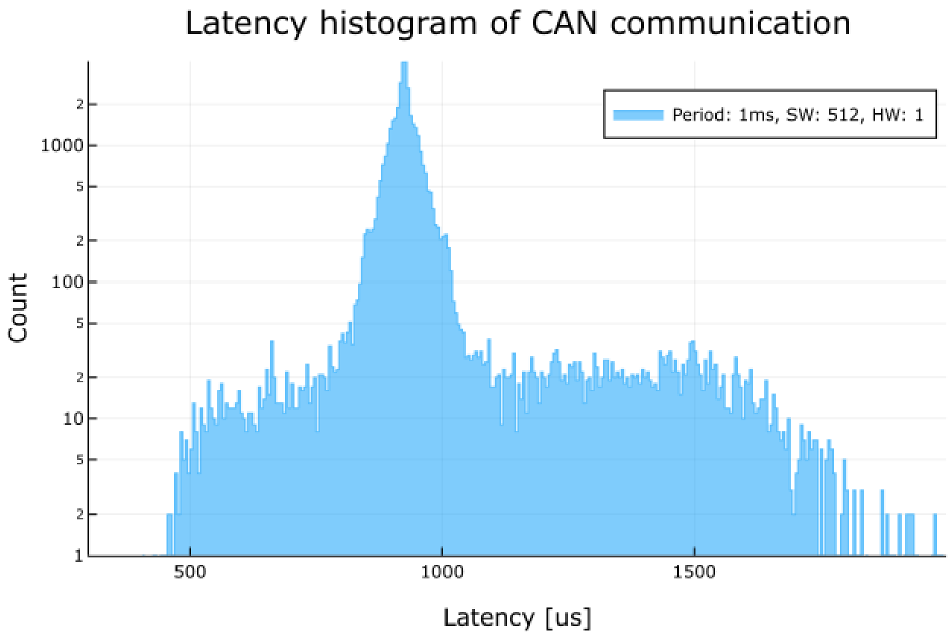


Figure C.5: Multi-core CAN example latency with reply task period 1 ms SW buffer size 512 messages and HW buffer size 1 message

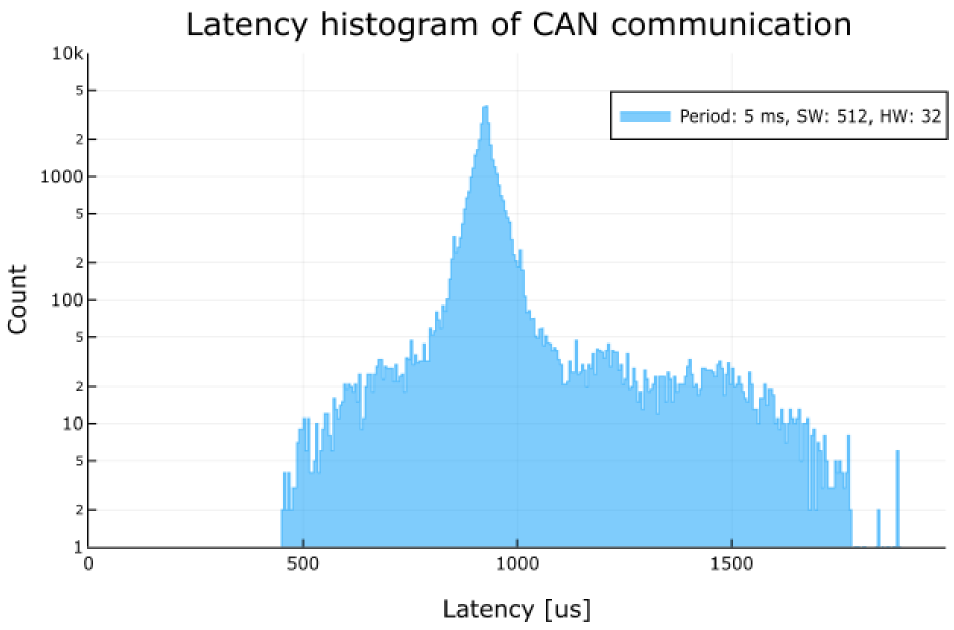


Figure C.6: Multi-core CAN example latency with reply task period 1 ms SW buffer size 512 messages and HW buffer size 32 message

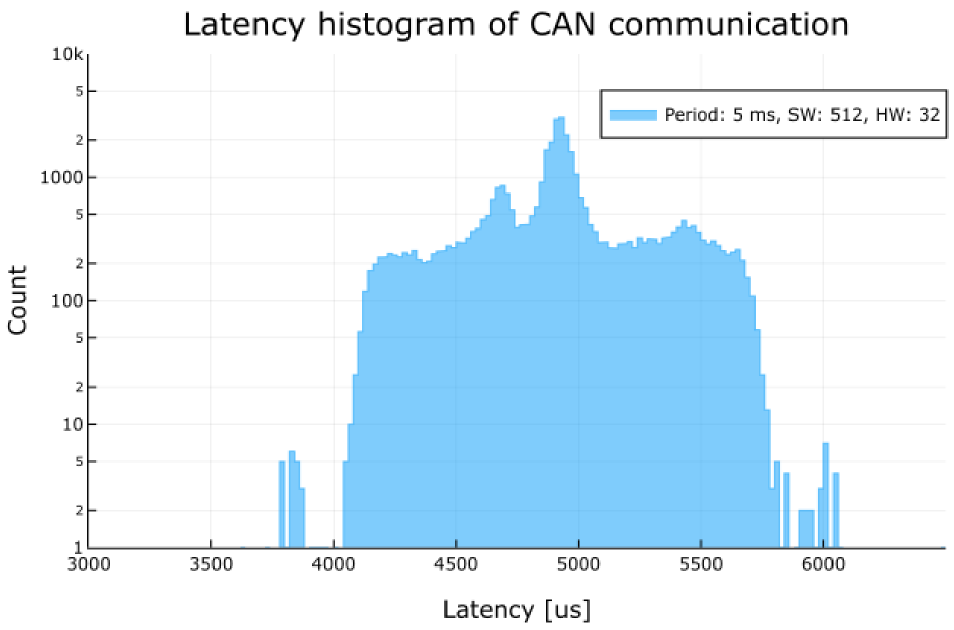


Figure C.7: Multi-core CAN example latency with reply task period 5 ms SW buffer size 512 messages and HW buffer size 32 message

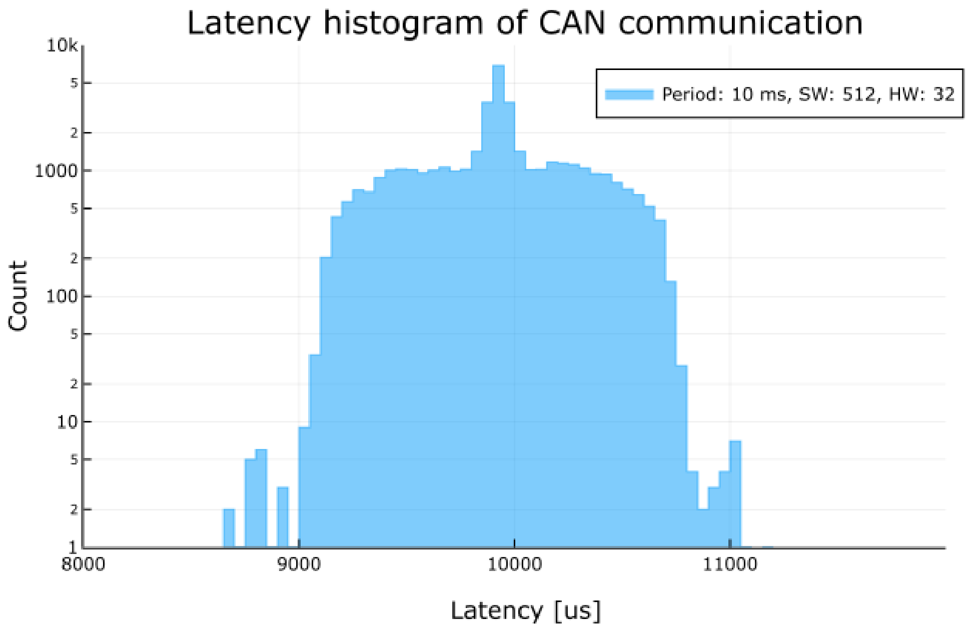


Figure C.8: Multi-core CAN example latency with reply task period 10 ms SW buffer size 512 messages and HW buffer size 32 message