Politecnico di Torino

College of Computer Engineering, Cinema and Mechatronics

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Study and Development of Advanced Core Communication and Memory Management Techniques for Hard Real-Time Battery Electric Vehicle (BEV) Applications

Supervisor:
Ph.D. Sarah AZIMI

Candidate:
Giuseppe DI CAROLO

Co-supervisor:
Prof. Luca STERPONE

Academic year 2022-2023

Abstract

This thesis focuses on the optimization of electric motor control for battery electric vehicles (BEVs) by leveraging Direct Memory Access (DMA)-based core communications. Various procedures were studied, tested, and compared to determine the most suitable approach to meet the operating specifications of the target study.
The first thesis activity involved a comprehensive examination of state-of-the-art data transfer techniques found in the scientific literature with the objective of discerning advantages and disadvantages within the context of hard real-time systems. Simultaneously, an in-depth exploration of the memory modules and communication components of the AURIX™ TriBoard TC399 was conducted, complemented by practical experimentation. The DMA constituted the main focus of the research alongside its configurations, which not only offer enhanced flexibility but also relieve the cores from the burden of data copying at the expense of an increased initial programming overhead.
In the concluding part of this work, the acquired knowledge was translated into practical application through the development of C-code programs using the "Aurix Development Studio" platform. These programs were designed to implement various strategies. The outcomes of these implementations were compared to identify the approach that best aligns with the specified requirements.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and goal of the thesis

Nowadays, a gradual but rapid evolution is taking place in the automotive world, leading to the replacement of internal combustion engines with pure electric or hybrid engines. The world of research has shifted almost entirely to the latter, specifically discovering the various possibilities arising from the integration of more specific control systems allowed by the substantial electronic component.
In this particular study, the final application involves the realization of a two-cell hybrid electric motor.
The image displayed in Fig. 1.1 shows a simplified block diagram of a hybrid motor. The hydrogen cell is responsible for generating electricity through a chemical reaction of electrolysis between hydrogen and oxygen. This current powers the electric motor and provides traction.
The power source section represents the electric motor system's dual-cell power source, which includes both a battery and a fuel cell. These two power sources work together to provide electrical energy to the system. On the other hand, the electrical control unit manages and controls the flow of electricity within the system. The inverter in this unit is responsible for converting the direct current (DC) from the battery, fuel cell, and generator into alternating current (AC) that the motor can use.

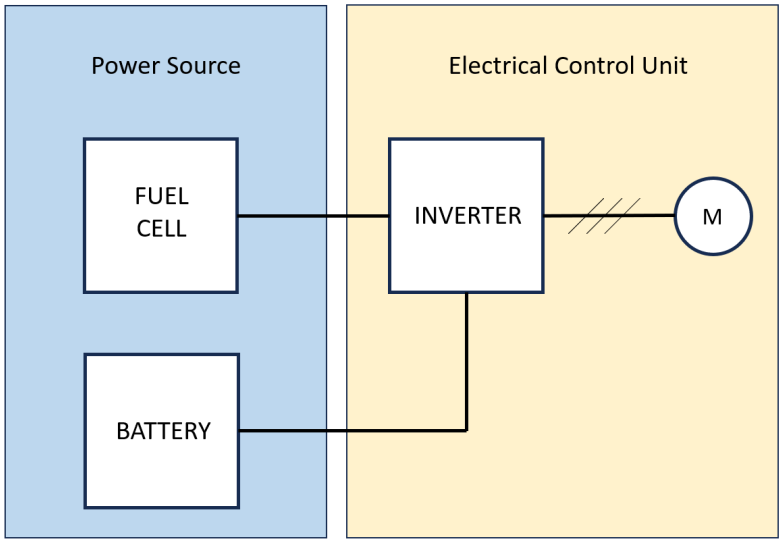


Figure 1.1: Hybrid motor dual cell block diagram

Regenerative braking occurs when a vehicle brakes or decelerates, converting motion into electrical energy. The electric battery acts as an energy buffer for the whole system. The electrical energy generated during regenerative braking is stored in the battery for later use in situations where the power demand exceeds the capacity of the hydrogen cell, powering the electric motor. As a result, energy management is a critical aspect that is essential for coordinating energy distribution and optimizing vehicle efficiency.

The strategies used to best achieve this goal are becoming increasingly varied, diversified, and specific to the individual end-use application.

Therefore, it is necessary to pay attention to the specific model and instrumentation in order to properly implement it.

The objective of the work was to investigate the potential of the Infineon TriBoard TC399 in the automotive field. The primary focus was to experiment and determine the most effective intra and extra-core communication mechanism, a crucial component of the project.

For the initial phase of the project, I researched the architecture and components of the board. Simultaneously, I also researched scholarly works on fundamental communication to gain a deeper insight into the latest advancements. Afterward, I moved on to the programming phase, which involved utilizing the AURIX Development Studio

to test out various timing and performance strategies. Finally, I drafted an initial program that would be capable of supporting the end goal of the project.

1.2 Outline of the Thesis

The thesis work has been divided into the following chapters:

1. **Introduction**: preamble to the research project and to the goals of this thesis.
2. **Background**: presentation of the main arguments and components that the thesis focuses on.
3. **State of the art**: a short overview of the current methods utilized in DMA communications.
4. **Developed Methodologies** : presentation and discussion on the final application project and the techniques developed to better meet its requirements.
5. **Experimental results**: test development on the critical components of the project and presentation of a code to accommodate the final application.
6. **Conclusion**: Completing the final version of the code application.

Chapter 2

Background

Since this thesis primarily addresses memory management and its function in communicating and exchanging data between the processor and peripheral modules, it is essential to provide a brief introduction of the main components under focus.

2.1 Memory

In informatics, the term "memory" refers to a fundamental component of a computer system that is used to store and retrieve data and instructions for processing by the central processing unit (CPU). Memory is crucial for the functioning of computers and is divided into several types, each with its specific purpose and characteristics.

2.1.1 Volatile and non-volatile memory

Volatile and non-volatile memory are two fundamental types of computer memory. Volatile memory refers to temporary storage that is lost when the computer is turned off, such as RAM. Non-volatile memory, on the other hand, retains its data even when power is lost, such as a hard drive. While volatile memory is faster than non-volatile memory, it is also more expensive and cannot store as much data. Non-volatile memory, while slower, is generally more reliable and can store large amounts of data. These two memory types are used in different ways depending on the specific needs of a computer system. As a result, nonvolatile memory is necessary for booting up the operating system, but it is much slower than volatile memory. Therefore, both

types of memory are required.
Memory management is critical as performance and capacity are opposing parameters that require careful consideration. In particular, the hierarchy of memory is shown in the figure below.
Disks are based on magnetic properties, they can store a lot of information, but they are slow. On the other hand, all the others are silicon are constructed with silicon and are volatile memories. Dynamic RAM (DRAM) is built with capacitors, while cache memories use static RAM, which is ten times faster than DRAM.

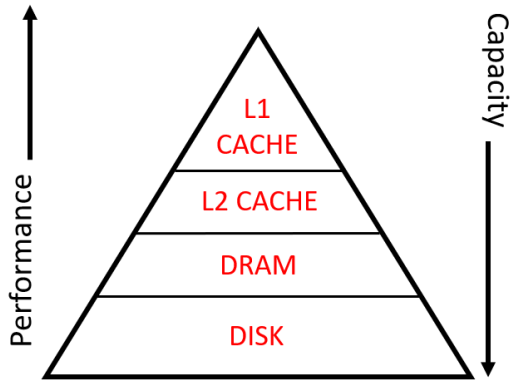


Figure 2.1: Memory hierarchy

2.1.2 Basic computer architecture

Software, consisting of data to be processed and instructions for processing the data, are fed into computers through input devices and are made available to the user through output devices.
The CPU's role is to connect to memory input/output devices via control/data/address buses. CPU operation always consists of two complementary cycles. A read cycle is a sequence of operations that the CPU initiates to get bits of data from an external peripheral, and a write cycle is the operation to be performed to deliver bits of information. More specifically, the CPU always performs an infinite loop: An instruction is fetched from memory (read cycle) and its execution may result in a read cycle, a write cycle, or an update of the internal state of the CPU.

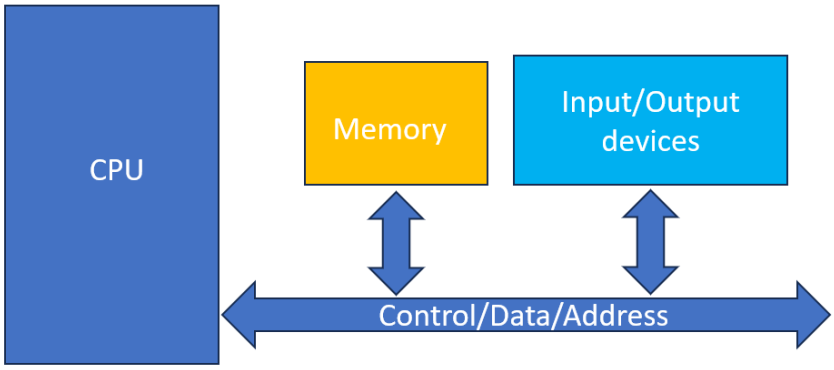


Figure 2.2: Basic computer architecture

If the memory elements are physically placed inside the CPU itself, they are called registers and can be divided into categories according to their role. The general purpose registers are used as containers for data to be processed, results of operations, and other specific elements. Special purpose registers are intended for specific tasks while the Program Counter stores the address of the next instruction to be executed. The CPU can work directly with the memory or with the memory in the I/O ports. The main advantage of registers is that the read/write procedure is faster since in general, the rule applies that the closer, the faster.

2.1.3 CPU memory map

Finally, Let's discuss how the CPU and peripherals communicate. The processor generates addresses that depend on the BUS dimension in bits, which can be up to 2^{N} [1]. Each peripheral is assigned a range of addresses, such as RAM and ROM. This association between addresses and peripherals is described in the memory map. Even something as simple as a keyboard is considered a memory element for the purpose of clarity and simplicity.
Inserire immagine della memory map.

[1] N = number of bits of the address bus

2.1.4 Direct Memory Access (DMA)

The central processing unit (CPU) is responsible for controlling the address, data, and control bus. However, the direct memory access (DMA) controller can perform the same operations simultaneously with other processor activities. In fact, the DMA can perform memory read/write cycles when instructed by the CPU, but no other operation. Utilizing DMA, the processor is only involved once to initiate the transfer, allowing it to perform other tasks simultaneously. However, there is only one bus, meaning that if DMA is accessing the memory, the CPU cannot do the same. It can only execute different operations or wait. Therefore, the CPU and DMA cannot access the memory simultaneously.

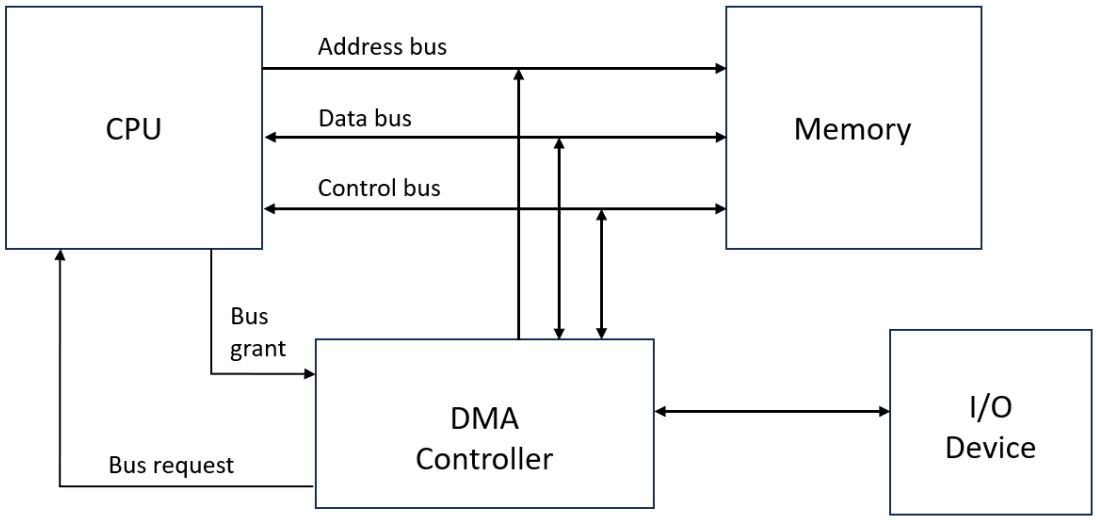


Figure 2.3: Typical architecture of the DMA

There are three different ways to program the DMA in order to prevent conflicts with the processor. The first method is Burst, which involves transferring data in a single operation. However, if the CPU needs to use the bus during the transfer, it must pause, which could cause issues if the CPU requires immediate attention. The second option is Cycle Stealing, where the data is divided into smaller chunks. After each transfer, the CPU can access memory, so it won't be stuck for an extended period. Finally, the DMA only operates when the CPU isn't using the bus, resulting in the slowest transfer rate.

2.2 Microcontroller (MCU)

2.2.1 General architecture

The term "microcontroller" refers to a small integrated circuit capable of independently performing a set of specific tasks or functions. In the realm of embedded systems and automotive applications, microcontrollers are widely used for various purposes, including anti-lock braking systems, infotainment, and airbag systems. These microcontrollers consist of a processor (CPU) that is solely responsible for executing instructions and performing calculations. The CPU's performance varies based on the application, depending on the architecture.
Furthermore, a microcontroller also has memory components, including read-only memory (ROM) for firmware storage and random access memory (RAM) for temporary data during program execution. The microcontroller also has input/output (I/O) peripherals that allow it to interact with external devices and sensors, such as digital and analog pins, timers, counters, and serial communication interfaces like UART and SPI.
Finally, an internal clock generator is essential to synchronize the microcontroller components' operation by providing timing signals.

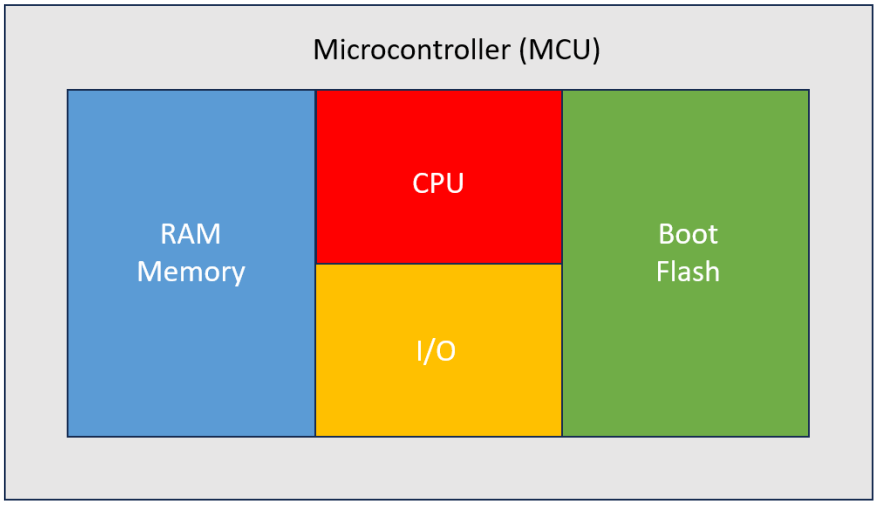


Figure 2.4: Architecture of a microcontroller

2.2.2 SOC

For the sake of completeness, let's discuss the System on Chip architecture.

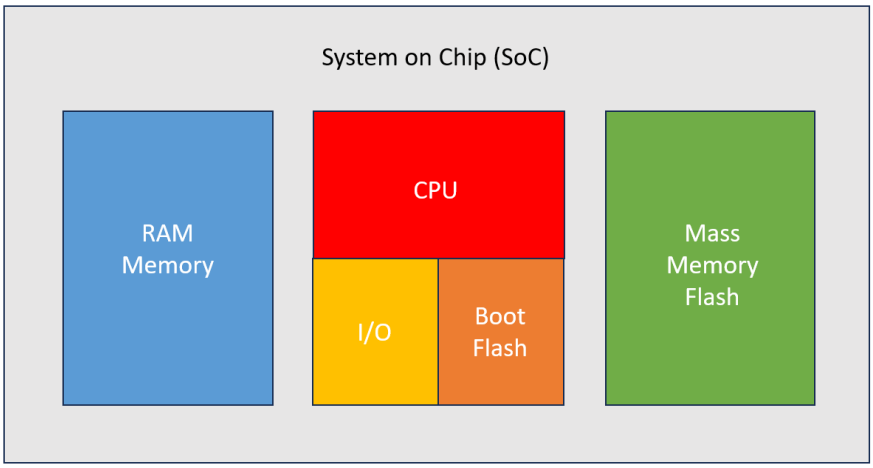


Figure 2.5: Architecture of a System on Chip

This design includes discrete components, with the CPU only integrating certain elements (see Fig. 2.5 for reference, namely the I/O and boot flash). While this architecture is more costly, it is also more adaptable. One example of an SoC in the automotive industry is an infotainment system, where additional components can be added based on customer preferences to enhance functionality. Nevertheless, the foundation remains the same SoC.

2.2.3 AURIX^{TM} TriBoard TC399

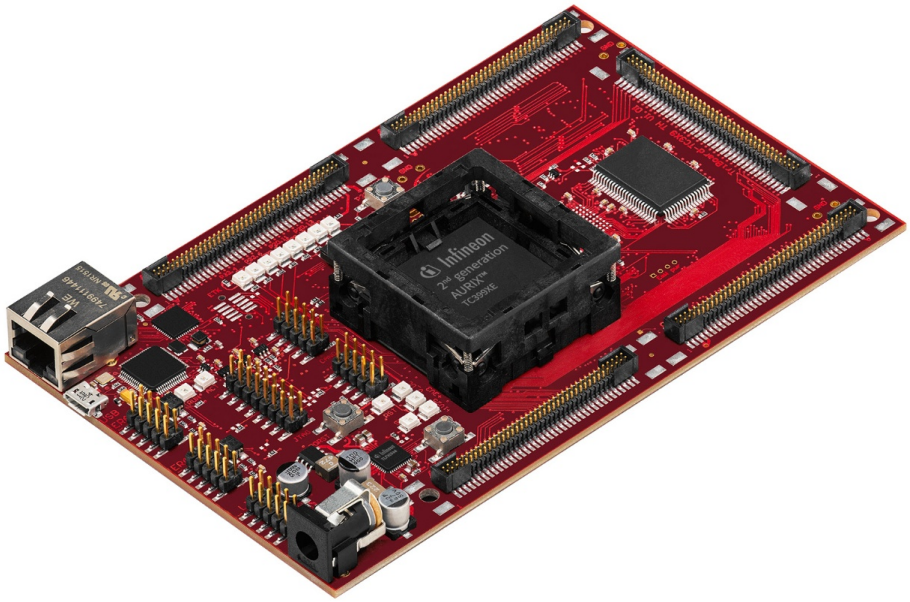


Figure 2.6: AURIX^{TM} TriBoard TC399 [1]

As extrapolated from [2], the Aurix TC399 microcontroller is part of the Infineon Aurix family, which is specifically designed for automotive and industrial applications that require high performance, safety, and reliability. Accordingly, it offers a number of features and characteristics that make it suitable for demanding real-time systems. In terms of architecture, the Aurix TC399 is built around a powerful 32-bit TriCore processor. The TriCore architecture combines the capabilities of a microcontroller, a microprocessor, and a digital signal processor (DSP). This architecture is optimized for real-time control tasks, making it well-suited for automotive and industrial applications. The TC399 is optimized for real-time control applications where deterministic and predictable task execution is critical. It includes dedicated hardware for handling interrupts, timers, and event-driven operations. Furthermore, Infineon provides a comprehensive development ecosystem including software development tools, libraries, and documentation to support the development of applications for the Aurix TC399 microcontroller.
To conclude this brief overview of Infineon's TriCore, please take a look at the block diagram below in Fig. 2.7. It shows the main components of the integrated circuit

and how they interact.

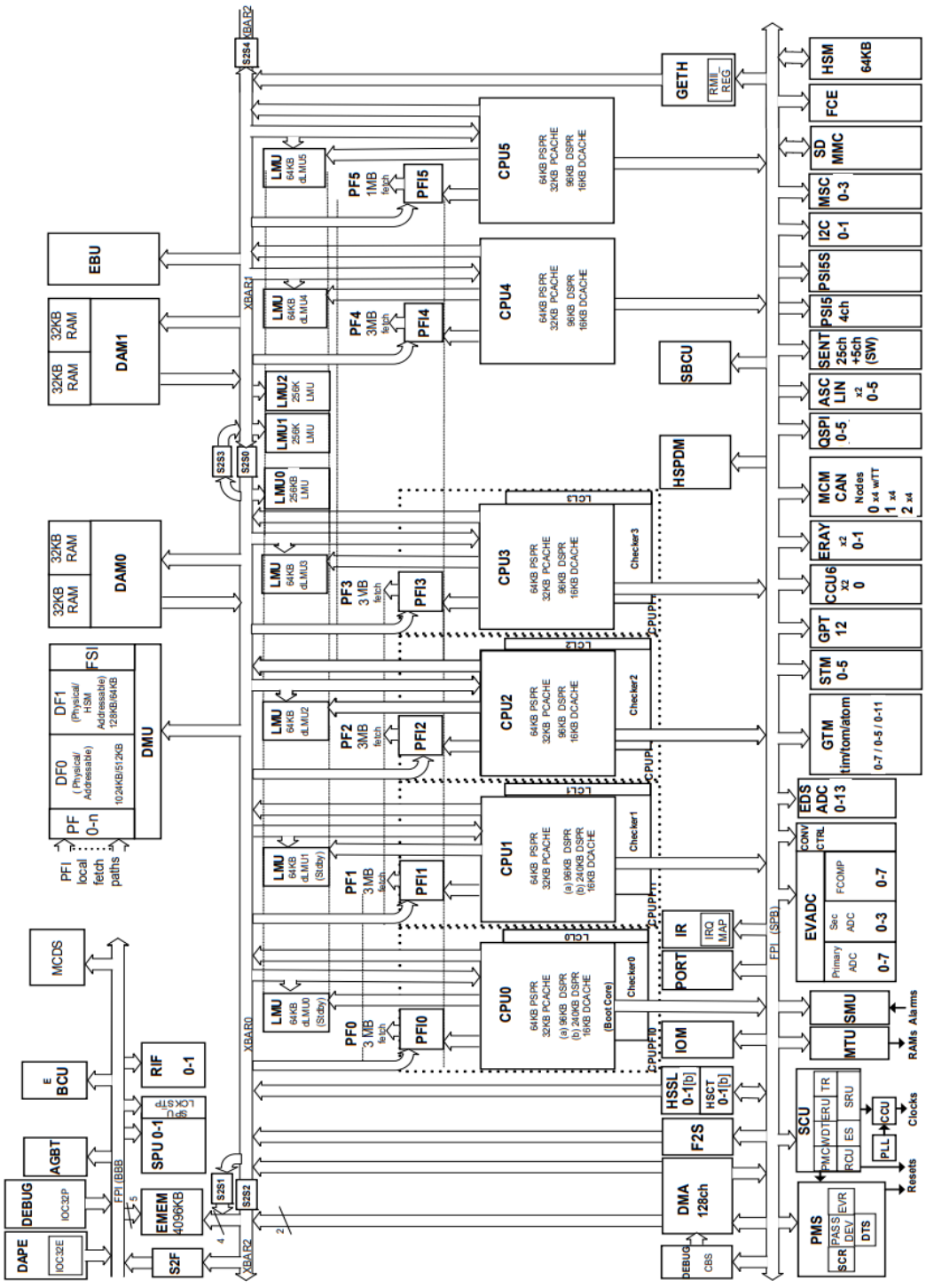


Figure 2.7: Block diagram [2]

2.3 DMA in AURIX^{TM} TriBoard TC399

As introduced before, the DMA shall move data from source locations to destination locations without the intervention of the CPU or other chip devices.
Moving on to the description of the main components and related characteristics, it is worth noting that:

- The DMA supports 128 independent and individually programmable channels, each of which is assigned to a resource partition and stores the context of an independent DMA operation.
- A DMA channel is activated by a DMA request that can be:
  - DMA Software Request initiated by a CPU.
  - DMA Hardware Request initiated by the Interrupt Router (IR) Interrupt Control Unit (ICU).
  - DMA Daisy Chain Request initiated by the next higher priority DMA channel.

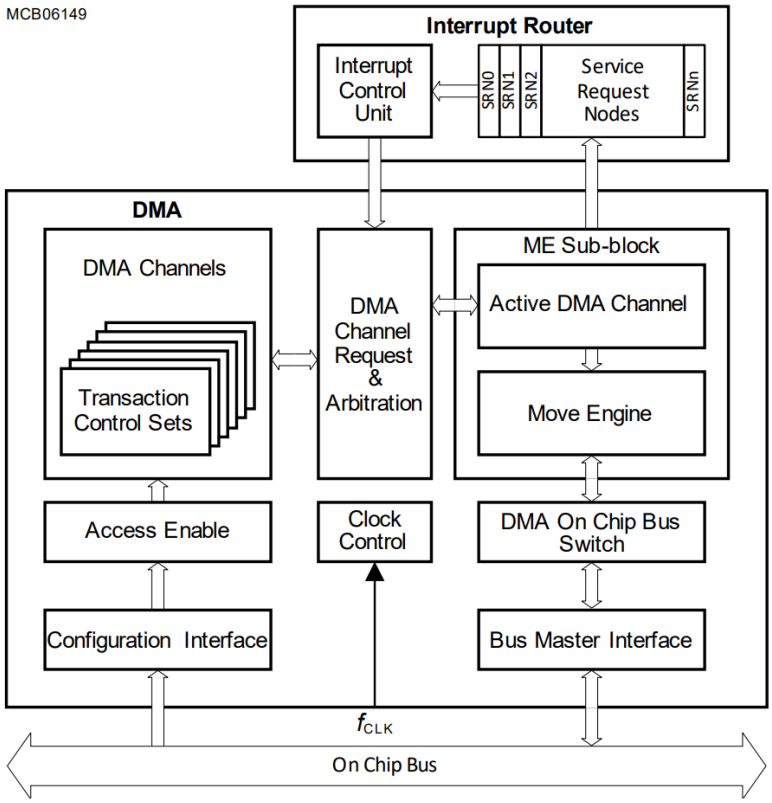


Figure 2.8: Block Diagram of DMA in [2]

- A DMA transaction, as represented in Fig. 2.9, can be divided into:
  - Transfers: whose sum defines a transaction.
  - Moves: whose sum defines a transfer.

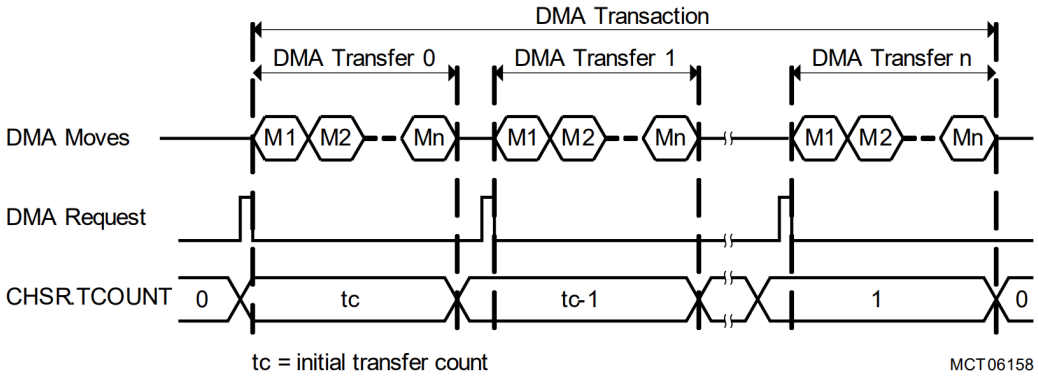


Figure 2.9: Transaction, Transfer, Move [2]

This is noteworthy because it is possible to define the next source and destination addresses after each DMA move. Such addresses can be selected by choosing their offset from the previous one (e.g. from $00H$ to $08H$ and their direction (addition, subtraction, or none). Finally, a circular buffer configuration is also possible.

2.4 Target memory

Still referring to [2], this section will provide a brief review of the two main memory types studied for the final application. Their key features will be presented to enhance comprehension of the decisions and reasoning in chapter 4.

2.4.1 Local Memory Unit RAM (LMU) & Data Scratch-Pad RAM (DSPR)

The Local Memory Unit RAM (LMU) provides 256 KiB[2] of local memory for general purpose usage (Fig. 2.7) that can be used for code execution, data storage or overlay memory. It also can be configured with a Memory Protection Unit (MPU) that allows fine-grained control over access to memory regions.

On the other hand, Data Scratch-Pad RAM (DSPR) provides 240 kB in CPU0 & CPU1, 96 kB in the other CPU's (Fig. 2.7).

[2]kilo binary byte: 1 KiB = 2^{10} byte

One of the main differences between the two configurations is that LMU has interconnection with the CPU core via Shared Resource Interconnection (SRI) while DSPR can be addressed on each core without using the bus. As a result, accessing the DSPR is generally faster than accessing the LMU.

Chapter 3

State of the art

This chapter aims to provide a comprehensive overview and analysis of the literature concerning the utilization of Direct Memory Access (DMA) for both Intra and Extra-core communications. It will also examine the advantages and disadvantages of this technique in the context of the automotive industry compared to other commonly used methods.

3.1 Logical Execution Time (LET) paradigm

The concept of Logical Execution Time (LET) is a programming technique that provides predictable timing and can be easily combined with other programming tools. This approach has gained popularity in the automotive industry, where it has been effectively utilized to manage the distribution of software applications across multiple electronic control units.

The LET paradigm, first introduced in [3], operates on the following principles:

- Periodic tasks, denoted as τ_i, are tasks that are initiated at regular intervals.
- LET read: Each periodic task updates its input values at the moment it is released. Defined as a triple $W(\tau_i, d_x, t)$, task τ_p makes available to a consumer τ_p the instance of d_x produced during its job completed at t.
- LET write: These values are then used to calculate new output values, which

are available to consumers at the end of the period. Defined as $R(d_x, \tau_c, t)$, task τ_c acquires the value of d_x available at t.

- LET read and LET write are generally known as LET communications.
- Communications are carried out instantaneously, resulting in a deterministic behavior over time.

In Fig. 3.1, the timeline demonstrates the intended invocation of task τ based on the principles stated. Note that $\tau_s start$ and $\tau'_s start$ coincide due to instantaneous communication.

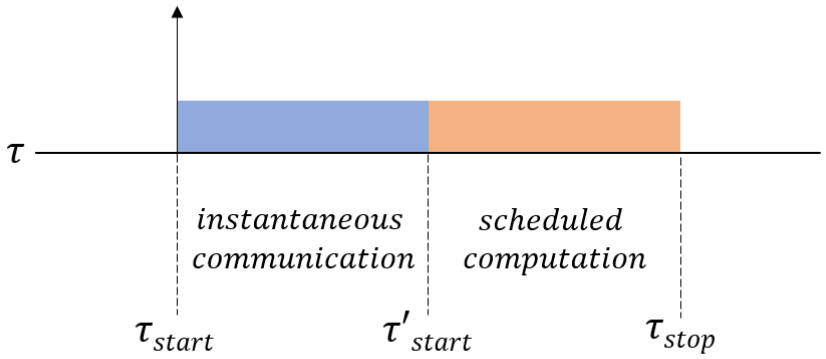


Figure 3.1: Timeline for invocation of task τ

The key properties can be summarized into three fundamental properties:

1. Property: All LET writes must occur before causally-related LET reads.[1]

$$W(\tau_i, d_a, t_{i,x}) \prec (d_b, \tau_i, t_{i,x})$$

   This condition should apply to all release times $t_{i,x}$ and each variable d_b.

2. Property: At any point in time, Before starting LET reads of τ_c, LET writes of producer task τ_p for data d_a must be completed.

$$W(\tau_p, d_a, t) \prec (d_a, \tau_c, t)$$

[1] $a \prec b = a$ must be completed before starting b.

3. Property: Communications that are issued at different times must not overlap. For each pair t_1, t_2 where $t_1 < t_2$, all LET communications assigned to t_1 must be completed before starting those assigned to time t_1.

3.1.1 Implementation of LET

The authors of the cited book [3] proposed an order of execution for LET communications to satisfy the aforementioned properties. The sequence to be followed is:

1. At t, each task instance must complete all of its LET writes.
2. Subsequently, each task instance released at t performs all its LET reads.
3. Finally, all task instances that were released at time t are now set as ready.

3.1.2 Limits and issues

By carefully analyzing the strategy presented, it is possible to discern its limitations and major issues.
Firstly, when a task is released at a certain time t, it must wait for all LET write and read operations from other task instances that complete and start at t, regardless of whether they have any causal dependencies with the task in question. This can cause unnecessary delays, particularly for tasks that require frequent communication.
Additionally, this approach does not consider that tasks may have varying levels of priority. Thus, higher-priority tasks may have to wait for lower-priority tasks to finish communicating, resulting in a priority inversion.

3.2 DMA LET communications

According to [4], using the DMA module with refinements can resolve problems that make using the LET protocol impossible for hard real-time applications.

After reviewing the benefits of using DMA, the article highlights two main advantages. Firstly, using DMA allows for limited interference during task execution by offloading

data transfers. Secondly, it offers the possibility of a more flexible order of LET communications.

3.2.1 Intra-Core Communications

Let's consider the scenario of two tasks: a producer (τ_p) and a consumer (τ_c) that are functionally dependent and mapped onto the same core. To handle communication within the core, a technique called triple buffering can be used, as explained in [5] and displayed in Fig. 3.2.

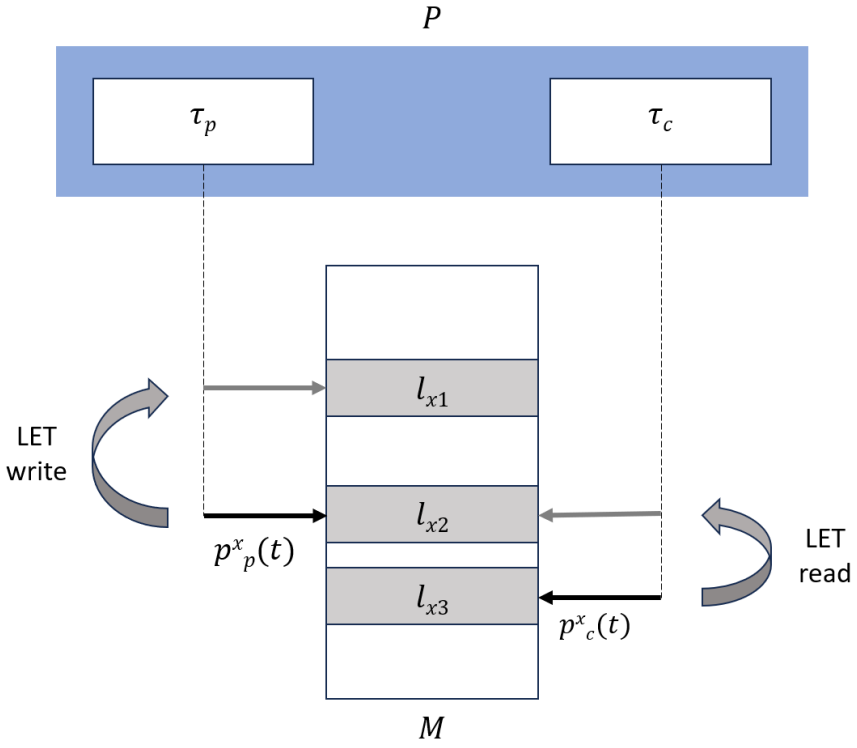


Figure 3.2: Intra-Core [4]

Again with reference to Fig. 3.2, P refers to the core, l_{x1}, l_{x2} and l_{x3} are the labels associated to the local memory M, $p_p^x(t)$ and $p_c^x(t)$ are the pointer assigned to the tasks τ_p and τ_c respectively.
Initially, $p_p^x(t)$ and $p_c^x(t)$ point respectively to l_{x2} and l_{x3}. As soon as τ_p is released, $p_p^x(t)$ is reassigned to l_{x1}, while l_{x2} retains its previous value. During the next instance

of τ_c, $p_c^x(t)$ will switch to l_{x2}, which is left pending. The process continues cyclically.

3.2.2 Inter-Core Communications

Resuming the work done by [4], let's consider the case in which the communication follows the local to global to local pattern, as shown in Fig. 3.3.

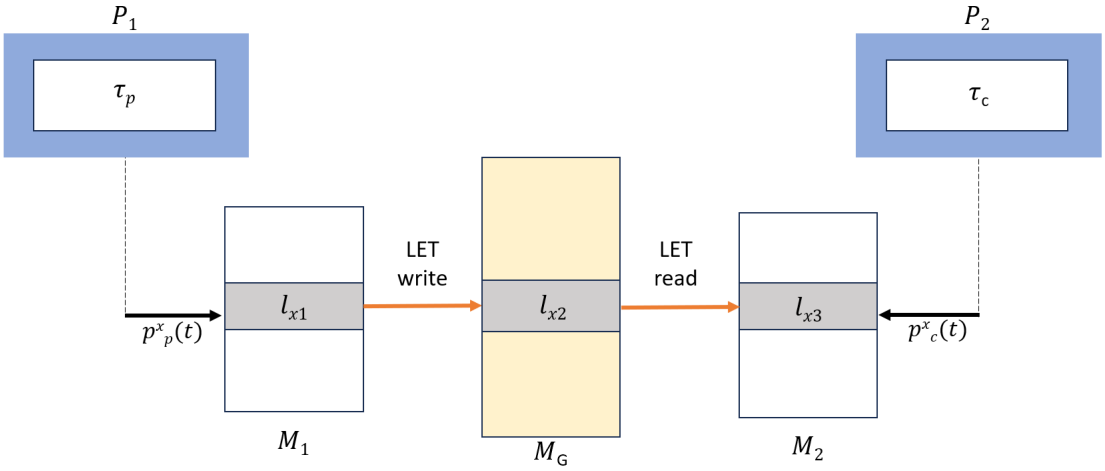


Figure 3.3: Inter-Core [4]

When executing two functional dependency tasks, consumer τ_c and producer τ_p, on separate cores, both LET writes and reads are implemented as physical copies of data between labels in a global memory M_G. This means that pointers $p_p^x(t)$ and $p_c^x(t)$ will always point to l_{x1} and l_{x3} without switching. Either way, intra-core communications can also be handled with this design.
It is important to note that the global memory in this platform is typically slower, which may not make it the ideal choice for hard real-time systems such as the one being the main focus of this project.

3.3 DMA Configurations

The DMA is responsible for transferring shared data from one source memory M_s to a different destination memory M_d. When programming the DMA, multiple LET communications can be combined into a single DMA transfer. Each DMA transfer

involves a continuous portion of memory in both M_s and M_d. As specified in the previous chapter, a DMA transfer requires the definition of the start address of the label, the start address in the destination, and the size of the data transfer.

Reviewing the possible configurations as reported in [4] referring to [6]:

- SIMPLE: The DMA is programmed for each data transfer and triggers an interrupt upon completion.
- "LL-EOT" and "LL-EOL" modes: The DMA is programmed to perform multiple transfers. LL-EOL triggers an interrupt at the end of each transfer, while LL-EOT triggers an interrupt at the end of the list.

3.3.1 Timing of DMA

Let's now discuss the timing characteristics of the two techniques presented, based on the results in [7].

When using the SIMPLE mode for transfers, we can divide the time needed into two parts: σ_{IN}, which is the initialization time for the DMA, and σ_{DT}, the time it takes to upload the command for a single transfer. According to [7], this overhead time σ_{DT} is always less than σ_{IN}, and it does not depend on the label size. Additionally, we need to factor in the time σ_{ISR} needed for the ISR to notify the DMA once the transfer is complete.
After analyzing the data provided, it can be inferred that the Direct Memory Access (DMA) process requires a longer time to execute a linked list transfer in LL-EOT and LL-EOL modes, as opposed to a single transfer performed in SIMPLE mode. Nonetheless, it is still more efficient than programming each transfer individually [7].[2]

Given that $\sigma_{DT} < \sigma_{IN}$ and $\sigma_{DT} < \sigma'_{DT}$;

$$\sigma_{DT} + \sigma_{IN} < \sigma'_{DT} + \sigma'_{IN}$$

$$N \cdot \sigma_{DT} + N \cdot \sigma_{IN} > N \cdot \sigma'_{DT} + \sigma'_{IN}$$

[2] N = number of DMA transfer.

3.4 Final example and considerations

To utilize the findings of the preceding sections, this chapter will conclude with an example summary that draws some conclusions. Firstly, regarding the memory layout in Fig. 3.4, task τ_1 on processor P_1 performs write operations on l_c and read operations on l_a and $l_{b'}$, while task τ_2 on P_2 performs a write operation on l_d.

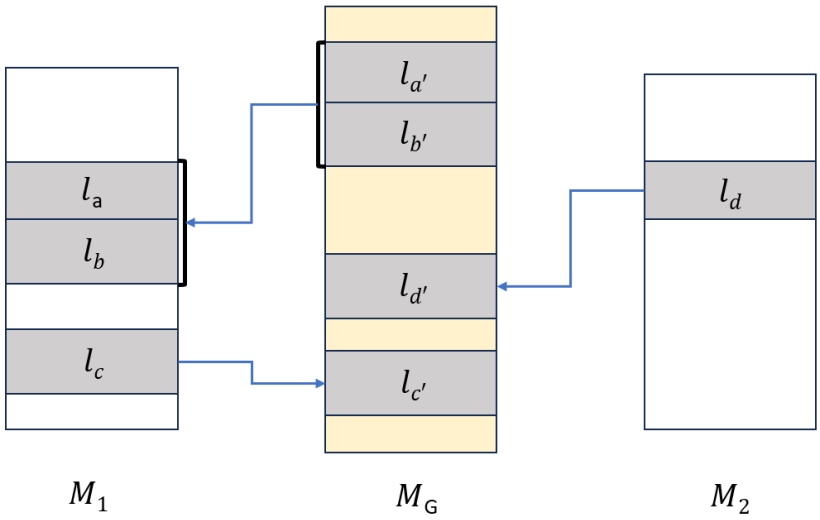
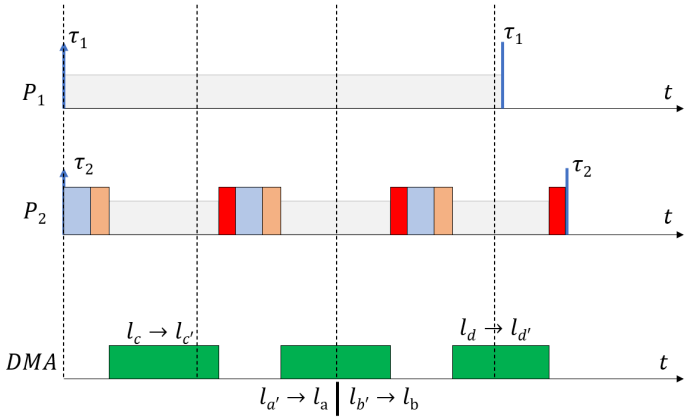


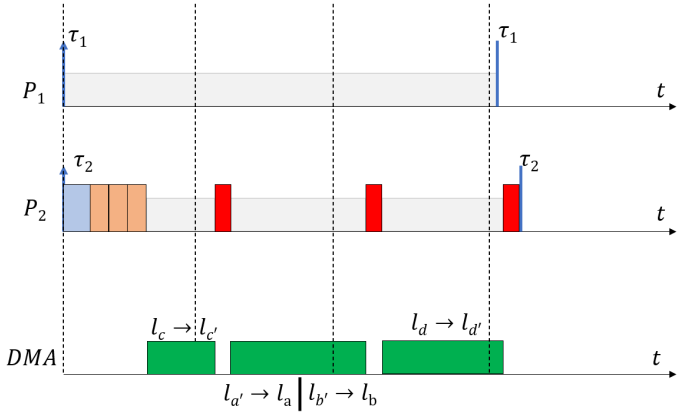
Figure 3.4: Memory layout [4]

In this example, the three studied configurations: SIMPLE, LL-EOT, and LL-EOL, are represented in Fig. 3.5 and described in the legend shown in Fig. 3.6. The first line shows how a task τ_1 is managed without using DMA, which is the same for all configurations. However, the second and third lines show the differences in time for each configuration.

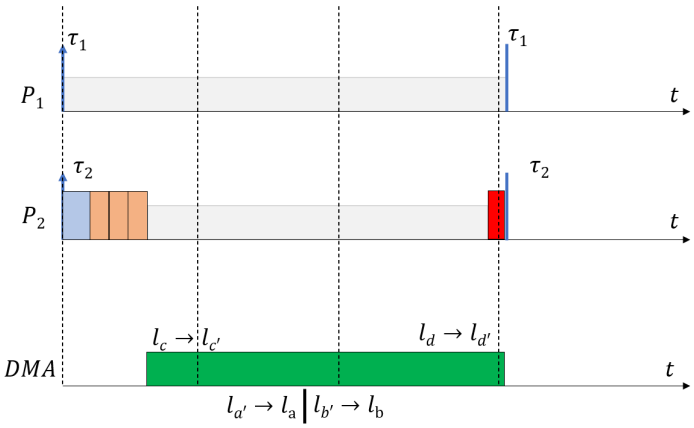
If the timeline for processor P_2, which is managed by DMA, is analyzed, it can be seen that the inequalities in section 3.3.1 are respected. Specifically, the SIMPLE configuration is the slowest due to the delay time caused by σ_{IN} and σ_{DT}. In contrast, LL-EOT and LL-EOL are quicker and can finish task τ_2 more rapidly because they don't need to constantly recall the DMA. Specifically, LL-EOL is the fastest since it doesn't require time to call the interrupt after each transfer completion.



(a) *SIMPLE configuration [4]*



(b) *LL-EOT configuration [4]*



(c) *LL-EOL configuration [4]*

Figure 3.5: DMA configurations [4]

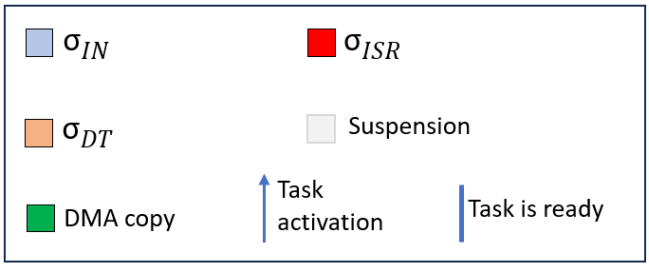


Figure 3.6: Legend [4]

To summarize, DMA engines can significantly improve the performance of automotive systems by efficiently transferring multiple labels with minimal processor intervention. However, it's important to note that a single DMA transfer or list may contain data from various tasks, which can cause delays until the DMA completion ISR.
Therefore, determining the optimal grouping of LET communications in DMA transfers, as well as the order and grouping in linked lists, requires careful analysis on a case-by-case basis. It cannot be generalized for all scenarios.

Chapter 4

Developed Methodologies

4.1 Introduction to the final application

To better understand the methods and techniques developed and tested in this thesis, let's delve into a more specific description of the target system and its time-related aspects.
In particular, the attention in this thesis, as shown in Figure 1.1, is directed towards the "Power Source" block. This "Power Source" block comprises two identical DC/DC converters, a fuel cell, and a battery.
By manipulating the MOSFET gate signals, which, in turn, regulate the duty cycle (i.e., the duration they are in the "on" state), the speed of the motor is controlled.

4.1.1 Electric scheme

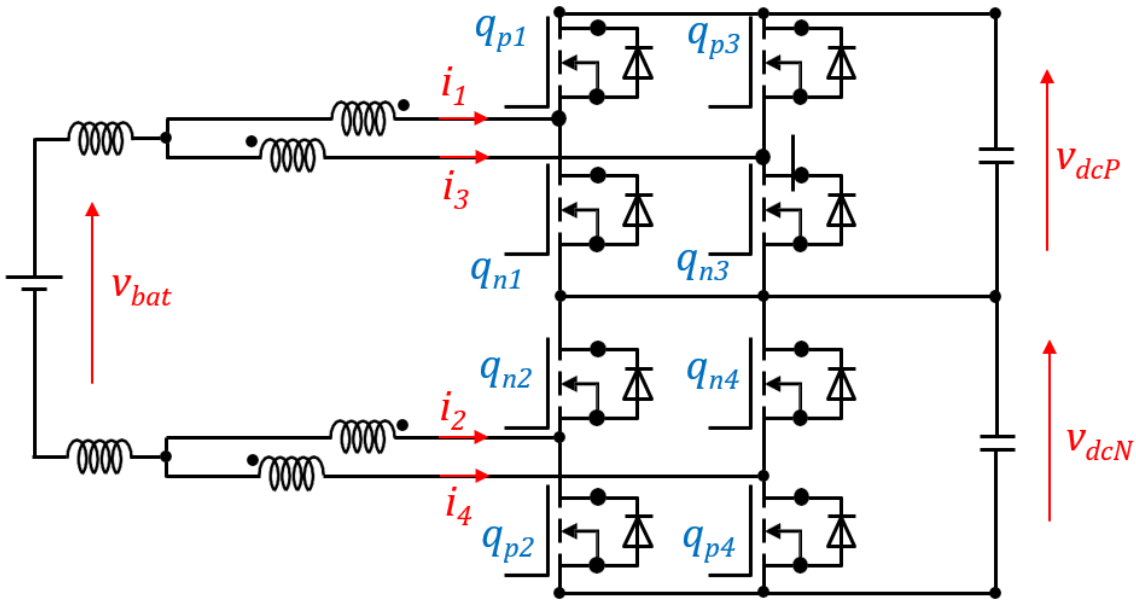


Figure 4.1: DC/DC converter

In Fig. 4.1, the schematic diagram of a DC/DC converter's electrical configuration is depicted. The parameters that require sampling, representing the system's inputs, are highlighted in red, while the controlled quantities are denoted in blue.

4.1.2 Requirements

Let's delve into the time-specific requirements: The selected switching frequency for the MOSFETs is set at 100 kHz.
There are eight switches, four on the upper leg and four on the lower leg, arranged in parallel. This configuration resembles a full bridge due to the central node, resulting in pairs of switches operating together. As a result of the geometry of the system, the ripple current exhibits a frequency four times higher than that of the inductors.

The objective is to formulate a strategy for calculating the mean value of this ripple frequency.

4.2 Communication between DMA & ADC

The DMA is employed to transfer data from an Analog-to-Digital Converter (ADC) to a designated memory location. An ADC is a fundamental electronic component responsible for converting continuous analog signals, typically voltage levels, into discrete digital values.
The specific ADC module used here is the Enhanced Versatile ADC, which boasts 8 independent analog-to-digital converters (EVADC groups). Each of these converters is capable of handling up to 16 analog input channels and provides digital output with a maximum resolution of 12 bits.
Following the completion of each analog-to-digital conversion, an interrupt is triggered. This interrupt serves as a signal to initiate the transfer of the converted ADC results to a predefined memory destination. This memory destination can be, for example, the CPU Data Scratch-Pad RAM (DSPR0).
The system supports multiple request sources that can initiate analog-to-digital conversions with various configurations. These configurations encompass options like single or repetitive conversions. Additionally, interrupts can be generated upon completion of these conversions, and their behavior can be tailored as described in 3.3.

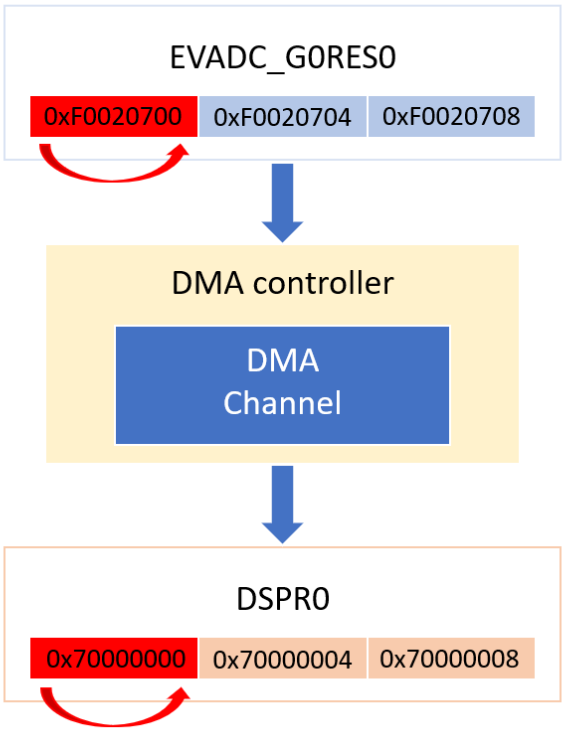


Figure 4.2: DMA-EVADC communication [8]

As can be seen in Fig. 4.2, the result of the conversion is taken from the result register of the EVADC (0xF0020700) and brought to the starting address of the DSPR0 (0x70000000).

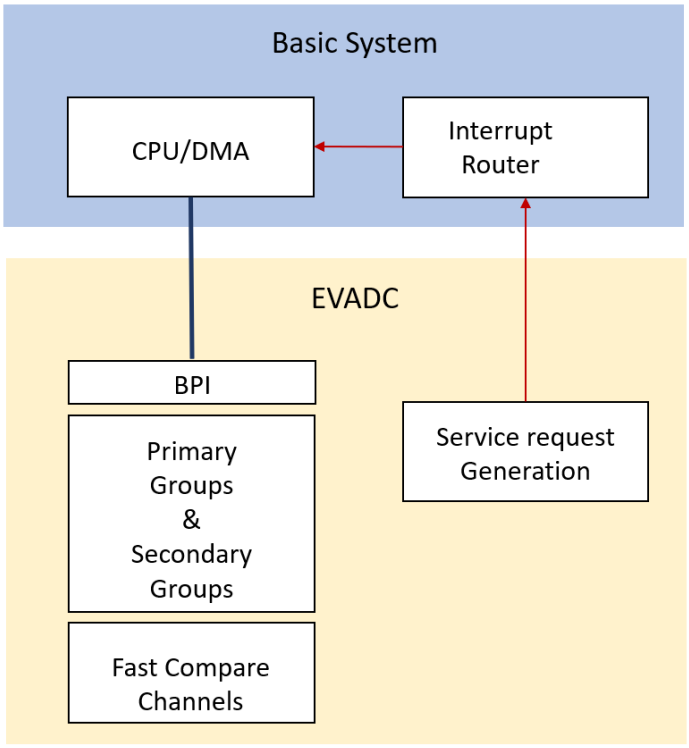


Figure 4.3: DMA-EVADC block diagram

The block diagram in Fig. 4.3 illustrates the inter-module communication framework from an electronic perspective. Specifically, the Interrupt Router takes over handling Service Requests in the ADC and can be commanded either by the CPU or the DMA based on the chosen settings. Additionally, the "Bus Peripheral Interface" (BPI), registers related to the configuration and control of the peripheral interface, are shared to optimize the communication path between the two components.

4.3 Tecniques implemented

In this section the techniques implemented in trying to best accommodate the final application presented in 4.1 will be widely discussed.

4.3.1 Moving average

The first technique taken into consideration was the "moving average". As reported in [9], it "is a time series constructed by taking averages of several sequential values of another time series."
In the example below is reported the "one-sided moving average" of y_t:

$$z_t = \frac{1}{k+1} \sum_{j=0}^{k} y_{t-j}$$

$$t = k+1, k+2, ..., n$$

The term "moving average" is used to describe this procedure because each average is computed by dropping the oldest observation and including the next observation. The averaging "moves" through the time series until z_t is computed at each observation for which all elements of the average are available. Note that in the above examples, the number of data points in each average remains constant.

Implementation

Specifically, it was implemented a "centered moving average" to ensure that the average is centered in the middle of the data values being averaged.
The selection of this method was driven by the need to sample the high-frequency variations in ripple current from the inductors (Fig. 4.4) and calculate their average to determine the mean value. Additionally, the incorporation of a "dynamic" averaging approach was deemed essential to periodically recompute the mean value, thereby accommodating potential fluctuations.

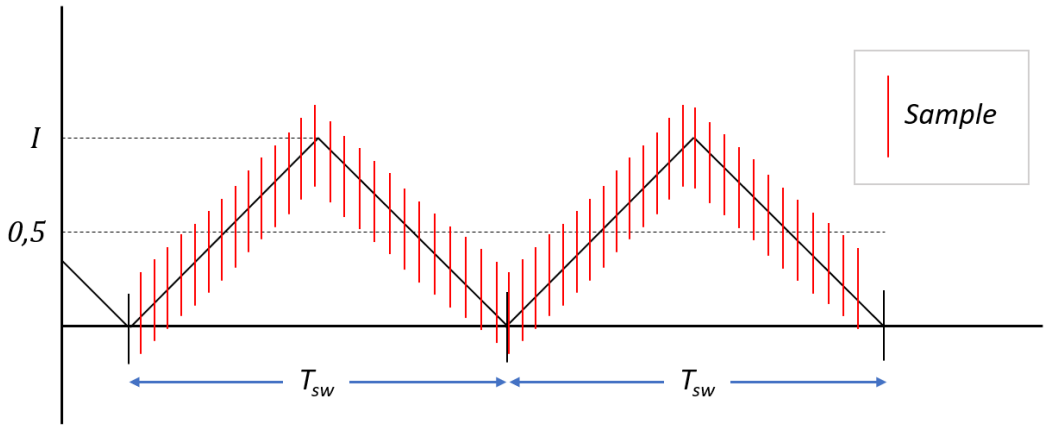


Figure 4.4: Moving Average

The current is sampled through the EVADC module, which triggers an interrupt after each conversion, notifying the DMA module. The DMA module, as outlined in the 4.2 section, transfers the data to the designated memory and arranges it according to the specifications in 5.2. Later on, the information is retrieved from the memory and processed using the subsequent algorithm.

Now, let's delve into the critical aspects in the provided pseudo-code 1: The code demonstrates the ability to reconstruct a triangular wave by effectively detecting changes in the direction of the sampled data. Specifically, when two consecutive changes in direction are identified, it signifies the completion of a full period of the triangular wave analysis. Consequently, by computing the average of the values within this period, the mean value is determined and updated.
Furthermore, in the subsequent period of the triangular wave, this calculated mean value can be compared with the new incoming samples while adhering to a predefined tolerance threshold. This approach ensures the algorithm remains synchronized with the waveform and can accurately identify variations within the specified tolerance limits.

Algorithm 1 Pseudo-code for moving target

```
Initialize variables:
buffer[N]                          ▷ N = number of samples saved in memory
value                                                    ▷ current value
target                                                   ▷ target value
tol                                                      ▷ predefined tolerance
i,k                                                      ▷ counters
flag = 0

for i from 0 to (N − 1) do
    Read "value" from destination memory
    Store "value" in "buffer" at position [i]
    Calculate "error" as ("target" − "value")

    if "err" is between "(−tol)" and "tol" then
        Print "target reached"
    else
        Print "target not reached"
    end if

    if (change of direction condition) then
        Increment flag
    end if

    if flag = 1 then
        Add "buffer" at position [i] to "sum"
        Increment k
    end if

    if flag = 2 then
        Calculate new target as (sum/k)
        Set tol to 5% of "target"
        Print the new target value, the number of samples, and the tolerance.

        Reset "flag" to (−1), "sum" to 0, "k" to 0
    end if
end for
```

```
target not reached 2174
target not reached 2387
target not reached 2634
target not reached 2870
target not reached 3103
target not reached 3197
target not reached 2968
                          new target value: 1820
                          samples: 14
                          tolerance (5 per cent) : 182
target reached 1947
target reached 1763
target not reached 1531
target not reached 1295
target not reached 1066
target not reached 822
target not reached 590
target not reached 344
target not reached 129
target not reached 101
target not reached 329
target not reached 567
target not reached 815
target not reached 1039
target not reached 1268
target not reached 1507
target reached 1727
target reached 1968
target not reached 2197
target not reached 2421
target not reached 2668
target not reached 2895
target not reached 3129
target not reached 3175
target not reached 2949
                          new target value: 1836
                          samples: 14
                          tolerance (5 per cent) : 183
target reached 1953
target reached 1716
target not reached 1479
target not reached 1240
target not reached 1001
```

Figure 4.5: Implementation on the serial monitor of AURIX development studio

The implementation of the strategy in a test condition is depicted in Figure 4.5. In this scenario, a triangular waveform ranging from 0 to 2 Volts, corresponding to a range of 0 to 1600 levels [1], was generated using a wave generator.

Considerations

The described method can accurately determine the mean current of the tested circuit but turned out to be impractical when compared to the requirements and specifications of the board being used. In fact, to ensure the technique's correct

[1] "1" quantization level = 0,00125V

implementation, it's necessary to collect a sufficient number of measurements during each cycle. Consequently, the EVADC (Analog-to-Digital Converter) must operate at a significantly higher frequency. By analyzing and referring the requirements with the provided datasheet ([10]), it becomes evident that the EVADC can require tens to hundreds of microseconds to complete a single conversion, depending on the chosen mode, synchronization, calibration, and post-calibration steps. Considering the frequency of the ripple triangular wave mentioned in section 4.1, it becomes clear that the chosen strategy is not always feasible.

4.3.2 Double sampling

The "double sampling method," as described in the work by Chattopadhyay et al. (2017)[11], involves two key steps: current sampling and phase shift updates. These steps occur at specific points in the triangular carrier wave's cycle: at the zero point and at the peak.

To elaborate, the phase angles are calculated during half of the most recent switching cycle. These calculated phase angles are then updated when the carrier wave reaches its next peak, which is half a switching cycle later.

Similarly, when sampling currents at the peak of the carrier wave, the reference point is set to the next peak, and the phase angles are subsequently updated when the carrier wave reaches its next zero point. In essence, this method synchronizes current sampling and phase angle updates with the carrier wave's specific points to ensure accurate control and measurement.

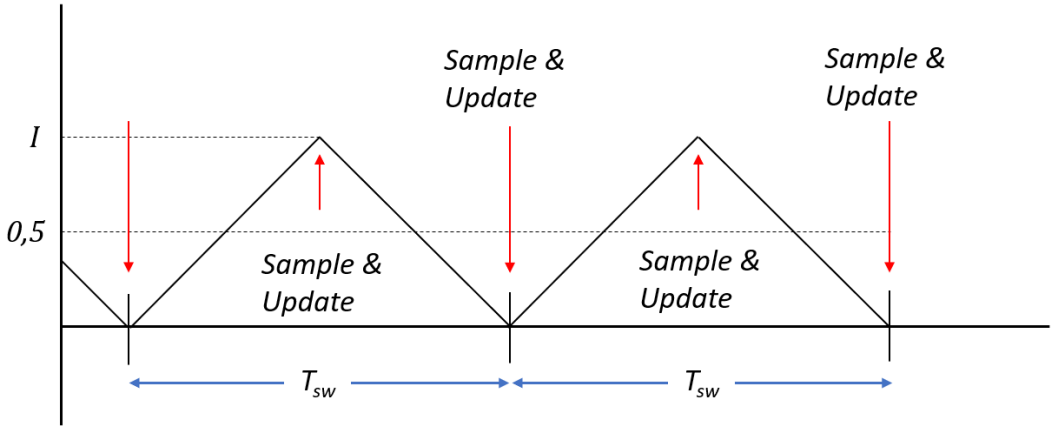


Figure 4.6: Double sampling

Fig. 4.6 represents the sampling timing on a generic triangular wave to mimic the ripple current of the main project. Sampling two times per period assures finding the mean value of the current.

$$\frac{I_{low} + I_{high}}{2} = I_{mean}$$

This result holds true under the condition that both edges, the rising edge (denoted as k_{rise}) and the falling edge (denoted as k_{fall}), have identical slopes. However, when these two edges have different slope coefficients,$k_{rise} \neq k_{fall}$, a more sophisticated technique called "weighted double sampling" is required. This approach takes into account the duty cycle as a weighting factor, as explained in greater detail in the work by Chen et al. (2018)[12].
In simpler terms, when the slopes of the rising and falling edges are not the same, we need to use a modified method that considers the duty cycle of the signal to ensure accurate measurements.

Considerations

The "double sampling" method represents a technique designed to rectify inaccuracies arising from signal misalignment. This approach entails acquiring two samples within each signal cycle and subsequently computing their average. Importantly, this method possesses the theoretical capability to entirely mitigate errors stemming from signal

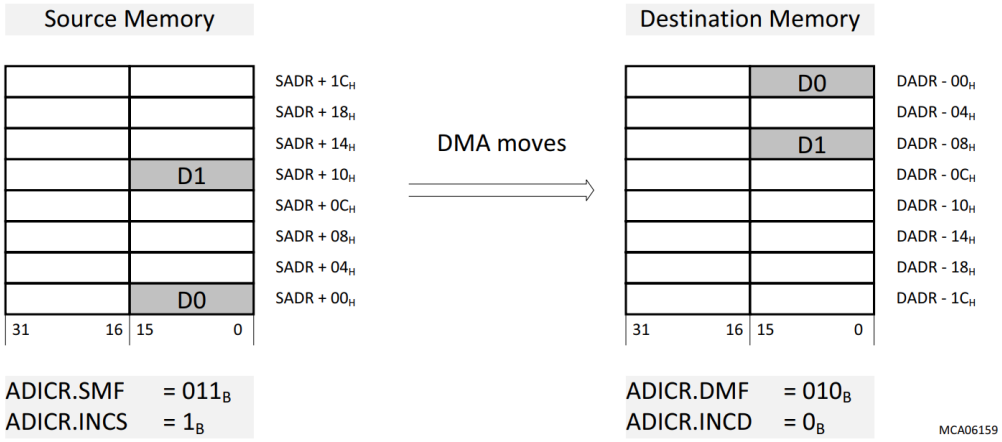
misalignment.
Furthermore, when considering its application within the context of **??** (as previously referenced), it becomes evident that this method seamlessly aligns with the specific frequency requisites of the EVADC module.

4.4 Memory selection and organization

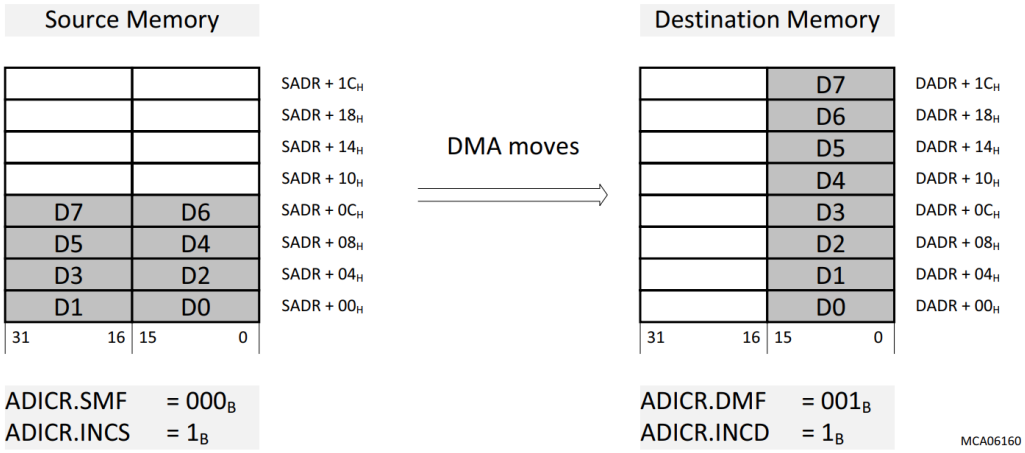
This section will clarify the data organization approach used for storing data acquired through EVADC conversions in memory. The goal is to enhance data retrieval speed within the control strategy application.
Additionally, an analysis and discussion will be conducted on the considerations influencing the choice between the two main target memory options, as outlined in 2.4.1, within the context of the final application.

4.4.1 DMA moves

As briefly mentioned in 2.3, a transaction consists of transfers and moves. During each move, the option to select memory addresses and destination registers is available, enabling the organization of data based on factors like quantity, size, characteristics, and application needs.



(a) *Programmable Address Generation : example 1 [2]*



(b) *Programmable Address Generation : example 2 [2]*

In the figures provided, Fig. 4.7a and Fig. 4.7b, two examples are presented to illustrate the concept. In Fig. 4.7a 16-bit half-words are seen to be transferred from a source memory. The source address offset increments by 10_H, while the data is moved to a destination memory with a decrementing destination address offset of 08_H. Meanwhile, in Fig. 4.7a, another instance demonstrates the transfer of 16-bit half-words. In this case, they originate from a source memory with an incrementing source address offset of 02_H and are directed to a destination memory with an incrementing destination address offset of 04_H.

4.4.2 Circular buffer

The circular buffer configuration needs special care since it is the one implemented in the final application. It is implemented by specifically placing the source address and destination address that are updated within the circular buffer wrap-around limits. Possible buffer sizes of the circular buffer in the Tricore TC399 can be (1, 2, 4, 8, 16, ... up to 64k bytes). Source and destination addresses are incremented or decremented during a DMA move. It's crucial to note that if the circular buffer size is equal to or smaller than the selected address offset, the same circular buffer address will be repeatedly accessed. For example, if the circular buffer has a dimension able to accommodate a single value, such a value will be repeatedly rewritten at each iteration.

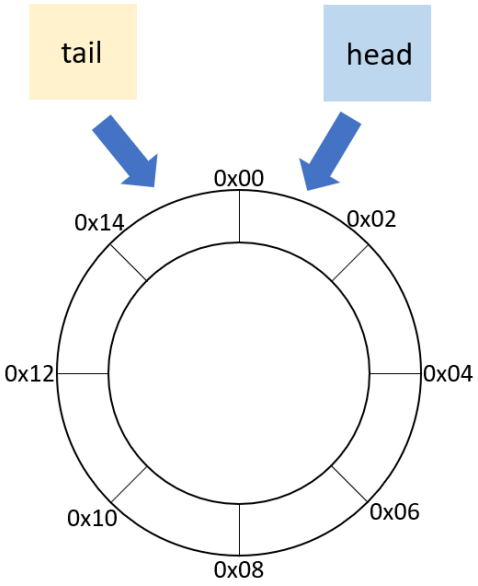


Figure 4.7: Circular Buffer

Fig. 4.7 shows a graphical and conceptual representation of a circular buffer. When the buffer reaches its tail, it starts rewriting its registers starting from the head.

4.4.3 Final disposition

As written in 4.4.2, a circular buffer was selected for the accommodation of data in memory for the final application both for the source and destination addresses.

Reasons

The major reasons that made this strategy preferred among the others possibly fall in the specific requirements the system needs.
Regarding the source address, it was decided to create a buffer designed to hold a single conversion result within a single register. It is important to note that the EVADC (Enhanced Versatile Analog-to-Digital Converter) places its conversion result in its result register, and this result is periodically updated with each new conversion. Therefore, the approach taken involves implementing a circular buffer within a single register. This buffer is configured to point to the address of the EVADC's results register. As a result, during each cycle, the DMA (Direct Memory Access) operation references and retrieves the result from that specific register, which is consistently overwritten with the latest conversion result.
On the other hand, the choice regarding destination addresses was driven by the necessity to organize conversion results in a compact manner to maximize retrieval speed. Furthermore, once the data is utilized, it becomes obsolete. Therefore, employing a circular buffer that continuously updates the registers by overwriting old data with new data is considered the most effective strategy.

Implementation

Fig. 4.8 reports an implementation of the strategy described above.
Specifically, the data to be stored in memory and coming from the EVADC had a dimension of 12-bit, the DMA however can only operate with predefined memory sizes, of which 16-bit is the closer. Thus, 16-bit was necessary to store a single result in memory. Both the LMU and DSPR employ 8-bit registers. Consequently, adjacent registers were paired to store a single conversion result. As mentioned earlier, to ensure efficient data retrieval, it's crucial to organize the data in memory without any wasted space, maintaining contiguous addresses within registers.

0000000009003FFF8	00	00	00	00	00	00	00	00
0000000090040000	E6	02	AB	02	E0	02	F0	02
0000000090040008	DB	02	A9	02	DC	02	E3	02
0000000090040010	F6	02	DF	02	E5	02	EB	02
0000000090040018	C3	02	1B	03	D8	02	EC	02
0000000090040020	07	03	D5	02	0C	03	D9	02
0000000090040028	E3	02	15	03	D6	02	E1	02
0000000090040030	E8	02	EA	02	F2	02	EB	02
0000000090040038	C3	02	E8	02	10	03	D0	02
0000000090040040	00	00	00	00	00	00	00	00
0000000090040048	00	00	00	00	00	00	00	00
0000000090040050	00	00	00	00	00	00	00	00
0000000090040058	00	00	00	00	00	00	00	00
0000000090040060	00	00	00	00	00	00	00	00

Figure 4.8: Memory organization

In particular, the registers utilized span from 0x90040000 (the LMU's starting address) to 0x90040040, as a circular buffer with a size of 64 bytes[2] has been implemented. This configuration allows concurrently storing 32 samples in memory, with each sample occupying two adjacent addresses.

4.4.4 Destination memory selection

Let's delve into the decision-making process between the two viable memory destinations discussed in 2.4.1 - the DSPR (Data Scratch-Pad RAM) and the LMU (Local Memory Unit).
The LMU represents an attractive option when critical factors such as data security, safety, or the imperative for isolated memory regions come into play. Its efficiency in storing data and allocating dedicated memory zones to individual cores helps reduce resource contention. This proves especially advantageous in scenarios where multiple cores simultaneously access a common memory space.
Still, the DSPR offers a different set of advantages, principally centered around faster inter-core communication. This is achieved through its shared memory space, accessible to all cores without necessitating inter-core communication protocols. In addition

[2] 8 bits = 1 byte

to facilitating rapid data exchange between cores, it's noteworthy that DSPR boasts superior intra-core communication as well. In fact, LMU relies on interconnection with the CPU core via Shared Resource Interconnection (SRI), while DSPR can be addressed directly on each core without utilizing the bus.
In summation, the decision between DSPR and LMU shall align with the technical prerequisites of the system.
Given that the final application under consideration is a hard real-time system, the DSPR emerges as the optimal choice, thanks to its capacity for rapid data transfer and superior intra-core communication capabilities to enhance overall system performance making it a perfect choice in the context of a hard real-time system where timing is of of prime importance.

Chapter 5

Experimental results

In this chapter, the implementation of the techniques outlined in the previous chapter 4 will be presented and discussed. Furthermore, the tests conducted and the results of the experiments will be reported.

5.1 Experimental Setup

The experimental hardware configuration employed to investigate the methodologies elucidated in the preceding chapter (4) comprises the following components:

- AURIX™ TriBoard TC399, extensively detailed in chapter (2).
- R&S® RTB2000 Oscilloscope [13], operating in waveform generator mode, boasting a 14-bit resolution, and a 250 Msample/s sample rate.

In particular, the waveform generator was employed to emulate the triangular ripple waveform originating from the inductors of the system presented in the chapter (4). This simulation served as a means to evaluate the performance of the implemented C-code.
The results were assessed using the debugging features of the "AURIX ™ Development Studio [14]," a versatile development environment that includes tools like Eclipse IDE, C-Compiler, Multi-core Debugger, and Infineon low-level driver (iLLD). Importantly, this environment doesn't impose any limitations on time or code size. It made tasks like editing, compiling, and debugging the application code straightforward and efficient.

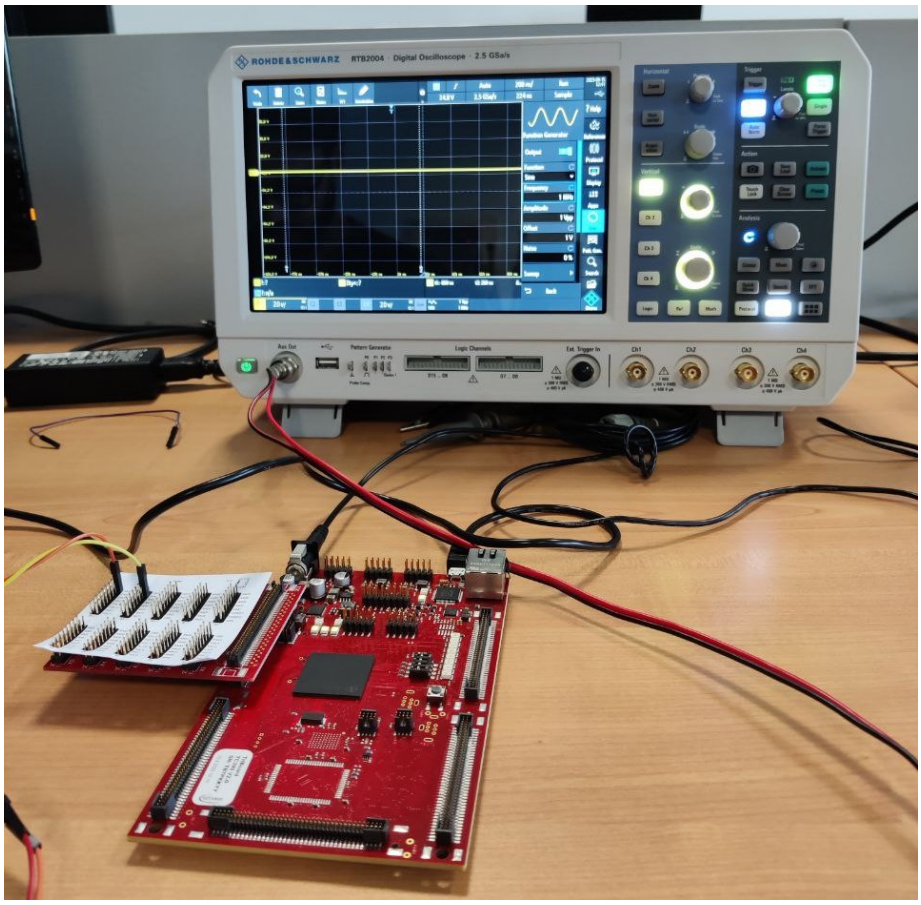


Figure 5.1: Experimental Setup

5.2 DMA settings

This section will detail the DMA settings and configurations applied in the experiments described below in the chapter.

DMA channel initialization

```
IfxDma_Dma_ChannelConfig dmaConfig;
IfxDma_Dma_initChannelConfig(&dmaConfig, &dma);
```

Code 5.1: DMA channel initialization

A DMA channel configuration structure, denoted as '`dmaConfig`' and of type `IfxDma_Dma_ChannelConfig`', is employed for configuring a DMA channel. Such

structure is initialized by invoking the 'IfxDma_Dma_initChannelConfig' function which accepts two parameters: a pointer to the 'dma' object and a pointer to the 'dmaConfig' structure to set default values within the structure.

DMA channel configuration

```
dmaConfig.requestMode = IfxDma_ChannelRequestMode_oneTransferPerRequest;
dmaConfig.moveSize = IfxDma_ChannelMoveSize_xbit;
dmaConfig.channelInterruptPriority = 1;
dmaConfig.channelId = IfxDma_ChannelId_1;
```

Code 5.2: DMA channel configuration

The DMA channel configuration consists of the selection of the following parameters:

- 'requestMode' to specify the modality of the transaction, which can be a single transfer or a complete transaction.
- 'moveSize' to specify the chunks of data to be transferred per request. The available options for this parameter span from 8 to 256 bits.
- 'channelInterruptPriority' and 'channelId' determine respectively the priority level of the DMA channel interrupt and the ID of the DMA channel to be initialized.

Circular buffer

```
dmaConfig.sourceAddressIncrementStep = IfxDma_ChannelIncrementStep_1;
dmaConfig.sourceAddressCircularRange = IfxDma_ChannelIncrementCircular_none;
dmaConfig.sourceCircularBufferEnabled = TRUE;
dmaConfig.destinationAddressIncrementStep = IfxDma_ChannelIncrementStep_x;
dmaConfig.destinationAddressIncrementDirection =
IfxDma_ChannelIncrementDirection_positive;
dmaConfig.destinationAddressCircularRange = IfxDma_ChannelIncrementCircular_x
dmaConfig.destinationCircularBufferEnabled = TRUE;
```

Code 5.3: Circular buffer implementation

The DMA settings for the circular buffer, as outlined in Chapter 4, are detailed below. The parameters source/destinationAddressIncrementStep are utilized to determine the increment of source and destination addresses after each transfer. In the

context of the current application, the source address remains fixed, with no increment. Conversely, the destination address can be incremented by up to 128 bits. The parameter `destinationAddressCircularRange` is employed to specify the size of the circular buffer.

5.3 DMA tests

Since the detailed implementation of the DMA is proprietary information of Infineon, and given the need to find the correct configuration of the module to achieve the best possible results in terms of timing and reliability, an experimental test was necessary.

5.3.1 Transfer numbers

The following pseudo-code is employed to measure transaction speed using DMA on the Infineon TriCore. The program tests different transfer numbers within a single transaction to find the best setup for a particular application. Its main goal is to determine whether it's better to have a higher number of smaller transfers or fewer larger ones in a transaction.

Implementation

The major steps executed in the pseudo-code include:

1. Initialization of the DMA module and the STM (System Timer Module).
2. Definition a transaction of 256 bits of data to be transferred.

```
// Define an array of data to be transferred (256 bits)
uint32 data[8] = {0xC0CAC01A, 0xBA5EBA11, 0xDEADBEEF, 0xC0CAC01A,
                  0xBA5EBA11, 0xDEADBEEF, 0xC0CAC01A, 0xBA5EBA11};
```

3. Execution of 1000 DMA transfers using the specified transfer number through the '`.transferCount`' field adjusting the '`.moveSize`' to allow each time the total data transferred to be the same and stopping the STM timer when each transfer is complete.

```
dmaConfig.transferCount = x; // from 1 to 32
dmaConfig.moveSize = IfxDma_ChannelMoveSize_xbit; // from 256 to 8 bits

// For loop
Ifx_TickTime startTime = IfxStm_get(stm);
for (int j = 0; j < 1000; j++)
{
    IfxDma_Dma_startChannelTransaction(&dmaChn);
    while (IfxDma_Dma_isChannelTransactionPending(&dmaChn));
}
Ifx_TickTime stopTime = IfxStm_get(stm);
```

4. Calculation of the average results to determine the "Transaction time".

```
Ifx_TickTime elapsedTime = stopTime - startTime;
elapsedTime = elapsedTime / 1000;

// Transaction time
float tickDuration = 1.0 / IfxStm_getFrequency(stm); // 100 MHz
float elapsedSeconds_ns = (float)elapsedTime * tickDuration *
1000000000.0;
```

The STM (System Timer Module), as displayed in the pseudo-code, is utilized to measure the elapsed time for each DMA transfer through the '`IfxStm_get(stm)`' function to retrieve the current time from the STM. By invoking it before and after the DMA transfers, it permits precise time measurement by calculating the time difference between the two obtained results: '`elapsedTime = stopTime - startTime`'.

Results and considerations

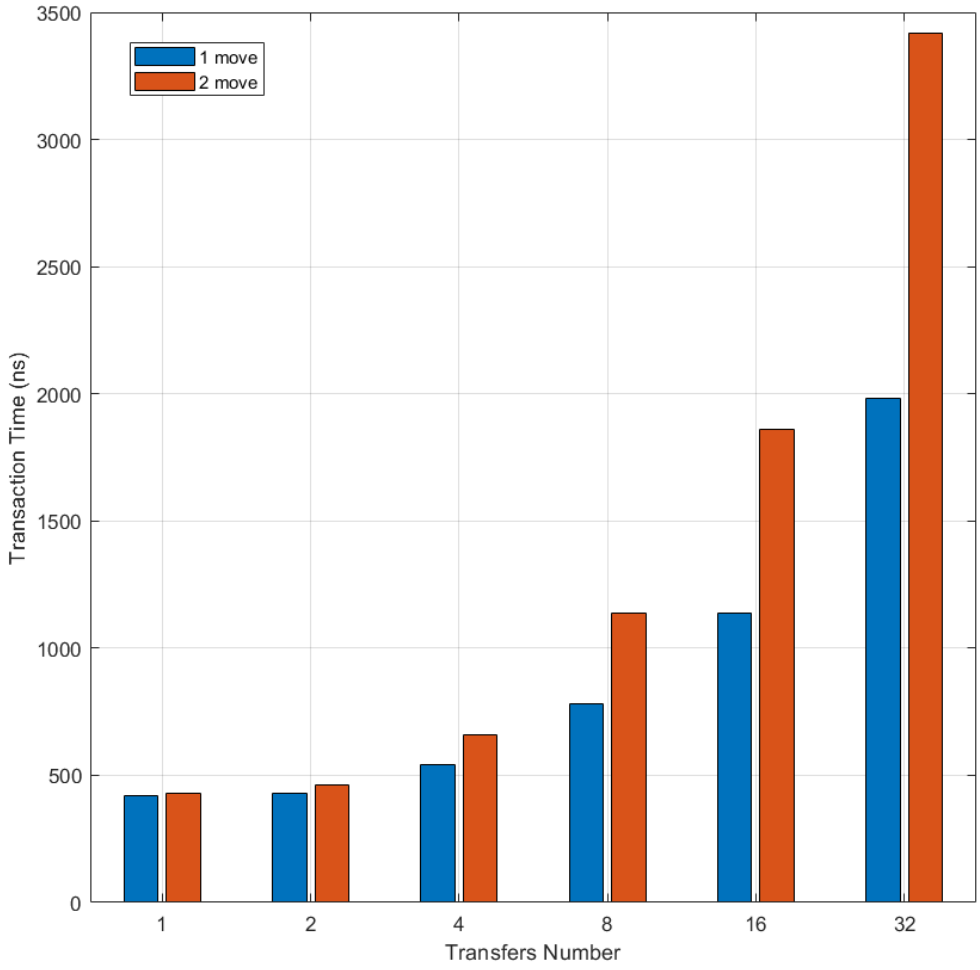


Figure 5.2: Transaction Time vs Number of Transfers

Transfer Number	bits transferred per Move	Time (ns)	
		1 Move	2 Moves
1	256	420	430
2	128	420	460
4	64	540	660
8	32	780	1140
16	16	1140	1860
32	8	1980	3420

Table 5.1: Transaction time vs Number of transfers

The results presented in Tab. 5.1 and Fig. 5.2 originate from internal measurements conducted using the STM module, as previously mentioned. It is important to note that the STM module measures the execution time of the code under analysis, specifically the "for loop," and not the actual transaction time. In this regard, the introduction of a blank code line results in a time increase of 10ns, corresponding to an STM clock stroke.
Also, let's acknowledge that the STM operates at a lower frequency than the DMA, with clock frequencies of 100 MHz and 300 MHz, respectively. Consequently, this disparity contributes to intrinsic measurement uncertainties that can extend to approximately 7ns.
To illustrate these limitations, let's examine the outcomes in time of the first two rows of the table. These findings indicate a deviation from the expected downward trend in execution time as the number of transfers increases. Specifically, the execution time does not continue to decrease in these initial rows. The 420ns threshold represents a practical lower limit beyond which obtaining precise measurements becomes unfeasible.

Conclusions

When dealing with variable transfer numbers, it becomes evident that it is highly advantageous to aim for a single large transfer whenever possible.
In fact, the most significant time savings are achieved by minimizing the total number of transfers, even if it means increasing the size of individual transfers.
In this context, please refer to Tab. 5.1 and 5.2, where it can be observed how reducing

the number of transfers per transaction can enhance the overall transfer time. For instance, when examining the trend of the two moves represented by the orange bars in the graph, which more effectively account for the uncertainties previously described, it is evident that the transaction time decreases from 6300 ns for 32 transfers to 430 ns for a single transfer.

5.3.2 Moves per transfer

Another important aspect to be determined is the optimal number of moves to utilize when dividing a transfer.
As indicated in chapter 2, a transaction is segmented into transfers and moves. This analysis investigates whether larger moves when dividing a transfer result in a faster transaction, thus if the pattern observed in transfers within a transaction is also applicable in the number of moves within a transfer.

Implementation

The approach employed in this section closely resembles the one presented in the previous section. The same modules and calculation procedures were utilized.
However, the primary focus of this section is to examine varying numbers of moves within a fixed number of transfers while ensuring that the total transaction size remains consistent for the sake of comparability. To achieve this, adjustments were made to the parameters '`.moveSize`' and '`.blockMode`' during each measurement. These adjustments ranged from 256 bits and one move to 16 bits and sixteen moves, respectively.
The measurements were repeated with different numbers of transfers to more effectively identify and explicit any potential trends.

```
// number of transfers
dmaConfig.transferCount = x;
// size of the move
dmaConfig.moveSize = IfxDma_ChannelMoveSize_xbit;
// number of moves
dmaConfig.blockMode= IfxDma_ChannelMove_x;
```

Results and considerations

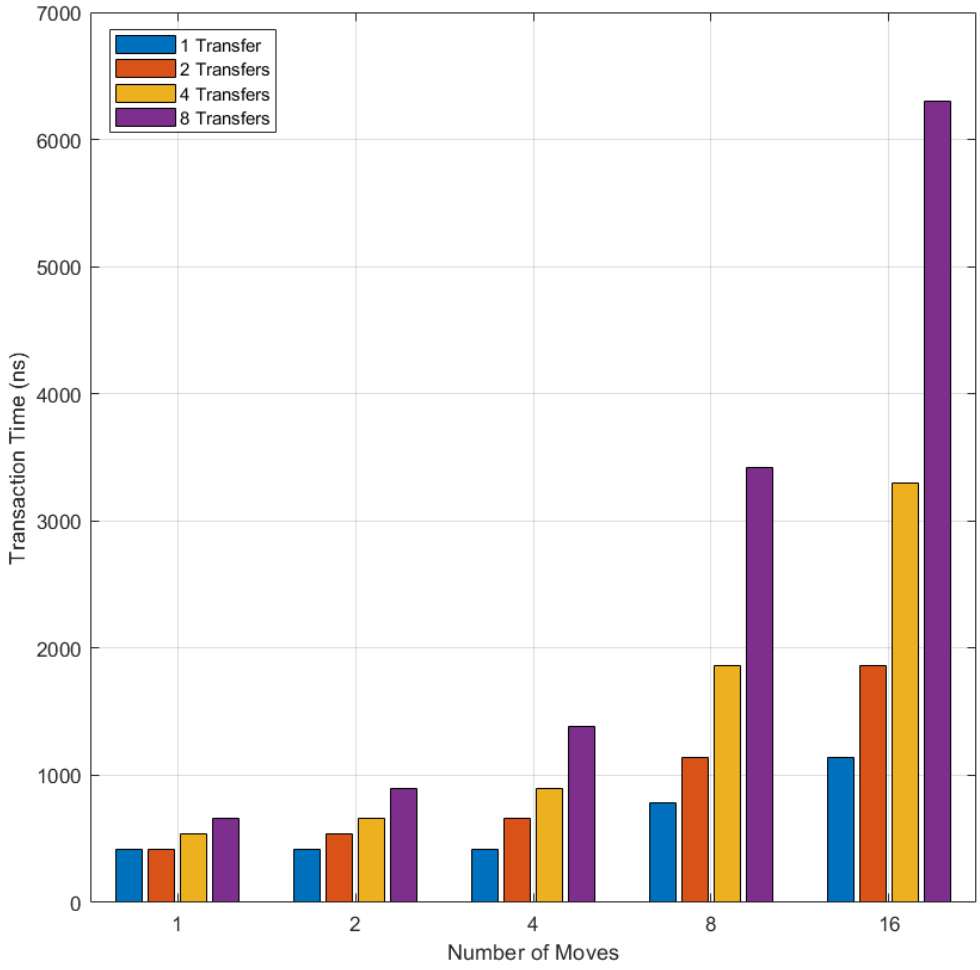


Figure 5.3: Transaction Time vs Number of Moves

Table 5.2: Transaction Times for Different Moves number

Moves	bits transferred per Move	Transfer time (ns)			
		n=1	n=2	n=4	n=8
1	256	420	420	540	660
2	128	420	540	660	900
4	64	420	660	900	1380
8	32	780	1140	1860	3420
16	16	1140	1860	3300	6300

Five different configurations of numbers of moves and move sizes were tested across four different transfer packages, ranging from one to eight. The data were organized systematically to emphasize the time differences arising from variations in the number of moves.
It's important to note that the measurements are subject to the same uncertainties described in the previous section test.

Conclusions

Once again, in order to mitigate the impact of the intrinsic measurement uncertainties, let's focus on the values derived from the configuration with eight transfer moves, represented by the purple bars in Figure 5.3 and detailed in Table 5.2.
The total transaction time presents a declining trend as the number of moves decreases, mirroring the behavior observed with the number of transfers. Specifically, the total transaction time decreases from 6300ns in the last row for the 16-bit sixteen-move configuration to 660ns for the 256-bit one-move configuration.
Additionally, it's worth noting an interesting observation: the time added to perform one additional transaction or move appears to be consistent and identical. This can be observed in the table. For instance, consider the configuration with 2 moves and 2 transfers per 128-bit and the configuration with one move and four transfers per 256-bit; both exhibit the same total transaction time and the same number of steps, namely four each.
In light of these findings, it appears that organizing the total transaction with the fewest possible transfers and moves, thereby transferring the largest possible amount of information in bits in the fewest steps, represents the most effective strategy.

5.4 Configuration for the final application

By summarizing and exploiting all the results and considerations made, it is presented here a pseudo code able to follow the strategy chosen to best meet all the requirements. The following considerations were made:

DMA settings

Given that the minimum required data resolution for the transfer is 12 bits, and the '`.moveSize`' field of the DMA lacks direct support for this specific value, the decision was made to utilize the closest available option of 16 bits. This choice aligns with the principles outlined in the preceding section, emphasizing the optimization of speed through the selection of a single move and a single transfer per transaction.
The primary time requirement to be met is to ensure that the control strategy executes within approximately 10 microseconds (10μs).
Upon a thorough examination of the tables presented in the preceding section, it is evident that the time required for data transfer is nearly negligible. To be precise, transferring a 16-bit value in a single move and transfer operation takes less than 500 nanoseconds (500ns).

Target memory

The memory selected for allocating the data transferred by DMA, in accordance with the considerations outlined in Section 4.4.4, is the DSPR (Data Scratch-Pad RAM). For CPU0, the starting address is designated as 0x70000000.
Given that each register is 8 bits wide, it's important to note that a single move will utilize two consecutive registers to store the data.

Memory organization

The organization of data in the destination memory follows a circular buffer logic. This approach was chosen because it aligns with the requirement of retrieving only the most recent values from memory for use in the control algorithm. In fact, given the limited storage capacity of the DSPR and the need to prioritize the latest data, the circular buffer was the most suitable choice.

By carefully selecting the size of the circular buffer, it becomes possible to determine how much data should reside in memory at any given moment.
Specifically, the source register, which corresponds to the result register of the EVADC where the last conversion result is stored, remains constant across all transfers. This is because the EVADC overwrites it with each new conversion.
Moreover, to accommodate more than one conversion result at a time, the destination register is incremented by 2 widths (16 bits) with each move. The offset in memory between two consecutive moves is set to zero, a deliberate choice to minimize overhead and optimize data retrieval speed.

```
// Channel i
IfxDma_Dma_initChannelConfig(&chn[i], &dma);
chn[i].transferCount = 1;
chn[i].requestMode = IfxDma_ChannelRequestMode_completeTransactionPerRequest;
chn[i].moveSize = IfxDma_ChannelMoveSize_16bit;
chn[i].operationMode = IfxDma_ChannelOperationMode_continuous;
chn[i].hardwareRequestEnabled = TRUE;
chn[i].interruptRaiseThreshold = 0;
chn[i].sourceAddressIncrementStep = IfxDma_ChannelIncrementStep_1;
chn[i].sourceAddressCircularRange = IfxDma_ChannelIncrementCircular_none;
chn[i].sourceCircularBufferEnabled = TRUE;
chn[i].destinationAddressIncrementStep = IfxDma_ChannelIncrementStep_1;
chn[i].destinationAddressIncrementDirection = //positive
chn[i].destinationAddressCircularRange = IfxDma_ChannelIncrementCircular_4;
chn[i].destinationCircularBufferEnabled = TRUE;
chn[i].channelInterruptEnabled = FALSE;
chn[i].channelInterruptTypeOfService = IfxSrc_Tos_dma;
chn[i].channelInterruptPriority = i;

// Channel specific configuration
chn[i].channelId = IfxDma_ChannelId_i;
// result register of the EVADC
chn[i].sourceAddress = MODULE_EVADC.G[y].CHCTR[0].B.RESREG;
chn[i].destinationAddress = DSPR0;

// initialize the channel
IfxDma_Dma_initChannel(&dmaChn[i], &chn[i]);
```

Code 5.4: Final application for a channel 'i'

The pseudo-code provided above outlines the selected configuration for a single DMA 'i' of the DMA.

The DMA was selected as the interrupt service provider, as indicated by the '`.channelInterruptTypeOfService`' field, with the aim of minimizing the CPU's overhead. It is the CPU that bears the responsibility for executing the control strategy.
Observing the size of the destination circular buffer, as indicated by the '`.destinationAddressCircularRange`' field, it becomes evident that, in this particular case, the decision has been made to store data from only the two most recent transactions in memory. In fact, each channel 'i' will have a memory destination between '$0x70000000 + 4i$' and '$0x70000000 + 4(i + 1)$', a space of 4 bytes or 32 bits.
Another decision to make pertains to the '`.requestMode`' setting. If the application demands a high-speed data transfer rate with minimal CPU overhead, then utilizing the DMA to initiate a complete transaction for each request it receives can be the optimal choice, as it can result in faster and more efficient data transfer.
Conversely, if the application necessitates greater control over the data transfer process and requires additional data processing or filtering, then initiating a single transfer for each received request can provide the desired level of control and allow for additional data processing or filtering.
Nevertheless, considering the frequent emphasis on the need for fast and efficient transfers, particularly in the context of hard real-time systems, the '`.completeTransactionPerRequest`' mode was ultimately selected as can be seen in the pseudo-code.

In the complete program, seven DMA channels, labeled from '0' to '6', were configured as described above, in accordance with the input quantities to be sampled as indicated in 4.1.
The '`.channelInterruptPriority`' parameter corresponds to the channel number, with channel '0' having the highest priority and channel '6' having the lowest. However, the DMA channels are triggered by hardware requests, indicated by '`.hardwareRequestEnabled = TRUE`', which are generated by their respective EVADC channels when a conversion is completed and the result is available in the result register. In fact, each DMA channel is specifically associated with a distinct EVADC channel and is responsible for handling only its own request.

Chapter 6

Conclusion

In this thesis, the investigation centered on intra-core communication within hard real-time systems in the automotive control domain.
The primary focus of this study revolved around the examination of data transfer configurations using Direct Memory Access (DMA) and the systematic organization of data within the target memory.

To achieve the objectives of the thesis, the following activities were undertaken:

- A comprehensive study of the memory modules of the TriCore was conducted to determine the most suitable option for accommodating the final application, ensuring fast and reliable memory access, and efficient data retrieval.
- An exhaustive analysis of the DMA module was performed, covering all its functions and settings. Two programs were implemented to assess its working modes, enabling the precise selection of the best configuration in accordance with specific requirements.
- Development of a final application program capable of meeting the requirements of the DC/DC converter and seamlessly integrating with the EVADC module.

The selected memory for data storage was the DSPR (Data Scratch-Pad RAM), chosen for its superior access speed and efficient data transfer compared to the LMU (Local Memory Unit).

Based on the conducted tests, the optimal DMA configurations involved unifying the entire transaction into a single move and transfer operation to maximize data transfer speed.

The final application program established connections between each EVADC channel responsible for sampling input data (Fig. 4.1) and a distinct DMA channel, ensuring efficient transfer and organization of conversion results in the designated target memory.

6.1 Future steps

The future steps of this work will encompass inter-core communications. It is important to reiterate that the system being examined consists of two DC/DC converters, which constitute the fuel cell and the battery, along with an inverter (refer to Fig. 1.1). Consequently, there is a need to establish connections between three boards: two for the DC/DC converters and one for the inverter. The communication architecture will involve a master-slave configuration, where the inverter assumes the role of the master.

In this context, the definition and implementation of a 'communication protocol', along with the establishment of data exchange mechanisms to facilitate seamless interaction between the master and slave boards, will be crucial to the project's successful execution.

Furthermore, the undertaking of a 'data synchronization' mechanism becomes essential. This mechanism will ensure synchronized data transfer and coordination among the boards, thereby guaranteeing the proper functioning of the overall system.

Bibliography

[1] *AURIX ™ TriBoard TC399*. URL: https://www.infineon.com/cms/en/product/promopages/AURIX-microcontroller-boards/AURIX-TC3xx-TriBoards/AURIX-TriBoard-TC399-5V-with-socket/.

[2] *AURIX™ TC3xx User Manual part1*. Infineon, 2021.

[3] C. M. Kirsch T. A. Henzinger B. Horowitz. *Giotto: A Time-Triggered Language for Embedded Programming*. Springer, 2001, pp. 166–184.

[4] Paolo Pazzaglia et al. "Optimizing inter-core communications under the let paradigm using dma engines". In: *IEEE Transactions on Computers* 72.1 (2022), pp. 127–139.

[5] Jorge Martinez, Ignacio Sañudo and Marko Bertogna. "End-to-end latency characterization of task communication models for automotive systems". In: *Real-Time Systems* 56 (2020), pp. 315–347.

[6] *General Purposes Direct Memory Access (GPDMA)*. URL: https://www.infineon.com/dgdl/Infineon-GPDMA-XMC4000-AP32290-AN-v01_00-EN.pdf?fileId=5546d4624e765da5014ed9145c601e95.

[7] Selma Saıdi. "Optimizing dma data transfers for embedded multi-cores". In: *PhD disseratation, univercity of Grenovale* (2012).

[8] *DMA ADC Transfer 1 for KIT AURIX TC375 LK DMA transfer of ADC conversion results*. URL: https://www.infineon.com/dgdl/Infineon-DMA_ADC_Transfer_1_KIT_TC375_LK-Training-v01_00-EN.pdf?fileId=5546d4627883d7e00178a2b187ce386c.

[9] Rob J Hyndman. *Moving Averages*. 2011.

[10] *AURIX™ TC3xx User Manual part2*. Infineon, 2021.

[11] Ritwik Chattopadhyay et al. “One switching cycle current control strategy for triple active bridge phase-shifted DC-DC converter”. In: *2017 IEEE Industry Applications Society Annual Meeting*. IEEE. 2017, pp. 1–8.

[12] Fang Chen. “Weighted double sampling to obtain the average value of triangular current for accurate droop control in DC power distribution systems”. In: *IEEE Transactions on Industrial Electronics* 66.11 (2018), pp. 8733–8740.

[13] *R&S® RTB2000 OSCILLOSCOPE*. URL: `https://www.farnell.com/datasheets/3216077.pdf`.

[14] *AURIX ™ Development Studio*. URL: `https://www.infineon.com/cms/en/product/promopages/aurix-development-studio/`.