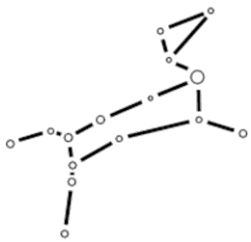
NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF MECHANICAL ENGINEERING
DEPARTMENT OF M.D. & C.S.
Control Systems Laboratory

Diploma Thesis

Design & implementation of a real-time distributed EtherCAT-based motion control system for a multi-DoF quadruped robot

Aristotelis Papatheodorou

Supervisor: E.G. Papadopoulos



ATHENS, 2021

Περίληψη

Η παρούσα εργασία πραγματεύεται τον σχεδιασμό λογισμικού πραγματικού χρόνου για ενσωματωμένα συστήματα διασυνδεδεμένα σε ειδικά σχεδιασμένο δικτύο EtherCAT για τον έλεγχο κίνησης του τετράποδου ρομπότ Laelaps II του Εργαστηρίου Αυτομάτου Ελέγχου της Σχολής Μηχανολόγων Μηχανικών. Η εν λόγω υλοποίηση αποτελεί αποτέλεσμα εκτεταμένης μελέτης που συνδυάζει ποικιλία πεδίων και ειδικοτήτων. Η εργασία βασίζεται σε προηγούμενες υλοποιήσεις των υποσυστημάτων του ρομπότ και εστιάζει στον βέλτιστο επανασχεδιασμό τους και στην προσθήκη νέων στοιχείων με στόχο την μεγιστοποίηση της ευελιξίας του ρομπότ με επιδόσεις συγκρίσιμες ή ανώτερες των σημαντικότερων ρομποτικών συστημάτων σε παγκόσμιο επίπεδο.

Στο δεύτερο κεφάλαιο παρουσιάζεται η αρχιτεκτονική του κατανεμημένου συστήματος επεξεργασίας του Laelaps II, με έμφαση στο τελευταίας γενιάς δίκτυο EtherCAT. Η ευελιξία και η ντετερμινιστική φύση του πρωτοκόλλου, το καθιστούν κατάλληλο για τέτοιες εφαρμογές.

Στο τρίτο κεφάλαιο, παρουσιάζεται ένας νέος αλγόριθμος σχεδιασμού τροχιάς για την κίνηση των ποδιών που αντιμετωπίζει ορισμένες παθογένειες του υπάρχοντος, όσον αφορά βηματισμούς μικρής ταχύτητας όπως το περπάτημα. Ενώ ο υπάρχων αλγόριθμος στόχευε σε δυναμικές συμπεριφορές που απαντώνται σε κινήσεις υψηλών ταχυτήτων, ο νέος αλγόριθμος χρησιμοποιεί τροχιές σταθερής πρόσθιας ταχύτητας και ομαλότερων μεταβάσεων μεταξύ των φάσεων εδάφους και αέρα, επιτυγχάνοντας έτσι ομαλότερη κίνηση σε βηματισμούς χαμηλής ταχύτητας. Ακόμα, μειώθηκε σε μεγάλο βαθμό το υπολογιστικό κόστος του συνολικού λογισμικού ελέγχου του ρομπότ, χρησιμοποιώντας αρχιτεκτονικές παράλληλης επεξεργασίας που αξιοποιούν καλύτερα τις δυνατότητες των ενσωματωμένων υπολογιστικών συστημάτων. Ιδιαίτερη έμφαση δόθηκε στη διασφάλιση της ορθής λειτουργίας του λογισμικού, με την υιοθέτηση αυστηρών προτύπων ανάπτυξης κώδικα, όπως το MISRA C.

Στο τέταρτο κεφάλαιο, παρουσιάζεται ο σχεδιασμός και η υλοποίηση δύο συστημάτων αδρανειακών αισθητήρων που προστίθενται στο δίκτυο EtherCAT για την εκτίμηση της θέσης και του προσανατολισμού του ρομπότ. Αυτό αποτελεί ουσιαστικό βήμα για την επίτευξη των περισσότερων σχημάτων ελέγχου κίνησης που απαντώνται στη βιβλιογραφία. Η ενσωμάτωση των συγκεκριμένων αισθητήρων υποβοηθήθηκε από την ευκολία επεκτασιμότητας που παρουσιάζει ο βασισμένος στο EtherCAT σχεδιασμός του ρομπότ.

Το πέμπτο κεφάλαιο αποτελεί έναν οδηγό για την χρήση και την ορθή λειτουργία του τετραπόδου. Παρέχει ένα σύνολο βημάτων που εγγυώνται την απρόσκοπτη εκτέλεση πειραμάτων, ενώ βοηθά το χρήστη να κάνει ρυθμίσεις ανάλογα με τις ανάγκες του.

Στο έκτο κεφάλαιο, παρουσιάζονται μέθοδοι για την αναγνώριση των σημαντικότερων παραμέτρων του ρομπότ, π.χ. των αδρανειακών παραμέτρων των τμημάτων του. Η ανάγκη αυτή προκύπτει στα πλαίσια της εκτέλεσης αξιόπιστων προσομοιώσεων και του σχεδιασμού ελεγκτών που βασίζονται στις παραμέτρους του συστήματος.

Τέλος, στο έβδομο κεφάλαιο, περιγράφεται ένας αλγόριθμος προσαρμογής της ισχύος των επενεργητών σε πραγματικό χρόνο, ώστε να επιτυγχάνεται μείωση της θερμικής τους καταπόνησης και συνακόλουθη αύξηση του προσδόκιμου ζωής τους. Έτσι, το τετράποδο μπορεί να προσαρμόζεται στις εκάστοτε απαιτήσεις ισχύος και ταυτόχρονα να υλοποιεί βραχυπρόθεσμες δυναμικές συμπεριφορές υψηλής ισχύος.

Abstract

The present thesis focuses on the design and implementation of a distributed EtherCAT-based motion control system for the quadruped robot Laelaps II. The new design is based on prior studies conducted by the Legged Robots Team of the Control Systems Lab in NTUA, and adds new state of the art features to bring the resulting architecture among the most advanced motion control systems worldwide. The final system combines several promising technologies in a distributed EtherCAT-based architecture, aiming at providing the robot with maximum agility. Specifically, the main focus lies on the firmware and electronics aspects of the motion control system.

The second chapter includes a thorough study of the distributed control architecture of Laelaps II, focusing on the key properties of the underlying EtherCAT network. The modularity of the protocol, along with its high-end features, makes it ideal for the desired deterministic, high-frequency network used for the interconnection of all the robot's subsystems.

In the third chapter, a new trajectory planning algorithm is developed to improve the locomotion skills of the quadruped at low speed gaits such as walking. While the previous planner promoted highly dynamic behaviors, such as running, the new planner stands out for its robustness and stability in slow constant speed gaits. Besides, the computational cost of the underlying firmware was significantly reduced by setting up a parallel processing scheme exploiting the respective capabilities of the onboard embedded systems. The designed software complies with MISRA C, one of the industry's leading security and safety standards.

In the fourth chapter, inertial sensors are introduced to the robot's network to assess the location and orientation of the robot, making a key step towards autonomy. The integration of the sensors was fascilitated significantly by the modularity and extendability of the adopted EtherCAT-based architecture.

The fifth chapter includes a guide for the proper operation of the system. It provides a set of steps that guarantee the flawless execution of experiments, while helping the user understand how to modify important settings according to various needs.

The sixth chapter presents a framework designed for the parameter identification of the robot's legs. This step enables the execution of precise simulations and the design of model-based control schemes.

Finally, in the seventh chapter, an online algorithm has been developed for adjusting the power output of Laelaps II actuators, to account for thermal fatigue and thus increase their life expectancy. The final system is capable of adapting to any condition and allowing short-term overloads safely whenever necessary.

Acknowledgements

First and foremost, I wish to thank my supervisor Professor E. Papadopoulos for introducing me to the fascinating world of robotics and guiding me through my initial steps in the field. I feel truly obliged to him, for allowing me to become a member of his laboratory from the early stages of my academic life.

Furthermore, I would like to express my gratitude to the Ph.D. candidates of the CSL-EP lab and good friends of mine Konstantinos Koutsoukis, Athanasios Mastrogeorgiou and especially Konstantinos Machairas not only for their invaluable contribution in the elaboration of this thesis but also for their encouragement, support and knowledge that offered me all these years. Also, I would like to acknowledge the remarkable work of Stamatis Athiniotis, the creator of the initial firmware architecture of Laelaps II, along with the work of John Valvis on Laelaps' hardware design. Moreover, I wish to thank Antonis Aggouridis for creating some much-needed components with the lab's CNC and George Bolanakis for his software support in the initial stages of this thesis.

Since this dissertation is the last part of my undergraduate studies, I would like to express my gratitude to associate Professor Vasilios Spitas for his advice, work ethic and guidance all these years. Moreover, I would like to thank Alexandros Anastasiadis, Nikolaos Kallieros, Konstantinos Athanasopoulos and Georgia Nikolaou for their profound support and help. Last but not least, I owe much to my family for their incessant love and encouragement.

Dedicated to my Family

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Definition
ADC	Analog to Digital Converter
AI	Artificial Intelligence
AL	Application Layer
CCS	Code Composer Studio
CLA	Control Law Accelerator
CNC	Computer Numerical Control
CPU	Central Processing Unit
CRMS	Cumulative Root Mean Square
CS	Chip Select
CSL	Control Systems Lab
CSV	Comma Seperated Values
DAC	Digital to Analog Converter
DC	Distributed Clocks
DCL	Digital Control Library
DLL	Data Link Layer
DMA	Direct Memory Access
DSP	Digital Signal Processor
ECAT	EtherCAT
EEPROM	Electrically Erasable Programmable Read-Only Memory
EM	EtherCAT Master
ENI	EtherCAT Network Information
EoMs	Equations of Motion
ESC	EtherCAT Slave Controller
ESI	EtherCAT Slave Information
ESM	EtherCAT State Machine
ETG	EtherCAT Technology Group
EtherCAT	Ethernet for Control and Automation Technology
FIR	Finite Impulse Response
FMMU	Fieldbus Memory Management Unit
FPU	Floating Point Unit
GA	Genetic Algorithm
GPIO	General Purpose Input/Output
GVL	Global Variable List
HAL	Hardware Abstraction Layer
IIT	Istituto Italiano di Tecnologia
IMU	Inertial Measurement Unit
IPC	Inter-Processor Communications
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LDO	Low Drop-Out
LVDS	Low Voltage Differential Signaling
MCU	Microcontroller Unit

MDP	Modular Device Profile
MEMS	Micro-Electro-Mechanical Systems
MII	Media-Independent Interface
MMU	Memory Management Unit
MOPS	Motor Overheating Protection System
ODE	Ordinary Differential Equation
PDI	Physical Device Interface
PDO	Process Data Object
PHY	Physical Layer
PIE	Peripheral Interrupt Expansion
PLC	Programmable Logic Controller
PWM	Pulse Width Modulation
SF	Safety Factor
SM	State Machine
SNR	Signal to Noise Ratio
SOTA	State Of The Art
SPI	Serial Peripheral Interface
SSC	Slave Stack Code
TCL	Time Control Loop
TMU	Trigonometric Math Unit

1 Introduction

1.1 Motivation

The objective of the present thesis is to enhance the Laelaps II quadruped robot (Figure 1-1) with new capabilities by exploiting optimally its hardware and lay the foundations for advanced control schemes leading ultimately to advanced locomotion skills. Although the previous Laelaps setup was functional, certain improvements were considered necessary to fully exploit its capabilities. The EtherCAT network, currently operating on the robot, has made the development modular, facilitating the fast integration of new features. In this way, any sensory subsystem or other installed hardware is controlled by a microcontroller unit (MCU), which is handled as a slave by the EtherCAT master. Certain aspects regarding locomotion and dynamics are analyzed, along with their implementations, by installing sensors and designing optimally the respective firmware. Towards that, the first step of a thorough literature review is of utmost importance to outline current trends in the field, assess their performance and incorporate them into the system.

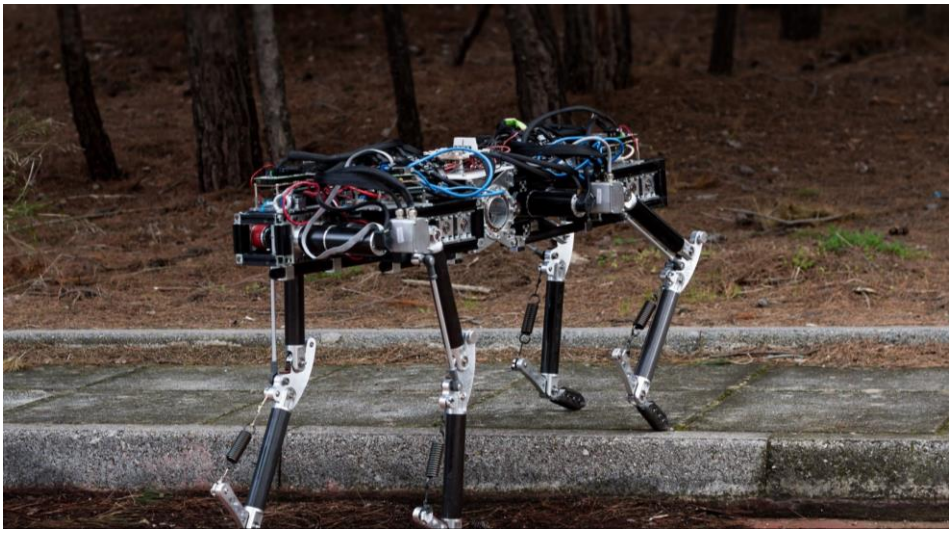


Figure 1-1. The Laelaps quadruped robot.

1.2 Literature Review

1.2.1 Legged Robots

Legged robots constitute a whole new branch of mobile machines that are not bounded by restrictions that the wheeled ones are. The mobility advantage and agility that a leg presents cannot be replicated by a simple wheel layout. This is one of the reasons why legs are the main locomotion mechanism in nature. From the speed of a cheetah and the climbing skills of a goat to the strength that leafcutter ants demonstrate (relative to their size), legs are there to tackle every mobility challenge that nature has to offer (Figure 1-2). For sure animals have obvious differences in how they look and move, but they share the same fundamental locomotion principles. That alone may be a convincing proof that with the right hardware and software, along with the ongoing advancements in engineering and technology, legged machines could soon have a leading role in robotic operations in unstructured environments.

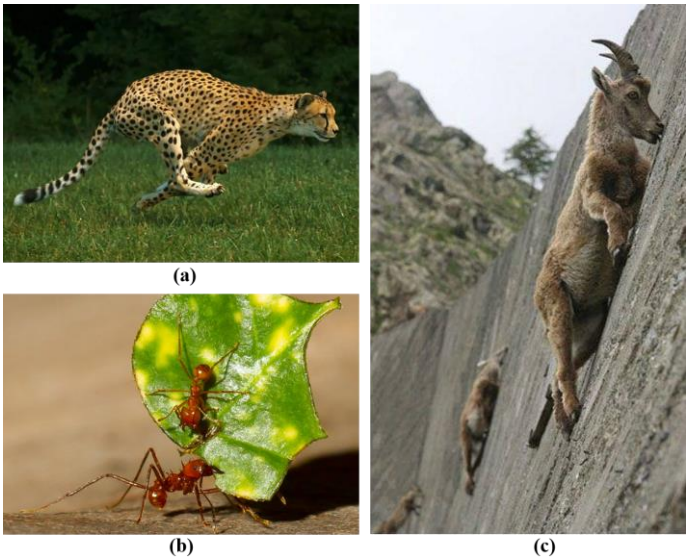


Figure 1-2. Legged animals in various mobility challenges: (a) a cheetah, (b) a leafcutter ant, (c) a goat.

Today, at the dawn of legged robots, many questions remain without a definitive answer. An increasing number of academic institutions and tech companies are creating evolutionary platforms trying to address the key problems. The designs are gradually improving aiming to undertake a diversity of tasks, including every-day ones, space exploration and more. By employing bleeding-edge technologies, the scientific field is rapidly expanding and ready to change everyday life at its core. Boston Dynamics' Atlas [1], Spot [2] and BigDog [3], along with others such as Anybotics' ANYmal and its recent successor ANYmal C [4], MIT Cheetah [5] and Agility's most recent biped DIGIT [6], are some state of the art attempts that demonstrate the current research trends (Figure 1-3). Some of them, like Spot, are mature enough for commercial distribution, with experimental integrations in big supply chains, like the Ford Motor Company's [7]. Finally, Laelaps II (Figure 1-1) is a quadruped robot designed and built by the CSL-EP Legged Robots Team at NTUA [8], which primarily serves as a research platform for addressing the most significant open design and control problems in the field.

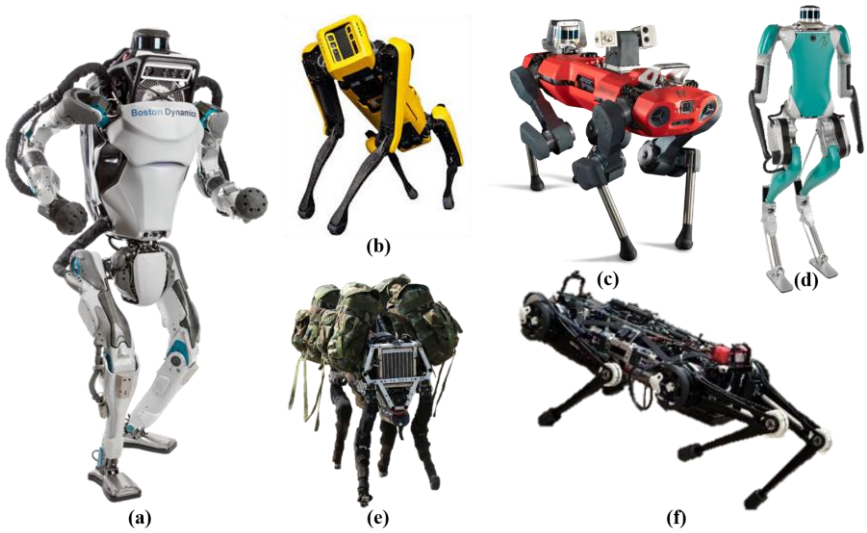


Figure 1-3. (a) Boston Dynamics Atlas. (b) Boston Dynamics Spot. (c) ANYbotics ANYmal-C. (d) Agility Robotics DIGIT. (e) Boston Dynamics BigDog. (f) MIT Cheetah 3.

1.2.2 State Estimation Hardware and Sensors

An important aspect of every robot is its state estimation scheme and the general understanding of its surroundings (localization). Besides the obvious control use of state estimation, recent endeavors focus on leveraging its task-driven counterpart by making robots interact with their environment and execute a task. In most cases such interactions require many states for their description and some of them may be hard to measure directly. This subject has been extensively discussed in the scientific community [9] [10], in an attempt to account for challenges such as the minimization of the processing power needed to accomplish practical tasks. Research suggests that there are many milestones to reach in this field and numerous different aspects to improve. Since localization is a complex and layered engineering field, research focuses on perception, planning and control, but also on the key technologies that can enable the application of the proposed concepts. Importantly, besides the traditional approaches, learning-based approaches have demonstrated the ability to efficiently execute such tasks by providing a fair amount of time for training beforehand.

In this thesis, an attempt is made to equip Laelaps II with sensors commonly used in most state estimation schemes [11], like joint encoders and inertial measurement units (IMUs). For instance, the previous version of Laelaps did not use any IMUs to determine its pose and as a result its body position and orientation was controlled indirectly, in an open-loop manner. The wide applicability of these devices is justified by their simplicity, well-proven technology and the vital information they provide. This work aims to integrate two IMU sensors into the current layout. This effort will enable accurate body motion that was not possible in the past. In general, IMUs come in two configurations, gimbaled and strap-on systems, both with different perks [12]. At this point, the latter configuration was chosen as the simplest one, to avoid complex housing structures with moving parts.

The most common sensors for rotational measurements are the rotary encoders [13]. They are electromechanical devices that translate their rotor's angular position to digital readings. There are two major types with respect to the kind of output needed. The incremental encoders are used when the relative position is sought out. On the other hand, absolute encoders produce a unique reading assigned to every angular position of their rotor. Contrary to the incrementals, they do not lose track of their rotor's position when they are not powered on. Currently, on Laelaps II quadrature incremental encoders are installed. This choice is justified by the low noise and finer resolution that these modules offer. A quadrature encoder is an optical module that combines two sensing channels that present a phase shift [14]. This results in the generation of two shifted square waves and another supplemental one that provide information for position, direction and velocity (Figure 1-4). The downside of this feedback architecture is that the aforementioned modules need manual initialization at the desired zero-angle position.

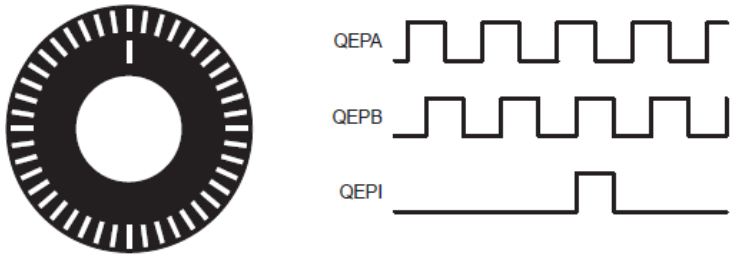


Figure 1-4. Quadrature optical encoder overview.

The optical nature of the quadrature encoders implies certain limitations regarding their nominal operational conditions. In dirty environments with dust and other non-magnetic contaminants, magnetic encoders are more suitable. They are inherently rugged and shock resistant. Generally, Laelaps II is designed to operate in a variety of harsh outdoor conditions. With that said, in the current thesis magnetic absolute encoders are placed on each leg's drivetrain to keep track its position even without power. This way, not only the robot's startup sequence becomes fully automated, but the power transmission's health can be monitored as well.

1.2.3 Motion Planning and Control

The basic Laelaps II motion control architecture consists of a higher and a lower level controller. In this thesis only the latter one is considered. The already implemented control scheme consists of three major parts [15]: (a) a trajectory planning part, generating a task space trajectory in real-time context, (b) an inverse kinematics routine that translates the former task-space trajectory to joint-space angles and (c) a joint-level active compliance controller driving each leg.

The trajectory planner that is currently used by Laelaps II performs semi-elliptical trajectories, designed for efficient trotting. The simplicity of the approach makes it ideal for deployment in the embedded systems that currently operate on the robot. Compared to other designs [16] [17], Laelaps II executes highly dynamic motions with lower processing cost [18]. The main downside of the method is the periodic accelerations that comes with during stance phase. To enable smoother motion regimes like walking, a constant stance velocity planner [19] was implemented in firmware. The simulations and experiments have demonstrated the expected smoother, car-like motion of the robot, without rapid accelerations that disrupt its gait cycles.

The key part of the quadruped's architecture is its active compliance joint controllers that are coupled to its trajectory planning scheme. They are based on the principle of a spring-damper system acting among the current joint-angles and the desired ones. This architecture is materialized on the real robot by optimized PV controllers [20]. The rationale behind this choice over the traditional PD ones is that the former ones do not add any zero terms in the closed loop system, resulting in lower oscillations. In this thesis an effort was made to optimize the whole firmware's execution time and add safety features for flawless operation. These matters are furtherly discussed in Chapter 3.

1.2.4 Motor Overheating

Legged machines are characterized by increased torque demands at their joints, since they have to support their own weight even when they stand still. For electrically actuated robots, increased torque means increased current and thus high motor temperature. To enable dynamic behaviors and exploit the hardware at its limits, the motor heating problem should be accounted for and treated accordingly. Maxon, the manufacturer of the installed motors, proposes methods to predict their thermal behavior as a function of their operation [21]. Nevertheless, none of the methods demonstrates online prediction capabilities and as a result the need to develop one arises. Due to the complexity of the temperature measurements, one requirement would be that the proposed method must rely solely on current measurements to regulate each motor's input current. The complete study is presented in Chapter 7 of this thesis.

1.2.5 System Identification

The distributed control scheme of Laelaps II is currently a model-free and manually tuned control system [15] [22]. However, to reach optimal performance, the application of model-based controllers could be a good solution by taking into account the key nonlinearities that the system presents. Towards this, the exact inertial and frictional parameters must be accurately measured, to have a precise mathematical model for the actual physical system. Besides control-related objectives, knowing the basic system parameters accurately enables accurate simulations, which is a key step in the entire development process.

Although the masses and lengths of each link can be measured with simple methods, the friction coefficients of the joints and gearboxes, along with the inertias and center of mass positions need complex mathematical algorithms involving modeling and dynamics. The two most widely used methods to estimate those parameters are:

- **Extended Kalman Filter**. The inertial and frictional parameters are treated like system states that stay constant throughout time. This fact establishes an online method that linearizes the dynamics at a point and fuses the measurements to get the known and unknown states [23].
- **Least Squares Identification**. The linearity of the mathematical model with respect to the unknown parameters is exploited. By executing least squares regression, the parameters are estimated from torque and kinematic measurements. It is an offline algorithm that requires separate experiments to determine the nonlinear parts (e.g. Stribeck frictional term) of the friction model and the remaining linear ones, like inertias. The method's main disadvantage is that it requires position, velocity, acceleration and torque inputs. Hence, in unknown quantities that cannot be measured and thus are approximated with computational methods, noise becomes a very important factor that downgrades the algorithm's performance significantly [24] [25] [26] [27].

After reviewing both methods' pros and cons, the Least Squares Identification was chosen for the task.

1.2.6 Current Sensing

Accurate measurement of the current consumption of a robot's actuators is important in legged robotics. There are several design options to solve the challenges associated with accurate current-measurement; approaches range from using general-purpose operational amplifiers (op-amps) to analog-to-digital converters (ADCs), either standalone or embedded in an MCU.

In Laelaps' case, current sensing indicates the torque demands of the system. Also, it is a measure of the thermal fatigue of the motors. Most importantly, combined with voltage sensing, it is a direct measure of the power consumption, the minimization of which, is crucial for any mobile machine. In the context of this thesis, an effort has been made to find suitable sensors and integrate them into the existing hardware setup. Related application guides simplify the task and provide powerful insights [28].

2 Laelaps II EtherCAT Network

2.1 Introduction

The Laelaps II quadruped is a 40kg robotic platform designed for highly-dynamic behaviors [29]. It is the result of multidisciplinary studies performed by the CSL-EP Legged Robots Team. Every hardware component is designed and calculated with a variety of methods, both analytical and optimization-based. The majority of the system's components have been manufactured in house, with state of the art equipment and processes. The present thesis aims to improve some of the most critical subsystems of the robot and also add several new features. Specifically, this work includes significant improvements in the sensory system and the control firmware of the robot.

The EtherCAT (Ethernet for Control and Automation Technology) network of the robot plays a vital role, since it enables synchronization and ultra-high bandwidth communications among its various distributed processing units. Designed by Beckhoff, EtherCAT has become the leading trend in the industry's hard real-time data transfers [30]. Its integration in quadruped robots constantly increases; for instance it is used in the well-known IIT's HyQReal [31]. However, along with the great advantages of this technology comes a significant amount of low-level programming effort to make a system like the Laelaps II operational.

In this section, some EtherCAT theoretical preliminaries are presented. Moreover, an overview of the Laelaps II EtherCAT network is presented, both from hardware as well as from software aspects.

2.2 EtherCAT Protocol Overview

EtherCAT constitutes an Ethernet-based Fieldbus communication system. It is suitable for soft and also for hard real-time applications. The combination of short update periods, reduced jitter and low cost, has made EtherCAT the leading protocol in both academia and industry. From the control of a simple servo motor to the demanding application of controlling the mirrors of the Giant Magellan Telescope [32], the extent of use of the protocol becomes apparent.

2.2.1 Functional Principle

EtherCAT uses a master-slave architecture. The slave nodes are considered as black boxes and specifications about their internal architecture and operation are irrelevant as far as the master is concerned. The standard Ethernet frame is not processed at every node sequentially, rather only a specific address of the telegram is read and written by the corresponding slave. This "on the fly" processing approach prioritizes the critical data transfers, while at the same time eliminates redundant delays in the data circulation.

The master node is in complete control of the traffic exchanged in an EtherCAT network. It initializes every communication sequence by sending frames via its Ethernet interface. Subsequently, they are processed "on the fly" by every EtherCAT Slave Controller (ESC) in the logical ring. It should be noted that the term "logical ring" does not refer to the daisy-chain topology of the slaves, rather than to the internal architecture of the network. The slaves are connected in series, but the Ethernet cable along with each slave's internal architecture enables the full-duplex communication scheme, illustrated in Figure 2-1(a).

The frame processing at each of the ESCs is executed by logical units, called ports. Every port executes a part of the total frame processing and forwards the frame to the next one, in a roundabout fashion, like the one presented in Figure 2-1(b).

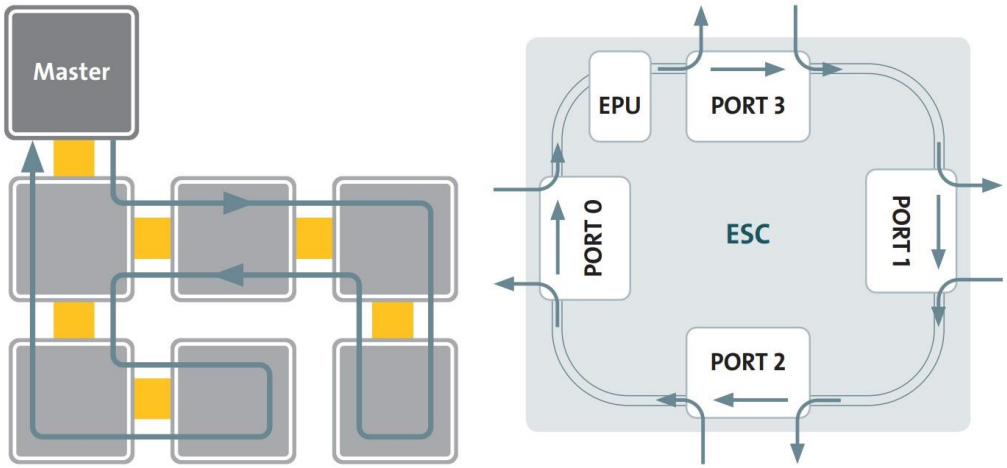


Figure 2-1. (a) The logical ring of an EtherCAT network. (b) ESC frame processing.

2.2.2 EtherCAT Frame Structure

The EtherCAT frame comes as a component of the standard Ethernet frame. Every frame contains a significant amount of fields, each one performing a different functionality or carrying a fraction of the total payload. The EtherCAT frames contain multiple datagrams that adress different slaves in the network. Each datagram contains headers that specify commands and specific addresses, for read/write operations, to particular ESC's memory sections. The feature that enables the high-bandwidth data circulation is that each slave processes only the datagram that the master dictates, designated by the prescribed values in the datagram's header. This mechanism becomes apparent in Figure 2-2. The frame's structure is shown in Figure 2-3, while detailed information can be found in the literature [33].

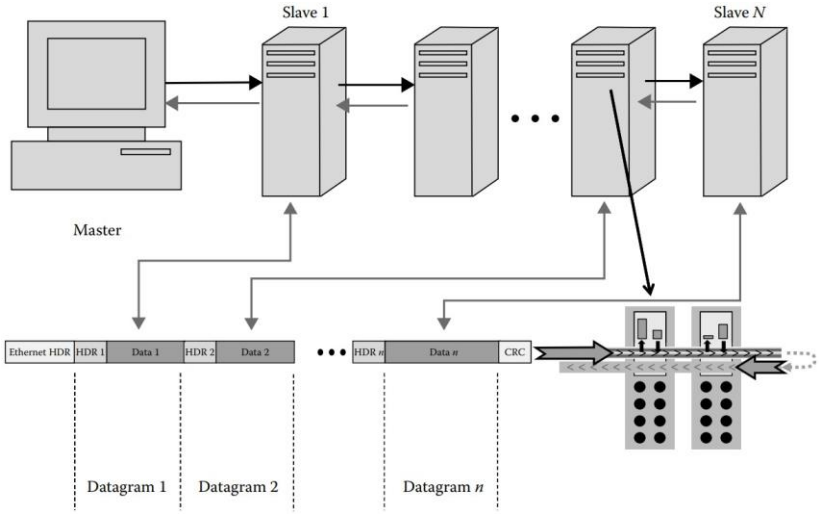


Figure 2-2. Each slave processes only one prescribed datagram.

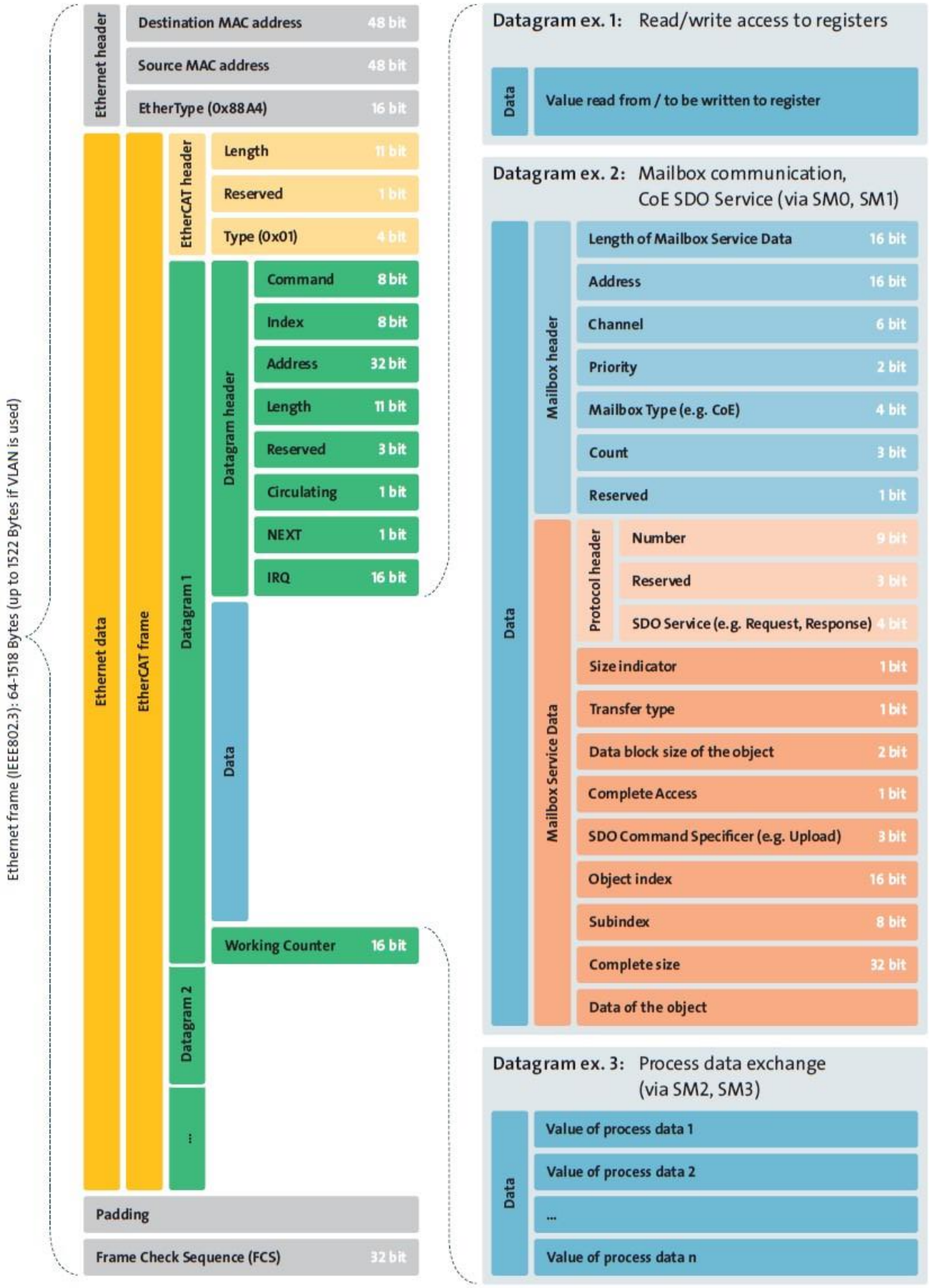


Figure 2-3. Typical EtherCAT frame structure.

2.2.3 EtherCAT Master

EtherCAT Masters (EMs) manage the total amount of communications in an EtherCAT network. In a nutshell, EMs may be materialized by software in Linux-like operating systems, without any special demands as far as the hardware is concerned. This is their main difference with the slaves' architectures that demand tailored made electronics, often FPGA solutions, to fulfill their tasks. In their simplest form, EMs implement a real-time control loop that handles the communications and the intermediate processing required among the cycles. An intuitive overview is given in Figure 2-4. For an in-depth understanding of EM concepts, the reader is referred to [34].

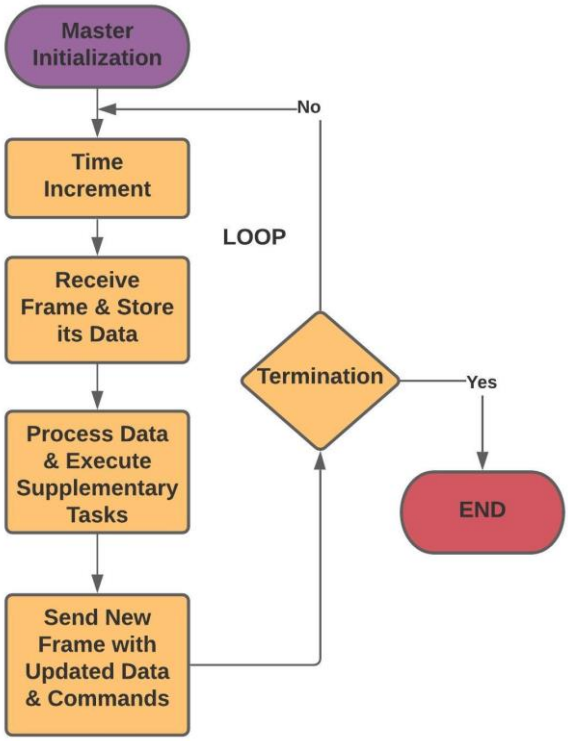


Figure 2-4. Basic EtherCAT master operation.

2.2.4 EtherCAT Slave Overview

EtherCAT slaves present a number of hardware and firmware design challenges that require specialized solutions. As mentioned before, ESCs process each frame "on the fly", while the requirements of such processes demand the employment of hardware solutions to fulfill the low run-time demands. This constitutes the so-called Data Link Layer (DLL) [33], which in turn comprises several modules such as SyncManagers (SMs), Fieldbus Memory Management Units (FMMUs) and Distributed Clocks (DCs).

The second layer of a generic slave is the Application Layer (AL) [33], usually realized by a microcontroller. This layer includes the implementation of the user application. The interface between the two aforementioned layers is possible due to the Physical Device Interface (PDI), which constitutes a communication protocol such as the Serial Peripheral Interface (SPI). These concepts will be clarified in the next sections of this chapter, while an overview may be observed in Figure 2-5.

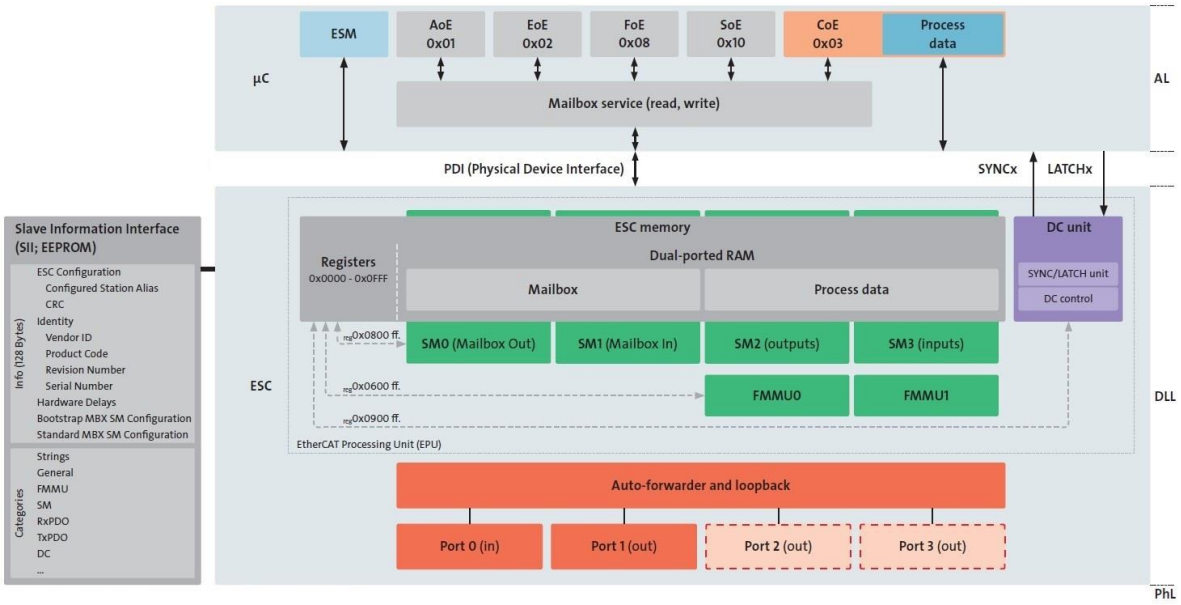


Figure 2-5. ESC architecture overview.

SyncManager

The ESC memory is used for exchanging data between the EM and the application running on the slave. The master can access the memory through the network by using the data link layer services, whereas the local application makes use of the process data interface (PDI) provided by the ESC. As a consequence, problems may arise if concurrent accesses are carried out without any restriction. In particular, the consistency of data is not guaranteed by the basic data link communication services, unless a mechanism, like semaphores, is implemented in software for dealing with data exchanges in a coordinated way. Moreover, both the EM and the application running in the slave have to poll the memory explicitly, in order to determine when it is no longer used by the competing entity.

SyncManagers support two communication modes:

1. **Buffered mode.** In this case, the interaction between the producer and the consumer of data is uncorrelated and each entity can access the buffer at any time. The consumer is always provided with the newest data. In the case data are written into the buffer faster than they are read out, old data are simply discarded. The buffered mode is typically used for cyclic process data. This mechanism is also known as 3-buffer mode, because the SyncManager manages three buffers of identical size (denoted as 0, 1, and 2). One buffer is allocated to the producer (for writing), another buffer to the consumer (for reading), and a third buffer helps as intermediate storage. Reading or writing the last byte of the buffer results in an automatic buffer exchange. It is worth noting that both the EM and the local application must always refer to buffer 0 when accessing memory. It is up to the SyncManager redirecting accesses to the right buffer.
2. **Mailbox mode.** In this case, a handshake mechanism is implemented for data exchanges, which prevents buffer overwriting and ensures that no data will be lost. Just one buffer is allocated for each mailbox; moreover, reading and writing are enabled alternatively. The mechanism implemented by mailboxes is straightforward. At first the producer writes to the mailbox buffer. When done, the SyncManager locks it for writing and enables read access to the consumer. Only when the consumer has

finished reading data out of the buffer, the producer is granted write access again. At the same time, the mailbox turns to the locked state for the consumer. The mailbox mode is typically used for application layer (AL) protocols, where the time taken to exchange information is not very relevant.

According to the literature [35], the buffered mode is used for cyclic data exchanges, while mailbox is usually used for application layer's protocols, such as CAN over EtherCAT (CoE). Therefore, the user-defined process data (Section 2.2.5), which are considered as cyclic data, are exchanged in buffered mode.

Fieldbus Memory Management Unit

The Fieldbus Memory Management Unit (FMMU) is a common concept in the field of embedded systems and computer science. These modules are responsible for the mapping between the ESC's logical address space to its physical counterpart.

Distributed Clock Unit and Synchronization

EtherCAT is designed to serve hard real-time applications with large data transfers. To achieve that, a synchronization mechanism with *ns* time resolution is used. The operation of this mechanism requires the calculation of the various time-drifts and delays that are present in the network. These are summarized in three categories:

1. **Propagation Delay.** The master calculates the time that the frame needs to arrive at each slave in the network, by employing a timestamp mechanism. The receive times and delays acquired by that process are stored in registers *0x0900* and *0x0928* of the ESC, respectively.
2. **Offset.** Each network has a reference clock, most frequently the free-running counter of the first slave. Each node has its own clock and subsequent offsets among them are present. The master, by reading the clocks of every slave in the bus, calculates the required offsets and stores them both in its memory and at each slave controller, more specifically in the *0x0920* ESC register.
3. **Drift.** The reference clocks have a drift in their operation as time passes. To account for such a phenomenon, a time control loop (TCL) is implemented by the master and the drifts are compensated.

The DC Synchronous mode is used in this work, since it is the most hard real-time configuration that EtherCAT supports. This is achieved by activating the major processes of a slave, by a hardware interrupt. The implemented mechanism can be better understood by consulting Figure 2-6. A major ESI element that enables this configuration is called *Dc:AssignActivate* and is configured during the design phase of the slave's EtherCAT stack. It is also, highly dependent on the ESC's hardware architecture.

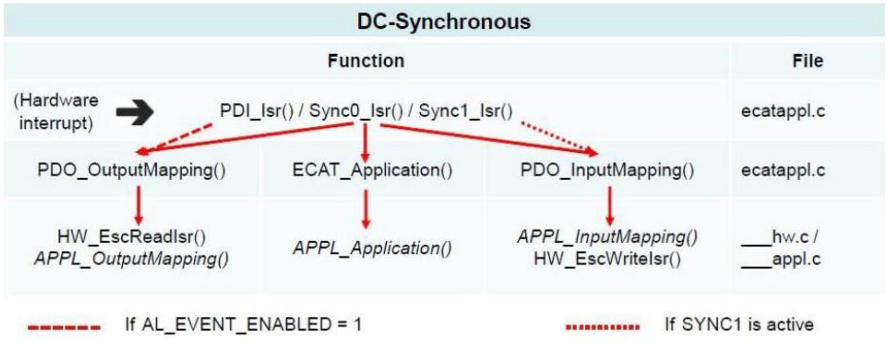


Figure 2-6. DC Synchronous mode overview.

2.2.5 Process Data

EtherCAT networks have a very flexible process data scheme that enables large data transfers without violating other ESC's mechanisms. Each slave's internal organization can be perceived as multiple Process Data Object (PDO) instances, called object dictionaries. These modules have a strict addressing organization based on the CANopen standard [36]. The object dictionary structure and corresponding behavior of the entries is defined by the Modular Device Profile (MDP) [37]. Figure 2-7 provides a better insight into the MDP concept. It defines a modeling of structures within a device. The intention is to provide an easy way for the master to handle the network's devices.

MDP Device				
Object Dictionary	**Module 0**	**Module 1**	**...**	**Module n**
Communication area (0x1000 – 0x1FFF) e.g. object 0x1000, 0x1018, 0x10F3				
RxPDOs (0x1600 – 0x17FF)	0x1600	0x1601	...	0x16nn
TxPDOs (0x1A00 – 0x1BFF)	0x1A00	0x1A01	...	0x1Ann
Manufacturer specific area (0x2000 – 0x5FFF)				
Input area (0x6000 – 0x6FFF) Tx-mappable, read-only	0x6000 – 0x600F	0x6010 – 0x601F	...	0x6nn0 – 0x6nnF
Output area (0x7000 – 0x7FFF) Rx-mappable, read-writeable	0x7000 – 0x700F	0x7010 – 0x701F	...	0x7nn0 – 0x7nnF
Configuration area (0x8000 – 0x8FFF) read-writeable, usually not mappable	0x8000 – 0x800F	0x8010 – 0x801F	...	0x8nn0 – 0x8nnF
Information area (0x9000 – 0x9FFF) read-only, usually not mappable	0x9000 – 0x900F	0x9010 – 0x901F	...	0x9nn0 – 0x9nnF
Diagnosis area (0xA000 – 0xAFFF)	0xA000 – 0xA00F	0xA010 – 0xA01F	...	0xAnn0 – 0xAnnF
Device area (0xF000 – 0xFFFF) e.g. object 0xF000, 0xF010, 0xF030, 0xF050				

Figure 2-7. Modular Device Profile overview.

Most of the procedure of creating such modules is automated, due to the user-friendly Slave Stack Code Tool (SSC) [38], provided by the EtherCAT Technology Group (ETG) [39]. The reader is advised to understand those concepts by referring to the available documentation [37]. More specifically, the memory sections *0x6000-0x6FFF* and *0x7000-0x7FFF* are defined manually because they contain the application's process data definitions. The mapping of the described modules to the ESC's hardware, along with the suitable SyncManagers that are responsible for all the data transactions, is described in the slave's ESI file and may be viewed in Figure 2-8.

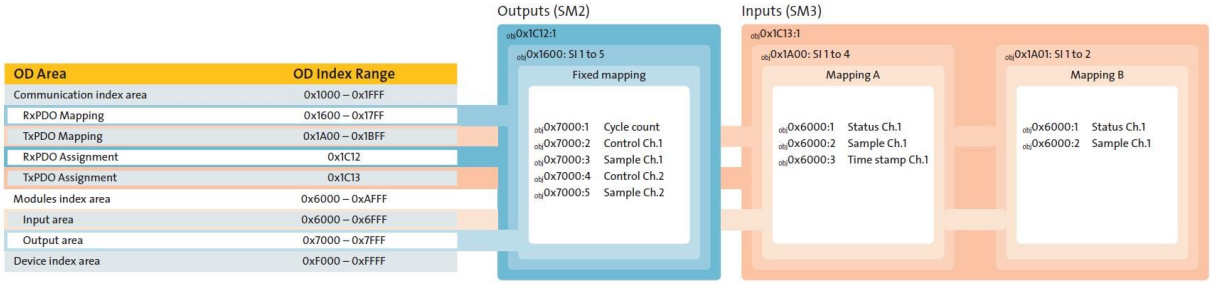


Figure 2-8. PDO mapping overview.

2.2.6 EtherCAT Network Operation

The present section gives a description of the operation of an EtherCAT network and discusses the details of several basic notions.

When the power switch is flipped on, both the master and the slaves in the bus initialize their intrinsic hardware and firmware. The master scans the bus and identifies every node in it. Before the network becomes operational, several diagnostics and synchronizations are performed. Delays and time offsets are calculated as well.

Each slave implements the so-called EtherCAT State Machine (ESM) [33], which has four possible states and one supplementary.

- **INIT:** Only access to ESC registers.
- **PRE-OPERATIONAL:** Mailbox communication available.
- **SAFE-OPERATIONAL:** Process Data available, outputs still in safe-state.
- **OPERATIONAL:** Input and Output Process Data available.
- **BOOTSTRAP**: Supplementary state for writing the ESC's EEPROM memory.

The master requests state transitions from every slave on the bus. The latter ones may verify and execute that transition on a prescribed timeout period, or else an error is thrown in the corresponding AL Status register. By viewing that register, every slave's status can be monitored and determine if the whole bus is operational. The ESM is presented in Figure 2-9, graphically.

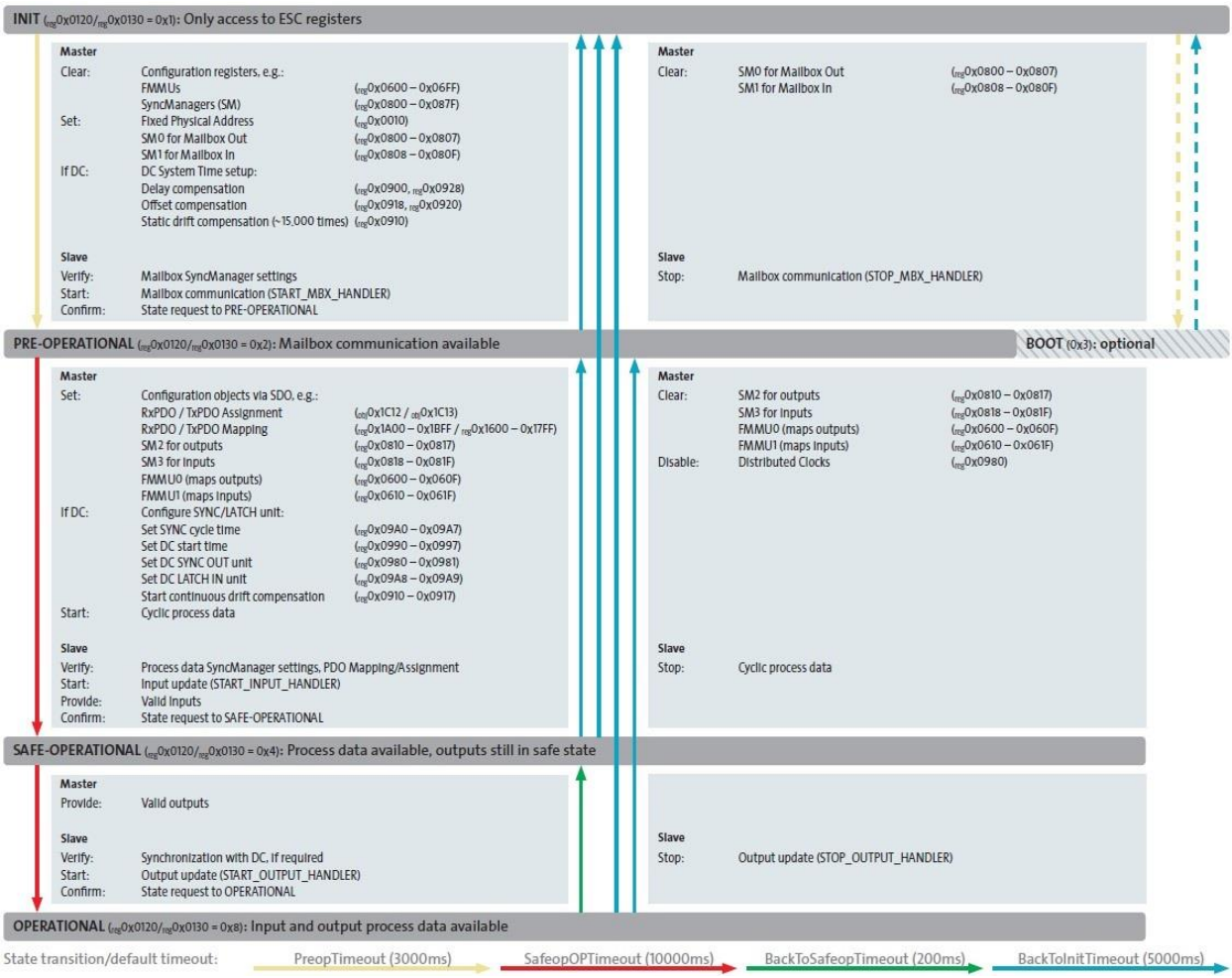


Figure 2-9. EtherCAT Slave's State Machine (ESM).

2.3 Laelaps II EtherCAT Network Description

The above-described concepts are materialized on Laelaps II quadruped. The robot's network inherits the protocol's modularity and flexibility, along with its hard real-time capabilities. Control applications in such complex machines need highly deterministic implementations that only state of the art technologies can provide. In this section, all of the different components of the network are analyzed. In later chapters, the actual hardware and firmware implementations of those concepts will be discussed to develop a complete picture.

The Laelaps II EtherCAT network consists of one master PC, namely an Intel NUC8i7HVK (Hades Canyon Edition) and several LaunchXLF28379D - Beckhoff FB1111-0141 ESC combo slaves. The described network, along with its components, is presented in Figure 2-10.

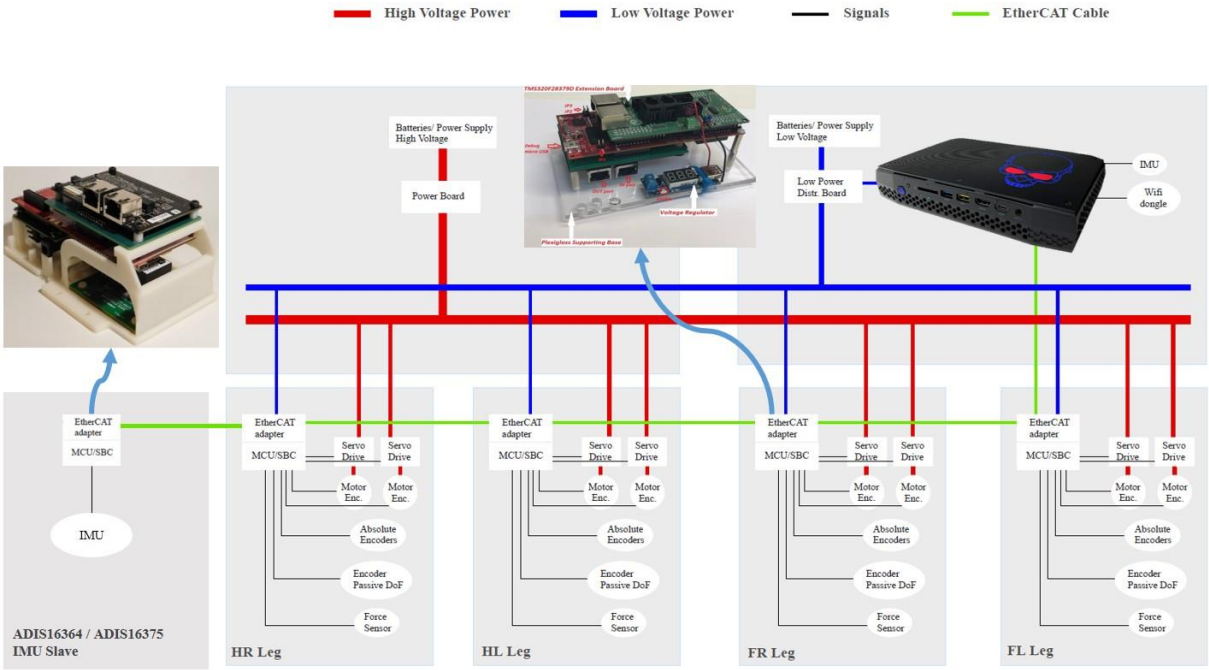


Figure 2-10. Laelaps II architecture overview.

2.3.1 Laelaps II Master

The network's master is implemented with two software packages; one closed industrial-grade solution called TwinCAT 3 [40], provided by Beckhoff, and an open-source solution called EtherLAB [41]. For the latter, the reader is encouraged to refer to the **ether_ros** package [34]. This middle-ware is actively maintained by the CSL-EP Legged Robots Team, on its way to become a complete ROS(2) package with a generic and modular design. On the other hand, TwinCAT 3 constitutes a closed, proven solution with a user-friendly interface and is used extensively in this thesis.

2.3.2 Laelaps II Slave Nodes

Currently, two types of slaves operate on Laelaps II, which both consist of a LaunchXL-F28379D [42] implementing the slave's Application Layer, while the Data Link Layer is realized by a Beckhoff's FB1111-0141 piggyback controller [43]. The slave's intrinsic communications are executed via SPI, which materializes the Physical Device Interface.

The first slave type is responsible for the leg motion control algorithm and the circulation of the related parameters. The control application and the EtherCAT functionalities are running

on the same core, at present. However, the particular microcontroller has a dual-core architecture that could be exploited in the future.

The other EtherCAT slave type handles the communications with an IMU. The work described herein incorporates two IMUs and thus two EtherCAT nodes into the system. The application is not identical for the two sensors, but their core operation remains the same.

2.3.3 Network Characteristics

The bus currently operates at a **2.5 kHz** loop frequency, a rather high-bandwidth for motion control applications. The network is robust and highly deterministic, since no frame drop occurs in a variety of operational conditions. The default cycle time, in theory, can be calculated by taking the sum of the different processes that occur in each cycle, presented in Figure 2-11.

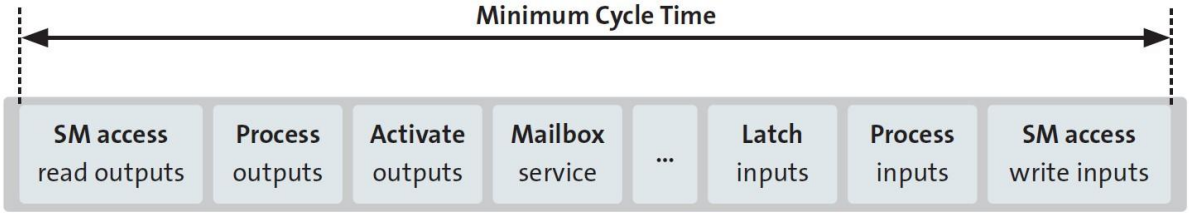


Figure 2-11. Theoretical determination of the minimum cycle time.

In practice, there is no definitive answer regarding the minimum cycle time of an arbitrary EtherCAT network, since its speed is heavily dependent on the attributes of its components. The main contributions in the end-result are the time needed for sending the frame, the propagation delay for its circulation in the network and finally the time necessary for receiving it. In turn these procedures depend on the below parameters.

Bandwidth

Until recently, the protocol's bandwidth was 100 Mbit/s, but with the introduction of EtherCAT G and G10 flavors, this parameter became 1 Gbit/s and 10 Gbit/s, respectively. In a 100 Mbit/s network, like the one currently operating on Laelaps II, the frame processing will take 80 nanoseconds per byte, whereas in a 1 Gbit/s network, it will only take 8 ns per byte and in 10 Gbit/s networks just 800 ps per byte. However, by simply transitioning from EtherCAT to EtherCAT G or G10 will not automatically translate to 10x or 100x performance gain, since as can be realized, the end result depends on many different parameters.

Master Software Processing Time

The processing time needed by the master to perform the required tasks depends on its processing power, hardware architecture, memory performance, etc. In Linux-like EMs certain tuning can be made to optimize their performance. Also, with routines like *ftrace* [44] this amount of time can be estimated accurately. TwinCAT 3 provides an estimation of the required processing time (**11.92 μs**) illustrated in Figure 2-12. To clarify, this number corresponds to a single leg motion control slave (Chapter 3) connected on the bus, as suggested in [20].

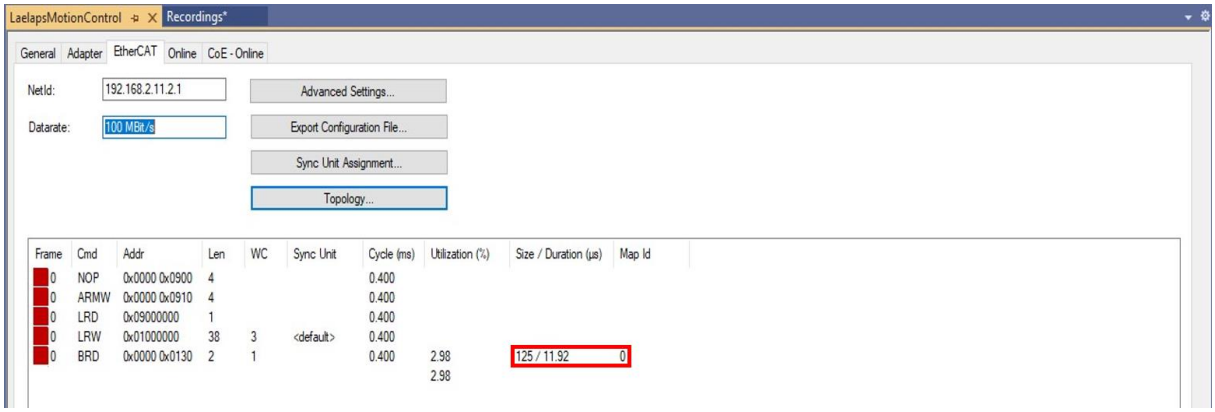


Figure 2-12. TwinCAT 3 frame processing time.

EtherCAT Slaves

The number of EtherCAT slave devices in the network, along with their Ethernet interface contributes to the total cycle time. The typical delay for an EtherCAT slave that uses standard Ethernet MII/PHY is about 1 μs, whereas for another one that uses Low Voltage Differential Signaling (LVDS), like E-bus, the delay is only about 0.3 us. In the Laelaps II case, the ESCs have Ethernet MII/PHY ports, resulting in **1 μs** delay for each slave on the bus.

Another thing to consider is that in DC mode, all of the functions of the Application Layer are synchronized with hardware interrupts. To maintain such tight synchronization each slave should be able to execute all of the required routines within a specific timeframe, dictated by the cycle-time. In the case of Laelaps II motion control firmware, this time is about **169.63 μs**. This value comes as the sum of the execution time of the three major slave stack functions, namely ***Sync0_Isr()***, ***Sync1_Isr()*** and ***PDI_Isr()*** (see Section 3.5). More information about these routines can be found in the literature [45].

Process Data Size

The amount of process data handled by the network has a significant impact on its performance. Note that their size is not limited to that of a maximum Ethernet packet (~1500 bytes). However, above this limit, multiple cyclic frames will be sent to handle the additional bytes. This results in additional overhead and delays. Currently, there are four leg motion control slaves with 60 bytes (22 inputs and 38 outputs) each, one ADIS16364 IMU slave with 40 bytes (36 inputs and 4 outputs) and an ADIS16375 slave with 58 bytes (52 inputs and 6 outputs). Hence the total user-defined process data size is 338 bytes. Apparently, in the current configuration no additional Ethernet packets are necessary even after the addition of the default bytes that come with each frame. Thus, no additional overhead is added. Lastly, the number of fixed bytes in the frame depends on the synchronization mode and EtherCAT's addressing.

Laelaps II Network Cycle Time

From the above analysis, it becomes apparent that the main factor to be considered in defining the minimum cycle time in Laelaps II case is the slave processing time. TwinCAT 3, the EM used in this thesis, prevents the slaves from going to operational mode if the chosen cycle time does not suffice and multiple frame-drops occur. To determine a safe value for the network's cycle, a trial and error methodology was adopted. Initially, a single slave was connected to the EM and by progressively increasing the cycle time, the minimum one was determined at **200 μs**. In the same manner, with all of the slaves connected, the default

operational frequency for the network was chosen at **2.5 kHz** (**400 μs**). Admittedly, the proposed method does not give the fastest possible EtherCAT cycle that the Laelaps II network can achieve, but it is safe to say that the system is robust and deterministic, since no frame drops are observed during its operation.

All of the real-world applications come with hard to model behaviors and noise. To account for such phenomena, like jitter, the master calculates the required “shift times” for each of the slaves. These values, if chosen correctly, can lead the slaves to maximum intrinsic synchronization without frame drops and other corruptions. Admittedly there is an entire interval of values that those shift times may take, while the optimal ones are hard to calculate and depend heavily on the network’s hardware. The results of setting them incorrectly are illustrated in Figure 2-13.

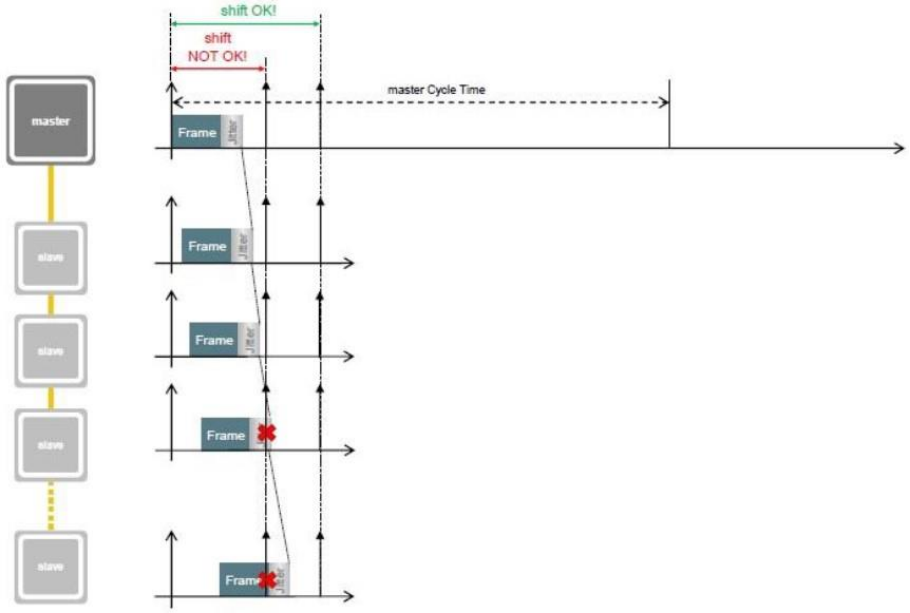


Figure 2-13. Results of correct/ incorrect shift times.

However, there is a rule of thumb for a bulk approximation of the outputs’ shift time as the algebraic sum of the following contributions.

- Hardware Delay introduced internally, by slaves.
  - 1 μs for every slave of the network with MII Ports (Current Configuration).
  - 0.3 μs for every slave of the network with only EBUS Ports.
- Hardware Delay introduced by the cables, which is approximately 5.3 ns for every meter of CAT III Ethernet copper cables.

As previously mentioned those shifts are calculated by the master and in the case of TwinCAT 3, a brilliant job has been made, ensuring automated synchronization for relatively low cycle times. In case of Laelaps II, by adopting a trial and error methodology the initial guess given by the above rule of thumb was corrected and determined to be 30 μs. Admittedly, this is not an optimal value, but it gave very promising results during the test runs since no frame drops and other desynchronizations were observed. Last but not least, there are extensive guides in the corresponding appendices (Appendix A and Appendix B) that illustrate the process of creating a slave’s application and setting up TwinCAT 3 to host those implementations. Significant, but on-going work, is done in the direction of ether_ros master, but requires extensive reference that is out of the scope of this thesis.

3 Laelaps II Motion Control

This chapter is dedicated to the Laelaps II motion control scheme, analyzing both its hardware and firmware aspects. The already operational decentralized architecture [20] resulted in a working prototype with robust gait cycles and great stability. Despite the remarkable behavior of the planner used, the intermittent accelerations that it comes with, downgrade its performance. In the present thesis, an improved constant stance velocity planner, which was introduced by the Team in [19], is implemented and incorporated in the Laelaps II control architecture. Special attention has been given to improve further the performance of the application by taking advantage of the hardware's various capabilities.

At first, the theoretical background concerning kinematics and planning is briefly given. Then, the implementation of the developed planner is described, focusing on the modifications that the setup requires. Details on how to assemble the used hardware components and setup the CCS project are given in Section 3.4. Certain adjustments in the existing EtherCAT stack are made, followed by others in the EtherCAT slave's ENI file, to make TwinCAT's master compatible with the new application. Last, the validation process is described and conclusions are derived.

3.1 Laelaps II Control Architecture

3.1.1 Low-level Control Overview

Laelaps II is a quadruped robot with a distributed, EtherCAT-based control architecture. This means that for each leg, a single microcontroller is in charge for the total control payload. Figure 3-1 gives an overview of the control architecture of Laelaps II, according to the available literature [15] [20]. This thesis analyzes only the low-level controller of the aforementioned architecture (highlighted in the blue box).

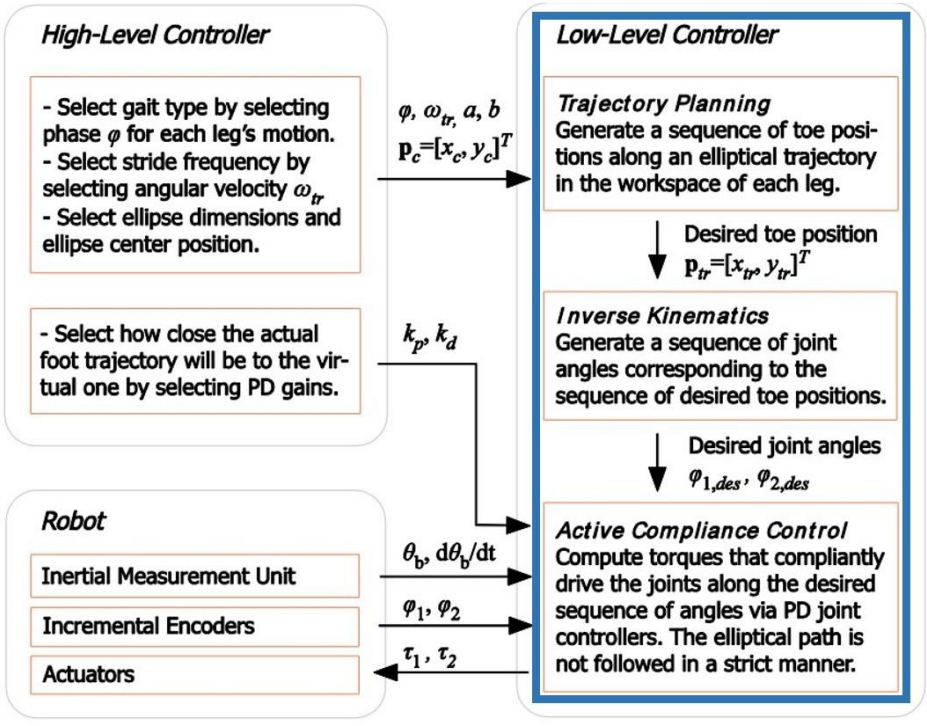


Figure 3-1. Laelaps II motion control architecture [15].

The LaunchXL-F28379D evaluation board (Appendix C) materializes the Data Link Layer (DLL) of the EtherCAT slave model, introduced in Section 2.2.4. It is a high-end, bare-metal system built for deterministic, hard real-time control applications. The underlying firmware architecture consists of purely control-related implementations, along with others that materialize the EtherCAT slave functionalities. The latter ones are necessary since each of the control modules operates in a distributed network that controls the whole robot's operation.

The low-level control design of each leg can be divided in three separate parts: (a) the trajectory planning module that in the current implementation generates two joint-space angle sequences (one for each joint) corresponding to a task-space, semi-elliptical motion of the leg's toe, (b) the low-level active compliance controllers that drive the two actuators of the leg, and (c) the feedback loop that is materialized by high-resolution quadrature encoders with velocity estimation capabilities. All of the discussed concepts are analyzed in this chapter thoroughly, with a special focus on their implementations.

3.1.2 Laelaps II Legs

The Laelaps II legs use the parallel mechanism shown in Figure 3-2. The hip and knee joints, are actuated by two Maxon motors along with gearboxes and timing belts (both with 26/48 reduction ratio). For the hip, a powerful *Maxon EC45 250W* (Part No. 136209) brushless motor [46] is installed, along with its GP52C planetary gearhead (Part No. 223089, with reduction ratio 8/343) [47]. The knees share the same family of gearboxes (Part No. 223090, with reduction ratio 12/637) with the hips, but they are actuated by *Maxon RE50 200W* (Part No. 370356) brushed motors [48]. Figure 3-2 illustrates the leg's design.

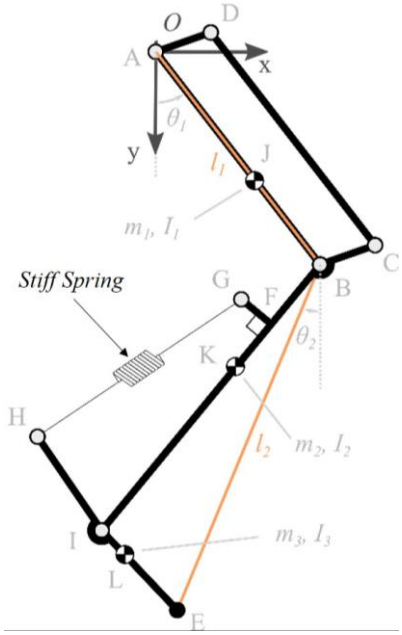


Figure 3-2. Laelaps II leg geometry.

Forward Kinematics

The forward kinematics for a leg are given in (3-1) and (3-2), according to Figure 3-2; (x_E, y_E) denote the toe's coordinates, while the spring *GH* is stiff enough to approximate the leg with two virtual segments of constant length, namely l_1, l_2. Therefore, the spring is neglected in this study. Moreover, the virtual segment angles are denoted by θ_1 and θ_2 .

$$x_E = l_1 \sin(\theta_1) + l_2 \sin(\theta_2) \tag{3-1}$$

$$y_E = l_1 \cos(\theta_1) + l_2 \cos(\theta_2) \tag{3-2}$$

Inverse Kinematics

In the same manner, the inverse kinematics (3-3)-(3-7) are derived below. Let:

$$\varphi = \theta_2 - \theta_1 \tag{3-3}$$

According to the generalized Pythagorean Theorem:

$$x_E^2 + y_E^2 = l_1^2 + l_2^2 - 2l_1 l_2 \cos(\pi - \varphi) \Rightarrow x_E^2 + y_E^2 - \left(l_1^2 + l_2^2\right) = 2l_1 l_2 \cos(\varphi) \Rightarrow$$
$$\cos(\varphi) = \frac{x_E^2 + y_E^2 - \left(l_1^2 + l_2^2\right)}{2l_1 l_2},\ \sin(\varphi) = -\sqrt{1 - \cos^2(\varphi)} \tag{3-4}$$

Note that the negative *sine* solution corresponds to the leg's forward knee configuration. Using the results of (3-4), the φ angle is calculated in (3-5).

$$\varphi = \arctan 2(\sin(\varphi), \cos(\varphi)) \tag{3-5}$$

Finally,

$$\theta_2 = \frac{\pi}{2} - \arctan 2(y_E, x_E) + \arctan 2(l_1 \sin(\varphi), l_2 + l_1 \cos(\varphi)) \tag{3-6}$$

$$\theta_1 = \theta_2 - \varphi \tag{3-7}$$

3.2 Trajectory Planning

3.2.1 Planner Description

A planner recently introduced in [19] is implemented here to allow the real robot to move with constant stance velocity, resulting in smoother gaits. This is a vital step towards the integration and testing of various motion modes, like trotting that was originally investigated in [49]. Each leg follows a trajectory which includes separate formulations for swing and stance phases.

The planner dictates the appropriate formula, by measuring the time progression of each step in comparison with the step's total period. By using the modulo operation, the whole process becomes independent of the step number. Moreover, a time phase shift is introduced, to enable different modes of motion (e.g. trotting, walking, etc.). Let T_{step} be the total duration of a step, Δt_{phase} its phase shift, while T_{swing} and T_{stance} the periods of the different trajectory phases. To make the trajectory invariant to the step's number, t_{traj} is introduced (3-8). Finally, the variable dir_{leg} is used to change the leg's motion direction and can be 0 or 1, for forward or backward motion, respectively.

$$t_{traj} = \mathrm{mod}(t + \Delta t_{phase}, T_{step}), \text{ with } T_{step} = T_{swing} + T_{stance} \tag{3-8}$$

Swing Phase

To avoid impacts with the ground, the trajectory's angle θ_{traj}, given in (3-9), is introduced. More specifically, its derivative $\dot{\theta}_{traj}$ in (3-10), which is the swing phase's angular velocity, becomes zero at the beginning and at the end of the swing phase (Figure 3-3 for

$T_{swing} = 0.5$ [s]). So, the leg takes-off and touches-down smoothly, avoiding unecessary impacts that could stress its drivetrain.

$$\theta_{traj} = \frac{\pi}{2}\left(\cos\left(\frac{\pi t_{traj}}{T_{swing}}\right)+1\right) \tag{3-9}$$

$$\dot{\theta}_{traj} = -\frac{\pi^2}{2T_{swing}}\cdot\sin\left(\frac{\pi t_{traj}}{T_{swing}}\right) \tag{3-10}$$

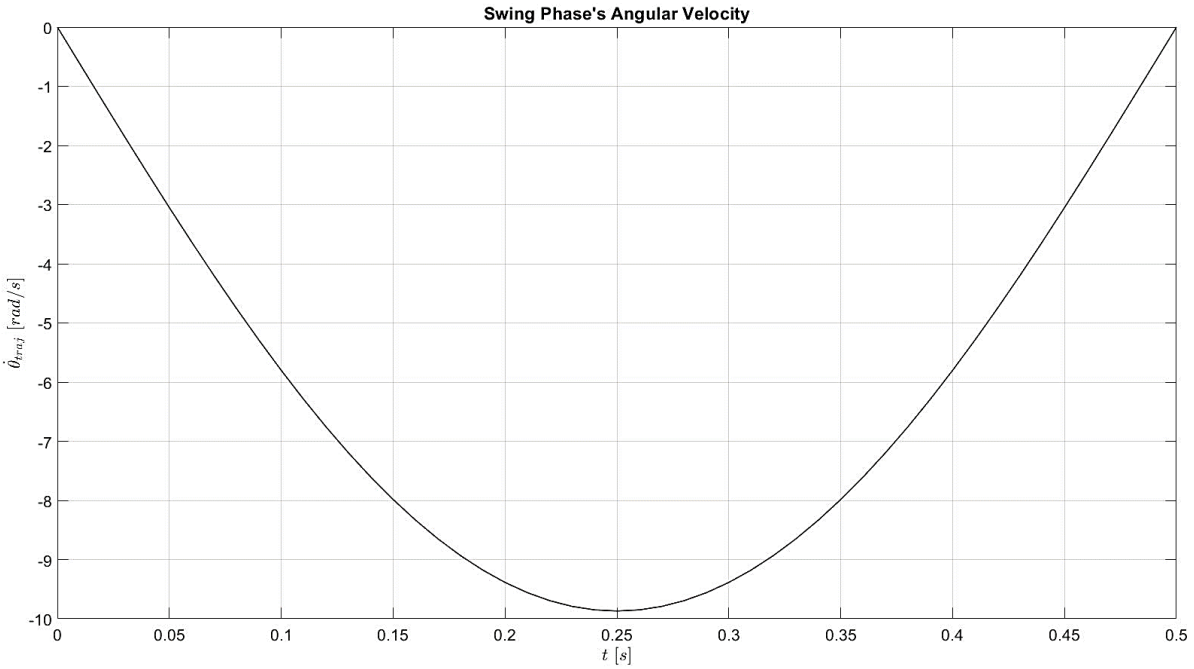


Figure 3-3. Swing phase's angular velocity for a single T_{swing}.

The horizontal and vertical axes of the ellipse are represented by a and b symbols, respectively, while (x_c, y_c) is the ellipse's center in equations (3-11)-(3-12). The swing phase occurs when $t_{traj} \le T_{swing}$. The toe's coordinates are given by:

$$\begin{aligned} x_{E,des}(t_{traj}) &= x_c + a\cdot\cos\left(\theta_{traj} + dir_{leg}\cdot\pi\right) \\ y_{E,des}(t_{traj}) &= y_c + b\cdot\sin\left(\theta_{traj}\right) \end{aligned} \tag{3-11}$$

Stance Phase

The stance phase occurs when $t_{traj} > T_{swing}$. The toe's coordinates are given by:

$$\begin{aligned} x_{E,des}(t_{traj}) &= x_c + \left(1-2dir_{leg}\right)\cdot\left(a-(t_{traj}-T_{swing})\cdot V_{stance}\right), \text{ with } V_{stance} = \frac{2a}{T_{stance}} \\ y_{E,des}(t_{traj}) &= y_c \end{aligned} \tag{3-12}$$

The above formulations are plotted in Figure 3-4. The trajectory is colored according to the linear velocity of each point. This speed along the trajectory is given in (3-13).

$$v = \sqrt{\dot{x}_{E,des}^2 + \dot{y}_{E,des}^2} \tag{3-13}$$

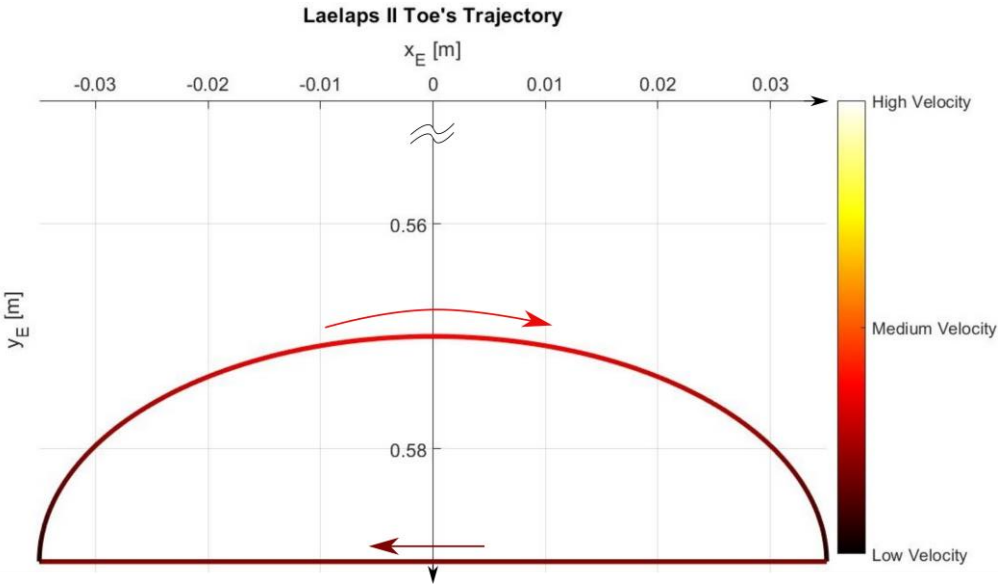


Figure 3-4. The constant velocity trajectory with velocity gradient (colorbar).

Note that the velocity is almost zero at the toe's take-off and touch-down (highlighted with black), contrary to the rest of the swing phase. Throughout the stance phase, the uniform dark red color indicates constant velocity motion. Moreover, various snapshots of a leg during motion along the planned trajectory are shown in Figure 3-5.

The implementation of the planner has provisions for smooth braking and standing still modes, if necessary (see Section 3.5.3). The smooth braking scheme is also used when the planner is commanded to change direction, to avoid unnecessary accelerations that would threaten stability. In a few words, the smooth-braking is achieved by gradually decreasing the trajectory's parameters (a and b in (3-11)-(3-12)). When these parameters reach zero, the robot stands still and waits for further commands.

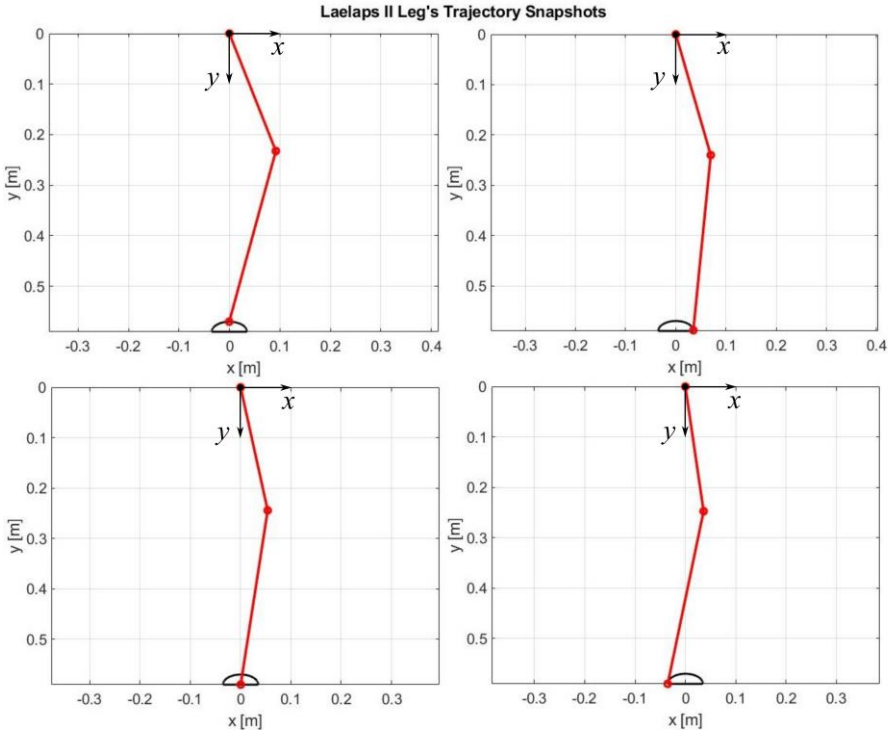


Figure 3-5. Snapshots of a leg during motion along the planned trajectory.

3.2.2 Leg's Workspace and Safety Features

The leg's workspace is dictated by its geometry, the position of the actuators and the knee's actuation mechanism, illustrated in Figure 3-6. The rod that actuates the knee creates a closed kinematic chain. Moreover, the knee is confined to move from 0^O to 90^O relative to the upper leg segment, with the highlighted mechanical stops.

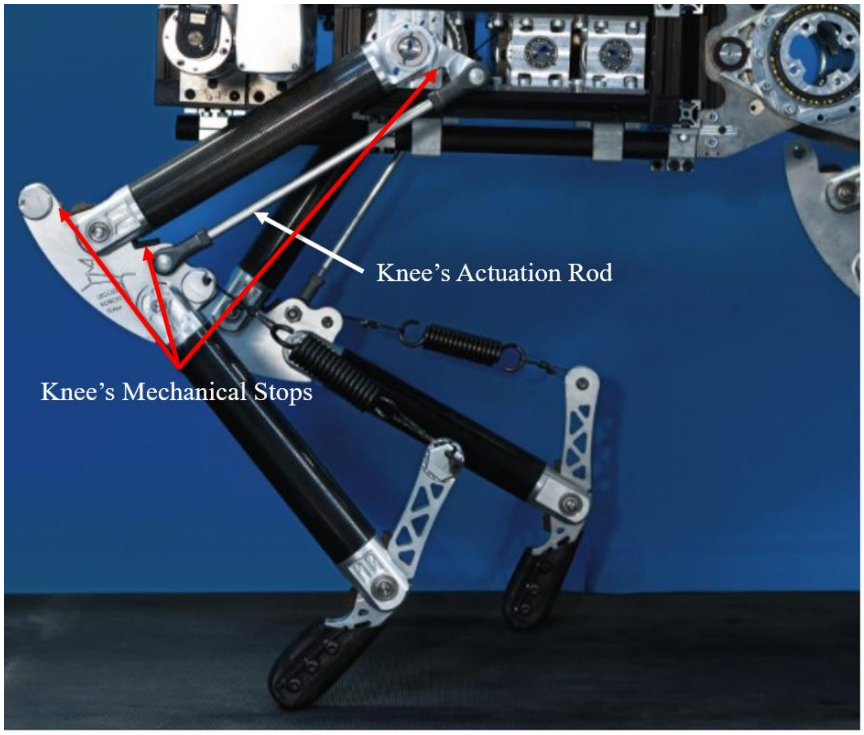


Figure 3-6. Laelaps II real-life leg.

The hips are also confined by the position of the actuators of the opposite side. The actuators of the right legs confine the left legs' hips in their front side by 40^O, while the motors of the left legs limit the right legs' hips in their hind side by 63^O. For safety reasons, two torsion end-stop springs are materialized in firmware, one for the fore and one for the hind side of the leg, by using the measurements from the installed absolute encoders (see Section 3.3). These virtual springs have, in their default configuration, 7^O workspace and ensure that the leg's upper limb is not going to collide with the robot's body. More details about them will be discussed in Section 3.3.2. Subsequently, the angular limits for each hip are 47^O for the fore and 70^O for the hind side. The resulting area that the leg's toe can move freely is highlighted with green color in Figure 3-7. The leg's geometry dictates the area's radial limits, given in (3-14).

$$\begin{aligned} l_{eff,min} &= \sqrt{l_1^2 + l_2^2} = \sqrt{0.25^2 + 0.35^2} = 0.4301 \text{ [m]} \\ l_{eff,max} &= l_1 + l_2 = 0.25 + 0.35 = 0.6 \text{ [m]} \end{aligned} \tag{3-14}$$

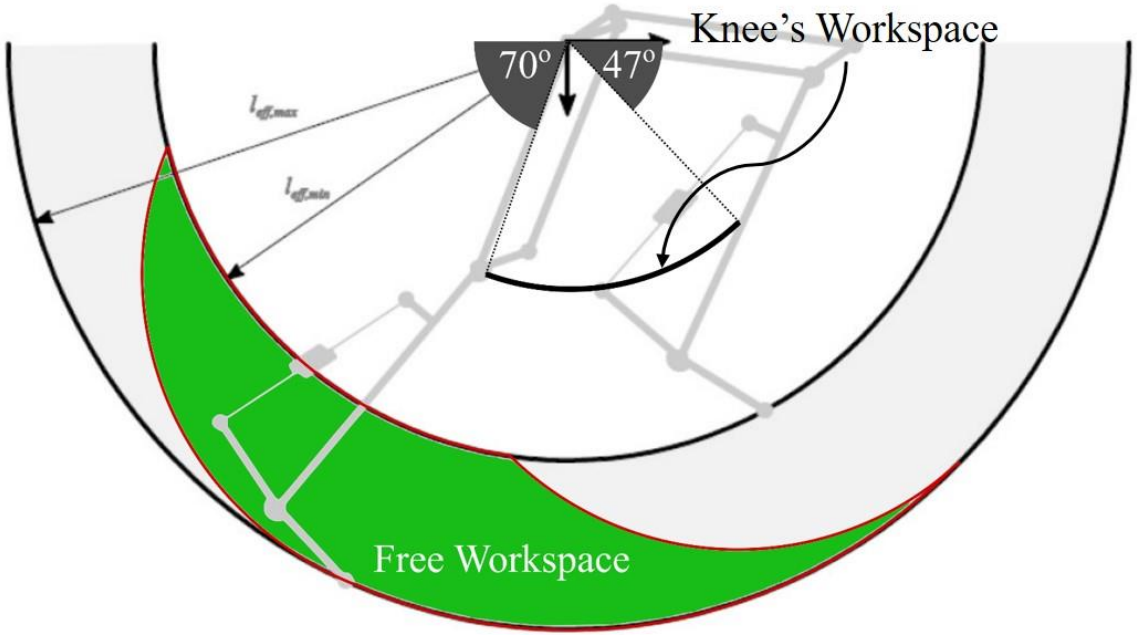


Figure 3-7. Laelaps II leg's workspace.

Apparently, the planner must generate trajectories inside the aforementioned area in order for them to be feasible. These semi-elliptical trajectories are described by certain parameters. These parameters should result in trajectories that comply with the leg's workspace. To ensure this, the parameter saturation feature is introduced and discussed next.

Parameter Saturation Feature

The planner generates trajectories that are bound by the geometric and angular restrictions of the leg. So, the trajectory's geometric parameters, namely x_c, y_c, a and b, have to be constrained accordingly. Figure 3-8 illustrates the underlying geometric problem intuitively. The leg's workspace is confined by two radial limits, namely (3-14) and two angular ones for the hip. The fore angular limit, namely φ_f, restricts the trajectory's vertical dimensions, namely y_c and b. Their values should comply with the equations in (3-15) that ensure that each point of the trajectory is under the y_{min} line.

$$y_{min} = l_1 \sin\left(\varphi_f\right) + l_2 \cos\left(0°\right) = 0.25 \sin\left(47°\right) + 0.35 = 0.5328 \text{ [m]}$$

$$y_{min} \le y_c \le l_{eff,max} \tag{3-15}$$

$$b_{max} = y_c - y_{min} \rightarrow 0 \le b \le b_{max}$$

On the other hand, the hind angular limit φ_h along with the outer ring of the workspace, restrict the trajectory's horizontal dimensions, namely namely x_c and a, as the detail-views of Figure 3-8 reveal. Their sum must be lower than the horizontal lengths introduced by the aforementioned boundaries. More specifically, the hind angular limit results in a maximum length (l_{hind}), while the outer radial boundary ($l_{eff,max}$) results in another linear limit (l_{ring}). So, the ellipse's horizontal dimensions are given in (3-16). These equations result in excluding the triangular-like shapes at the workspace's horizontal edges (highlighted in Figure 3-8).

$$\left.\begin{aligned} l_{hind} &= l_{eff,max}\cos(\varphi_h) = 0.6\cos(70°) = 0.2052 \text{ [m]} \\ l_{ring} &= \sqrt{l_{eff,max}^2 - y_c^2} \end{aligned}\right\} \Rightarrow l_{min} = \min(l_{hind}, l_{ring}) \qquad (3\text{-}16)$$

$$-l_{min} \le x_c \le l_{min} \text{ and } 0 \le a \le l_{min} - |x_c|$$

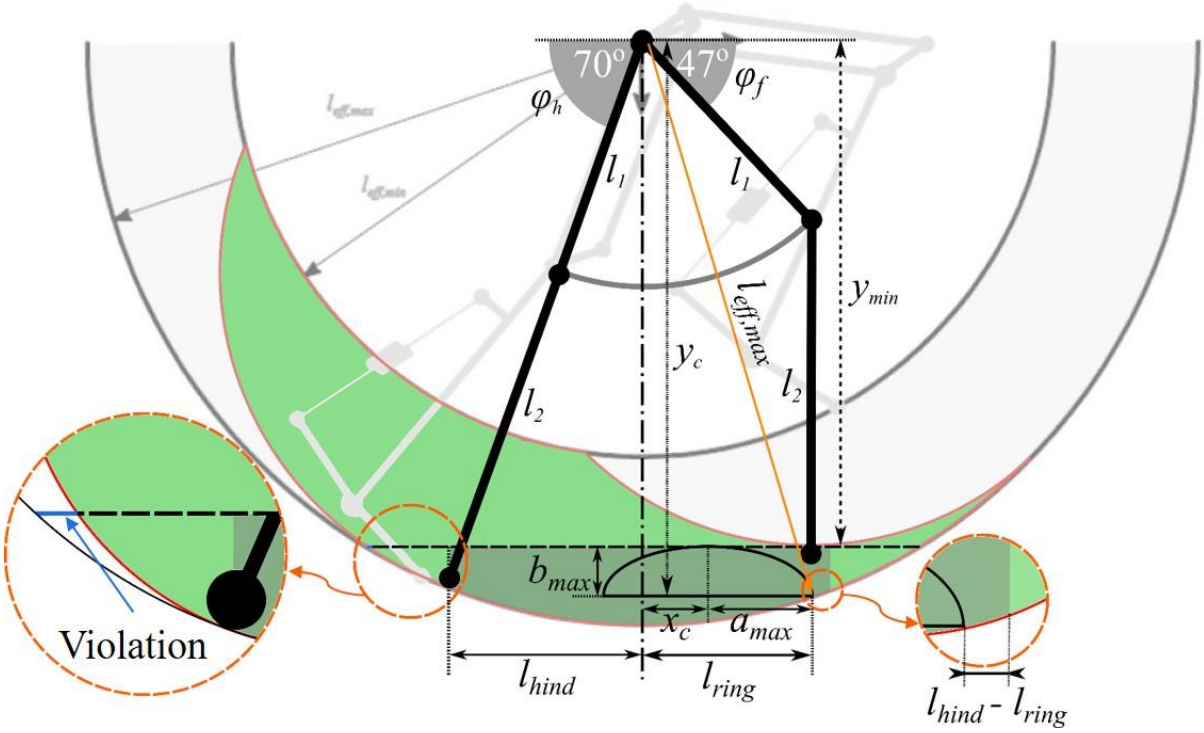


Figure 3-8. The geometric restircitons of the toe trajectory's parameters.

So, the trajectory's parameters that the EtherCAT master changes are bounded by the above discussed limits to have a valid end-result. So, if the EtherCAT master commands the leg to move in a bigger, "illegal" trajectory, the leg slave will perform a valid trajectory inside the leg's workspace, as close to the requested as possible.

From all of the above, it is realized that the planner's active workspace (dark area in Figure 3-8) is significantly smaller than the one that is dictated by the leg's geometry alone (light-green area in Figure 3-8). The radial angular limits for the hip that are introduced due to the topology of the Laelaps' drivetrain, restrict the toe's workspace to a very small area, compared to what could have been achieved with a different mechanical design. This insight should be taken into account in future designs.

Admittedly, by following the proposed methodology, the actual active workspace of the planner is restricted heavily in the vertical direction. There is, however, an analytical approach that can increase the workspace to cover the entire green area, shown in Figure 3-8. If the intersections of an arbitrary semi-ellipse with the four circular boundaries of the workspace are found analytically, certain restrictions can be drawn. This analysis is out of the scope of this thesis and is left for future work. In this investigation, the gain in the workspace's area should be evaluated taking into account a possible increase in the computational needs of the respective calculations. In the end, the choice may come as a trade-off between computational power needs and the workspace's area.

3.3 Absolute Encoders

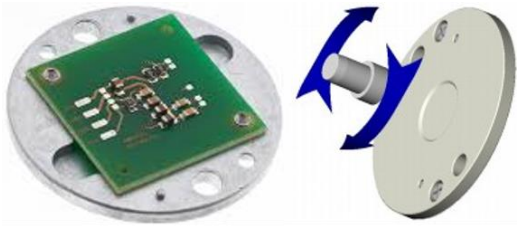
The robot's main electrical subsystem was not modified during this work; it is described in detail in [50], while the EtherCAT-related hardware is covered in [20]. However, in this work, a useful addition was made by installing absolute encoder modules at all joints. This modification was made mainly to automate the Laelaps II initialization procedure, in which the legs were positioned at their zero-angle positions to set the zero reference for the measurements of the incremental encoders, attached at the back of the motors. This tedious job was automated by installing absolute encoders that keep track of the legs' motions.

3.3.1 Hardware Integration

A rotary absolute encoder is a proprioceptive sensor that measures an object's angular position. Examples of such sensors consist of a glass disk with opposite opacities that correspond to 0 or 1 and a light emitter coupled with a photodetector (e.g. transistor). The number of rows or different bits that the disk is divided into corresponds to the resolution that the sensor achieves. In Figure 3-9, an example of such a disk is shown.

Figure 3-9. Example of an absolute 12-bit encoder's disk.

However, the absolute encoders that were installed on Laelaps II are of a different kind; they are magnetic 10-bit encoders, which means that the whole mechanism consists of a magnet and an IC-sensitive circuit that translates the motion of this magnet to a voltage gradient. More specifically, the *RLS RMF44VE10BA10* [51] magnetic linear encoder was selected. This module, along with its principle of operation, is visualized in Figure 3-10**.**



(a) **(b)**

Figure 3-10. (a) The RMF44VE10BA10 magnetic absolute encoder. (b) Magnetic absolute encoder's operational principle.

Its operation involves a DAC converting the magnet's rotation to a scaled 0 - 5 V linear voltage signal and eventually, overflowing after the prescribed angular range. The described behavior is presented in Figure 3-11.

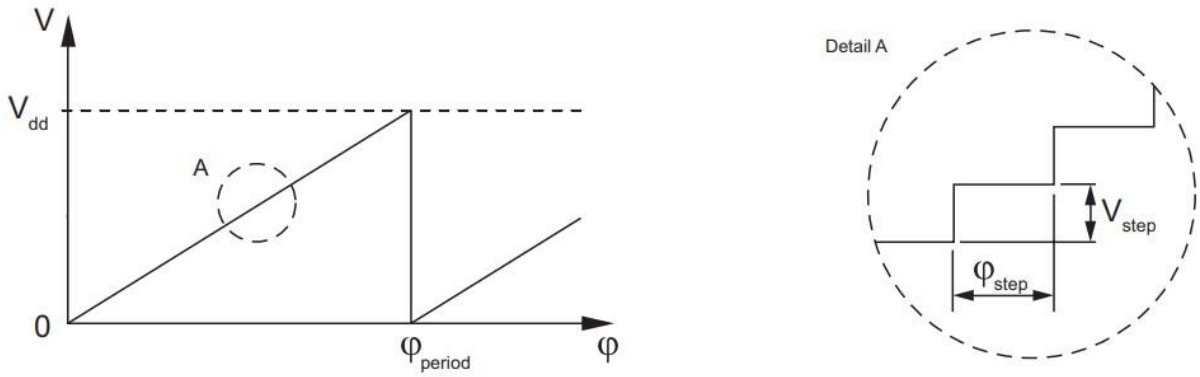
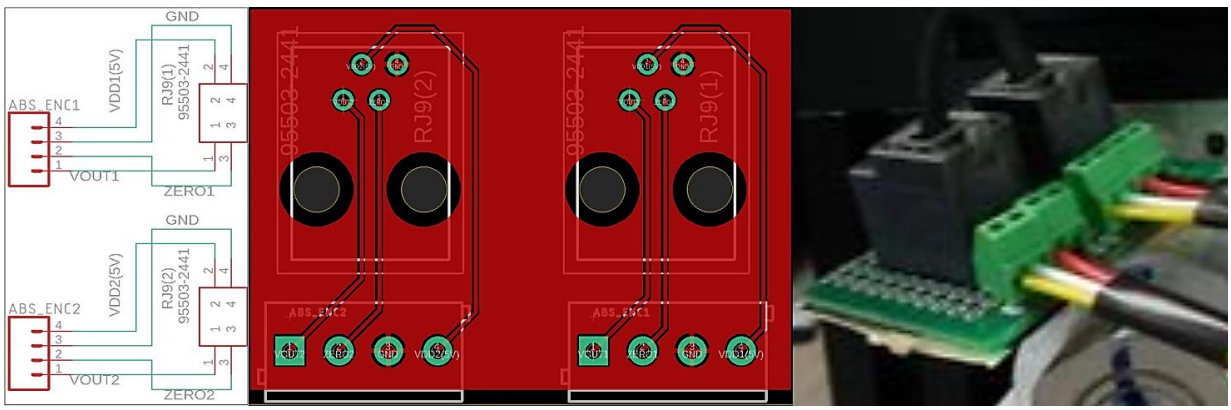


Figure 3-11. RMF44VE10BA10 voltage output behavior.

By consulting the accompanying datasheet [51], information about the module's power consumption and resolution can be extracted to embed it in a system. Provisions on the Delfino's shields had already been made [50] to accommodate these absolute encoders. The first step towards integrating the aforementioned modules is to design an appropriate wiring scheme and choose suitable connectors. After an extensive search, *JST XH 4-wire* connectors (Figure 3-12a) were chosen to interconnect the encoder PCBs with an intermediate small PCB (Figure 3-13) [52] that rewires the signals to be compatible with an *RJ9* type connection (Figure 3-12b). The construction of a custom wire, with *JST XH* and *RJ9* connectors at its edges, was not possible with the lab's equipment due to the tiny diameter of the specified wires.

(a) **(b)**

Figure 3-12. (a) JST XH 4-wire connectors. (b) RJ9 connector (male).



(a) **(b)** **(c)**

Figure 3-13. Absolute encoders' rewiring PCB: (a) schematic (b) board (c) assembled board.

Note that there are two *RJ9* cable configurations on the market, since such cables are usually used in telephony, in which their polarity does not matter. In this thesis, the **reversed** *RJ9* wiring layout has been adopted (Figure 3-14). Note that the wire colors in Figure 3-14 do not correspond to the *JST-XH* connector's cable-colors.

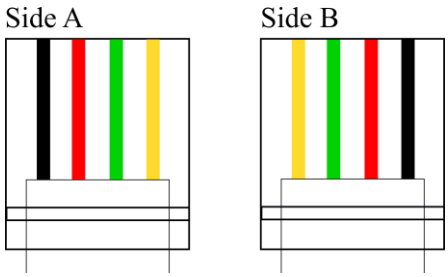


Figure 3-14. RJ9 wiring layout.

This layout results in reversing the pinout of the PCBs that the cable connects. Hence, the Delfino shield's pinout [50] is not the same with the one of the intermediate PCB. Table 3-1 illustrates the resulting pin mapping for the two PCBs.

Table 3-1. Reversed RJ9 cable's pin mapping.

Delfino Shield PCB #Pin	RJ9 to JST-XH PCB #Pin	Pin Description
1	4	**GND**
2	3	**ZERO (Reset)**
3	2	**V_{DD} (5 V)**
4	1	**V_{OUT}**

To make the mounting flawless, a housing was designed to ensure the whole structure's integrity. It was manufactured using the lab's 3D printer and the ABS material was selected. In Figure 3-15, the housing's design is illustrated, with the characteristic geometric strain relief elements, added to protect the cable from fatigue. The accompanying CAD files may be found in [52], and the corresponding schematics in Appendix L. To assemble the housing with the encoders, standard M4 bolts and nuts were used. The bolts stabilize the nuts in the grooves, without the need of an adhesive glue. Nevertheless, to facilitate the assembly process, standard multipurpose glue can be used to lock the nuts in place.

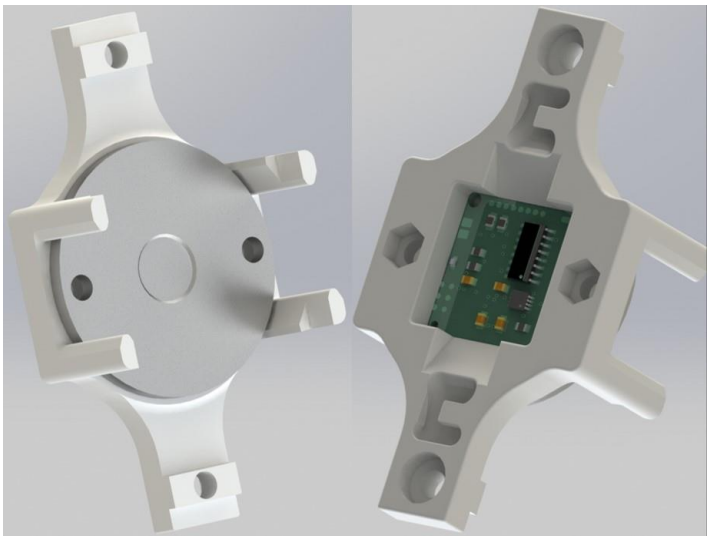


Figure 3-15. RMF44VE10BA10 housing assembly CAD.

The assembly, along with the magnet that accompanies each encoder, is designed to be placed on Laelaps II with ease. At first, the magnets are positioned at the center of the gearbox's shaft, using multipurpose glue. The encoder assembly is hooked onto the metal grooves of the Laelaps II body with M5 bolts. To stabilize it in place, Misumi's M5 slot-nuts [53] were used. Special attention was given to align each sensor with its magnet, in compliance with the manufacturer's tolerances [51]. The result of this procedure is shown in Figure 3-16.

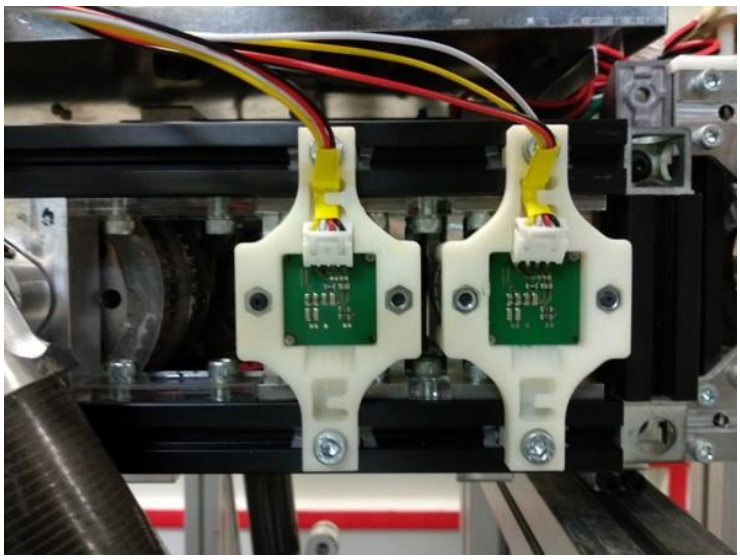


Figure 3-16. The absolute encoder assembly installed.

The sensors connect through the *JST-XH* cables to the *JST-RJ9* interfacing boards, which in turn connect to the prescribed terminals of the Delfino's shield. This wiring scheme is presented in Section 3.4.2.

3.3.2 Firmware Integration

The main task of the firmware drivers created to handle the absolute encoders is to retrieve their readings, translate them to be comparable with the readings from the incremental encoders and feed these to the Delfino eQEP initialization registers. The incremental encoders (*HEDL-5640* [54]) are mounted directly to the motors, contrary to the absolute ones that are mounted at the high-torque side of the gearboxes (Figure 3-17); this means that a certain reduction ratio has to be taken into account.

The 10-bit resolution of the RLS encoders corresponds to a total count of 1024 increments or a step-size of 0.35^{o} for the 360^{o} range of the module. As mentioned previously, in this thesis, the C2000 LaunchPads come with 12-bit and 16-bit ADC resolutions, with a 3.3V level of operation. To read the encoders, a voltage divider had been created on the Delfino's shields and the 12-bit ADC variant had been assigned to them. This means that the total measurement counts become 4095 due to the finer resolution that the ADCs provide. This does not mean that the measurements are more accurate. On the contrary, if the resolution increases furtherly, the measurements become noise sensitive (sensitivity errors) [55]. Concisely, the main firmware driver realizes the following mathematical formula, presented in (3-17). The idea here is to convert the absolute encoder's readings (n_{abs}) to the corresponding incremental encoder's counts (n_{inc}). Note that both modules are configured to report the angular position of their rotor with respect to the zero-angle position of the leg.

$$n_{inc} = \frac{range_{inc,counts} \cdot n_{abs}}{range_{abs,counts}} \cdot i_g \tag{3-17}$$

With:

- i_g : Gearbox reduction ratio (637/12 for the knee and 341/8 for the hip).
- $range_{inc,counts} = 2000$: Total incremental encoder's counts.
- $range_{abs,counts} = 4095$: Total absolute encoder's counts.

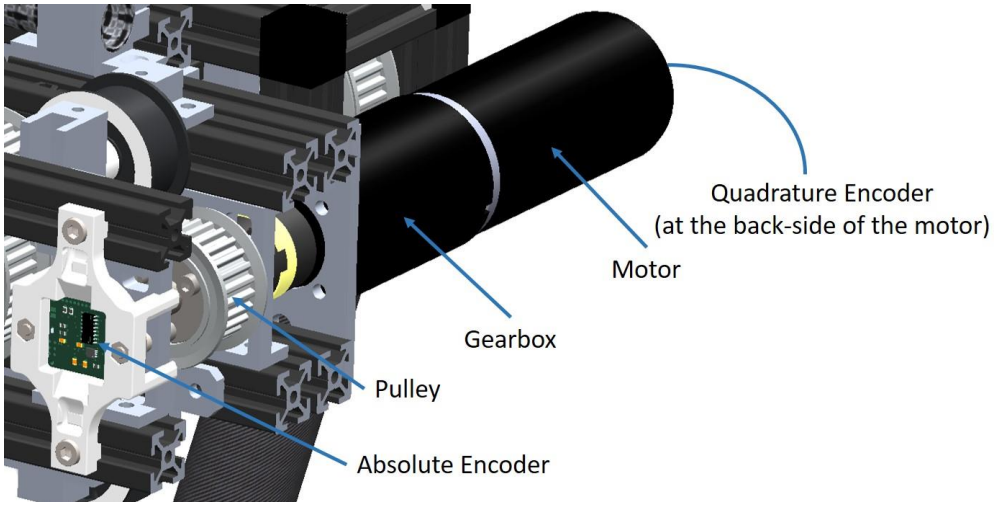


Figure 3-17. Laelaps II single joint's drivetrain.

Moreover, the introduction of absolute encoders in the Laelaps II setup allows the integration of a safety feature ensuring that the legs operate always inside the predefined angular limits, to avoid collisions with other parts of the robot. In Figure 3-18, the principle of operation becomes apparent. Note that the figure is exaggerated to give a better view of the discussed concepts.

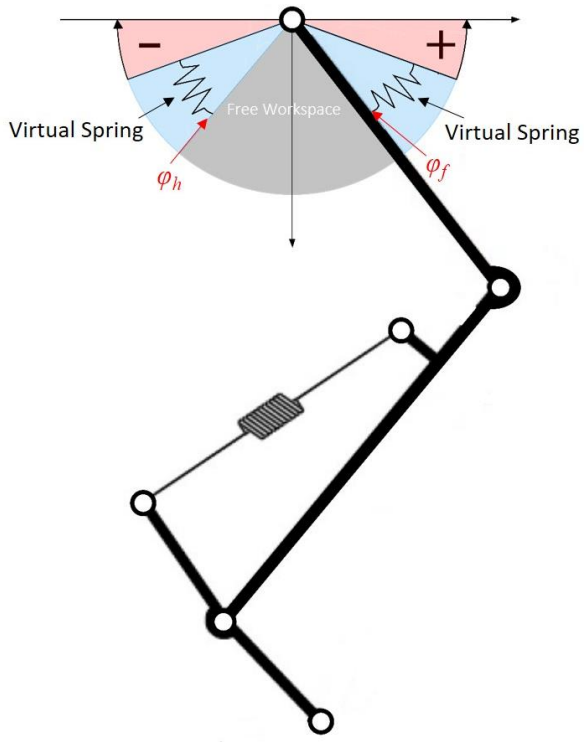


Figure 3-18. Laelaps II leg's end-stop routine.

The main task is to check the hip's encoder readings and to calculate the corresponding virtual spring's control effort, if the leg is moving in the light-blue region of Figure 3-18. More specifically, a linear interpolation scheme is used to emulate a linear torsion spring. Let φ_f and φ_h be the angles of the relaxed fore and hind springs, respectively and K be the user-defined spring's constant. By taking the absolute encoder's measurement θ, the virtual spring's control contribution (τ_{vs}) is given by (3-18).

$$\tau_{vs} = \begin{cases} K \cdot (\varphi_f - \theta), & \theta \geq \varphi_f \\ 0, & \theta \in (\varphi_h, \varphi_f) \\ K \cdot (\theta - \varphi_h), & \theta \leq \varphi_h \end{cases} \tag{3-18}$$

Note that the discussed end-stop springs are introduced to ensure that the upper limb of the leg does not crash with the robot's body due to an external load or other disruptions. The desired trajectory is configured not to violate the hip joint's free workspace (dark area in Figure 3-18). So, if the leg ends up inside the virtual spring's workspace (light-blue area), it will be the result of an externally applied load.

3.4 EtherCAT Motion Control Firmware Setup

Previous studies describe the EtherCAT technology and the former implementation of the Laelaps II firmware solution in detail [56]. This section initially lays a guide on how to import the created TI CCS project and run it on the C2000 LaunchXL-F28379D platform. Furthermore, the various aspects of the firmware are discussed in a structured way, to make the learning curve less steep and the overall process more efficient. As far as the EtherCAT slave software modules are concerned, the aim here is not to replicate the extensive available literature, but to highlight the significant routines and operations related to the user-application.

3.4.1 Required Hardware

Here, the required components for a slave node are given.

- 1x EtherCAT Slave piggyback Controller (ESC) with ET1100 chipset. All the required frame processing and EtherCAT functionality are implemented with this board. It is the hardware in which the Physical and Data Link Layers are realized.
- 1x TI's LaunchXL-F28379D (MCU). The application layer of the app is realized here, along with the generic EtherCAT stack and the IMU SPI communications.
- 1x Delfino shield designed in [50].
- 1x Custom EtherCAT shield designed in [20] that connects the MCU with the ESC.
- 1x DC-DC Step-Down Regulator 5V, 2A with USB [57].
- 2x RLS RMF44VE10BA10 absolute encoder assemblies for the initialization of the leg (see Section 3.3).

3.4.2 Hardware Connections

The control tower is shown in Figure 3-19. To assemble the boards, the components of the previous section are required.

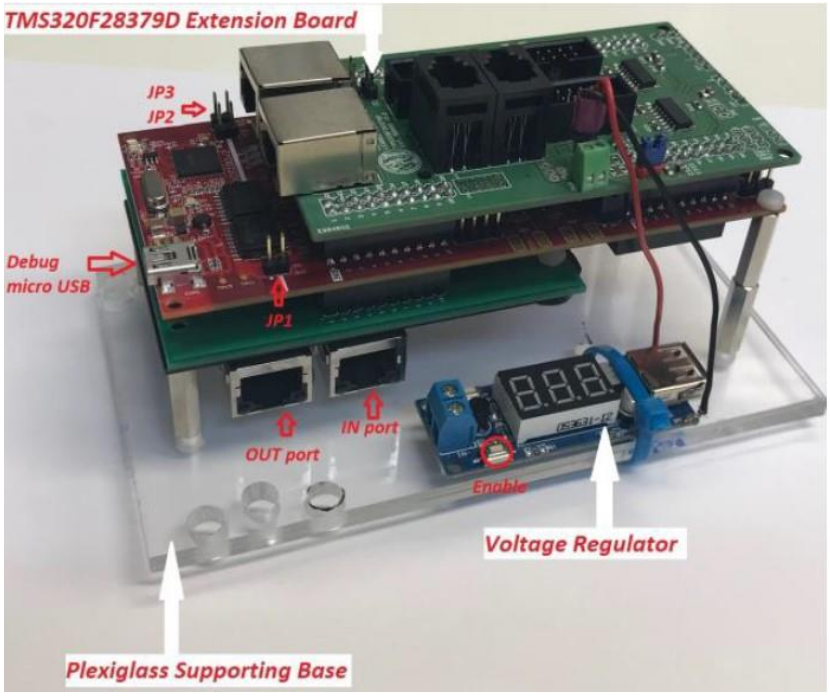


Figure 3-19. Laelaps II motion control's tower.

The whole EtherCAT slave (see Section 2.3.2) is supplied by the low-power supply channel of Laelaps II. This channel is fixed at 9 V as a future provision, since this voltage is commonly met in most mobile battery systems and it is supported by all low-power electronics currently operating on Laelaps II. Nevertheless the control tower's regulators [57] (highlighted in Figure 3-19) can receive 6.5 V to 40 V as input and output a 5 V, up to 2 A DC supply. Note that as the input voltage increases, the thermal power dissipated in the LDOs increases, too. So, in order not to stress their electronics, the input voltage should remain low. Moreover, since Delfino requires both 5 V and 3.3 V supply channels, an additional LM1117 Low-Drop Out (LDO) voltage regulator [58] is integrated on to the Delfino's shield. According to its manufacturer the aforementioned LDO outputs a 3.3 V power supply, with currents up to 800 mA.

The Delfino should be configured to operate with external supplies only. Specifically, the jumpers **JP1**, **JP2** and **JP3** must be removed (see Section C.8). Furthermore, the slave should be connected to the various subsystems that realize the forward and feedback control branches of the underlying control architecture. An overview of the required connections is illustrated in Figure 3-20. More specifically, the motors' drives are connected to the *RJ11* female terminals of the Delfino's shield. The incremental encoders are connected to the analog temrinals of the board, while the absolute encoders are connected to its female *RJ9* connectors. The highlighted *ZERO* pins (see Figure 3-20) reset the absolute encoders at the desired zero-angle position. To do this, a voltage pulse (3.3 V to 5 V) must be applied to the respective pins (*JZ1* for the knee's encoder and *JZ2* for the hip's). The board has provisions for this operation to be controlled by the Delfino, if necessary; hence the dual-pin headers highlighted in Figure 3-20. However, this capability remains disabled in the current setup.

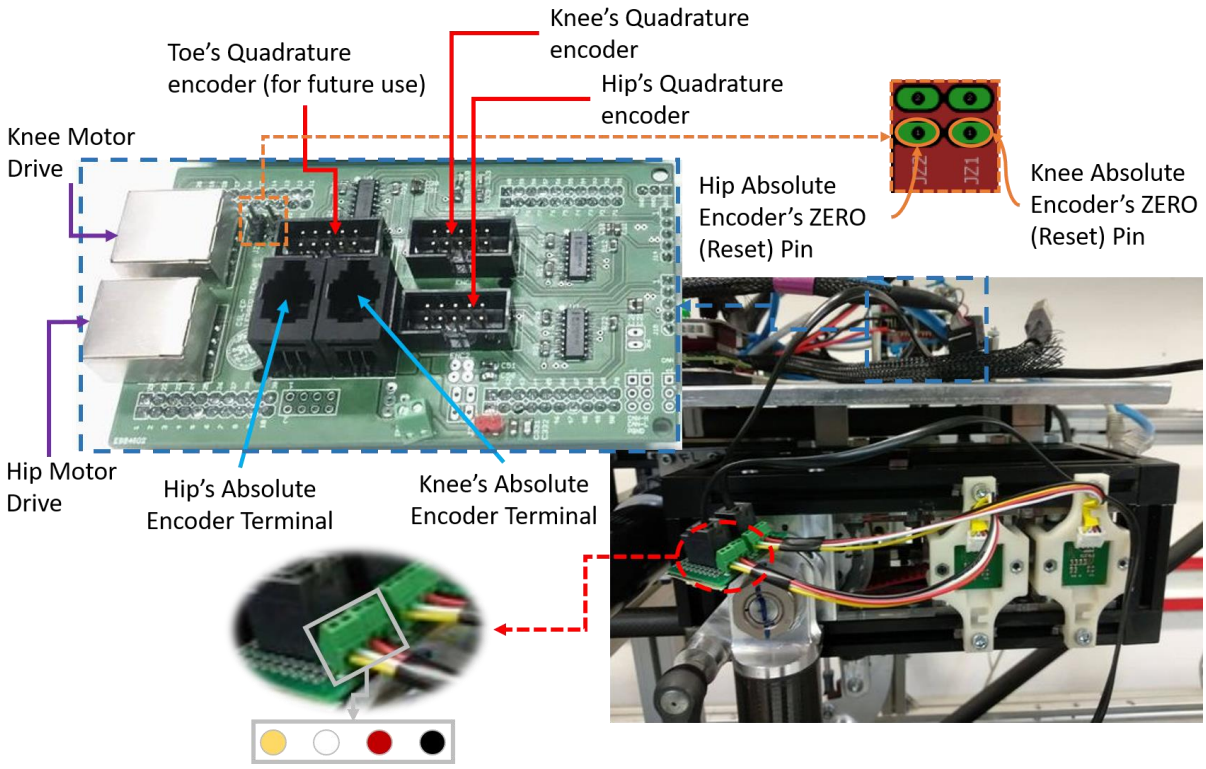


Figure 3-20. Leg slave hardware connections.

3.4.3 Required Software

Next, all the software components need to be installed. By navigating in the TI's site [59], the latest version of Code Composer Studio (CCS) should be installed with C2000 software components checked in the corresponding installation pane. One should also download the latest C2000Ware from the CCS *Resource Explorer*. Currently, the project supports version *3.03.00.00,* but with few adjustments in the properties menu, the update process to any future version should be easy.

Also, a master has to be installed, not necessarily on the same PC. There are plenty of choices for Windows and Linux masters. The Legged Robots Team currently maintains two master setups. The first uses Twincat 3, a robust and well-proven suite running on Windows with Visual Studio core. The second is a Linux hard real-time EtherLab Master with ROS enhancements. Due to the complexity of the high-end, but still in development Linux master, the use of Beckhoff's Twincat 3 is suggested. The interested reader may find and install Twincat 3 in the following link [40]. The CSL-EP's EtherLab master may be found, along with build and deployment information, in the corresponding Bitbucket repository [60]. Note that the slave-specific ENI files, required by the TwinCAT 3 master, can be found in [52], inside *TwinCAT Configuration Files* folder. There are two flavors, one for the newly introduced planner (designated by the suffix *CONSTVEL*) and one for the former planner (designated by the suffix *ELLIPTICAL*). So, depending on the desired planner, the appropriate ENI should be imported to TwinCAT 3 (refer to Appendix A).

3.4.4 CCS Project Import and Setup

To import and setup the CCS project, follow the next steps.

1. Download all the required project files by navigating to the CSL-EP Legged Robots Team BitBucket repository [52]. Open TI's CCS. If it is not already installed, see details on how to do it as well as import the downloaded project in Appendix E (Section E.1).
2. Next, a required Target Configuration for the debugger should be set. If such configuration is not available from prior use, details on how to create one from scratch may be found in Appendix E (Section E.2).
3. Modify the custom path variables to appropriate ones for the PC used at the time. Those variables make the C2000Ware libraries visible to the project's scope. To do this:
    - *Right-Click* on the project's entry in the solution explorer's tab and select *Properties* (Figure 3-21).
    - Opt for *Linked Resources* tab and modify the below variables.
        i. **C2000Ware_LOC** to match the local installation folder of C2000Ware.
        ii. **CLAMATH_ROOT** to match the CLA Math library's installation folder, commonly in C2000Ware main folder in *./libraries/math/CLAmath/c28.*

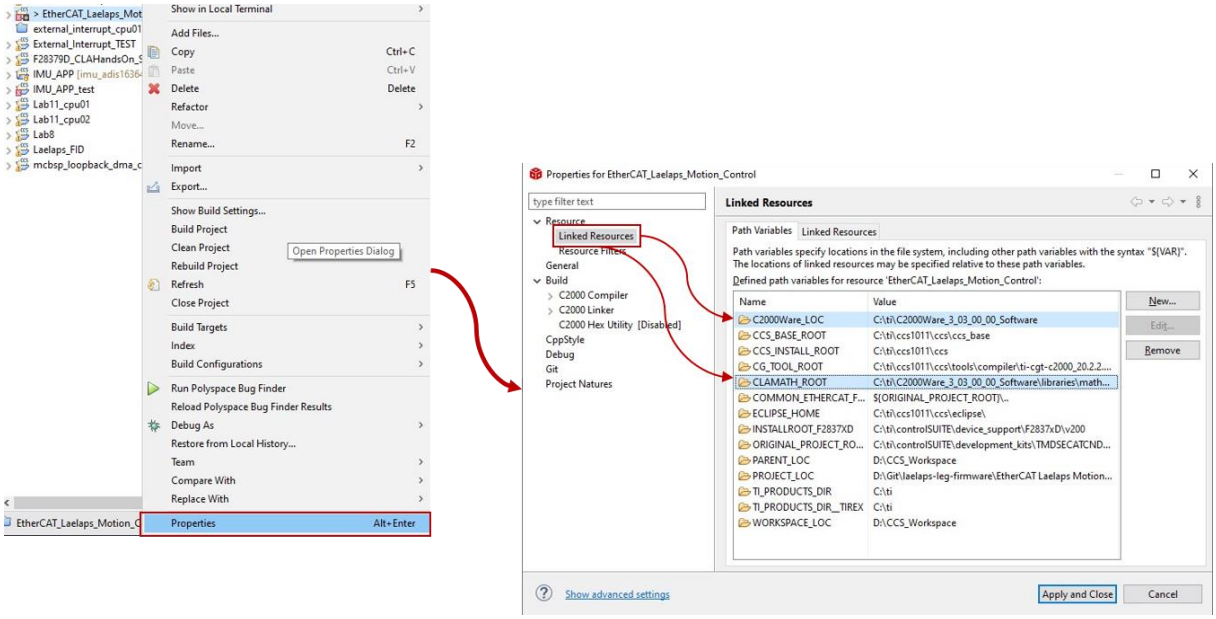


Figure 3-21. CCS project's path variables.

Note that the project has multiple configurations. Details on how to enable and adjust these configurations are given in Section 3.5.7. In the firmware description that follows, whenever a feature is described, the corresponding predefined symbol will be referenced to have a clear view of the underlying dependencies.

Finally, details on how to deploy the firmware to a LaunchXL-F28379D can be found in Appendix E (Section E.3). At this point, the firmware can be deployed in its default configuration. If the defaults do not suffice, the next sections describe thoroughly all the available options.

3.5 Firmware Description

The firmware's main operation can be adumbrated in four ISRs, as Figure 3-22 illustrates. Three of them are triggered by the ESC's DC unit, as external hardware interrupts (see Section 2.2.4) and the fourth one (highlighted with gray) is triggered by Delfino's hardware timer 0.

Note that in the default TI's configurations, this hardware timer is used as a free-counter for EtherCAT-related operations. For interrupt priority reasons that will be discussed in Section 3.5.5, the TI's default EtherCAT Hardware Abstraction Layer (HAL) (ControlSuite's *TMDSECATCND379D_V1.0* development kit [61]) was modified.

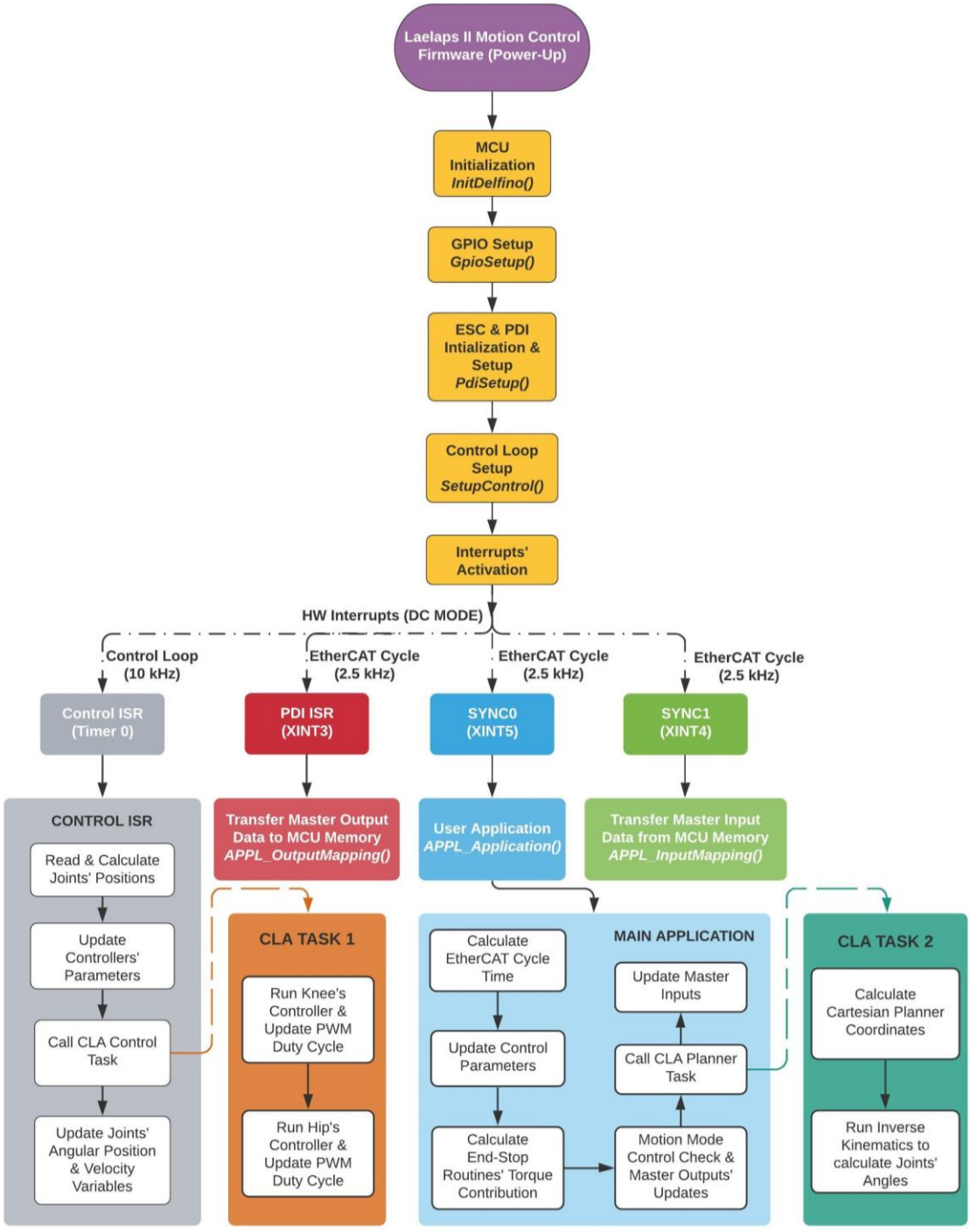


Figure 3-22. Laelaps leg firmware overview.

To facilitate the future development and understanding of the firmware, the MISRA C:2012 coding standard has been adopted [62] (refer to Section 3.6). Moreover, the code complies with the coding style discussed in Appendix G. Note that to this end, the created Doxygen documentation [52] could help in getting an overall grasp of the firmware faster.

The program may be considered in three distinct sections. The EtherCAT stack, the planner and the controllers, with interconnections and dependencies among them to achieve the desired functionality. The CCS project is organized in the following folders:

- *hal*: This folder materializes the EtherCAT HAL. Specifically, it contains all the peripheral drivers and routines for handling the PDI's communications along with other application-related functionalities.
- *SPI_EtherCAT_slave_stack*: In here, the generic EtherCAT stack's files can be found. They materialize the procedures discussed in Chapter 2. From now on, this folder will be referenced as *stack*, for short.
- *Planners*: This folder contains all the planning-related files. They realize the discussed planners and other supplemental procedures (Section 3.2).
- *cmd*: This folder contains the created linker command files that dictate the project's memory allocation.

Those said, the firmware's execution flow is analyzed next. Firstly, the required initializations of the most significant peripherals and modules are discussed (highlighted with yellow in Figure 3-22). Moreover, the control-related routines (highlighted in gray) are described. Finally, the main application and its dependencies are analyzed (highlighted in light-blue), along with other EtherCAT related implementations.

3.5.1 Firmware Initialization

The firmware's execution begins at the ***main()*** function located in the *EtherCAT_Laelaps_Motion_Control.c* file (inside the project's *stack* folder). The ***main()*** function calls the ***HW_Init()*** (located inside the *c28xxhw.c,* under *stack*) that in turn initializes the Delfino MCU and the ESC. Eventually, the Delfino MCU is initialized by calling the ***ESC_initHW()***, located inside the *etherCAT_slave_c28x_hal.c* (under the *hal* folder). Then, the low-level modules of the system are initialized by calling ***InitDelfino()*** routine. In this function, depending on the run-time mode (debug or release), the required memory sections are copied from FLASH to RAM. Next, the ***GpioSetup()*** routine is called and configures the GPIO muxing options for every peripheral of the MCU. Then, the ***PdiSetup()*** handles the PDI layer's initialization, by configuring the EtherCAT-related external interrupt XINT3 for the ***PDI_Isr()*** and the used SPI module. Currently, Delfino's SPIA is assigned to perform the PDI-related communications (**USE_SPIA** symbol, see Section 3.5.7). Moreover, the PDI-related ISR is configured here, as XINT3 interrupt, having the highest priority among the three EtherCAT ISRs (Figure 3-22). The purpose of the aforementioned interrupt will become apparent in the following sections. These routines are components of the HAL and can be found in *etherCAT_slave_c28x_hal.c* (under the *hal* folder).

After the generic low-level configurations, the ***SetupControl()*** routine is called. This function initializes the control peripherals used in the application. First, the CLA unit (see Section C.3) of the microcontroller is initialized by calling the ***InitCla()*** routine, located inside the *clasetup.c* (under *hal*). This happens only if the application is preconfigured to use the CLA capabilities, as it will be discussed thoroughly in Section 3.5.7. At this point, the control ISR, triggered by the timer 0, is mapped to the appropriate interrupt handler. If the CLA control execution is enabled (**CLA_CTRL** symbol, see Section 3.5.7), the ***ClaControl_Isr()*** serves the control loop. In any other case, the ***CpuTimer0_Isr()*** is used. Both of them are located in *etherCAT_slave_c28x_hal.c*, under the *hal* folder. More on this matter will be discussed in Section 3.5.2.

Next, ***SetupControl()*** runs the quadrature encoder startup sequence. In the previous version of the firmware, the encoders had to be initialized manually. With the introduction of the absolute encoders, the whole process has been automated (**AUTO_STARTUP** symbol, see Section 3.5.7). The described functionality is materialized by two functions located in *posspeed.c* (inside *hal* folder), ***ReadAnalog()*** and ***POSSPEED_InitJoints()***. These functions are described next.

The function ***ReadAnalog()*** controls the ADC operation. To reduce the noise in the measured quantities, the oversampling technique [63] is used. In brief, the routine takes four readings from each ADC and averages them. Next, it calculates the relative angle w.r.t. the zero-angle position of the leg and stores the results to memory. Besides its usage in the initialization process, the function may be used also during normal run-time, cyclically, to output information about the shafts' position and diagnose possible slippage or other failures in the drivetrain. Note that the ADCs are used in blocking mode. This means that they produce interrupts that do not propagate to the CPU, since the control flags, like the *PIERx*, are disabled (see Section C.3). This way, each time the CPU requests an ADC reading, it has to wait until the operation is finished; hence, ***while*** loops are used in the code.

The function ***POSSPEED_InitJoints()*** is responsible for the conversion of absolute encoders' measurements to the corresponding incrementals' readings. In brief, the routine performs a sampling procedure that takes the absolute encoders' readings, provided by calling the ***ReadAnalog()*** function, averages them and converts the results to two 32-bit unsigned integers that correspond to the sought incremental encoders' counts. The described procedure is actually the realization of (3-17), for both joints of the leg. The user may change the averaging filter's duration by modifying the **ABS_ENC_SAMPLING_TIME** (in [s]) definition in *posspeed.h* file (under *hal*). Lastly, the whole process can be monitored, since Delfino's blue and red LEDs are blinking throughout the routine's run-time.

Next, the end-stop routine functionalities discussed in Section 3.3 are configured. To activate this functionality, the **END_STOP** Predefined Symbol (Section 3.5.7) should be defined. The routines involved in the process are listed below. They are located inside the *posspeed.c*, under the *hal* folder.

- ***ENDSTOP_Init()***. Initializes the end-stop routine's instance. By default, it is called inside ***SetupControl()*** routine, during startup.
- ***ENDSTOP_FvsCalc()***. Calculates the virtual spring's torque in the configured EtherCAT loop cycle.
- ***ReadHipPos()***. Reads the hip's absolute encoder and feeds the measurements to ***ENDSTOP_FvsCalc()*** function.

To modify the behavior of the end-stop routine, there is a variety of different options to consider in order to achieve the desired result. Nevertheless, by default, the firmware emulates a linear torsion spring (3-18), with the parameters shown in Figure 3-23. The aforementioned options are adumbrated below:

- Change ***CalcFvs()*** to realize the desired spring type. By default, a linear one is used.
- Modify **ENDSTOP_DEFAULTS** (located in *posspeed.h*, under *hal*) to change the range of the restricted angles of Figure 3-18. By default, these angles are set to 63^{o} and 40^{o} for the hind and fore boundaries, respectively with a 7^{O} workspace for each virtual spring. Note that the measurements of the absolute encoders have significant inherent noise, and thus very small springs' workspaces are not recommended. Generally, the virtual springs' workspace should lie in the interval $[5°,10°]$.

- Finally, the initialization function's call at startup should be changed to match the requirements at the time, as shown in Figure 3-23.

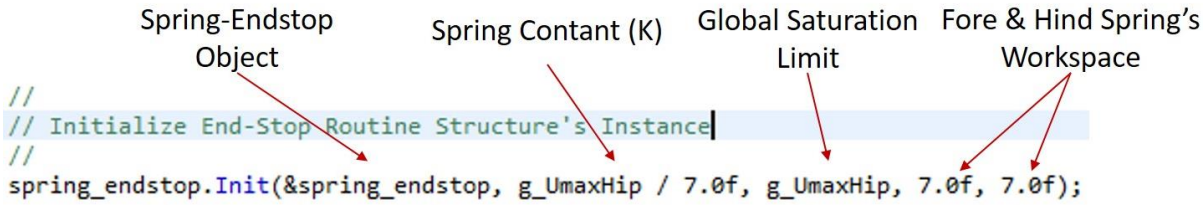


Figure 3-23. End-stop routines initialization function call.

After the end-stop routines, ***SetupControl()*** calculates the planner's boundary values, namely y_{min} in (3-15) and l_{min} in (3-16), by calling ***TrajectoryBoundaries()***. Then it sets up the control loop's timer period. To do this C2000Ware's ***ConfigCpuTimer()*** is called, as Figure 3-24 illustrates.

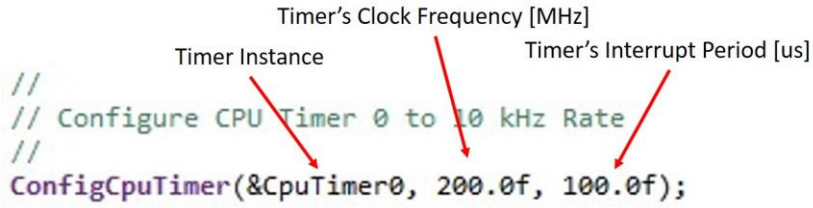


Figure 3-24. Timer 0 interrupt period setup.

Moreover, the ***SetupControl()*** configures the PWM channels that control the leg's actuators, by calling the ***InitEpwm()*** routine. The ePWM1 module is assigned to the knee's drive, while the ePWM2 module is assigned to hip's one. Their clocks are running at a 100 MHz rate ($f_{pwm,clock}$). In the current application, the ePWMs are configured to operate in a 20 kHz rate (f_{pwm}), as the equation (3-19) suggests, in up and down count mode, with $TBPRD$ being the time-based period register that contains the total period's counts, according to [64]. This frequency is dictated by the capabilities of the motors' drives [65] and must be greater than the respective control loop's frequency (see next section). According to Nyquist theorem [66], the control loop's frequency cannot be higher than the half of the ePWM operational frequency, since duty-cycle updates are configured to occur at the ePWM counter's zero-count event (refer to [64]). This prevents PWM signal disruptions by "illegal" duty-cycle updates.

$$f_{pwm} = \frac{f_{pwm,clock}}{2 \cdot TBRD} = \frac{100\ [\text{MHz}]}{2 \cdot 2500} = 20\ \text{kHz} \tag{3-19}$$

Moreover, the control-related interrupts are enabled in the corresponding PIE groups (see Section C.3 for further information about Delfino's interrupt architecture). Next, the EtherCAT Sync interrupts are configured. ***Sync0_Isr()*** is configured by ***ESC_configureSync0GPIO()*** as XINT5 interrupt. Likewise, ***ESC_configureSync1GPIO()*** configures the ***Sync1_Isr()*** to be triggered by the XINT4 external hardware interrupt. These interrupts will be furtherly discussed in Section 3.5.5. At last, the MCU's interrupts are activated and after some preconfigured EtherCAT checks the firmware advances to the main-loop.

3.5.2 Motion Control

An important function of the main-loop is the timer 0 interrupt that materializes the joints' controllers. This ISR requests and reads the position indicated by the quadrature encoders,

updates the controllers' parameters to have up-to-date gains and eventually calls them. To this end, two PIV (Proportional-Integral-Velocity) controllers, previously introduced in [20], were used. The firmware has provisions for CPU or CLA controller run-time. The C2000 Digital Control Library (DCL) comes with a variety of optimized choices to cover a wide range of possible applications. The reason PIV control is selected over the traditional PID, is that it does not add any zero in the closed-loop transfer function of the system, and as a result, it presents low oscillations during the transient state. The controller is displayed in Figure 3-25. For a deeper understanding of its architecture, refer to [67].

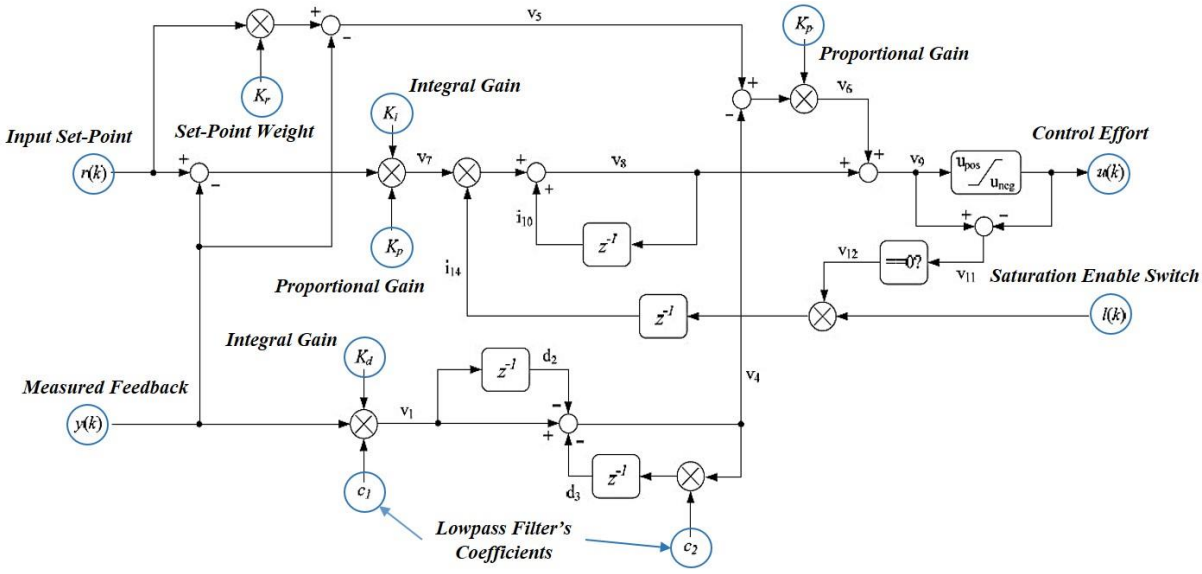


Figure 3-25. DCL's PIV controller overview [67].

To give a more intuitive understanding of the used PIV controller, its continuous time equivalent, without the implemented filter and saturation, is given in equation (3-20); with K_p being the proportional gain, K_d the velocity gain, K_i the integral gain and K_r the input set-point's weight parameter that is left to 1. The EtherCAT master sets these gains to each slave, along with the saturation limits and the filter's bandwidth. Note that the proportional gain takes part in all of the PIV's terms.

$$u(t) = K_p\left(K_r r(t) - y(t)\right) - K_p K_d \dot{y}(t) + K_p K_i \int \left(r(t) - y(t)\right) dt \tag{3-20}$$

The DCL's PIV implementations come with tunable low-pass filters to cutoff noise from the velocity approximation in the derivative path. Concisely, the controller's most significant configuration parameters are the control gains, a parameter related to its low-pass filter and the control effort's saturation limits (variables *g_UmaxHip* / *g_UmaxKnee* in firmware, inside *etherCAT_slave_c28x_hal.c*). The saturation limits are currently kept under the actuators' continuous operation current limits, namely 41.17% and 38.25% for the hip's and knee's motors, respectively. In Figure 3-25, the controller's low-pass filter is materialized by the c_1 and c_2 coefficients that in turn are calculated in (3-21), with T_s being the digital controller's sampling period and τ_{cbw} a time constant related to the filter's bandwidth (f_{cbw}) (3-22).

$$c_1 = \frac{2}{T_s + 2\tau_{cbw}} \text{ and } c_2 = \frac{T_s - 2\tau_{cbw}}{T_s + 2\tau_{cbw}}, \text{ with } T_s = 0.1\ [ms]\ (10\ kHz) \tag{3-21}$$

$$\tau_{cbw} = \frac{1}{2\pi \cdot f_{cbw}} \tag{3-22}$$

Special attention should be given to the saturation limit of the hip, if the end-stop routine is activated. Then, the end-stop routine's contribution (τ_{vs}) is added to the saturated output of the controller ($\tau_{sat,hip}$) and their sum is saturated again, according to the global saturation limit defined in Figure 3-23. This does not apply to the knees' control output, since the end-stops are located only at the hips.

The ***DCL_runPID_L1()*** function of the PIV controller is coded in assembly, like the one used in the previous implementation, namely ***DCL_runPID_C1()***, but presents better CLA execution time than the corresponding FPU32 run-time. Definitive proof of that claim is provided by the results of some benchmarks executed by TI and may be found in DCL User's Guide [67]. A comparison table is extracted and presented as Table 3-2 below.

Table 3-2. DCL controller performance comparison.

Function	Controller Type	CPU Compatibility	Cycles	Size [Words]
DCL_runPID_C1	PIV	FPU32	83	99
DCL_runPID_C4	PID	FPU32	86	92
DCL_runPID_L1	PIV	CLA	53	70
DCL_runPID_L2	PID	CLA	45	58

To make DCL operational there are a couple of steps, which should be taken care of. The reader is encouraged to look up the information on DCL in its accompanying manual [67]. Nevertheless, a brief workflow is laid out in Appendix F.

Control ISR

The default control ISR, namely ***ClaControl_Isr()***, is located in *etherCAT_slave_c28x_hal.c*, under *hal* folder. It materializes the digital control loop and is called by timer 0 at a 10 kHz rate. In the following code snippet, the routine's operation becomes apparent. Briefly, the handler retrieves the angle feedback of the quadrature encoders, updates the controller instances by calling ***UpdatePidParameters()*** and triggers the CLA control task 1. The alternative ISR, instead of triggering the CLA task, runs the CPU flavor of the same controller type, namely ***DCL_runPID_C1()***.

```
__interrupt void ClaControl_Isr(void)
{
    //
    // Acquire Leg's Joints Position Readings -- Reviewed SA Warning
    //
    qep_posspeed.Calc(&qep_posspeed);

    //
    // Translate Raw Positions to Normalized Angles with Gear Ratio
    //
    g_NormAngleKnee = (float)g_RawPosKnee * BELT_REDUCTION_RATIO
                    * KNEE_REDUCTION_RATIO / 2000.0f;

    g_NormAngleHip = (float)g_RawPosHip * BELT_REDUCTION_RATIO
                    * HIP_REDUCTION_RATIO / 2000.0f;

    //
    // Update Leg Controllers' Parameters
    //
    UpdatePidParameters();
```

```
    //
    // Enable EALLOW Register Access
    //
    EALLOW;

    //
    // Software Trigger CLA's Control Task (Task 1
    //
    Cla1ForceTask1();

    //
    // Disable EALLOW Register Access
    //
    EDIS;

    //
    // Update Angle Application's Variables in [100*Deg]
    //
    g_KneeAngle100 = (float)g_NormAngleKnee * 360.0f * 100.0f;
    g_HipAngle100 = (float)g_NormAngleHip * 360.0f * 100.0f;

    //
    // Update Speed Application's Variables in [1000*rad/s]
    //
    g_VelocityKnee1000 = (int32_t)((float)(qep_posspeed.SpeedPr1 * 1000.0f));
    g_VelocityHip1000 = (int32_t)((float)(qep_posspeed.SpeedPr2 * 1000.0f));

    //
    // Increment Timer 0 Interrupt Counter
    //
    CpuTimer0.InterruptCount++;

    //
    // Acknowledge the PIE Group Interrupt
    //
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

} // ClaControl_Isr
```

CLA Setup

Following the steps, referenced in Section C.3, the Laelaps II motion control firmware comprises functions that set up the CLA module along with task-specific code. The code is modular and may be easily generalized to support any common application. This framework consists of the following files:

- *clasetup.c*, which includes ***InitCla()*** function that is responsible for the module's initialization and setup.
- *clashared.h*, a header that contains all the declarations of the routines, variables, and definitions that CLA uses.
- *clatasks_C.cla*, a *.cla* source file that contains task-specific routines that realize user-code functionalities.
- *cla_utilities.cla*, a *.cla* source file that accommodates supplementary CLA routines.

The initialization of the CLA is a generic procedure and as such, it is presented in Appendix C (Section C.3). With a few modifications concerning the used tasks, it can be adjusted to serve different applications. More on this can be found in the literature [64].

CLA Controller Task

Two of the total eight CLA tasks are used. The first one, with the highest priority, runs the controllers and sets their PWM outputs. It is called inside the ***ClaControl_Isr()*** (located in *etherCAT_slave_c28x_hal.c*, under *hal* folder) by a software trigger at 10 kHz. The second materializes the planner and will be furtherly discussed in Section 3.5.3.

```
__interrupt void Cla1Task1(void)
{

//
// CLA Control Task is used
//
#if defined(CLA_CTRL)

    //
    // Call Knee's & Hip's Control Routines
    //
    RunKneeController();
    RunHipController();

    //
    // Increment CLA Task 1 Debug Counter
    //
    g_ClaCounter++;
#endif // CLA_CTRL

}   // End Of Cla1Task1()
```

3.5.3 Trajectory Planning

The ***ConstVelApp()*** routine (located in *constvel.c*, under *Planners*) realizes the main user-application that calls the newly-introduced planner's routine and processes the used variables, interfacing every application's aspect with the EtherCAT protocol. The CPU planner is materialized in ***ConstVelPlanner2()*** routine. The alternative CLA planner is called with a software trigger, namely ***Cla1ForceTask2()***.

Note that the ***ConstVelApp()*** is called inside the ***APPL_Application()*** (discussed in the next section) at the EtherCAT's loop rate (2.5 kHz). Also, by replacing the **CONSTVEL_PLANNER** symbol with the **ELLIPTICAL_PLANNER** one, the previous implementation's planner can be used (see Section 3.5.7).

After the necessary EtherCAT cycle calculation, the routine calls the ***UpdateControlParameters()*** to refresh the control-related parameters of the application by processing the latest EtherCAT slave PDO objects received by the ESC. Next, the end-stop routine's control effort is acquired, if used (**END_STOP** symbol, see Section 3.5.7). Moreover, the time variable of the trajectory (3-8) is calculated. Next, the routine implements a state machine, illustrated in Figure 3-26. Entering suitable parameters to all EtherCAT output variables and switching into Operational State (1) will force Laelaps to execute the desired movement (see Section 3.5.4).

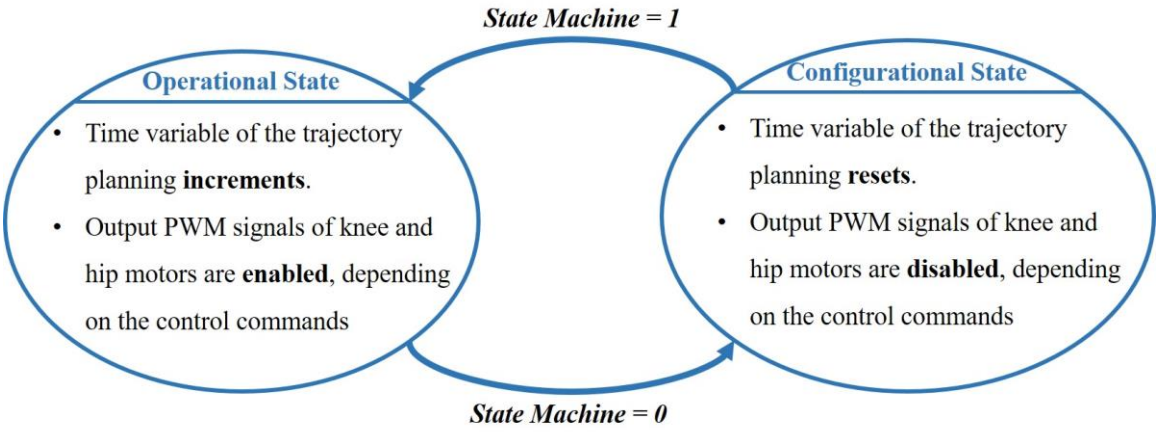


Figure 3-26. Laelaps II motion control's state machine.

Special attention should be given in the ***MotionModeControl()*** routine that controls the planner's operational state according to the master's commands. There are specific variables that dictate the robot's motion state, like moving forward, backward or standing still (see Section 3.5.4). Specifically, the ***LaelapsHalt()*** (located in ***constvel.c***, under *Planners*) immobilizes the quadruped smoothly, whenever the master requests it. To define how smooth this transition should be, modify the **HALT_TRANSITION** symbol (in [s]), located at the same source file. Also in here, by calling the ***UpdateTrajectoryParameters()*** the trajectory-related parameters of the application are refreshed and the parameter saturation feature (Section 3.2.2) is materialized. Finally, the CLA or CPU planner is called, if the **CLA_PLAN** symbol is defined or not, respectively (see Section 3.5.7). The ***ConstVelApp()*** routine that materializes the described procedures is illustrated in the following code snippet.

```
void ConstVelApp(void)
{
    //
    // Update State Machine Variable
    //
    g_StateMachine = (Buttons0x7000.State_Machine > 0);

    //
    // Calculate EtherCAT Cyclic Loop's Period in [s]
    //
    const uint32_t CycleTime = sSyncManOutPar.u32Sync0CycleTime;
    const float dt = (float)CycleTime / 1000000000.0f;

    //
    // Update Control Related Parameters
    //
    UpdateControlParameters(dt);

    //
    // Update Debug LEDs States
    // -- MISRA Rule 10.8 Violation (Reviewed): Type castings of composite
    // expressions of different or wider type category are allowed in this
    // project, since the compiler is fixed and implementation specific.
    //
    GPIO_WritePin(31U, (uint16_t)!(Buttons0x7000.Blue_LED > 0U));
    GPIO_WritePin(34U, (uint16_t)!(Buttons0x7000.Red_LED > 0U));

//
// End-Stop Routines are enabled
//
#if defined(END_STOP)

    //
    // Calculate Virtual Spring's Control Effort Contribution
    // -- Static Analysis Warning Reviewed
    //
    spring_endstop.FvsCalc(&spring_endstop);
#endif  // END_STOP

    //
    // State Machine is Enabled (Operational State)
    //
    if (g_StateMachine == true)
    {
        //
        // Increment Time
        //
        g_t += dt;

        //
        // Calculate Step-Invariant Time Variable in [s]
        //
        if ((g_TrajTf + g_TrajTs) > 0.0f)
        {
            g_tMod = fmodf(g_t + g_TimePhase, g_TrajTf + g_TrajTs);
```

```
        }

        //
        // Make Step's Time Zero if Step's Period is Zero
        //
        else
        {
            g_tMod = 0.0f;
        }

        //
        // Check Laelaps Motion Mode
        //
        MotionModeControl(dt);

//
// CLA Planner Task is used
//
#if defined(CLA_PLAN)

        //
        // Enable EALLOW Register Access
        // -- MISRA DIR1.1 Violation (Reviewed): Justified use of Assembly
        //
        asm(" EALLOW");

        //
        // Call Planner's CLA Task (Task 2)
        //
        Cla1ForceTask2();

        //
        // Disable EALLOW Register Access
        // -- MISRA DIR1.1 Violation (Reviewed): Justified use of Assembly
        //
        asm(" EDIS");

//
// CLA Planner Task is not used
//
#else

        //
        // Call CPU's Constant Velocity Planner
        //
        ConstVelPlanner2();
#endif  // CLA_PLAN

    }

    //
    // State Machine is Disabled
    //
    else
    {
        //
        // Reset Time Variable
        //
        g_t = 0.0f;
    }

    //
    // Update Master Input PDOs
    //
    UpdateMasterInputs();

}   // End Of ConstVelApp()
```

The same principles apply in the case of the former planner implementation [20]. Its firmware structure is modified to comply with the one above for consistency.

CLA Planner Task

If the CLA planner is used, the CPU activates a software trigger to the CLA. The planner's code is executed inside CLA's task 2. This task, with lower priority compared to the control's one (task 1), is responsible for the planning and inverse kinematics procedures. It is triggered at the rate of the EtherCAT cycle (default 2.5 kHz) by a software trigger (**IACK** command). It may be viewed in the following code snippet.

```
__interrupt void Cla1Task2(void)
{

//
// CLA Planner Task is used
//
#if defined(CLA_PLAN)

//
// Constant Velocity Planner is used
//
#if defined(CONSTVEL_PLANNER)

    //
    // Call CLA's Constant Velocity Planner Routine with Ground Collision
    // Avoidance Feature.
    //
    ConstVelPlanner2();

//
// Elliptical Planner is used
//
#elif defined(ELLIPTICAL_PLANNER)

    //
    // Call CLA's Elliptical Planner Routine with Ground Collision
    // Avoidance Feature.
    //
    EllipticalPlanner();
#endif  // CONSTVEL_PLANNER

    g_ClaCounter2++;
#endif  // CLA_PLAN

}   // End Of Cla1Task2()
```

3.5.4 EtherCAT Communication

The EtherCAT stack is responsible for the whole slave setup, which involves all the data structures and communications between the *TMS320F28379D* microcontroller and *ET1100* EtherCAT controller. Its main part is generated automatically using ETG's Slave Stack Code tool (SSC) [38].

The user or the supervisory logic that controls the robot's operation must be able to modify the step's parameters of each leg and adjust the low-level controllers based on the intended whole-body motion. As in every control architecture, feedback is necessary to assess the current state of the robot and make decisions for the next ones. To this end, the EtherCAT network has to circulate these parameters and update the PDO objects of all nodes in the network.

The input and output variables of the project can be monitored via EtherCAT communication. Each variable has a specific type (e.g. BOOL, INT), belongs to a Record (general address) containing more variables of identical or different type and has a unique name within the Record. Output variables are those which are controlled and determined by

the master node during the execution of the stack and their index always begins with 0x70, while input variables are designated by each slave and their index always begins with 0x60. They are configured in *EtherCAT_Laelaps_Motion_ControlObjects_CONSTVEL.h* (under the *stack* folder) and may be reviewed in Table 3-3 and
Table 3-4.

Table 3-3. Master's input process data objects.

Index	Sub-Index	Data Type	Name	Comments
0x6010	**Record**		**hip angle**	
	0x01	INT16	hip_angle	Rotational angle of hip [deg] * 100
	0x02	INT16	Desired_hip_angle	Desired rotation angle of hip [deg] * 100
0x6012	**Record**		**Feedback Time**	
	0x01	UINT16	Time	Time variable from slave device (t in (3-8)) [s]
0x6014	**Record**		**knee_angle**	
	0x01	INT16	knee_angle	Rotational angle of knee [deg] * 100
	0x02	INT16	Desired_knee_angle	Desired rotation angle of knee [deg] * 100
0x6020	**Record**		**Commands**	
	0x01	INT16	PWM10000_knee	Output of PIV control for knee [%] * 100
	0x02	INT16	PWM10000_hip	Output of PIV control for hip [%] * 100
0x6030	**Record**		**Velocity**	
	0x01	INT32	velocity_knee1000	Rotational speed of knee [rad/s] * 1000
	0x02	INT32	velocity_hip1000	Rotational speed of hip [rad/s] * 1000

Table 3-4. Master's output process data objects.

Index	Sub-Index	Data Type	Name	Comments
0x7000	**Record**		**Buttons**	
	0x01	BOOL	State_Machine	State machine variable (Figure 3-26)
	0x02	BOOL	Initialize_clock	Not used

	0x03	BOOL	Initialize_angles	Not used
	0x04	BOOL	Inverse_Kinematics	Not used
	0x05	BOOL	Blue_LED	On-board blue LED switch
	0x06	BOOL	Red_LED	On-board red LED switch
	0x07	BOOL	LAELAPS_HALT	Stand-still mode switch
	0x08	BOOL	Reverse_Dir	Reverse robot's direction
	0x09	BOOL	Transition_time	Time for smooth transition functions [s]
0x7010	**Record**		**Desired_x_value**	
	0x01	INT32	Desired_x_value	Variable for future use
0x7012	**Record**		**TargetMode**	
	0x01	UINT16	FilterBandwidth	1st order filter's f_{cbw} in (3-22) [Hz]
0x7014	**Record**		**Desired_y_value**	
	0x01	INT32	Desired_y_value	Variable for future use
0x7020	**Record**		**ControlGains**	
	0x01	INT16	Kp100_knee	Proportional gain of the knee's controller ($K_p \cdot 100$ in (3-20))
	0x02	INT16	Kd1000_knee	Velocity gain of the knee's controller ($K_d \cdot 1000$ in (3-20))
	0x03	INT16	Ki100_knee	Integral gain of the knee's controller ($K_i \cdot 100$ in (3-20))
	0x04	INT16	Kp100_hip	Proportional gain of the hip's controller ($K_p \cdot 100$ in (3-20))
	0x05	INT16	Kd1000_ hip	Velocity gain of the hip's controller ($K_d \cdot 1000$ in (3-20))
	0x06	INT16	Ki100_ hip	Integral gain of the hip's controller ($K_i \cdot 100$ in (3-20))
0x7030	**Record**		**TrajectoryParameters**	
	0x01	INT16	x_cntr_traj1000	x center of the semi-ellipse (x_c in (3-11)-(3-12)) [mm]
	0x02	INT16	y_cntr_traj1000	y center of the semi-ellipse (y_c in (3-11)-(3-12)) [mm]

0x03	INT16	a_ellipse100	Amplitude of ellipse's x-axis (a in (3-11)-(3-12)) [cm]
0x04	INT16	b_ellipse100	Amplitude of ellipse's y-axis (b in (3-11)) [cm]
0x05	INT16	traj_Tf100	Swing-phase period (T_{swing} in (3-8)) [s / 100]
0x06	INT16	phase_T100	Phase shift (Δt_{phase} $[\%T_{step}]$ in (3-8))
0x07	INT16	traj_Ts100	Stance-phase period (T_{stance} in (3-8)) [s / 100]

The PDO variables are handled by several functions of the project; the most significant of these functions are triggered by the DC Unit of the ESC (see Section 2.2.4). Any user-code related to the application should be called by the following routines. The code is extensive and may be reviewed in *EtherCAT_Laelaps_Motion_Control.c* source file, under the project's *SPI_EtherCAT_slave_stack* folder.

APPL_GenerateMapping() declares the PDO data sizes and is configured by measuring how many bytes each PDO tethers.

```
UINT16 APPL_GenerateMapping(UINT16 *pInputSize, UINT16 *pOutputSize)
{
    //
    // Master Input PDOs # bytes
    //
    *pInputSize = 22U;

    //
    // Master Output PDOs # bytes
    //
    *pOutputSize = 38U;
    return ALSTATUSCODE_NOERROR;

}   // End Of APPL_GenerateMapping()
```

APPL_InputMapping() performs the transfer of the master's input variables from Delfino's local memory to the ESC's.

APPL_OutputMapping() performs the transfer of the master's output variables from the ESC's memory to Delfino's.

APPL_Application() is the main function which is used for user-specific code and materializes the main application. It is called in a cyclic manner by the ***Sync0_Isr()*** (DC Mode) or in the stack's ***Mainloop*()** routine, if no real-time execution is configured (Free Run). Its code implementation is presented in the following code snippet.

```
void APPL_Application(void)
{

//
// Elliptical Planner is used
//
#if  defined(ELLIPTICAL_PLANNER)

    //
    // Call Elliptical Planner's Application
    //
    EllipticalApp();
```

```
//
// Constant Velocity Planner is used
//
#elif  defined(CONSTVEL_PLANNER)

    //
    // Call Constant Velocity Planner's Application
    //
    ConstVelApp();

#endif  // ELLIPTICAL_PLANNER

}   // End Of APPL_Application()
```

3.5.5 Interrupt Priorities

In an EtherCAT slave, the ESC triggers the interrupts with external signals (XINT interrupts) when the former is configured to operate in DC synchronization. Section 2.2.4 pointed out that there are three ISRs that handle the major operations of EtherCAT, namely ***Sync0_Isr()***, ***Sync1_Isr()*** and ***PDI_Isr()***. These interrupts have the following mapping to the routines mentioned in the previous section, when the slave operates in DC Synchronization mode:

- ***Sync0_Isr()*** → ***APPL_Application()***
- ***Sync1_Isr()*** → ***APPL_InputMapping()***
- ***PDI_Isr()*** → ***APPL_OutputMapping()***

In this section, their functionality is not discussed, but they are investigated regarding their interrupt priorities. According to the HAL configuration provided by TI (ControlSuite's *TMDSECATCND379D_V1.0* development kit [61]), the aforementioned ISRs are mapped to XINT5 (INT12.3), XINT4 (INT12.2) and XINT1 (INT1.4), respectively. Table C-1 (Appendix C) states that their core priorities are 16, 16 and 5, respectively. This would create a problem in the current application. The XINT1 interrupt, as can be seen in Figure C-5 (Appendix C), has very high priority (5.4), higher than that of the control loop's timer 0 (5.7). However, a major requirement of the current firmware is to execute the control loop with the highest determinism possible. To account for this matter, the ***PDI_Isr()*** trigger has been modified to XINT3 (INT12.1). This implicates that the control loop has now higher interrupt priority compared to the EtherCAT ISRs. This means that the motion control loop is promoted and runs in a hard real-time context. To verify this, the procedure of Section 3.5.6 was executed. To conclude, all of the ISR priorities are set as follows.

- ***Sync0_Isr()*** has 16.3 priority.
- ***Sync1_Isr()*** has 16.2 priority.
- ***PDI_Isr()*** has 16.1 priority.
- Control ISR has 5.7 priority.

In the previous configuration of the firmware, the two controllers had been triggered by separate ePWM ISRs, namely EPWM1 (INT3.1) and EPMW2 (INT3.2) (Figure C-5, in Appendix C). These ISRs have 7.1 and 7.2 priorities, respectively. In other words, the previous knee's controller (7.1) had a higher priority than the one of the hip (7.2), while both of them had lower priorities compared to the previous ***PDI_Isr()*** (5.4), but higher than those of the other EtherCAT-related ISRs (16.2 and 16.3). So, in the current configuration there is an explicit dependency regarding the priority of the control and EtherCAT-related routines. This was not the case in the previous implementation.

The modification of the timer in HAL was made, since few peripherals can compete the TI's default interrupt priority assigned to the ***PDI_Isr()***; in Figure C-5 (Appendix C), only some ADCs have higher priority than XINT1 interrupt. Also, the timers 1 and 2 have very low core

priorities, since they belong to the last two PIE groups. To understand the logic of the C2000 interrupt architecture refer to Section C.3. Note that the priorities apply only in the case that more than one interrupts are being triggered in the same CPU cycle. Interrupt nesting is not allowed by default. The ISRs that were not serviced and possible others that are triggered during the servicing of the interrupt are stacked. After completion, the highest pending ISR will eventually be serviced.

The described behavior implicates that each ISR is serviced to completion and as a consequence servicing the time-critical control ISR may be delayed if it is triggered during the execution of the EtherCAT-related ISRs. According to Section C.3.2, nesting can be enabled in software with specific commands inside the ISR in which, servicing other interrupts is intended. This way, specific ISRs may be enabled, utilizing the activated software interrupt nesting. However, in the current case results have indicated that there are data access conflicts when the EtherCAT ISRs are interrupted. In the future, the tasks that cause such problems should be located in the code. This way, the interrupt nesting may be activated after their execution. This analysis is extensive and is left for future work.

Another way to approach this matter is to decouple the control loop from the CPU. By consulting Section C.4, it could be argued that the CLA can handle every peripheral on the device. Currently, it handles only the PWM. If the eQEP modules were added, the control ISR would have been redundant, since the CLA could have been triggered in hardware by any peripheral. This way, the interrupt priorities are bypassed, since timer 0 hardware signal to the CLA does not interfere with the CPU at all. Since task 1 (CLA control task) has the highest possible task priority, the control loop becomes fully deterministic. This could have been implemented in the current version, but there is a conflict regarding how CPU and CLA represents pointers in memory.

Pointers are interpreted differently on the C28x CPU and the CLA. The CPU treats them as 32-bit data types (address bus size being 22-bits wide can only fit into a 32-bit data type), while the CLA only has an address bus size of 16 bits. The eQEPs are currently handled by the **POSSPEED** structure, located in *posspeed.h*, under *hal* folder. This structure was introduced in all of the previous firmware versions, and it is the TI's proposed way in the available examples of the C2000Ware library. It is a compact way to manage and store the quantities that are calculated from the eQEP readings, like the angular velocity approximation, but contains pointers. However, if in a future version the previously described pointer conflict is solved, the control loop could run in a definitive fully-deterministic manner. One solution is discussed in [68]. Note that this architecture may result in data access conflicts that are used simultaneously by the CLA and CPU. This must be taken into account when designing the proposed setup.

To conclude, in the current implementation, the control loop is promoted, having higher interrupt priority. However, since the interrupt nesting is not supported, the problem only applies if an EtherCAT-related ISR is being serviced at the time that timer 0 triggers the control ISR. Nevertheless, in the current configuration, it is ensured that the control interrupt will be serviced first, after the completion of the running ISR. Furthermore, the controllers have the same priority in contradiction with the previous versions that were inconsistent in this regard. Lastly, they run in a very high frequency, significantly higher than the one of EtherCAT's ISRs and no disruptions in the system's response were observed due to possible latencies, at the test phase of the firmware.

HAL Modifications

In the previous sections the whole firmware has been described. As may be understood, the firmware consists of the EtherCAT stack, generated by the SSC tool and the TI's default HAL implementation (ControlSuite's *TMDSECATCND379D_V1.0* development kit [61]). To account for the problems that this implementation came with (see previous section), the HAL was modified.

The first modification concerns the use of timer 0 triggering the control ISR. EtherCAT stack uses by default this timer as a free counter and thus, no modifications are necessary. Nevertheless, to decouple the two processes, the timer 0 has been replaced with the timer 1 in the following HAL routines:

- ***ESC_getTimer()***
- ***ESC_clearTimer()***

The second modification concerns the ***PDI_Isr()*** external interrupt. The XINT1 interrupt of the TI's implementation was changed to XINT3 for the reasons that were analyzed in the previous section. This implicates certain changes in the Delfino's XBar module [64]. The described adjustment is illustrated in Figure 3-27.

TI's default

```
//
// Enable EALLOW Register Access
//
EALLOW;

//
// Remap XINT1 ISR to ESC_applicationLayerISR
//
PieVectTable.XINT1_INT = &ESC_applicationLayerISR;

//
// Setup SPI XBar Options
//
InputXbarRegs.INPUT4SELECT = ESC_SPI_INT_GPIO;
GPIO_SetupPinOptions(ESC_SPI_INT_GPIO, GPIO_INPUT,
                     GPIO_PULLUP | GPIO_ASYNC);

//
// Falling Edge Interrupt
//
XintRegs.XINT1CR.bit.POLARITY = 0x0;

//
// Setup XINT1 Configuration Register
//
XintRegs.XINT1CR.bit.ENABLE = 1;

//
// Disable EALLOW Register Access
//
EDIS;
```

Modified

```
//
// Enable EALLOW Register Access
//
EALLOW;

//
// Remap XINT3 ISR to ESC_applicationLayerISR -- MISRA VIOLATION
//
PieVectTable.XINT3_INT = &ESC_applicationLayerISR;

//
// Setup SPI XBar Options -- MISRA VIOLATION (Reviewed)
//
InputXbarRegs.INPUT6SELECT = ESC_SPI_INT_GPIO;
GPIO_SetupPinOptions(ESC_SPI_INT_GPIO, GPIO_INPUT,
GPIO_PULLUP | GPIO_ASYNC);

//
// Falling Edge Interrupt
//
XintRegs.XINT3CR.bit.POLARITY = 0x0;

//
// Setup XINT1 Configuration Register
//
XintRegs.XINT3CR.bit.ENABLE = 1;

//
// Disable EALLOW Register Access
//
EDIS;
```

Figure 3-27. PDI external interrupt modification.

3.5.6 Firmware's Performance

Timing Considerations

In this section, the execution time of each ISR is measured. To this end, the HAL's timer function ***ESC_getTimer()*** was used at the starting point and at the end of each ISR. Then, their execution time Δt_{CPU} was calculated with (3-23), with n_{start} being the timer reading at the beginning of the ISR and n_{end} the one at the end. The results are listed in Table 3-5.

$$\Delta t_{CPU} = \frac{n_{end} - n_{start}}{f_{clock}}, \text{ with } f_{clock} = 200 \text{ [MHz]} \tag{3-23}$$

Table 3-5. Firmware ISRs' execution time.

Routine	Execution Time [µs]
ClaControl_Isr()	2.040
Sync0_Isr	17.085
Sync1_Isr	18.805
PDI_Isr	131.700

From the results of Table 3-5, it becomes clear that the control ISR is significantly faster compared to the others. This is partially due to the use of the CLA. The ISR in the current configuration stores data and triggers the CLA, with both being tasks that the CPU can execute fast. Another interesting thing is that the ***PDI_Isr()*** takes significantly more time compared to the others and surpasses the control loop's period (100 µs). This means that the control ISR is likely to be triggered during the execution of ***PDI_Isr()*** and consequently become a pending interrupt. To verify that this does not cause significant problems, the firmware has been tested extensively. Also, the procedure of the next sub-section comes as a profiling to determine the interrupt triggering period of each ISR.

Interrupt Profiling

To measure the triggering period of the firmware interrupts, a vector was placed at the beginning of each ISR. In this vector, a number of consecutive timer counts was stored, using the ***ESC_getTimer()*** routine. This procedure can be understood intuitively in the following code snippet. Note that the vectors and variables used for this procedure are removed from the mainline version of the firmware.

```
interrupt void ESC_applicationLayerISR()
{
    if(g_SwitchProf == 1U)
    {
        g_SwitchProf = 2U;
        g_InterISR = true;
        CpuTimer2Regs.TIM.all = 0;
    }

    if(bDcSyncActive == 1U && g_InterISR == true)
    {
        g_CounterPDI++;
        if(g_CounterPDI <=2)
        {
            g_PDIStart[g_CounterPDI-1] = ~((uint32_t)(CpuTimer2Regs.TIM.all));
        }
    }

#ifdef ETHERCAT_STACK
    PDI_Isr();
#endif

    PieCtrlRegs.PIEACK.all = PIEACK_GROUP12;
    if(bDcSyncActive == 1U && g_InterISR == true)
    {
        if(g_CounterPDI <=2)
        {
            g_PDIEnd[g_CounterPDI-1] = ~((uint32_t)(CpuTimer2Regs.TIM.all));
        }
    }
}
```

The results can be reviewed in Figure 3-28. Each ISR returns to the Main-Loop after completion. This adds return and call latency, for example at each control ISR right after the completion of ***PDI_Isr()***. The EtherCAT-related ISRs have insignificant delay that can be justified, if the increased overhead that the external interrupts present (refer to [64]), compared to other interrupt sources, is taken into account.

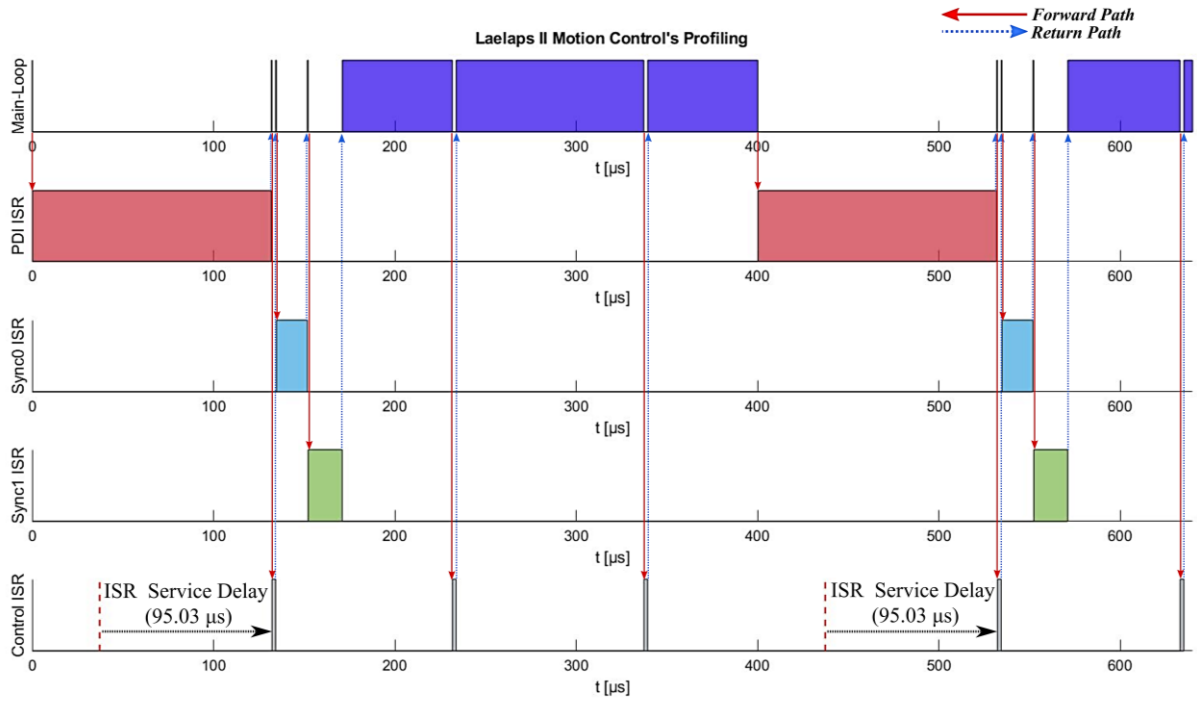


Figure 3-28. Interrupt profiling results.

On the other hand, the control ISR has insignificant run-time (2.04 μs), but its execution is not strictly deterministic. In Figure 3-28 a pattern is observed; every four ISRs, one is delayed for 95.03 μs. This delay is significant compared to the desired cycle of the control loop (100 μs). This implies certain flunctuations in the motion control's bandwidth, since in a digital control system, the sampling frequency is associated with the system's bandwidth (Nyquist-Shannon theorem [66]). Currenlty the robot does not operate at such high frequencies and its motions are not disrupted by the aforementioned control frequency's drops. Nevertheless, this fact should be considered in future designs.

Lastly, the EtherCAT PDOs are updated by the ***PDI_Isr()***, but the application variables, like the control parameters are refreshed during the ***Sync0_Isr()***. This means that the delayed control ISR, being serviced right after the ***PDI_Isr()***, runs with the control parameters that were updated in the the previous EtherCAT cycle.

CLA Performance Gain

This section is dedicated to the achieved performance. The benchmark consisted of a CPU timer measuring the elapsed time of each CLA task's beginning to completion. The results are compared with the corresponding timings achieved by the older setup. The acquired data are presented in Table 3-6. Also, the main function run-times are measured for future reference and completion.

Table 3-6. Firmware's performance overview.

Processor	Routine	Execution Time [μs]	Call Overhead [μs]
CPU	Planner	3.02	-
	Controllers	3.29	-
CLA	Planner	2.76	0.14
	Controllers	1.39	0.135

From Table 3-6, one may deduce that the actual CPU gain from CLA usage is at minimum 6.31 μs. That is not a significant reduction, but through integrating this module into Laelaps firmware, experience and documentation have been acquired. The robot's current firmware is still under development and its present processing power requirements are relatively low. With time, the software will eventually evolve along with its needs, while CLA, IPC for dual-core functionalities and DMA may make the actual difference. At last, it should also be noted that both of the routines are running faster on CLA than on CPU.

3.5.7 Managing the Project's Configurations

From the previous sections, it should be obvious that there are multiple configurations of the project that are managed with the *Predefined symbols* in the *Project Properties* pane. In the above sections, each time that a configuration was referenced, the corresponding symbols were referenced too. These symbols can be managed in project's properties *(Right Click on Project's Name -> C2000 Compiler -> Predefined Symbols)*, while a list is given in Table 3-7.

Table 3-7. Firmware's Predefined Symbols List.

Predefined Symbol Name	Serves	Comments
CPU1	System	Choose run-time CPU
ABS_DEBUG	User Code	Send absolute encoders readings via EtherCAT for debug purposes instead of velocities
CLA_CTRL	User Code	Execute controllers with CLA instead of FPU
END_STOP	User Code	Enable end-stop routines
CLA_PLAN	User Code	Execute planner with CLA instead of CPU
CONSTVEL_CONTROLLER	User Code	Use the constant velocity planner
ELLIPTICAL_CONTROLLER	User Code	Use the elliptical planner (previous)
LEFT_LEG	User Code	Left-leg slave configuration
RIGHT_LEG	User Code	Right-leg slave configuration
AUTO_STARTUP	User Code	Use automatic initialization for the encoders
INTERFACE_SPI	EtherCAT Stack	Choose SPI as slave's PDI
_LAUNCHXL_F28379D	System	LaunchPad firmware configuration
USE_SPIA	EtherCAT Stack	Choose SPI channel for PDI use
FLASH	System	Initialization from FLASH

RAM	System	One time initialization from RAM
ETHERCAT_STACK	EtherCAT Stack	Enable the prescribed EtherCAT Stack code files
CLA_C / CLA_P	Linker	Create CLA-specific linker sections

As for the **CLA_C / CLA_P** symbols, they should be specified in the Project's properties pane *(Right Click on Project's Name -> Advanced Options -> Command File Preprocessing)*. They manage the memory allocation in the linker file if CLA tasks are used. More about the command file will be discussed in the next section.

Major Project Configurations

To define the most significant options and configure the major aspects of the project, the following steps must be followed.

1. There are four different project configurations. Two for the left leg, and two more for the right. Each leg has a debug configuration, in which the firmware is stored in RAM. The other one is a Flash configuration that constitutes the release version of the application. Generally, during experiments, the Flash configuration should be used. These options are demonstrated in Figure 3-29. .

Figure 3-29. Different leg firmware configurations.

2. To further exploit and manage the different functionalities that the project comes with, certain Preprocessor and Linker symbols have to be defined (Section 3.5.7). Figure 3-30 gives an overview of each symbol's use. The symbols may be left to default if no special configuration is desired. If the readings of absolute encoders during tests need to be displayed, the **ABS_DEBUG** symbol has to be defined.

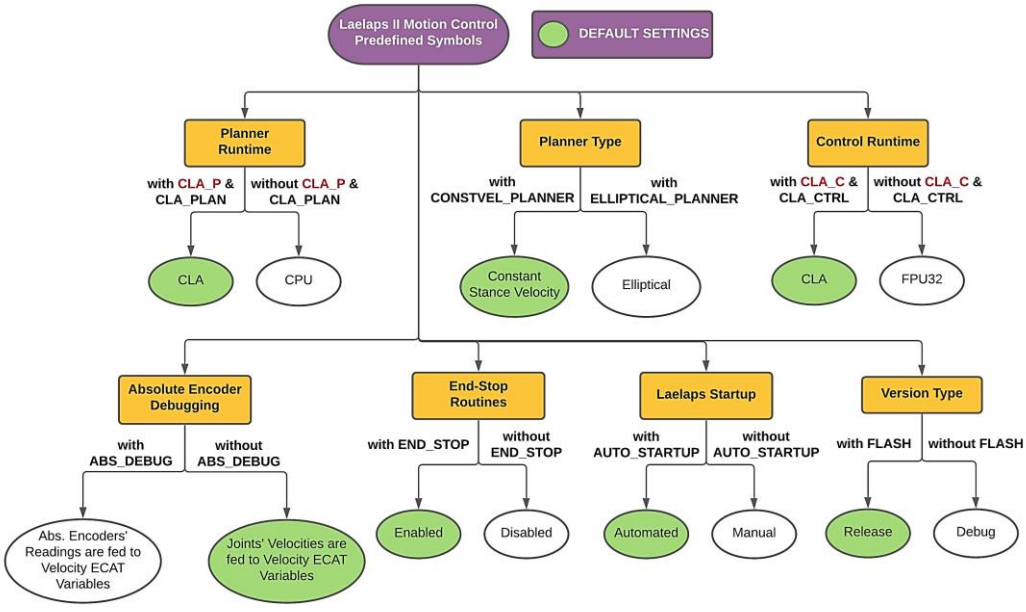


Figure 3-30. Project's configurations.

The above configurations can be defined from the tabs displayed in Figure 3-31.

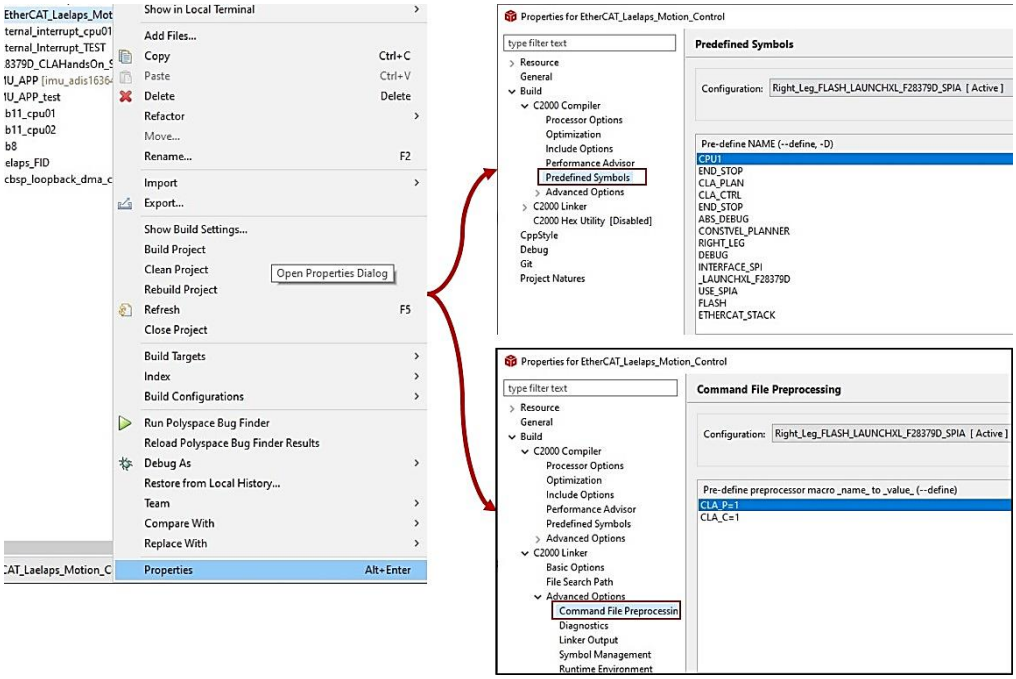


Figure 3-31. Predefined Symbols and Linker Symbols tabs overview.

3. This configuration step involves the settings of the absolute encoders' sampling time to reduce ADC sampling's inherent noise. To modify this parameter, navigate to *posspeed.h* (under *hal* folder) and change **ABS_ENC_SAMPLING_TIME** to the desired sampling time in [s] (Section 3.5.1).
4. Another setting that requires attention is each controller's saturation limit. To modify them, change the corresponding global variables (g_UmaxHip / g_UmaxKnee). Refer to Section 3.5 for further information on configurations. The valid range is 0.0 to 1.
5. Next, configure the end-stop routines, by changing the settings displayed in Figure 3-23 if the defaults do not suffice.
6. Finally, the source files of the controllers are located inside the *hal* folder. Depending on the desired configuration and the previously set Predefined Symbols (**CLA_CTRL** and **CLA_C**), only the corresponding planner should be included in the building process. To exclude the others, *Right-Click* on their source files and opt for *Exclude from Build* option.

After the above configurations, the firmware along with the applied modifications can be deployed. For further details refer to Appendix E (Section E.3).

3.5.8 Firmware's Linker Command File

For the linking process, partially described in Section C.7, a custom command file was created to optimize the resources and integrate CLA and DCL capabilities. The command file comes in two flavors. One in RAM run-time configuration and another in FLASH configuration. These files may be found under the *cmd* folder, inside the project tree, namely *2837x_RAM_lnk_cpu1_CLA.cmd* and *2837x_FLASH_lnk_cpu1_CLA.cmd*. To view the memory allocation designated by those files, navigate to *View->Memory Allocation* at TI's CCS menu ribbon. It should be noted that in order to change the available stack and heap sizes, one should navigate to *Project Properties (Right Click on Project's Name) -> Basic Options* as in Figure 3-32. In the current firmware version, the assigned heap and stack sizes were chosen

to be close to the highest possible dictated by the command file's memory allocation. Currently there are no specific requirements regarding these parameters. They should be tuned to achieve an optimal result in a potential future version that will have more specific requirements.

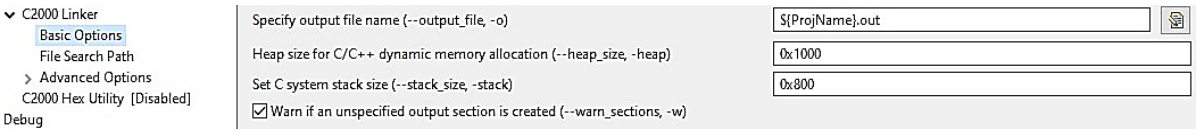


Figure 3-32. Stack & Heap size setup pane.

Note that the CLA scratchpad section configured in the custom command file, along with its size, is similar to the CPU's stack. The CLA C compiler stores all the local, auto variables, and function arguments to this logical memory section. Again, there were no specific requirements for its size and thus a relatively high value (0x800) compared to the TI's default (0x100, in the original C2000Ware's *2837xD_FLASH_CLA_lnk_cpu1.cmd*) was chosen.

Last but not least, specific **#pragma** instructions have been used throughout the code, to dictate to the compiler, which time-critical functions should be loaded to RAM during the firmware's startup. This is a good practice, due to the proven faster execution capabilities of RAM's routines. Generally, it is a trade-off between additional data space and code execution time, which should be considered during the design stage of the application. In the current firmware, the control ISR is executed on RAM, since it is called in high frequencies and its execution should be as fast as possible. Moreover, TI has configured in its original HAL implementation (ControlSuite's *TMDSECATCND379D_V1.0* development kit [61]) specific, time-critical functions to be executed on RAM.

3.5.9 Compiler Information

The firmware uses the TI's default C28x compiler *v20.2.4LTS***.** It is the standard compiler for the C2000 Delfino microcontrollers. The compiler optimization level is currently left to 0 (none). In a future version, a higher level could be applied, but with due care, since it can lead to heavily erroneous firmware run-time. These options are located inside the *Project Properties* pane (*Right Click on Project's Name->Properties*).

The C28x compiler is used to build CPU source files (*.c*) along with their headers. For the CLA, a different compiler is used that builds all the *.cla* files of the project along with their headers. These headers may be included in both *.c* and *.cla* files. If there are variables that either compiler should not "see", the developer could create an **#ifdef** structure with the symbol **__TMS320C28XX_CLA__**.

3.6 Firmware Check and Verification

A significant part of any software development process should be dedicated to ensuring that the code is free from glitches. Bugs impose a major threat for the system and its users. Specifically, the Laelaps II is a high-power machine with a primary target to operate in a wide range of environments, while interacting with people during tests. Hence, the need for assurances of a clean and bug-free code is imperative.

Another aspect of such processes is to create an application that would be easily understood and maintained by a third person. For this reason, certain standards and rules must be followed. The introduction of such protocols is a vital part of modern workflow approaches in software development, along with the preservation of a strict version control

system. Also, to assist further the maintenance and scalability of the code, documentation should be created. In the current case, the firmware's description has been generated with Doxygen. The resulting *HTML* may be inspected in the project's Bitbucket repository [52].

The CSL-EP's Legged Robots Team has adopted such procedures to ensure that every application is safe and ready to play. Likewise, in the present thesis, the developed firmware is written in compliance with the industry's leading safety-critical and security standard, MISRA C: 2012 [69]. Of course, there are certain deviations to adjust the standard to the current project's needs. More on the MISRA compliance are discussed in Appendix H (Section H.1). Note that only the herein developed code complies with the formerly mentioned coding style (Appendix G) and MISRA standard. TI's HAL and ETG's EtherCAT stack source files have their own coding style and were not revised.

The code is analyzed with industry's leading code analysis tools that guarantee zero False Negative warnings. This means that the code becomes immune to any run-time errors due to overflows, illegal function calls, and divisions by zero, etc. The application has been reviewed and relieved from the reported bugs. In Table 3-8, the results of the executed analysis are presented, taking into consideration the defects illustrated in Figure 3-33. Only the custom files have been revised, because the additional TI's libraries and EtherCAT stack had already been tested by their respective distributors. The listed defects of the custom source files are mostly False Positives of the static analysis tool and data type conversion warnings. Nevertheless, they are not considered as defects in the current firmware and were left on purpose. In the source files, every violation that was identified by the Static Analysis but was not modified has been documented in comments. For many more details, refer to the auto-generated reports which accompany the software, located in the respective Bitbucket repository [52].

Figure 3-33. Possible defects, the Static Analysis may find.

Table 3-8. Static analysis results.

File	Status	Defects
constvel.c	Rev.	2
elliptical.c	Rev.	1
planners.h	Rev.	2

EtherCAT_Laelaps_Motion_Control.c	Rev. (Only custom entries)	9
etherCAT_slave_c28x_hal.c	Rev. (Only custom entries)	11
posspeed.c	Rev.	15
posspeed.h	Rev.	3
Total		**43**

3.7 Conclusion

In this chapter, the Laelaps II motion control firmware is described and analyzed thoroughly. The newly introduced implementation improves many aspects of the former ones by adding several features and exploiting Delfino's architecture. The introduction of MISRA C:2012 coding standard facilitates the overall development process and in combination with the adopted static analysis tools, many bugs have been eliminated (see Section 3.6). Also, the automation of the startup sequence and the introduced firmware safety features can accelerate the experiments with the robot and prevent possible failures. Overall, the firmware fulfills the real-time requirements that were set throughout the development process with many test results indicating it (see 3.5.6). Note that the delay of the control ISR should be accounted and solved in a future version. Nevertheless, no problems were observed during the system's operation.

4 Inertial Measurement Units

An Inertial Measurement Unit (IMU) is an electronic device that measures and reports a body's acceleration, angular rate and sometimes the surrounding magnetic field, using a combination of accelerometers, gyroscopes and sometimes magnetometers. This technology is mature enough to be broadly applicable in many technological fields; from applications in everyday life (e.g. smartphones) to aerospace, high-accuracy systems. In robotics, IMUs are essential in state estimation and without them, autonomy would not be feasible.

The CSL-EP Legged Robots Team has at its disposal two Analog Devices IMUs, namely ADIS16364 and ADIS16375. They are two high-precision 6-DOF inertial units, combining data from 3 embedded angular accelerometers and gyroscopes. Each sensor incorporates industry-leading iMEMS technology with signal conditioning that optimizes its dynamic performance. The communication between ADIS163xx and Delfino is established through the standard SPI protocol. According to their datasheets [70] [71], they present an exceptional dynamic response in acceleration and angular velocity changes and can withstand and measure severe impacts. Therefore, the integration of these systems into the Laelaps II robot is a big step towards autonomy and awareness of its environment.

This chapter investigates the development and incorporation of two IMU slaves to the current Laelaps' EtherCAT network. Initially, some theoretical preliminaries are laid. Furthermore, the used hardware components and electronics are presented. Most of the parts are designed and manufactured in-house with SOTA methods. Moreover, a major section in this chapter is the firmware of each IMU EtherCAT slave and its capabilities. Lastly, the verification of the aforementioned systems is considered.

4.1 Gyroscopes

The goal of a gyroscopic system is to track changes in a body's orientation by taking into account predictable phenomena produced by physical laws [12]. The first mechanical system which materialized this idea was invented by Bohnenberger [72]. Initially, such systems relied on the conservation of the angular momentum of a nearly frictionless rotating wheel like the one illustrated in Figure 4-1.

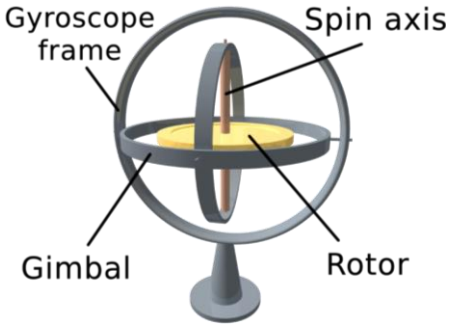


Figure 4-1. Gimbaled gyroscope.

Nowadays, technology has made leaps in the field of Micro Electro-Mechanical Systems (MEMS), introducing small devices with incredible accuracy and almost no moving parts. Commonly, such sensors rely on the transfer of energy among vibratory modes, based on the Coriolis effect [73]. There are plenty of configurations available with the most common being the one of the tuning-fork (Figure 4-2). A crystal vibrates the tines of the fork linearly (driven vibration) and as the whole structure rotates, the Coriolis causes the forks to oscillate out of

the plane. This displacement from the plane of oscillation is measured to produce a signal related to the system's rate of rotation.

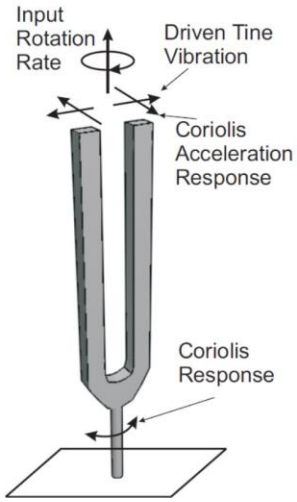


Figure 4-2. Tuning-fork's principle of operation.

4.2 Accelerometers

Accelerometers are simple structures with the most widely known being the one of the mass-spring-damper (m-c-k) with the well-known dynamic behavior. By tuning the m-c-k parameters, the system should reach a stable fixed point whenever a static force is present. A more accurate, small and generally modern setup uses piezoelectric phenomena [74] to measure a small mass' applied forces by translating its displacement into voltage differential.

An IMU package houses all of the above components, along with others that support measurement conditioning and interfacing. It may be visualized as a microcontroller that takes each sensor's measurements, processes them and makes them available to another computing system.

4.3 Serial Peripheral Interface

The Serial Peripheral Interface (SPI) is a synchronous communication protocol used in embedded systems for short-distance communications. SPI devices communicate in full-duplex mode using a master-slave architecture with a single master on the bus. The master device creates a frame for reading and writing. Multiple slave devices are supported through selection with individual Chip Select (CS) lines. To begin communication, the bus master configures the clock, using a frequency supported by the slave device, typically up to a few MHz. During each SPI clock cycle, full-duplex data transmission occurs. The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it (Figure 4-3). This sequence is maintained even when only one-directional data transfer is intended. The use and the realization of this protocol is clarified in Section 4.4.3.

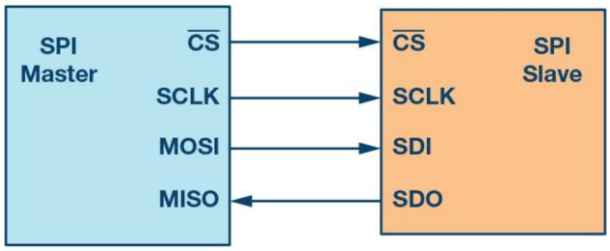
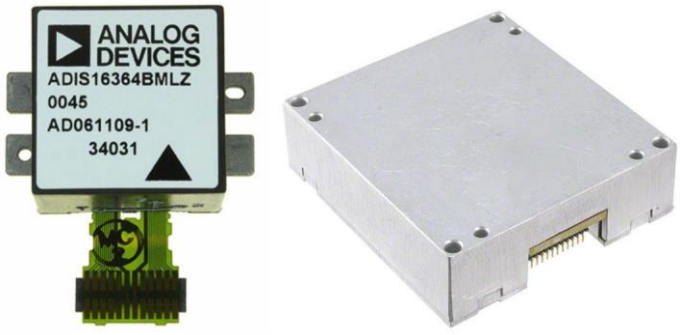


Figure 4-3. Serial Peripheral Interface (SPI) overview.

4.4 ADIS163xx Hardware Description

The ADIS163xx IMUs have specific requirements for mounting and connectivity. To this end, Analog Devices provides tailor-made mounting boards, namely the *ADIS16IMU1,* with robust mounting and easy, plug and play connectivity. The inertial units, along with the mounting breakout board may be reviewed in Figure 4-4 and Figure 4-5.



(a) **(b)**

Figure 4-4. (a) The ADIS16364 IMU. (b) The ADIS16375 IMU.

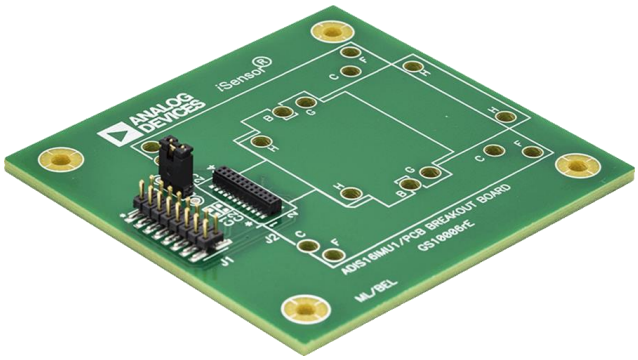
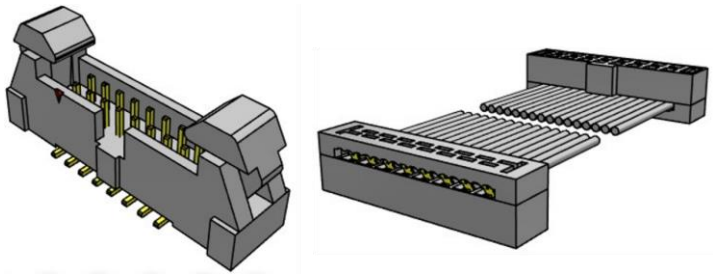


Figure 4-5. The ADIS16IMU1 breakout board.

4.4.1 ADIS16xx Connection with LaunchXL-F28379D

On the discussed breakout board, the connection with Delfino is materialized with an IDC cable, specifically the ***Samtec TCSD-08-D-02.00-01-N-R***. This interface requires an intermediate custom board to handle the IMU signals and power supplies. This board is developed and analyzed in Section 4.4.2. The IDC cable and the connector (***Samtec EHT-108-01-S-D-SM)*** mounted on the previously mentioned custom PCB, are illustrated in Figure 4-6.



(a) (b)

Figure 4-6. (a) Samtec EHT-108-01-S-D-SM. (b) Samtec TCSD-08-D-02.00-01-N-R.

The Analog Devices breakout board rewires the IMU signals to the IDC connector. Apparently, any design should take into account Table 4-1 to avoid mistakes and possible damage to the equipment.

Table 4-1. ADIS breakout board circuit scheme.

CIRCUIT LAYOUT	ADIS163xx	ADIS163xx BREAKOUT BOARD
PIN NAME	**PIN #**	**PIN #**
POWER SUPPLY	10 | 11 | 12	10 | 11 | 12
GND	13 | 14 | 15	7 | 8 | 9
DOUT (MISO)	4	4
DIN (MOSI)	5	6
SCLK	3	2
CS	6	3
DIO1	7	13
DIO2	9	14
DIO3	1	15
DIO4	2	16

4.4.2 IMU and LaunchXL-F28379D Interface Board

As previously stated, ADIS16364 and ADIS16375 are two high-end IMUs that will be exploited in the pose estimation of the Laelaps II quadruped. To integrate them into the robot's architecture, the creation of an interface board along with the required firmware is of utmost importance. This design should follow certain guidelines and rules to create a high-performance sensory system.

Circuit Schematics

The aforementioned intermediate PCB was designed using Autodesk's Eagle software [75]. The created schematics are shown in Figure 4-7. To incorporate only the required components, Delfino's library was dissected to use only the lower half of the board. Also, the chosen step-down regulators were imported from Mouser online electrical symbol repository. The wiring of the created board is illustrated in Table 4-2.

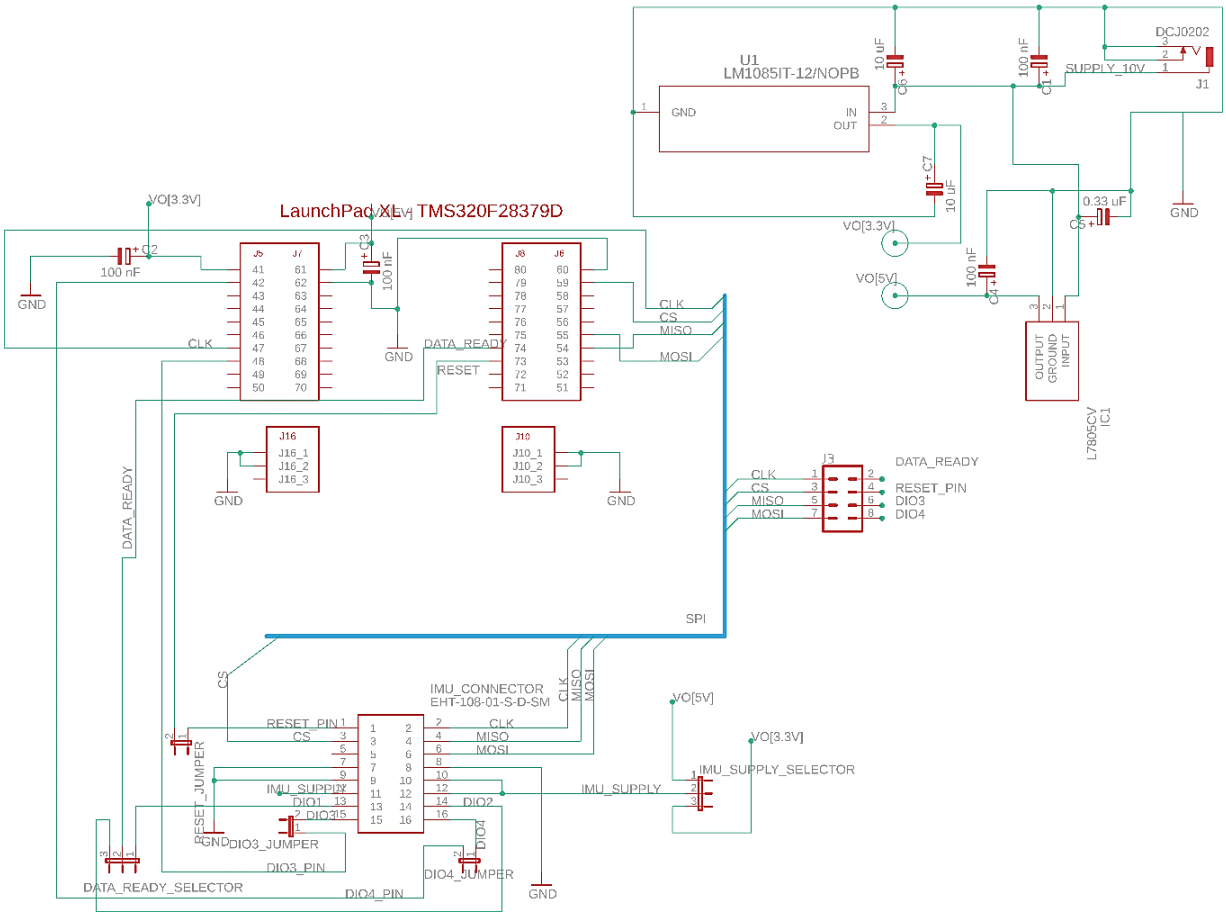


Figure 4-7. Intermediate custom PCB schematics.

Table 4-2. Intermediate-board's signal layout.

CIRCUIT LAYOUT	LAUNCHXL-F28379D	ADIS163xx BREAKOUT BOARD
PIN NAME	***PIN # (HEADER J#)***	***NAME (PIN #)***
POWER SUPPLY	41 (J5) [3.3V] | 61 (J7) [5V]	V_{CC} (10 | 11 | 12)
GND	1 | 2 (J10-16) | 60 (J6) | 62 (J7)	GND (7 | 8 | 9)
MOSI	55 (J6)	DIN (6)
MISO	54 (J6)	DOUT (4)
CLK	47 (J5)	SCLK (2)
CS	59 (J6)	CS (3)
DATA READY	74 (J8)	DIO1 (13) or DIO2 (14)
RESET	73 (J8)	RST (1)
AUX SIGNAL 1	48 (J5)	DIO3 (15)
AUX SIGNAL 2	42 (J5)	DIO4 (16)

Power Sub-Circuit

The board consists of two low-power supply channels, one for 5V and another for 3.3V. Both of these channels are imperative for the function of the LaunchPad. On the other hand, only one is rooted in the corresponding IMU. To achieve this, since ADIS16364 demands a 5V power supply [70] in contradiction with ADIS16375 that requires 3.3V [71], a selector jumper

was placed on the board. To choose appropriate supply regulators, the power consumptions for each electronic component included in the design are presented in Table 4-3.

Table 4-3. Different components' power consumption.

PROPERTIES	Delfino	ESC	ADIS16364	ADIS16375
Input Voltage [V]	3.3 | 5	5	5	3.3
SS Current [mA]	325 (440 max)	350 (700 max)	49	173
Transient Current [mA]	N/A	N/A	N/A	1500 [400us]

In the case of the Delfino microcontroller, the listed current values are representative of the test conditions given in [76]. The actual device's current consumption in an application will vary with the application's code and pin configurations. In the manufacturer's test conditions there are significantly higher power demands compared to the current application's, since both CPUs are enabled and many peripherals, like ePWMs that are not used in the current application. The regulators were chosen to fulfill even these high current consumptions in order to be on the safe side under any operational condition, with provisions for further expansion in the future. Note that the 5 V power supply channel is not used by the microcontroller, but it supplies external components (e.g. ESC) connected to the LaunchPad.

Since the cost remains at the same levels, an "on the safe side" design was adopted. For the 3.3 V channel, the TI's LM1085IT fixed, Low Dropout Regulator (LDO) was selected. As may be seen in the corresponding datasheet [77], it outputs up to 3 A. As for the 5 V supply branch, the ST's L7805CV LDO [78] was chosen, which can output up to 1.5 A. Generally, LDOs entail decoupling capacitors to cutoff transients and high-frequency noise phenomena (Figure 4-8). By referring to their datasheets, a list of peripherals and rooting tips are proposed for each LDO. To furtherly enhance the proposed filters, additional 100 nF capacitors were positioned near the coupling point of each load.

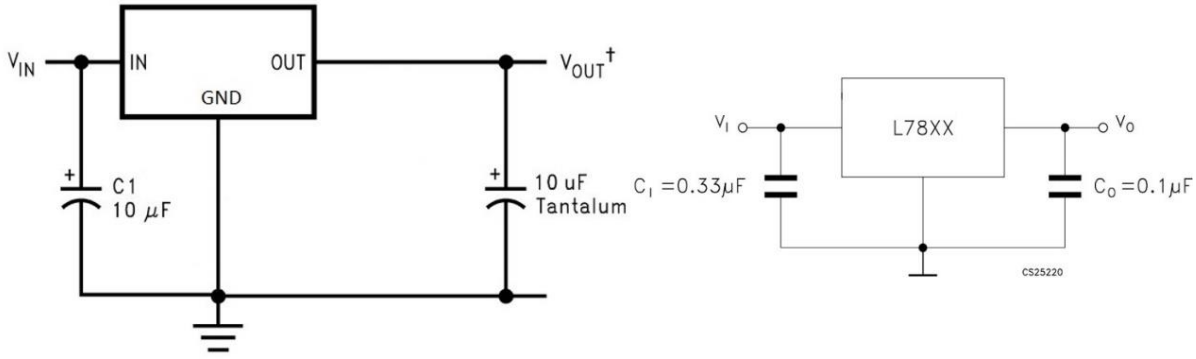


Figure 4-8. Proposed decoupling capacitors for the chosen linear voltage regulators.

To prove that the proposed design is on the safe side, specific Safety Factors (SFs) have been calculated. From the datasheets' worst-case conditions (Table 4-3) and for the steady-state operation, the SFs of each power supply channel for the two IMU slaves are listed in Table 4-4. Note that the SFs are calculated with the equation (4-1), with I_{max} being the maximum current consumption of each channel and $I_{supply,max}$ the maximum current that each LDO can output.

$$SF_{supply} = \frac{I_{max}}{I_{supply,max}} \tag{4-1}$$

Table 4-4. IMU slaves' power supply channels Safety Factors (SFs).

Power Supply	ADIS16364 SLAVE SF [%]	ADIS16375 SLAVE SF [%]
5 V (1.5 A max.)	200.3	214.3
3.3 V (3 A max.)	681.8	489.4

The slaves are going to be supplied by the low-power supply channel of Laelaps II (**9 V** by default). According to [79], the output current of the LDOs depends on the drop-out voltage, which is the difference between the input with the output voltage. The minimum drop-out voltage required to maintain regulation is rated at 1.5 V for the LM1085IT and 2 V for the L7805CV. So, the input voltage for each slave should be higher than 7 V. Since Laelaps II operates at 9 V, the requirement is fulfilled. Note that as the input voltage rises significantly higher than the minimum required, the power dissipated in each LDO rises too. So certain thermal limits should be taken into account in the design. A thermal analysis is performed in Appendix I, using the thermal specifications that the datasheets provide. Besides this analysis, an experiment was executed to verify the results. In the same Appendix, its description can be found. The results indicate that for input voltages above 12.7 V, the LM1085IT LDO overheats and its thermal protection disables it. So, the whole EtherCAT slave must always be supplied with lower than 12.7 V input voltages.

SPI Bus Sub-Circuits

As formerly mentioned, the two Analog IMUs are communicating, transferring readings and commands via SPI. In this section, the hardware realization of this protocol is considered, along with techniques to ensure minimum noise and seamless operation in high frequencies.

General bus design guidelines propose minimum trace impedance and the same low trace length for each of the four SPI signals [80]. The PCB software suite used here comes with an arsenal of great tools for a diversity of needs. By running the ULP script *"length-freq-ri.ulp",* one may review each trace's attributes and act accordingly. In the current scheme, the SPI's traces were kept in almost the same length and impedance. In Table 4-5, all of the traces' characteristics are summarized. The maximum SPI functional frequency is dictated by the IMU datasheets [70] [71] and Delfino's capabilities. That said, a 12 MHz SPI clock is used with ADIS16375 and a 600 kHz one with ADIS16364. In the case of ADIS16364, if burst read (refer to Section 4.6.3) is not intended to be used, this value can be increased.

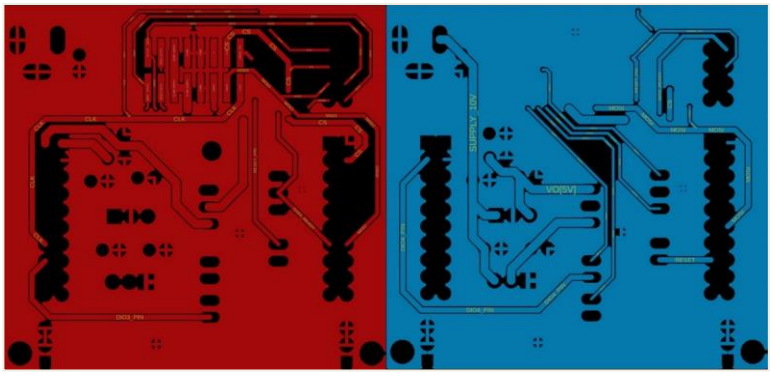
Table 4-5. Custom PCB traces' attributes.

SIGNAL	F_{max} [MHz]	L [mm]	A [mm^2]	R [mOhm]	W_{min} [mm]	W_{max} [mm]	I_{max} [A]
DIO3	3245.13	92.385	0.014	113.01	0.406	0.406	1.25
DIO4	3269.15	91.706	0.014	112.18	0.406	0.406	1.25
RESET_PIN	5361.33	55.919	0.014	68.4	0.406	0.61	1.25
CLK	5625.08	53.297	0.028	32.6	0.813	0.813	2.45
VO[3.3V]	5886.04	50.934	0.044	19.94	1.27	1.27	3.75
DIO4_PIN	6217.4	48.219	0.021	39.32	0.61	0.813	1.85
MISO	6253.86	47.938	0.021	39.09	0.61	0.61	1.85
DATA_READY	6564.9	45.667	0.021	37.24	0.61	0.61	1.85
SUPPLY_10V	6653.77	45.057	0.044	17.64	1.27	1.27	3.75

MOSI	7058.21	42.475	0.028	25.98	0.813	0.813	2.45
CS	7642.41	39.228	0.014	47.99	0.406	0.813	1.25
IMU_SUPPLY	7985.16	37.545	0.044	14.7	1.27	1.27	3.75
DIO3_PIN	8639.69	34.7	0.028	21.22	0.813	0.813	2.45
VO[5V]	9374.19	31.981	0.044	12.52	1.27	1.27	3.75
DIO1	9935.61	30.174	0.014	36.91	0.406	0.406	1.25
DIO2	10003.2	29.97	0.014	36.66	0.406	0.406	1.25
RESET	38487.7	7.79	0.028	4.76	0.813	0.813	2.45

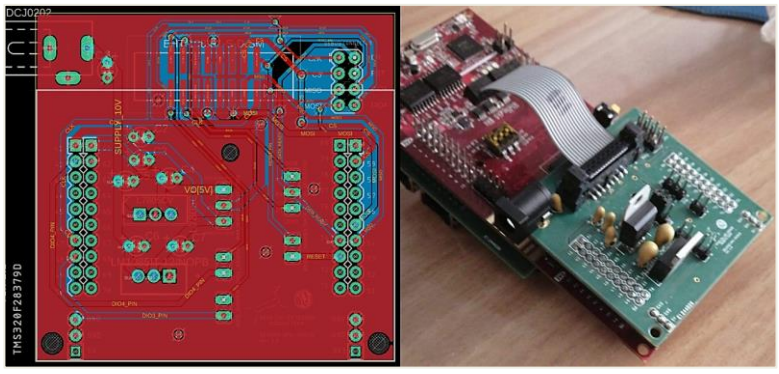
Board Overview

The designed PCB is using a limited space at the lower half of the LaunchPad, to reduce cost along with its volume demands. The traces are optimized having adequate clearance among them, to eliminate signal interferences. The reader may refer to the extended work in [81] and the guide in [82], to get a deep insight into the field of high-end PCB design. During the design process, Eagle provides tools, like ERC and DRC checks to optimize the design and ensure its manufacturability. The rules that these checks follow can be imported if the default ones do not suffice. Many PCB manufacturers, like Eurocircuits, provide their own *.dru* configuration files, compatible with all major PCB design software packages. Since the boards were purchased from Eurocircuits, the PCBs were designed in compliance with these *.dru* files. In Figure 4-9 and Figure 4-10, the final PCB design may be examined.



(a) **(b)**

Figure 4-9. (a) Upper PCB side. (b) Lower PCB side.



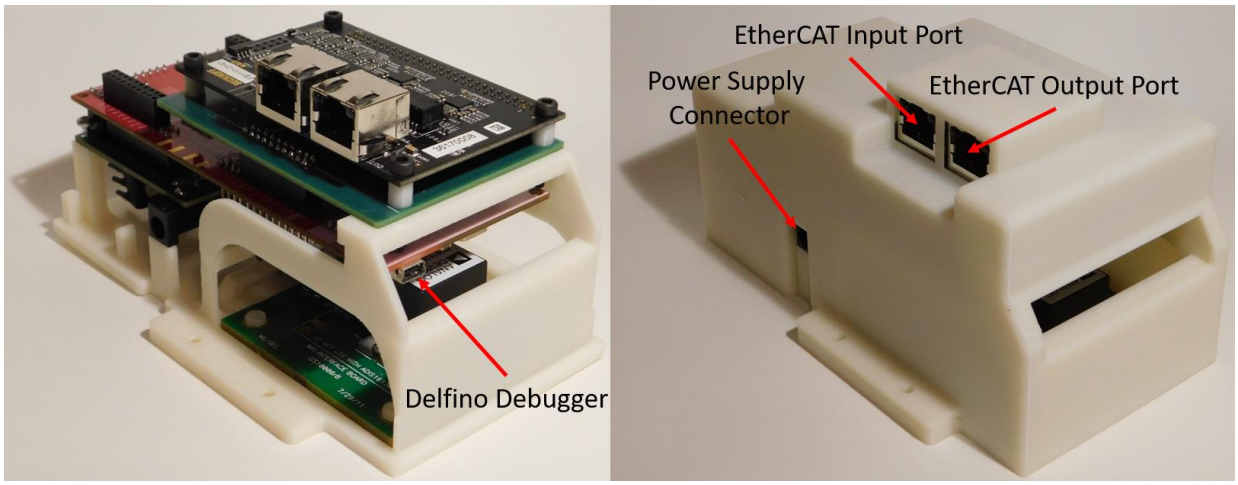
(a) **(b)**

Figure 4-10. (a) Custom PCB design overview. (b) Assembled boards.

As may be realized by the board in Figure 4-10, the bottom and top copper layers are short-circuited as ground (GND) to reduce the total number of traces and, along with some vias, the total impedance of the circuit. The exported from Eagle *.brd* file was uploaded to be analyzed with the more advanced Eurocircuits tools. The verdict of those manufacturability tests was exceptional, with copper plating fault indexes nearly 100% pass. All of the design files may be found in the respective Bitbucket repositories [83] [84].

4.4.3 IMU Slave Housing

To enclose the aforesaid board assembly, a 3D printed housing was created. An effort was made to be as compact as possible, ensuring the integrity of the hosted components and their cooling needs. The housing is illustrated in Figure 4-11. For the drawings and the basic dimensions refer to Appendix L.



(a) **(b)**

Figure 4-11. (a) Housing – inside view. (b) Housing – closed view.

4.5 ADIS163xx Delfino Firmware Setup

To use the sensors with LaunchXL-F28379D, the appropriate firmware was developed. The main tasks of the firmware are to communicate via SPI with the connected IMU, set the necessary parameters, get the sensor readings and establish the EtherCAT communication. Figure 4-13 outlines the general functionality of the designed firmware. To furtherly understand the structure, refer to the created Doxygen documentation [83] [84].

The various aspects of the firmware are discussed in an organized way, to make the learning curve less steep and the overall process more efficient. As far as the EtherCAT slave software modules are concerned, the aim here is not to replicate the extensive available literature, but to highlight the significant routines and operations that interfere most with the user-application.

4.5.1 Required Hardware

Firstly, a sum-up of all the necessary components is illustrated in the list below.

- 1x EtherCAT slave piggyback controller (ESC) with *ET1100* chipset. All the required frame processing and EtherCAT functionalities are implemented with this board. It is the hardware in which the physical and data link layers are realized.

- 1x TI's LaunchXL-F28379D (MCU). The application layer of the app is realized here, along with the generic EtherCAT stack and the IMU SPI communications.
- 1x ADIS163xx IMU, along with its breakout board for proper connectivity.
- 1x Custom EtherCAT shield PCB. The interface board that connects properly the MCU with the ESC, designed in [20].
- 1x Custom IMU Shield PCB. The intermediate board operating as a rewiring scheme between the MCU and the IMU. Also, it provides the required power supply for the whole board assembly.
- 1x IMU Housing (Figure 4-11). It is used to host the assembled boards and electronics and can be placed anywhere on the robot.

The reader should consult the Appendix K for a detailed description of the required components.

Note that depending on the IMU (ADIS16364 or ADIS16375), the custom IMU shield's power supply selector should be configured accordingly, by placing a female jumper to the appropriate pins (see Figure 4-12). Note that ADIS16364 operates with a 5 V supply (green box), while ADIS16375 operates with a 3.3 V supply (blue box).

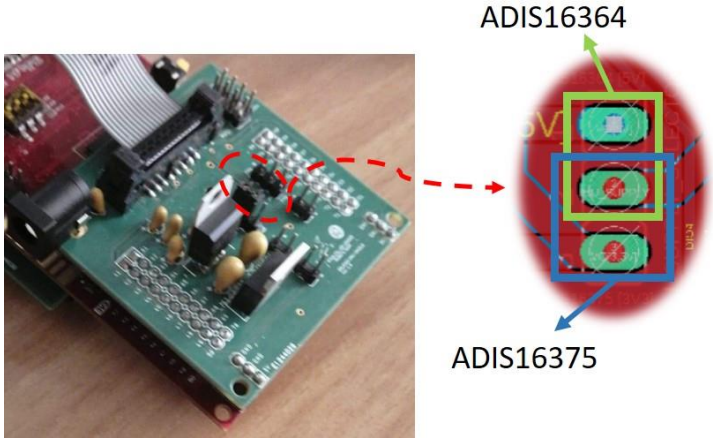


Figure 4-12. ADIS163xx IMU shield's power supply configurations.

4.5.2 Required Software

After all the physical components are gathered, the software components must be installed. By navigating in the TI's site [59], the latest version of Code Composer Studio (CCS) could be installed with C2000 software components checked in the corresponding installation pane. One should also download the latest C2000WARE from the Resource Explorer. Currently, the project supports version *3.03.00.00,* but with few adjustments in the properties menu, the update process to any future version should be easy.

Secondly, a master has to be installed, not necessarily on the same PC. There are plenty of choices for Windows and Linux masters. The Legged Robots Team currently maintains two master setups; one using Twincat 3, a robust and well-proven suite running on Windows with Visual Studio Core, and another, Linux hard real-time EtherLab Master with ROS enhancements. Due to the complexity of the high-end, but still in development Linux master, the use of Beckhoff's Twincat 3 is suggested. The interested one may install Twincat 3 in [40]. The EtherLab master may be found, along with build and deployment information, in the corresponding Bitbucket repository [60].

4.5.3 Building and Deployment

After completing the initial steps, one can import, build and deploy the IMU application. To complete the task, several steps must be followed:

1. Navigate to the link in [83] and download ***imu_adis16364-ecat-slave*** for ADIS16364 or in [84] to download ***adis16375_imu_ecat_slave*** for ADIS16375. In here, the master's ENI *.xml* file (under *TwinCAT Configuration Files* folder), the corresponding Excel files, and the SSC's projects (to modify or rebuild the EtherCAT stack) are included, too.
4. Open TI's CCS. If it is not already installed, see details on how to do it and how to import the downloaded project in Appendix E (Section E.1).
2. Next, a required Target Configuration for the debugger should be set. If such configuration is not available from prior use, details on how to create one from scratch may be found in Appendix E (Section E.2).
3. Modify the custom path variable to the one of the PC used at the time. This variable makes the C2000Ware libraries visible to the project's scope. To do this:
    a. Right-Click on the project's entry in the solution explorer's tab and select *Properties* like in Figure 3-21.
    b. Opt for the *Linked Resources* tab and modify the variable **C2000Ware_LOC** to match the local installation folder of C2000Ware.

Finally, details on how to deploy the firmware to a LaunchXL-F28379D can be found in Appendix E (Section E.3). At this point, the firmware will be deployed in its default configuration. If the defaults do not suffice, consult the next sections to understand it thoroughly and make the required adjustments.

4.6 Firmware Description

The firmware's main operation can be adumbrated in three ISRs. They are triggered by the ESC's DC unit as external hardware interrupts. Inside these ISRs, the user-application is implemented. The application's main tasks involve setting up and handling the IMU communications, requesting sensory readings and pass them to the respective EtherCAT PDOs to be sent to the master. These matters will be discussed in detail in the sections that follow.

The CCS project is organized in the following folders:

- *hal*: This folder materializes the EtherCAT hardware abstraction layer (hal). Specifically, it contains all peripheral drivers and routines for handling the PDI's communications.
- *SPI_EtherCAT Slave Stack*: In here, the generic EtherCAT stack's files can be found. They materialize the procedures discussed in Chapter 2.
- *ADIS163xx Library*: This folder contains all the IMU related routines. Some of them handle the SPI communications with the sensory module and others materialize the user-functionalities discussed in the current section.
- *cmd*: This folder contains the created linker command files that dictate the project's memory allocation.

Those said, the firmware's execution flow is analyzed next. Firstly, the required initializations of the most significant peripherals and modules will be discussed (highlighted in yellow, in Figure 4-13). Moreover, the main application and its dependencies will be analyzed

(highlighted in light-blue), along with other EtherCAT related implementations. Note that both IMUs operate in a similar manner and their applications have basically the same architecture. Thus, the discussion that follows applies to both of them. Note that the parts that differ will be pointed out.

To facilitate the future development and understanding of the firmware, the MISRA C:2012 coding standard has been adopted [62] (refer to Appendix H, Section H.2). Moreover, the code complies with the coding style discussed in Appendix G. Note that to this end, the created Doxygen documentation [83] [84] could help in getting an overall grasp of the firmware faster.

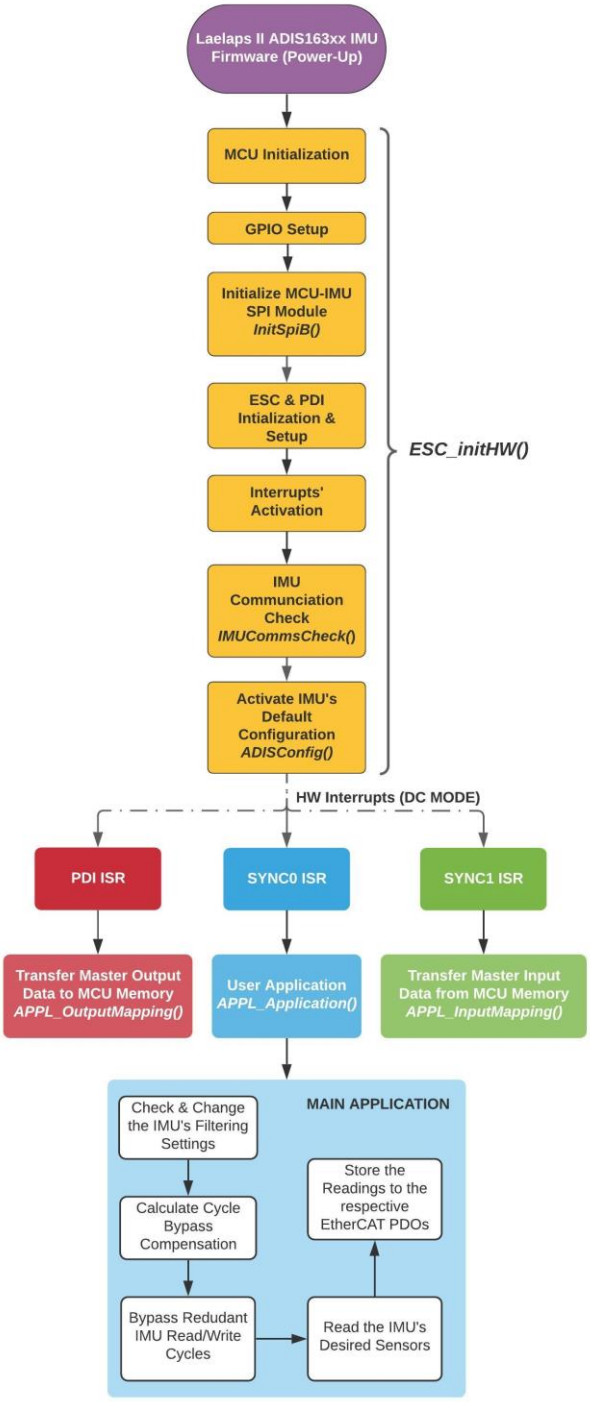


Figure 4-13. IMU firmware overview.

4.6.1 Initialization

The firmware's execution begins at the ***main()*** function located in the *ADIS163xx_IMU_ECAT_SLAVE.c* file (inside the project's *stack* folder). The ***main()*** calls the ***HW_Init()*** function (located inside the *c28xxhw.c,* under *stack*) that in turn initializes the Delfino MCU and the ESC. Eventually, the Delfino MCU is initialized by calling the ***ESC_initHW()***, located inside the *etherCAT_slave_c28x_hal.c* (under the *hal* folder). In this function, depending on the run-time mode (debug or release), the required memory sections are copied from FLASH to RAM. Next, the low-level modules of the system are initialized. Furthermore, the ***InitGpio()*** routine is called and configures the TI's default GPIO muxing options for every peripheral of the MCU. Next, the PDI layer's initialization occurs, by configuring the SPIA module (**USE_SPIA** symbol, see Section 4.6.8). Moreover, the SPIB channel is configured to serve the MCU-IMU communications. In the next section, the configured SPI settings are examined. Finally, the EtherCAT-related external interrupts that trigger the firmware's main ISRs are configured.

After the generic, low-level configurations, the interrupts are activated. Next, the ***IMUCommsCheck()*** routine is called. This function performs a communication test with the connected IMU to ensure its proper operation. The initialization concludes by calling the ***ADISConfig()*** routine that configures the IMU module to a desired setup. These two functions are located *in ADIS163xx_utilities.c*, under *ADIS163xx Library* folder. For further information refer to Sections 4.6.4.

4.6.2 SPI Protocol Software Setup

The primary task of the firmware is to establish the SPI communication with the corresponding IMU. From the devices' datasheets [70] [71], the required SPI setup parameters are exported in Table 4-6. At this point, the developer should be careful on how each manufacturer defines the different SPI modes. In the specified table, SPI mode settings are written according to each distributor's preference. For the rest of this thesis, the convention of [85] is adopted.

Table 4-6. IMUs' SPI supported settings.

SPI SETUP	LAUNCHXL-F28379D	ADIS16364	ADIS16375
ROLE	Master	Slave	Slave
CLOCK [MHz] ≤	12.5 (Normal Mode)	2 (1 for Burst Read)	15
MODE	2 [POL 1 | PHA 0]	3 [POL1 | PHA1]	3 [POL1 | PHA1]
MSB MODE	N/A	First	First
TRANSFER MODE	16-bit	16-bit	16-bit

Delfino comes with three built-in SPI channels. Since channel A is preoccupied with the ESC communications, the SPIB is configured to service the IMUs. For an in-depth analysis of the SPI and GPIO muxing refer to [76]. Note that the SPI is used in a blocking manner. This means that the CPU waits for the results of a requested transcation, before advancing to the next task.

All of the above settings are materialized in the *SpiConfig.c* file (under *ADIS163xx Library* folder), by the ***InitSpiB()*** routine. To shorten the development time, C2000 driverlib's enhancements are enabled. This library is used as a learning shortcut, but the caveat is in lower register parametrization capabilities, compared to bitfield coding. In the current project, driverlib's capabilities are more than adequate.

4.6.3 ADIS163xx General Setup

After initializing the SPI communications, both devices (Delfino and IMU) are ready to communicate and transfer data. To get well-conditioned measurements, ADIS163xx modules have filtering and dynamic range options to reject noise and get clean readings. The developer has to decide the range of the IMU readings as well as the averaging preprocessing. Increased averaging filtering comes at the expense of a lower sampling rate and bandwidth. The above-described functionality is exploited by setting appropriate values at the prescribed registers, designated in ADIS163xx datasheets [70] [71]. These filtering options are materialized in the corresponding routines, stated in the below lists. Also, in the Bitbucket repositories [83] [84], *Matlab* files exist that model the frequency response of the aforementioned filters.

ADIS16364 LIBRARY

- ***TapFIRCtrl()***. Sets the averaging filter options. Specifically, if m is the user-input, 2^m samples will be averaged, eventually. It manipulates the 2 lower bits of ***SENS_AVG*** register (refer to [70]).
- ***DRngFIRCtrl()***. Sets the gyroscope's dynamic range. Lower range translates to increased sensitivity. It manipulates the appropriate bits of ***SENS_AVG*** register (refer to [70]).

ADIS16375 LIBRARY

- **GyroFIRCtrl() / AcclFIRCtrl()**. Activates one of the manufacturer's FIR filters. Special filtering options are supported, but not implemented in the current routine. It manipulates the appropriate bits of ***FILTER_SEL1*** *and* ***FILTER_SEL2*** registers (refer to [71]).
- ***ChangeDECRate()***. Sets the averaging low-pass filter options. Here, if m is the user input, m samples will be averaged. It manipulates the appropriate bits of ***DEC_RATE*** register (refer to [71]).

Many other settings can be tuned to activate a variety of supported features; however the above ones are more than sufficient for Laelaps' current needs. Note that both IMUs support detailed frame alignment, at a datasheet specified point of precursion. Last but not least, to account for the unpleasant MEMs gyroscope bias artifact caused by perpetual linear accelerations, ADIS163xx modules have an optional compensation feature that may be enabled. These configurations are activated with the ***ADISConfig()*** routine.

4.6.4 ADIS163xx Main Routines and Operation

ADIS16364

After the initial settings, ADIS16364 is ready to transmit the required sensor measurements as long as it receives the prescribed commands. The general communication scheme is simple and is completed in two consecutive SPI transfers. The first one transmits the requested register address, and in the second one, the IMU sends a 16-bit unsigned integer that corresponds to the contents of the aforementioned address. The second transfer is optionally triggered by a prescribed data-ready pulse. To get the final signed floating-point value of the reading, some software processing is required since the raw readings are configured in two's complements format. Furthermore, the 16-bit word is scaled by a factor. The location of each register's MSB differs; commonly, it is either the 14^{th} bit or the 12^{th} bit depending on the register. The functions along with their description may be reviewed below.

- ***SensorRead()***. Implements single register reads (2 SPI transfer cycles). It is the main function for reading the sensors' registers.
- ***BurstRead()***. With this function, ADIS16364 outputs multiple readings in consecutive cycles, with a single command as a trigger (Figure 4-14).

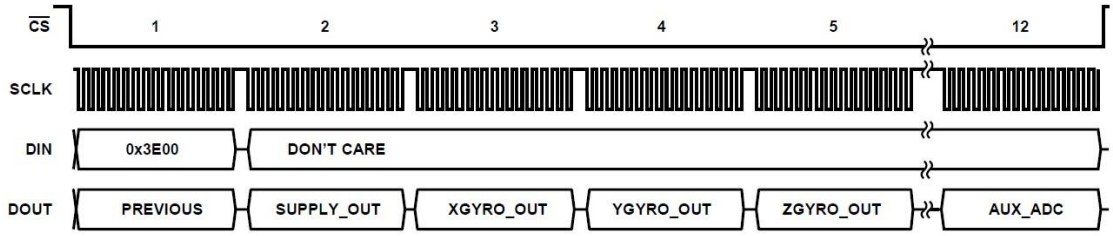


Figure 4-14. BurstRead's SPI operation overview.

- ***RawToReal()***. Raw to float conversion of the register readings.
- **SensorWrite()**. Single register-write routine, mainly used for writing values to command-registers that change the IMU's general setup. It is also an auxiliary function, which is used by others.
- ***ResetIMU()***. Sensor software-reset function.
- ***TwosComp()***. Two's complements, signed to unsigned conversion of a value.
- ***IMUCommsCheck()***. Requests the fixed ADIS16364-ID number and checks if the received value is correct as a simple and indirect SPI communication check.

All of the above routines come as a module and may be used with an **#include** of the corresponding header file (*ADIS16364.h*). There are two versions of ADIS16364 firmware available, with and without EtherCAT capabilities. By adding EtherCAT capabilities to the designed drivers, the IMU device may be used as a slave and be integrated directly into the Laelaps' network.

ADIS16375

A slightly modified firmware adapted to the requirements of ADIS16375 IMU is developed. Some routines utilize the two SPI cycles to read the contents of a register, while others serve general purposes. The difference of the ADIS16375 IMU is that it produces measurements with greater precision than the former one. So, to read a measurement register in high precision mode, four SPI cycles are required. Two of them are for transmitting high and low registers' addresses and the remaining ones are for receiving the contents of each register. Admittedly, this procedure may be executed with three SPI cycles by modifying the firmware's routines, but the gain of this would be insignificant, and the firmware would become more complex. Figure 4-15 illustrates the described procedure.

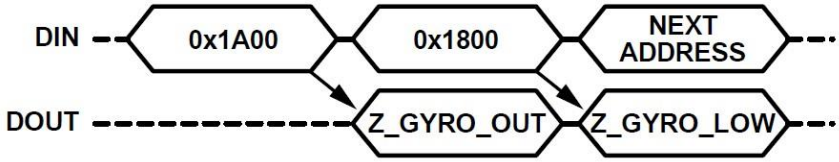


Figure 4-15. ADIS16375 sensor's high-precision read operation.

An important remark to avoid confusion is that due to the large number of registers that ADIS16375 disposes of, they are organized into separate register pages. So to modify a

specific register, its actual page needs to be activated beforehand. Refer to the accompanying datasheet [71] for further information. The remaining procedures to get the actual floating-point value of the register are the same with the described in ADIS16364 case (Section 4.6.4). The developed routines alongside their descriptions may be found in the list below.

- ***SensorRead()***. Implements single sensor reads (4 or 2 SPI transfer cycles-depending on the precision mode).
- ***RegRead()***. Reads a single 16-bit register.
- ***RawToReal()***. Performs raw to float conversion of the register readings.
- ***RegWrite()***. Single register-write routine, mainly used in setting-registers, to change the IMU's general setup. It is also an auxiliary function, which is used by other functions.
- ***ResetIMU()***. Sensor software-reset function.
- ***TwosComp()***. Implements a two's complements, signed to unsigned conversion of a value.
- ***PageCheck()***. Executes the register-page check and change functionalities.
- ***X/Y/Z_VelocityReading()***. Cumulative linear velocity approximation routine.
- ***X/Y/Z_AngleReading()***. Cumulative angle approximation routine.
- ***IMUCommsCheck()***. Requests the fixed ADIS16375-ID number and checks if the received value is correct, as a simple and indirect SPI communication check.

4.6.5 ADIS163xx EtherCAT Application

The main target of the firmware is to integrate the IMUs into the robot's EtherCAT network. With this in mind, two slave stack applications were developed, each tailored to get the most of the corresponding IMU. The EtherCAT master must be able to request the readings of specific sensors of the IMUs and receive them. Also, the filtering options must be handled by the master, too. With this in mind, in the following sections the created EtherCAT applications are discussed.

ADIS16364 EtherCAT Application

The main aspect of an EtherCAT application is the process data that it handles (see Section 2.2.5). They are distinguished in output and input objects. The former ones are all of the variables that can be modified by the master and are interpreted as commands or virtual switches by the slave. The latter represent the data that is provided to the master by the slave. There is extensive documentation on how to design and build an EtherCAT application's stack [35] [86] [87] [88]. The reproduction of such manuals in this thesis would be redundant. Nevertheless, a short guide could be found in Appendix B for completeness purposes. The ADIS16364 process data are presented in Table 4-7 and Table 4-8.

Table 4-7. ADIS16364 EtherCAT PDO-Inputs.

Index	Sub-Index	Data Type	Name	Comments
0x6000	**RECORD**		**READINGS**	
	0x01	REAL	XGyro_out	X-Axis gyroscope reading
	0x02	REAL	YGyro_out	Y-Axis gyroscope reading
	0x03	REAL	ZGyro_out	Z-Axis gyroscope reading
	0x04	REAL	XAccl_out	X-Axis accelerometer reading

	0x05	REAL	YAccl_out	Y-Axis accelerometer reading
	0x06	REAL	ZAccl_out	Z-Axis accelerometer reading
	0x07	REAL	XTemp_out	X-Axis temperature reading
	0x08	REAL	YTemp_out	Y-Axis temperature reading
	0x09	REAL	ZTemp_out	Z-Axis temperature reading

Table 4-8. ADIS16364 EtherCAT PDO-Outputs.

Index	Sub-Index	Data Type	Name	Comments
0x7000	RECORD		IMU_CONTROLS	
	0x01	BOOL	XGyro_sw	X-Axis gyroscope switch
	0x02	BOOL	YGyro_sw	Y-Axis gyroscope switch
	0x03	BOOL	ZGyro_sw	Z-Axis gyroscope switch
	0x04	BOOL	XAccl_sw	X-Axis accelerometer switch
	0x05	BOOL	YAccl_sw	Y-Axis accelerometer switch
	0x06	BOOL	ZAccl_sw	Z-Axis accelerometer switch
	0x07	BOOL	BurstRead_sw	Burst-read switch
	0x08	BOOL	XTemp_sw	X-Axis temperature switch
	0x09	BOOL	YTemp_sw	Y-Axis temperature switch
	0x10	BOOL	ZTemp_sw	Z-Axis temperature switch
	0x11	BOOL	Reset_sw	Software reset switch
	0x12	Pad_5		Alignment padding
	0x13	USINT	Tap_Ctrl	Averaging filter tap number
	0x14	USINT	DRng_Ctrl	Dynamic range selection

The main application, namely the ***APPL_Application()*** routine (in *ADIS16364_IMU_ECAT_SLAVE.c*, under *stack*) is triggered by an external hardware interrupt by the ESC (DC mode) at the rate of the defined EtherCAT cycle (2.5kHz in Laelaps' case). ADIS16364 has 819.2 default sampling rate. So, the aforementioned EtherCAT frequency results in redundant IMU transactions. To avoid this behavior there are two solutions. In the first one, the master sets the external interrupts to be triggered at a lower frequency compared to the default one of the Laelaps' network. An alternative way that is implemented in the current application is to calculate the redundant cycles and with the use of a software counter bypass IMU reads in the redundant triggers.

Briefly, the application sets the filtering and measurement conditioning settings that the master requests, calculates the cycle bypass count and calls the ***IMUOpStateMachine()*** that implements the bypass mechanism and reads the requested by the master IMU registers. The described routine follows in the below code snippet.

```
void APPL_Application(void)
{
    //
    // Bypass Frequency Count Variable Initialization
    //
    uint16_t SamplingCount = 0U;

    //
    // DC Mode Operation Variable
    //
    const uint16_t DC_MODE = bDcSyncActive;

    //
    // Debug LEDs State Update
    //
    GPIO_WritePin(34, !IMU_CONTROLS0x7000.BurstRead_sw);
    GPIO_WritePin(31, !IMU_CONTROLS0x7000.Reset_sw);

    //
    // Active Taps Change User Request
    //
    if (g_ActiveTaps != IMU_CONTROLS0x7000.Tap_Ctrl)
    {
        //
        // Desired #Taps is Valid
        //
        if (IMU_CONTROLS0x7000.Tap_Ctrl >= g_MinTaps)
        {
            //
            // Change Active Taps
            //
            TapFIRCtrl(IMU_CONTROLS0x7000.Tap_Ctrl);
        }

        //
        // Active Taps different from Minimum Allowed
        //
        else if (g_ActiveTaps != g_MinTaps)
        {
            //
            // Change Active #Taps to Minimum Allowed
            //
            TapFIRCtrl(g_MinTaps);
        }

        //
        // Any other Case
        //
        else
        {
            ;
        }

    }

    //
    // Active Dynamic Range Change User-Request
    //
    if (g_ActiveDRng != IMU_CONTROLS0x7000.DRng_Ctrl)
    {
        //
        // Change Active Dynamic Range
        //
        DRngFIRCtrl(IMU_CONTROLS0x7000.DRng_Ctrl);
    }

    //
    // EtherCAT DC Mode Operation
    //
    if (DC_MODE == 1U)
    {
        //
        // Cycle Time Variable in [us]
        //
```

```
        uint32_t CycleTime = sSyncManOutPar.u32Sync0CycleTime;

        //
        // EtherCAT Loop Frequency in [Hz]
        //
        float CycleFrequency = 1000000000.0f / (float)CycleTime;

        //
        // Calculate Frequency Bypass Total Count
        //
        SamplingCount = (uint16_t)(CycleFrequency
                        / (ADIS_SMPL / exp2f(g_ActiveTaps)));
    }

    //
    // EtherCAT non-DC Mode Operation
    //
    else
    {
        //
        // Reset Frequency Bypass Count
        //
        SamplingCount = 0U;
    }

    //
    // Run IMU Operation State Machine
    //
    IMUOpStateMachine(SamplingCount);

}   // End Of APPL_Application()
```

ADIS16375 EtherCAT Application

Likewise, the process data, presented in Table 4-9 and Table 4-10, are adapted to ADIS16375 needs.

Table 4-9. ADIS16375 EtherCAT PDO-Inputs.

Index	Sub-Index	Data Type	Name	Comments
0x6000	RECORD		READINGS	
	0x01	REAL	XGyro_out	X-Axis gyroscope reading
	0x02	REAL	YGyro_out	Y-Axis gyroscope reading
	0x03	REAL	ZGyro_out	Z-Axis gyroscope reading
	0x04	REAL	XAccl_out	X-Axis accelerometer reading
	0x05	REAL	YAccl_out	Y-Axis accelerometer reading
	0x06	REAL	ZAccl_out	Z-Axis accelerometer reading
	0x07	REAL	XAngle_out	X-Axis delta-angle reading
	0x08	REAL	YAngle_out	Y-Axis delta-angle reading
	0x09	REAL	ZAngle_out	Z-Axis delta-angle reading
	0x10	REAL	XLinVel_out	X-Axis delta-linear velocity reading
	0x11	REAL	YLinVel_out	Y-Axis delta-linear velocity reading
	0x12	REAL	ZLinVel_out	Z-Axis delta-linear velocity reading
	0x13	REAL	Temp_out	Temperature reading

Table 4-10. ADIS16375 EtherCAT PDO-Outputs.

Index	Sub-Index	Data Type	Name	Comments
0x7000	RECORD		IMU_CONTROLS	
	0x01	BOOL	XGyro_sw	X-Axis gyroscope switch
	0x02	BOOL	YGyro_sw	Y-Axis gyroscope switch
	0x03	BOOL	ZGyro_sw	Z-Axis gyroscope switch
	0x04	BOOL	XAccl_sw	X-Axis accelerometer switch
	0x05	BOOL	YAccl_sw	Y-Axis accelerometer switch
	0x06	BOOL	ZAccl_sw	Z-Axis accelerometer switch
	0x07	BOOL	XAngle_sw	X-Axis delta-angle switch
	0x08	BOOL	YAngle_sw	Y-Axis delta-angle switch
	0x09	BOOL	ZAngle_sw	Z-Axis delta-angle switch
	0x10	BOOL	XLinVel_sw	X-Axis delta-linear velocity switch
	0x11	BOOL	YLinVel_sw	Y-Axis delta-linear velocity switch
	0x12	BOOL	ZLinVel_sw	Z-Axis delta-linear velocity switch
	0x13	BOOL	Temp_sw	Temperature switch
	0x14	BOOL	High_Precision	High precision sensor reading mode
	0x15	BOOL	DEBUG_LED	Debug LED switch
	0x16	BOOL	RESET_IMU	Software reset switch
	0x17	UINT	Dec_Rate	Set decimation filter rate
	0x18	USINT	Gyro_FIR_Ctrl	Gyroscopes lowpass filter control
	0x19	USINT	Accl_FIR_Ctrl	Accelerometers low-pass filter control

The above-presented process data modules are easily expandable and adaptable to any application. There are certain rules to follow that are partially described in the Excel file, in which the app is constructed. The reader should certainly refer to these documents [45] [89] for an in-depth understanding.

Likewise in the previous case, the same bypass mechanism is implemented and by calling the ***IMUOpStateMachine()*** routine, the firmware reads the sensor registers that the master has requested.

```
void APPL_Application(void)
{
    //
    // Bypass Frequency Count Variable Initialization
    //
    uint16_t SamplingCount = 0U;

    //
    // DC Mode Operation Variable
    //
    const uint16_t DC_MODE = bDcSyncActive;
```

```
    //
    // Gyroscopes' FIR Options Change User-Request
    //
    if (IMU_CONTROLS0x7000.Gyro_FIR_Ctrl != g_GyroFIR)
    {
        //
        // Change Gyroscopes' FIR Options
        //
        GyroFIRCtrl(IMU_CONTROLS0x7000.Gyro_FIR_Ctrl);
    }

    //
    // Accelerometers' FIR Options Change User-Request
    //
    if (IMU_CONTROLS0x7000.Accl_FIR_Ctrl != g_AcclFIR)
    {
        //
        // Change Accelerometers' FIR Options
        //
        AcclFIRCtrl(IMU_CONTROLS0x7000.Accl_FIR_Ctrl);
    }

    //
    // EtherCAT DC Mode Operation
    //
    if (DC_MODE == 1U)
    {
        //
        // Cycle Time Variable in [us]
        //
        uint32_t CycleTime = sSyncManOutPar.u32Sync0CycleTime;

        //
        // EtherCAT Loop Frequency in [Hz]
        //
        float CycleFrequency = 1000000000.0f / (float)CycleTime;

        //
        // Decimation Rate Change User-Request
        //
        if (IMU_CONTROLS0x7000.Dec_Rate != g_ActiveDECRate)
        {
            //
            // Change Decimation Rate
            //
            ChangeDECRate(IMU_CONTROLS0x7000.Dec_Rate);
        }

        //
        // Calculate Frequency Bypass Total Count
        //
        SamplingCount = (uint16_t)CycleFrequency
                        / (2.0f * ADIS_SMPL / ((float)g_ActiveDECRate + 1.0f));

    }

    //
    // EtherCAT non-DC Mode Operation
    //
    else
    {
        //
        // Reset Frequency Bypass Count
        //
        SamplingCount = 0U;
    }

    //
    // Run IMU Operation State Machine
    //
    IMUOpStateMachine(SamplingCount);

}   // End Of APPL_Application()
```

4.6.6 EtherCAT PDO routines

Besides the main ***APPL_Application()*** routine described above, there are other two that handle the data transaction between the MCU and the ESC. The routines are listed below:

- ***APPL_InputMapping()***. Performs the transfer of the master's input variables from the Delfino's local memory to the ESC's.
- ***APPL_OutputMapping()***. Performs the transfer of the master's output variables from the ESC's memory to Delfino's.

Lastly, the ***APPL_GenerateMapping()*** declares the PDO data sizes and is configured by measuring how many bytes each PDO tethers.

4.6.7 Interrupt Priorities

In an EtherCAT slave, the ESC triggers the interrupts with external singals (XINT interrupts) when the former is configured to operate in DC synchronization. Section 2.2.4 points out that there are three ISRs that handle the major routines of EtherCAT, namely ***Sync0_Isr()***, ***Sync1_Isr()*** and ***PDI_Isr()***. These interrupts have the following mapping to the routines mentioned in the previous section, when the slave operates in DC Synchronization mode:

- ***Sync0_Isr()*** → ***APPL_Application()***
- ***Sync1_Isr()*** → ***APPL_InputMapping()***
- ***PDI_Isr()*** → ***APPL_OutputMapping()***

In the current section, their functionality is not discussed. They are investigated regarding their interrupt priorities. According to the configuration provided by the TI, the aforementioned ISRs are mapped to XINT5 (INT12.3), XINT4 (INT12.2) and XINT1 (INT1.4), respectively. According to Table C-1 (Appendix C), their core priorities are 16, 16 and 5, respectively. Since they are organized in PIE groups, each one has an internal group priority. So the eventual priorities are listed below:

- ***Sync0_Isr()*** has 16.3 priority
- ***Sync1_Isr()*** has 16.2 priority
- ***PDI_Isr()*** has 5.4 priority

In the current firmware setup, there are no other interrupts running. So, the TI's default configuration has been adopted.

4.6.8 Managing the Project's Configurations

There are multiple configurations of the project that are managed with the *Predefined symbols* in the *Project Properties* pane. In the above sections, each time that a configuration was referenced the corresponding symbols were referenced, too. These symbols can be managed in project's properties *(Right Click on Project's Name -> C2000 Compiler -> Predefined Symbols)*, while a brief list is given in Table 4-11.

Table 4-11. Firmware's Predefined Symbols List.

Predefined Symbol Name	Serves	Comments
CPU1	System	Choose run-time CPU
DEBUG	System	Choose debug style of compilation
INTERFACE_SPI	EtherCAT Stack	Choose SPI as slave's PDI
_LAUNCHXL_F28379D	System	LaunchPad firmware configuration
USE_SPIA	EtherCAT Stack	Choose SPI channel for PDI use

FLASH	System	Initialization from FLASH
RAM	System	One time initialization from RAM
ETHERCAT_STACK	EtherCAT Stack	Enable the prescribed EtherCAT Stack code files
CLA_C / CLA_P	Linker	Create CLA-specific linker sections

4.6.9 Compiler Information

The firmware uses the TI's default C28x compiler *v20.2.4LTS***.** It is the standard compiler for the C2000 Delfino microcontrollers. The compiler optimization level is currently left to 0 (none). In a future version, a higher level could be applied, but with due care, since it can lead to heavily erroneous firmware run-time. These options are located inside the *Project Properties* pane (*Right Click on Project's Name->Properties*).

4.7 Firmware's Memory Management

To optimize and enlarge the portion of memory that the current application has at its disposal, TI's default linker file was modified. In the current firmware, the same custom linker file is used as in the leg slaves' case (Section 3.5.8). Here, the CLA capabilities are not required and therefore, disabled with a simple definition change in the in the Project's properties pane *(Right Click on Project's Name -> Advanced Options -> Command File Preprocessing* and define **CLA_P = 0** and **CLA_C = 0**). As for the stack and heap memory sizes (0x800 stack and 0x1000 heap), there are no specific requirements. So, the same sizes with the motion control's firmware are used. For more details, refer to Section 3.5.8.

4.8 IMU Validation

The designed IMU EtherCAT slaves should be verified before their final integration on the robot. This thesis investigates all of the components of the aforementioned slaves. The main goals of the validation process are:

1. To verify the correct operation of the designed firmware.
2. To test the integration of the IMU slaves in the robot's EtherCAT network.
3. To make an initial assessment of ADIS163xx general performance.

4.8.1 Experimental Setup

Benchmark tests were designed and executed for both IMUs. Each one was compared to a ground truth IMU with superior performance, namely the Xsens MTi-200 [90] (Figure 4-16). It comes as a complete IMU solution that incorporates industry's leading iMEMs to generate high quality inertial readings.

Figure 4-16. Xsens MTi-200 IMU.

For the experiments, the Xsens IMU was mounted below the ADIS163xx, at the position illustrated in Figure 4-17. To avoid complex frame transformations and other possible error sources, the experimental setup was kept to the simplest possible. Note that the designed test-rig's aim is to validate the correct operation of the whole slave, not just the IMU as a sensory package. If the primary target was the latter, a more sophisticated structure should had been designed.

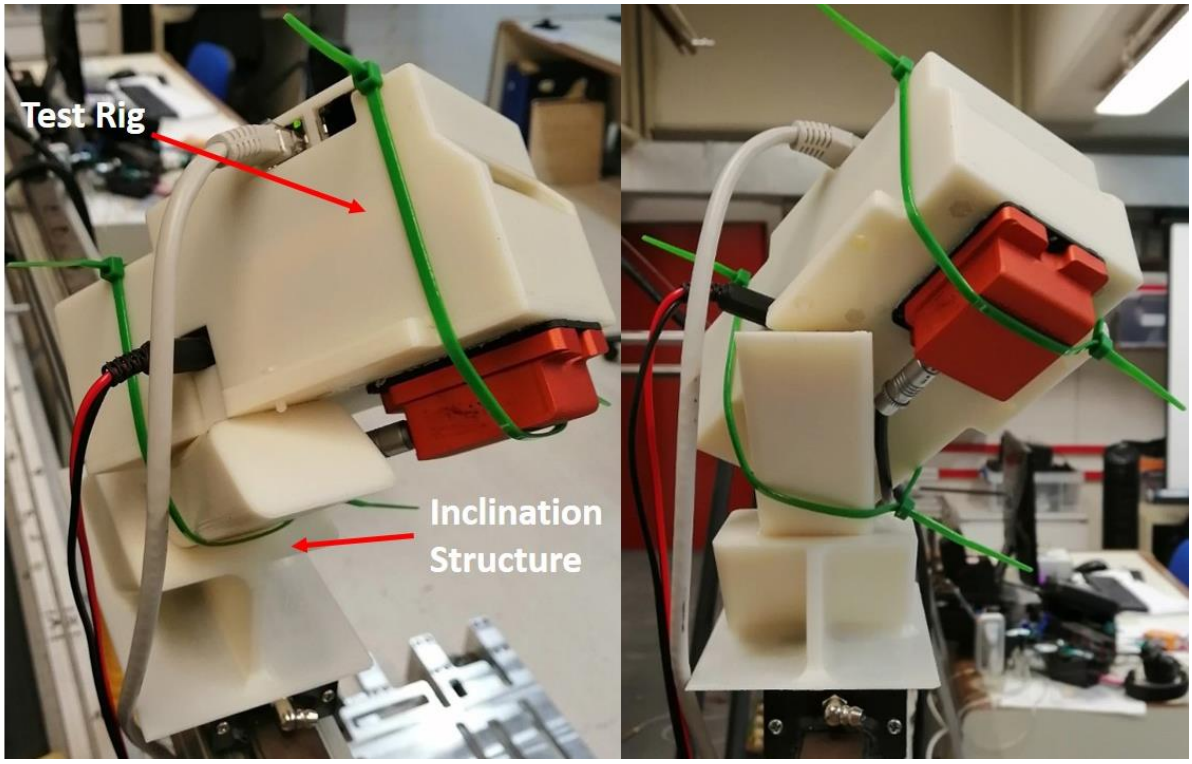


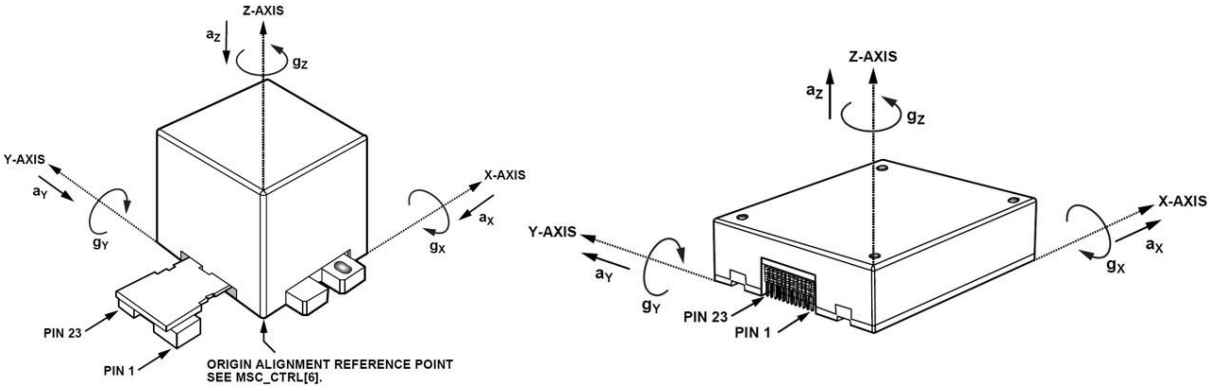
Figure 4-17. ADIS16375 accelerometers' validation setup.

To validate the gyroscopes, random rotational movements were executed. As it will become clear in the experiments' results, the test-rig was rotated by hand in every axis. Near the end of each experiment, a complex movement was executed to excite the gyroscopes of every axis simultaneously.

On the other hand, to validate the accelerometers, the created test-rig was mounted on a linear rail-guide, as Figure 4-17 illustrates. In the accelerometer experiments, the whole structure was being moved, linearly, back and forth w.r.t. the rail. The particular orientation (Figure 4-17) allowed the excitation of all accelerometers simultaneously. Note that in ADIS16364 case, there was no inclination at the Xsens' z-axis. In other words, the whole test-rig was placed directly on the rail-guide, without the inclination structure of Figure 4-17. There were some technical difficulties with TwinCAT 3 at the time that the inclination structure was added and the experiments with ADIS16364 were not completed. So, for this IMU, the results of older experiments are presented. Note that the only difference regarding the experimental setups of the two ADIS163xx IMUs was the test-rig's orientation (no inclination).

To synchronize the Xsens' measurements with the ones of the ADIS163xx, artificial impacts were created at every experiment. Also, each experiment took few seconds, since the IMUs have drift during their operation. So, as the time passes their measurements deviate significantly.

According to ADIS163xx datasheets [70] [71], the reference frame of each module is illustrated in Figure 4-18. By consulting the experimental setup (Figure 4-17) and Xsens' reference frame (Figure 4-16), the required transformations of Table 4-12 become obvious.



(a) **(b)**

Figure 4-18. (a) ADIS16364 reference frame. (b) ADIS16375 reference frame.

Table 4-12. Required frame transformations.

Xsens MTi-200 (Ref.)	ADIS16364	ADIS16375
X-Axis gyro (+)	X-Axis (+)	X-Axis (+)
Y-Axis gyro (+)	Y-Axis (-)	Y-Axis (-)
Z-Axis gyro (+)	Z-Axis (-)	Z-Axis (-)
X-Axis accelerometer (+)	X-Axis (-) [a_X]	X-Axis (+) [a_X]
Y-Axis accelerometer (+)	Y-Axis (+) [a_Y]	Y-Axis (-) [a_Y]
Z-Axis accelerometer (+)	Z-Axis (+) [a_Z]	Z-Axis (-) [a_Z]

4.8.2 IMU Specifications

According to the Xsens MTi-200 datasheet [91], the module's high resolution sampling frequency is 1 kHz. On the other hand, the sampling rate of ADIS16364 is 819.2 Hz, while the one of ADIS16375 is 2.46 kHz. Furthermore, several parameters are gathered in Table 4-13 and Table 4-14.

Table 4-13. Gyroscope specifications.

Parameter	Unit	ADIS16364	ADIS16375	MTi-200
Standard full range	[deg/s]	350	350	450
Initial bias error	[deg/s]	3	1	0.2
In-run bias stability	[deg/h]	0.42	12	10
Bandwidth [-3 dB]	[Hz]	330	330	415
Noise density	[deg/(s*Hz$^{1/2}$)]	0.044	0.02	0.01
g-Sensitivity	[deg/(s*g)]	0.05	0.013	0.003
Nonlinearity	[%]	0.1	0.025	0.01

Table 4-14. Accelerometer specifications

Parameter	Unit	ADIS16364	ADIS16375	MTi-200
Standard full range	[m/s^2]	51.485	176.520 (min)	200
Initial bias error	[m/s^2]	0.078	0.156	0.05
In-run bias stability	[μg]	100	130	15
Bandwidth [-3dB]	[Hz]	330	330	375
Noise density	[μg/Hz$^{1/2}$]	270	60	60
Nonlinearity	[%]	0.1	0.1	0.1

In the experiments, the Xsens' default filtering configuration is used (vru general), which is optimized for general purpose applications. The ADIS163xx have both preconfigured by the manufacturer, fixed filtering stages and filters to be adjusted by the user. In the current case, the custom filtering options are listed in Table 4-15. This way, the noise of all IMUs remains at the same levels. Furthermore, by enabling the filters, the firmware commands are tested in practice.

Table 4-15. Analog Devices ADIS163xx IMUs custom filtering options.

Parameter	ADIS16364	ADIS16375
Gyro. dynamic range [deg/s]	±300	±300
Decimation rate	2	2
Custom FIR filter bank	N/A	None

4.8.3 Bandwidth Considerations

The Xsens MTi-200 gyroscopes' bandwidth is 415 Hz, while the one of its accelerometers' is 375 Hz. The ADIS163xx modules provide several different options for filtering and measurement preconditioning. Specifically, in the case of ADIS16364, the user can customize the last averaging filter stage. On the other hand, in addition to the aforementioned filter, ADIS16375 offers a customizable stage of discrete FIR filter banks to accomplish the desired response. Figure 4-19 illustrates the filtering stages of each module.

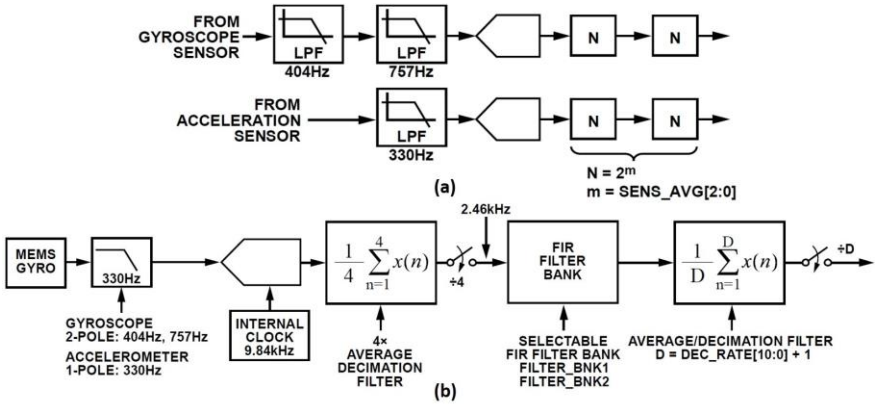


Figure 4-19. (a) ADIS16364 sampling stages. (b) ADIS16375 sampling stages.

By modeling the illustrated in Figure 4-19 filters in *Matlab* with the options provided in Table 4-15, certain conclusions can be drawn regarding the bandwidth and the phase delay

of each module. Note that the decimation filter was modeled according to the transfer function (4-2), with n being the number of the averaged samples at each time instance.

$$H(z) = \frac{1}{n} \cdot \frac{z^n - 1}{z^{n-1}(z-1)}, \text{ with } n = 2 \tag{4-2}$$

ADIS16364 IMU

For this module, a soft filtering option was chosen to reduce the noise at the respective Xsens' levels. The Bode plots of each filtering stage for this custom configuration are illustrated in Figure 4-20, while the composite response is displayed in Figure 4-21.

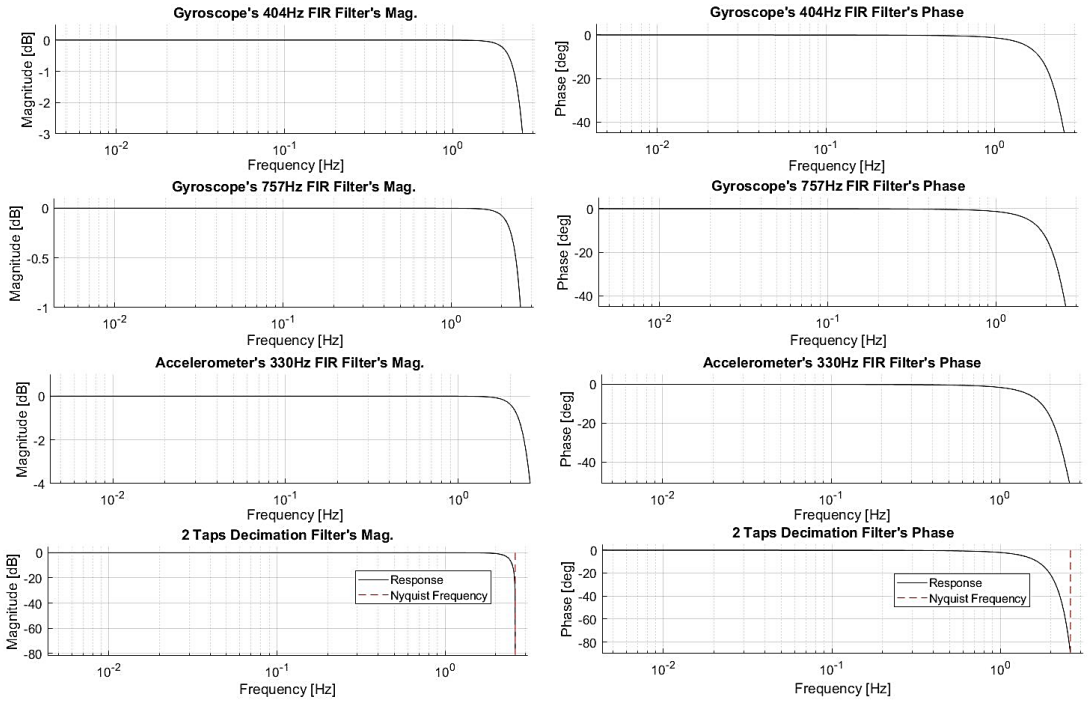


Figure 4-20. ADIS16364 filters.

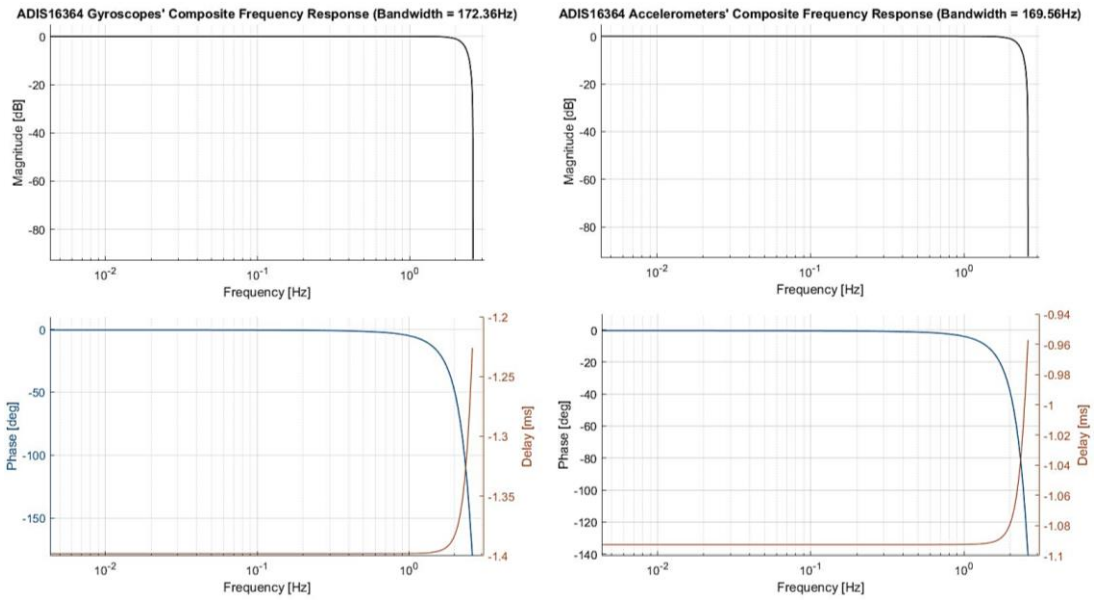


Figure 4-21. ADIS16364 composite frequency response.

The results indicate that ADIS16364 gyroscopes' bandwidth is 172.36 Hz, while the one of its accelerometers is 169.56 Hz. The total phase delay for the gyroscopes is 1.398 ms and for the accelerometers 1.093 ms.

ADIS16375 IMU

ADIS16375 module presents better attributes in terms of sampling rate and precision and thus the impact of the filters in terms of bandwidth is expected to be lower than in ADIS16364 case. The Bode plots of each filtering stage for the chosen configuration are illustrated in Figure 4-22, while the composite response is displayed in Figure 4-23.

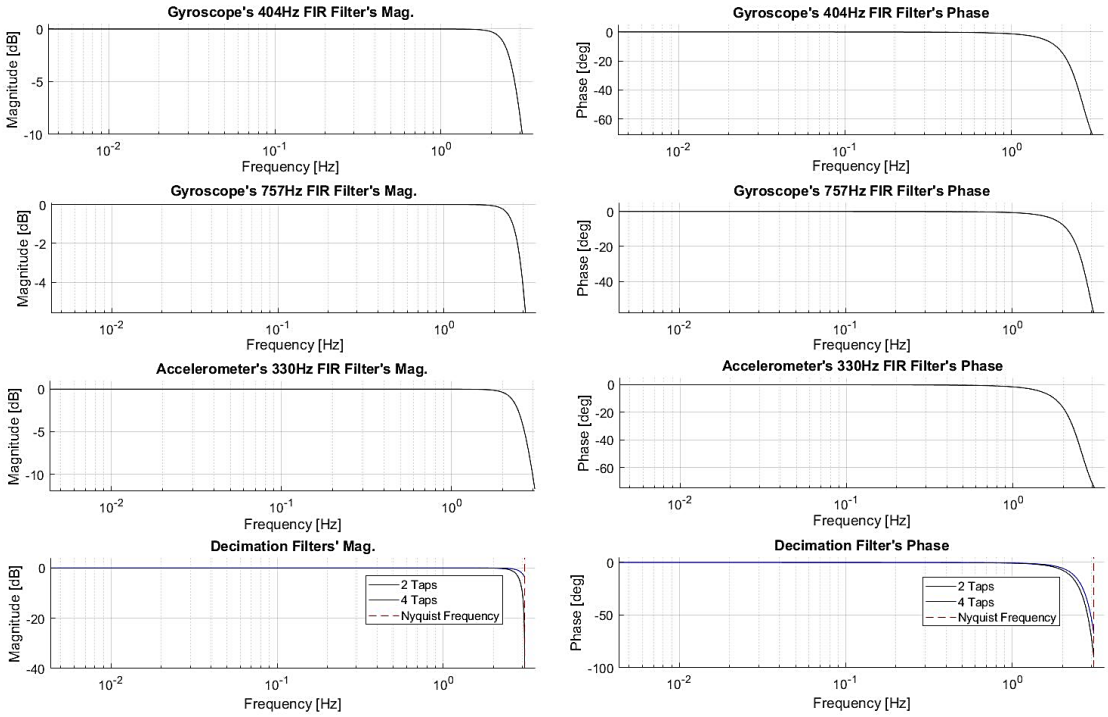


Figure 4-22. ADIS16375 filters.

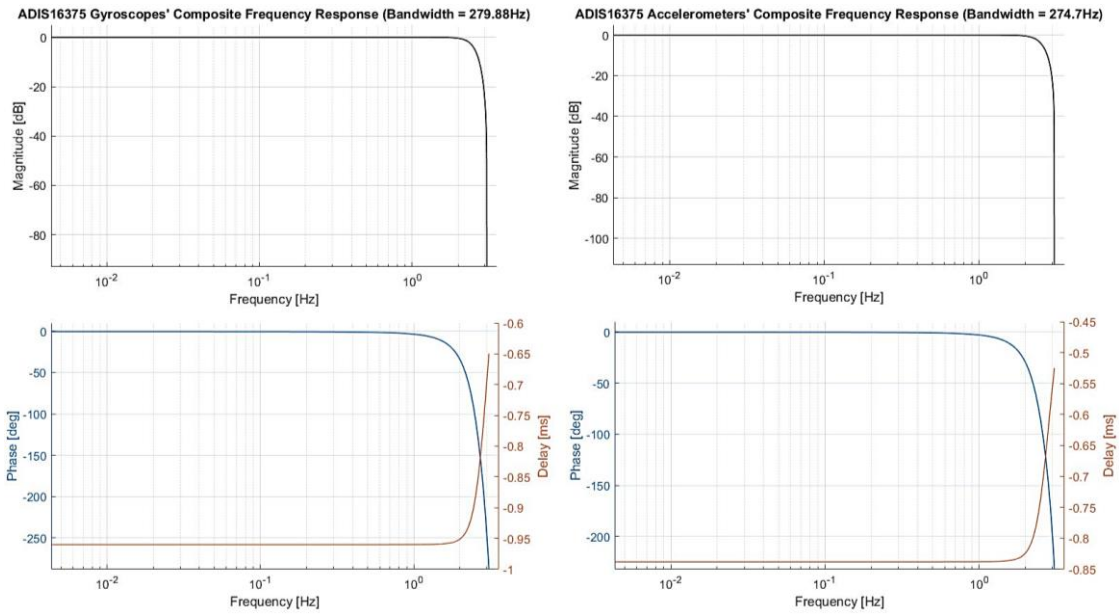


Figure 4-23. ADIS16375 composite frequency response.

The results indicate that ADIS16375 gyroscopes' bandwidth is 279.88 Hz, while the one of its accelerometers is 274.70 Hz. The total phase delay for the gyroscopes is 0.96 ms and for the accelerometers is 0.838 ms.

Generally, the filtering should be altered and adjusted to any application's specifications. In the current case, the goal of the applied filters was to reduce the ADIS163xx noise to the Xsens MTi-200 levels. Towards this, static experiments were executed to determine each sensor's noise levels and biases. The results proved that the applied filters were effective, since the gyroscope's standard deviations were close to those of the Xsens'.

On the other hand, in motion control, any phase delays should be avoided and thus no filter should be used. The noise reduction and generally the convergence of the measurements is dealt with state estimation algorithms, like Kalman filters.

4.8.4 Validation Workflow

The target of the designed experiments was to validate two types of sensors, the gyroscopes and the accelerometers. The former are invariant of the reference frame's location and require only the orientation transformation (${}^{xsens}\mathbf{R}_{adis}$) designated in Table 4-12. On the other hand, to compare the accelerometers' measurements, a fairly complex vector transformation should be executed since linear acceleration during rotation depends on the reference frame's position. The main idea here is that the accelerations must be transformed w.r.t. the same reference frame, since each IMU produces measurements w.r.t. its local frame. Towards this, the ADIS163xx accelerations w.r.t. its local frame (${}^{adis}\mathbf{a}_{adis}$) are transformed to the corresponding acceleration values w.r.t. to the Xsens' frame (${}^{xsens}\mathbf{a}_{adis}$).Thus, the aforementioned transformation depends on the fixed distance vector of the two IMUs (${}^{adis}\mathbf{d}$).

The described procedure is materialized in the equation (4-3). The angular velocity terms $\boldsymbol{\omega}$ included in the formula add unwanted variance to the calculation. Also, the requirement of angular acceleration $\dot{\boldsymbol{\omega}}$ and its subsequent numerical approximation based on noisy measurements amplifies the noise dramatically. So, this procedure results in very noisy acceleration measurements that cannot be compared.

$$ {}^{xsens}\mathbf{a}_{adis} = {}^{xsens}\mathbf{R}_{adis} \cdot {}^{adis}\mathbf{a}_{adis} + \dot{\boldsymbol{\omega}} \times \left({}^{xsens}\mathbf{R}_{adis} \cdot {}^{adis}\mathbf{d} \right) + \boldsymbol{\omega} \times \left(\boldsymbol{\omega} \times \left({}^{xsens}\mathbf{R}_{adis} \cdot {}^{adis}\mathbf{d} \right) \right) \qquad (4\text{-}3) $$

The described procedure should have been unnecessary by the current experimental setup (Figure 4-17), but the latter turned out to be insufficient in eliminating any rotation during the accelerometer validation experiments. Thus, their strict validation is left for future work with better equipment.

On the other hand, the validation of the gyroscopes turned out to be simpler since they are independent of their reference frame's position. Random spatial movements were executed and recorded by the IMUs. After the necessary orientation transformations, the signals were synchronized manually and compared in a common *Matlab* plot. To determine if the results are within the datasheets' prescribed tolerances, an error analysis was of utmost importance.

Error Analysis

The idea here is that measurements acquired from each IMU during the experiments are stochastic quantities given in (4-4). Specifically, they consist of the ideal noise-free part ω, a bias that may be interpreted as Brownian process noise (4-5), and a Gaussian white-noise part (4-6).

$$\hat{\omega}_i = \omega + b_{\omega,i} + w_{\omega,i}, \text{ with } i = \{\text{adis,xsens}\} \tag{4-4}$$

$$\dot{b}_{\omega,i} = w_{b\omega,i} \sim (0, \sigma^2_{b\omega,i}) \tag{4-5}$$

$$w_{\omega,i} \sim (0, \sigma^2_{\omega,i}) \tag{4-6}$$

To simplify the analysis, the biases are considered to be constant since their dynamics relative to each experiment's duration are slow according to the datasheets. To have a metric for the validity of the acquired results, the measurements recorded by the Xsens were subtracted from the ones that were recorded by each ADIS163xx IMU. The resultant error points (4-7) should have a mean value close to the difference of the IMU biases and standard deviations according to (4-8). The notations $V(\cdot)$ and $COV(\cdot)$ represent the variance and covariance of a stochastic variable, respectively.

$$\begin{gathered} \Delta\hat{\omega} = \left(b_{\omega,adis} - b_{\omega,xsens}\right) + \left(w_{\omega,adis} - w_{\omega,xsens}\right) = b_{\omega s} + w_{\omega s} \\ w_{\omega s} \sim \left(0, \sigma^2_{\omega s}\right) \end{gathered} \tag{4-7}$$

$$\begin{gathered} \sigma^2_{\omega s} = V\left(w_{\omega,adis} - w_{\omega,xsens}\right) = V\left(w_{\omega,adis}\right) + V\left(w_{\omega,xsens}\right) - 2COV\left(w_{\omega,adis}, w_{\omega,xsens}\right) \xrightarrow{Uncorrelated} \\ \sigma_{\omega s} = \sqrt{\sigma^2_{\omega,adis} + \sigma^2_{\omega,xsens}} \end{gathered} \tag{4-8}$$

According to the literature [92], the noise of the IMU sensors is a complex matter with multiple causes and sources. Thankfully, some of them dominate and in the current case, only two are considered, namely the inherent sensor noise and the g-sensitivity.

The inherent sensor noise represents the random variation in each gyroscope's output when the latter is operating in static inertial and environmental conditions. IMU datasheets typically offer the Rate Noise Density (RND) parameter to describe their gyroscope's inherent noise ($\sigma_{nd,i}$) with respect to frequency.

Since gyroscopes measure the angular rate of rotation, their response to linear motion introduces errors to their measurements. The corresponding datasheets typically describe this response to linear motion through parameters such as linear acceleration effect on bias or linear-g. According to the manufacturers, the resultant noise's standard deviation is given in (4-9).

$$\sigma_{\omega,i} = \sqrt{\left(\sigma_{nd,i}\sqrt{f_{nbw,i}}\right)^2 + \sigma^2_{gs,i}}, \text{ with } i = \{adis, xsens\} \tag{4-9}$$

That said, the error points (4-7), namely $\Delta\hat{\omega}$, should lie in the vicinity of $b_{\omega s} \pm 3\sigma_{\omega s}$ tolerance, calculated with the above-described method. Outliers are expected since there are multiple error sources. These deviations are present in every stochastic dataset. They may represent sampling errors, change of the system's behavior due to an unspecified cause, or equipment failure. The interpretation of such phenomena in the current dataset will be discussed in the concluding part of this section.

4.8.5 ADIS16364 Results

Gyroscopes

The IMU responses are compared in Figure 4-24, while the error plots in Figure 4-25 suggest that ADIS16364 EtherCAT slave operates according to its design specifications. As expected,

there are outliers in the dataset, the interpretation of which follows in the concluding remarks. The motions recorded by the IMUs were executed by hand.

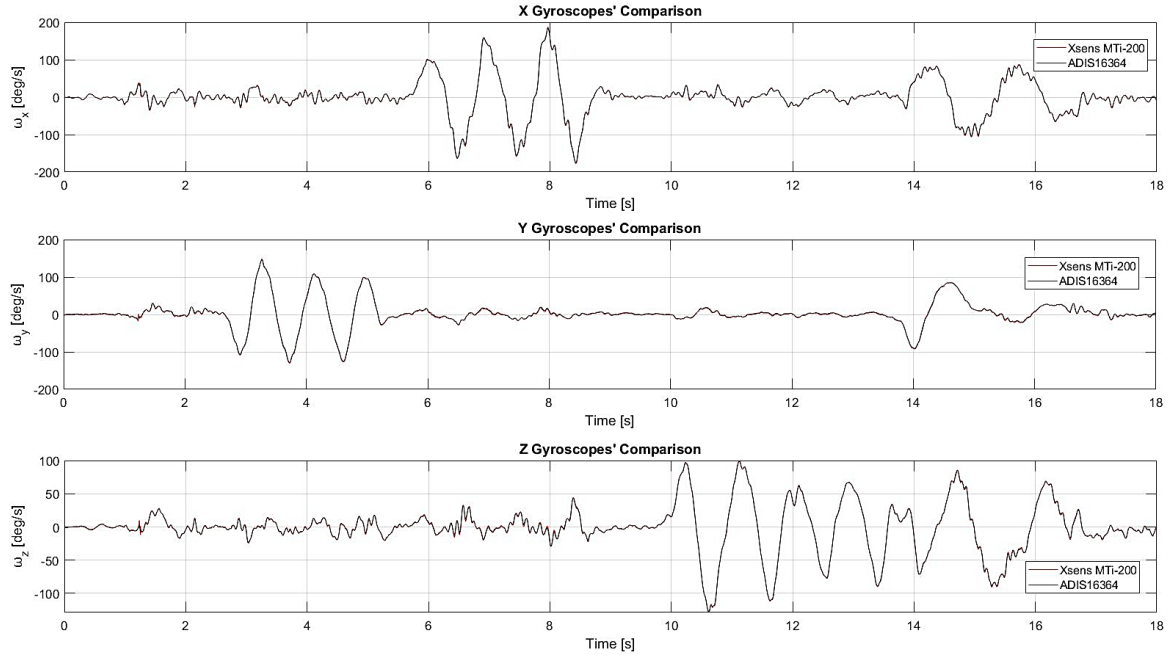


Figure 4-24. ADIS16364 and Xsens MTi-200 gyroscopes' response.

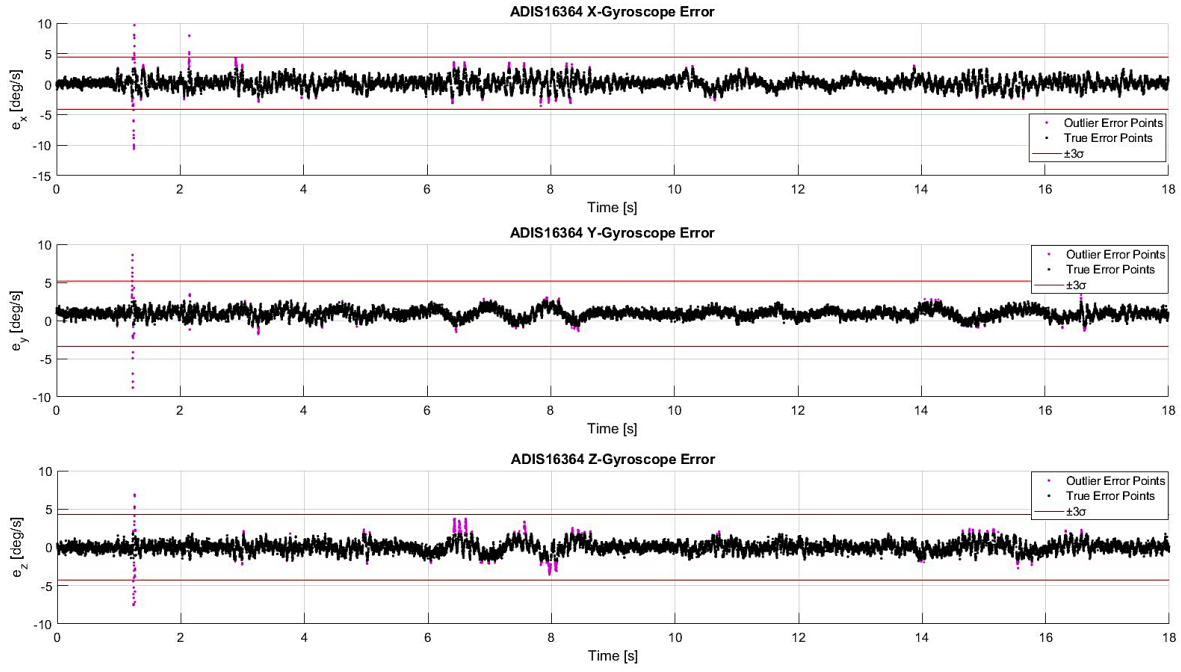


Figure 4-25. ADIS16364 gyroscopes' error.

Accelerometers

The ADIS16364 tracks the Xsens MTi-200 curves (Figure 4-26), but errors occur. Xsens' superior performance is obvious, especially on rapid accelerations and impulses, cross-verifying the specifications of Table 4-14. Note that with different filtering settings and subsequently different bandwidth, ADIS16364 would be able to capture impulses with higher accuracy, but the overall noise in its measurements would increase significantly. As expected

the linear movements that were executed with the rail-guide, resulted in recording the same patterns by the x-axis and y-axis accelerometers, but each one was subjected to accelerations of different magnitudes. The z-axis accelerometer was subjected to gravity and three vertical impulses. The little spikes near the end of the experiment are caused by minor vibrations due to the linear motion.

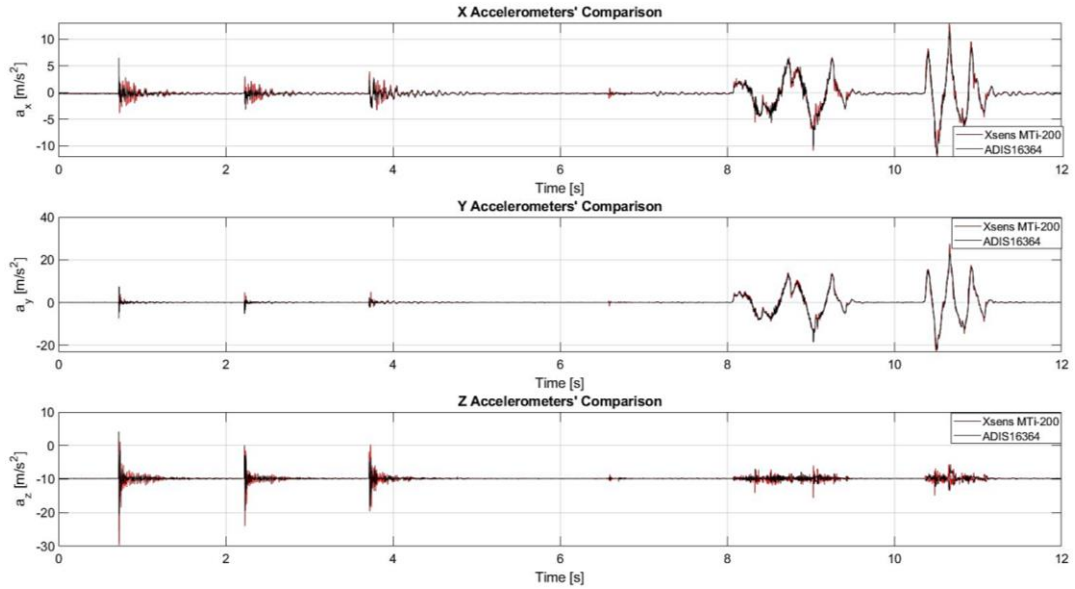


Figure 4-26. ADIS16375 and Xsens MTi-200 accelerometers' response.

4.8.6 ADIS16375 Results

Gyroscopes

Likewise, the gyroscope responses are compared in Figure 4-27, while the error plot of Figure 4-28 suggests that ADIS16375 has superior performance in comparison with ADIS16364. Note that in the case of ADIS16375 almost none of the error points violate the 3-sigma rule and at the same time these limits are significantly lower than the ones of ADIS16364.

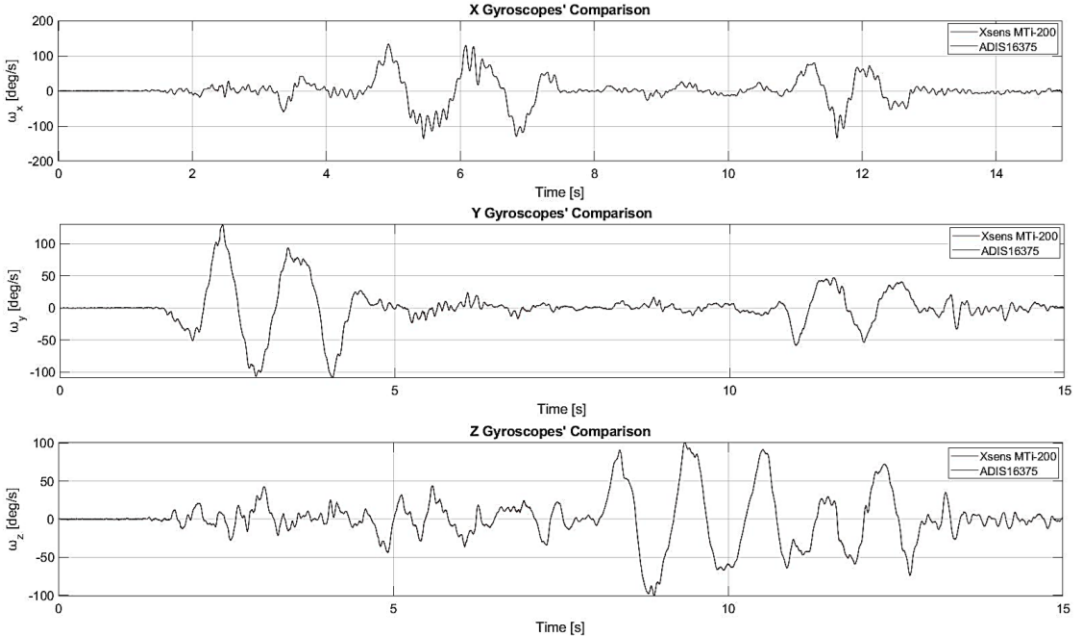


Figure 4-27. ADIS16375 and Xsens MTi-200 gyroscopes' response.

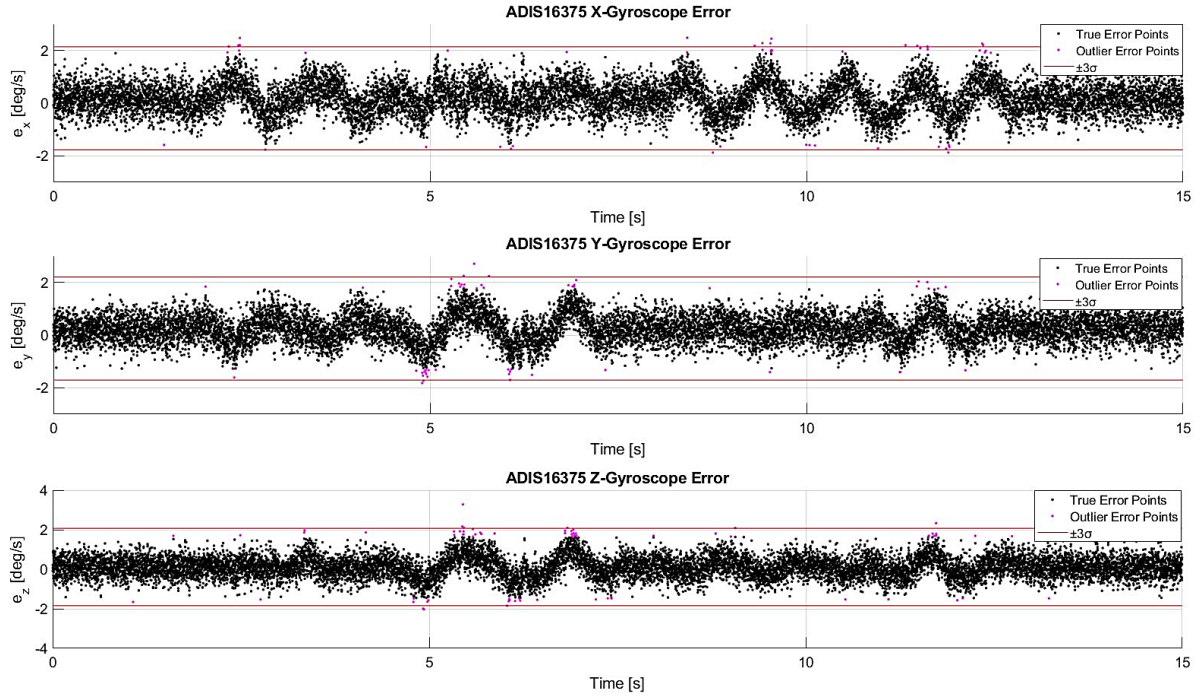


Figure 4-28. ADIS16375 and Xsens MTi-200 gyroscopes' error.

Accelerometers

The ADIS16375 accelerometers present good performance since the curves in Figure 4-29 track the Xsens' readings with fair accuracy. Nevertheless, due to the ignored contributions of the minor angular motions, the error analysis could not be applied. As expected the linear movements that were executed with the rail-guide, resulted in recording the same patterns by all of the accelerometers, but each axis was subjected to accelerations of different magnitudes.

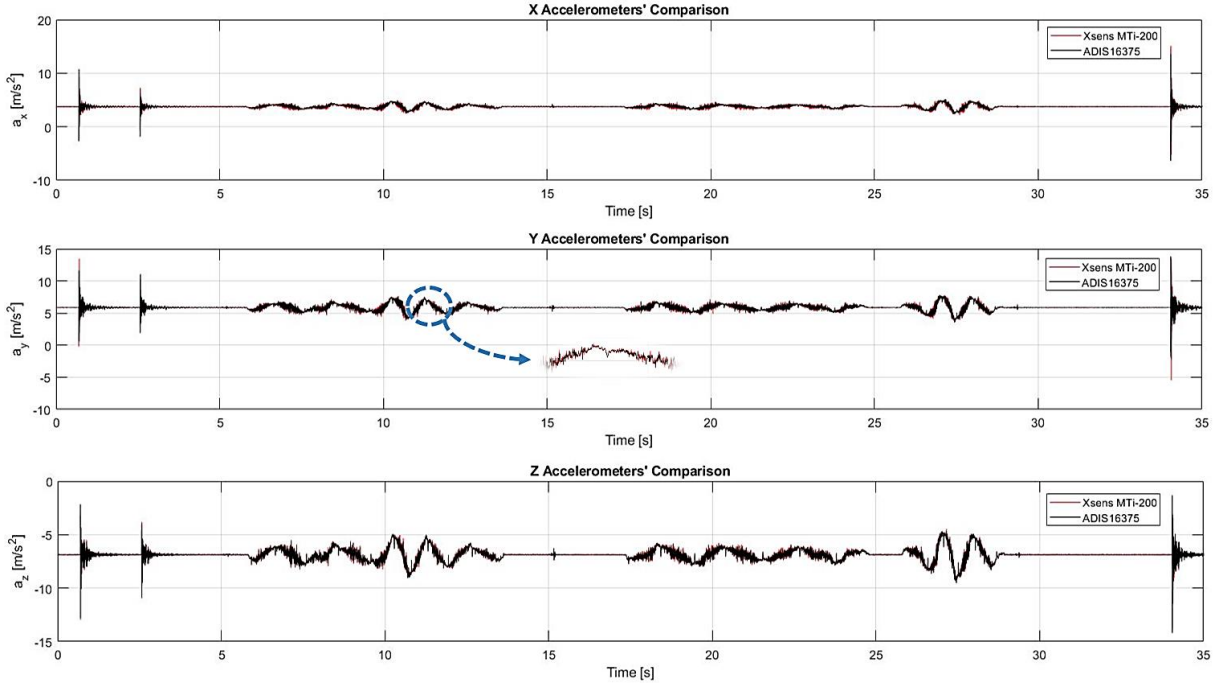


Figure 4-29. ADIS16375 and Xsens MTi-200 accelerometers' response.

4.8.7 Validation Conclusions and Discussion

The designed experiments indicate that both ADIS163xx IMU slaves operate as expected. The acquired measurements are valid since their error is within the prescribed tolerances. The integration of the IMU slaves in the Laelaps II network has been proven to be flawless since it did not require any adjustments in the existing network. In all experiments, outliers are present. These behaviors are expected since:

- There is a bandwidth difference between each ADIS163xx IMU and the Xsens. Therefore, high-frequency harmonics and other impulses are captured with larger magnitudes by the Xsens (Sensitivity Errors).
- The applied filters, establish some time-lags in the IMUs' response since certain phase delays are introduced.
- The IMUs taking part in the experiment are not aligned accurately, thus certain misalignment errors downgrade the experiments' quality.

Apparently, ADIS16375 presents a superior performance in terms of accuracy and bandwidth. Nevertheless, both IMUs are considered ready to be integrated into the Laelaps II EtherCAT network.

5 Laelaps II Experiments

This chapter describes a typical experiment's procedure and guarantees the proper setup of the robot. It comes as a step-by-step guide that incorporates all of the required actions that ensure the safety not only for the system itself, but most importantly, for its users. An effort was made to account for most, if not all, of the possible situations that a user may confront during an experiment. However, this does not imply that all of the required knowledge regarding the proper operation of the robot is encapsulated solely into this chapter. The reader is advised to read the previous chapters, thoroughly, before advancing to the current one.

5.1 Typical Experiment's Procedure

Laelaps II is a complex machine and its operation is not straight forward. Thus, the typical experiment's proposed workflow for Laelaps II is presented below. The user is encouraged to deviate from the suggested steps if circumstances dictate it.

Hardware Checks

1. Check the integrity of the main power supply channels and set the voltages to appropriate values. For the low power line, a typical value is 9 V, while for the high power the default is 48 V (which is the motors' nominal voltage [46] [93]).
2. Ensure that the absolute encoder assemblies, along with their electrical wirings are properly mounted and connected. Details may be found in the corresponding section of this thesis (Section 3.3). Special attention should be given to the *JST-XH* cables not to incur any loads, otherwise, the danger to be cut off is lurking.

EtherCAT Network Checks

3. Verify that the EtherCAT nodes are connected via their respective Ethernet cables, according to the prescribed topology that is set in TwinCAT's project. The default configuration is the following:
    a. Hind Right Leg's EtherCAT Slave (HR_LEG).
    b. Hind Left Leg's EtherCAT Slave (HL LEG).
    c. Fore Left Leg's EtherCAT Slave (FL_LEG).
    d. Fore Right Leg's EtherCAT Slave (FR_LEG).
    e. Analog Devices ADIS16364 IMU's EtherCAT Slave (ADIS16364_IMU).
    f. Analog Devices ADIS16375 IMU's EtherCAT Slave (ADIS16375_IMU).

The slaves listed in the TwinCAT's solution explorer must have the same order in the physical network. So, the master PC is connected to HR_LEG's input port (Figure 3-19), while its output port is connected to the HL_LEG's input port and so on. In other words the list in TwinCAT's solution explorer represents the actual EtherCAT network. So, any changes in the slave order of the physical network must be made to its virtual counterpart, in TwinCAT's solution explorer, too.

4. Verify that Delfino's power supply configuration is set to external for both of its channels. To do this, consult Section C.8. Furthermore, turn on the low power supply of the robot and activate each slave's regulator by pressing the onboard buttons. To cross-reference that all of the slaves are properly powered, the Delfinos' and motor drives' LEDs should start flashing as soon as the nodes are turned on.

5. Download the correct firmware for each slave. Open TI's Code Composer Studio and locate the corresponding firmware in the Project Explorer's pane. If from a past setup, the firmware has already been downloaded to the respective nodes, this step may be skipped. In any other case, it is required for the desired projects to be properly imported and configured inside the IDE. Towards this, it is of utmost importance to refer and execute the steps described in Sections 3.4 and 4.5 for the legs and IMUs, respectively. If the absolute encoder checks are necessary before the experiment, enable **ABS_DEBUG** Predefined Symbol. If the absolute encoders are not going to be used, both **END_STOP** and **AUTO_STARTUP** Predefined symbols must be disabled.

TwinCAT Project Setup

6. This step setups the TwinCAT 3 project. To do this, an extensive guide has been created in Appendix A. If the TwinCAT 3 project has already been created, feel free to skip this step. Also, if not all the slaves are used, remove the redundant PLC variables to reduce the size of the experiment's log file. Note that if an experiment does not require all of the slaves, the latter ones should be disabled in TwinCAT 3 (*Right-Click* on the slave at the solution explorer's pane and opt for *Disable*) and subsequently be removed from the physical bus. This only applies to the last slaves on the bus. No intermediate slaves in the chain can be removed with this procedure.
7. Next, write the ESC's EEPROM memory. Again, if from a prior use the slaves' EEPROMs are properly configured, this step may be ignored. Also, this procedure makes sense if the application's PDOs have been modified. In other words, the execution of step 5, does not necessarily require the current step's actions. Nevertheless, to write the EEPROM of a single slave using TwinCAT 3:
    a. *Left-Click* on the master entry, inside the solution explorer's menu.
    b. Navigate to its *Online* tab.
    c. *Right-Click* on the desired slave and choose the *EEPROM Update…* option.
    d. Lastly, opt for the corresponding slave's ENI file, located inside the drop-down list.

    The described procedure is displayed in Figure 5-1. As for a more informative guide, please refer to the respective VPRS TwinCAT Tutorial video [94].

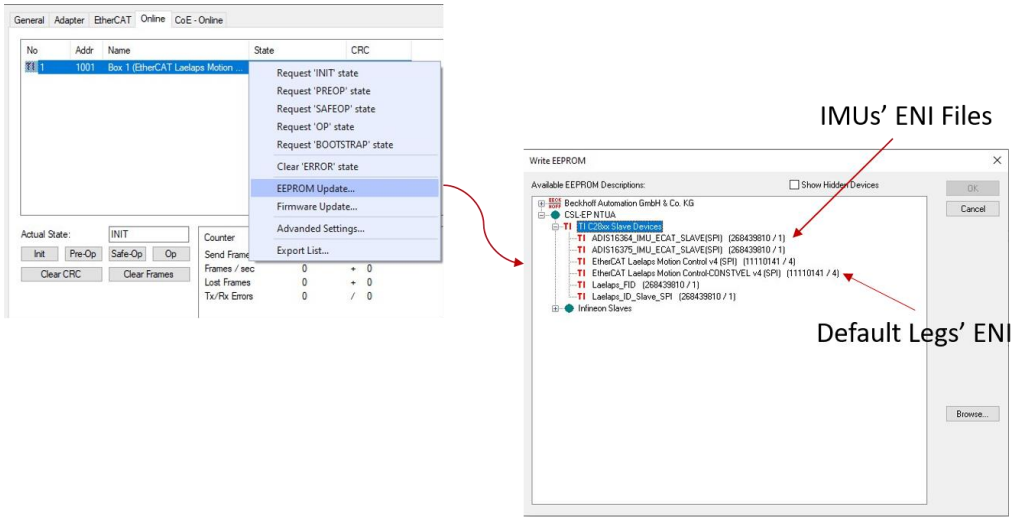
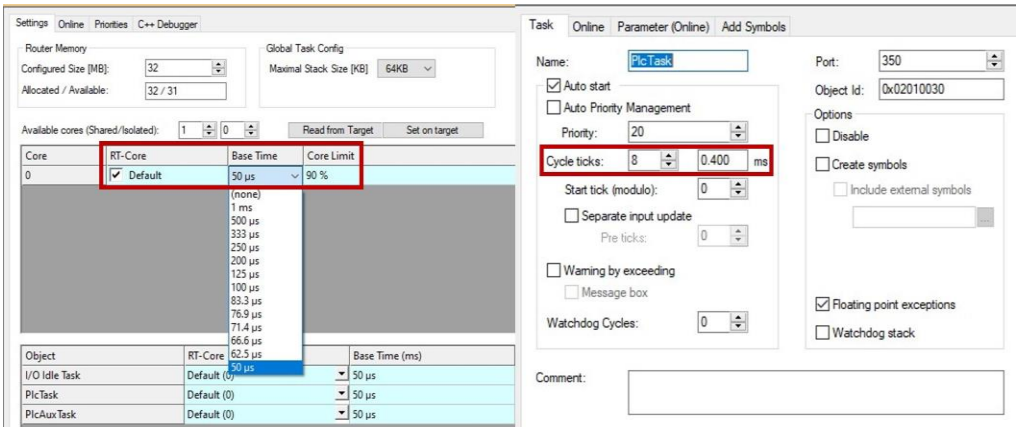


Figure 5-1. EEPROM update option.

Note that after the necessary slaves' EEPROM updates, the TwinCAT node should be refreshed as suggested in the previously mentioned video. If this procedure is performed inside the main Laelaps II motion control TwinCAT project, all PLC variables are going to lose their connections and a tedious process like the one illustrated in Appendix A (step 17) ought to be repeated. Another remark is whether the desired ENI is located in the aforesaid drop-down list. If not, follow steps 6 and 7 of the Appendix A.

8. Next, the EtherCAT cycle should be checked. The cycle time comes as an integer multiple of the TwinCAT's base cycle-time. In the current case, this cycle must be configured to the minimum possible value (50 μs). This setting may be configured at the *Real-Time* tab as Figure 5-2a suggests. The default cycle's period is 400us (**2.5 kHz**), while the corresponding TwinCAT setting is located at the solution explorer's tab, under *SYSTEM -> Real-Time -> PLC Task* (Figure 5-2b). At last, hit the *Reload Devices* button located in the main ribbon, illustrated in Figure 5-3.



(a) **(b)**

Figure 5-2. (a) Base cycle time setting. (b) PLC Task's cycle time setting.

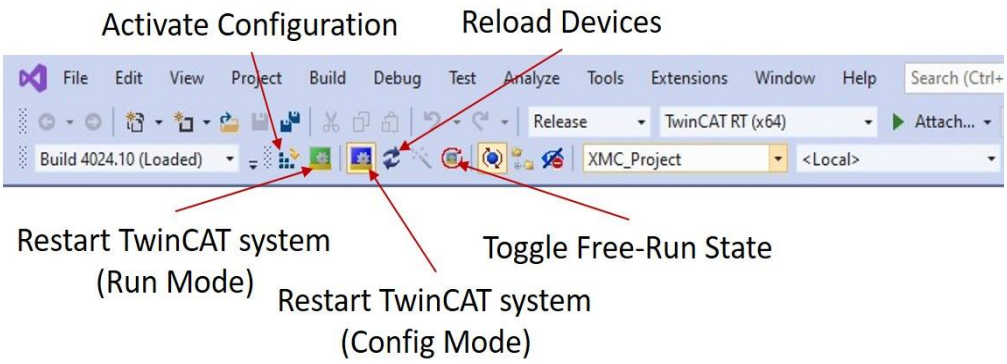


Figure 5-3. TwinCAT main ribbon.

9. A good practice is to verify the configured cycle-times after the refresh. To do this, navigate to each slave's object in the solution explorer's pane and choose the *DC* tab. Moreover, opt for *DC-Synchron* mode as the *Operation Mode* and left-click on *Advanced Settings*. The default settings for every slave on Laelaps' bus are presented in Table 5-1, while the whole step's procedure is illustrated in Figure 5-4.

Table 5-1. DC-Mode TwinCAT slave settings.

DC Parameter	Default Value [µs]
Sync 0 cycle	400
Sync 0 shift time	0 (Auto)
Sync 1 cycle	400 (x1 Sync0 cycle time)
Sync 1 shift time	30

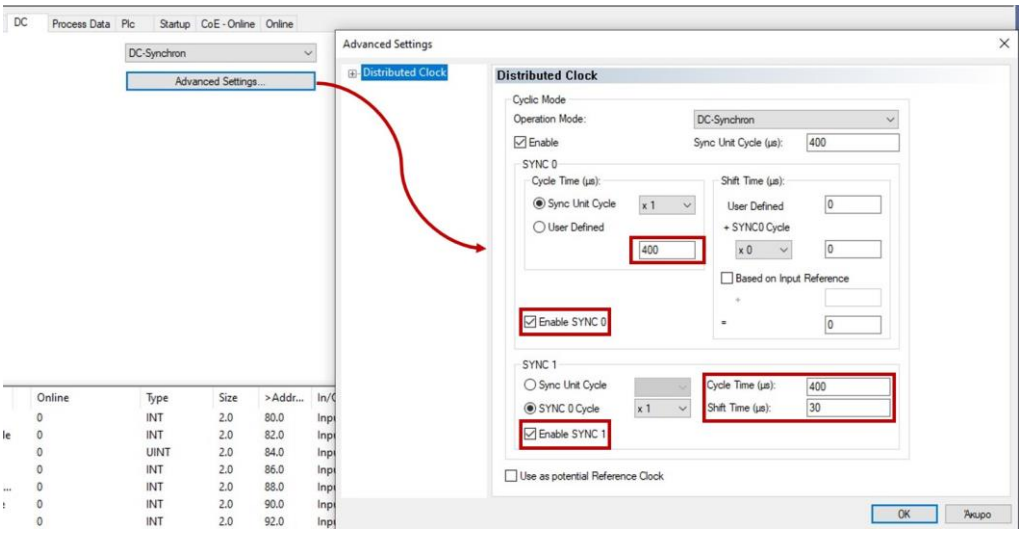


Figure 5-4. Verify the slave's DC TwinCAT settings.

10. Activate TwinCAT's Run Mode. To do this, left-click on *Activate Configuration* Button (Figure 5-3). This builds the solution and restarts TwinCAT in Run Mode configuration. If a warning about run-time licenses appears, just copy the displayed code to the designated text-box. By the time the procedure finishes, all the slaves in the Online Tab of the Master should be in **OP mode**. If any errors arise in any of the slaves, press the Delfino's reset button and repeat this step.
11. Login and Start the PLC Task, by clicking on the buttons highlighted in Figure 5-5.

Figure 5-5. Login and Start PLC Task's buttons.

Leg Encoder Checks

12. Check the encoders. Start a draft *SCOPE* recording, by clicking on *Record* illustrated in Figure 5-6.

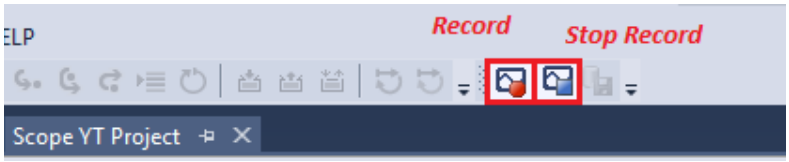


Figure 5-6. Record and Stop Record buttons.

13. Move the legs manually and verify that the corresponding graphs display the expected feedback. With this in mind, the incremental encoders' positive directions are clockwise

with respect to the lateral view that points to Laelaps body, for both leg configurations. This corresponds to the right legs executing a backward swing and the left ones performing a forward one.

14. Check and reset the absolute encoders, if necessary. This step may be skipped if a reset has already been performed in a prior setup. Nevertheless, resetting the encoders requires the manual positioning of each leg to the intended zero-angle configuration. Then, a voltage pulse (3.3 V to 5 V) should be momentarily applied to the corresponding reset pins (***JZ1*** for the knee's encoder and ***JZ2*** for the hip's) of the Delfino's shield (Figure 5-7).

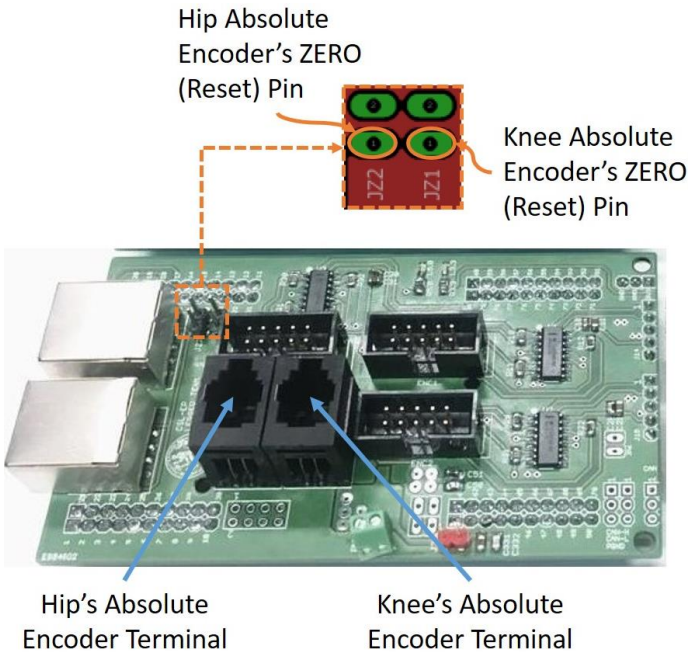


Figure 5-7. Absolute encoders' reset pins.

15. A very important test, especially during the first-run after a modification, is to observe the absolute encoders' *SCOPE* charts and ensure that they are correctly configured. To display them, an indirect way is used. By enabling the **ABS_DEBUG** symbol during the firmware download step, the EtherCAT velocity variables (Table 3-3) are in fact the absolute encoder readings. Knees' velocity graphs correspond to their absolute encoders and the same goes for hips. The absolute encoders are configured in software to have the same positive direction (clockwise) as the incremental ones. Also, by resetting each slave and compare the aforementioned plots, it should be clear if the initialization has been successful.
16. After the encoders' hardware reset, press all of the Delfinos' reset buttons to refresh each slave. Now, all of the incremental encoders' charts should display the correct angle.
17. If end-stop routines are enabled, by swinging the leg back and forth, their impact in the PWM charts may be observed. To display the control efforts, enable **State Machine**. After the checks are completed, **disable** it again.
18. Now that all of the checks are completed, it is time to get on the main experiment. At this point, if the **ABS_DEBUG** symbol was activated for the checks, the respective Delfinos need firmware refresh with this symbol removed from the CCS' *Project Properties -> Predefined Symbols* tab.

19. A good practice is to run the experiment without high power before the actual one. This way, by viewing the TwinCAT charts the planner and the control efforts can be verified. To do this, execute steps 20 to 35 without activating the high power supply. Remember to disable **State Machine** and reset all the output PDOs before progressing to the actual experiment.

Main Experiment

20. Given that the slaves are all in OP Mode and PLC Task's Login and Start buttons have been pressed, it is time to stop the remaining data recordings and start new ones, by clicking on the buttons highlighted in Figure 5-6. This step along with the next ones are more intuitively demonstrated by the corresponding TwinCAT 3 tutorial video [94].
21. Open the outputs of the PLC task, shown in Figure 5-8. From here, the entire system can be controlled.

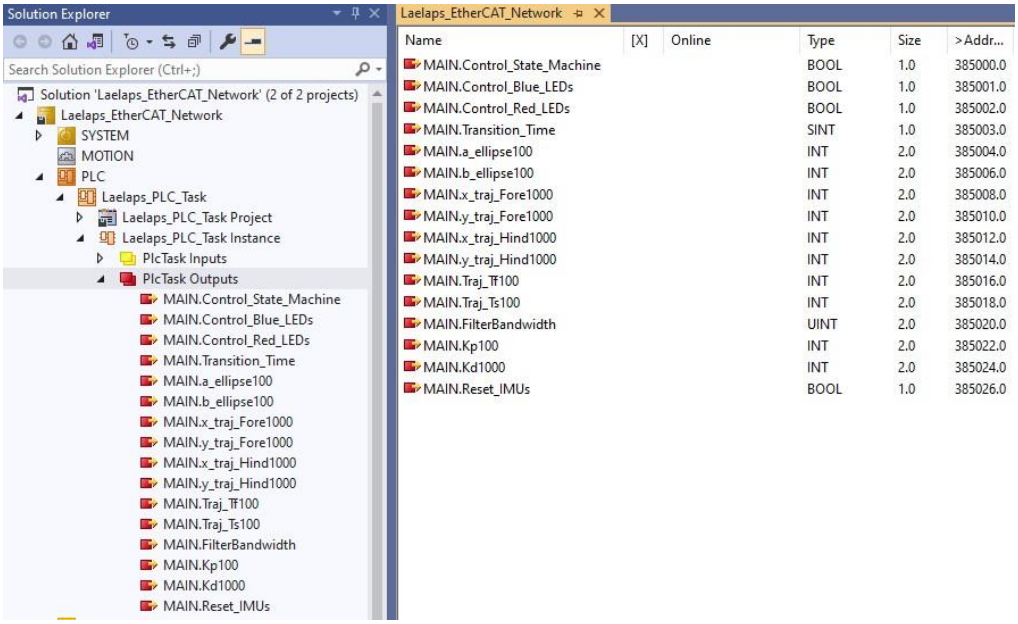


Figure 5-8. PLC Task's Outputs tab.

22. First, define the **Transition Time** smoothing factor, the default value of which is 2 [s].
23. Next, define the **center coordinates** of the ellipse (fore and hind x-y coordinates). The tested range for x values is 0 to 15 [mm], while for the y values is 585 to 598 [mm].
24. Moreover, the phase for each leg should be set accordingly. This cannot be executed inside the PLC Outputs tab, but rather under each slave's object at the TwinCAT 3 solution explorer's pane. According to what gait mode is intended, the phases should be set accordingly. The time phase comes as a percentage of the total step's period. For trotting set it to 50 [%] at either set of diagonal legs.
25. If the IMUs are used, activate the desired readings and set filtering options to the corresponding IMU node object, inside the solution explorer's pane.
26. Set **Stance and Flight phase periods** [100*s]. For example, to execute a step every second, set both values to 50.
27. Set the **Filter Bandwidth** at the default value of 20 [Hz], but only if the velocity gain is necessary.
28. Next, turn on the **high-power supply**. No motion is expected yet.
29. Switch the **State Machine** to *HIGH* state. No motion is expected yet.

30. Now with extra care, start increasing the **Proportional Gain** slowly. The legs should be observed to move to their designated zero-angle positions. The tested gain-values are up to 9000.
31. To damp any oscillations and impulses that the proportional gain results in, add **Velocity Gain**. However, it is highly sensitive to noise and should be set **no higher** than 50.
32. Laelaps II is ready to walk. Increase the **vertical axis (b)** of the ellipse to start a static gait. Common values are up to 5 [cm] for a stance height of 595 [mm].
33. To move Laelaps, increase the ellipse's **horizontal axis (a)**. A common range is 0 to 7 [cm] for a stance height of 595 [mm].

In an emergency, shut down the power supply and/or the State Machine. There is also the LAELAPS_HALT* EtherCAT switch *that makes the quadruped standstill.

34. To end the experiment:
    a. Gradually decrease and make the **a** of the ellipse zero.
    b. Gradually decrease and make the **b** of the ellipse zero.
    c. Now that the quadruped is standing still, secure its body and make the **State Machine** zero.
    d. Turn-off the readings of the IMUs, if used.
    e. Shut down the **power supplies.**
35. Stop the recordings and save the results by exporting them to the desired format. To do this, navigate to the *Export to CSV* option under the *SCOPE* menu (Figure 5-9).

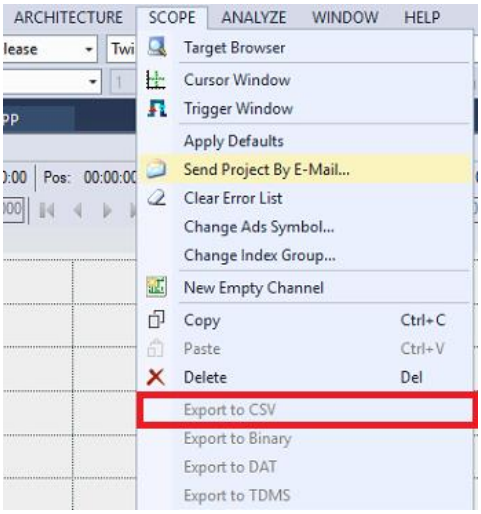


Figure 5-9. TwinCAT Export to CSV option.

6 Parameter Identification of Laelaps Leg

In the coming years, quadrupeds may be called to perform in a variety of different unstructured environments. The mobility advantage that they present makes them ideal for a wide range of applications in the oil industry [95], agriculture, etc. To address these issues, engineers seek to design efficient systems capable of adapting to different conditions. A major prerequisite in tuning their performance is to have an accurate model representation. In most cases, the CAD-based approximations are insufficient, since they provide rough estimations about the inertial parameters and none about the friction characteristics of the real system. Therefore, parameter identification has been a major subject of research throughout the years.

The objective of this chapter is to design methods for estimation of the parameter of Laelaps' legs. By knowing Laelaps' actual model, its governing dynamics could be studied and exploited. Also, the simulations would give more realistic results and help improve its performance. The proposed framework creates an appropriate trajectory that excites the leg's dynamics, and subsequently estimates its parameters after executing the trajectory.

This chapter consists of three major sections; friction identification, experimental inertial parameter identification, and the whole-leg identification framework. At the beginning of each section, a certain amount of theory is presented for the reader to gain a better insight into the discussed concepts. Moreover, the main part of each idea is discussed along with some concerns to become fully functional in the current robot's setup. Furthermore, the results of a simulation in *Matlab* are presented, thoroughly. In the end, concluding remarks and verification are considered.

6.1 Theory and Equipment

Towards the leg model identification, the first part of the study is focused on the estimation of the joint's internal friction characteristics. To verify the methods and equipment that are used in this study, several benchmark tests were designed.

Initially, only the motors are considered and specifically the knees' DC motors. According to Maxon's standard specification sheets [96], winding resistance cannot be measured accurately, due to its dependence on the rotor's position. On the other hand, the motor's speed constant may be verified with a simple sampling scheme. According to the datasheet [93], Maxon's RE50 200 Watt motor (Part num. 578298) has the following specifications (Table 6-1).

Table 6-1. Maxon motor's datasheet values.

$\boldsymbol{R_a}$ **[Ω]**	0.608
$\boldsymbol{K_V}$ $\left[\mathbf{rad/(s\cdot V)}\right]$	10.6814
$\boldsymbol{K_T}$ **[N·m / A]**	0.0934

Note that Maxon is using the reciprocal of the commonly used voltage constant K_V^*, as equation (6-1) suggests. Beware that Maxon's convention is adopted throughout this thesis.

$$\omega_m = K_V \cdot V_{emf} = \frac{1}{K_V^*} \cdot V_{emf} \tag{6-1}$$

Brushed DC Motors come with a simple model that may be viewed in Figure 6-1. The parameters discussed above are given by the equations (6-2) and (6-3) below, as far as the steady-state operation is considered.

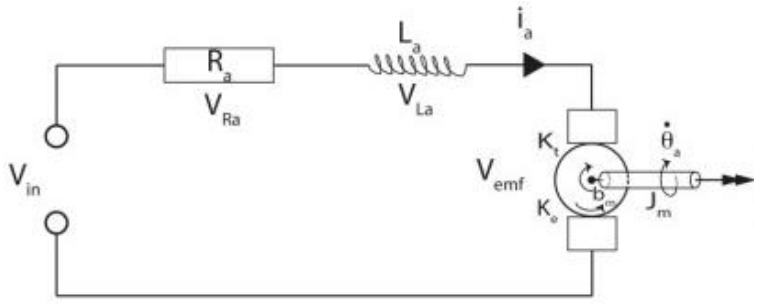


Figure 6-1. Generic DC brushed motor model.

$$K_T = \frac{T_m}{i_a} = \frac{1}{K_V} \tag{6-2}$$

$$K_V = \frac{\omega_m}{V_{in} - i_a R_a} \tag{6-3}$$

The used sampling equipment is chosen based on low-noise and accuracy criteria. For voltage measurements, *Agilent's MSOX3014A* oscilloscope was connected at the motor's terminals. To determine the current consumption *FLUKE's 289 True RMS* multimeter was used, with bias compensation. The equipment's attributes are listed in Table 6-2.

Table 6-2. Sampling equipment's parameters.

	Range	Resolution	Accuracy
FLUKE 289	400 mA	0.01 mA	0.15 %
Agilent MSOX3014A	135 V (RMS)	Adjustable	N/A

The sampling was executed at a constant velocity, to eliminate the various inertial effects and transients that accelerations come with. For the speed estimation, a HEDL incremental quadrature encoder [97] was employed, connected to a Delfino LaunchPad (LaunchXL-F28379D). The major nonlinearities in the friction model are expected in the vicinity of kinematic friction, especially right after the break-away torque of the static friction. Hence, a low-speed calculation (6-4) was implemented in firmware, with X being the angle-step, while t_1 and t_2 the timestamps.

$$\omega = \frac{X}{t_2 - t_1} \tag{6-4}$$

This means that the drift between the actual and approximated speed is expected to increase at higher velocities. Therefore, a deviation in K_V motor's constant is expected at the aforementioned speeds. Nevertheless, the speed estimation is sufficient for cross-verifying the motor's datasheet. The goal here is to test the performance and validity of the experimental equipment with a simple benchmark procedure.

6.1 Actuator's Speed Constant Determination

Towards the experimental validation of the motor's speed constant and subsequently its torque constant, multiple measurements were taken and averaged. This way, a valid conclusion about the performance of the proposed sampling method could be drawn.

Generally, the speed constant is calculated at the motor's rated voltage and the corresponding no-load speed, provided by the accompanying datasheet. The equations (6-2) and (6-3) are applicable for any voltage that the ideal DC motor operates (Figure 6-1). However, the real motor's operation implies certain electromechanical and magnetic losses. According to Maxon [98], the motor's speed constant is not affected significantly by these losses, so the equation (6-5) is in effect.

$$\omega_m \approx K_V \cdot \left(V_{in} - i_a R_a\right) \tag{6-5}$$

To verify that claim, as previously mentioned, a sampling was conducted for several voltage levels. The results of the discussed method are presented in Table 6-3. Note that the calculated relative error is w.r.t. the Maxon's speed constant, listed in Table 6-1. The approximated K_V constant is calculated with equation (6-3), using the manufacturer's resistance R_a of Table 6-1.

Table 6-3. Speed constant experimental results.

ω_m [rad / s]	V_{in} [V]	I [A]	Approx. K_V [rad / (s · V)]	Relative Error [%]
110.945404	10.533	0.0781	10.6221246	0.555
211.772919	19.911	0.1080	10.70156891	0.189
270.919373	25.429	0.1146	10.7084864	0.253
344.794922	31.922	0.1256	10.84940657	1.573
$\overline{K_V}$ [rad / (s · V)]			**10.72039662**	**0.365**

The main cause of the deviations presented in Table 6-3 are the various magnetic and frictional losses that the motor's operation comes with. Moreover, as the speed increases the velocity measurement quality degrades, partially due to the low-speed calculation that is implemented. This is depicted in the increased error that the approximated motor's speed constant comes with, at high velocities. Moreover, the increased error of the first result may be caused, partially, by the noise of the current measurements, since the Noise-to-Signal (SNR) ratio becomes significant in low magnitudes as well as by the contribution of the gearbox's friction. Still, the deviation is fairly low and as such, the experiments indeed approach the values of the datasheet [93].

6.2 Leg's Friction Parameters Identification

Initially, only the motor with its gearbox are considered, since details about the friction of each component in the drivetrain are sought out. To have the whole joint's friction model, the belt drive should be included in a future experiment.

6.2.1 Motor-Gearbox Friction Identification

To measure the response and come with a valid model, a vast amount of constant velocity experiments were executed (Figure 6-3). The sampling points were carefully chosen to be in the vicinities that the most “exotic” behaviors are expected. Such phenomena arise at the boundary of the breakaway torque. In Figure 6-2, several different friction models are presented.

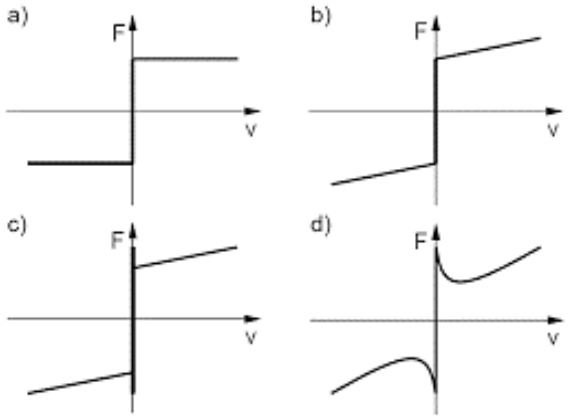


Figure 6-2. Different friction regimes.

Moreover, the sampling accounts only for the positive direction of the motor, neglecting any deviations in the reverse direction. This choice was made to simplify the overall process, since tests indicated that the difference was insignificant. However, if a more accurate model is necessary, the procedure described in this section should be performed separately for each direction of the motor, since friction in general is not symmetric w.r.t. the zero angular velocity.

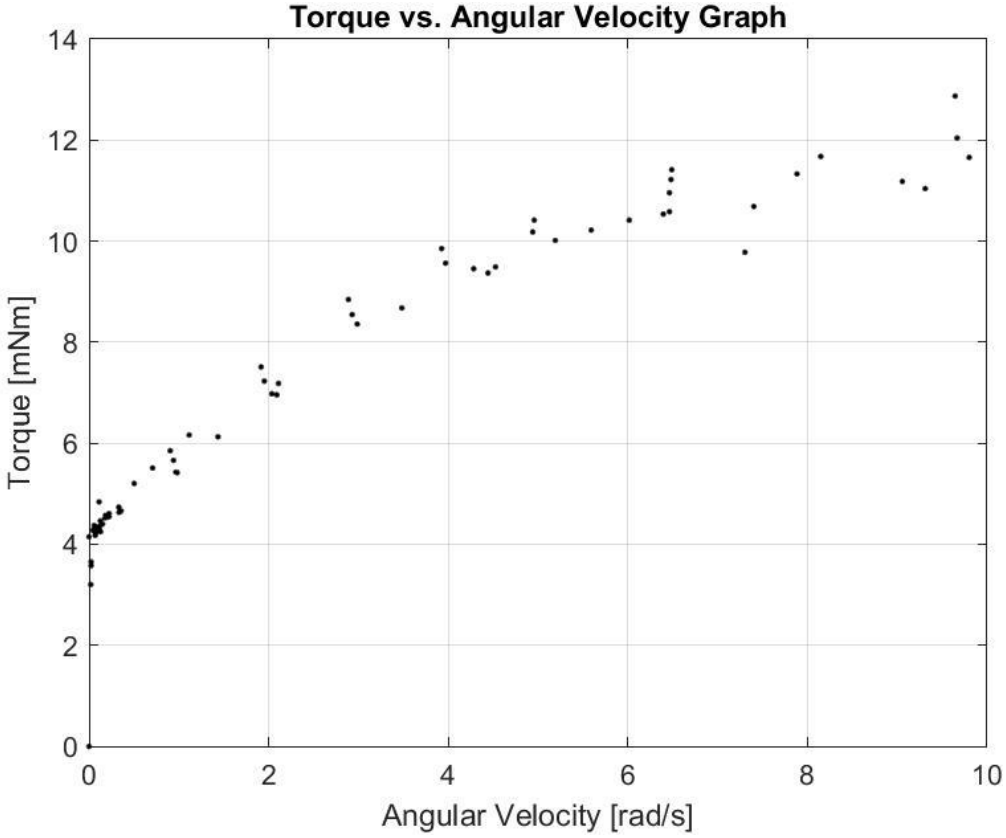


Figure 6-3. Sampled data.

For a wide rangle of voltages, the motor's current consumption and encoder's speed approximation were being recorded. Then by solving the equation (6-2) for T_m, the motor's torque has been obtained. Finally, the sampled angular velocities were translated at the high-torque side of the gearbox, using the gearbox's reduction ratio [47] (Part No. 223090).

From a simple observation of the dataset shown in Figure 6-3, certain assumptions can be drawn.

- A static friction component may be observed in the zero-speed vicinity.
- At medium to high speeds, the friction is nonlinear with respect to the angular velocity.
- There is no clear evidence of nonlinear Stribeck behavior directly after the breakaway torque barrier.

From the above-discussed observations, three models were promoted to be tested on how accurate representations of the above dataset might be. The first is linear and accounts only for the static and linear viscous friction components. A simple linear least-squares scheme was employed to determine its coefficients. The other two are nonlinear models that include terms to account for the aforementioned nonlinear behaviors. To solve these problems, an iterative solver was used. Note that the source files of the friction identification procedure are located in [99].

CASE I: Linear Friction Model

In this case, a simple friction model is created with a linear least-squares regression scheme. It is ideal for a model-based control design in terms of execution time, while at the same time it presents a good R^2_{adj} index that will be discussed in Models Comparison section. The formula is given below (6-6), along with the graph of Figure 6-4 for a visual interpretation of the result. In the mentioned equation q_i represents the angle, $i_{a,i}$ the input current, and f_k the unknown regression coefficients of an arbitrary sample (i).

$$\tau_i = f_1\dot{q}_i + f_2 sign\left(\dot{q}_i\right)$$

$$\mathbf{Y}\cdot\boldsymbol{\pi}=\boldsymbol{\tau}, \text{ with } Y_i=\left[\dot{q} \quad sign\left(\dot{q}\right)\right]_i, \boldsymbol{\pi}=\begin{bmatrix} f_1 \\ f_2 \end{bmatrix}, \text{ with } f_k \geq 0,\ k=\{1,2\}\ \& \tag{6-6}$$

$$\tau_i = \mathrm{K_T}\cdot i_{a,i} \text{ with } i \in \mathbb{N} \text{ the sample number}$$

The velocity term is scaled with respect to the high-torque side of the gearbox. The results are presented in Table 6-4.

Table 6-4. Linear model's coefficients.

Term	Value
$\boldsymbol{f_1}$ $[Nm\cdot(rad/s)^{-1}]$	8.9131e-04
$\boldsymbol{f_2}$ $[Nm]$	0.0047

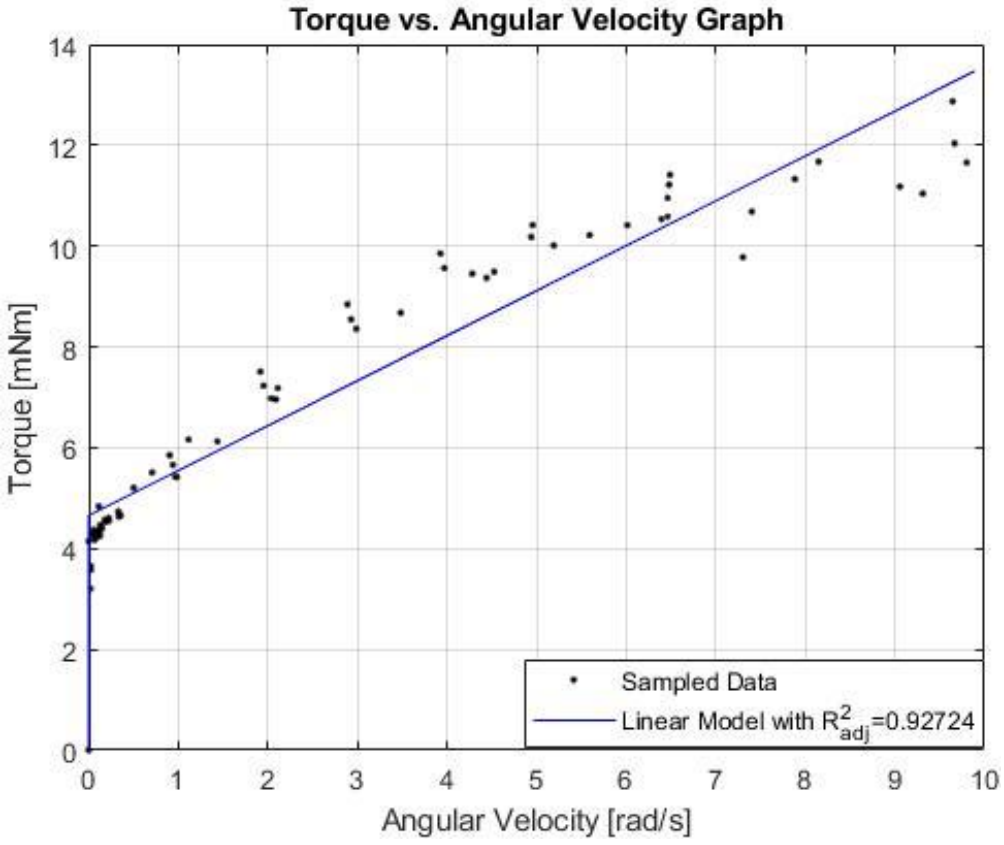


Figure 6-4. Linear model graph.

CASE II: Nonlinear Model 1

In this case, a nonlinear friction model is designed to account for the Stribeck term [25]. To solve the problem, an iterative method was used, with the *Matlab* function *lsqnonlin()*. The function (6-7) was supplied along with its analytical jacobian (6-8) to increase the method's performance. In these equations q_i represents the angle, $i_{a,i}$ the input current, and f_k the regression coefficients of an arbitrary sample (i).

$$\tau_i = f_1\dot{q}_i + f_2 sign(\dot{q}_i) - f_3 sign(\dot{q}_i) e^{-\frac{|\dot{q}_i|}{f_4}}, \text{ with } f_k \geq 0 \, and \, k = [1,4]$$
$$\tau_i = \mathrm{K_T} \cdot i_{a,i} \text{ with } i \in \mathbb{N} \text{ the sample number} \quad (6\text{-}7)$$

The Jacobian is calculated as below:

$$J_i = -\begin{pmatrix} \frac{\partial \tau_i}{\partial f_1} & \frac{\partial \tau_i}{\partial f_2} & \frac{\partial \tau_i}{\partial f_3} & \frac{\partial \tau_i}{\partial f_4} \end{pmatrix} = -\begin{pmatrix} \dot{q} & sign(\dot{q}) & -sign(\dot{q}) e^{-\frac{|\dot{q}|}{f_4}} & -\frac{f_3|\dot{q}|}{f_4^2} sign(\dot{q}) e^{-\frac{|\dot{q}|}{f_4}} \end{pmatrix}_i \quad (6\text{-}8)$$

The results may be viewed in Table 6-5 and Figure 6-5.

Table 6-5. Nonlinear model 1 coefficients.

Term	Value
f_1 $[Nm \cdot (rad/s)^{-1}]$	4.7574e-05
f_2 $[Nm]$	0.0120
f_3 $[Nm]$	0.0080
f_4 $[rad/s]$	3.9575

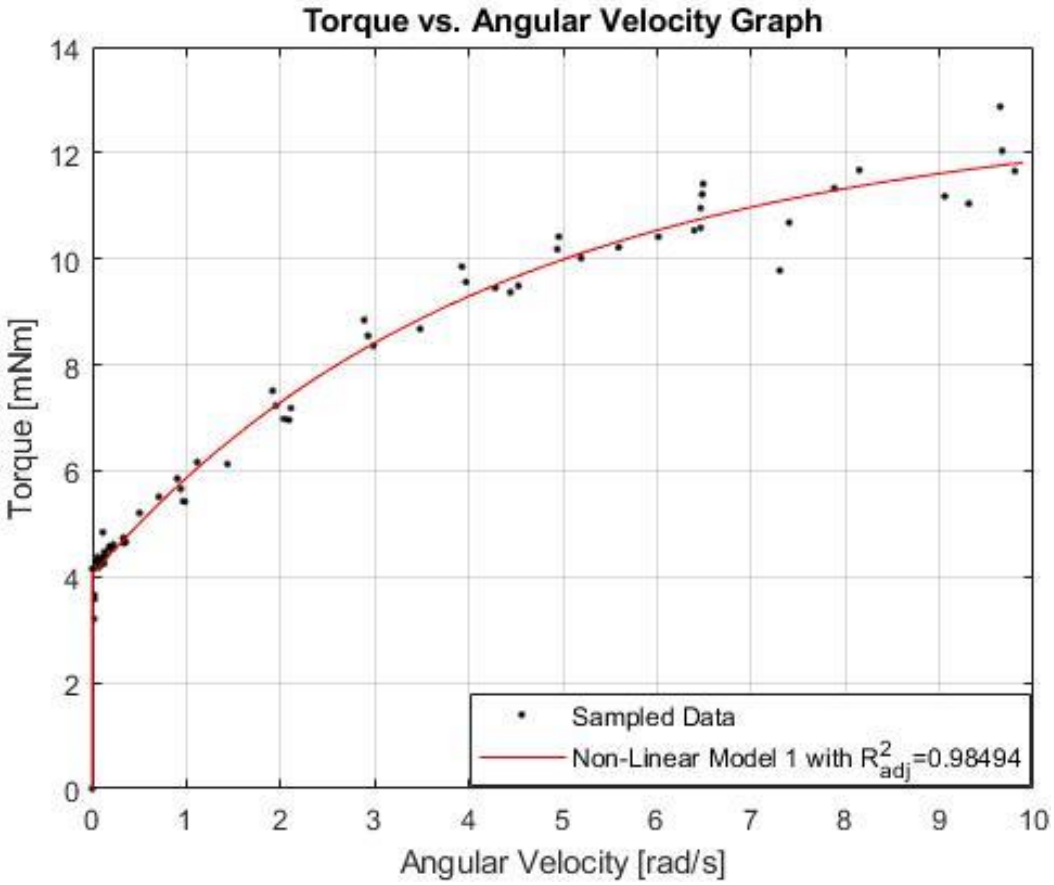


Figure 6-5. Nonlinear model 1 graph.

CASE III: Nonlinear Model 2

Like in the previous model, here a nonlinear friction model is employed to account not only for the Stribeck term alone, but also for the exponential decay in the transition from kinetic to low-speed viscous friction [25]. To solve the problem of nonlinear regression, an iterative method was used, with the function *lsqnonlin()*. Like in the previous case, both the function (6-9) and the analytical jacobian (6-10) were supplied to the algorithm. In these equations, q_i represents the angle, $i_{a,i}$ the input current, and f_k the regression coefficients of an arbitrary sample (i).

$$\tau_i = f_1\dot{q}_i + f_2 sign(\dot{q}_i) - f_3 sign(\dot{q}_i) e^{-\frac{|\dot{q}_i|}{f_4}} - f_5 sign(\dot{q}_i) e^{-\frac{1}{f_6|\dot{q}_i|}}, \text{ with } f_k \geq 0 \, and \, k = [1,6] \qquad (6\text{-}9)$$
$$\tau_i = \mathrm{K_T} \cdot i_{a,i} \text{ with } i \in \mathbb{N} \text{ the sample number}$$

$$J_i = -\left(\frac{\partial \tau}{\partial f_1} \quad \frac{\partial \tau}{\partial f_2} \quad \frac{\partial \tau}{\partial f_3} \quad \frac{\partial \tau}{\partial f_4} \quad \frac{\partial \tau}{\partial f_5} \quad \frac{\partial \tau}{\partial f_6} \right) = -\left(\dot{q} \quad sign(\dot{q}) \quad -sign(\dot{q}) e^{-\frac{|\dot{q}|}{f_4}} \right. \qquad (6\text{-}10)$$
$$\left. -\frac{f_3|\dot{q}|}{f_4^2} sign(\dot{q}) e^{-\frac{|\dot{q}|}{f_4}} \quad -sign(\dot{q}) e^{-\frac{1}{f_6|\dot{q}|}} \quad -\frac{f_5}{f_6^2|\dot{q}|} sign(\dot{q}) e^{-\frac{1}{f_6|\dot{q}|}} \right)_i$$

The results may be viewed in Table 6-6 and Figure 6-6**.**

Table 6-6. Nonlinear model 2 coefficients.

Term	Value
f_1 $[Nm \cdot (rad/s)^{-1}]$	4.7573e-05
f_2 $[Nm]$	0.0120
f_3 $[Nm]$	0.0080
f_4 $[rad/s]$	3.9575
f_5 $[Nm]$	0.0040
f_6 $[(rad/s)^{-1}]$	1.4406

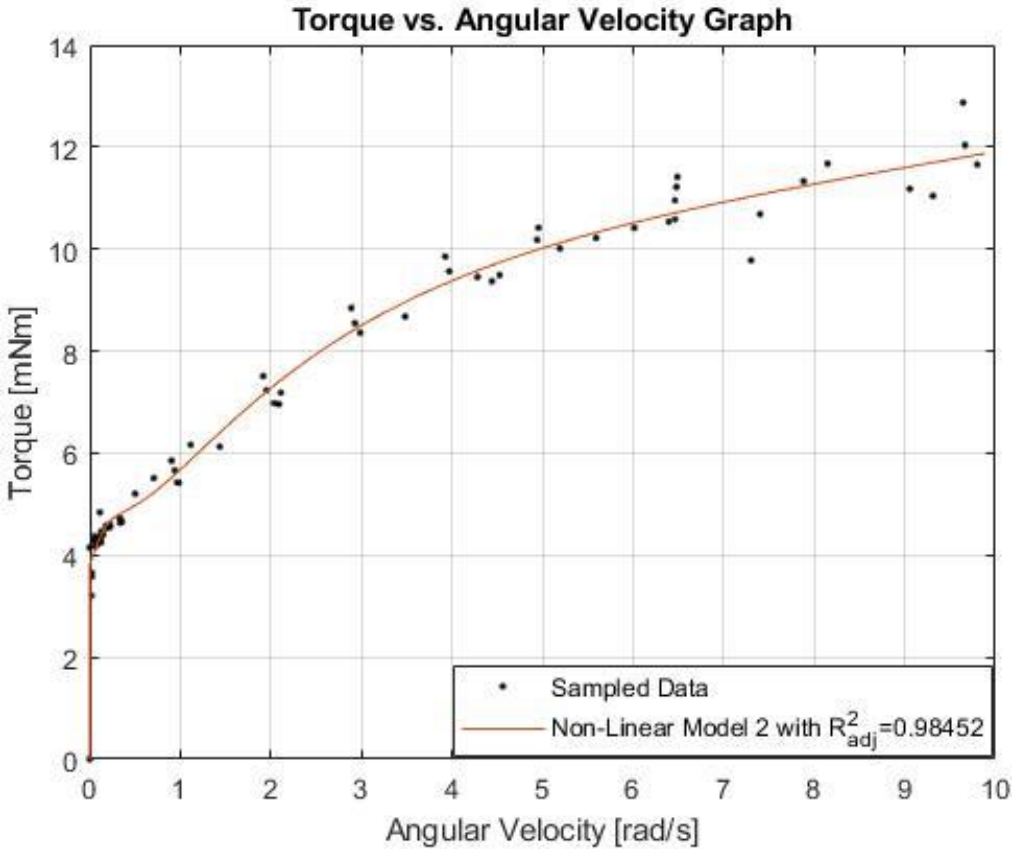


Figure 6-6. Nonlinear model 2 graph.

Models Comparison

The three models are compared concerning their quality and accuracy with the R^2_{adj} regression index. Here, the adjusted version is used to account for the different number of terms of each model. That index is given by the formula (6-11), with n representing the number of sampling points and p the degrees of freedom (terms) of the model.

$$R^2_{adj} = 1 - \left(1 - R^2\right)\frac{n-1}{n-p-1}, \text{ with } R^2 = 1 - \frac{\sum_i^n \left(y_i - \hat{y}_i\right)^2}{\sum_i^n \left(y_i - \bar{y}\right)^2} \tag{6-11}$$

For an intuitive comparison of the different models, their regression curves are presented together in Figure 6-7.

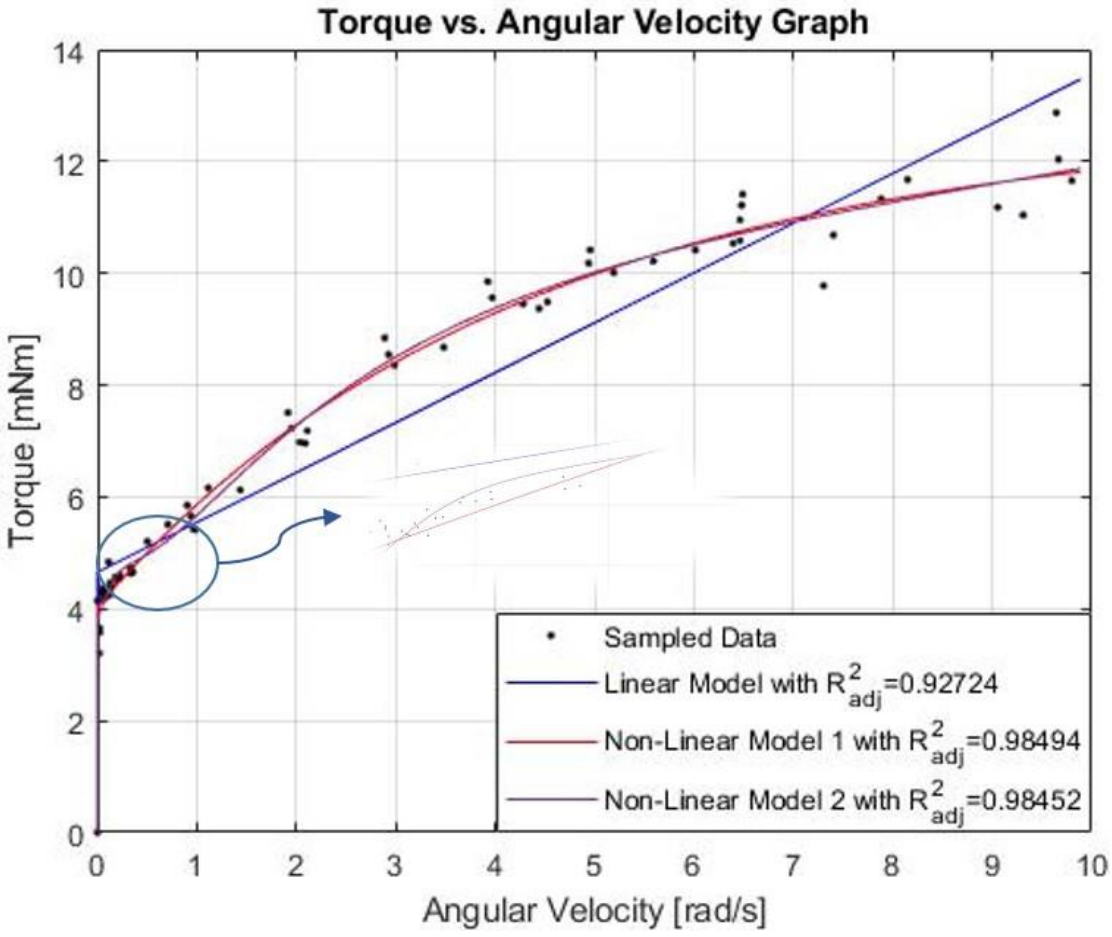


Figure 6-7. Friction model comparison graph.

Conclusion

All of the models come with a trade-off regarding their computational complexity and accuracy. From the above analysis and the R^2_{adj} index, the first nonlinear model is the best approximation of the joint's friction model among the three. But it is up to the developer to decide which is more suitable for a specific application. In many time-critical applications (e.g model-based control), the low processing requirements of the linear model may compensate for its accuracy loss. However, the additional computational cost may be insignificant for a simulation that in general accuracy is promoted, since there are no real-time processing requirements. Whatever the case may be, the developer should consider multiple parameters before choosing, to end up with an optimal design.

6.2.2 Joint's Friction Identification

The Laelaps II leg joints, as previously stated, consist of a geared motor that gives motion to a belt drive to furtherly increase the output torque. That belt system consists of two pulleys with rolling bearings. Such mechanical components have an impact on the joint's total frictional behavior. It is widely accepted that in bearings' friction characteristics, the viscous type dominates. So, the expected response in the measurements is an increase of the torque levels in curves of Figure 6-7 and a decrease of their nonlinear behavior.

On the other hand, the belt's mechanics may increase nonlinearities, due to the elasticity that this study neglects. However, this elasticity is expected to be insignificant when the belts

have sufficient pretension and operate under the proposed by their manufacturer, nominal conditions.

To come with the whole joint's friction model, future experiments, similar to the above ones, may be executed. It would be interesting to compare the results and observe the belt's contribution to the system's dynamics.

6.3 Leg's Inertial Parameters Identification – Method 1

To cross-check the results of the main identification framework, the inertial parameters of the robot should be estimated with simple, yet insightful experiments. The key idea here is to isolate each unknown parameter. By exploiting the underlying physics, each unknown quantity can be approximated easily. As for the accuracy of the resultant estimation, it strongly depends on the used experimental equipment and the human factor. To reduce the error impact, by executing the procedure multiple times and averaging the results, better accuracy could be achieved.

At first the required equipment will be discussed. Next, the motor and the gearbox inertial identification experiment is concerned. Note that the actual experiments are left for future work. In the current section they are described and the identification problem is presented, along with some implementation concerns.

6.3.1 Required Equipment

The experiments dictate the necessity of current and kinematic measurements. To accomplish these tasks an extensive market search was made to find the appropriate equipment. For the kinematic measurements, a HEDL encoder was used. The speed and acceleration data were derived by differentiating the encoder's position readings. As for the current measurements, after a review of the available hardware, two sensor evaluation modules were purchased. The specifications of each current sensor are presented below.

In general, current sensing is a very delicate procedure. The measurements are prone to noise and they are heavily dependent to the implementation. So, a careful design is of utmost importance, taking into account the application's specific needs and the hardware's limitations. The reader is strongly encouraged to refer to [28] that investigates every aspect of current sensing. To get current measurements that they could be eventually translated to the motor's torque, a different approach should be adopted in case of the knees' DC motors and the hips' brushless, since the latter ones are three-phased motors.

INA253 EVM

The INA253 EVM evaluation module consists of shunt-resistors. It produces high-precision current readings and integrates three different gains to account for any level of sensitivity an application would require. A list of the module's key features is presented in Table 6-7, but the reader is encouraged to consult the manufacturer's datasheets [100] [101], to gain a better understanding of the platform. The board is shown in Figure 6-8.

Table 6-7. INA253 EVM specifications.

Parameter	Value
Measured current range (max) [A]	±15
Shunt inductance [nH]	3

Shunt resistor tolerance [%]	0.1
Shunt resistor [mΩ]	2
Bandwidth [kHz]	350
Offset current [mA]	15
Offset drift [µA/°C]	125
Common voltage range [V]	-4 to 80

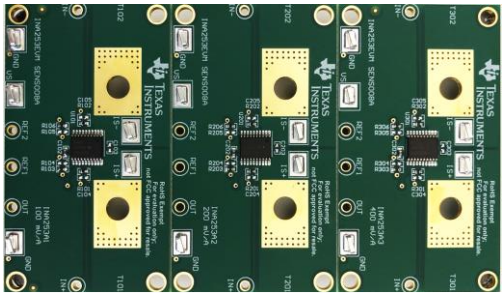


Figure 6-8. INA253 EVM board overview.

TMCS1100 EVM

The TMCS1100 constitutes a galvanically isolated Hall-effect current sensor [102] capable of accurate DC and AC measurements. The most important characteristics extracted from the manufacturer's datasheets [103] [104] are presented in Table 6-8. Among them, the one that stands out is the wide voltage range that this sensor operates, which offers flexibility to its integration into any design.

Table 6-8. TMCS1100 EVM specifications.

Parameter	Value
Measured current range (max) [A]	±46
Total error (typical) [%]	±0.4
Low power voltage [V]	3 to 5.5
Isolation rating [kV_{RMS}]	3
Bandwidth [kHz]	80
Offset current [mA]	15
Offset drift [µA/°C]	40
High power voltage range [V]	±600

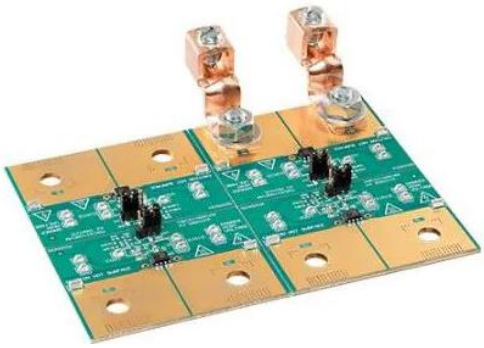


Figure 6-9. TMCS1100 EVM overview.

6.3.2 Motor – Gearbox Inertial Identification

In the current section, the identification problem of the motor – gearbox system's inertias is presented. The formulation can be easily expanded to involve the inertias of the belt's gears. But as previously mentioned, the idea here is to isolate each parameter. So firstly, the motor – gearbox system's inertias should be estimated and then in a similar manner the belt-gears could be added, given that the belt's elasticity is neglected by the current formulation. This way, the inertias provided by the motor's datasheet [46] [93] could be cross-checked and the whole method's performance can be evaluated.

In the future implementation, it is recommended to connect the sensor to the high-voltage side of the motor, to avoid ground-side distortions and noise. It is widely known that the motion equations of multi-body systems can be written in linear form with respect to the model's parameters. With the nonlinear friction entering the equation, that claim becomes invalid. However, by knowing the friction's regression model (discussed in Section 6.2), with a single subtraction on the right-hand side of the equation (6-12), the linear model of the equation (6-13) is acquired. The problem is summarized in (6-13).

$$\left(I_m + I_g\right)\ddot{\theta} + \tau_{friction} = K_T \cdot i_a \tag{6-12}$$

$$\mathbf{Y} \cdot \boldsymbol{\pi} = \boldsymbol{\tau}, \text{ with } Y_i = \ddot{\theta}_i,\ \boldsymbol{\pi} = \left(I_m + I_g\right)\ \&\ \tau_i = K_T \cdot i_{a,i} - \tau_{friction,i} \tag{6-13}$$

Since the model is simple and linear, no special trajectory is necessary for sampling. This fact simplifies significantly the experiment's workflow. For example, the motor could be subjected to a ramp input voltage [105] that could be controlled manually or even to white noise. The only hardware requirements is an encoder for the required kinematic measurements and a current sensor.

6.3.3 Identification of the Leg's Parts Inertial Parameters

To estimate the inertial parameters of the leg, namely the mass, the inertia and the position of the Centers of Mass (CoM) of its parts, simple experiments that isolate each parameter were designed. The literature provides several different approaches with different prerequisites. Their performance relies heavily on the used equipment.

Mass Estimation

The mass estimation should be very simple, by using a weight gauge. Note that CSL-EP's laboratory owns very accurate gauges that should be up for the task.

Center of Mass (CoM) Position Estimation

There are multiple ways to estimate the CoM position of an arbitrary object. The precision of each method is bound to the quality of the experimental setup that is used.

- **Seesaw mechanism.** This method relies on the creation of a seesaw-like mechanism that supports the measured part. By moving the part in the axial direction the goal is to find the position, which makes the seesaw horizontal. This method requires the creation of such a mechanism, with low friction. A conceptual design is illustrated in Figure 6-10.

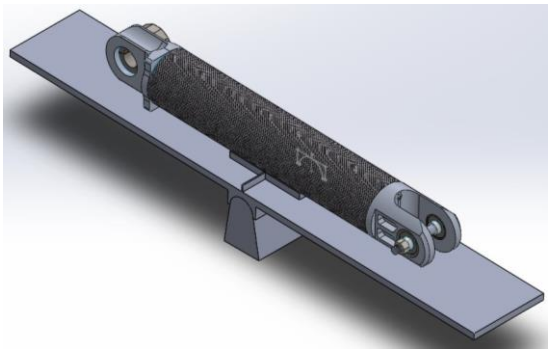


Figure 6-10. Conceptual design of a seesaw mechanism.

- **Rotational mechanism.** This technique uses a rotational joint that supports the leg's segment. In its two edges, weights are mounted to make the part horizontal. By taking the static equilibrium and deriving the free-body's equations of the aforesaid part, its center of mass position can be determined. This method is illustrated in Figure 6-11.

Figure 6-11. A rotational mechanism for CoM determination.

- **Two-point hanging method.** This setup relies on the hanging of the object in question from two arbitrary points, one at a time. The CoM is determined as the intersection of the two resultant vertical lines that start from each hanging point.
- **Weight-gauges method.** This method is similar to the rotational mechanism. It is simpler and more straightforward. Both ends of the part are positioned on weight-gauges, which measure the subsequent reaction forces. By taking the free-body diagram of the leg-segment, its CoM can be determined. This method's simplicity and accuracy make it ideal for a future experiment with the lab's equipment.

Inertia Estimation

The inertia estimation of the leg segments requires attention because it involves measurements of a nonlinear dynamical system that in most cases is linearized around a stable point. So slight deviations in measurements and conditions may give heavily erroneous results. There are a couple of approaches available, while the most suitable ones follow. The below methods consider a part of mass m, inertia w.r.t. its CoM I_{CoM} and CoM-distance from the rotation point C_m.

- **Simple Pendulum:** The simple pendulum is one of the simplest nonlinear dynamical systems. If the original nonlinear ODE that describes the system is linearized to the lower stable equilibrium point, certain dependencies arise among its parameters. So, a justified choice is to measure the period of a free oscillation and derive the inertia with (6-14). For the derivation of this formula, one may consult Appendix J. The main drawback of this setup is that the object should not be permitted to rotate on any other

axis because this would change the underlying dynamics. In the current case, this is easily achievable, but if the measured part has an irregular shape, things are much more complicated.

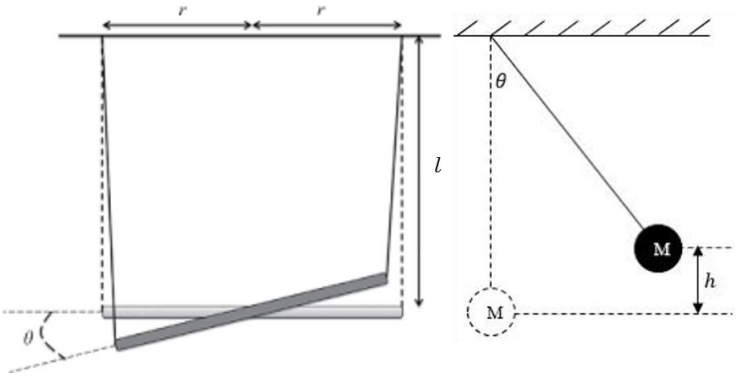
$$I_{CoM} = \frac{mgC_m T^2}{4\pi^2} - mC_m^2 \tag{6-14}$$

- **Bifilar Pendulum:** This method eliminates the main drawback of the simple pendulum since the object is not permitted to rotate on any other axis, by design. The dynamics are more complicated, but the resulting linearized system is simple. The inertial formula is given in (6-13) and as for its derivation, refer to Appendix J.

$$I_{CoM} = \frac{mgr^2 T^2}{4\pi^2 l} - mC_m^2 \tag{6-15}$$

The performance of the bifilar pendulum can be verified if the latter is conceived as a simple pendulum from its lateral view. From this perspective, the system constitutes a rotating mass pendulum without rigid-body inertia. With a method similar to the one of (6-14), the gravity constant g is estimated as a function of its free oscillation period T using (6-16). If the result is reasonable, the experimental setup is valid and can be used to approximate the inertia. The proposed system is illustrated in Figure 6-12.

$$g = \frac{4\pi^2 l}{T^2} \tag{6-16}$$



(a) **(b)**

Figure 6-12. (a) Bifilar pendulum overview. (b) Bifilar pendulum lateral view.

Beware that the inertias calculated by the equations (6-14) and (6-15) are functions of the measured period squared (T^2). So, a small error in the measured period, results in a large error in I_{CoM}.

6.4 Leg's Inertial Parameters Identification – Method 2

In this section a simulated whole-leg identification is considered, exploiting the linearity of the Equations of Motion (EoMs) of multibody systems, like Laelaps' legs w.r.t their parameters. The realization of the method in the actual robot is left for future work, since at present Laelaps II is not able to fulfill the experiment's hardware prerequisites.

The algorithm consists of an optimization step, in which a suitable joint-space trajectory is created. Then, the leg is commanded to follow this trajectory by a controller. During this

procedure kinematic and current measurements are sampled. At last, the identification problem is solved.

In this section, initially, the simulation environment is explained, along with its major components. Next, the identification problem is formulated, specifically for the leg's multibody system. Moreover, the trajectory optimization problem is described and solved. At last, the identification results are evaluated and the whole framework's future implementation is concerned.

6.4.1 Simulation Environment

The whole framework consists of several *.m Matlab* files. The main file, namely *LaelapsID.m* materializes the formerly described steps of the algorithm. In the simulation, the system's parameters are imported from the leg's CAD files and the whole idea of the simulated identification process is to approximate them. In the next section, the derivation of the leg's model is considered. To control this model and track the previously mentioned optimal trajectory, a controller is created.

Dynamical Model

The Laelaps leg, as in the majority of quadruped robots, constitutes a two-segment pendulum. The model is very similar to a double pendulum, with small differences concerning the layout of the joints and the motors. A schematic of the system is presented in Figure 6-13.

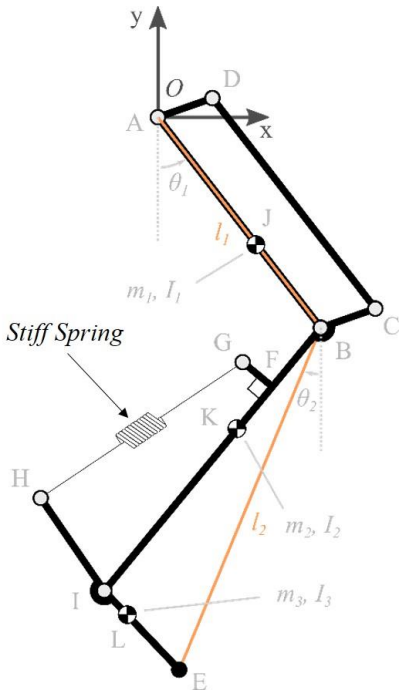


Figure 6-13. Laelaps II leg model.

Kinematics

The forward kinematics of the leg, based on the configuration of Figure 6-13, are given by the equations (6-17). Here, the difference with the model of Figure 3-2 is the reference frame's orientation.

$$\begin{aligned} x_E &= l_1 \sin(\theta_1) + l_2 \sin(\theta_2) \\ y_E &= -l_1 \cos(\theta_1) - l_2 \cos(\theta_2) \end{aligned} \tag{6-17}$$

The respective Jacobian ($\mathbf{J}_{\mathbf{v}}$) for this configuration is presented in equation (6-18).

$$\mathbf{J}_{\mathbf{v}} = \begin{pmatrix} l_1 \cos(\theta_1) & l_2 \cos(\theta_2) \\ l_1 \sin(\theta_1) & l_2 \sin(\theta_2) \end{pmatrix} \tag{6-18}$$

Dynamics

The equations of motion are presented below (6-19)-(6-23), for the derivation of which *Mathematica* was used [52]. Let θ_i designate the angle of each joint, m_i the mass of each link, l_i its length, I_i its inertia, and $C_{m,i}$ its CoM distance w.r.t. the corresponding joint. Also, $I_{M,i}$ is each motor's rotor inertia, and n_i each drivetrain's reduction ratio.

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{c}(\mathbf{q},\dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q}) + \boldsymbol{\tau}_{\mathbf{friction}}(\dot{\mathbf{q}}) = \boldsymbol{\tau}_{\mathbf{in}}, \quad \mathbf{q} = \begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} \tag{6-19}$$

$$\mathbf{M(q)} = \begin{pmatrix} I_1 + m_1 C_{m,1} + m_2 l_1^2 + I_{M,1} n_1^2 & m_2 C_{m,2} l_1 \cos(\theta_1 - \theta_2) \\ m_2 C_{m,1} l_1 \cos(\theta_1 - \theta_2) & I_2 + m_2 C_{m,2}^2 + I_{M,2} n_2^2 \end{pmatrix} \tag{6-20}$$

$$\mathbf{c}(\mathbf{q},\dot{\mathbf{q}}) = \begin{pmatrix} m_2 C_{m,2} l_1 \sin(\theta_1 - \theta_2)\dot{\theta}_2^2 \\ -m_2 C_{m,2} l_1 \sin(\theta_1 - \theta_2)\dot{\theta}_1^2 \end{pmatrix} \tag{6-21}$$

$$\mathbf{G}(\mathbf{q}) = \begin{pmatrix} g sin(\theta_1)(m_1 C_{m,1} + m_2 l_1) \\ m_2 C_{m,2} g sin(\theta_2) \end{pmatrix} \tag{6-22}$$

$$\boldsymbol{\tau}_{\mathbf{in}} = \begin{pmatrix} K_{T,1} n_1 i_{s,1} \\ K_{T,2} n_2 i_{s,2} \end{pmatrix} \tag{6-23}$$

For the friction model, the previously derived nonlinear model would be normally used (Section 6.2). But the actual friction model of the whole joint remains unknown, since the solutions in Section 6.2 concern only the motor – gearbox system and not the belts. So, to simplify things in the current simulation, the common viscous friction model of the equation (6-24) is used, to avoid unnecessary complexity. Note that in the aforementioned equation, each joint's friction coefficient is denoted with b_i.

$$\boldsymbol{\tau}_{\mathbf{friction}}(\dot{\mathbf{q}}) = \begin{pmatrix} b_1 \dot{\theta}_1 \\ b_2 \dot{\theta}_2 \end{pmatrix} \tag{6-24}$$

These EoMs are materialized in *Leg_Dynamic.m* and *Mass_Dynamic.m* source files. Using this files, the *Matlab* ODE solvers are able to solve the EoMs and obtain the system's response to any input. In the current configuration, the system's input is the control effort, generated by the controller that is considered next. To integrate (6-19) with *Matlab,* the equations must be transformed in the form of (6-25). Note that $\mathbf{q}_{\mathbf{aug}}$ is the augmented state vector that contains both the leg's angles and their first derivatives.

$$\mathbf{M}_{aug}\left(\mathbf{q}_{aug}\right)\cdot\dot{\mathbf{q}}_{aug}=\mathbf{F}\left(\mathbf{q}_{aug}\right),\ \mathbf{q_{aug}}=\begin{pmatrix}\theta_1 & \theta_2 & \dot{\theta}_1 & \dot{\theta}_2\end{pmatrix}^{\mathrm{T}}$$

$$\mathbf{M}_{aug}\left(\mathbf{q}_{aug}\right)=\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & I_1+m_1C_{m,1}+m_2l_1^2+I_{M,1}n_1^2 & m_2C_{m,2}l_1\cos\left(\theta_1-\theta_2\right)\\ 0 & 0 & m_2C_{m,1}l_1\cos\left(\theta_1-\theta_2\right) & I_2+m_2C_{m,2}^2+I_{M,2}n_2^2\end{pmatrix} \tag{6-25}$$

$$\mathbf{F}\left(\mathbf{q}_{aug}\right)=\begin{pmatrix}\dot{\theta}_1\\ \dot{\theta}_2\\ \boldsymbol{\tau}_{\mathbf{in}}-\mathbf{c}\left(\mathbf{q}_{aug}\right)-\mathbf{G}\left(\mathbf{q}_{aug}\right)-\boldsymbol{\tau}_{\mathbf{friction}}\left(\mathbf{q}_{aug}\right)\end{pmatrix}$$

Control

The equations (6-25) have been coded and tested with *Matlab*. In order for the identification algorithm to work, the simulated leg must follow a trajectory (see Section 6.4.4), specifically designed for giving a well-conditioned solution to the subsequent identification problem (see Section 6.4.2). To do this, a controller has to be designed and implemented. In general, trajectory tracking controllers require knowledge of the plant, especially in the model-based case that the system's parameters are involved directly in the control process. However, here, the actual plant's parameters remain unknown. The only knowledge comes from the CAD files of the leg. Given that the actual parameters deviate from the CAD ones, a robust design should be adopted. Also, the controller should be easily implemented in the current distributed control architecture of the robot.

To this end, the system was linearized at the lower stable fixed-point (refer to [99]). Arguably, due to the large region of attraction of this fixed point, the linearization is valid for a wide range of angles around it. So, any stable controller designed for the linearized plant, should perform well in controlling the actual nonlinear one near the fixed point's vicinity. The controller was created with the pole placement technique, using *Matlab*'s *place()* function (refer to *LaelapsID.m)*. Since the leg's model is a fourth-order system, to come with a manageable second-order one, the dynamics of which are well known, two of the total four closed loop poles must dominate the whole system's response. This is achieved by placing the remaining two far to the left of the pole-zero map. This way, the fast poles' contributions can be ignored and the whole system can be approximated by a second-order one with the two dominant poles. Moreover, the design should result in a closed-loop system that has adequate bandwidth to track the identification's trajectory (see Section 6.4.4) and at the same time minimal oscillations during the transient phase. So, the dominant poles are placed on the real axis, adequately far from zero. Since the critically damped systems have the fastest response possible without overshooting, the two poles are placed at the same position. The designed controller is expected to present increased stability and track the identification's trajectory for a wide range of angles. The resultant gains are presented in (6-26), with $\mathbf{K}$ being the gain matrix, $\mathbf{i_s}$ a vector containing the current inputs and $\mathbf{q_{des}}$ the desired trajectory that will be discussed in Section 6.4.4.

$$\mathbf{i_s}=\begin{pmatrix}i_{s,1}\\ i_{s,2}\end{pmatrix}=\mathbf{K}\left(\mathbf{q_{des}}-\mathbf{q_{aug}}\right),\ \textit{with}\ \mathbf{K}=\begin{pmatrix}\mathbf{K_1}\\ \mathbf{K_2}\end{pmatrix}=\begin{pmatrix}21.3569 & 1.6667 & 4.5734 & 0.3571\\ 1.9331 & 12.2753 & 0.4108 & 2.6365\end{pmatrix} \tag{6-26}$$

The designed controller results in the pole-zero map of Figure 6-14 for the closed-loop system.

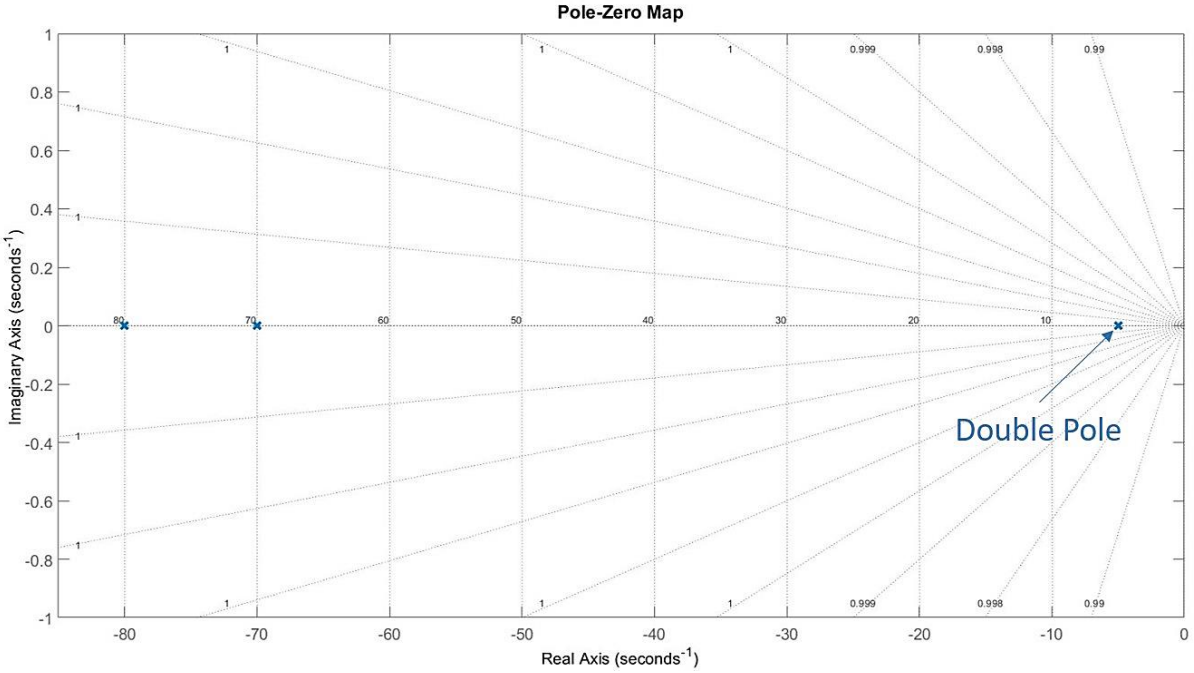


Figure 6-14. Linearized leg's model pole-zero map.

6.4.2 Identification Problem Formulation

The derived model (6-19) can be expressed with linear relationships w.r.t. its parameters. In this configuration, by doing a regression with least-square algorithms, these parameters can be found with great accuracy. As shown in the literature [24] [106], the equations of motion of each link of an arbitrarily n-DoF manipulator can be expressed as a function of its inertial parameters and friction coefficients (6-27).

$$\boldsymbol{\pi}_i = \begin{pmatrix} m_i & m_i C_{mx,i} & m_i C_{my,i} & m_i C_{mz,i} & I_{xx,i} & I_{xy,i} & I_{yy,i} & I_{yz,i} & I_{zz,i} & I_{M,i} & b_i \end{pmatrix}^T \tag{6-27}$$

After eliminating the zero elements in $\boldsymbol{\pi}$ and linear dependencies in regressors matrix $\mathbf{Y}$ to prevent ill-conditioning and rank deficiencies, the former is given by,

$$\boldsymbol{\pi} = \begin{pmatrix} \boldsymbol{\pi}_1 \\ \boldsymbol{\pi}_2 \end{pmatrix} \in \mathbb{R}^6 \Rightarrow$$
$$\boldsymbol{\pi} = \begin{pmatrix} m_1 C_{m,1} + m_2 l_1 & I_1 + m_1 C_{m,1}^2 + I_{M,1} n_1^2 + m_2 l_1^2 & b_1 & m_2 C_{m,2} & I_2 + m_2 C_{m,2}^2 + I_{M,2} n_2^2 & b_2 \end{pmatrix}^T \tag{6-28}$$

The mathematical formulation of the regression problem is illustrated in (6-29), with j being the sample number.

$$\mathbf{Y} \cdot \boldsymbol{\pi} = \boldsymbol{\tau}_{\mathbf{in}}, \text{ with}$$
$$\mathbf{Y}_j = \begin{pmatrix} gsin(\theta_1) & \ddot{\theta}_1 & \dot{\theta}_1 & l_1 \sin(\theta_1 - \theta_2)\dot{\theta}_2^2 + l_1 \cos(\theta_1 - \theta_2)\ddot{\theta}_2 & 0 & 0 \\ 0 & 0 & 0 & gsin(\theta_2) - l_1 \sin(\theta_1 - \theta_2)\dot{\theta}_1^2 + l_1 \cos(\theta_1 - \theta_2)\ddot{\theta}_1 & \ddot{\theta}_2 & \dot{\theta}_2 \end{pmatrix}_j \tag{6-29}$$

By using Least-Squares, the solution of (6-29) is optimal if and only if (6-30) is minimized.

$$J(\boldsymbol{\pi}) = \sum_{j=1}^{N} \varepsilon^2(t) = \sum_{j=1}^{N} \left(\tau_{in,j} - \mathbf{Y}_j \cdot \boldsymbol{\pi}\right)^2 \tag{6-30}$$

A detailed derivation, along with more advanced algorithms can be found in [26]. The result is given in (6-31), below.

$$\boldsymbol{\pi}^* = \operatorname{argmin}_{\boldsymbol{\pi}\in\mathbb{R}^6} \left[J(\boldsymbol{\pi})\right] = (\mathbf{Y}^T \cdot \mathbf{Y})^{-1} \cdot \mathbf{Y}^T \cdot \boldsymbol{\tau}_{\mathbf{in}} \tag{6-31}$$

The regression matrix $\mathbf{Y}$ is formed in *Obs_DIM.m*, while the least-squares problem is solved using the standard *Matlab* solver *lsqlin()*.

6.4.3 Identification Framework Overview

To find a well-defined and reliable least-squares solution, all of the leg-dynamics have to be excited and the regression matrix $\mathbf{Y}$ has to be robust and insensitive to small changes of the inputs. So, the proposed algorithm starts by finding a good trajectory that meets these requirements. Then, the identification problem is solved using a linear least-squares solver, with samples acquired from the aforementioned trajectory. An overview of this procedure is presented intuitively in Figure 6-15.

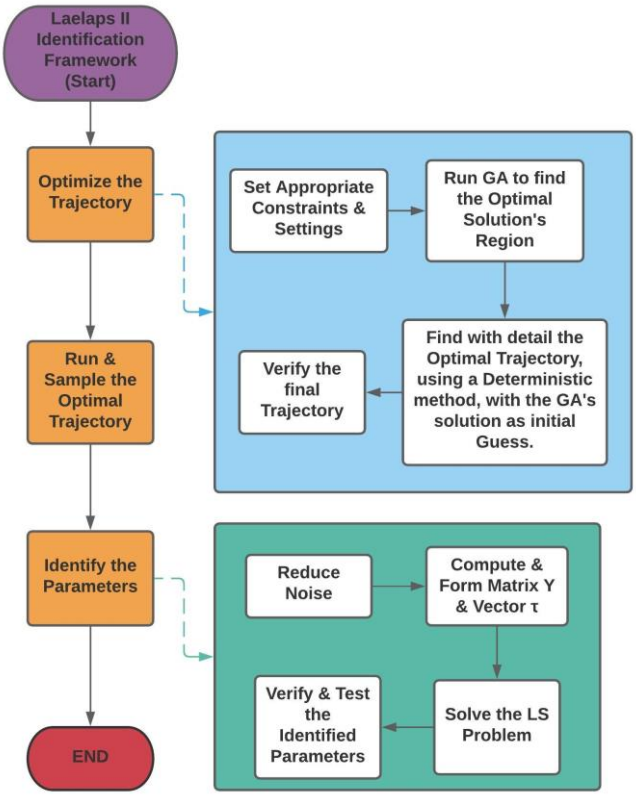


Figure 6-15. Identification framework workflow.

6.4.4 Trajectory Optimization

From the literature [107], trajectories for identification experiments are formed as truncated Fourier series. In the current case, a 3-term Fourier series is chosen. In (6-32), ω_f is the angular velocity of the trajectory, a_{ij} and b_{ij} are each harmonic's amplitudes, and $\theta_{j,off}$ represents the amplitude of the zero-harmonic.

$$\theta_j(t) = \theta_{j,off} + \sum_{i=1}^{3}\left[\frac{a_{ij}}{\omega_f i}\sin\left(\omega_f i \cdot t\right) - \frac{b_{ij}}{\omega_f i}\cos\left(\omega_f i \cdot t\right)\right], \text{ with } j = \{1,2\} \quad (6\text{-}32)$$

The optimization's design variables are the coefficients $\theta_{j,off}$, a_{ij} and b_{ij} , while $\omega_f = 0.5\pi$. This choice is justified, since the higher the frequency, the more dynamical the resulting trajectory is going to be. This means that the trajectory's accelerations will increase, without the need of additional terms that would increase the planner's computational cost. This way, a balance is kept among the terms of the regression problem (6-29), necessary to come with an accurate estimation of all parameters in (6-28) and especially the inertias. Since the inertias depend on the sampled accelerations, if the Signal to Noise Ratio (SNR) of the accelerations is low, the dependent parameters (inertias) will not be estimated accurately.

Furthermore, the constraints (6-33)-(6-42) should be considered for the resulting trajectory to be feasible in the real system. The constraint of (6-33) limit's the trajectory's angle range to prevent collisions with the robot's body. The problem has already been described in Chapter 3.

$$\left|\theta_{1,max}\right| \le 27^o \ \& \ \left|\theta_{2,max}\right| \le 80^o \quad (6\text{-}33)$$

In (6-34) the angular velocities of the trajectory are limited below the gearboxes' maximum velocity limits [47].

$$\left|\dot{\theta}_{1,max}\right| \le 450^o / s \ \& \ \left|\dot{\theta}_{2,max}\right| \le 360^o / s \quad (6\text{-}34)$$

The constraint of (6-35) is created to ensure that both joints will be moved for an adequately wide range of angles.

$$\theta_{1,max} - \theta_{1,min} \ge 15^o \ \& \ \theta_{2,max} - \theta_{2,min} \ge 20^o \quad (6\text{-}35)$$

The trajectory should not have very high demands in terms of power. With the constraint of (6-36), the input currents are kept below the drives' continuous operation limit [65].

$$\left|i_{s,1}\right| \le 6\,A \ \& \ \left|i_{s,2}\right| \le 6\,A \quad (6\text{-}36)$$

The (6-37)-(6-38), along with the (6-33) constraint are limiting the resultant end-effector's position inside the leg's valid workspace (Figure 3-7).

$$\sqrt{x_{E,min}^2 + y_{E,min}^2} \ge \sqrt{l_1^2 + l_2^2} \ \left(\text{Workspace Upper Bound}\right) \quad (6\text{-}37)$$

$$\sqrt{x_{E,max}^2 + y_{E,max}^2} \le l_1 + l_2 \ \left(\text{Workspace Lower Bound}\right) \quad (6\text{-}38)$$

The constraint (6-39) ensures that the knee's relative angle restriction with the leg's upper limb is not violated.

$$\min\left(\theta_2 - \theta_1\right) \ge 0^o \ \left(\text{Knee Relative Angle Restriction}\right) \quad (6\text{-}39)$$

Lastly, the constraints (6-40)-(6-42) dictate the initial and final angle positions.

$$\theta_1(t_0) = \theta_1(t_f) = 0^o \ \ \& \ \theta_2(t_0) = \theta_2(t_f) = 0^o \quad (6\text{-}40)$$

$$\dot{\theta}_1(t_0) = \dot{\theta}_1(t_f) = 0^o / s \ \ \& \ \dot{\theta}_2(t_0) = \dot{\theta}_2(t_f) = 0^o / s \quad (6\text{-}41)$$

$$\ddot{\theta}_1(t_0) = \ddot{\theta}_1(t_f) = 0^o / s^2 \;\; \& \;\; \ddot{\theta}_1(t_0) = \ddot{\theta}_2(t_f) = 0^o / s^2 \tag{6-42}$$

Most of the aforementioned constraints are nonlinear w.r.t. the optimization's design variables and they are materialized in *nonlcon.m*. The(6-40)-(6-42) are linear and as such, they are realized in matrix form, inside the *LaelapsID.m*.

6.4.5 Stochastic Optimization

The Genetic Algorithm (GA) optimization is a metaheuristic procedure that belongs to a larger group, the so-called Evolutionary Algorithms (EA). This method was employed to solve the current problem, since it does not require an initial guess by the user. Given in enough run-time, it is ensured that a close to the optimal result will be found. So, the objective function (6-43) is chosen to minimize the sensitivity of the regressor matrix. This cost function is located in *Obj_Func.m* file.

$$\text{minF} = Cond(\mathbf{Y}) \tag{6-43}$$

The rationale behind that choice is the following. Consider the matrix $\mathbf{Y}$ and its condition number $Cond(\mathbf{Y})$, which is defined as the ratio between its largest and smallest singular values. When solving a linear system $\mathbf{Y} \cdot \boldsymbol{\pi} = \boldsymbol{\tau}$, the larger the condition number, the more sensitive is the solution $\boldsymbol{\pi} = \mathbf{Y}^{\dagger}\boldsymbol{\tau}$ to small perturbations (e.g. noise) of the vector $\boldsymbol{\tau}$. Hence, if $Cond(\mathbf{Y})$ is large, tiny distortions in $\boldsymbol{\tau}$ can be hugely amplified when solving for $\boldsymbol{\pi}$. Also, the quantity $\log_2(Cond(\mathbf{Y}))$ is an estimate of how many bits are lost in solving the linear system $\mathbf{Y} \cdot \boldsymbol{\pi} = \boldsymbol{\tau}$. For example, if $Cond(\mathbf{Y}) = 2^{10}$ and $\boldsymbol{\tau}$ is encoded in 64 bits, solving for $\boldsymbol{\pi}$, will result in losing 10 bits of precision out of the total 64 [108].

The progress of the GA algorithm is shown in Figure 6-16. For the procedure, the standard *ga() Matlab* function was used. The solution's condition number is very low and thus a robust Least Squares solution is expected. Note that since it is a stochastic optimization method, executing the process again may produce different results. Moreover, the function is called inside the *LaelapsID.m* source file.

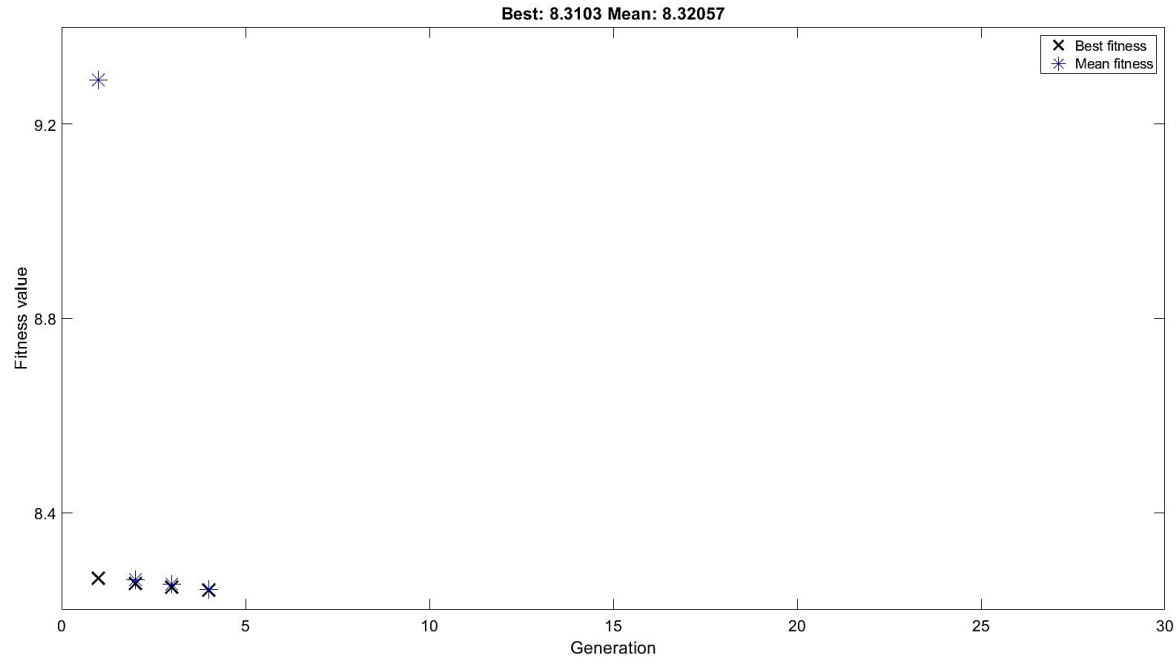


Figure 6-16. GA optimization progress.

6.4.6 Deterministic Optimization

For better accuracy, a nonlinear solver for deterministic optimization was fed with the optimal guess of GA. After a few iterations, a more detailed solution was achieved. The process is executed by *Matlab's fmincon()* function using the SQP (Sequent Quadratic Programming) solver. That optimization algorithm tries to minimize a quadratic approximation of the cost function. Thus at each step, the solution is projected back to the original cost function value, until its value change exceeds a given numerical tolerance. For a better insight into the subject check [109]. Moreover, the function is called inside the *LaelapsID.m* source file.

6.4.7 Optimal Trajectory Results

The optimal coefficients produced with the above-described process are given by (6-44).

$$\mathbf{a} = \begin{pmatrix} -0.1865 & 0.5676 & -0.3811 \\ -0.1600 & 0.5286 & -0.3686 \end{pmatrix} [rad]$$
$$\mathbf{b} = \begin{pmatrix} -0.1281 & 0.4609 & -0.2646 \\ -0.0373 & 0.5107 & -0.3280 \end{pmatrix} [rad] \quad (6\text{-}44)$$
$$\boldsymbol{\theta}_{\mathbf{off}} = \begin{pmatrix} 0.0090 \\ 0.0692 \end{pmatrix} [rad]$$

And the optimal condition number of $\mathbf{Y}$ is given by (6-45).

$$Cond(\mathbf{Y}) = 4.7742 \quad (6\text{-}45)$$

In the following figures, the leg's angles (Figure 6-17), angular velocities (Figure 6-18) and angular accelerations (Figure 6-19) can be viewed. Finally, the motors' current efforts are plotted in Figure 6-20. The resulting trajectory is generated by a planner in *Traj_Gen.m*. The total time of the simulation was chosen to be 20 [s], to gather enough samples.

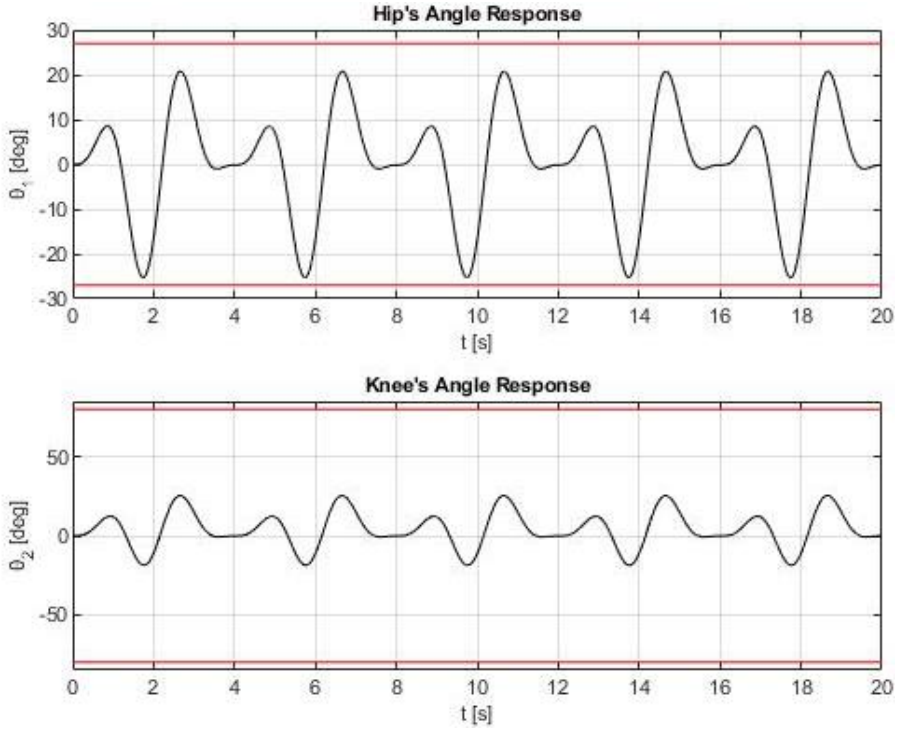


Figure 6-17. Leg's angle responses.

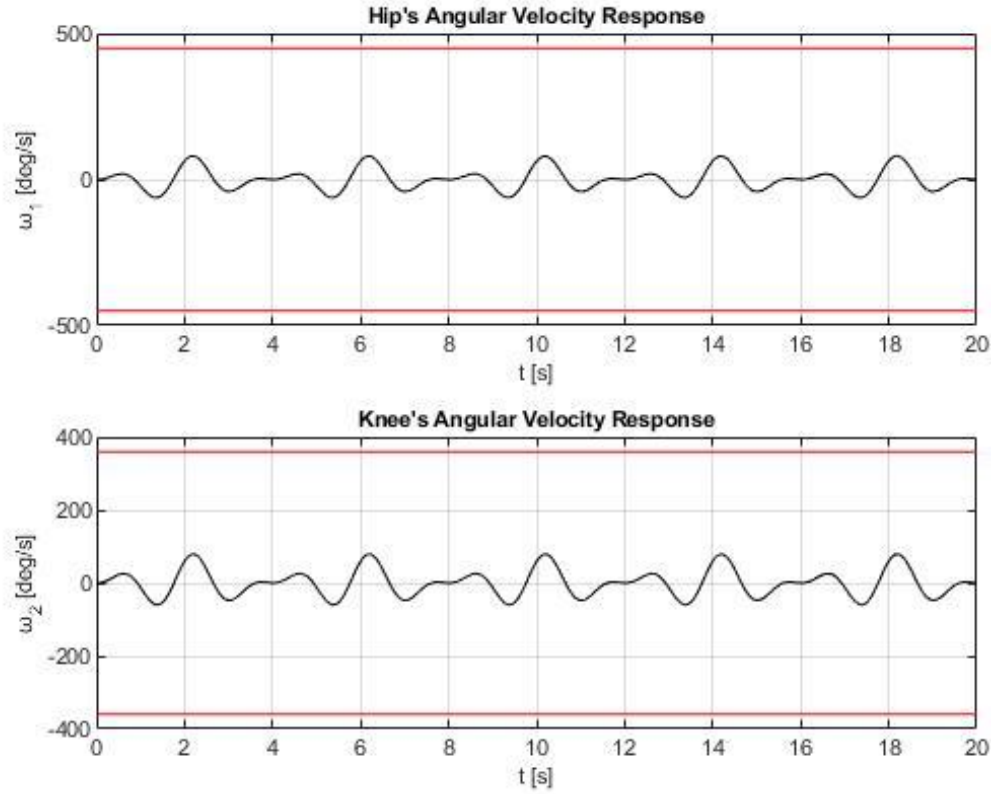


Figure 6-18. Leg's angular velocity responses.

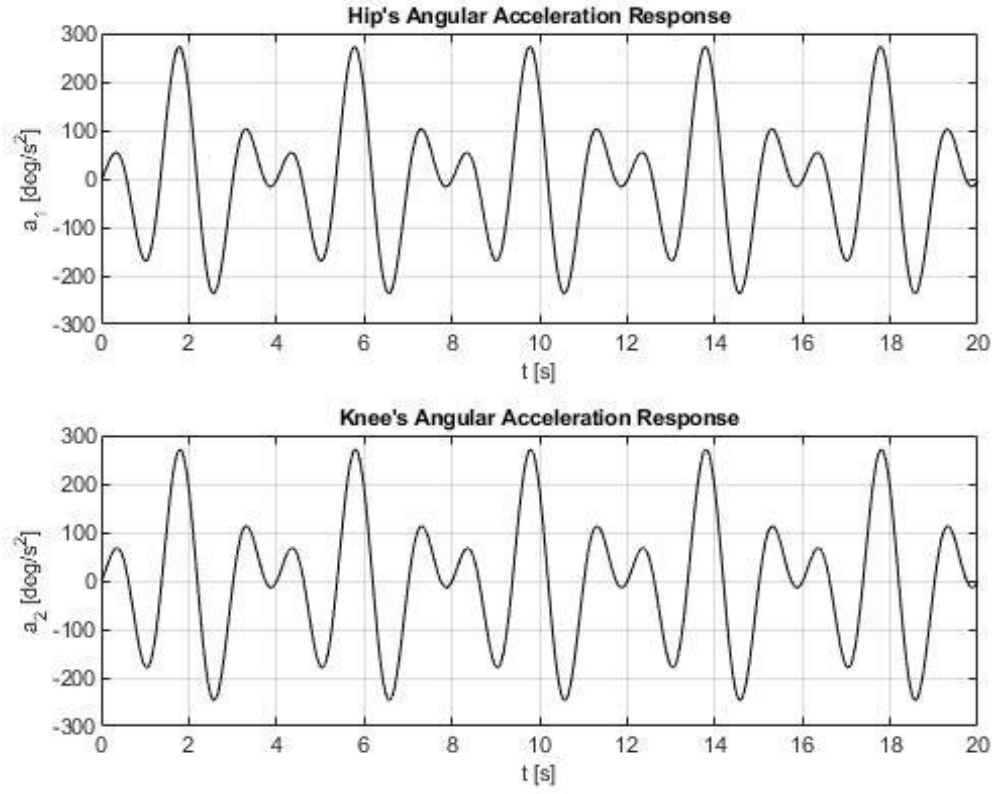


Figure 6-19. Leg's angular acceleration responses.

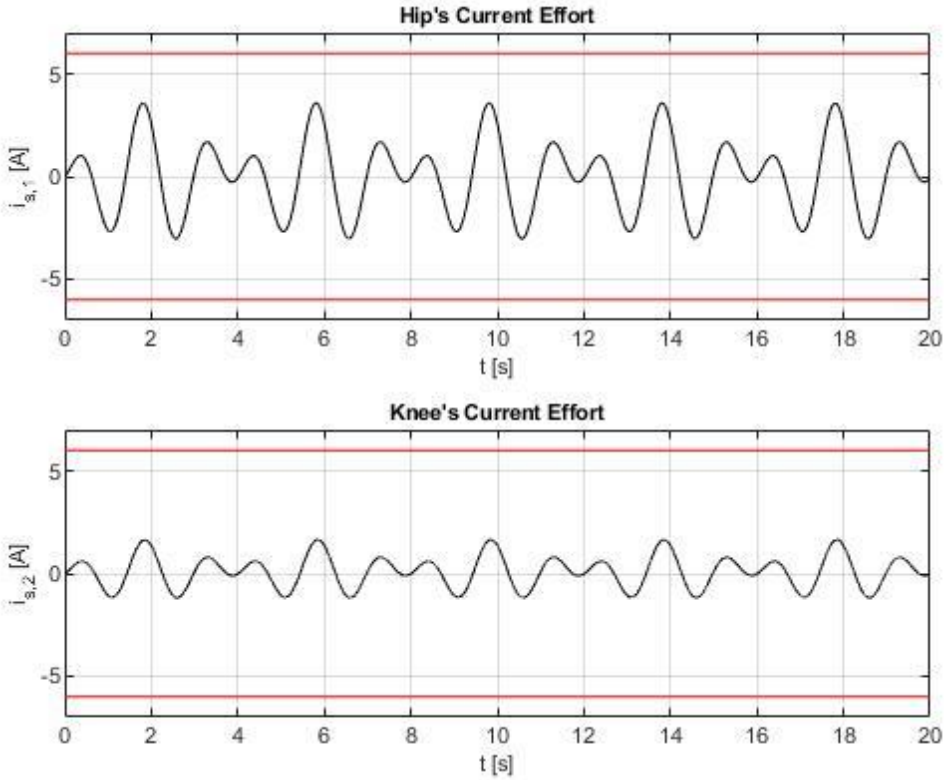


Figure 6-20. Leg's current inputs.

In the ideal case of full state feedback and absence of noise, parameter identification produces exceptional results. The noise and various perturbations in the real experiments downgrade the algorithm's performance significantly. To evaluate the algorithm's performance white-noise was added to current inputs and kinematic measurements. The idea here is that the input trajectory excites certain harmonics. Since the trajectory is a Fourier series, the excited harmonics are known. This way, low-pass filters can be designed to have bandwidth that matches the trajectory's highest harmonic. Subsequently, these filters can be applied to the noisy kinematic measurements and decrease the algorithm's overall error.

To have decisive proof of the effectiveness of the cost function used in the optimization (6-43), the resultant trajectory of the GA is compared with the final one after the deterministic optimization's step. Figure 6-21 indicates that the condition number has a significant impact on the method's accuracy. Note that the indexes of the parameter vector in each bar refer to (6-28) (e.g. $\boldsymbol{\pi}[3] = b_1$). For sure, further modifications should be made to capture the real experiment's conditions and equipment characteristics. This way a more realistic performance evaluation would be possible.

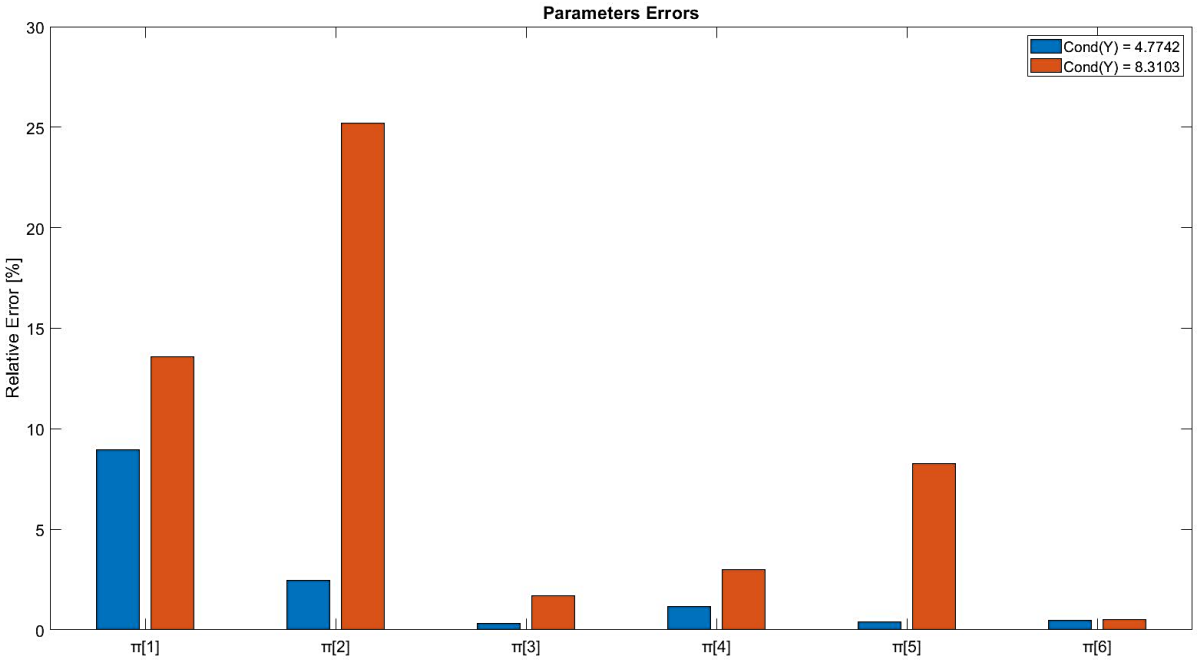


Figure 6-21. Performance evaluation of the ID framework.

6.4.8 Experimental Procedure for Whole Leg Identification

Laelaps II cannot yet accommodate the aforesaid experiments due to the lack of hardware provisions and necessary electronics. So, the experimental validation of the above method is left for future work. Nevertheless, for completeness, the necessary tasks and procedures are described in this section.

For the experiments, a current sensor is necessary, integrated on the high side of each motor. Preferably the TMCS1100, which according to Table 6-8 presents better specifications, compared to INA253 (Table 6-7). A problem arises, due to the hip's brushless motor and the complexity of how exactly the applied current is translated to torque. Also, the above system model accounts for brushed DC motors in both joints. To overcome the problem, the hip's motor could be replaced with another knee's brushed DC motor. This way, the proposed model becomes effective and the results would be valid. Any deviations will be eliminated by suitable tuning of the controllers' gains.

Another aspect of the experiments is the firmware that could be used. The current Laelaps II motion control firmware supports the easy integration of arbitrary planners and EtherCAT stacks. After reading this thesis and other works like [20] [34], one should be able to design a new EtherCAT application and integrate it into the original leg firmware. The planner should execute the optimal trajectory that was derived in Section 6.4.7. The method is robust, as Figure 6-21 suggests, so good results might be achieved even if the system differs a little from the modeled one. The results of the proposed experiments (Section 6.3.3) should be close to the ones acquired by the whole leg identification.

7 Laelaps Motor Overheating Protection System

Every physical system has limitations regarding its operation. Given all the necessary power, an arbitrary controller can achieve any desired response. However, even by ignoring the limited availability of energy, stressing any system beyond its capabilities will result in failure and damage. The Laelaps II motors have such boundaries that must be accounted for any design that they are included in. Some of these are the current limits of their drives and the thermal strength of their windings. To overcome these obstacles, an insightful hardware design should be made in advance, along with a firmware that does not stress the system beyond its maximum potential.

The objective of this chapter is to address the limits of Laelaps II motors and find simple software solutions to account for them. This way, optimal performance may be achieved and at the same time, their life is prolonged. The manufacturer does not provide solutions for online adaptive current regulation, hence developing a custom one deemed necessary. Due to the complexity and additional cost of direct temperature measurements, a solution without temperature sensors is developed. The only hardware requirement here is knowing the current applied to the motors by using current sensors.

Initially, some theoretical concepts are explained along with the development of a typical thermal model of a DC Brushless Maxon motor. Furthermore, the Motor Overheating Protection System (MOPS) is presented, along with its requirements. Last but not least, simulation results are discussed along with an evaluation of whether the algorithm's final goal is met.

7.1 Theoretical Preliminaries

7.1.1 Thermal Modeling and Problem Formulation

Maxon, the manufacturer of Laelaps II motors, has specific guidelines and standards for their efficient operation and health. There are extensive guidelines available [21] that could facilitate any design. The proposed motor's thermal model is displayed in Figure 7-1.

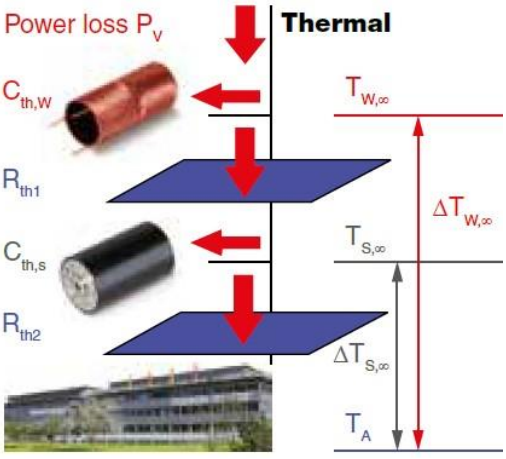


Figure 7-1. Maxon's proposed motor thermal model.

The equations that capture the behavior of the above-described system are presented in (7-1). The parameters with index 1 refer to the motor's winding, while the parameters with index 2 refer to the housing. The relative to ambient temperature of each of the

aforementioned modules is denoted with T_i, the absolute temperature with $T_{abs,i}$, the thermal resistance with R_i, the thermal capacitance with C_i and lastly the power loss of the motor with q_{in} (thermal input).

$$\frac{d}{dt}\begin{pmatrix} T_1 \\ T_2 \end{pmatrix} = \begin{bmatrix} -\dfrac{1}{R_1 C_1} & \dfrac{1}{R_1 C_1} \\ \dfrac{1}{R_1 C_2} & -\dfrac{R_1 + R_2}{R_1 R_2 C_2} \end{bmatrix} \begin{pmatrix} T_1 \\ T_2 \end{pmatrix} + \begin{pmatrix} \dfrac{1}{C_1} \\ 0 \end{pmatrix} \cdot q_{in}, \text{ with } T_i = T_{abs,i} - T_{ambient} \ \& \ i = \{1, 2\} \tag{7-1}$$

At first glance, the above ODEs are linear, but this is not entirely accurate since the winding's electrical resistance depends on its temperature. Consequently, the thermal input is increased in a nonlinear, temperature-dependent fashion (7-2). Let R_{mot} be the winding's electrical resistance, the a_{Cu} be the thermal coefficient of copper and $T_{abs,1}$ the winding's absolute temperature.

$$R = R_{mot}\left[1 + a_{Cu}\left(T_{abs,1} - T_{ambient}\right)\right], \text{ with } q_{in} = I^2 R\left(T_1\right) \text{ and } T_{ambient} = 25^o C \tag{7-2}$$

In the motors' datasheets [46] [93], the thermal and electrical resistances are given alongside the system's thermal time constants. To proceed with this analysis and determine the thermal capacitances, the minor nonlinearity of the resistance (7-2) is neglected. This results in an insignificant error in capturing the transient system's response. As it will be proven, the steady-state response is not a function of capacitance and thus this simplification does not interfere with MOPS performance. The thermal time constants τ_i can be converted to the system's poles p_i using the equation (7-3).

$$\tau_i = -\frac{1}{p_i}, \text{ with } i = \{1, 2\} \tag{7-3}$$

By performing the required calculations, the following system of nonlinear algebraic equations (7-4) is formed.

$$\begin{aligned} \frac{C_1 R_1 + \left(C_1 + C_2\right) R_2 - \sqrt{\left(C_2 R_2 + C_1\left(R_1 + R_2\right)\right)^2 - 4 C_1 C_2 R_1 R_2}}{2} &= \tau_1 \\ \frac{C_1 R_1 + \left(C_1 + C_2\right) R_2 + \sqrt{\left(C_2 R_2 + C_1\left(R_1 + R_2\right)\right)^2 - 4 C_1 C_2 R_1 R_2}}{2} &= \tau_2 \end{aligned} \tag{7-4}$$

Using a mathematical software package, like *Mathematica*, the system is easily solved. The obtained values of thermal capacitances (C_1 and C_2) follow in Table 7-1. Note that the thermal resistances (R_1 and R_2) and the thermal time constants (τ_1 and τ_2), listed in the aforementioned table, are provided by the manufacturer's datasheets [46] [48].

Table 7-1. Motors' thermal attributes.

Motor	Knee	Hip
C_1 [J / K] (Winding)	77.7359	29.1191
C_2 [J / K] (Housing)	277.1107	893.8039
R_1 [K / W]	1.2	1.1
R_2 [K / W]	3.8	1.7

τ_1 [s]	71.7	31
τ_2 [s]	1370	1570

The values of the steady-state system's temperatures are given with the equations (7-5) for intermittent operation. Hence, the use of I_{rms}.

$$T_1 = \frac{\left(R_1 + R_2\right) R I_{rms}^2}{1 - a_{Cu}\left(R_1 + R_2\right) R I_{rms}^2}, \text{ with } a_{Cu} = 0.0039\left[K^{-1}\right]$$
$$T_2 = \frac{R_2}{R_1 + R_2} T_1 \tag{7-5}$$

Now that the thermal model is defined, the algorithm may be designed. From [46] [93], the following useful features are extracted in Table 7-2.

Table 7-2. Useful motor characteristic values.

Parameter		Knee	Hip
R_{mot} [Ω]	Electrical resistance	0.608	0.617
$I_{n,max}$ [A]	Motors' max. continuous current	4.58	6.21
I_{max} [A]	Drives' max. current	12	12
T_{max} [°C] (Winding)	Windings' max. temperature	125	125

7.1.2 Cumulative RMS Formula

The necessary online updates of the Root-Mean-Square (RMS) value are calculated (7-6) in a periodic cumulative manner. This formula serves a crucial purpose in the MOPS algorithm since the resultant saturation limits depend solely on it.

$$I_{rms[n]} = \sqrt{\frac{(n-1) I_{rms[n-1]}^2 + I_{[n]}^2}{n}}, \text{ with } n \in \mathbb{N} \tag{7-6}$$

7.2 MOPS Algorithm

Any thermal protection algorithm must take into account every possible operating condition to ensure the nominal operation and safety of the system. The basic form of MOPS is summarized in calculating the Cumulative RMS (CRMS) current value and depending on it, choosing a current saturation limit. Intuitively, the dependence among saturation limits and CRMS current values is illustrated in Figure 7-2.

Due to the diversity of the conditions that Laelaps II operates in, the basic architecture of the aforementioned algorithm is enriched with features that guarantee its robustness. For example, the algorithm was failing to regulate the current after extensive non-periodic coolings. The cumulative nature of the CRMS formula resulted in reducing its sensitivity with time. Subsequently, the winding's temperature rised faster than the CRMS value, if large currents were applied to the motor immediately after the cooling period. As one can imagine, in the real system the windings would overheat and eventually melt. To address this issue a CRMS reset mechanism was implemented to avoid fatal temperature overshooting. If the motor has been cooled down for over a winding's thermal time constant, the CRMS counter n of (7-6) is resetted.

As shown in Figure 7-2, three different levels of saturation are distinguished for each motor type. The first level accounts for low CRMS current values, below the motor's continuous operation limit. In such a case, the control effort is saturated at the drive's maximum potential of 12 A. For medium CRMS values, the saturation limit is chosen to be the one that gives the prescribed relaxed temperature limit in steady-state. To calculate its value, the formulas are solved for the steady-state current. If the CRMS value exceeds a certain limit, Laelaps II has to do the absolute minimum, which is standing still and wait for the actuators to cool-down. This is why, above 9 A of CRMS current the saturation limit becomes 2 A, which is more than enough for Laelaps II to stay static in some body postures close to the legs' singular configurations. The governing logic is better interpreted by observing Figure 7-3. Note that the displayed flow chart ignores some steps of the algorithm in order not to overwhelm the reader, since it is a qualitative representation.

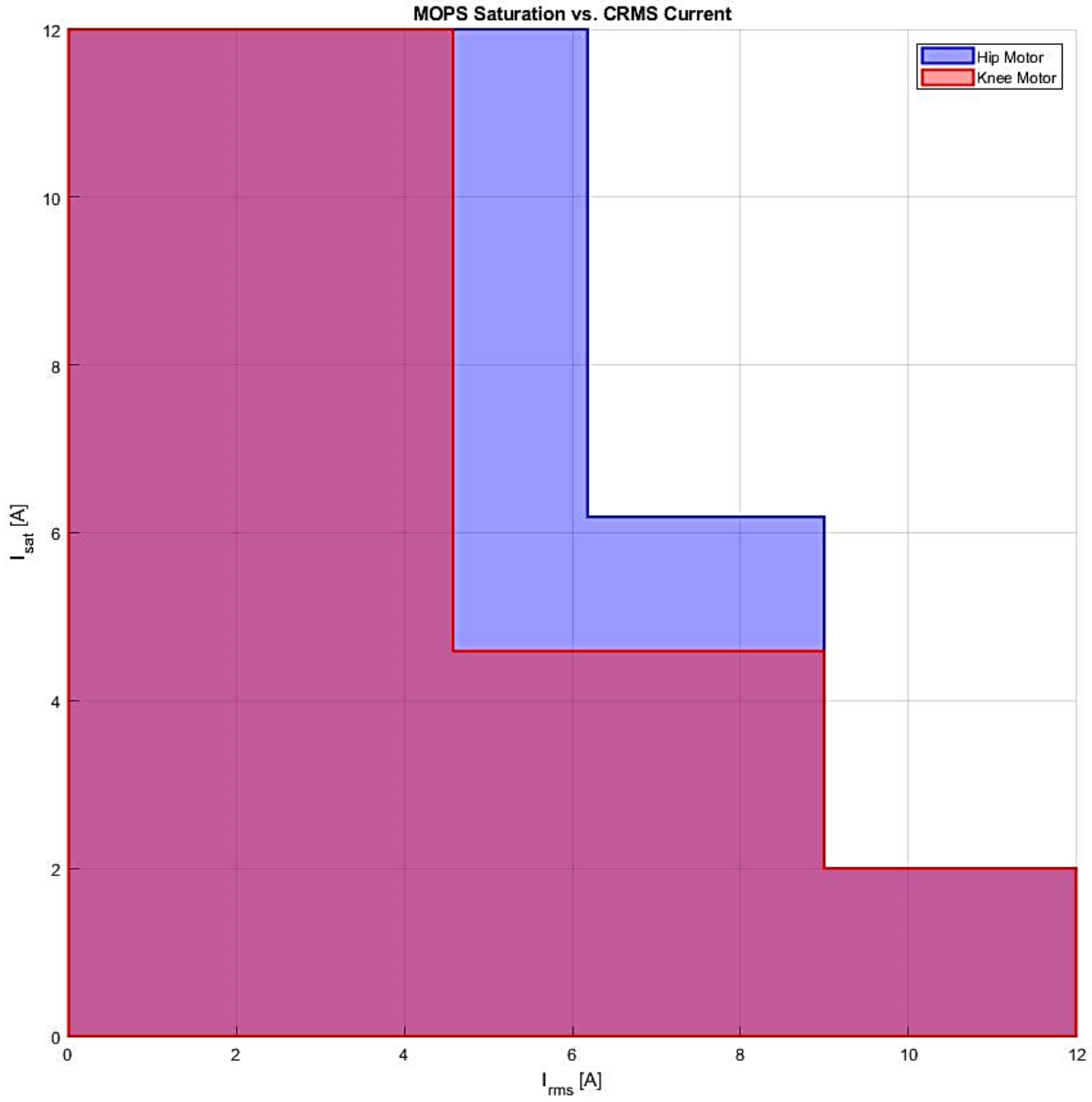


Figure 7-2. MOPS saturation limits as a function of CRMS.

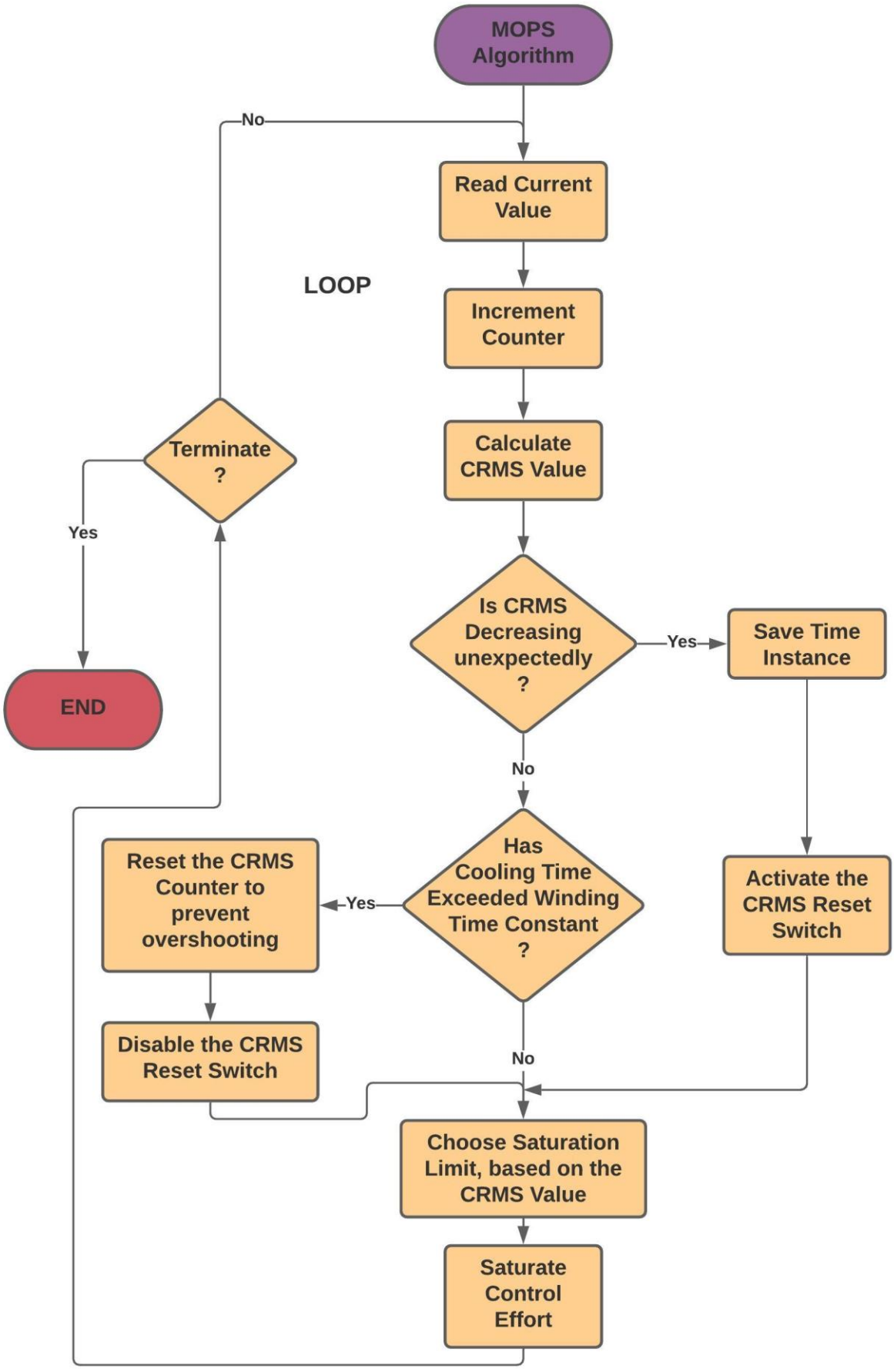


Figure 7-3. MOPS algorithm overview.

The full algorithm is presented in [52] and more specifically in the *Input_Current.m*, as well as in the following code snippet. Note that the CRMS values are buffered for debug purposes. In the actual implementation only the previous value is necessary (7-6). Moreover, the variable *offset* with which the CRMS counter is resetted, must be tuned according to the loop frequency of the algorithm to prevent overshooting. It must be ensured that the winding's temperature is not going to overshoot its limit under any circumstances. This should be considered in the actual implementation of the algorithm. Note that the whole *Matlab* simulation is controlled by *MOPS_Framework.m* source file. The notations used in the code are documented in the aforementioned source file.

```
% MOPS ALGORITHM %

% Calculate & Buffer Current's RMS Value
n = n+1;
u_rms = [u_rms ;CRMS(Current_Log(end),n)];

% Unexpected Cooling-Down identified
if(u_rms(end)- u_rms(end-1) <=0 && reset_switch == 0)

    % Turn on Reset Switch & Save the time & current RMS values
    reset_switch = 1;
    t0 = t;
    u0 = u_rms(end);
end

% Unexpected Cooling-Down stopped
if(reset_switch == 1 && u_rms(end) - u_rms(end-1) > 0)

    % Reset CRMS Counter (Anti-Windup Feature) and turn-off the Reset
    % Switch
    if(t-t0 >=tc1)
        n = offset;
    end
    reset_switch =0;

end

% Determine Current Saturation Level
if(u_rms(end) > 9 && sat_override == 0)

    % Low Power Mode
    u_sat = I_still;
    sat_override = 1;
    t_over = t;

elseif(u_rms(end) > In_bound)

    % Continuous Power Mode
    u_sat = In;

else

    % Maximum Power Mode
    u_sat = Imax;
end

% Cooling Down Period Override for Low Power Mode
if(sat_override == 1)
% In here the LaelapsHalt() routine should be called
%  in the actual firmware
    u_sat = I_still;
    sat_override = ~((t-t_over)>=2.*tc1);

    if(sat_override == 0)
        n = offset;
    end
end
```

```
% Saturate the Input Current
u = sat(u,u_sat);
```

7.3 MOPS Validation and Results

Towards validating the proposed design, a simulation was created in *Matlab* [52]. The dynamic equations (7-1) represent the motor-plant. The user defines a current input as a function of time, as well as the desired time horizon of the simulation. MOPS is running in the background and the results may be reviewed in graphs that pop-up in the post-processing step of the framework. Note that there are provisions for both types of Laelaps II motors.

The accuracy of the results is bulk, as expected from an indirect thermal protection software solution like this, without real-time temperature measurements. However, this does not imply that the motors are in danger of failure. By tuning the thresholds (Figure 7-2), it is ensured that the design is on the “safe-side”. The results indicate that the algorithm was stretched in very demanding, rather unlikely operating conditions and nevertheless delivers what is promised.

Figure 7-4 and Figure 7-5 show a non-stop current, below the corresponding continuous operation limit, supplied to the knee’s and hip’s motor, respectively. Furthermore, the inputs in Figure 7-6 and Figure 7-7 exceed the drive’s maximum current potential and as such have to be saturated. Also, there are some added disturbances like operation halt for a significant amount of time that simulates an unexpected cooling-down period. The previously presented modifications of the algorithm’s basic idea were created to prevent a failure in such unpredictable circumstances. Lastly, Figure 7-8 and Figure 7-9 illustrate a mixture of the above situations for both motors, again.

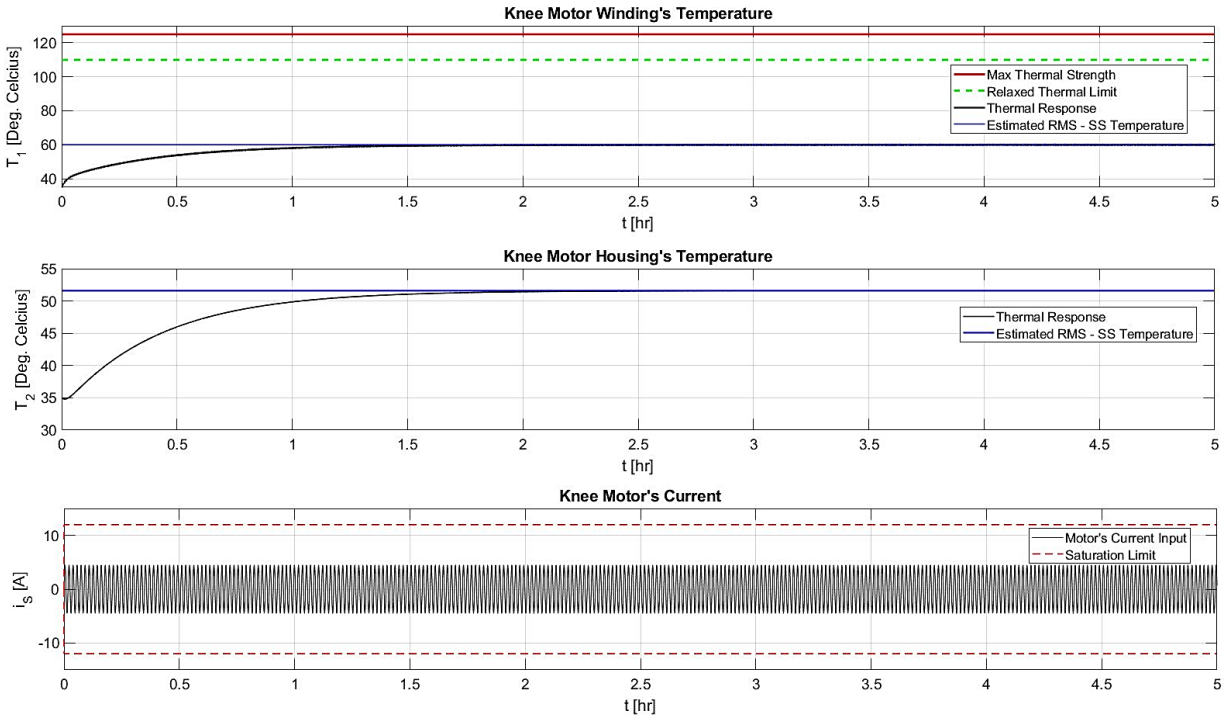


Figure 7-4. Knee’s continuous non-stop operation.

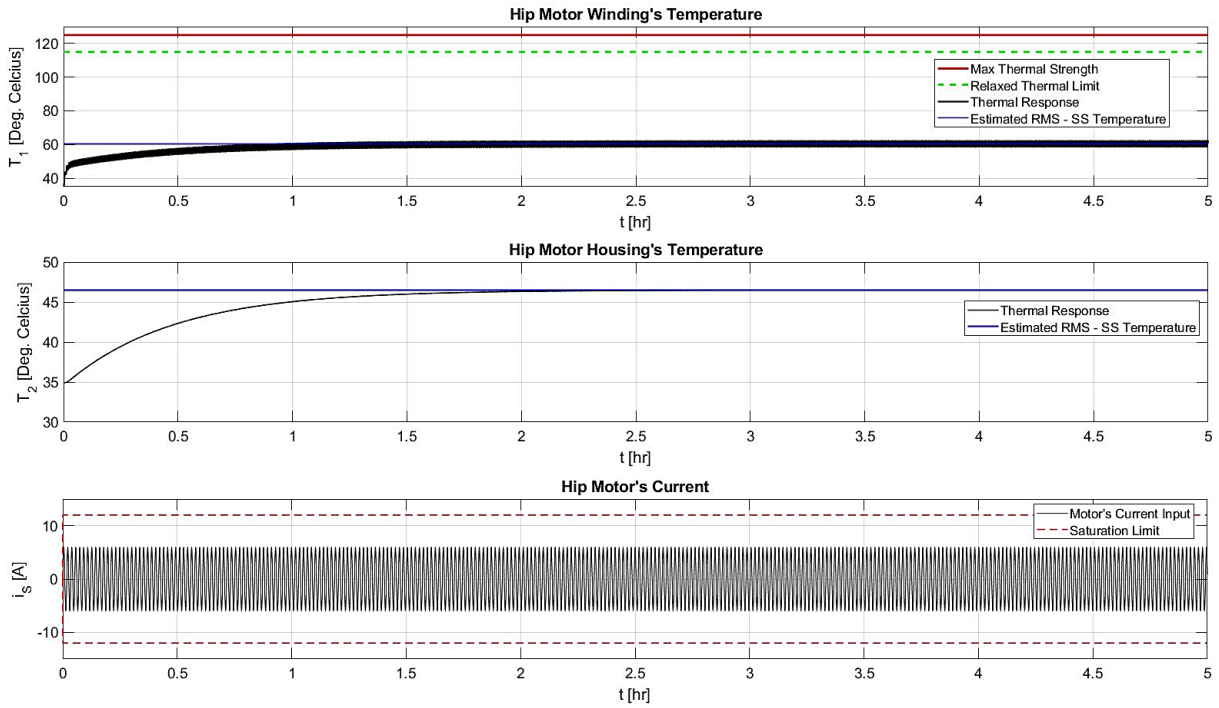


Figure 7-5. Hip's continuous non-stop operation.

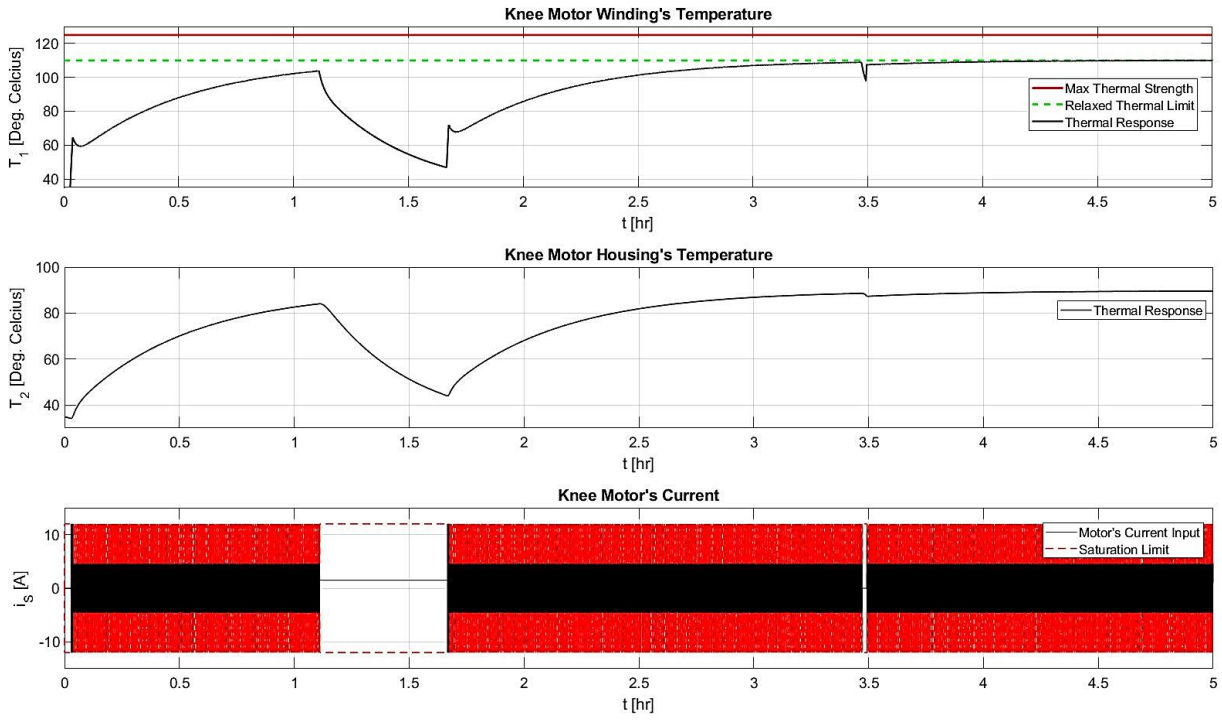


Figure 7-6. Knee's overloading operation.

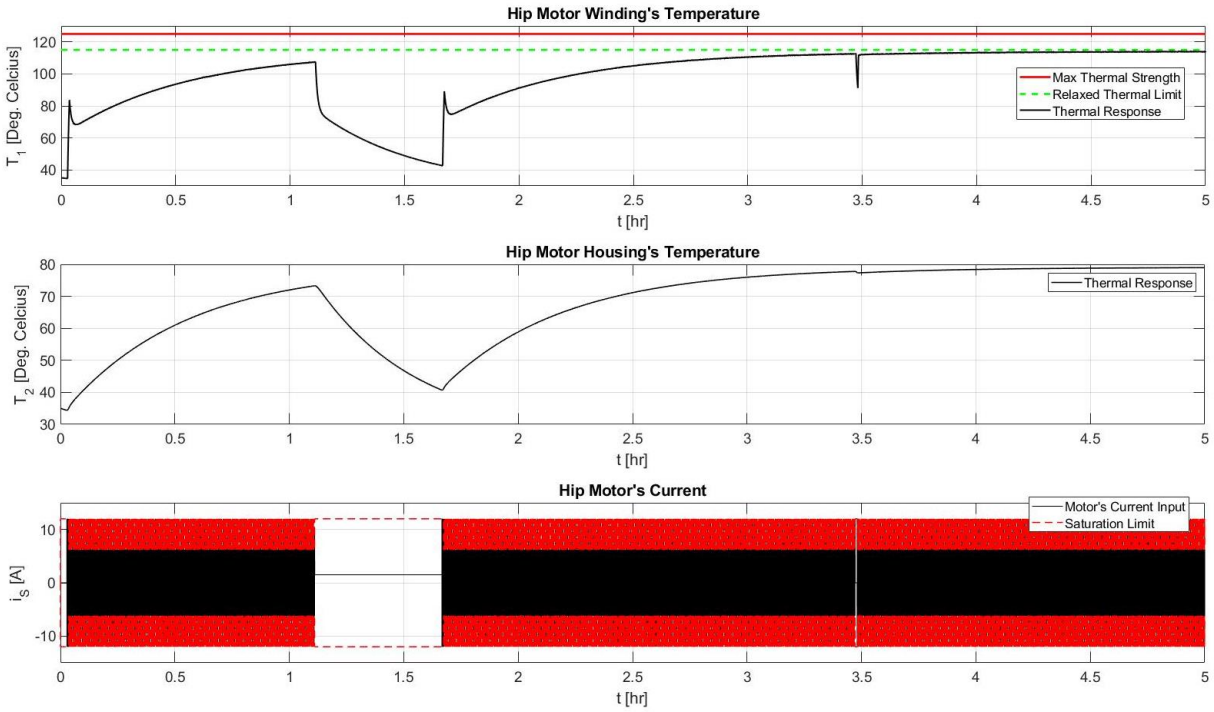


Figure 7-7. Hip's overloading operation.

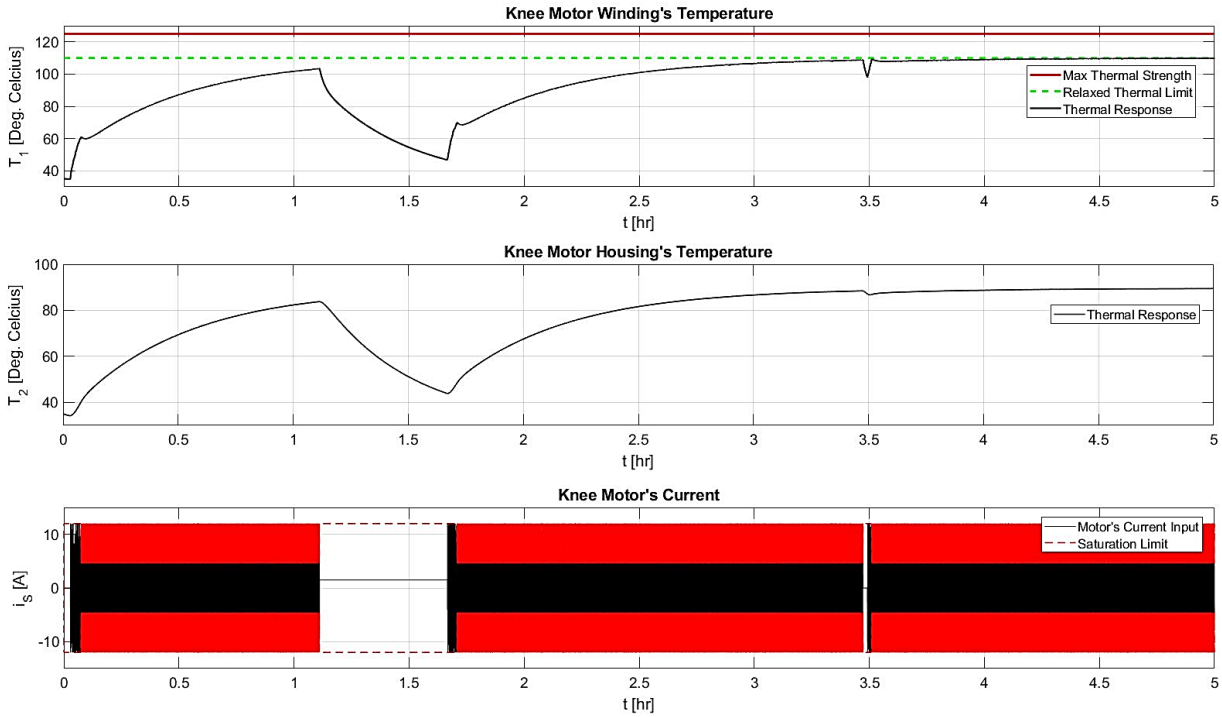


Figure 7-8. Knee's mixed conditions operation.

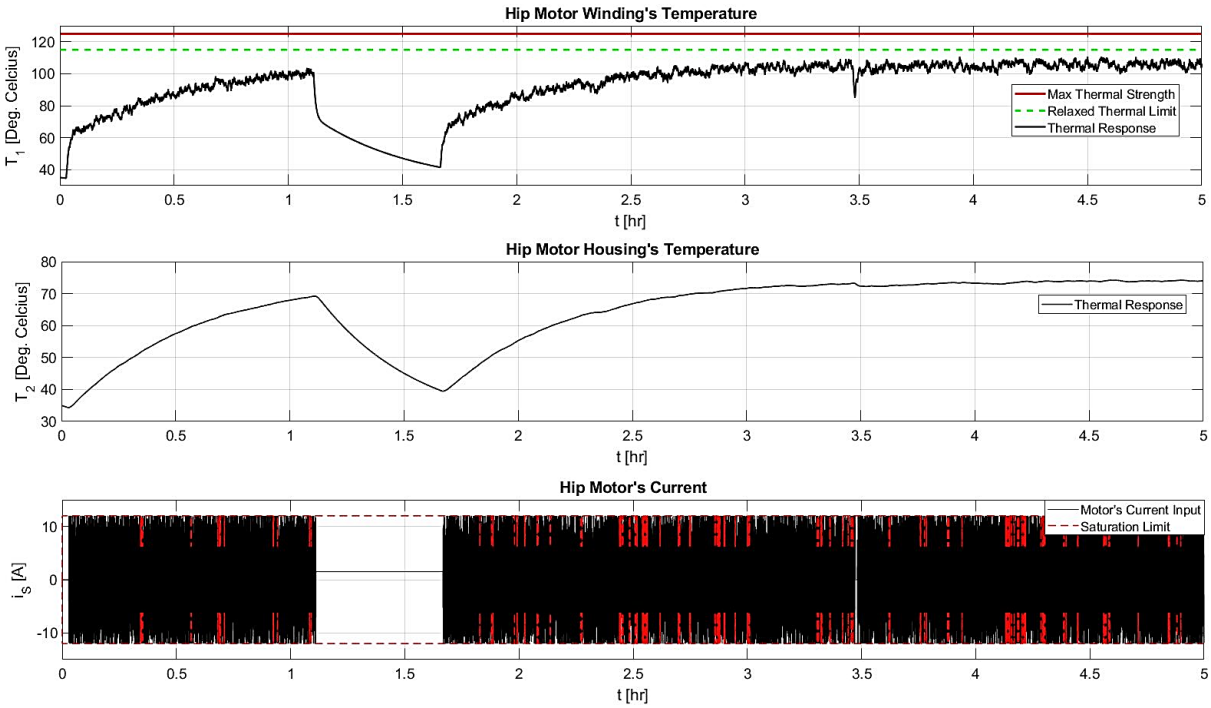


Figure 7-9. Hip's mixed conditions operation.

7.4 Implementation Concerns

The algorithm is designed to operate on a digital-processing unit, like an MCU. In digital control systems, to ensure that the captured response is accurate, the sampling frequency is a major parameter that needs to be tuned. Maxon provides the temperature limit for each motor's winding. To ensure that this limit is not violated, MOPS must be tuned according to the winding's response. Thus, each motor's winding bandwidth should be determined. To this end, the Bode plots of the system (7-1) are illustrated in Figure 7-10 and Figure 7-11 for the hips' and the knees' motors, respectively. By consulting these plots, the aforementioned bandwidth frequencies (f_{bw}) are determined and presented in equation (7-7) for each motor type.

$$\omega_{bw} = \begin{cases} 0.0362, \text{ for the Hip} \\ 0.0129, \text{ for the Knee} \end{cases} [rad/s] \Rightarrow F_{sampling} \geq 2 f_{bw} = \begin{cases} 0.0115, \text{ for the Hip} \\ 0.0041, \text{ for the Knee} \end{cases} [Hz] \qquad (7\text{-}7)$$

The Nyquist theorem [66] states that any sampling frequency ($F_{sampling}$) above the second harmonic of the system's bandwidth is free of aliasing distortions and can capture the actual system's behavior. Hence, the loop frequency of the algorithm should be kept above that frequency, given by equation (7-7). Apparently, the dynamics of the thermal system are significantly slower than the control system's bandwidth (Chapter 3). So, executing MOPs in the same frequency with the control loop would be an exaggeration. Generally, a rule of thumb would be to run MOPs in ten times the frequency of (7-7) to be on the safe side. However, since the algorithm's computational needs remains low, a rationale choise would be to run MOPs inside the EtherCAT loop in order to use the firmware's current configuration and peripherals.

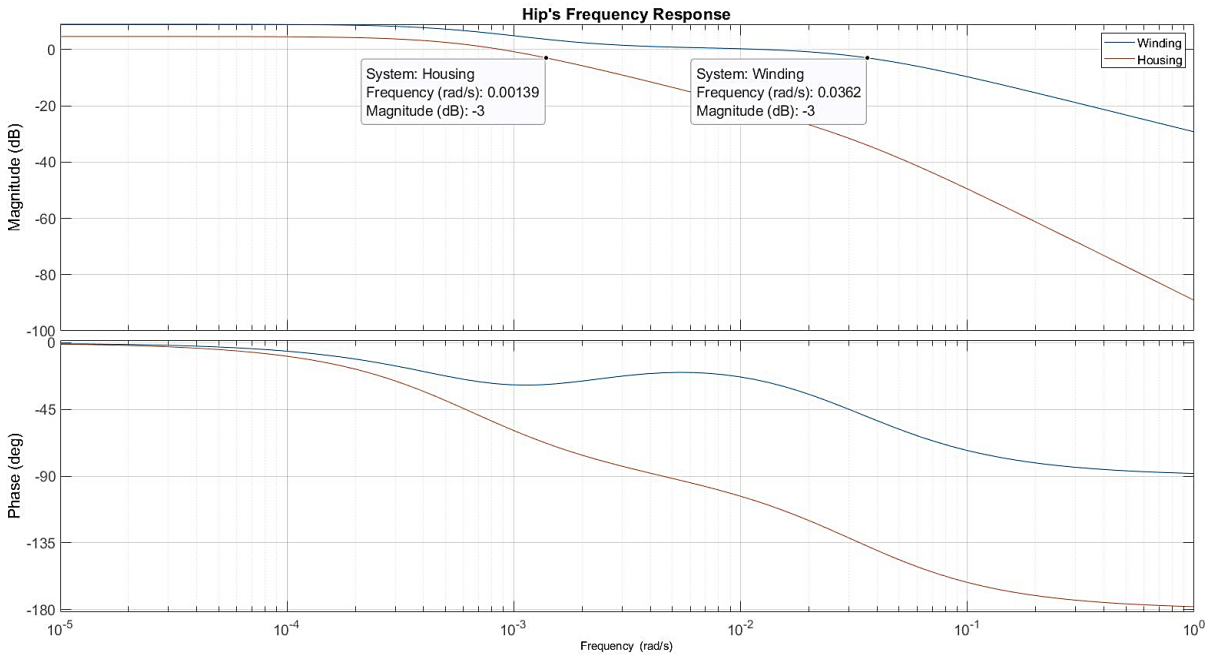


Figure 7-10. Hip motor's Bode plot and bandwidth.

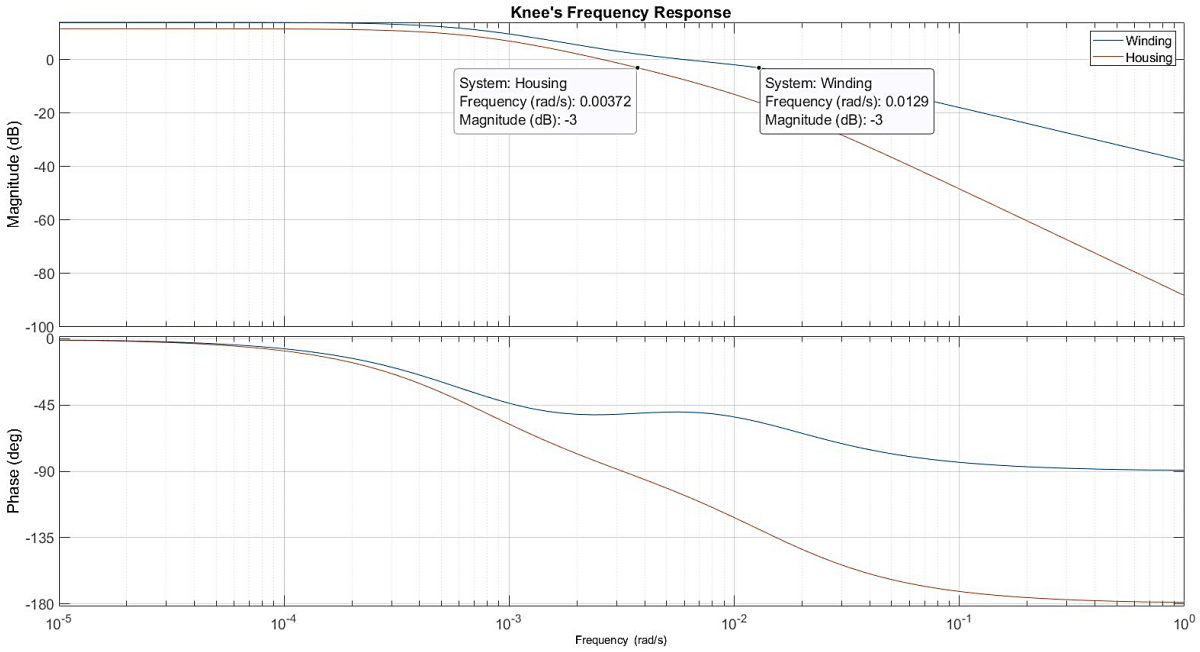


Figure 7-11. Knee motor's Bode plot and bandwidth.

7.5 Conclusion

The above analysis suggests that MOPS operates within the desired operational limits and has proven robustness. The design goals are met in a variety of different conditions with non-periodic phenomena and extensive periods of operation. The results indicate that the hip's brushless motor demonstrates superior thermal performance and is capable of operating in extended overloading conditions. A remark is that the motors should be at approximately room temperature at the firmware's startup for the correct initialization of the algorithm, since the temperature is controlled indirectly. Admittedly, the algorithm neglects the thermal behavior of

the installed drives [65]. This should be taken into account in a potential future implementation of MOPs. Note that since the algorithm is supplied with current measurements after each motor's drive, its operation could not lead to damaging the actuators. The drives have internal safety controls that regulate their output current depending on their status and temperature. At the same time, the current sensors that will be installed to the motors will supply MOPs with the actual current that is applied to each actuator. So, the system is expected to operate flawlessly.

8 Conclusions and Future Work

8.1 Conclusions

The already implemented architecture of Laelaps II has been proven to be most insightful, by exploiting several state-of-the-art technologies and design trends to facilitate new feature integration and development. The combination of a highly demanding control application, such as the motion control of a quadruped, with the most deterministic communication protocol available on the market, was a remarkably wise choice. The significant amount of time, necessary to get an overall grasp of EtherCAT protocol is compensated by its modularity, scalability and performance.

In this thesis, an effort was made to improve certain aspects of the already implemented firmware. Towards this, a smoother planner was introduced that resulted in a more stable, car-like motion of the robot. Simulations, along with videos [94], indicate that the robot's motion presents increased stability since for the first time it can move without lateral body support. Furthermore, special attention was given to improve the execution time of the firmware in order to exploit the increased bandwidth that Laelaps' distributed architecture offers and make provisions for future advancements. For the first time, the actual execution flow of the firmware was investigated (see Section 3.5.6), proving that the control loop has not been running as expected, even after this thesis' improvements. The achieved performance is illustrated in Table 3-6. Note that even with this latency in the control loop's execution, its frequency is adequate and no disruptions were observed during the robot's operation. Additionally, this work adopted the latest standards in code proofing, since the developed firmware complies with the industry's MISRA C: 2012 standard. The code is designed to facilitate future maintenance tasks by other team members. As it was realized many run-time bugs were found using static analysis tools, while the adopted coding style resulted in having a clean and straight-forward code structure. This has significantly improved the efficiency of the code reviews.

This work introduced IMU sensors for the first time in Laelaps' ecosystem. Their integration proved to be seamless, while results (Section 4.8) indicate that the created firmware operates within design specifications. From its hardware aspect, the created slave can operate in any mobile system, since it has separate power supply channels that support a wide range of the common voltages met in batteries. Its portable design is both easy to assemble and mount, while the firmware structure is easy to comprehend and modify to cater to any needs that emerge.

Laelaps II was designed to execute highly dynamic behaviors and to this end, it is equipped with powerful actuators and a relatively light body frame. A major drawback in exploring such regimes had been the motor overheating problem in high currents. To account for this, a simple algorithm was designed to ensure the motors' nominal operation. Even though that it is based solely on current measurements to regulate the output power, the simulations indicate that it is a robust, on the "safe-side" design (Section 7.3).

To optimize the development of future quadrupeds and Laelaps II itself, its internal parameters should be estimated accurately. The interdependent nature of its hardware components and its overall dynamic performance cannot be ignored by the design process. So a framework was developed for the design of parameter estimation experiments. Moreover, certain techniques were employed to overcome the pathogenies of the proposed

methods. Admittedly, the optimized trajectory minimizes the noise's impact and other distortions that downgrade the end-result of the identification process (Section 6.4.7). Of course, it constitutes an initial effort and to become functional on the real robot, certain hardware and firmware modifications are required.

To summarize, this thesis demonstrates how certain improvements on Laelaps II can lead to many interesting results. From the firmware aspect, the overall execution time is reduced and several features were added to ensure the integrity of the equipment and the safety of its users. Moreover, MOPs will eventually unlock the otherwise locked PWM saturation limits and the investigation of high-power dynamic regimes will become possible. Last but not least, by the introduction of the absolute encoder modules the experiment workflow has been significantly shortened, through automating the formerly tedious startup sequence of the quadruped.

8.2 Future Work

Nevertheless, the various improvements that Laelaps II was subjected to, there is always room for even more. This study has involved many aspects of its firmware and hardware. For starters, it became obvious that the Delfino dual-core capabilities and various peripherals, like DMA, could increase the firmware's performance and service more demanding applications in the future. One idea is the implementation of state estimation algorithms in the second core of Delfino to design even better control schemes. Moreover, the control loop's latency should be accounted and solved, since the motion control is a very time critical task. To this end, certain proposals were made in this thesis (see Sections 3.5.5 and 3.5.6) that should be considered in the next version of the firmware.

Another step could be the execution of the necessary hardware modifications to accommodate current sensors. This will eventually enable the execution of the designed identification experiments and estimate each leg's parameters. Also, it will become possible to implement in firmware, the described MOPS algorithm and unlock the PWM saturation limits.

As for the IMUs, they could be used in an actual whole-body state-estimation algorithm and have an insight into the body's actual motion during gait-cycles. Moreover, another idea could be the design of an algorithm to calculate the free acceleration, which is the gravity-free reading of the accelerometers that currently is not calculated. Apparently, there are multiple options and it is up to the CSL-EP Legged Robots Team how exactly these modules are going to be exploited in the future.

The installation of absolute encoders has simplified significantly the experiments' workflow, by automating the startup sequence of Laelaps II. The next step in this direction is to enable the Linux-ROS EtherCAT master that is still in development by members of the CSL-EP Legged Robots Team. This way, Laelaps II is going to exploit the ROS capabilities that this EM offers, which translate to better interconnection with more tools available for further development. The potential of such a modification would be remarkable since ROS is gradually becoming the leading standard in robotics research. Last but not least, power consumption should be studied and optimized along with the introduction of batteries to achieve true mobility. In this work, certain provisions were made to facilitate this transition, since all of the low-power electronics can operate with standard battery voltages.

Generally, there are many things to consider and I cannot possibly know what the future holds, but I am sure of one thing; great things are coming and I am glad that I have been involved in the overall process.

9 References

[1] "Boston Dynamics Atlas," *Boston Dynamics*. https://www.bostondynamics.com/atlas.

[2] "Boston Dynamics Spot," *Boston Dynamics*. https://www.bostondynamics.com/spot.

[3] "Boston Dynamics Legacy Robots," *Boston Dynamics*. https://www.bostondynamics.com/legacy.

[4] "ANYbotics ANYmal C," *ANYbotics*. https://www.anybotics.com/anymal-legged-robot/.

[5] "Biomimetic Robotics Lab," *MIT MECHE*. https://biomimetics.mit.edu.

[6] "Agility Robotics DIGIT," *Agility Robotics*. https://www.agilityrobotics.com/digit.

[7] Mich. Dearborn, "NO BONES ABOUT IT: FORD EXPERIMENTS WITH FOUR-LEGGED ROBOTS, TO SCOUT FACTORIES, SAVING TIME, MONEY," Jul. 27, 2020. https://media.ford.com/content/fordmedia/fna/us/en/news/2020/07/27/no-bones-about-it-ford-experiments-with-four-legged-robots.html.

[8] "CSL-EP Legged Robots Team Research," *CSL-EP Legged Robots Team Website*, 2020. http://nereus.mech.ntua.gr/legged/.

[9] R. Buchanan, L. Wellhausen, M. Bjelonic, T. Bandyopadhyay, N. Kottege, and M. Hutter, "Perceptive whole-body planning for multilegged robots inconfined spaces," *J Field Robotics*, vol. 38, no. 1, pp. 68–84, Jan. 2021, doi: 10.1002/rob.21974.

[10] M. Camurri, M. Ramezani, S. Nobili, and M. Fallon, "Pronto: A Multi-Sensor State Estimator for Legged Robots in Real-World Scenarios," *Frontiers in Robotics and AI*, p. 18.

[11] M. Bloesch *et al.*, "State Estimation for Legged Robots - Consistent Fusion of Leg Kinematics and IMU," presented at the Robotics: Science and Systems 2012, doi: 10.15607/RSS.2012.VIII.003.

[12] "Inertial Sensors, GPS and Odometry," in *Handbook of Robotics*, 2nd ed., Springer.

[13] "Basics of Rotary Encoders: Overview and New Technologies," *Machine Design*. https://www.machinedesign.com/automation-iiot/sensors/article/21831757/basics-of-rotary-encoders-overview-and-new-technologies.

[14] "Sensing and Estimation," in *Handbook of Robotics*, Springer, pp. 88–106.

[15] K. Machairas and E. Papadopoulos, "An Active Compliance Controller for Quadruped Trotting," in *2016 24th Mediterranean Conference on Control and Automation (MED)*, Athens, Greece, Jun. 2016, pp. 743–748, doi: 10.1109/MED.2016.7536064.

[16] D. J. Hyun, S. Seok, J. Lee, and S. Kim, "High speed trot-running: Implementation of a hierarchical controller using proprioceptive impedance control on the MIT Cheetah," *The International Journal of Robotics Research*, vol. 33, no. 11, pp. 1417–1445, Sep. 2014, doi: 10.1177/0278364914532150.

[17] V. Barasuol, J. Buchli, C. Semini, M. Frigerio, E. R. De Pieri, and D. G. Caldwell, "A reactive controller framework for quadrupedal locomotion on challenging terrain," in *2013 IEEE International Conference on Robotics and Automation*, Karlsruhe, Germany, May 2013, pp. 2554–2561, doi: 10.1109/ICRA.2013.6630926.

[18] K. Machairas and E. Papadopoulos, "On Dynamic Quadrupedal Gaits Using Active Compliance Control," Camp Ohiyesa, MI, USA, Jun. 2016, [Online]. Available: http://nereus.mech.ntua.gr/Documents/pdf_ps/dw16a.pdf.

[19] A. S. Mastrogeorgiou, Y. S. Elbahrawy, K. Machairas, A. Kecskemethy, and E. Papadopoulos, "Evaluating deep reinforcement learning algorithms for quadrupedal slope handling," presented at the 23rd International Conference Series on Climbing and Walking Robots and the Support Technologies for Mobile Machines, Moscow, Russia, Aug. 2020, doi: 10.13180/clawar.2020.24-26.08.58.

[20] Stamatis Athiniotis, "Firmware design for microcontrollers on EtherCAT network for quadruped robot motion control," *Diploma Thesis*, National Technical University of Athens, Athens, 2018.

[21] J. Braun, *Formulae Handbook*, 4th ed. Sachseln: Maxon Group, 2018.

[22] S. Dallas, K. Machairas, K. Koutsoukis, and E. Papadopoulos, “A Leg Design Method for High Speed Quadrupedal Locomotion,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vancouver, BC, Canada, Sep. 2017, pp. 4877–4882, doi: 10.1109/IROS.2017.8206365.
[23] “Extended Kalman Filter and System Identification,” in *Kalman Filtering*, Berlin, Heidelberg: Springer, 2009, pp. 108–130.
[24] M. P. Wensing, K. Sangbae, and E. J.-J. Slotine, “Linear Matrix Inequalities for Physically Consistent Inertial Parameter Identification: A Statistical Perspective on the Mass Distribution,” *IEEE Robotics and Automation Letters*, vol. 3, no. 1, pp. 60–67, Jan. 2018, doi: 10.1109/LRA.2017.2729659.
[25] A. N. Bompos, K. P. Artemiadis, S. A. Oikonomopoulos, and J. K. Kyriakopoulos, “Modeling, full identification and control of the mitsubishi PA-10 robot arm,” in *2007 IEEE/ASME international conference on advanced intelligent mechatronics*, Zurich, Switzerland, Sep. 2007, pp. 1–6, doi: 10.1109/AIM.2007.4412421.
[26] J. K. Keesman, *System Identification: An Introduction*, 1st ed. Springer.
[27] L. Ljung, *System Identification: Theory for the User*, 2nd ed. Upper Saddle River, NJ 07458: Prentice Hall PTR.
[28] “Simplifying Current Sensing.” Texas Instruments Incorporated, [Online]. Available: https://www.ti.com/lit/eb/slyy154a/slyy154a.pdf.
[29] “Laelaps II Quadruped Overview,” *CSL-EP Legged Robots Team Website*, 2020. http://nereus.mech.ntua.gr/legged/?page_id=161.
[30] “Beckhoff’s EtherCAT Technology,” *ETG Main Website*, 2020. https://www.ethercat.org/default.htm.
[31] “IIT’s HyQReal Quadruped,” *IIT Dynamic Legged Systems*, 2020. https://dls.iit.it.
[32] “Giant Magellan Telescope with site-wide real-time connectivity and 3,000 precisely controlled servo axes,” Sep. 14, 2020. https://www.beckhoff.com/english.asp?press/news3320.htm?pk_campaign=News&pk_kwd=GMTO&pk_source=LinkedIn&pk_medium=socialmedia_post.
[33] “EtherCAT Specification.” EtherCAT Technology Group, Sep. 15, 2017, [Online]. Available: https://www.ethercat.org/memberarea/download/ETG1000-x-R-V1i0i4.zip.
[34] M. Karamousadakis, “Real-time programming of EtherCAT master in ROS for a quadruped robot,” *Diploma Thesis*, National Technical University of Athens, Athens, 2019.
[35] “Hardware Data Sheet Section I.” Beckhoff, Feb. 21, 2017, [Online]. Available: https://download.beckhoff.com/download/Document/io/ethercat-development-products/ethercat_esc_datasheet_sec1_technology_2i3.pdf.
[36] “CANopen – The standardized embedded network,” *CiA Knowledge Base*. https://www.can-cia.org/canopen/.
[37] “EtherCAT Modular Device Profile ETG.5001.” EtherCAT Technology Group, Feb. 17, 2016, [Online]. Available: https://www.ethercat.org/memberarea/download/ETG5001_1_V0i9i0_S_D_MDP_GeneralSpec.pdf.
[38] “EtherCAT Slave Stack Code (SSC) ET9300,” *ETG Main Website*. https://www.ethercat.org/en/products/54FA3235E29643BC805BDD807DF199DE.htm.
[39] “EtherCAT Technology Group.” https://www.ethercat.org/default.htm.
[40] “TwinCAT 3 Automation Software,” *Beckhoff Website*. https://www.beckhoff.com/en-us/products/automation/twincat/.
[41] “EtherLAB EtherCAT Master,” *IgH Website*. https://www.etherlab.org/en/index.php.
[42] “C2000 Delfino MCU F28379D LaunchPad™ development kit,” *Texas Instruments Website*. https://www.ti.com/tool/LAUNCHXL-F28379D.

[43] “FB1111 | EtherCAT piggyback controller board.” Beckhoff, [Online]. Available: https://download.beckhoff.com/download/Document/io/ethercat-development-products/beckhoff_fb1111-014x_v22.pdf.
[44] “ftrace - Function Tracer,” *ftrace Documentation*. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.
[45] “Application Note ET9300 (EtherCAT Slave Stack Code).” Beckhoff, Nov. 14, 2017, [Online]. Available: https://download.beckhoff.com/download/document/io/ethercat-development-products/an_et9300_v1i8.pdf.
[46] “EC45 Brushless 250 Watt Motor Catalogue.” Maxon Group, [Online]. Available: https://www.maxongroup.com/medias/sys_master/root/8841182445598/EN-231.pdf.
[47] “Planetary Gearhead GP 52 C Catalogue.” Maxon Group, [Online]. Available: https://www.maxongroup.com/medias/sys_master/root/8841219145758/EN-401-402.pdf.
[48] “RE50 Graphite Brushes 200 Watt Motor.” Maxon Group, [Online]. Available: https://www.maxongroup.com/medias/sys_master/root/8841119367198/EN-142.pdf.
[49] K. Machairas and E. Papadopoulos, “An Analytical Study on Trotting at Constant Velocity and Height,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Madrid, Spain, Oct. 2018, pp. 3279–3284, doi: 10.1109/IROS.2018.8593686.
[50] George Bolanakis, “Σχεδιασμός & Υλοποίηση Ηλεκτρονικού Συστήματος Τετράποδου Ρομπότ,” *Diploma Thesis*, National Technical University of Athens, Athens, 2018.
[51] “RMB28 / RMF44 angular magnetic encoder modules.” RLS, May 13, 2020, [Online]. Available: https://www.rls.si/eng/fileuploader/download/download/?d=1&file=custom%2Fupload%2FRMB28D01_16_EN_data_sheet.pdf.
[52] CSL-EP’s Legged Robots Team, “EtherCAT Laelaps Motion Control.” https://bitbucket.org/csl_legged/laelaps-leg-firmware/src/master/.
[53] “Misumi 5 Series/Post-Assembly Insertion Stopper Nuts,” *Misumi Website*. https://uk.misumi-ec.com/vona2/detail/110302608320/.
[54] “HEDL-5640 Datasheet.” Broadcom, [Online]. Available: https://docs.broadcom.com/doc/AV02-0993EN.
[55] R. Reeder, “An Inside Look at High Speed Analog-to-Digital Converter Accuracy,” *Analog Devices*. https://www.analog.com/en/technical-articles/an-inside-look-at-high-speed-analog-to-digital-converter-accuracy.html.
[56] Richard Zurawski, *Industrial Communication Technology Handbook*, 2nd ed. CRC Press, 2017.
[57] “Grobotronics DC-DC Step-Down Regulator,” *Grobotronics*. https://grobotronics.com/dc-dc-step-down-5v-2a.html.
[58] “LM1117 800-mA, Low-Dropout Linear Regulator.” Texas Instruments Incorporated, [Online]. Available: https://www.ti.com/lit/ds/symlink/lm1117.pdf.
[59] “TI’s Code Composer Studio.” https://www.ti.com/tool/download/CCSTUDIO.
[60] *ether_ros CSL-EP Legged EtherCAT Master Package*. 2020.
[61] “TI’s ControlSuite,” *Texas Instruments Website*. https://www.ti.com/tool/CONTROLSUITE.
[62] “MISRA,” *MISRA Coding Standard*. https://www.misra.org.uk/MISRAHome/tabid/55/Default.aspx.
[63] “Oversampling,” *Wikipedia*. https://en.wikipedia.org/wiki/Oversampling.
[64] “TMS320F287xD Dual-Core Delfino Microcontrollers Technical Reference Manual.” Texas Instruments Incorporated, [Online]. Available: https://www.ti.com/lit/ug/spruhm8h/spruhm8h.pdf.
[65] “Analog Servo Drive AZBDC12A8.” Advanced Motion Controls, [Online]. Available: https://dpk3n3gg92jwt.cloudfront.net/domains/amc/pdf/AMC_Datasheet_AZBDC12A8.pdf.

[66] “Nyquist–Shannon sampling theorem,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Nyquist–Shannon_sampling_theorem.
[67] “C2000 Digital Control Library User’s Guide.” Texas Instruments Incorporated, [Online]. Available: https://dev.ti.com/tirex/explore/node?node=AKZRbh4oxv98HaO0YKjygQ__gYkahfz__LATEST.
[68] “C2000™ CLA Software Development Guide,” *Texas Instruments Website*. https://software-dl.ti.com/C2000/docs/cla_software_dev_guide/faq.html.
[69] M. Barr, *Embedded C Coding Standard*, BARR-C: 2018. Barr Group.
[70] “ADIS16364 IMU Datasheet.” Analog Devices, 2019 2009, [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ADIS16364.pdf.
[71] “ADIS16375 IMU Datasheet.” Analog Devices, 2019 2010, [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ADIS16375.pdf.
[72] J. Bohnenberger, *Beschreibung einer Maschine zur Erläuterung der Geseze der Umdrehung der Erde um ihre Axe, und der Veränderung der Lage der letzteren. Nebst einer Abbildung.* Tübingen, 1817.
[73] “Coriolis force,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Coriolis_force.
[74] “Piezoelectricity,” *Wikipedia*. https://en.wikipedia.org/wiki/Piezoelectricity.
[75] *Autodesk Eagle*. Autodesk.
[76] “TMS320F2837xD Dual-Core Delfino™ Microcontrollers Datasheet.” Texas Instruments Incorporated, Dec. 2013, [Online]. Available: https://www.ti.com/lit/ds/sprs880m/sprs880m.pdf.
[77] “LM1085 3-A Low Dropout Positive Regulators Datasheet.” Texas Instruments Incorporated, [Online]. Available: https://www.ti.com/lit/ds/symlink/lm1085.pdf.
[78] “L78 Positive Low Dropout Regulators Datasheet.” STMicroelectronics, Sep. 2018, [Online]. Available: https://www.st.com/resource/en/datasheet/l78.pdf.
[79] “LDO Basics.” Texas Instruments Incorporated, [Online]. Available: https://www.ti.com/lit/eb/slyy151a/slyy151a.pdf.
[80] CADENCE PCB SOLUTIONS, “Tips for Optimal High Speed SPI Layout Routing,” *Tips for Optimal High Speed SPI Layout Routing*. https://resources.pcb.cadence.com/blog/2019-tips-for-optimal-high-speed-spi-layout-routing.
[81] K. Machairas, “Design and Implementation of the Electrical/ Electronic Subsystem, and Real-time Programming for a Space Emulator Robot,” *Diploma Thesis*, National Technical University of Athens, Athens, 2013.
[82] “Printed Circuit Board (PCB) Design Issues,” in *Linear Circuit Design Handbook*, Newnes/Elsevier, 2008.
[83] A. Papatheodorou, *ADIS16364 IMU EtherCAT Slave CSL-EP Legged BitBucket Repository*. Athens: CSL-EP Legged - NTUA, 2020.
[84] A. Papatheodorou, *ADIS16375 IMU EtherCAT Slave CSL-EP Legged BitBucket Repository*. Athens: CSL-EP Legged - NTUA, 2020.
[85] P. Dhaker, “Introduction to SPI Interface.” Analog Devices, Sep. 2018, [Online]. Available: https://www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf.
[86] “EtherCAT Slave Implementation Guide ETG2200.” EtherCAT Technology Group, Nov. 23, 2018, [Online]. Available: https://www.ethercat.org/download/documents/ETG2200_V3i1i0_G_R_SlaveImplementationGuide.pdf.

[87] “Hardware Data Sheet Section II.” Beckhoff, Oct. 09, 2020, [Online]. Available: https://download.beckhoff.com/download/Document/io/ethercat-development-products/ethercat_esc_datasheet_sec2_registers_3i0.pdf.
[88] “Hardware Data Sheet Section III.” Beckhoff, Feb. 21, 2017, [Online]. Available: https://download.beckhoff.com/download/Document/io/ethercat-development-products/ethercat_et1100_datasheet_v2i0.pdf.
[89] “ETG.2100 EtherCAT Network Information (ENI) Specification.” EtherCAT Technology Group, Sep. 30, 2015, [Online]. Available: https://www.ethercat.org/memberarea/download/ETG2100_V1i0i1_S_R_ENISpec.pdf.
[90] “Xsens MTi-200 IMU,” *Xsens Web Page*. https://www.xsens.com/products/mti-100-series.
[91] “Xsens MTi 100 Series IMU Manual.” Xsens, [Online]. Available: https://content.xsens.com/mti-100-manual?hsCtaTracking=46bfb65b-9f43-4de6-92ef-8a1ca7cffa71%7C030313c6-79ce-4ee2-b90a-dedd4347178a.
[92] M. Looney, “Anticipating and Managing Critical Noise Sources In MEMS Gyroscopes,” *Analog Devices*. .
[93] “RE 50 ø50 mm, Graphite Brushes, 200 Watt Datasheet.” Maxon Group, Apr. 2020, [Online]. Available: https://www.maxongroup.com/medias/sys_master/root/8841119367198/EN-142.pdf.
[94] “Control Systems Lab - EP Team - NTUA,” *Youtube*. https://www.youtube.com/user/CSLabEP/featured.
[95] “Spot BP application,” *Boston Dynamics Spot Applications*. https://www.bostondynamics.com/spot/applications/bp.
[96] “Maxon Standard Specification.” Maxon Group, [Online]. Available: https://www.maxongroup.com/medias/sys_master/root/8828714647582/Standardspez-100-103-2018-EN.pdf.
[97] “Encoder HEDL 5540.” Maxon Group.
[98] “Maxon DC motor and maxon EC motor Key information.” Maxon Group, Nov. 2014, [Online]. Available: https://www.maxongroup.com/medias/sys_master/root/8815460712478/DC-EC-Key-Information-14-EN-42-50.pdf?attachment=true.
[99] A. Papatheodorou, “Leg Identification Bitbucket Repository,” *Bitbucket*. https://bitbucket.org/csl_legged/leg_identification/src/master/.
[100] “INA253 Datasheet.” Texas Instruments Incorporated, Jul. 2018, [Online]. Available: https://www.ti.com/lit/ds/symlink/ina253.pdf?ts=1606079283435&ref_url=https%253A%252F%252Fwww.google.com%252F.
[101] “INA253 EVM User’s Guide.” Texas Instruments Incorporated, May 2018, [Online]. Available: https://www.ti.com/lit/ug/sbou194/sbou194.pdf?ts=1606128363377&ref_url=https%253A%252F%252Fwww.ti.com%252Ftool%252FINA253EVM.
[102] “Hall-effect sensor,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Hall-effect_sensor.
[103] “TMCS1100 Datasheet.” Texas Instruments Incorporated, Jun. 2020, [Online]. Available: https://www.ti.com/lit/ds/symlink/tmcs1100.pdf.
[104] “TMCS1100 EVM User’s Guide.” Texas Instruments Incorporated, Mar. 2020, [Online]. Available: https://www.ti.com/lit/ug/sbou209a/sbou209a.pdf.
[105] “Ramp Function,” *Wikipedia*. https://en.wikipedia.org/wiki/Ramp_function.
[106] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, “Dynamic Parameter Identification,” in *Robotics: Modelling, Planning & Control*, Springer.
[107] O. Christidi-Loumpasefski, K. Nanos, and E. Papadopoulos, “On Parameter Estimation of Space Manipulator Systems Using the Angular Momentum Conservation,” in *2017*

IEEE International Conference on Robotics and Automation (ICRA), Singapore, May 2017, pp. 5453–5458, doi: 10.1109/ICRA.2017.7989641.
[108] "Condition number," *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Condition_number.
[109] "Sequential Quadratic Programming," *Wikipedia*. https://en.wikipedia.org/wiki/Sequential_quadratic_programming.
[110] "Installing TwinCAT 3 Engineering," *Beckhoff Website*. https://infosys.beckhoff.com/content/1033/tc3_installation/6162705803.html.
[111] "C2000 Real-Time MCUs." https://www.ti.com/microcontrollers/c2000-real-time-control-mcus/overview.html.
[112] Kenneth W. Schachter, "TMS320F2837xD Microcontroller Workshop Guide & Lab Manual." Texas Instruments Incorporated, [Online]. Available: https://software-dl.ti.com/trainingTTO/trainingTTO_public_sw/c28x28379/F2837xD_Microcontroller_MDW_2-0.pdf.
[113] "C28x Interrupt Nesting," *Texas Instruments Website*. https://software-dl.ti.com/C2000/docs/c28x_interrupt_nesting/html/index.htm.
[114] Changyi Gu, "Power On & Bootloader," in *Building Embedded Systems*, Apress Media, pp. 5–25.
[115] "CLA Math Library User's Guide." Texas Instruments Incorporated, [Online]. Available: https://dev.ti.com/tirex/explore/node?node=AIYKj-IlPrDpivub6jyGWg__gYkahfz__LATEST.
[116] Changyi Gu, "RAM, DMA & Interrupt," in *Building Embedded Systems*, Apress Media, pp. 47–68.
[117] "F2837xD IPC (Inter-Processor Communication) Device Driver User's Guide." Texas Instruments Incorporated, [Online]. Available: https://dev.ti.com/tirex/explore/node?node=ANC.9pKXDeHu1ipF9J-o.g__coGQ502__LATEST&search=IPC.
[118] "TI Linker Command File Primer," *TI Linker Primer*. https://software-dl.ti.com/ccs/esd/documents/sdto_cgt_Linker-Command-File-Primer.html.
[119] "LaunchXL-F28379D User's Guide." Texas Instruments Incorporated, [Online]. Available: https://www.ti.com/lit/pdf/sprui77.
[120] "LM1085 Product Overview," *Texas Instruments Website*. https://www.ti.com/product/LM1085.

Appendix A. TwinCAT 3 Master Setup

In this section the steps for setting up the Laelaps II TwinCAT 3 project are laid. Note that the project is already configured in the CSL-EP's Control Center PC. Nevertheless, to set it up from scratch follow the steps below.

TwinCAT 3 Installation

Navigate to Beckhoff's website [40] and download *TwinCAT 3.1 – eXtended Automation Engineering (XAE)*. Follow the instructions in [110] to install the downloaded software package. In the current tutorial *Visual Studio Community 2019* (VS) was preinstalled. This however is not mandatory, but nevertheless recommended. Some menus and options are displayed differently in other versions of VS.

TwinCAT 3 Configuration

Before setting up the project, the TwinCAT 3 has to be configured. At first the necessary real-time capable Ethernet drivers should be installed. To do this:

1. Start TwinCAT 3 (*Right-Click* on the TwinCAT icon inside the hidden programs' menu as Figure A-1 suggests). Note that if a different version of Visual Studio is installed the highlighted menu entry is not going to have the *(VS2019)* at the end.

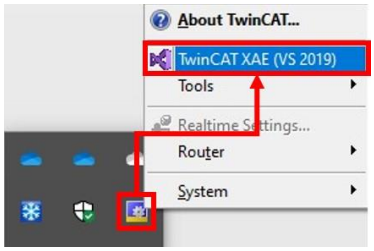


Figure A-1. Start TwinCAT 3.

2. From the TwinCAT 3 start-page, click on *Create a new project* entry and follow the procedure highlighted in Figure A-2. Name the project *LaelapsNetworkProject*.

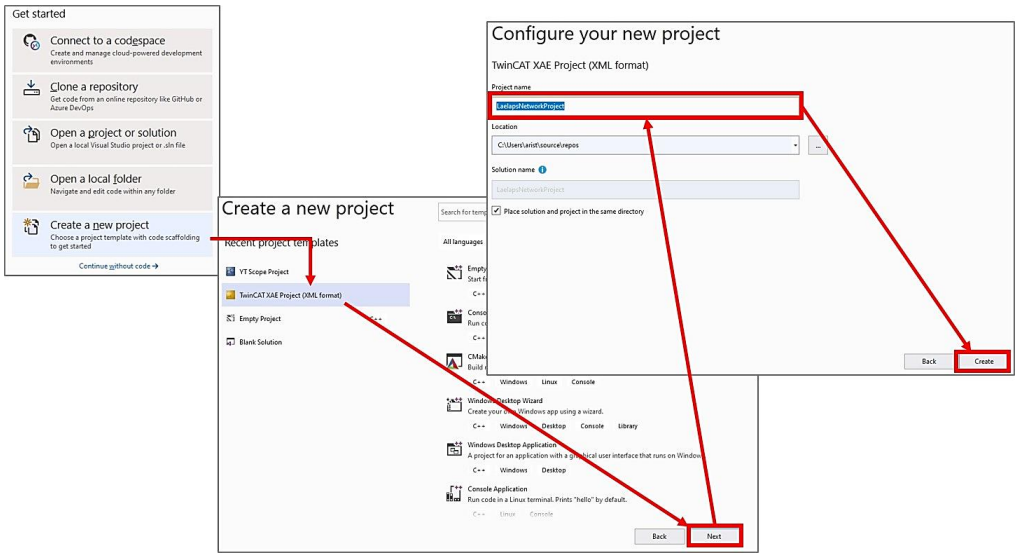


Figure A-2. Create new TwinCAT 3 project.

3. Click on the TwinCAT's entry, at the Extensions menu and select *Show Realtime Ethernet Compatible Devices…* (Figure A-3).

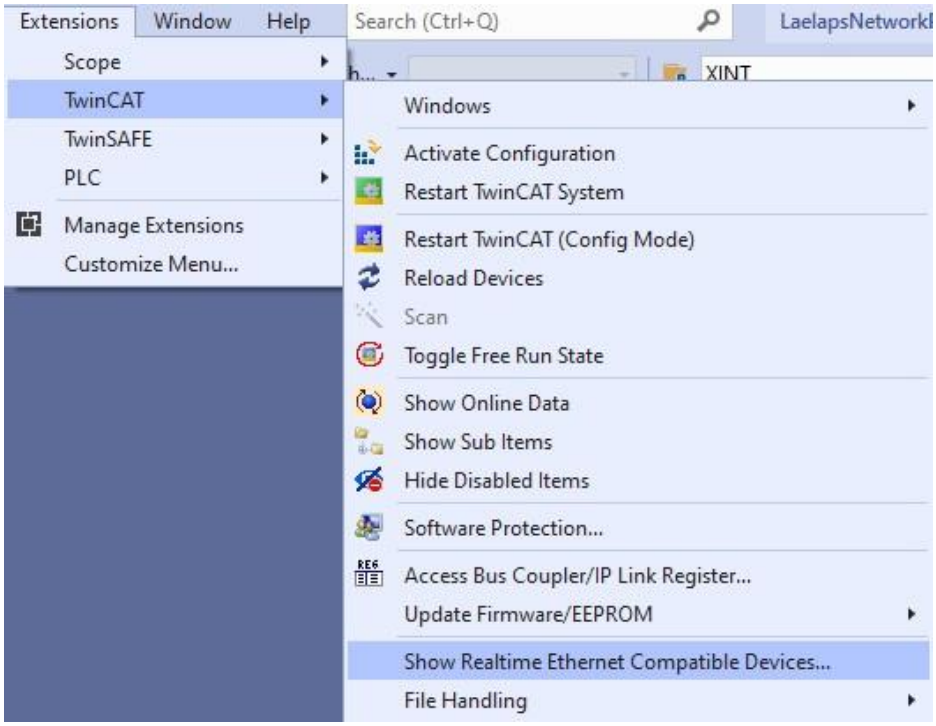


Figure A-3. Show *Realtime Ethernet Compatible Devices...* menu entry.

4. Select and *Install* your Ethernet network adapter, as shown in Figure A-4. Note that for hard real-time operation, only the real-time capable drivers must be used (*Compatible devices*). If the PC does not have such network adapter, TwinCAT 3 is not going to run at high EtherCAT speeds.

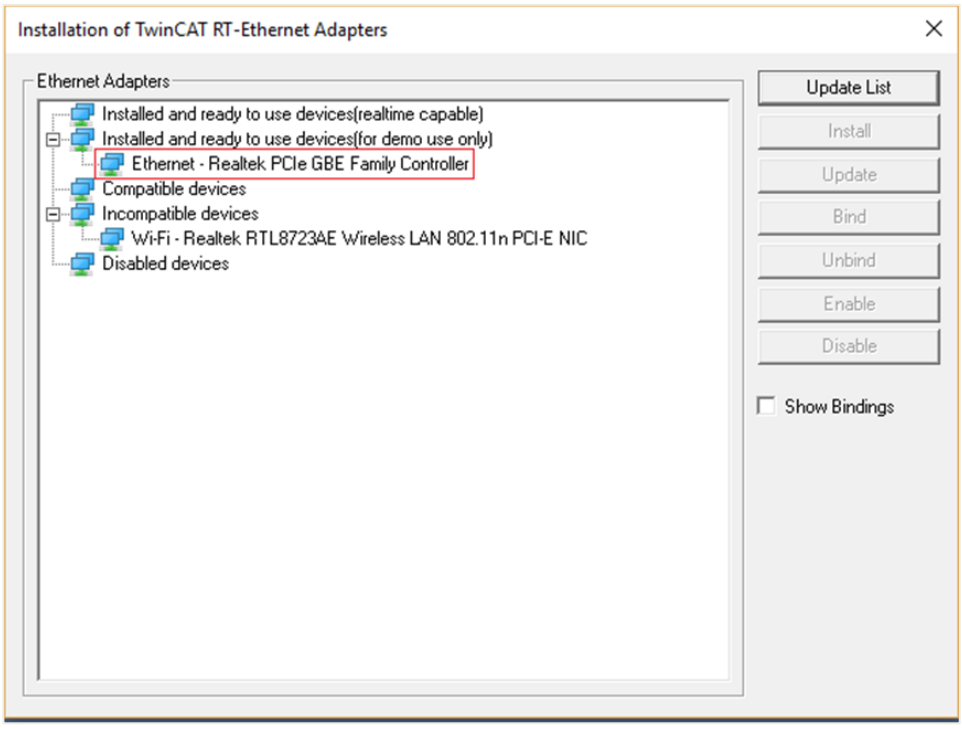


Figure A-4. Installation of RT-Ethernet adapter tab.

5. If this is the first time running TwinCAT 3 in the used PC, before proceeding with the setup, a *.bat* file must be executed to setup specific settings in BIOS. Close TwinCAT

3 instance and navigate to *{Hard Drive Label}:\TwinCAT\3.1\System*. Then **run as administrator** the *win8settick.bat*; note that it runs instantly. Then reboot and open again the recently created TwinCAT 3 project.

6. Next, the required slave ENI files must be imported to TwinCAT 3. With these files TwinCAT recognizes the different slave kinds in the network and understands their Input/Output PDOs. To do this copy and paste the required .xml files (found in [52] [83] [84], under the respective *TwinCAT Configuration Files* folder) to the folder with path: *{Hard Drive Label}:\TwinCAT\3.1\Config\Io\EtherCAT*.
7. Last, opt for the *Reload Device Descriptions*, as Figure A-5 illustrates. Now TwinCAT should be capable of understanding each slave's identity, PDOs and settings.

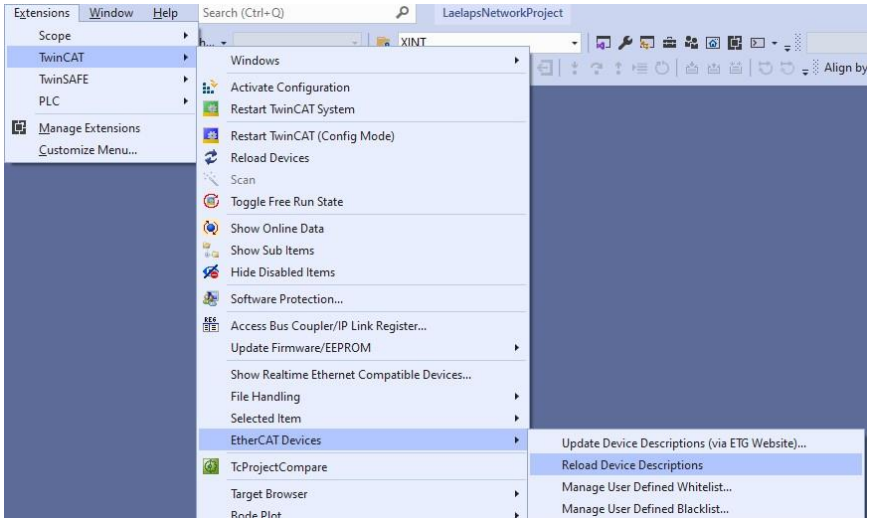


Figure A-5. *Reload Device Descriptions* entry.

Creating the EtherCAT network

In TwinCAT's solution explorer (under I/O) a virtual representation of the physical network should be created. This guide follows the slave order proposed in Chapter 5. To create the virtual network:

8. Configure the EtherCAT master. *Right-Click* on the *Devices* entry in the Project Explorer's window to insert EtherCAT master device, as shown in Figure A-6. Next, choose a suitable name. If a window with the available network adapters pops up, choose the previously installed real-time adapter to be used by the master.

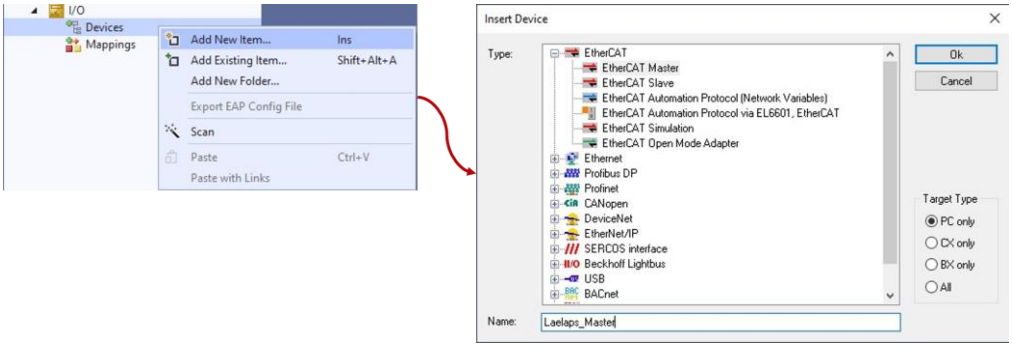


Figure A-6. EtherCAT master configuration procedure.

9. *Right-Click* on the Project Explorer's master entry and select *Add New Item…*

10. On the pop-up window, one may choose the ENI file of the slave that intends to add in the network. This procedure is illustrated in Figure A-7. Repeat this step for every slave on the physical bus. Rename them (two left-clicks on the slave instance) to match the proposed slave order in the network (see Chapter 5, e.g HR_LEG).

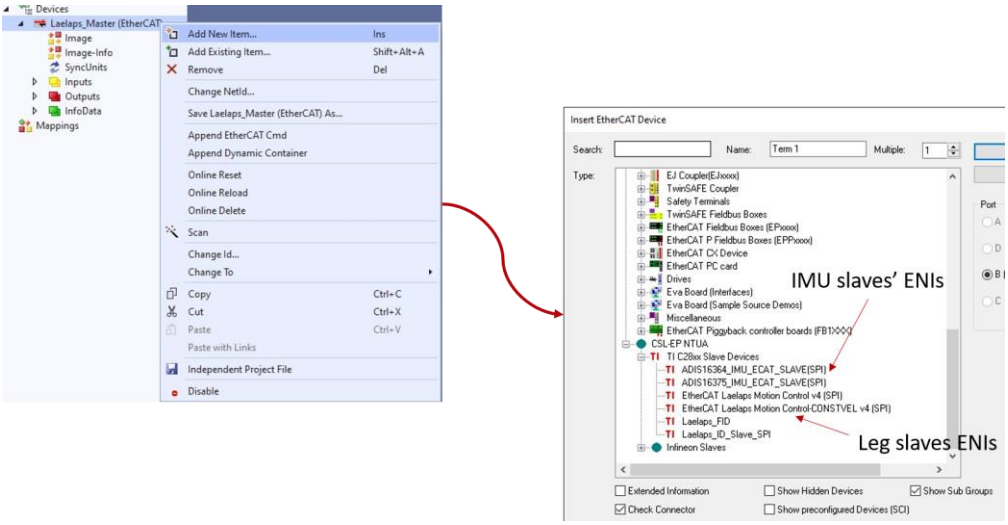


Figure A-7. Add slave configuration procedure.

At this point, the virtual network in the *Solution Explorer* should be like the one illustrated in Figure A-8.

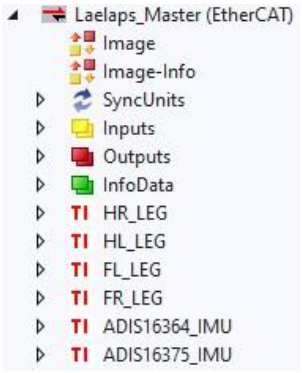


Figure A-8. TwinCAT 3 EtherCAT network.

Create TwinCAT 3 Scope Project and PLC Task

11. Add a TwinCAT Scope View to be able to save all EtherCAT variables during the experiments. Right-click on *Solution's Name -> Add -> New Project* and select Scope YT Project from the TwinCAT Measurement tab as shown in Figure A-9. Name it *Laelaps_Measurements*.

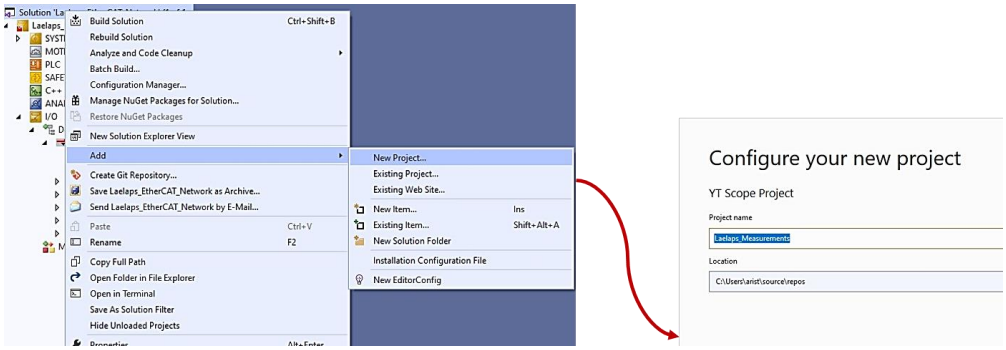


Figure A-9. Measurement project configuration procedure.

12. Add a PLC task to be able to use the Scope View. *Right-Click* on *PLC -> Add New Item -> Standard PLC Project* as indicated in Figure A-10. If no PLC task is created, TwinCAT 3 will not be able to plot and save the values of the desired EtherCAT variables. Name it *Laelaps_PLC_Task*.

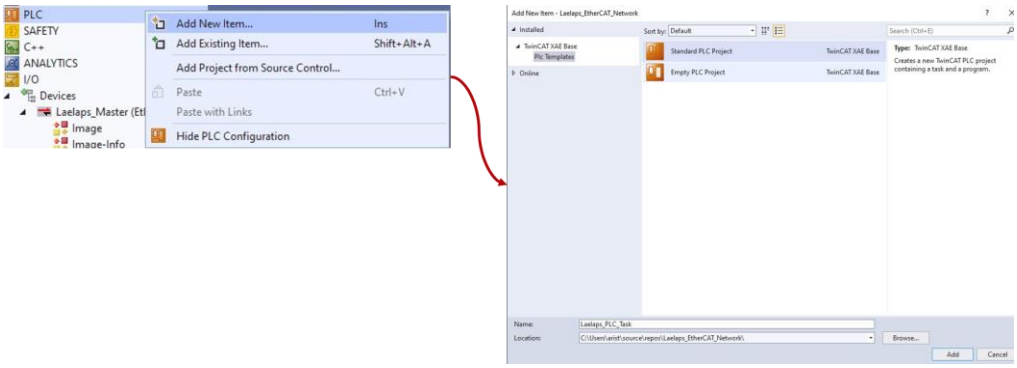


Figure A-10. Add PLC task procedure.

13. Create a Global Variable list of all input variables that are intended to be saved during the experiment. *Right-Click* on the *PLC Task GVL Folder -> Add -> Global Variable List* and name it *Inputs*. This procedure is presented in Figure A-11.

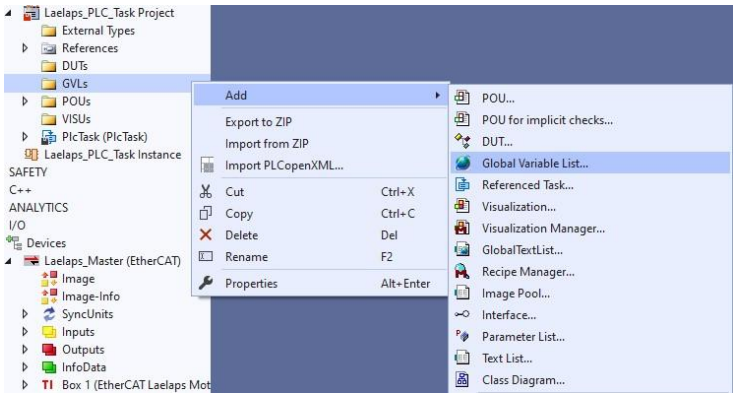


Figure A-11. Add Global Variable List (GVL).

14. Make sure to add all the necessary variables by following the format shown in Figure A-12, where all the required inputs to the master are being scoped and saved, making sure that they have the right variable type. Note that the aforementioned figure's variables refer only to the EtherCAT leg slaves. If the IMUs are going to be used, their variables should be added, too. For the user's convenience the variables for each kind

of the created slaves can be found in [52] [83] [84] (under *TwinCAT Configuration Files* folder).

```
VAR_GLOBAL
    HR_Hip_Angle AT%I*: INT;
    HR_Knee_Angle AT%I*: INT;
    HR_Desired_Hip_Angle AT%I*: INT;
    HR_Desired_Knee_Angle AT%I*: INT;
    HR_PWM_Hip AT%I*: INT;
    HR_PWM_Knee AT%I*: INT;
    HR_Velocity_Knee AT%I*: DINT;
    HR_Velocity_Hip AT%I*: DINT;
    HR_Time AT%I*: UINT;


    HL_Hip_Angle AT%I*: INT;
    HL_Knee_Angle AT%I*: INT;
    HL_Desired_Hip_Angle AT%I*: INT;
    HL_Desired_Knee_Angle AT%I*: INT;
    HL_PWM_Hip AT%I*: INT;
    HL_PWM_Knee AT%I*: INT;
    HL_Velocity_Knee AT%I*: DINT;
    HL_Velocity_Hip AT%I*: DINT;
    HL_Time AT%I*: UINT;


    FR_Hip_Angle AT%I*: INT;
    FR_Knee_Angle AT%I*: INT;
    FR_Desired_Hip_Angle AT%I*: INT;
    FR_Desired_Knee_Angle AT%I*: INT;
    FR_PWM_Hip AT%I*: INT;
    FR_PWM_Knee AT%I*: INT;
    FR_Velocity_Knee AT%I*: DINT;
    FR_Velocity_Hip AT%I*: DINT;
    FR_Time AT%I*: UINT;


    FL_Hip_Angle AT%I*: INT;
    FL_Knee_Angle AT%I*: INT;
    FL_Desired_Hip_Angle AT%I*: INT;
    FL_Desired_Knee_Angle AT%I*: INT;
    FL_PWM_Hip AT%I*: INT;
    FL_PWM_Knee AT%I*: INT;
    FL_Velocity_Knee AT%I*: DINT;
    FL_Velocity_Hip AT%I*: DINT;
    FL_Time AT%I*: UINT;

END_VAR
```

Figure A-12. Global Variable List Overview.

15. In the *POUS -> MAIN (PRG)*, create a list of all variables that are necessary to be handled simultaneously in all slaves. This process **MUST** be done at least for the State_Machine variables so that the clocks in all four slaves are initiated at the exact same time. An example list for the legs' slaves is shown in Figure A-13.

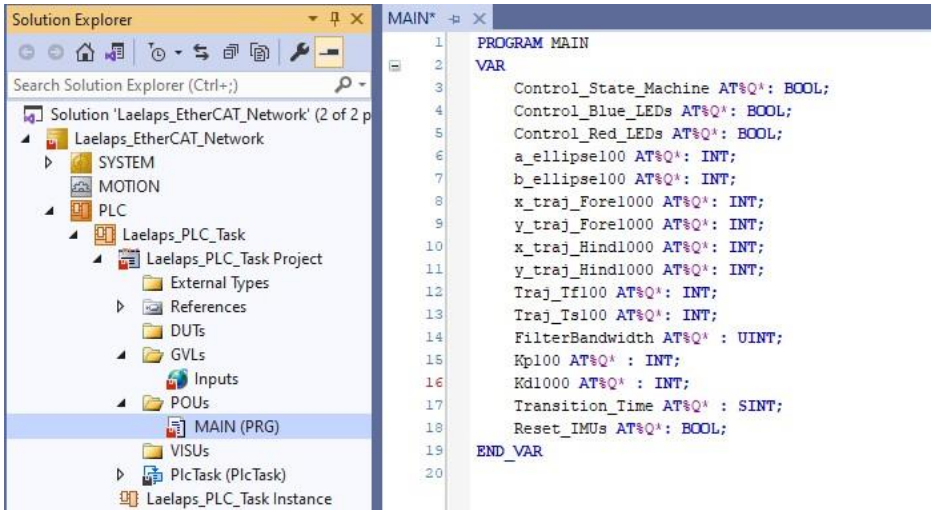


Figure A-13. Main variable list.

16. Build the solution (Figure A-14) and expand the PLC Task's Instance to inspect the created variables.

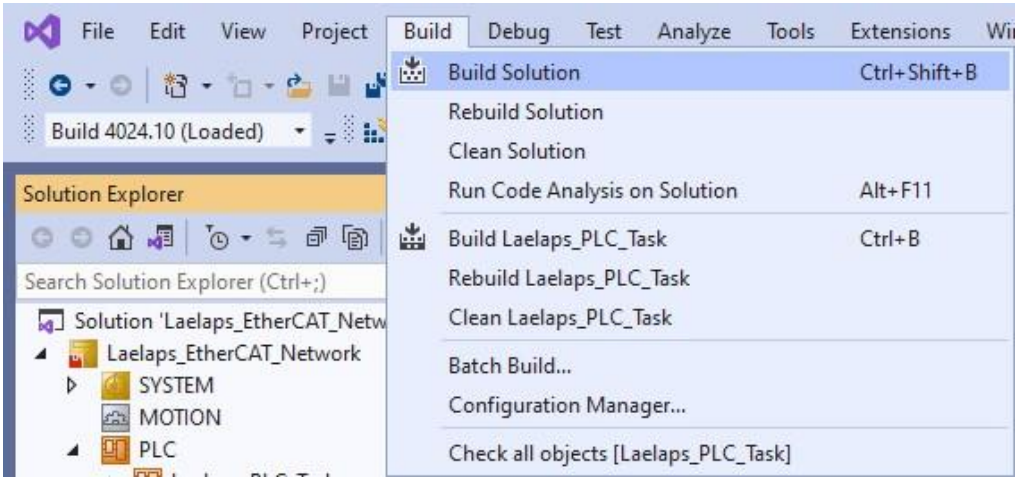


Figure A-14. Build Solution option.

17. Link all Input and Output variables of the list to the respective EtherCAT variables by clicking twice on each PLC variable, selecting *Linked to…* and choosing the desired from the list of all compatible variables (as far as the type is concerned), as shown in Figure A-15. Note that in order to link one PLC output variable to multiple EtherCAT output variables, select all desired EtherCAT variables from the pop-up window holding the *Ctrl* button. Now, all linked output variables of the project are handled by the PlcTask Outputs and Online Writes can only be executed through this list. Do not forget to link the State Machine PLC output variable with **ALL** State_Machine EtherCAT variables of all slaves to accomplish synchronous initiation of the trajectories' time variables.

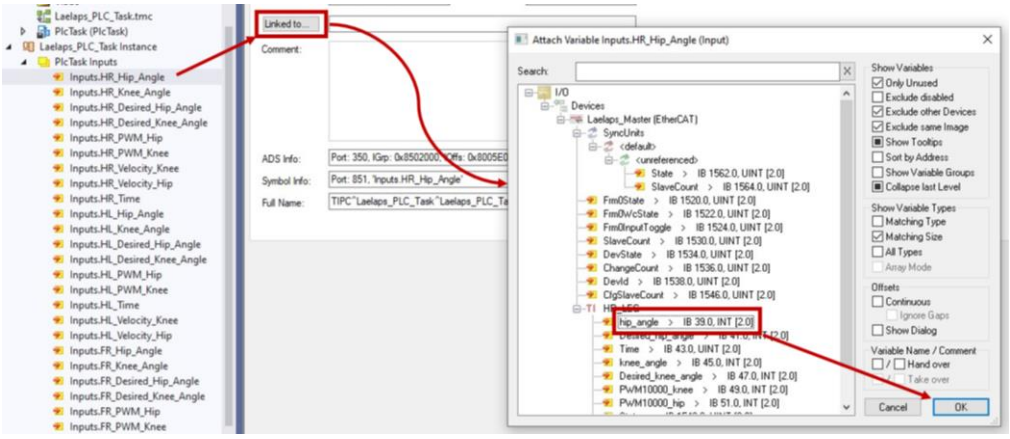


Figure A-15. Link PLC variables.

18. *Activate Configuration* (Figure 5-3) to Restart TwinCAT System and update the project with the linked variables. If a warning about run-time licenses appears, just copy the displayed code to the designated text-box
19. Login and Start the PLC task as indicated in Figure A-16.

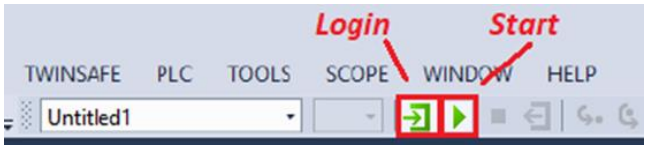


Figure A-16. PLC Login and Start buttons.

20. Navigate to the Measurement Project inside the Solution Explorer and right-click on *Axis -> Target Browser* and select all the desired variables from the Global Variable List (Inputs) that have to be monitored and saved during the experiment (Figure A-17).

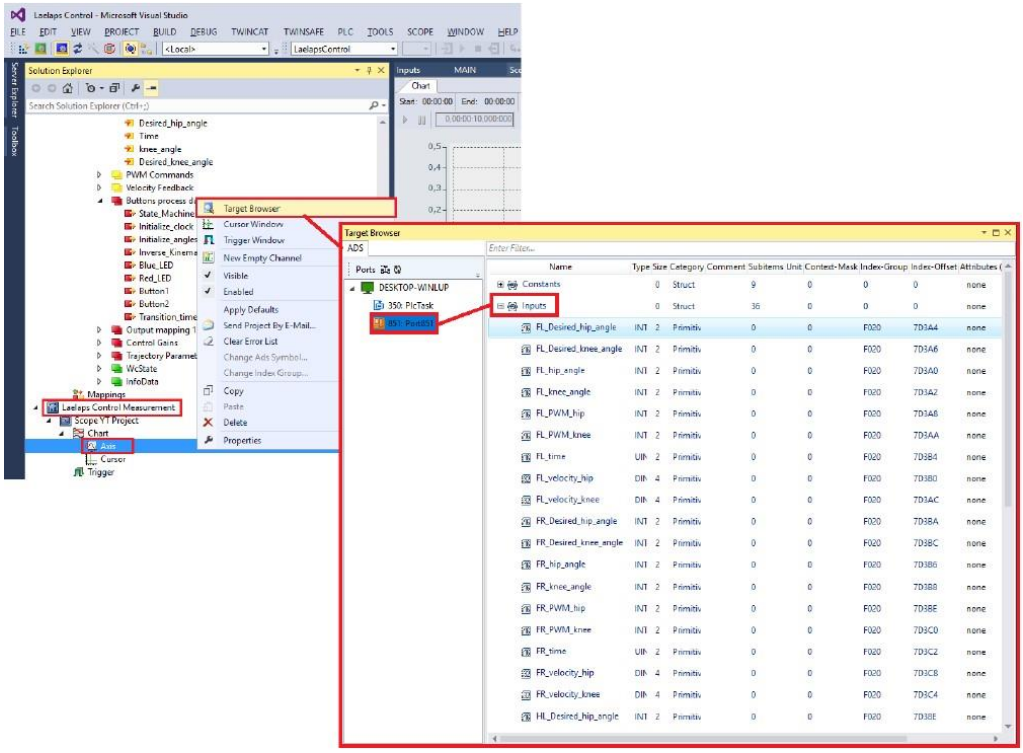


Figure A-17. Create GVL inputs' graphs.

If these steps have been followed together with the steps of the typical experiment's guide of Chapter 5, press the TwinCAT *Config* button, located in the menu taskbar (Figure 5-3). This is necessary in order to execute the upcoming steps of the aforementioned guide.

Appendix B. Create an EtherCAT Application from Scratch

In this section, the steps of creating from scratch an arbitrary EtherCAT application are presented. This guide is dedicated to the ADIS16364 EtherCAT app, to illustrate the process of developing it, but with a few changes, it can become generic. The software required to generate all the necessary files include the ETG's Slave Stack Code Tool (SSC) [38] and Microsoft's Excel.

1. Open the SSC Tool and create a new project by selecting the appropriate ESI file from the drop-down list, like the one presented in Figure B-1.

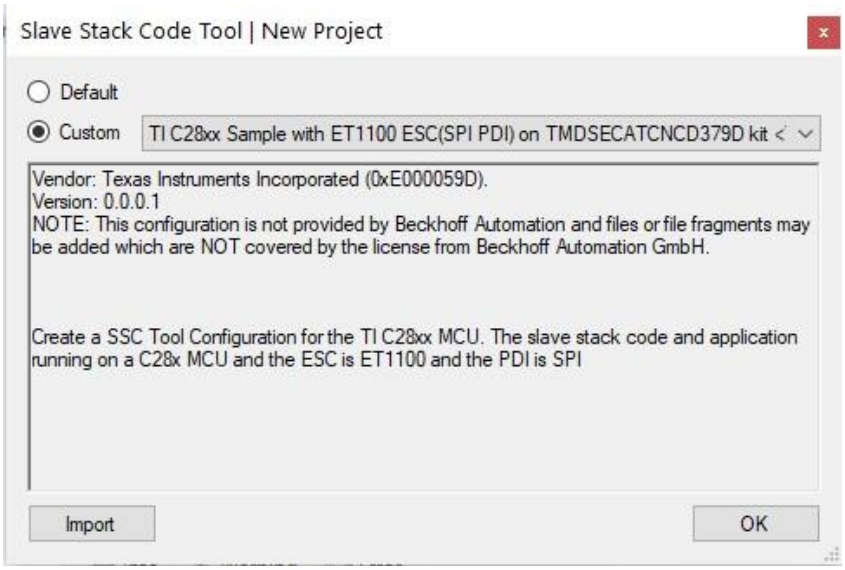


Figure B-1. SSC new project pane.

2. Navigate to the *EtherCAT Slave -> SlaveInformation* tab and change the device name to a desired one. In this case, as Figure B-2 suggests, **ADIS16364_IMU_SLAVE** is used.

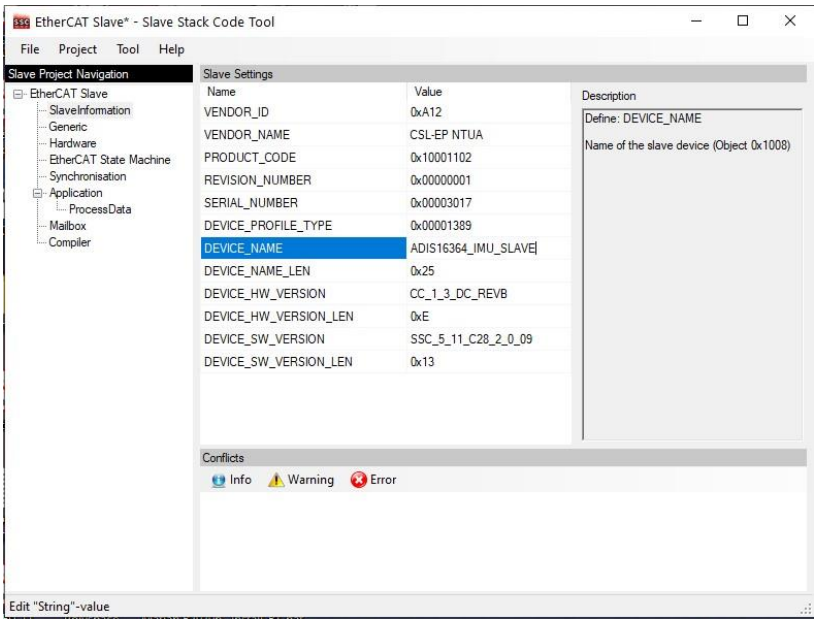


Figure B-2. SSC slave information tab.

3. Create the new EtherCAT application by setting the desired Process Data in the appropriate fields, like Figure B-3 shows. Details on how to define the PDOs may be found in the literature [33] [37].

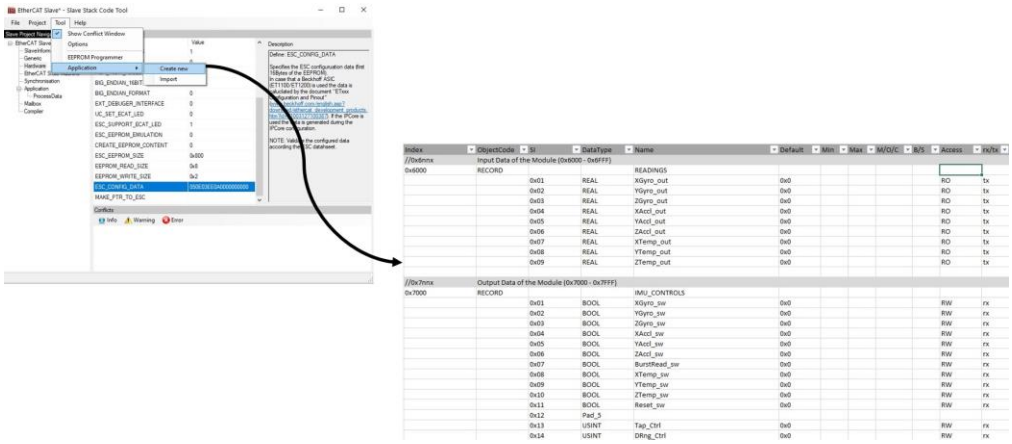


Figure B-3. SSC create new application overview.

4. Change the minimum PD cycle. This is the minimum allowable EtherCAT loop period and is dependent on the application and its execution time. If that information is unknown at this stage of development, leave it to the default (Figure B-4) and change it afterward, by modifying the EtherCAT stack's ecat_def.h header file.

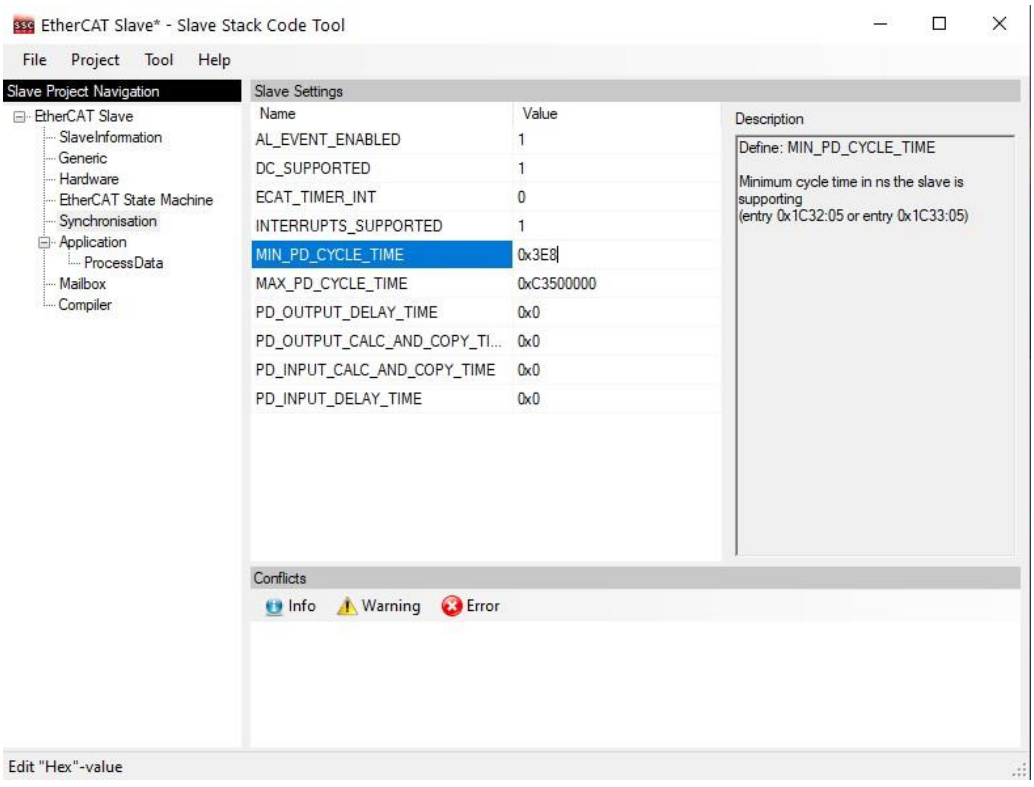


Figure B-4. SSC synchronization tab.

5. At last, navigate to *Project -> Create New Slave Files* to create the main EtherCAT stack and the corresponding ENI file, as Figure B-5 suggests.

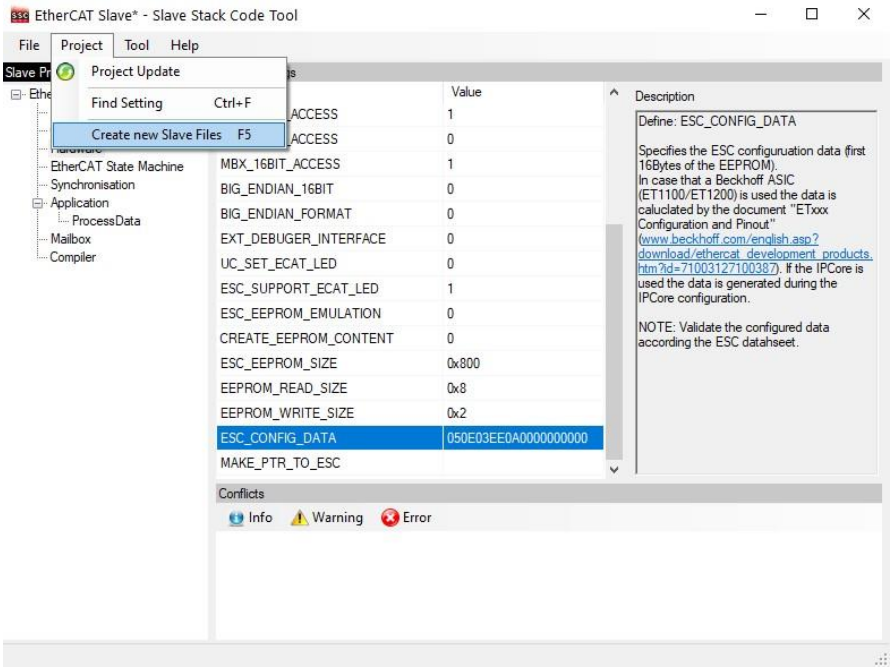


Figure B-5. SSC create new slave files option.

6. Copy and Paste the generated files in an appropriate location, inside the project files. Also, the ENI file should be imported to the master, to configure the network.

Appendix C. C2000 Delfino Microcontroller Unit

The LaunchXL-F28379D evaluation board (Figure C-1) materializes the Data Link Layer (DLL) of the EtherCAT slave model, introduced in Section 2.2.4. It is the hardware in which, the user-application is implemented. In the initial stages of any bare-metal design, the hardware requirements of the application should be considered. In the current case, the DLL's hardware has to serve highly deterministic, real-time control applications. The complexity of the underlying procedures, along with the increased number of sensors that are involved in the designs, creates the need of a powerful microcontroller, with provisions for possible future demands.

This Appendix investigates the various resources that the LaunchXL-F28379D offers. It provides a generic view of the major peripherals used in the developed firmware, described in Chapters 3 and 4. It is highly recommended to comprehend the discussed concepts, to be able to get a thorough understanding of the developed applications. The aim here is not to replicate the extensive documentation that this board comes with [64], but to focus on the key aspects that make it ideal for Laelaps II. To sum up, the major aspects of the aforementioned board are analyzed briefly, in a practical way with examples that facilitate the learning process.

C.1 LaunchXL-F28379D Development Board

The LaunchXL-F28379D (Figure C-1) is an evaluation package of the TMS320F28379D microcontroller, which belongs to the family of TI's Delfino C2000 MCUs [111]. It is exceptional when hard real-time processing is required and offers a variety of different peripherals that make designing sophisticated applications both low-cost and fast. The package comes with the following modules:

- 16 HRPWM outputs with 150 ps edge control
- 6 capture inputs
- 3 eQEP inputs
- 8 SDFM input channels
- 4 ADCs with selectable resolution
- 8 windowed comparators with 12-bit DAC references
- On-board isolated XDS100v2 debug probe
- Support for the most common communication protocols, like SPI, I^2C, CAN, etc.

The LaunchPad does not expose all of the microcontroller's GPIOs, but for the Laelaps II applications' needs, they suffice.

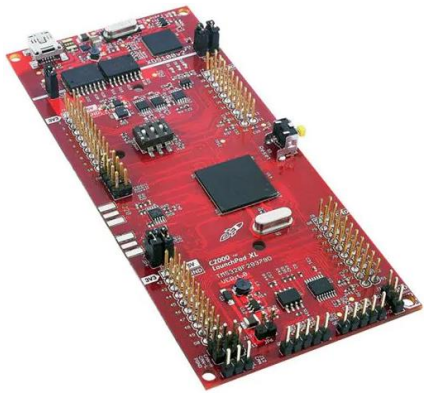


Figure C-1. The LaunchXL-F28379D.

C.2 TMS320F28379D Microcontroller

The 32-bit TMS320F28379D dual-core microcontroller supports parallel processing schemes, with increased RAM and FLASH storages. Each 32-bit core is enhanced with a Control Law Accelerator Unit (CLA), a separate processor with independent buses. Among other features, the Direct Memory Allocation (DMA) controller stands out, taking over data transfers among memory places and among peripherals, without withholding CPU bandwidth. Moreover, the existence of the Floating Point Unit (FPU), the Trigonometric (TMU) and the Viterbi (VCU) accelerators enable precision mathematical operations, which even include complex math, without disrupting high frequency, real-time tasks. There is extensive documentation one may address to get a deeper insight into Delfino's architecture [64] [112]. This work focuses on the features used in the implemented applications. Special focus is given in the programming of the CLA for concurrent task execution aiming at freeing the main CPU from several motion control and planning tasks. An overview of each core's architecture is illustrated in Figure C-2.

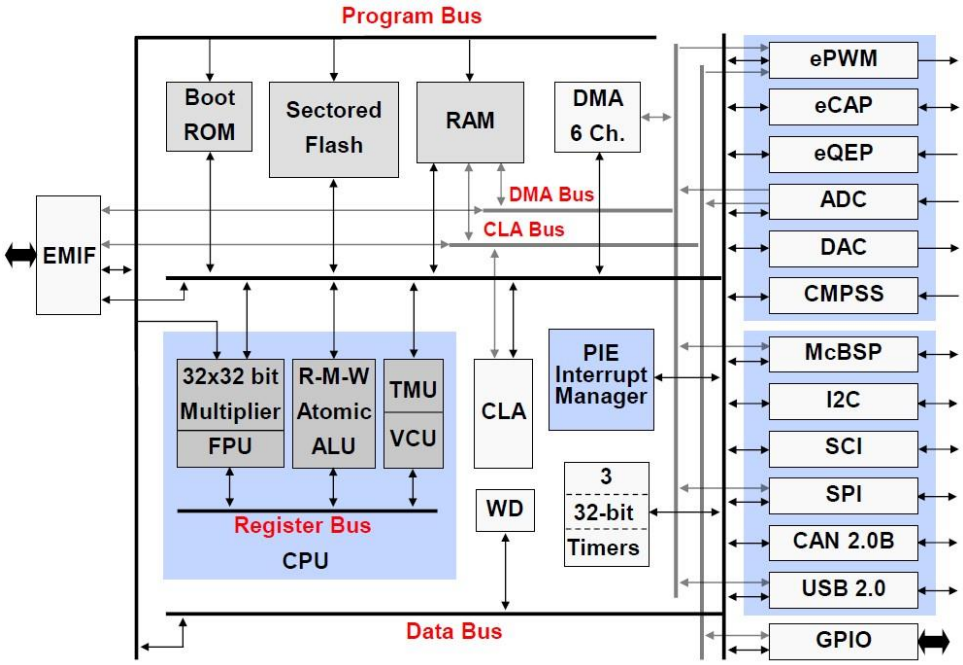


Figure C-2. TMS320F28379D microcontroller core architecture.

C.3 Interrupt Architecture

An interrupt is a signal that causes the CPU to pause its current execution and branch to a different piece of code known as an interrupt service routine (ISR). This is a useful mechanism for handling peripheral events, and involves less CPU overhead or program complexity than register polling. However, because interrupts are asynchronous to the program flow, care must be taken to avoid conflicts over resources that are accessed both in interrupts and in the main program code.

Interrupts propagate to the CPU through a series of flag and enable registers. The flag registers store the interrupt until it is processed. The enable registers block the propagation of the interrupt. When an interrupt signal reaches the CPU, the CPU fetches the appropriate ISR address from a list called the vector table.

The C28x CPU has fourteen peripheral interrupt lines. Two of them (INT13 and INT14) are connected directly to CPU timers 1 and 2, respectively. The remaining twelve are

connected to peripheral interrupt signals through the enhanced Peripheral Interrupt Expansion module (PIE). The PIE multiplexes up to sixteen peripheral interrupts into each CPU interrupt line. It also expands the vector table to allow each interrupt to have its own ISR. This allows the CPU to support a large number of peripherals. The described architecture is illustrated in Figure C-3.

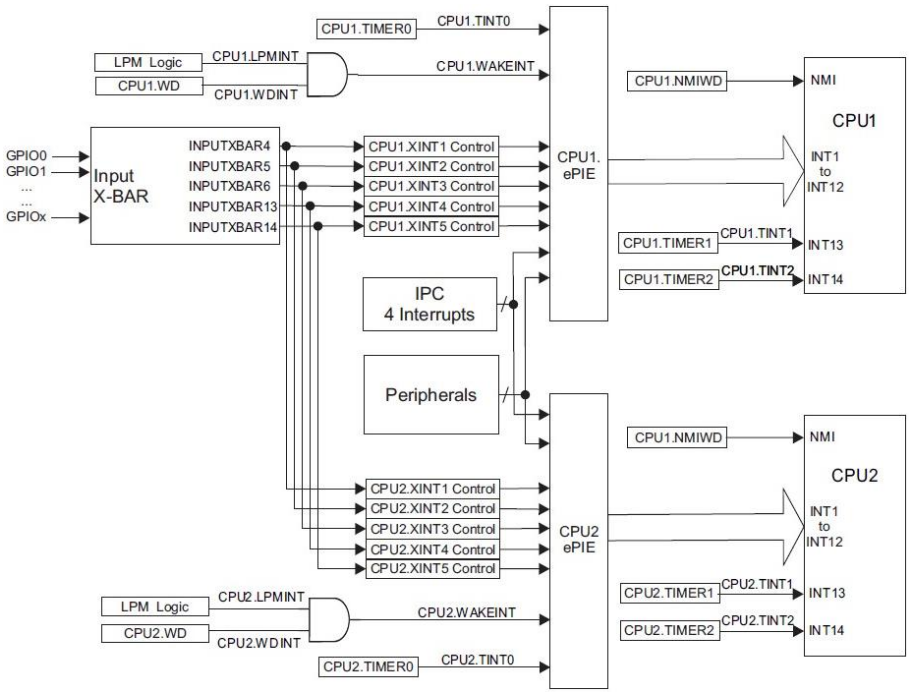


Figure C-3. Delfino's interrupt architecture.

An interrupt path is divided into three stages, namely the peripheral, the PIE, and the CPU. Each stage has its own enable and flag registers. This system allows the CPU to handle one interrupt while others are pending, implement and prioritize nested interrupts in software, and disable interrupts during certain critical tasks.

Each peripheral has its own unique interrupt configuration, which is described in that peripheral's chapter. Some peripherals allow multiple events to trigger the same interrupt signal. For example, a communications peripheral might use the same interrupt to indicate that data has been received or that there has been a transmission error. The cause of the interrupt can be determined by reading the peripheral's status register. Often, the bits in the status register must be cleared manually before another interrupt will be generated.

The PIE provides individual flag and enable register bits for each of the peripheral interrupt signals, which are sometimes called PIE channels. These channels are grouped according to their associated CPU interrupt. Each PIE group has one 16-bit enable register (PIEIERx), one 16-bit flag register (PIEIFRx), and one bit in the PIE acknowledge register (PIEACK). The PIEACK register bit acts as a common interrupt mask for the entire PIE group. When the CPU receives an interrupt, it fetches the address of the ISR from the PIE. The PIE returns the vector for the lowest-numbered channel in the group that is both flagged and enabled. This gives lower-numbered interrupts a higher priority when multiple interrupts are pending. If no interrupt is both flagged and enabled, the PIE returns the vector for channel 1. This condition will not happen unless software changes the state of the PIE while an interrupt is propagating.

Like the PIE, the CPU provides flag and enable register bits for each of its interrupts. There is one enable-register (IER) and one flag-register (IFR), both of which are internal CPU

registers. There is also a global interrupt mask, which is controlled by the INTM bit in the ST1 register. This mask can be set and cleared using the CPU's SETC instruction. Writes to IER and INTM are atomic operations. In particular, if INTM is cleared, the next instruction in the pipeline will run with interrupts disabled. No software delays are needed.

Each CPU has its own PIE. Both PIEs must be configured independently. Some interrupts come from shared peripherals that can be owned by either CPU, such as the ADCs and SPIs. These interrupts are sent to both PIEs regardless of the peripheral's ownership. Thus, a peripheral owned by one CPU can cause an interrupt on the other CPU if that interrupt is enabled in the other CPU's PIE. Writes to IER and INTM are atomic operations. In particular, if INTM is cleared, the next instruction in the pipeline will run with interrupts disabled. No software delays are needed. The described path is illustrated in Figure C-4.

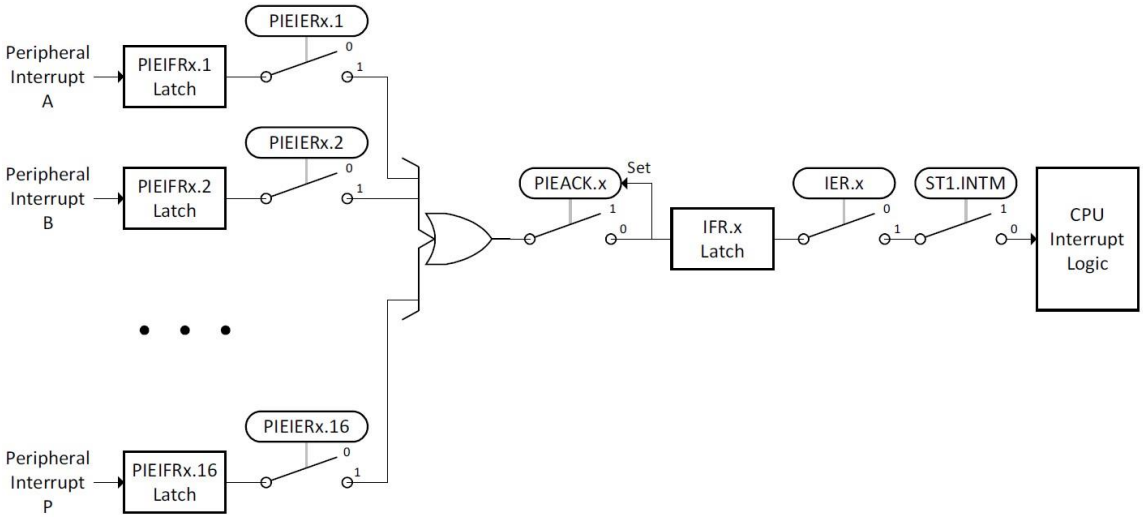


Figure C-4. Interrupt propagation path.

C.3.1 Interrupt Priorities

Interrupts are automatically prioritized by the C28x hardware. Group 1, which corresponds to CPU INT1, has the highest priority. Within each group there are up to 16 interrupts with INTx.1 being the highest priority and INTx.8 having the lowest. The core priorities of the total 14 interrupt lines are listed in Table C-1. A list of all interrupts organized in the PIE is illustrated in Figure C-5.

Table C-1. Interrupt core priorities.

Name	Description	Priority
Reset	Resets the Device	**1 (Highest)**
INT1	PIE Group 1	**5**
INT2	PIE Group 2	**6**
INT3	PIE Group 3	**7**
…	**…**	**…**
INT10	PIE Group 10	**14**
INT11	PIE Group 11	**15**
INT12	PIE Group 12	**16**
INT13	Timer 1	**17**
INT14	Timer 2	**18**
DATALOG	CPU data logging interrupt	**19 (lowest)**

	INTx.1	INTx.2	INTx.3	INTx.4	INTx.5	INTx.6	INTx.7	INTx.8	INTx.9	INTx.10	INTx.11	INTx.12	INTx.13	INTx.14	INTx.15	INTx.16
INT1.y	ADCA1	ADCB1	ADCC1	XINT1	XINT2	ADCD1	TIMER0	WAKE	-	-	-	-	IPC0	IPC1	IPC2	IPC3
INT2.y	EPWM1_TZ	EPWM2_TZ	EPWM3_TZ	EPWM4_TZ	EPWM5_TZ	EPWM6_TZ	EPWM7_TZ	EPWM8_TZ	EPWM9_TZ	EPWM10_TZ	EPWM11_TZ	EPWM12_TZ	-	-	-	-
INT3.y	EPWM1	EPWM2	EPWM3	EPWM4	EPWM5	EPWM6	EPWM7	EPWM8	EPWM9	EPWM10	EPWM11	EPWM12	-	-	-	-
INT4.y	ECAP1	ECAP2	ECAP3	ECAP4	ECAP5	ECAP6	-	-	-	-	-	-	-	-	-	-
INT5.y	EQEP1	EQEP2	EQEP3	-	-	-	-	-	SD1	SD2	-	-	-	-	-	-
INT6.y	SPIA_RX	SPIA_TX	SPIB_RX	SPIB_TX	MCBSPA_RX	MCBSPA_TX	MCBSPB_RX	MCBSPB_TX	SPIC_RX	SPIC_TX	-	-	-	-	-	-
INT7.y	DMA_CH1	DMA_CH2	DMA_CH3	DMA_CH4	DMA_CH5	DMA_CH6	-	-	-	-	-	-	-	-	-	-
INT8.y	I2CA	I2CA_FIFO	I2CB	I2CB_FIFO	SCIC_RX	SCIC_TX	SCID_RX	SCID_TX	-	-	-	-	-	-	UPPA (CPU1 only)	-
INT9.y	SCIA_RX	SCIA_TX	SCIB_RX	SCIB_TX	CANA_0	CANA_1	CANB_0	CANB_1	-	-	-	-	-	-	USBA (CPU1 only)	-
INT10.y	ADCA_EVT	ADCA2	ADCA3	ADCA4	ADCB_EVT	ADCB2	ADCB3	ADCB4	ADCC_EVT	ADCC2	ADCC3	ADCC4	ADCD_EVT	ADCD2	ADCD3	ADCD4
INT11.y	CLA1_1	CLA1_2	CLA1_3	CLA1_4	CLA1_5	CLA1_6	CLA1_7	CLA1_8	-	-	-	-	-	-	-	-
INT12.y	XINT3	XINT4	XINT5	-	-	VCU	FPU_OVERFLOW	FPU_UNDERFLOW	EMIF_ERROR	RAM_CORRECTABLE_ERROR	FLASH_CORRECTABLE_ERROR	RAM_ACCESS_VIOLATION	SYS_PLL_SLIP	AUX_PLL_SLIP	CLA OVERFLOW	CLA UNDERFLOW
Note: Cells marked "-" are Reserved																

Figure C-5. PIE channel mapping.

C.3.2 Interrupt Nesting

Generally, the C2000 does not support interrupt nesting. This means that whenever an ISR is triggered, it runs through completion, even if another one with higher priority is triggered during its servicing. Nevertheless, there is a software way to bypass this [113]. There are limitations, but the technique is generic and can be implemented in different applications. The firmware developed in the current thesis does not have such capabilities. However, it would be interesting to explore the capabilties that nesting has to offer and is left for future work. Concisely, by default interrupt nesting is not permitted in C2000 architecture and subsequently, the previously-discussed priorities apply only if two arbitrary interrupts are triggered at the same CPU cycle, a relatively rare event. Nevertheless, great attention should be given at the design stage of an application since data access conflicts could lead to heavily erroneous software.

C.4 Control Law Accelerator

The CLA unit is a Task Driven Machine (TDM), in contrast with the main CPU that is an Interrupt Driven Machine (IDM). To clarify, tasks constitute the main routines, which the CLA may be called to service, one at a time, similar to the Interrupt Service Routines. This attribute transfuses a fully deterministic operation to the CLA, in the sense that every task that is being executed at a time, finishes, before executing another one. Task requests are serviced in a priority manner dictated by their number in ascending order. The high-performance arithmetic capabilities and the direct access to most of the peripherals make this accelerator a powerful addition to the Laelaps II motion control arsenal.

C.4.1 Task Mechanism

There are eight tasks and each one's start address is stored in a special register, called **MVECT**, while its number states its priority. There are various, not only software but also hardware, triggers that initialize a task's execution and will be analyzed later on. Upon completion, an optional task-specific interrupt is flagged within the Peripheral Interrupt Expansion (PIE) module [64] to inform the main CPU of the event. The tasks may be written in modified C or Assembly for best performance. The CLA is specifically designed for 32-bit data handling, with the support of 16-bit computations and peripheral register accesses. There is no software or hardware support for 64-bit data types. In Table C-2 the different data type sizes are illustrated, in comparison with the respective CPU ones.

Table C-2. CPU vs CLA data types.

Type	CPU & FPU (COFF)	CLA
char	16 bit	16 bit
short	16 bit	16 bit
int	16 bit	32 bit
long	32 bit	32 bit
long long	64 bit	32 bit
float	32 bit	32 bit
double	32 bit	32 bit
Long double	64 bit	32 bit
pointers	32 bit	16 bit

C.4.2 Memory and Peripherals Access

There are independent *LSx* RAM blocks that may be mapped to the accelerator and be used as Data or Program Memory. Also, the CPU-CLA communication is accomplished through two dedicated message RAMs. Moreover, by specifying the appropriate CPU registers, direct peripheral access may be granted for CLA or DMA, as a secondary master, but not for both simultaneously. An overview of the described architecture is illustrated in Figure C-6.

There are two standard memory spaces a CPU may have at its disposal to execute code, store data and handle peripherals [114]. The Program memory contains program code for execution, while its Data counterpart contains all vital variables and coefficients that are used during program run-time. As previously mentioned, two configurable RAM sections are dedicated to the bidirectional communications of CPU and CLA. It is very important to understand that all available memory is mapped to the C28x CPU at reset and subsequently to the CLA, by CPU commands and suitable register parametrizations.

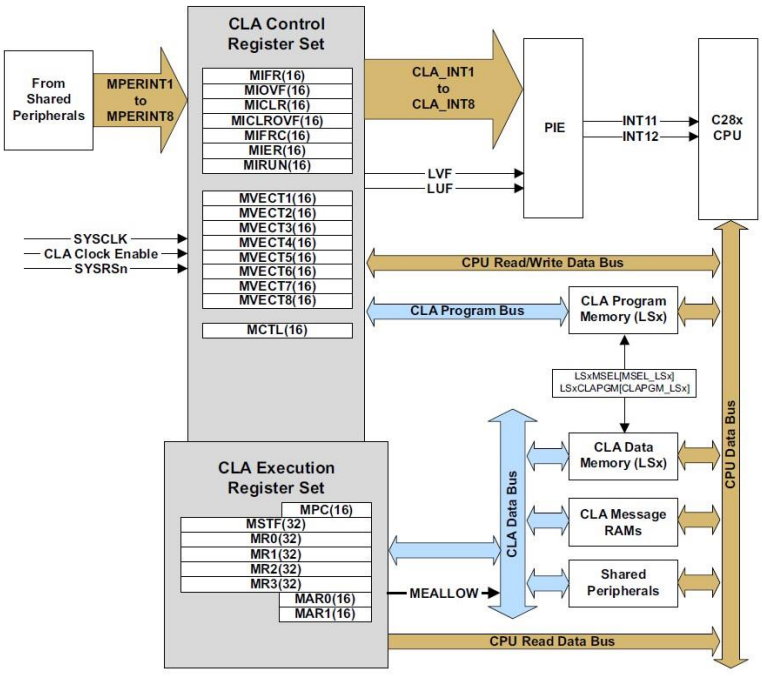


Figure C-6. CLA unit architecture.

CLA Program Memory

The C28x Master CPU may grant CLA specific RAM blocks by appropriately setting 1 to **MemCfgRegs.LSxMSEL [MSEL_LSx]** bits and subsequently specifying the type of the aforementioned memory blocks as Program by setting **MemCfgRegs.LSxCLAPGM [CLAPGM_LSx]** bits to High state.

CLA Data Memory

As in Program Memory's case, the CLA's Data Memory may be specified accordingly. The difference lies in **MemCfgRegs.LSxCLAPGM [CLAPGM_LSx]** bits that in this case should be set to zero. To assign memory blocks, it is vital to configure the respective linker command file (.cmd), which is responsible to make an efficient allocation of the various resources available on the device. This matter is discussed in Section C.7, because it concerns system-wide functionalities, not only the CLA's.

CLA Message RAMs

The CLA communicates with the main CPU through Message RAMs. There are different types of access that those modules offer for a variable stored in them. In Table C-3, an overview of the aforementioned data sections is presented. To store a variable to one of these sections, specific **#pragma** commands can be used just before the variable's declaration. An example of such process follows.

```
//
// CPU & CLA R/W Access
//
#pragma DATA_SECTION(common_var,"ClaDataRam");
uint32_t common_var = 0;

//
// CLA R/W & CPU R Access
//
#pragma DATA_SECTION(cla_var, "Cla1ToCpuMsgRAM");
int16_t cla_var = 0;

//
// CPU R/W & CLA R Access
//
#pragma DATA_SECTION(cpu_var, "CpuToCla1MsgRAM");
float cpu_var = 0.0f;
```

Table C-3. CPU-CLA Message RAMs overview.

Data Section	CPU Access	CLA Access
ClaDataRam	R/W	R/W
CpuToCla1MsgRAM	R/W	R
Cla1ToCpuMsgRAM	R	R/W

C.4.3 CLA Initialization

The CLA initiliazation process assigns the allocated memory sections defined in the linker *.cmd* file (Section C.7) as program or data memory. It is of outmost importance to allocate efficiently the available memory to avoid segmentation faults and other erroneous behaviors, such as stack overflow. Next, the task triggers are configured. Each task can be triggered by another peripheral or by software. For example, in case of Laelaps II motion control, a software trigger is used for the two tasks that are used by the firmware. Finally, the CLA post-processing

interrupts are configured and activated, if necessary. These interrupts are triggered after the completion of each task to inform the CPU for the event.

```
void InitCla(void)
{
    //
    // Enable EALLOW Register Access
    //
    EALLOW;

    //
    // Initialize and Wait for CLA1ToCPUMsgRAM
    //
    MemCfgRegs.MSGxINIT.bit.INIT_CLA1TOCPU = 1;
    while (MemCfgRegs.MSGxINITDONE.bit.INITDONE_CLA1TOCPU != 1)
    {
        ;
    }

    //
    // Initialize and wait for CPUToCLA1MsgRAM
    //
    MemCfgRegs.MSGxINIT.bit.INIT_CPUTOCLA1 = 1;
    while (MemCfgRegs.MSGxINITDONE.bit.INITDONE_CPUTOCLA1 != 1)
    {
        ;
    }

    //
    // Set CLA LS0 to LS5 RAM Access Privileges
    //
    MemCfgRegs.LSxMSEL.bit.MSEL_LS0 = 1;
    MemCfgRegs.LSxMSEL.bit.MSEL_LS1 = 1;
    MemCfgRegs.LSxMSEL.bit.MSEL_LS2 = 1;
    MemCfgRegs.LSxMSEL.bit.MSEL_LS3 = 1;
    MemCfgRegs.LSxMSEL.bit.MSEL_LS4 = 1;
    MemCfgRegs.LSxMSEL.bit.MSEL_LS5 = 1;

    //
    // Configure CLA RAM Blocks (0: Data Memory, 1: Program Memory)
    //
    MemCfgRegs.LSxCLAPGM.bit.CLAPGM_LS0 = 0;
    MemCfgRegs.LSxCLAPGM.bit.CLAPGM_LS1 = 0;
    MemCfgRegs.LSxCLAPGM.bit.CLAPGM_LS2 = 0;
    MemCfgRegs.LSxCLAPGM.bit.CLAPGM_LS3 = 1;
    MemCfgRegs.LSxCLAPGM.bit.CLAPGM_LS4 = 1;
    MemCfgRegs.LSxCLAPGM.bit.CLAPGM_LS5 = 0;

    //
    // Initialize CLA Task Interrupt Vectors
    // -- TI's Default way SA (Reviewed)
    //
    Cla1Regs.MVECT1 = (uint16_t)(&Cla1Task1);
    Cla1Regs.MVECT2 = (uint16_t)(&Cla1Task2);
    Cla1Regs.MVECT3 = (uint16_t)(&Cla1Task3);
    Cla1Regs.MVECT4 = (uint16_t)(&Cla1Task4);
    Cla1Regs.MVECT5 = (uint16_t)(&Cla1Task5);
    Cla1Regs.MVECT6 = (uint16_t)(&Cla1Task6);
    Cla1Regs.MVECT7 = (uint16_t)(&Cla1Task7);
    Cla1Regs.MVECT8 = (uint16_t)(&Cla1Task8);

    //
    // Set Software Trigger for Task Interrupts
    //
    DmaClaSrcSelRegs.CLA1TASKSRCSEL1.bit.TASK1 = 0;
    DmaClaSrcSelRegs.CLA1TASKSRCSEL1.bit.TASK2 = 0;
    DmaClaSrcSelRegs.CLA1TASKSRCSEL1.bit.TASK3 = 0;
    DmaClaSrcSelRegs.CLA1TASKSRCSEL1.bit.TASK4 = 0;
    DmaClaSrcSelRegs.CLA1TASKSRCSEL2.bit.TASK5 = 0;
    DmaClaSrcSelRegs.CLA1TASKSRCSEL2.bit.TASK6 = 0;
    DmaClaSrcSelRegs.CLA1TASKSRCSEL2.bit.TASK7 = 0;
    DmaClaSrcSelRegs.CLA1TASKSRCSEL2.bit.TASK8 = 0;
```

```
    //
    // Do not Lock Task Control Registers
    //
    DmaClaSrcSelRegs.CLA1TASKSRCSELLOCK.bit.CLA1TASKSRCSEL1 = 0;
    DmaClaSrcSelRegs.CLA1TASKSRCSELLOCK.bit.CLA1TASKSRCSEL2 = 0;

    //
    // Enable use software to start a task (IACK)
    //
    Cla1Regs.MCTL.bit.IACKE = 1;

    //
    // Enable CLA's Task 1 & 2
    //
    Cla1Regs.MIER.all = 0x0003;

    //
    // Disable EALLOW Register Access
    //
    EDIS;

}   // End Of InitCla()
```

C.4.4 CLA Math Library

The basic aim of the Control Law Accelerator is to take over and speed up complex mathematical procedures to relieve the CPU. Towards this, TI introduces CLA Math Library, which consists of a collection of highly optimized mathematical functions, with low CLA cycle run-times and easy code integration. Their contextual logic is founded upon look-up tables for trigonometric, logarithmic and exponential operations. These tables are already built-in but require an effort to use. The library's manual is very informative and the reader is strongly encouraged to consult it [115]. The basic workflow of adding the CLA Math library's capabilities to an application is laid in Appendix D.

C.5 Direct Memory Access Controller

C.5.1 Overview

Typically, memory is accessed in a Programmed Input/Output (PIO) manner, which simply means that memory reads and writes are managed by the CPU executing a piece of code. This is a straight-forward approach with the use of virtual addresses and no concerns of low-level processes by the user. However, the flip side is the increased CPU overhead that this logic results in. Also, the main performance penalty is that if one wants to write X times in memory, the write commands would be also X, since burst operations are not applicable [116].

Direct Memory Access Controller (DMA) comes to facilitate high throughput data transfers, replacing CPU with an efficient hardware solution. The Delfino's DMA supports data transfers and handling of peripherals. It should be perceived as a controller placed on the memory's bus that handles tedious transfers and offloads the CPU bandwidth. Its basic functionality consists of a software or hardware trigger that initiates the transfers and an interrupt triggered after each task's completion to inform the CPU of the event. The whole system's structure is presented in Figure C-7.

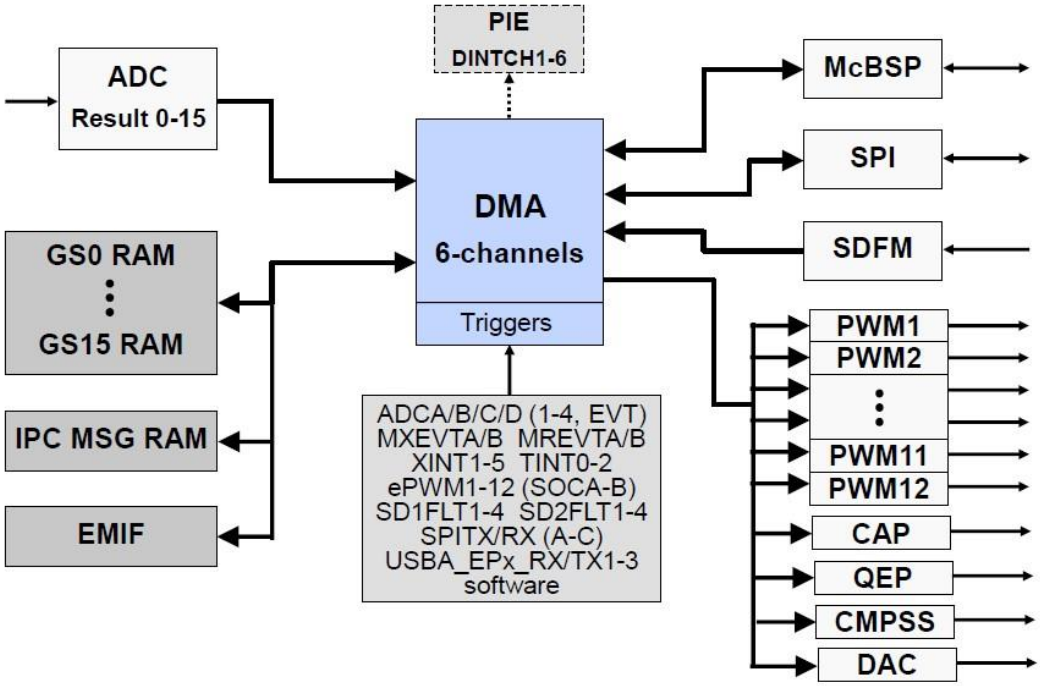


Figure C-7. DMA controller architecture.

C.5.2 DMA Setup

In the case of Delfino, each CPU is provided with a six-channel DMA controller. The necessary steps for its operation are:

1. Reset the entire DMA module and set an intended emulation halt response, by setting **DmaRegs.DMACTRL.bit.HARDRESET** to 1 and **DmaRegs.DEBUGCTRL.bit.FREE** to the desired value (0 =Unaffected/1=Affected by emulation halt).
2. Set channel 1 priority option if necessary, by modifying the **DmaRegs.PRIORITYCTRL1.bit.CH1PRIORITY** register bit.
3. Change mode of the channels that are planned to be used, by editing the **DmaRegs.CH1.MODE.all** register, accordingly [64].
4. Determine the triggers for each channel, by adjusting **DmaClaSrcSelRegs.DMACHSRCSELx.bit.CHy** to the appropriate trigger number.
5. Lock trigger source registers, before running DMA (optional).
6. Define the transfer parameters and addresses for each channel.
7. Change interrupt triggering behavior and enable the DMA controller, by editing the **DmaRegs.CH1.CONTROL.all** register.
8. Set PIE vector table with the desired DMA interrupt triggers and enable those, by setting the **MIER** register bits accordingly.

C.5.3 Example

An example function materializing the described actions is presented below. Although DMA is not used in the current Laelaps motion control application, this chapter refers to this useful hardware solution to outline its possible uses for future applications. For example, a possible usage could be the various periodic data transfers among the EtherCAT Data Objects and the application variables. Note that the DMA has multiple channels that can perform data transfers simultaneously.

```
void InitDma(void)
{
    //
    // Enable EALLOW Register Access
    //
    EALLOW;

    //
    // Reset DMA Module
    //
    DmaRegs.DMACTRL.bit.HARDRESET = 1;

    //
    // Make DMA Unaffected to Emulation HALT
    //
    DmaRegs.DEBUGCTRL.bit.FREE = 1;

    //
    // Disable Channel 1 Priority Mode
    //
    DmaRegs.PRIORITYCTRL1.bit.CH1PRIORITY = 0;

    //
    // Configure DMA Operation
    //
    DmaRegs.CH1.MODE.all = 0xCB01;

    //
    // Select Trigger Source for each Channel
    //
    DmaClaSrcSelRegs.DMACHSRCSEL1.bit.CH1 = 30
    DmaClaSrcSelRegs.DMACHSRCSEL1.bit.CH2 = 0;
    DmaClaSrcSelRegs.DMACHSRCSEL1.bit.CH3 = 0;
    DmaClaSrcSelRegs.DMACHSRCSEL1.bit.CH4 = 0;
    DmaClaSrcSelRegs.DMACHSRCSEL2.bit.CH5 = 0;
    DmaClaSrcSelRegs.DMACHSRCSEL2.bit.CH6 = 0

    //
    // Disable Register Lock
    //
    DmaClaSrcSelRegs.DMACHSRCSELLOCK.bit.DMACHSRCSEL1 = 0;
    DmaClaSrcSelRegs.DMACHSRCSELLOCK.bit.DMACHSRCSEL2 = 0;

    //
    // Configure 1 word Transfer per Burst & each Transfer's Size
    //
    DmaRegs.CH1.BURST_SIZE.bit.BURSTSIZE = 0;
    DmaRegs.CH1.TRANSFER_SIZE = Size;

    //
    // Step of Source Pointer increment in each Burst
    //
    DmaRegs.CH1.SRC_TRANSFER_STEP = Step;

    //
    // Select Source Address
    //
    DmaRegs.CH1.SRC_ADDR_SHADOW = (uint32_t)&(Desired Address);

    //
    // Step of Destination Pointer increment in each Burst
    //
    DmaRegs.CH1.DST_TRANSFER_STEP = 0;

    //
    // Select Destination Address
    //
    DmaRegs.CH1.DST_ADDR_SHADOW = (uint32_t)&(Desired Address);

    //
    // Configure Channel 1 Control Register
    //
    DmaRegs.CH1.CONTROL.all = 0x0091;
```

```
    //
    // Enable DMA's CPU Interrupts
    //
    PieCtrlRegs.PIEIER7.bit.INTx1 = 1;
    IER |= M_INT7;
}   // End Of InitDma()
```

C.6 Inter-Processor Communication

C.6.1 Overview

The Inter-Processor Communication module's main task is to establish communication between the TMS320F28379D cores [64]. Some of its features involve:

- Message RAMs
- IPC flags, interrupts and command registers
- Clock Configuration and Flash pump semaphores
- GS0-15 RAM blocks that may be assigned to either CPU by modifying the **MemCfgRegs.GSxMSEL** register appropriately.

The IPC's extensive features make communications easy to utilize. The interested reader may refer to the related documentation for further reading [64] [117]. Figure C-8 demonstrates graphically the module's architecture. For simple communication schemes, the IPC provides the developer with RAM space and registers presented in Table C-4, Table C-5 and Table C-6, respectively.

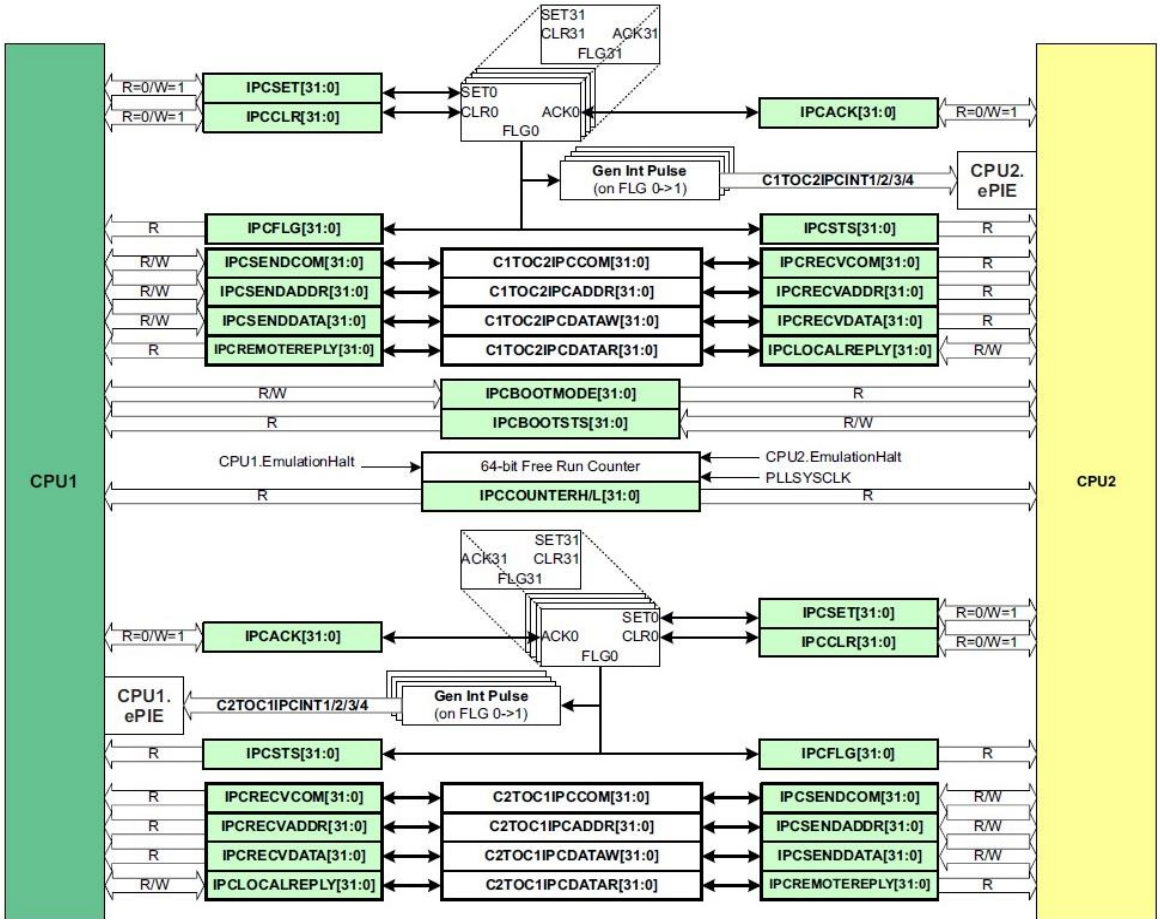


Figure C-8. IPC unit architecture.

Table C-4. IPC message RAMs overview.

Message RAM	CPU1 Subsystem		CPU2 Subsystem	
	CPU1	***CPU1.DMA***	***CPU2***	***CPU2.DMA***
CPU1 TO CPU2	R/W	R/W	R	R
CPU2 TO CPU1	R	R	R/W	R/W

Table C-5. IPC GSx RAM overview.

Ownership of GSx RAM	CPU1 Subsystem		CPU2 Subsystem	
	CPU1	***CPU1.DMA***	***CPU2***	***CPU2.DMA***
CPU1	R/W/Exe	R/W	R	R
CPU2	R	R	R/W/Exe	R/W

Table C-6. IPC communication registers overview.

Local Register Name	Local CPU	Remote CPU	Remote Register Name
IPCSENDCOM	R/W	R	**IPCRECVCOM**
IPCSENDADDR	R/W	R	**IPCRECVADDR**
IPCSENDDATA	R/W	R	**IPCRECVDATA**
IPCSENDREMOTEREPLY	R	R/W	**IPCLOCALREPLY**

C.6.2 Basic IPC communication

According to TI's manual [64], a basic IPC data transfer may be realized with a simple mechanism, avoiding the use of complex software solutions. Here, a simple example is given:

1. *CPUx* writes a command to **IPCSENDCOM** to edit a bunch of data from the remote CPU's (*CPUy*) local RAM in a specified address that **IPCSENDADDR** dictates. Moreover, **IPCSENDDATA** is loaded with any required data for transfer.
2. *CPUx* writes the predefined flags that the developer decided to use by setting the corresponding bits of I**PCSET** register.
3. After *CPUy* receives *CPUx's* triggered interrupt, it proceeds with actions that the corresponding ISR is designed to execute.
4. Then, *CPUy* acknowledges the interrupt and writes **IPCLOCALREPLY** register if necessary.
5. *CPUx* sees the acknowledgment and receives *CPUy*'s reply from **IPREMOTEREPLY** register.

Last but not least, there are software packages like IPC Drivers that extend the module's functionality, introducing more advanced features by supporting multiple message exchanges and ISRs. The downside is the increased development time that includes link-script modifications. To avoid such disadvantages, the IPC Drivers Lite support fewer options, resulting in less steep learning curves and wide coverage for most applications' needs. The reader should consult the extensive documentation provided by TI to get a better insight into these useful capabilities [117].

In the code snippet below, a simple dual-core communication scheme is illustrated. This implementation is IPC-Drivers free.

```
//
// CPU1
//
```

```
//
// CPU2
//
```

```
_interrupt void IPC1_isr(void)
{
    //
    // Acknowledge PIE Group
    //
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

    //
    // Receive Data
    //
    data1_cpu1 = IpcRegs.IPCRECVADDR;
    data2_cpu1 = IpcRegs.IPCRECVDATA;
    //
    // Clear Interrupt Flag
    //
    IpcRegs.IPCACK.bit.IPC1 = 1;
}
```

```
_interrupt void IPC0_isr(void)
{
    //
    // Acknowledge PIE Group
    //
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

    //
    // Clear Interrupt Flag
    //
    IpcRegs.IPCACK.bit.IPC0 = 1;

    //
    // Receive Data, Process & Send Data
    //
    GpioDataRegs.GPBTOGGLE.bit.GPIO34=1;
    data1_cpu2 = IpcRegs.IPCRECVADDR;
    data2_cpu1 = IpcRegs.IPCRECVDATA;
    IpcRegs.IPCSENDADDR = data_out_cpu2;
    IpcRegs.IPCSENDDATA = data_out_cpu2;

    //
    // Set Interrupt Flag
    //
    IpcRegs.IPCSET.bit.IPC1 = 1;
}
```

C.7 Delfino Linking Process

Like in every embedded system, whenever the C2000 microcontroller powers on, the first piece of firmware to run as initialization is the so-called bootloader that is responsible, among others, for the below tasks [114].

1. Initialize the system's memory, by consulting the memory mapping created by the linker.
2. Initialize the Debug Console.
3. Initialize the Peripherals.
4. Configure the cache and the MMU (the module responsible for translating virtual to physical memory addresses).
5. Load the code and data image from the memory.

This is an automated procedure and no further attention should be given if the reader is not interested in such low-level processes that take place. The significant point that will be analyzed below is the way that the linker produces the memory mapping file by concatenating memory blocks provided by the corresponding command file.

Currently, the object image used by Texas Instruments DSP processors is called Common Object File Format (COFF). A newer type is the so-called Embedded Application Binary Interface (EABI), a variant of the ELF Object File Format. In the Laelaps firmware, COFF is used due to the various compatibility issues that the migration may cause. Certainly, in the future, migration is suggested. It should be noted that it is generally up to the user to create the link script, while TI provides some common examples that cover most applications' needs.

The main task of the link script is to inform the linker about the number of different physical memory segments that exist in the hardware, their size and starting addresses [114]. The most common sections that an object file maps to memory are:

- **.text**: It contains all the executable code of the loaded application.
- **.stack**: All the global variables/static arrays that have non-zero initial values are stored in this section.

- **.ebss**: All uninitialized global/static variables are stored here.
- **.esymem**: This section reserves space for dynamic memory allocation.
- **.econst**: This section contains all of the firmware's constant coefficients.

The above sections are generic, while others are entailed for a very specialized cause like CLA memory space. To use the CLA unit, the following sections should be defined.

- **Cla1Prog**: It holds the CLA's executable routines and information for loading them to specific RAM blocks during runtime.
- **CLAscratch**: The scratchpad section is used as the memory space for all data structures that are handled by the CLA.

To clarify the above-described notions, the Laelaps Motion Control command file [52] should be consulted. In this thesis, the implemented memory allocation provides sufficient space for each firmware's data and code. This was cross-verified with the *Memory Allocation* tool provided by TI's CCS (located under *View* menu of the main task-bar). Admittedly, the allocation is not optimal and in the future by consulting the literature [114] [118], several improvements could be made.

C.8 Power Supply Considerations

The Delfino LaunchXL-F28379D evaluation package, supports three different power supply configurations [119]. By placing jumpers JP1, JP2, JP3 and JP6, these configurations may be exploited, with different levels of JTAG isolation. The available settings are listed in Table C-7, while Figure C-9 designates the aforementioned pins. It should be noted that all of the firmware solutions developed in this thesis are designed to be used in full JTAG isolation (Configuration 3). The power supplies are provided externally, by connecting the corresponding power pins of the Boosterpack headers.

Table C-7. LaunchXL-F28379D power supply configurations.

Configuration	JP1	JP2	JP3	JP6	External 5V	External 3.3V
1	Yes	Yes	Yes	No	No	No
2	No	No	No	Yes	No	Yes
3	No	No	No	No	Yes	Yes

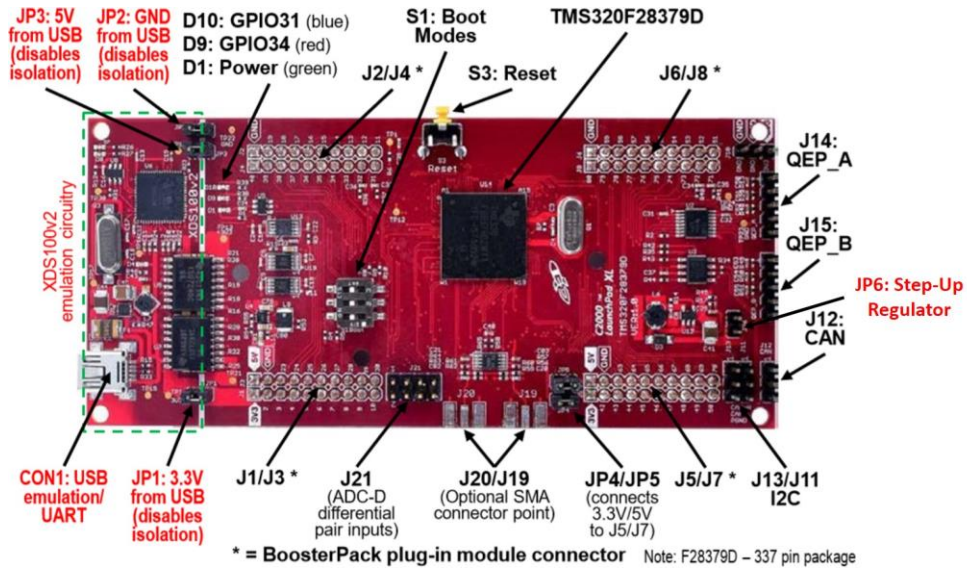


Figure C-9. LaunchXL-F28379D Pinout.

Appendix D. CLA MATH

The basic aim of the CLA is to accelerate complex mathematical procedures, relieving the CPU's bandwidth. Towards this, TI introduces CLA Math Library, which constitutes a collection of highly optimized mathematical oriented functions with low CPU run-times and easy code integration. Their contextual logic is founded in lookup tables for trigonometric, logarithmic and exponential operations. These tables are already built-in but require an effort to use. The Library's manual is very informative and the reader is strongly recommended to consult it [115]. The basic workflow to add the CLA Math library routines to any firmware that is required lies below.

1. From the *Project Properties (Right Click on Project's Name) -> Linked Resources*, add a new variable with the name *MATH_ROOT* and specify the path of the source files in C2000WARE's folder structure (e.g. c28 folder in Figure D-1).

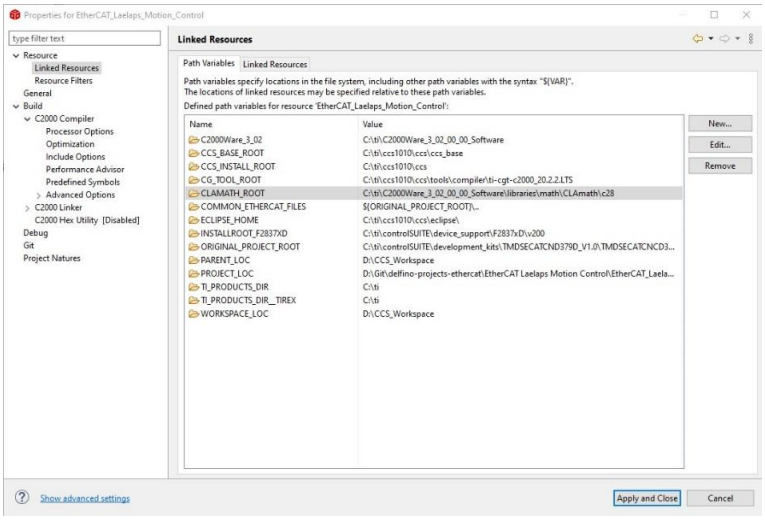


Figure D-1. Project's Linked Resources overview.

2. Add in the *Include Options* Pane, under *C2000 Compiler*, at the include search path window (Figure D-2) *CLAMATH_ROOT/include*. Also, create a folder in the project, name it CLA_MATH and add its path in the aforementioned pane, too

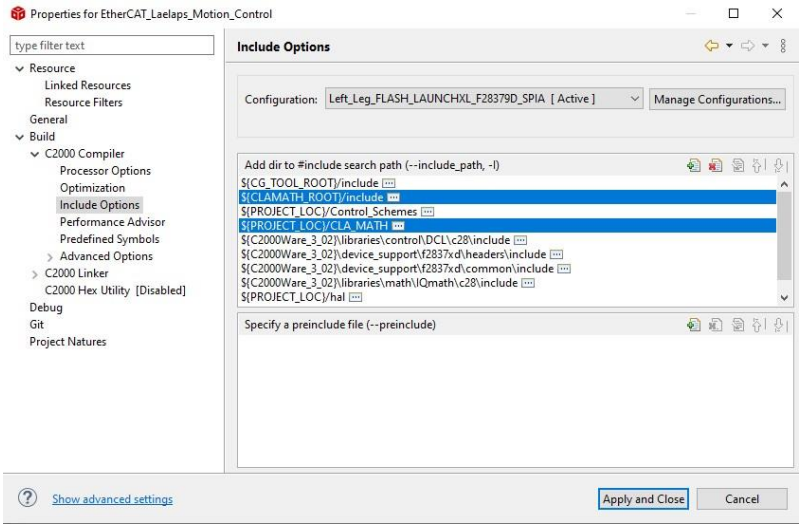


Figure D-2. Project's Linked Resources overview.

3. Select the CLA type support. The default option for TMS320F28379D is *cla1*. That selection is located inside the *Processor Options* window, like the one in Figure D-3. At this point, fpu32 should also be enabled, if needed. The configuration of CLA Math is directly dependent on the fpu32 option.

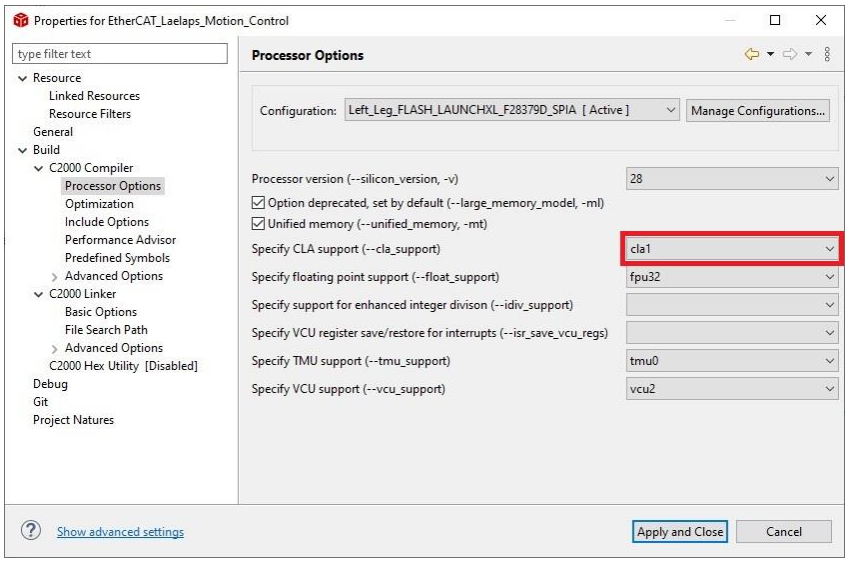


Figure D-3. Project's Processor Options.

4. Lastly, the lookup tables should be integrated into the program along with the required math routines and their header file. The two options here are:
    - Insert the created tables from the c28x CLA Math source folder to the respective project, manually copy them to FLASH and eventually load them to RAM, during runtime.
    - Add the symbols and the required Boot ROM paths to the project's properties. Those ROM tables do not require any further actions to become operational and do not reserve user memory.
5. In this workflow, the fpu32's capabilities along with the CLA's ROM tables are used. Navigate to *File Search Path*, under C2000 Linker structure, as shown in Figure D-4.

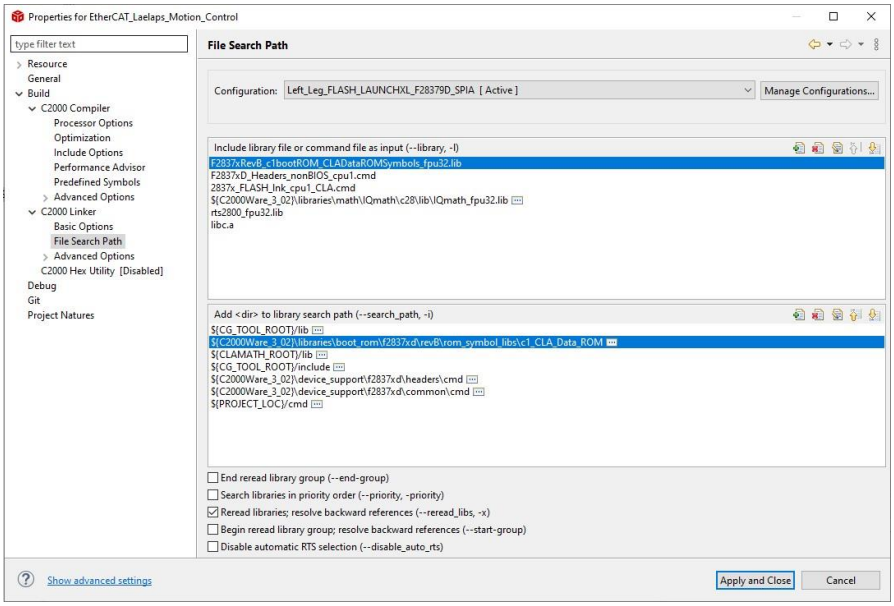


Figure D-4. Project's File Search Path tab.

Appendix E. Setup TI's CCS Projects

E.1 Download TI's CCS and Import Project

The instructions on how to import a CCS project are laid below.

1. Download the latest version of TI's CCS from the official site [59] and follow the installation's steps.
2. Download the required C2000Ware libraries. To do that, navigate to *View->Resource Explorer* Menu and choose the desired version, under *Software* Folder. The default version for the projects of this thesis is **v3.03.00.00**. The procedure is illustrated in Figure E-1. If the installation succeeds, the green check next to the downloaded version appears.

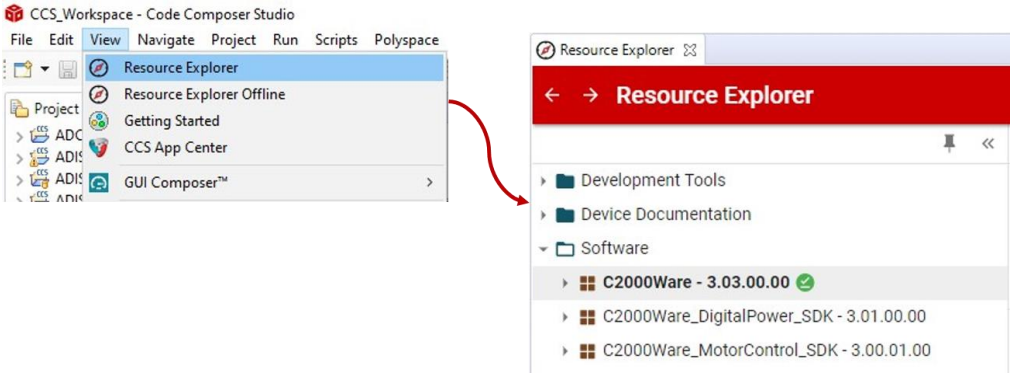


Figure E-1. C2000Ware installation procedure.

3. Click on *File -> Import…* and opt for *CCS Projects*, as shown in Figure E-2.

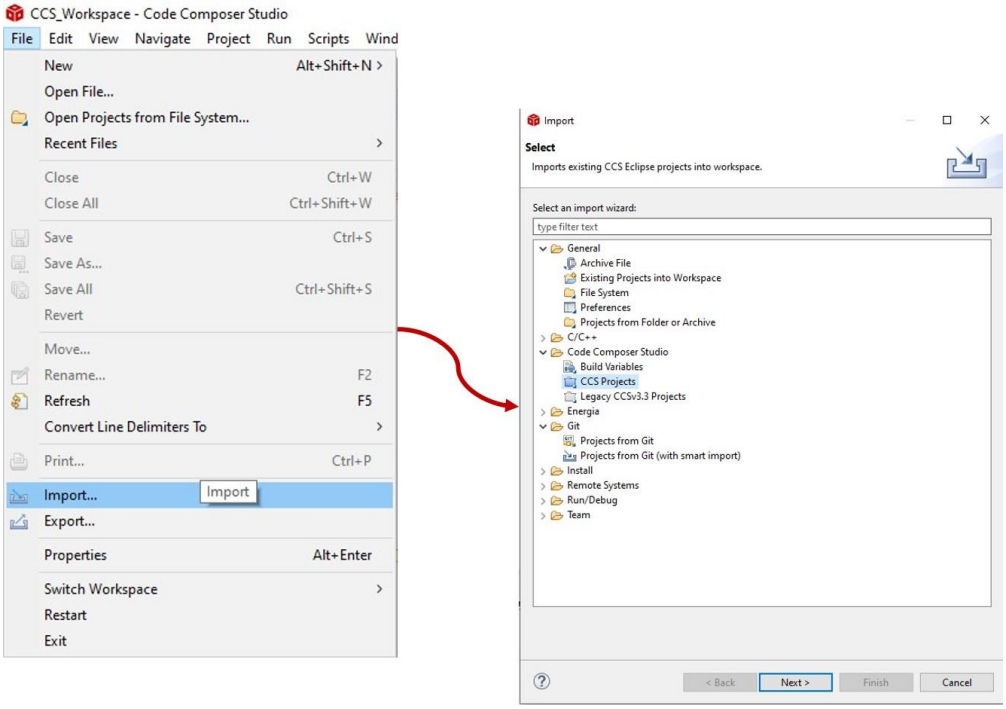


Figure E-2. CCS Import workflow.

4. Browse to the directory of the desired project's source files and hit ok, when prompted. The view should be like the one in Figure E-3. To perform a recursive import of the

referenced projects located in the same search directory, check the first checkbox, displayed in the aforementioned picture. If the project is being imported from a Git-repository do not opt for the “Copy projects into workspace” checkbox.

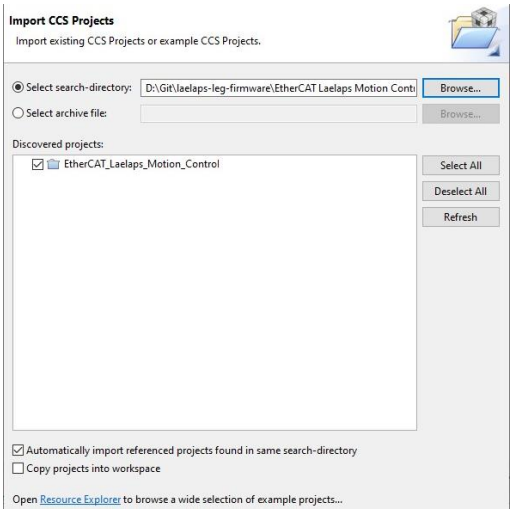


Figure E-3. Import CCS project options.

E.2 Create a Target Configuration

After the project is imported successfully, the target configuration may be corrupted. It should be noted that the target configuration is not linked externally to the current thesis’ projects, but rather included in each project’s source files. Hence, no further action is required. Nevertheless, to create it from scratch, the following guide should be consulted.

1. Select *View -> Target Configuration* as depicted in Figure E-4.

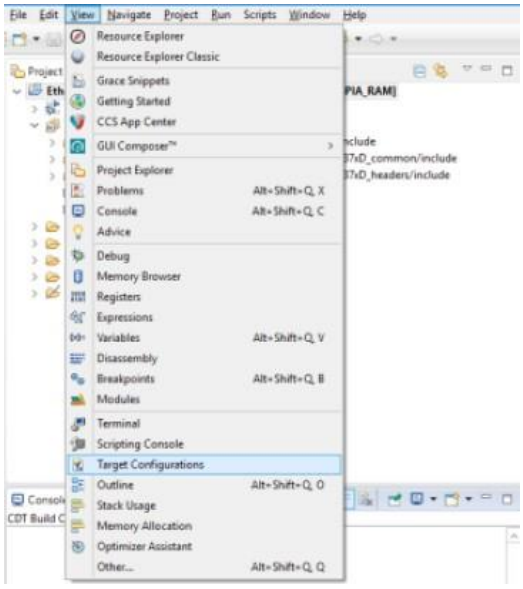


Figure E-4. “View” drop-down menu.

2. Select New Target Configuration from the Target Configurations window (Figure E-5).

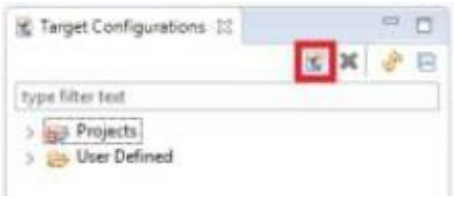


Figure E-5. New Target Configuration.

3. Name the Target Configuration after the MCU being used and click on *Finish* (Figure E-6).

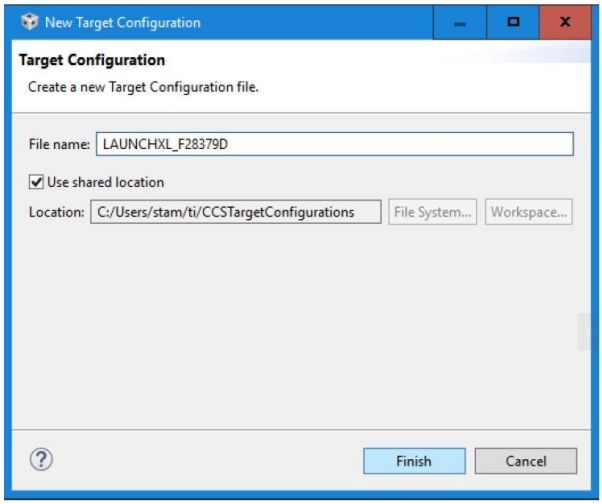


Figure E-6. Name Target Configuration.

4. Select the Texas Instruments XDS100v2 USB Debug Probe at the Connection tab, select TMS320F28379D at the *Board or Device* tab and click *Save* as shown in Figure E-7.

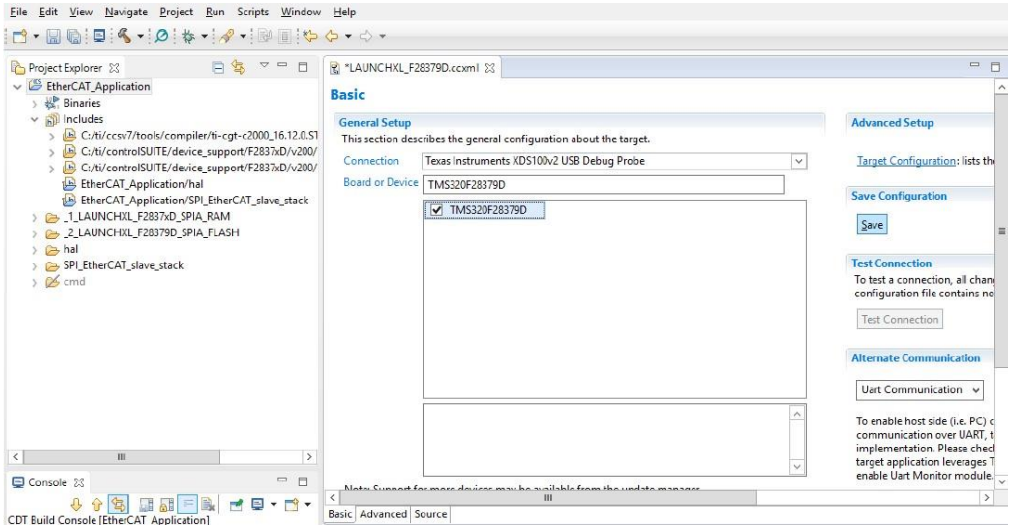


Figure E-7. Select connection and device.

5. Close the *LAUNCXL_F28379D.ccxml* window after the saving is completed.
6. To link the newly created configuration, navigate at the Target Configurations window (right-hand side) expand the User Defined directory, right-click on LAUNCHXL_F28379D.ccxml and select Link *File To Project -> [Project Name*] (e.g. EtherCAT_Application) as shown in Figure E-8.

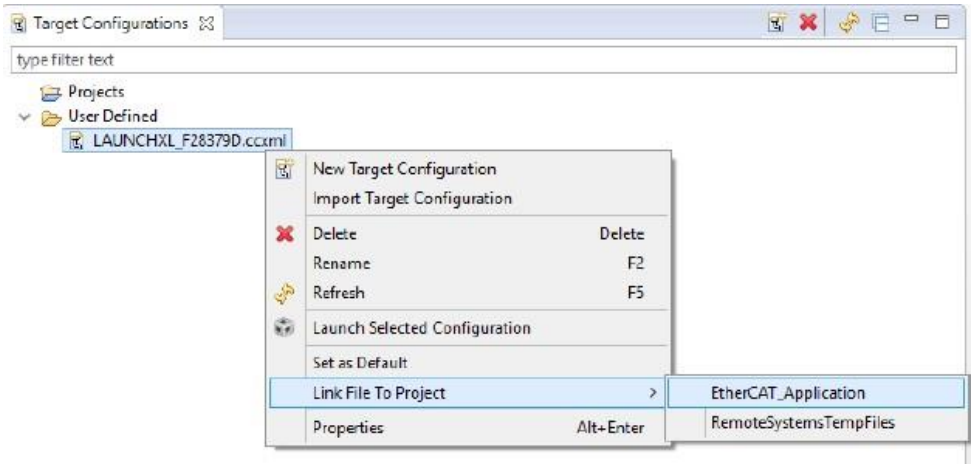


Figure E-8. Link the created Target Configuration to the project.

7. Close the Targer Configuration tab.

E.3 Build and Deploy the Project

1. Now the project is ready for deployment. Build the project to verify the setup, by clicking on the hammer icon, located at CCS' main taskbar (Figure E-9).

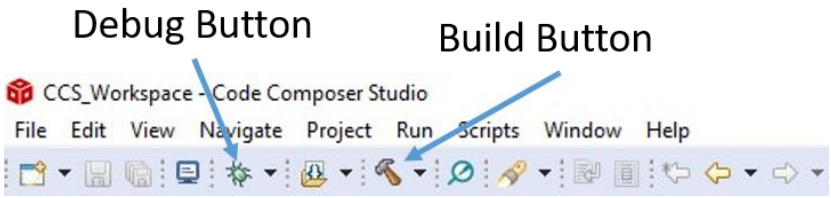


Figure E-9. TI's CCS main taskbar.

2. Debug the project, by clicking on the Debug button, located in the main taskbar. If the Debug session configuration pops-up, choose the CPU1 option, as shown in Figure E-10. Leave "Create a debug group for selected cores" unchecked.

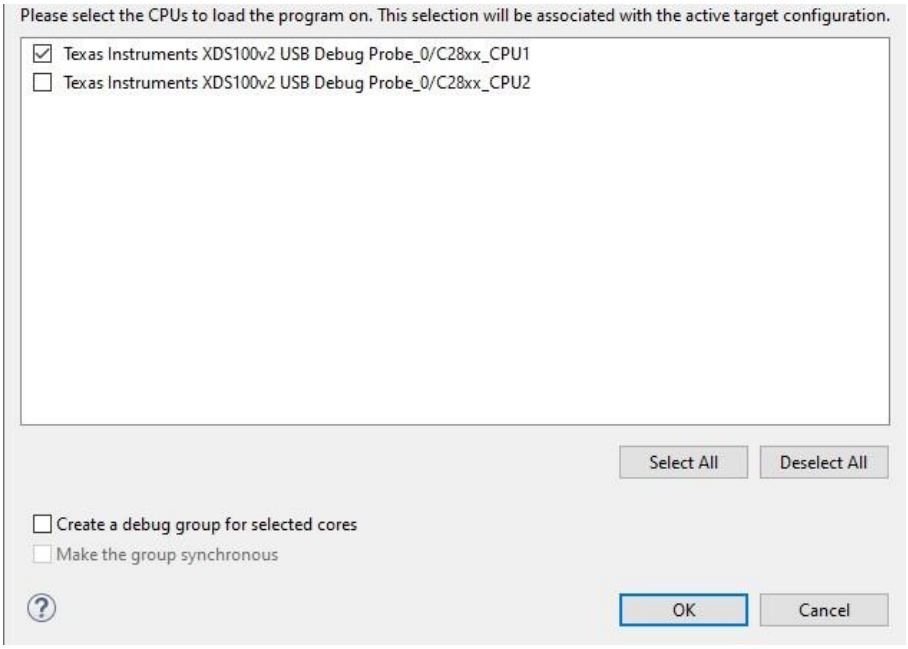


Figure E-10. Firmware's Debug Session configuration.

3. After the download process is finished, run the project by clicking the Run button at the Debug Perspective’s ribbon menu.

The project should be up and running. The Delfino’s onboard LEDs should be flashing for the predefined sampling period.

Now that the software has been deployed, it is time to set up the master and perform the necessary wire connections. A more detailed guide can be found here [20]. For completion, a brief one may be found in Appendix A.

Appendix F. Integrate TI's DCL Library

The steps required to integrate the TI's DCL libray into a CCS project are adumbrated below:

1. Add the appropriate include path of DCL's source folder (e.g. c28 in C2000WARE), in the Project Properties Pane (*Project Properties -> Include Options*) as in Figure F-1.

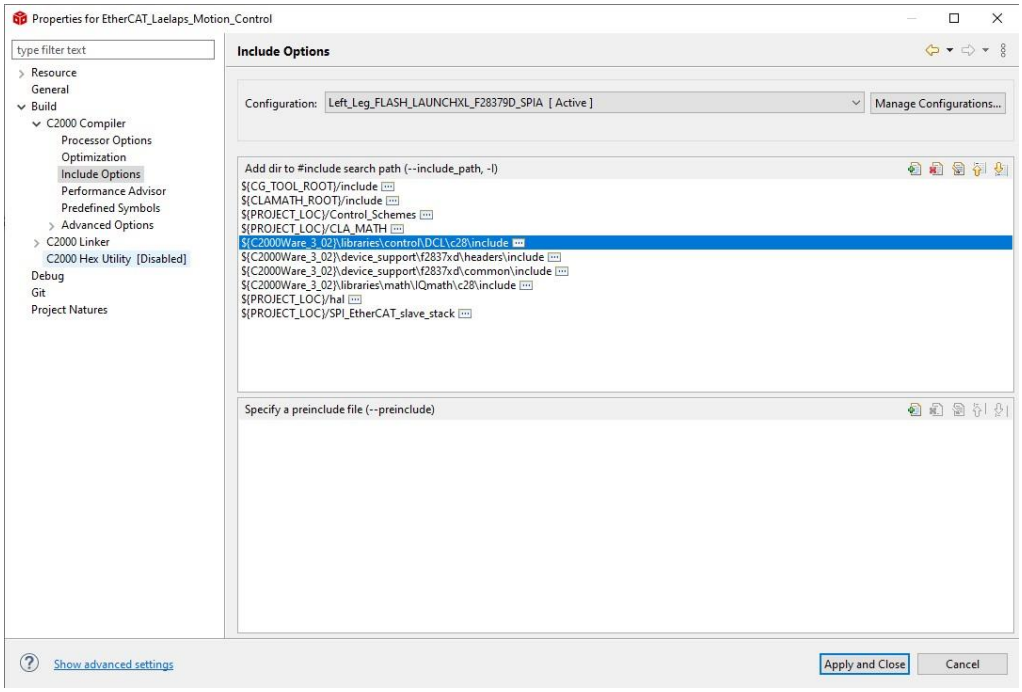


Figure F-1. Project's Include Options pane.

2. Some additions should be made in the linker command file (Section 3.5.8) regarding the assignment of memory space to DCL functions. That is crucial in case that DCL routines run on CPU rather than on CLA (***DCL_runPID_Cx*** family*).* An example of such modifications is considered in the snippet below.

```
dclfuncs : {} LOAD=FLASHD | FLASHE, RUN=RAMD0, TABLE(BINIT)
.binit : {} > FLASHD | FLASHE | FLASHF
```

3. Add the desired controllers' source files to the respective project's tree and include the accompanying headers in the project's source files.

Appendix G. Code Style Guideline

Modern software development requires the cooperation of multiple developers, with different knowledge backgrounds. For this procedure to be effective and efficient, the created code should be consistent, well-written, and documented. These features are of utmost importance, for the developed software to be maintained and furtherly expanded in the future. To this end, the adoption of coding style rules is imperative.

In the current thesis, the created firmware packages comply with a coding style tailored-made for embedded systems (Barr C [69]). Since every application has certain requirements and special needs, the adopted coding style deviates slightly from the aforementioned standard Barr C. The key differences are laid below.

- The names of all data types, including structures, unions and enumerations, shall consist of condensed words with each one's first letter being a capital (e.g. *AdcaResult0*).
- Specifically, for the global data types, in addition to the aformenetioned naming convention, their names should have a "*g_*" prefix (e.g. *g_StateMachine*).
- The names of functions also consist of condensed words, with their first letter being a capital (e.g. ***InverseKinematics()***).
- All definitions names shall consist of capital letters, with words separated by underscore "_" (e.g. **ZERO_ANGLE_POS**).
- Each routine should have a preamble that describes its functionalities, return types, input parameters and shall have Doxygen support. For example:

```
//*****************************************************************************
//
//! \brief CLA-Runtime Constant Velocity Planner Routine with Ground Collision
//! Avoidance Feature.
//!
//! It materializes the End-Effector Planner, Producing each Step's (x,y)
//! Coordinates. The ground collisions are prevented due to the zero angular
//! velocity that elliptical trajectory presents at toe's touch-down.
//!
//! \return None.
//
//*****************************************************************************
```

- All functions should be commented inline after their closing bracket. For example:

```
void RunHipController(void)
{
    //
    // Function's Code
    //

}   // End Of RunHipController()
```

- Inline code comments for documenting code are not permitted. Comments inside a routine should be placed above the line of code or the block of lines that are intended to be discussed and always in the following form.

```
    //
    // Initialize Joints' ePWM Modules
    //
    InitEpwm1();
    InitEpwm2();
```

- All **if()…else if()** structures shall end with an **else**, even if the latter remains empty. Also, each condition shall be explained above its conditional statement. The brackets should always be in a separate line. For example:

```
    //
    //  Explanation of Condition 1
    //
    if (Condition1)
    {
        //
        //  Code Block 1
        //
    }

    //
    //  Explanation of Condition 2
    //
    else if (Condition 2)
    {
        //
        //  Code Block 2
        //
    }

    //
    // Else Condition
    //
    else
    {
        //
        //  Code Block 3 (May be left empty)
        //
    }
```

- All source and header files shall have a file preamble at their beginning. This preamble is located after the Doxygen's \addgroup command, if present. For example:

```
//
//! \addtogroup {Group Name}
//! @{
//
//*****************************************************************************
//
//! \file cla_utilities.cla
//!
//! \brief Give a brie description about the file's contents
//!
//! Give more details, if necessary.
//!
// Author:      {Authors' Names}
// Maintainer:  {Maintainers' Names}
//
// Revision History
// Version {Ver. Number} | {Date} | {Developer's Name}
//
// (C) Copyright 2020, NTUA CSL-EP Legged Robots Team
//
//*****************************************************************************
```

- Comments located outside of functions shall have the below form:

```
//*****************************************************************************
//
// Controllers' Filter Bandwidth Variable
//
//*****************************************************************************
volatile uint16_t g_FilterBandwidth = 1U;
```

Appendix H. MISRA C 2012 Standard

The current thesis CCS projects comply with MISRA C:2012 standard [62]. The MISRA C guidelines define a subset of the C language in which the risk of making mistakes is either removed or reduced. Many standards for the development of safety-related software require, or recommend, the use of a language subset and this can also be used to develop any application with high integrity or high reliability requirements.

To use MISRA C, it is necessary to develop and document:

- A compliance matrix, showing how compliance with each MISRA C guideline is checked.
- A deviation process by which justifiable non-compliances can be authorized and recorded.

Every MISRA C guideline is classified as either being a “directive” or a “rule”. A directive is a guideline for which it is not possible to provide the full description necessary to perform a check for compliance. Static analysis tools may be able to assist in checking compliance with directives but different tools may place widely different interpretations on what constitutes a non-compliance. A rule is a guideline for which a complete description of the requirement has been provided. It should be possible to check that source code complies with a rule without needing any other information. Moreover, every MISRA C guideline is given a single category of “mandatory”, “required” or “advisory”. Mandatory rule violations are not permitted under any circumstances.

Except of protecting from run-time errors by writing clean, documented code, MISRA C tries to make the code implementation-free. This means that compiler-specific macros, etc. are treated as violations. On the other hand, in this thesis the developed firmware packages do not have such requirements. The compiler is provided by TI for the particular C2000 Delfino architecture (C28x Compiler). Also, it is common for embedded systems to have non-standard macros and other commands that utilize a functionality that only the architecture in question supports. This should be taken into consideration whenever MISRA C:2012 is adopted for an embedded application.

H.1 Laelaps II Motion Control’s MISRA C 2012 Compliance

To prove that the developed firmware complies with MISRA C: 2012 standard, a compliance matrix is created. Since this matrix includes every MISRA rule, its reproduction would be extensive and overwhelming for the reader. There are available reports in the project’s Bitbucket repository [52]. This matrix is exported by the used Static Analysis tool and states which directives and rules are being possibly violated. Then, the developer has to decide if these violations may be permitted or take the required steps to eliminate them. Table H-1 lists all the violations of MISRA C: 2012 rules and directives for the developed firmware, while each violation’s classification is also stated. For a more comprehensive representation, the bar-plot of Figure H-1 is created with the aforementioned table’s data. The orange bars indicate the number violations of a required rule, while the gray ones the violation of an advisory rule. No mandatory rules are violated. MISRA states that each required rule violation needs a formal deviation description (refer to MISRA C:2012, Section 6.2.2). On the other hand, advisory rules do not necessarily need formal deviation description (refer to MISRA C:2012, Section 6.2.3). Mandatory rules shall not be violated under no circumstances (refer to MISRA C:2012, Section 6.2.1).

Table H-1. MISRA C: 2012 motion control's guideline violations.

Guideline	Classification	#Violations
D1.1	Required	8
D4.1	Required	4
D4.4	Advisory	2
D4.6	Advisory	174
D4.8	Advisory	1
2.1	Required	2
8.2	Required	11
8.3	Required	2
8.4	Required	0
8.5	Required	1
8.7	Advisory	36
8.9	Advisory	5
8.13	Advisory	1
10.3	Required	19
10.5	Advisory	5
10.8	Required	17
11.1	Required	10
12.1	Advisory	39
12.4	Advisory	3
14.3	Required	2
20.5	Advisory	1
Total		**343**

From the Table H-1, it becomes apparent that most violations concern the "advisory" guidelines. Below, each violation of the required rules and directives is accounted and analyzed. As for the "advisory" guidelines, they are not furtherly analyzed, but they were investigated. Most of them are not considered as problems in the current implementation. Lastly, Figure H-1 presents the results in a more intuitive way.

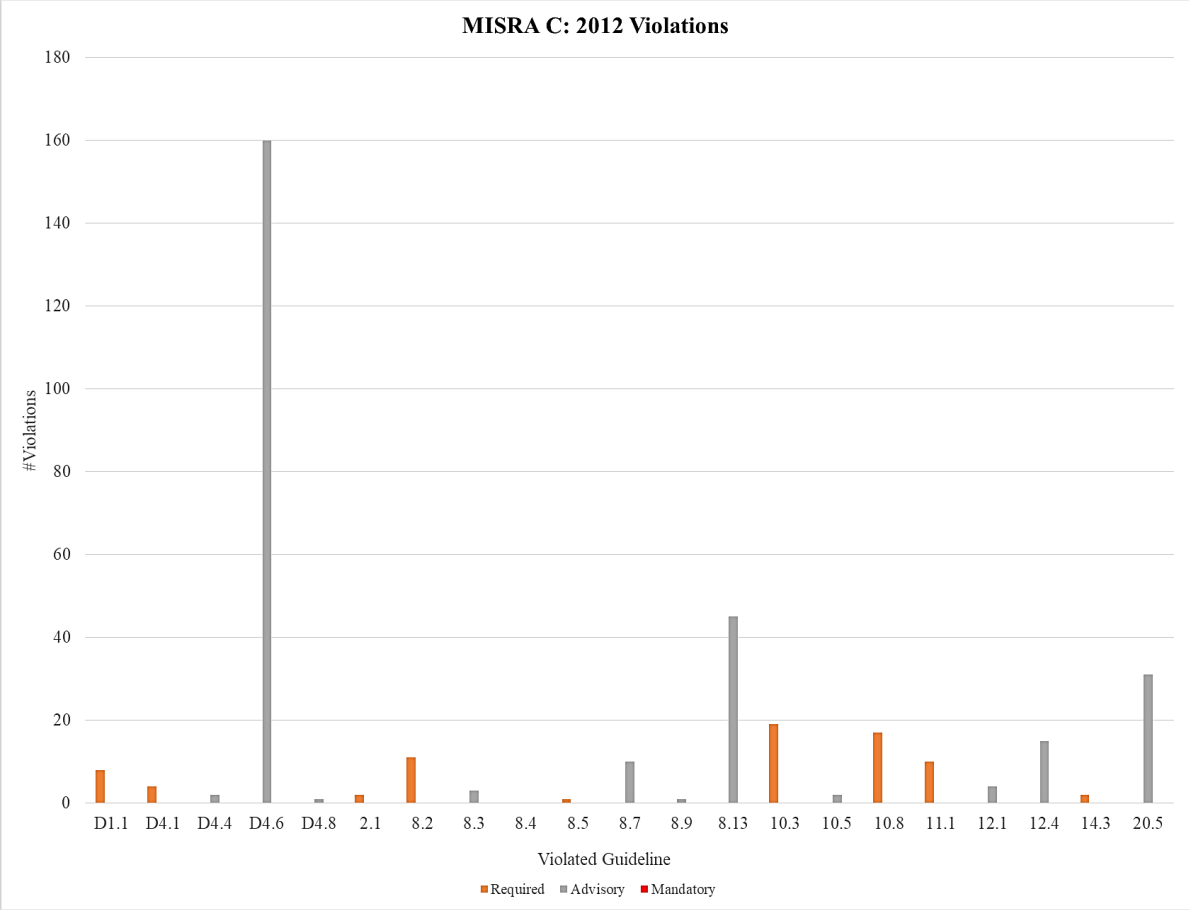


Figure H-1. MISRA C: 2012 motion control guideline violations' bar-plot.

DIR 1.1

Any implementation-defined behaviour on which the output of the program depends shall be documented and understood. This directive states that implementation-specific code, like pragmas, or integer to float cast should be avoided. In the current project, the compiler is fixed and pragma instructions are used to assign a variable to a specific memory section; thus cannot be avoided. Lastly, the integer to float cast should be avoided, because the result depends on the compiler (rounding behavior). Since the compiler is fixed, this is not required.

DIR 4.1

Run-time failures shall be minimized. This directive considers the overflow/underflow errors that may occur from casting a variable of a wider type to a narrower (e.g. uint32_t to int32_t) during run-time. In the current project casting types could not be avoided. To ensure that no overflow occurs, careful testing has been made, both in the real hardware and using Static Analysis.

Rule 2.1

A project shall not contain unreachable code. This is a False-Positive of the Static Analysis software. The if-conditions that are indicated by the software depend on an arbitrary register's value. The software cannot make this distinction and the if-conditions evaluate only to false, since the registers take values during run-time.

Rule 8.2

Function types shall be in prototype form with named parameters. This violation is understandable and refers to the **POSSPEED** structure. This structure violates this rule, since it contains unnamed parameters in its members (e.g. ****Init()***). Nevertheless, this is the TI's proposed way to handle modules, such as eQEPs. The original code can be found in C2000Ware's ***eqep_pos_speed_cpu01*** example project.

Rule 8.3

All declarations of an object or function shall use the same names and type qualifiers. This is a False-Positive of the Static Analysis tool. The *g_AnalogKnee* and *g_AnalogHip* variables are checked and have the same type qualifiers in all of their declarations.

Rule 8.5

An external object or function shall be declared once in one and only one file. This rule is violated in one specific case that ***GPIO_WritePin()*** cannot be declared in a header file due to TI's type conflict with the types of the EtherCAT stack (uint16_t vs. UINT16). The types are basically the same but come with different names in the aforementioned implementations. Thus, this violation could not be avoided.

Rule 10.3

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category. This rule as mentioned before cannot be followed, since casting types to narrower ones serves a crucial functionality in the firmware. All occurences of this violation have been reviewed.

Rule 10.8

The value of a composite expression shall not be cast to a different essential type category or a wider essential type. This MISRA rule states that casting a composite expression to a wider/different essential type, may have different results with different compilers. In the current case, this type of casting is permitted since the implementation here is specific and the compiler is fixed. The occurencies have been reviewed and tested and their results are the expected.

Rule 11.1

Conversions shall not be performed between a pointer to a function and any other type. This rule is violated by specific castings in function-pointers (e.g. (uint16_t)(***&Cla1Task1***)) that do not comply with MISRA standards. Nevertheless, this is TI's default way to handle certain assignments and thus is permitted.

Rule 14.3

Controlling expressions shall not be invariant. This is similar to the violations of ***Rule 2.1***. The if-conditions are considered invariant by the Static Analysis tool, but actually they are not.

H.2 IMU Fimrware's MISRA C 2012 Compliance

Both IMU libraries were designed to comply with MISRA C: 2012 standard. Below, each CCS project violations are analyzed. Note that only the custom files were checked for MISRA compliance (the contents of each *ADIS163xx_LIBRARY* folder).

H.2.1 ADIS16364 Library's Compliance

By reviewing Table H-2, it is deduced that the project violates few advisory guidelines and only one required one. Figure H-2 gives a more clear view of the results.

Table H-2. MISRA C: 2012 ADIS16364 guideline violations.

Guideline	Classification	#Violations
D4.4	Advisory	2
D4.6	Advisory	28
1.1	Required	2
12.4	Advisory	1
Total		**33**

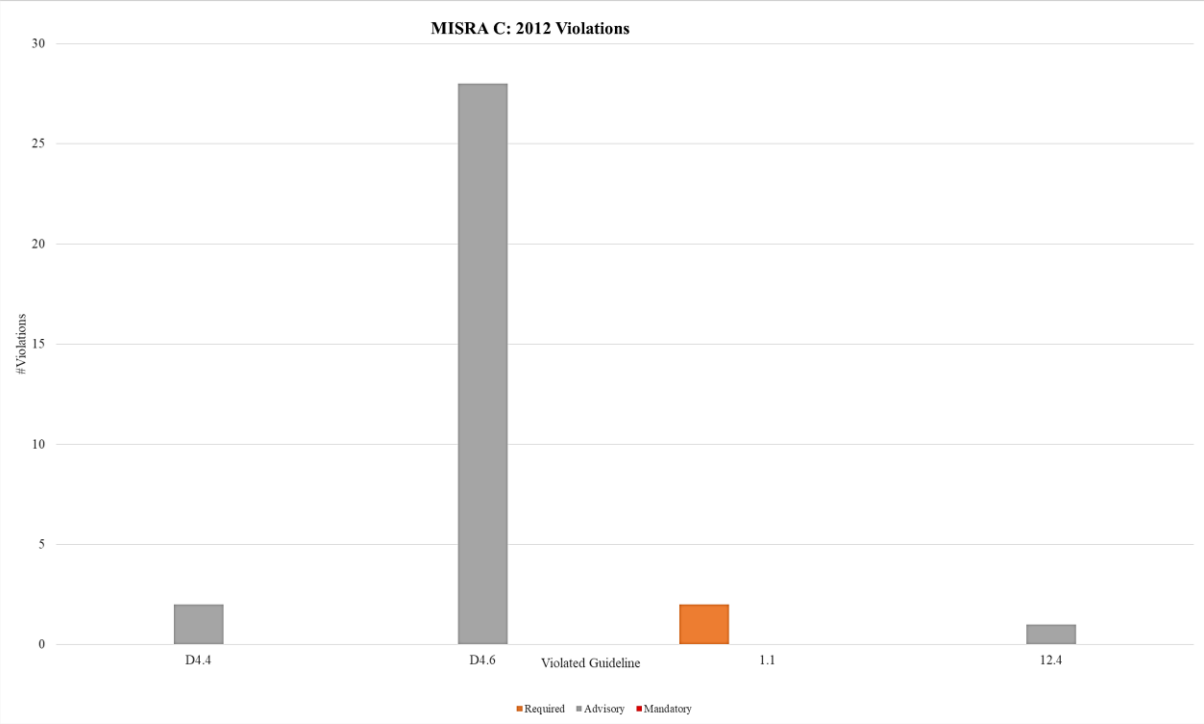


Figure H-2. MISRA C: 2012 ADIS16364 guideline violations' bar-plot.

The violated "advisory" guidelines have been reviewed and they are not considered as problems in the current implementation. The "required" guideline that is violated is analyzed below. For an in-depth analysis, refer to the Static Analysis auto-generated report that includes the detailed MISRA compliance matrix [83].

Rule 1.1

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits. This violation is triggered due to the large *ADIS16364.h* header file, in combination with the relatively large *ADIS16364.c* source file. In the current case the compiler supports their lengths and thus no actions are necessary. The used compiler is the default TI's C28x for the C2000 Delfino architecture.

H.2.2 ADIS16375 Library's Compliance

Likein the previous case, only the files of *ADIS16375_LIBRARY* folder were analyzed. The results were very similar. As can be seen in Table H-3 the library violates a single required guideline and few advisory ones. In Figure H-3, a more intuitive illustration of the results is given. The detailed reports are given in [84]. The justification of the required ***Rule 1.1*** violation remains the same with the one given in the previous section. Generally, the library's implementations is generic and can be used by other devices with a similar architecture.

Table H-3. MISRA C: 2012 ADIS16375 guideline violations.

Guideline	Classification	#Violations
D4.4	Advisory	2
D4.6	Advisory	65
D4.9	Advisory	1
1.1	Required	2
12.4	Advisory	1
Total		**71**

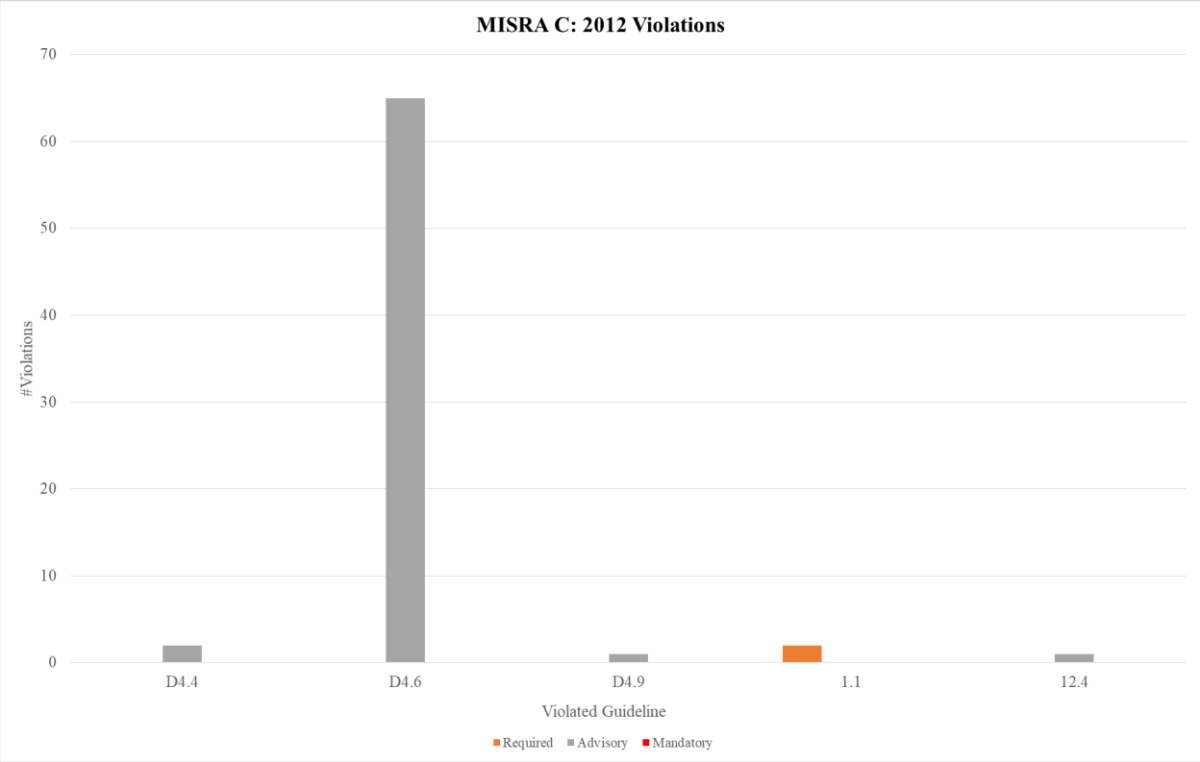


Figure H-3. MISRA C: 2012 ADIS16375 guideline violations' bar-plot.

The violated "advisory" guidelines have been reviewed and they are not considered as problems in the current implementation. For an in-depth analysis, refer to the Static Analysis auto-generated report that includes the detailed MISRA compliance matrix [84].

Appendix I. LDO Thermal Design

In the design of PCBs, using LDOs for power supply, a major challenge is to minimize thermal fatigue and overheating. Such components require extra care and special design to optimize their performance. According to the literature [79], the LDO's thermal behavior is heavily dependent to the drop-out voltage. In the current IMU-Interface PCB, the 5 V and 3.3 V LDOs are connected in parallel. Hence, the most stressed one is that of the lower voltage between the two.

The aforementioned logic supply's LDO is TI's LM1085IT [120]. The corresponding datasheet [77] presents a bulk method to estimate an arbitrary application's thermal stresses and is implemented in this section. According to the accompanying documentation, the IC is modeled in two different parts, namely Control and Output. Furthermore, their maximum junction temperatures are listed in Table I-1, below.

Table I-1. LM1085IT thermal limits.

IC Section	$T_{J,max}[^{o}C]$
Control	125
Output	150

To be on the safe side, the thermal design takes into consideration only the control's lower temperature as maximum for both parts and 25 ^{o}C as the ambient one. The maximum allowable thermal resistance ($\theta_{JA,max}$) may be obtained by the formulas in (I-1), with I_L and I_G being the load's and ground currents, respectively. Note that the ground current is the quiescent current, needed to maintain regulation. The input voltage is represented by V_{in}, while the output one by V_{out}.

$$\theta_{JA,\max} = \frac{(125 - T_{ambient})}{P_D}, with\ P_D = (V_{in} - V_{out}) \cdot I_L + V_{in} \cdot I_G \quad \text{(I-1)}$$

The maximum allowed heatsink's thermal resistance ($\theta_{HA,\max}$) is given by the equation (I-2), with θ_{HA} being the heatsink's resistance, θ_{JC} the package's top resistance and θ_{CH} thermal compound's resistance (if used).

$$\theta_{HA,\max} = \theta_{JA,\max} - (\theta_{JC} + \theta_{CH}) \quad \text{(I-2)}$$

A simple program has been developed in *Matlab* to do a parametric study of the required heatsink with respect to the input voltage. The used parameters are illustrated in Table I-2. The ADIS16375 IMU slave's current consumption was measured using a voltometer and determined to be 680 mA, approximately. This slave has significantly higher power consumption, supplied by the LM1085IT regulator, since its IMU operates at 3.3 V, compared to the ADIS16364 that operates at 5 V. The aforementioned current value refers to both of the slave's supply channels. Since only the ESC is supplied by the 5 V channel, its current consumption (350 mA [43]) was substracted from the specified total one, to come with the 3.3 V channel's current consumption (330 mA). All the other parameters in the aforementioned table are specified by the respective manufacturer's datasheet. Note that the chosen heatsink was not soldered at the LDO, but rather mounted on it along with thermal paste to ease off the heat dissipation.

Table I-2. Matlab study's used parameters.

Parameter	Value
$T_{ambient}$ [°C]	25
$T_{J,max}$ [°C]	125
$V_{in,range}$ [V]	7-15.85
V_{out} [V]	3.3
I_G [A]	0.005
I_L [A]	0.330
θ_{JC} [°C / W]	15.6
θ_{CH} [°C / W]	0.2
$\theta_{HA,actual}$ [°C / W] (Chosen Heatsink)	16

In Figure I-1, the formula in (I-2) is materialized. Heatsinks with thermal resistances inside the green area are valid for the corresponding input voltage ranges.

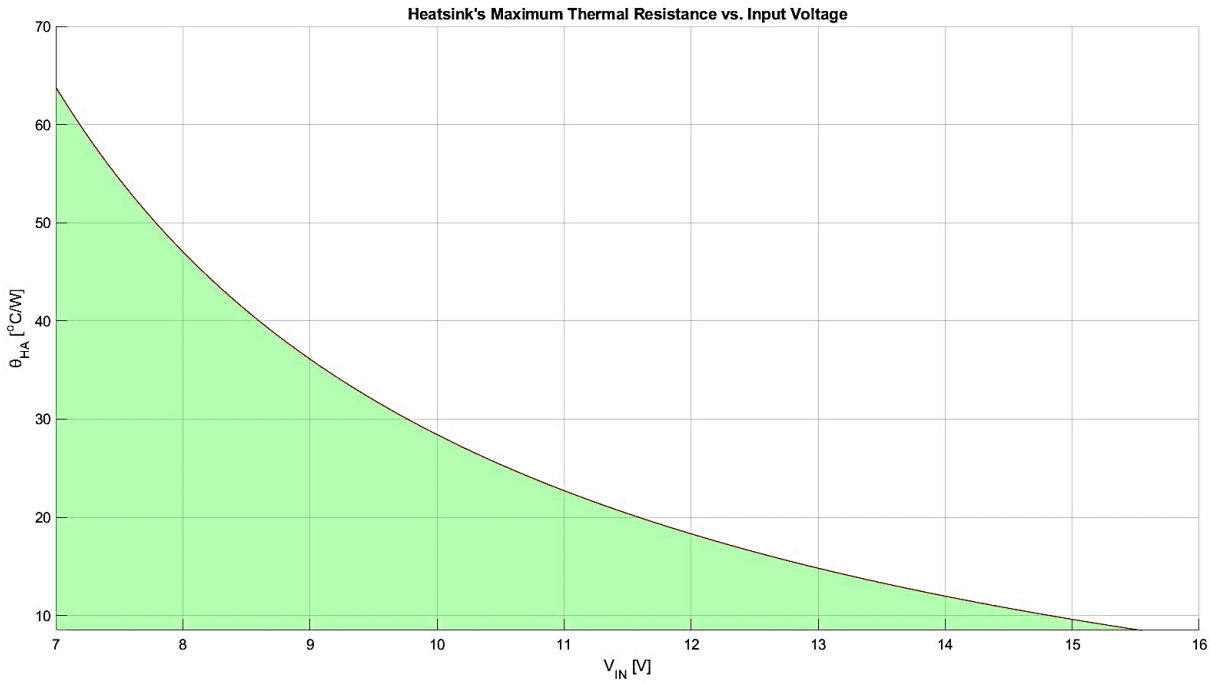


Figure I-1. LDO's maximum allowable thermal resistance.

So, for the chosen heatsink (16 ^{O}C/W) the temperature characteristic w.r.t. the input voltage is illustrated in Figure I-2. This characteristic curve's intersection with the 125 ^{O}C temperature limit, specifies the maximum allowable input voltage to be approximately 12.64 V. The aforementioned curve is given in (I-3) ,specified in [77].

$$T_J = T_A + P_D\left(\theta_{JC} + \theta_{CH} + \theta_{\mathrm{HA}}\right) \tag{I-3}$$

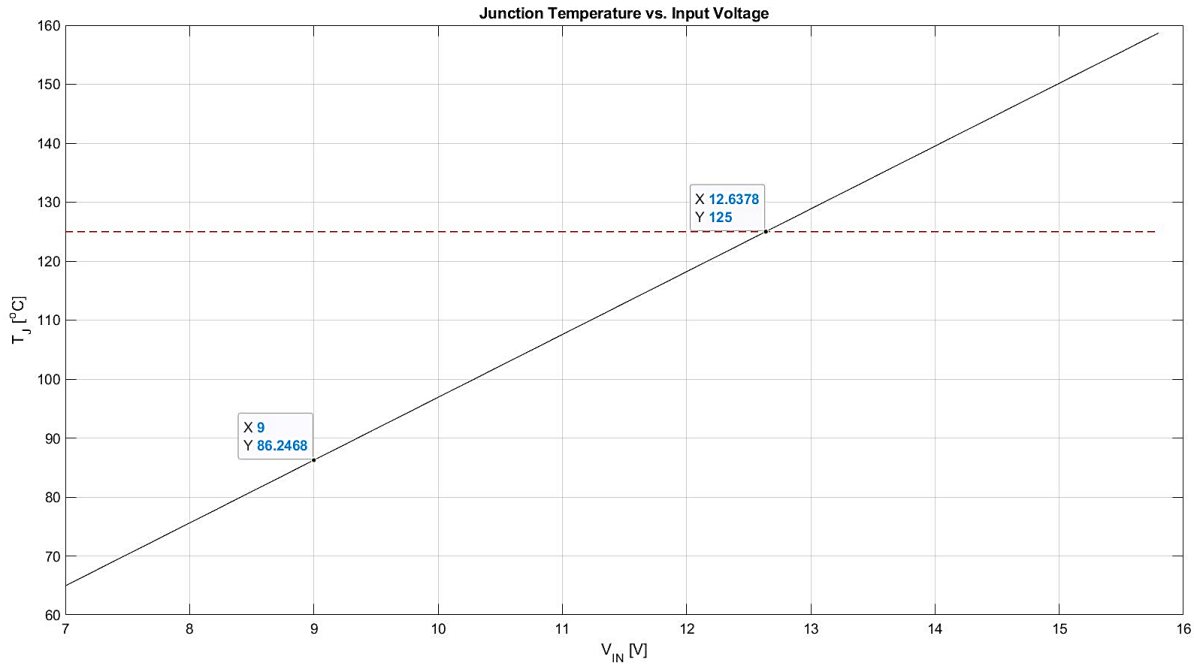


Figure I-2. LDO thermal behavior with chosen heatsink.

Note that the described approach is based on a bulk thermal model. To verify that claim, an experiment was conducted. The LM1085IT output voltage was monitored using Agilent's *MSOX3014A* oscilloscope for a variety of different input voltages. The results indicated that the LDO's thermal protection is activated for input voltages higher than 12.7 V.

To conclude with, the described analysis has been verified experimentally and its results proved to be valid. The input voltage of the IMU EtherCAT slaves must not pass the limit of 12.7 V. So, by taking into account the fact that Laelaps' low-power supply channel operates at 9 V, the overall design is on the safe side. Generally, the LDO thermal stresses rise as the input voltage rises.

Appendix J. Dynamical Modeling of Pendula

This Appendix is focused on the derivation of the mathematical formulas that describe the simple and bifilar pendulum's behavior. An effort has been made to avoid the exhausting reproduction of every calculation that is necessary, especially in the bifilar's case, in which a significant amount of computations arise.

J.1 Simple Rigid-Body Pendulum Dynamics

The pendulum constitutes one of the simplest, yet utterly insightful nonlinear system (Figure J-1). In this study, only 2D motion is considered. There are multiple ways to derive the equations of motion of such a system, but in this letter, Lagrange formulation has been chosen, due to its elegance and universal applications.

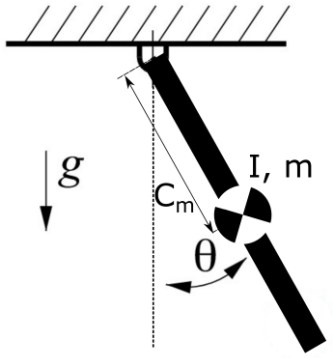


Figure J-1. Rigid-body pendulum overview.

To apply the Lagrange theory, first, the energy equations (J-1) of the system should be formed. The equations' notations are given in Figure J-1.

$$T = \frac{1}{2} I\omega^2 + \frac{1}{2} mv^2 = \frac{1}{2}(I + mC_m^2)\dot{\theta}^2$$
$$U = -mgC_m \cos(\theta) \tag{J-1}$$

The Lagrangian is given as $L = T - U$, while the EoMs are presented in (J-2).

$$\frac{d}{dt}\left[\frac{\partial L}{\partial \dot{q}}\right] - \frac{\partial L}{\partial q} + \frac{\partial P_C}{\partial \dot{q}} = \frac{\partial P_t}{\partial \dot{q}}, \text{ with } q = \theta \text{ as the generalized coordinate} \tag{J-2}$$

In the current case, no Rayleigh friction terms and external excitations are considered. From the equations (J-1)-(J-2) the resulting equation of motion is given in the equation (J-3).

$$\left(I + mC_m^2\right)\ddot{\theta} + mgC_m \sin\left(\theta\right) = 0 \tag{J-3}$$

To come with a unified formula that presents the direct dependency among the inertial parameters and the period of oscillation, the equation needs to be linearized around a stable point. Such procedure outputs the following ODE, which constitutes a simple initial value problem (J-4).

$$\left(I + mC_m^2\right)\ddot{\theta} + mgC_m \cdot \theta = 0, \text{ with } \theta(0) = \theta_0 \;\; \& \;\; \dot{\theta}(0) = 0 \Rightarrow$$
$$\theta(t) = \theta_0 \cos\left(\sqrt{\frac{mgC_m}{I + mC_m^2}} \cdot t\right) \tag{J-4}$$

From the above equations (J-4), the inertia (I) may be retrieved by measuring the period of oscillation and applying the equation (J-5).

$$I = \frac{mgC_m T^2}{4\pi^2} - mC_m^2 \tag{J-5}$$

J.2 Bifilar Rigid Body Pendulum Dynamics

The bifilar pendulum (Figure J-2) is a different flavor of the systems belonging to the pendula family. Its linearization is as simple and straight-forward as the simple pendulum's, but the derivation is trickier. In this section, the general methodology is illustrated, but the reader is encouraged to consult this Mathematica notebook [52] for a detailed derivation.

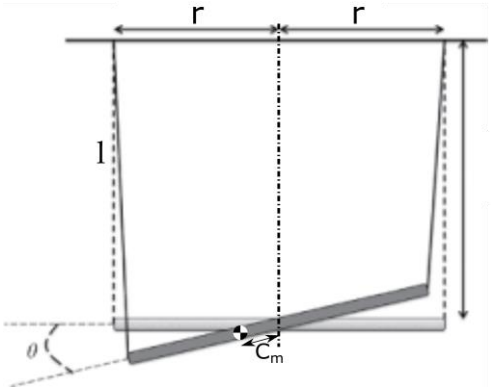


Figure J-2. Rigid-body bifilar pendulum overview.

Like in the previous case, equations (J-6) give the energy of the system. In this case, the additional notation z represents the vertical elevation of the rod.

$$T = \frac{1}{2}mv^2 + \frac{1}{2}I\omega^2 = \frac{1}{2}\left(I + mC_m^2\right)\dot{\theta}^2 + \frac{1}{2}m\dot{z}^2$$
$$U = mgz \tag{J-6}$$

In the above formulas (J-6), the only independent coordinate is θ and is chosen as the generalized coordinate of the Lagrangian. To express z as a function of the angle θ certain geometrical handling is necessary. Figure J-3 illustrates the process (J-7) graphically.

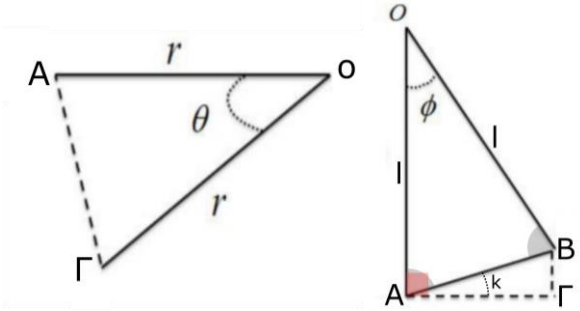


Figure J-3. Bifilar pendulum's geometry of motion.

From the graphs in Figure J-3, the equation (J-7) may be expressed as in (J-9).

$$\left(\mathrm{AB}\right)=2l\sin\left(\frac{\varphi}{2}\right)\ \&\ \left(\mathrm{A\Gamma}\right)=2r\sin\left(\frac{\theta}{2}\right) \tag{J-7}$$

$$\begin{gathered}\left(\mathrm{A\Gamma}\right)=\left(\mathrm{AB}\right)\cos\left(k\right),\ \text{with}\ k=\frac{\pi}{2}-\frac{\pi-\varphi}{2}=\frac{\varphi}{2}\Rightarrow\\ 2r\sin\left(\frac{\theta}{2}\right)=l\sin\left(\varphi\right)\Rightarrow\sin\left(\varphi\right)=\frac{2r}{l}\sin\left(\frac{\theta}{2}\right)\end{gathered} \tag{J-8}$$

$$\begin{gathered}z=\left(\mathrm{B\Gamma}\right)=\left(\mathrm{AB}\right)\sin(k)=\left(\mathrm{AB}\right)\sin\left(\frac{\varphi}{2}\right)\overset{\text{(J-7)}}{=}2l\sin^2\left(\frac{\varphi}{2}\right)\Rightarrow\\ z=l\left[1-\cos\left(\varphi\right)\right]\overset{\text{(J-8)}}{=}l\left[1-\sqrt{1-\frac{4r^2}{l^2}\sin^2\left(\frac{\theta}{2}\right)}\right]\end{gathered} \tag{J-9}$$

The linearized equation of motion is given in (J-10).

$$\begin{gathered}\left(I+mC_m^2\right)\cdot\ddot{\theta}+\frac{mgr^2}{l}\theta=0,\ \text{with}\ \theta(0)=\theta_0\ \ \&\ \ \dot{\theta}(0)=0\ \Rightarrow\\ \theta(t)=\theta_0\cos\left(\sqrt{\frac{mgr^2}{\left(I+mC_m^2\right)\cdot l}}\cdot t\right)\end{gathered} \tag{J-10}$$

From the above equations (J-10), the inertia (I) may be retrieved by measuring the period of oscillation and applying the equation (J-11).

$$I=\frac{mgr^2T^2}{4\pi^2l}-mC_m^2 \tag{J-11}$$

Appendix K. Bill Of Materials

K.1 Absolute Encoder Circuit

To interface a single absolute encoder with the Delfino shield PCB, the required components are listed in Table K-1.

Table K-1. Absolute encoder interface circuit.

Part	Quantity	Description
JST-XH 4-pin (Male)	x1	JST-XH Male Connector
JST-XH 4-pin (Female)	x1	JST-XH Female Cable
RJ9 (Female)	x1	RJ9 Female Connector
RJ9 Cable	x1	RJ9 Cable with Male Connectors
Abs. Enc. Housing	x1	Custom 3D-Printed Housing
RLS RMB28Vx Encoder	x1	RLS RMB28Vx Absolute Encoder Module
Prototyping PCB	x1	Prototyping PCB for rewiring signals

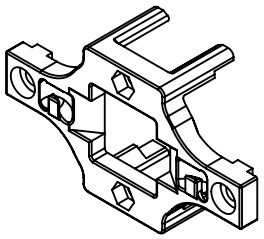
K.2 IMU-Delfino Interface PCB

The IMU-Delfino Interface PCB is a custom solution, developed in-house. The parts for building it, are listed in Table K-2.

Table K-2. IMU-Delfino PCB BOM.

Part	Quantity	Value	Description
C1, C2, C3, C4	x4	100 nF	Pol.Tantalum Capacitor
C5	x1	0.33 uF	Pol. Tantalum Capacitor
C6, C7	x2	10 uF	Pol. Tantalum Capacitor
DATA_READY_SELECTOR, IMU_SUPPLY_SELECTOR (JP2E)	x2	3-Pin	Jumper
DIO3/DIO4/RESET_JUMPER (JP1E)	x3	2-Pin	Jumper
IC1	x1	L7805CV	5V 1.5A Voltage Regulator STMicroelectronics
IMU_CONNECTOR	x1	EHT-108-01-S-D-SM	Samtec IDC Conn.
IMU IDC Cable	x1	TCSD-08-D-02.00-01-N-R	Samtec IDC Cable
J1	x1	DCJ0202	DC Power Jack
J3	x1	CONN_04x02	Male Pin Header
U1	x1	LM1085IT-12/NOPB	3.3V 3A Voltage Regulator TI

Appendix L. Schematics



SECTION A-A
SCALE 1 : 1

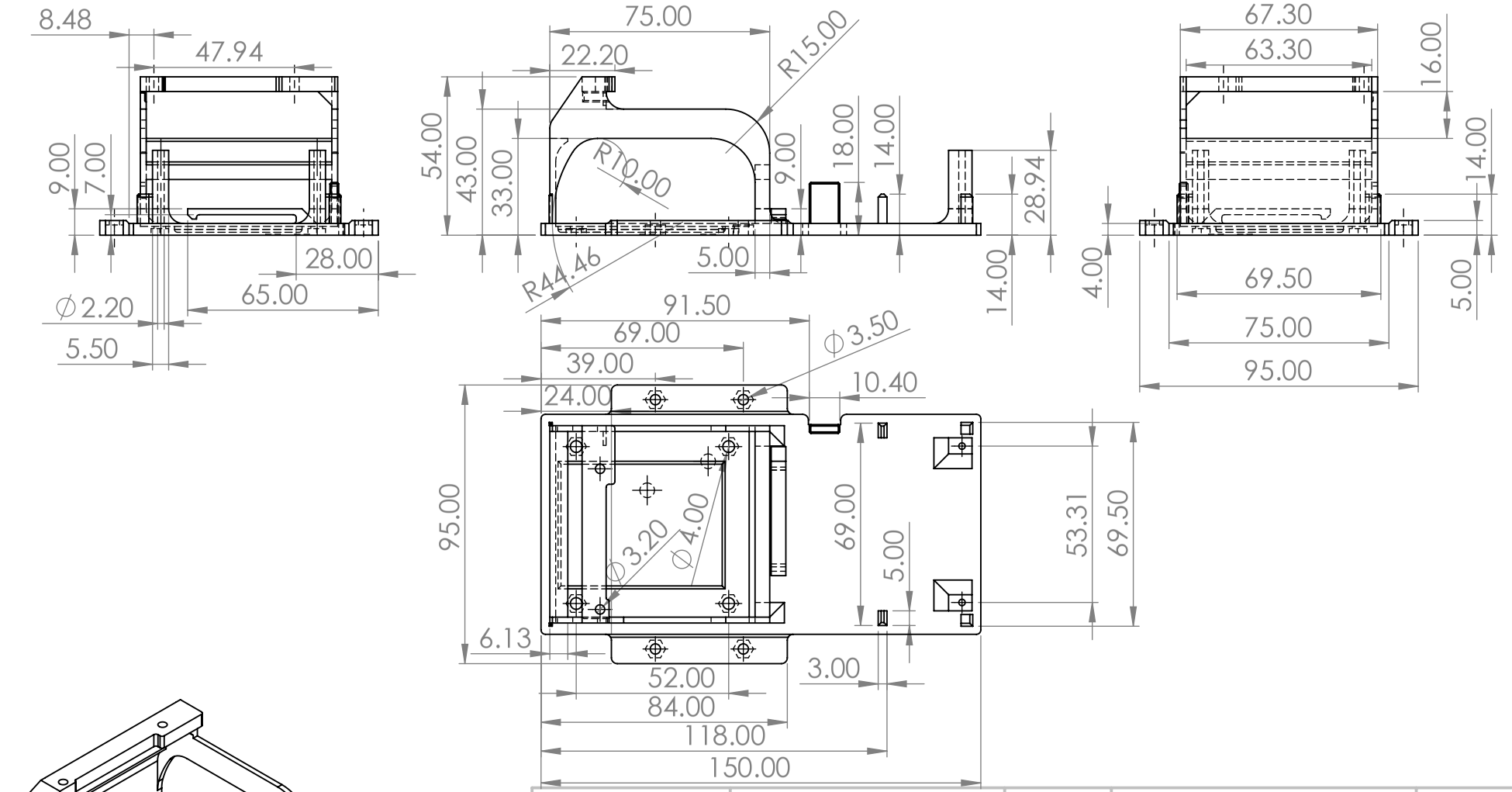
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:	DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING / REVISION

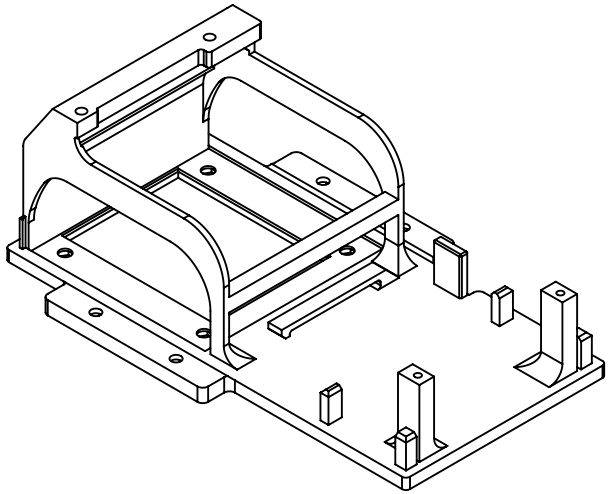
	NAME	SIGNATURE	DATE
DRAWN			
CHK'D			
APPV'D			
MFG			
Q.A			

TITLE:

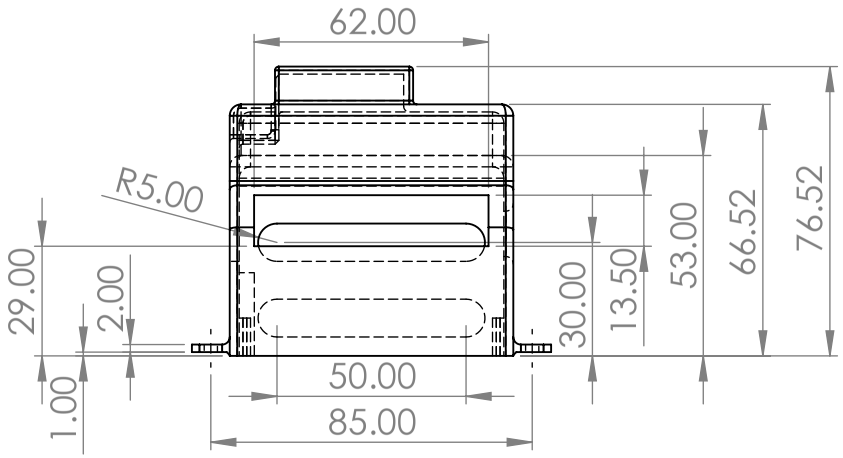
Absolute Encoder's Housing

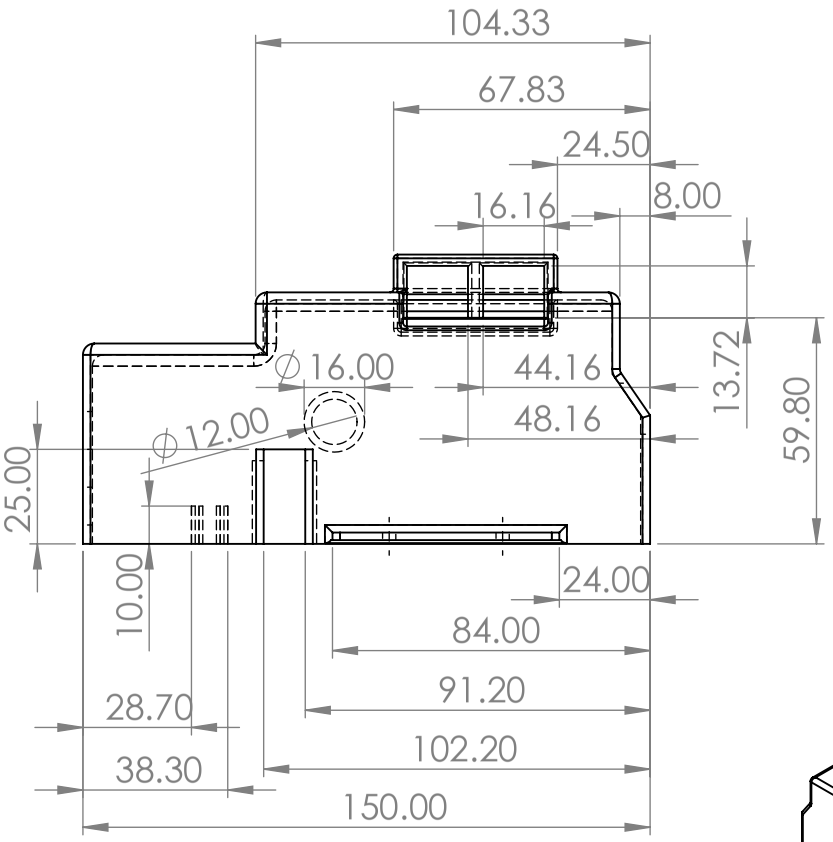
MATERIAL: ABS	DWG NO.	A4
WEIGHT:	SCALE:1:1	SHEET 1 OF 1

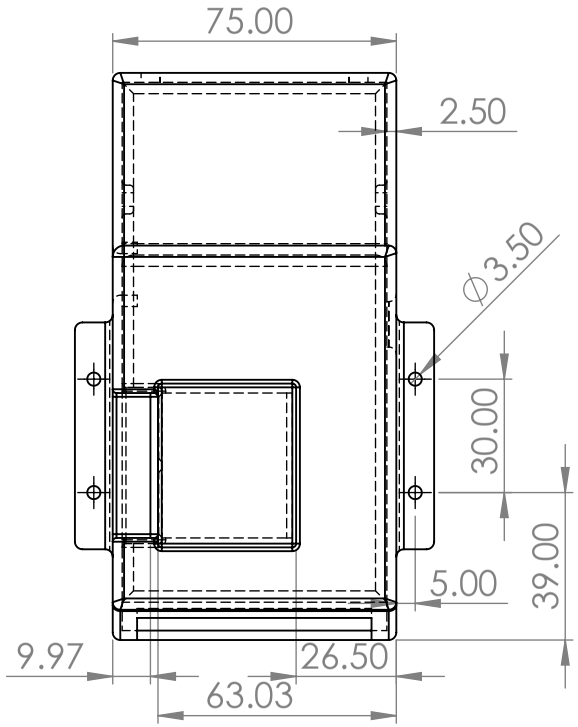


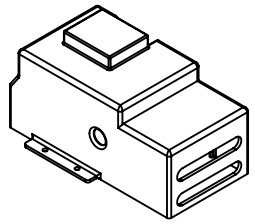


UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ± 0.2 ANGULAR:		FINISH:		DEBURR AND BREAK SHARP EDGES		DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN						IMU Housing Base	
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL:		DWG NO.	A4
				ABS			
				WEIGHT:		SCALE:1:2	SHEET 1 OF 1









UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: **±0.2** ANGULAR:		FINISH:		DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE		TITLE:	
DRAWN					IMU Housing Cover	
CHK'D						
APPV'D						
MFG						
Q.A				MATERIAL: ABS	DWG NO.	A4
				WEIGHT:	SCALE:1:2	SHEET 1 OF 1