Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων

Τομεας Σηματων Ελεγχου Και Ρομποτικης

Προγραμματισμός πραγματικού χρόνου τετράποδου ρομπότ σε δίκτυο EtherCAT μέσω ROS

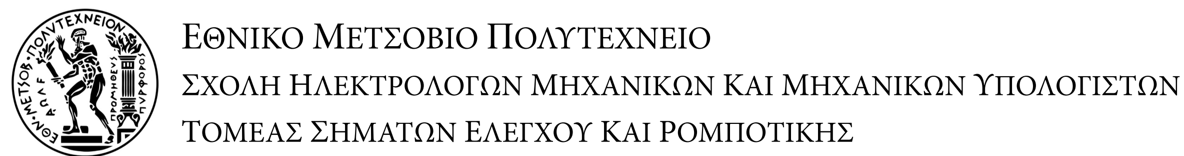
ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μιχαήλ Α. Καραμουσαδάκης

Επιβλέπων Καθηγητής: Κωνσταντίνος Τζαφέστας
Αν. Καθηγητής ΕΜΠ

Συνεπιβλέπων Καθηγητής: Ευάγγελος Παπαδόπουλος
Καθηγητής ΕΜΠ

Εργαστήριο Αυτομάτου Ελέγχου ΜΜ-ΕΠ
Αθήνα, Ιούλιος 2019



Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων
Τομεας Σηματων Ελεγχου Και Ρομποτικης

Προγραμματισμός πραγματικού χρόνου τετράποδου ρομπότ σε δίκτυο EtherCAT μέσω ROS

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μιχαήλ Α. Καραμουσαδάκης

Επιβλέπων Καθηγητής:	Κωνσταντίνος Τζαφέστας Αν. Καθηγητής ΕΜΠ
Συνεπιβλέπων Καθηγητής:	Ευάγγελος Παπαδόπουλος Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11η Ιουλίου 2019.

. . . .	. . . .	. . . .
Κωνσταντίνος Τζαφέστας Αν. Καθηγητής ΕΜΠ	Δημήτριος Σούντρης Καθηγητής ΕΜΠ	Ευάγγελος Παπαδόπουλος Καθηγητής ΕΜΠ

Εργαστήριο Αυτομάτου Ελέγχου ΜΜ-ΕΠ
Αθήνα, Ιούλιος 2019

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Signals, Control and Robotics

Real-time programming of EtherCAT master in ROS for a quadruped robot

DIPLOMA THESIS

Michail A. Karamousadakis

Control Systems Lab - EP
Athens, July 2019

. . . .

Μιχαήλ Α. Καραμουσαδάκης
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Copyright © Μιχαήλ Α. Καραμουσαδάκης, 2019
Με επιφύλαξη παντός δικαιώματος. All rights reserved.
Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η πρόοδος σε τεχνολογίες fieldbus, σε συστήματα πραγματικού χρόνου και προγραμματιστικά πλαίσια ρομποτικής, υπόσχονται ριζικό μετασχηματισμό των πεδίων της βιομηχανικής αυτοματοποίησης και της ρομποτικής. Δεδομένου ότι οι διεργασίες στη βιομηχανική ρομποτική υπόκεινται συνήθως χρονικούς περιορισμούς, η χρήση συστημάτων πραγματικού χρόνου προσπαθεί να αξιοποιήσει την απόδοση και την ασφάλεια σε αυτά τα πολύ απαιτητικά και κρίσιμα για την ασφάλεια περιβάλλοντα. Στις τεχνολογίες fieldbus, το πρωτόκολλο EtherCAT ξεχωρίζει για τα πολυάριθμα πλεονεκτήματα που προσφέρει για hard και soft συστήματα πραγματικού χρόνου, μεταξύ των οποίων είναι οι σύντομοι χρόνοι ενημέρωσης δεδομένων, η χαμηλή μεταβλητότητα στην ποιότητα της επικοινωνίας και μειωμένο κόστος εξοπλισμού. Μεταξύ των ρομποτικών πλαισίων, το λειτουργικό σύστημα για ρομπότ (ROS) ξεχωρίζει για την επεκτασιμότητα του, την ευκολία εκμάθησης του και τη δημοτικότητα του στην κοινότητα ρομποτικής. Αυτή η διπλωματική εργασία στοχεύει στο σχεδιασμό και την ανάπτυξη μιας εφαρμογής λογισμικού, η οποία εξασφαλίζει συγχρονισμένη κίνηση των ποδιών ενός τετράποδου ρομπότ που ονομάζεται Laelaps II, που δημιουργήθηκε και αναπτύχθηκε στο εργαστήριο CSL-EP στο Εθνικό Μετσόβιο Πολυτεχνείο (ΕΜΠ). Συγκεκριμένα, οι κύριοι στόχοι του έργου είναι: (α) Δημιουργία ενός λειτουργικού συστήματος πραγματικού χρόνου (RTOS) που βασίζεται στο GNU / Linux, με στόχο την ικανοποίηση των σκληρών περιορισμών πραγματικού χρόνου που επιβάλλει η συγχρονισμένη κίνηση του Laelaps II. (β) Χρήση ενός EtherCAT Master που ονομάζεται EtherLab στο προαναφερθέν RTOS, για τον έλεγχο του δικτύου που αποτελείται από EtherCAT slaves, που είναι τοποθετημένοι σε κάθε πόδι του ρομπότ και τα ελέγχουν. (γ) Ανάπτυξη μιας Διεπαφής Προγραμματισμού Εφαρμογών ROS (API) για τη διευκόλυνση της επέκτασης και της διαλειτουργικότητας με λογισμικό στο ROS περιβάλλον. Η υλοποίηση του έργου, μέσω του συνδυασμού αυτών των στόχων, αξιολογήθηκε με δοκιμή της ικανότητας βάδισης του Laelaps II. Τα αποτελέσματα δείχνουν ότι τα πειράματα βάδισης ήταν επιτυχημένα και το συνολικό έργο θα μπορούσε να ικανοποιήσει τις απαιτήσεις για ένα βιομηχανικό τετράποδο ρομπότ όπως το Laelaps II. Τέλος, προτείνονται προτάσεις για βελτιώσεις όσον αφορά την προσέγγιση του σχεδίου και κατευθύνσεις για περαιτέρω διερεύνηση του θέματος.

Λέξεις-Κλειδιά

Συστήματα πραγματικού χρόνου, ρομποτική, τετράποδα ρομπότ, χρονοδρομολόγηση πραγματικού χρόνου, GNU/Linux, PREEMPT-RT επέκταση, ROS, EtherCAT, EtherLab.

Abstract

Advances in fieldbus technologies, real-time systems and robotics frameworks hold a promise for radical transformation of the industrial automation and robotics fields. Since industrial robotics usually are subject to timing constraints, utilization of real-time systems attempts to leverage performance and safety in these highly demanding and “safety-critical” environments. In fieldbus technologies, the EtherCAT protocol stands out for it’s numerous benefits for hard and soft real-time systems, including short data update times, low communication jitter and reduced hardware costs. Among robotics frameworks, the Robot Operating System (ROS) stands out for it’s high customization, extendability, modularity, ease of learning and popularity in the robotics community. The project in the context of this thesis aims to design and develop a software application, which will ensure synchronized motion of the legs of a quadruped robot called Laelaps II, created and developed at the CSL-EP lab, at the National Technical University of Athens (NTUA). Specifically, the main objectives of the project are: (i) Creation of a Real-Time Operating System (RTOS), with modification and configuration of an Operating System (OS) based on GNU/Linux, aiming to meet the hard real-time constraints which Laelaps II synchronized motion imposes. (ii) Utilization of an EtherCAT master called EtherLab, in the aforementioned RTOS, for controlling the network consisting of EtherCAT slaves, which are placed at each leg of the robot and control them; (iii) Development of a ROS Application Programming Interface (API) for facilitating extendability, usability, maintainability and inter-operability with software (to be) written in the ROS environment. The implementation of the project, through the combination of these objectives, was evaluated by testing the trotting ability of Laelaps II. The results show that the trotting experiments were succesfull and that the overall project can meet the requirements for an industrial quadruped robot like Laelaps II. Finally, suggestions for improvements regarding the project’s approach and directions for further investigation on this topic are proposed.

Keywords

Real-time systems, robotics, quadruped, laelaps, real-time scheduling, software, GNU/Linux, Linux kernel, PREEMPT-RT patch, ROS, EtherCAT, EtherLab, real-time.

Στην Φλώρα και στην Ειρήνη

Αντί Προλόγου

Στο σημείο αυτό θα ήθελα να εκφράσω την ευγνωμοσύνη μου προς τους ανθρώπους που συνέδραμαν στην ολοκλήρωση αυτής της διπλωματικής εργασίας, αλλά και στην ευρύτερη ακαδημαϊκή μου πορεία. Αρχικά, χρωστώ την μεγάλη μου ευγνωμοσύνη προς τον πανάγαθο Θεό, που «συνεργεί εις παν έργον αγαθόν». Έπειτα, θα ήθελα να ευχαριστήσω τον καθηγητή κ. Ευάγγελο Παπαδόπουλο, για την αμέριστη συμπαράσταση και κατανόηση σε όλη την πορεία εκπόνησης της διπλωματικής μου. Ακόμη, θέλω να ευχαριστήσω τους υποψήφιους διδάκτορες Κώστα Μαχαιρά, Θανάση Μαστρογεωργίου και Κώστα Κουτσούκη που υπομονετικά με βοήθησαν στο θεωρητικό και πειραματικό μέρος της διπλωματικής. Επίσης, ευχαριστώ θερμά τον διδάκτορα κ. Βαγγέλη Κούκη που με ενέπνευσε και με καθοδήγησε να ασχοληθώ με το σημείο τομής του τομέα των Υπολογιστικών Συστημάτων με τον τομέα της Ρομποτικής. Στάθηκε αρωγός κάθε στιγμή εκπόνησης της διπλωματικής και έδωσε πολύτιμες και καίριες συμβουλές για την ολοκλήρωση του πειραματικού μέρους, και για αυτό το λόγο έχει την ευγνωμοσύνη μου. Επιπλέον, ευχαριστώ από καρδιάς την οικογένειά μου και τους αγαπημένους μου φίλους για τη στήριξη, την κατανόηση και την ανεκτίμητη συντροφιά τους. Τέλος, θέλω να εκφράσω την ευγνωμοσύνη μου προς όσες και όσους υποστηρίζουν έμπρακτα την ελεύθερη και δωρεάν διακίνηση της γνώσης, αναφέροντας χαρακτηριστικά τους προγραμματιστές ελεύθερου λογισμικού.

Μιχαήλ Καραμουσαδάκης
Απρίλιος 2019

Contents

Περίληψη **iii**

Abstract **iv**

Αντί Προλόγου **vii**

List of figures **xiii**

List of tables **xviii**

List of code blocks **xix**

List of Acronyms **xxiii**

Εκτενής Ελληνική Περίληψη **1**

Εισαγωγή 1

Σκοπός & Κίνητρο 1

Υπάρχουσες Προσεγγίσεις 2

Υπόβαθρο 4

Συστήματα Πραγματικού Χρόνου 4

GNU / Linux και Πραγματικός Χρόνος 4

Λειτουργικό Σύστημα για Ρομπότ (ROS) 6

Το πρωτόκολλο EtherCAT 6

O EtherLab Master 7

Σχεδιασμός & Υλοποίηση 9

Συνιστώσα Λογισμικού 9

Πειραματική Αξιολόγηση . . . 12
Αποτελέσματα . . . 12
Επίλογος . . . 19
Συμπεράσματα . . . 19
Μελλοντικές Δυνατότητες . . . 20

1 Introduction **23**
1.1 Problem Statement . . . 23
1.2 Literature Review . . . 24
1.2.1 Legged Robots Overview . . . 24
1.2.2 Fieldbus Systems Overview . . . 25
1.2.3 EtherCAT Robotic Applications Overview . . . 29
1.2.4 Real-time Systems Overview . . . 30
1.2.5 Real-Time Operating Systems Overview . . . 31
1.2.6 ROS 2 Overview . . . 34
1.3 Benefits . . . 36
1.4 Thesis Structure . . . 37

2 Background in Real-Time & ROS **39**
2.1 Real-time Systems Concepts . . . 39
2.1.1 General Concepts . . . 39
2.2 Real-time Task Scheduling . . . 41
2.3 Real-time GNU/Linux . . . 45
2.3.1 The PREEMPT_RT Patch . . . 46
2.4 Real-time Scheduling in GNU/Linux . . . 55
2.4.1 The first in, first out policy . . . 56
2.4.2 The round-robin policy . . . 57
2.4.3 The deadline policy . . . 57
2.4.4 The normal policy . . . 59
2.4.5 The batch policy . . . 59
2.4.6 The idle policy . . . 59
2.5 Robot Operating System (ROS) . . . 60
2.5.1 Components of ROS . . . 60
2.5.2 Basic ROS Terminology . . . 61
2.5.3 Message Communication in ROS . . . 65

3 Background in EtherCAT **71**

3.1 EtherCAT Technology 71

3.1.1 EtherCAT characteristics 71

3.1.2 Physical Layer 72

3.1.3 Data Link Layer 73

3.1.4 Application Layer (AL) 77

3.1.5 Distributed Clocks 79

3.1.6 Synchronization in the Slaves 81

3.1.7 Synchronization in the Master 87

3.2 EtherCAT Masters 89

3.2.1 EtherCAT Masters Overview 89

3.2.2 The IgH EtherCAT Master for GNU/Linux (EtherLab) 95

4 Requirements Analysis & Technical Specifications **105**

4.1 Requirements Analysis 105

4.1.1 Laelaps II 105

4.1.2 User Categories 106

4.1.3 Functional Requirements 107

4.1.4 Non-functional Requirements 107

4.2 Technical Specifications 108

4.2.1 Design Choices 108

4.2.2 System Architecture 110

4.2.3 Application Programming Interface 134

5 Implementation **139**

5.1 Software Implementation 139

5.2 Installation Process 160

5.2.1 The Preempt_RT Patch 160

5.2.2 EtherLab 165

5.3 Configuration & Optimization 168

5.3.1 Isolating the Application 168

5.3.2 Full Dynamic Ticks 170

5.3.3 Optimizing the Partitioned System 171

6 Experimental Evaluation **175**
6.1 Tools, Methodology & Environment 175
6.1.1 Building the application 176
6.1.2 Starting the EtherLab module 177
6.1.3 Slaves Initialization 178
6.1.4 Launching the application 180
6.1.5 Monitoring 181
6.2 Experiments & Results 181
6.2.1 Experiments 182
6.2.2 Results 185

7 Conclusions & Future Work **195**
7.1 Concluding Remarks 195
7.2 Future Work 196

Bibliography **199**

Appendices **211**
1 Appendix A 213
1.1 Final script 213

List of figures

1 Ρομπότ με πόδια της Boston Dynamics: (**a**) Handle, (**b**) SpotMini, (**c**) Atlas και (**d**) BigDog. 3

2 Ρομπότ αιχμής με πόδια, που βρίσκονται σε ερευνητικά ιδρύματα: (**a**) ANYmal, (**b**) Hermes, (**c**) Cheetah και (**d**) Inu. 3

3 Μία τυπική EtherCAT τοπολογία, με την "on-the-fly" επεξεργασία πλαισίων (frames) EtherCAT [1, Κεφάλαιο 38]. 7

4 Συνολική Αρχιτεκτονική του EtherLab[2]. 8

5 Συνολική Αρχιτεκτονική του Συστήματος. 9

6 Εσωτερική Αρχιτεκτονική της Μονάδας Λογισμικού. 10

7 Επιθυμητή ελλειπτική τροχιά όλων των άκρων των ποδιών (κόκκινο) μαζί με την πραγματική τους απόκριση (μαύρα) σε σχέση με τα συστήματα αναφοράς που βρίσκονται στις αρθρώσεις ισχίων των ποδιών. 13

8 Επιθυμητή απόκριση των γωνιών των γονάτων (κόκκινο) και πραγματική απόκριση των αρθρώσεων των γονάτων (μαύρο). 14

9 Επιθυμητή απόκριση των γωνιών των ισχίων (κόκκινο) και πραγματική απόκριση των αρθρώσεων των ισχίων (μαύρο). 15

10 Εντολές PWM του κινητήρα γονάτου κάθε ποδιού (μαύρο) και τα αντίστοιχα προκαθορισμένα όρια PWM (κόκκινο). 16

11 Εντολές PWM του κινητήρα ισχίου κάθε ποδιού (μαύρο) και τα αντίστοιχα προκαθορισμένα όρια PWM (κόκκινο). 17

12 Εκτίμηση ταχύτητας της άρθρωσης γονάτου κάθε ποδιού (μαύρο) και τα αντίστοιχα προκαθορισμένα όρια ταχύτητας του κινητήρα (κόκκινο). 18

13 Εκτίμηση ταχύτητας της άρθρωσης ισχίου κάθε ποδιού (μαύρο) και τα αντίστοιχα προκαθορισμένα όρια ταχύτητας του κινητήρα (κόκκινο). 19

1.1 Boston Dynamics legged robots: **(a)** Handle, **(b)** SpotMini, **(c)** Atlas, **(d)** BigDog. 25

1.2 State of the Art legged robots: **(a)** ANYmal, **(b)** Hermes, **(c)** Cheetah and **(d)** Inu. 26

1.3 **(a)** KR C4 Controller with robotic arm by KUKA and **(b)** MiniBOT Robot by NexCom. 29

1.4 Shadow Dexterous Hand by Shadow Rob Company 30

1.5 **(a)** Talos biped robot by PAL Robotics, **(b)** HyQ2Max quadruped robot by IIT and **(c)** ANYmal robot from ETH. 30

2.1 Spectrum of real-time systems. 40

2.2 A task model [3]. 42

2.3 Priority order of execution in ksoftirqd thread [4]. 48

2.4 Interrupt inversion [5]. 49

2.5 Threaded interrupt handling [5]. 50

2.6 Two paths by which softirqs run [4]. 51

2.7 A priority inversion example [6]. 52

2.8 A priority inheritance example [6]. 53

2.9 The usual task model of a real-time task defined with the Linux deadline-class parameters. 58

2.10 The ROS Meta-Operating System [7]. 61

2.11 ROS Components [7]. 62

2.12 Message Communication between Nodes [7]. 65

2.13 Topic Message Communication [7]. 66

2.14 Service Message Communication [7]. 67

2.15 Action Message Communication [7]. 68

2.16 Message Communication [7]. 69

3.1 EtherCAT typical topology, with the on-the-fly frame processing [1, Chapter 38]. 72

3.2 EtherCAT Frame Structure [8]. 74

3.3 EtherCAT datagram structure [9]. 75

3.4 EtherCAT Slave State Machine [8]. 78

3.5 Offset measurement in the DC mechanism [10]. 81

3.6 Concept of the TCL algorithm [11]. 81

3.7 EtherCAT Application Level [12]. 82

3.8 EtherCAT process data exchange [12]. 82

3.9 Time between Master and Slave Application [12]. 82

3.10 Slave in Free Run mode [12]. 83

3.11 EtherCAT network in Free Run mode [13]. 84

3.12 Slave in SM Synchronous mode [12]. 84

3.13 EtherCAT network in SM Synchronous mode [13]. 85

3.14 Slave in DC Synchronous mode [12]. 85
3.15 EtherCAT network in DC Synchronous mode [13]. 86
3.16 EtherCAT shift time [14]. 87
3.17 Acceptable vs wrong shift times [14]. 88
3.18 Master synchronized to DC Base [15]. 89
3.19 Pseudo-code of a typical EM control loop [16, Chapter 18]. 90
3.20 EtherCAT control loop timing diagram [16, Chapter 18]. 90
3.21 EtherLab Master Architecture [2]. 96
3.22 Multiple masters in one module [2]. 97
3.23 Master phases and transitions [2]. 97
3.24 Field Memory Management Unit (FMMU) Configuration [2]. 99
3.25 Master Configuration [2]. 100

4.1 Laelaps II. 106
4.2 Overall System Architecture. 111
4.3 A Use-Case Diagram for the Operator. 112
4.4 Internal architecture of the software project. 113
4.5 Synchronization scheme followed in the software project. 115
4.6 Sending Path anatomy. 118
4.7 Receiving Path anatomy. 120
4.8 Actual and virtual links of Laelaps II legs [17]. 123
4.9 The leg's model [17]. 123
4.10 The leg's workspace [17]. 124

4.11 Different positions along the semi-elliptical trajectory [17]. 125

4.12 EtherCAT Process Data handling in the slaves [17]. 129

4.13 EtherCAT slave software architecture [17]. 129

4.14 The EtherCAT slave MCU [17]. 130

4.15 The EtherCAT slave ESC [17]. 130

4.16 EtherCAT Control Tower Assembly [17]. 131

4.17 EtherCAT Control Tower Assembly on Laelaps II [17]. 132

4.18 Electrical System of Laelaps II [17]. 133

5.1 The PREEMPT_RT kernel configuration option using `menuconfig`. 162

6.1 The PC/104 computer. 176

6.2 Reset button to initialize legs' pose [17]. 178

6.3 Laelaps II on treadmill ready to perform experiments [17]. 179

6.4 Laelaps' State Machine [17]. 179

6.5 Desired elliptical trajectory of all legs toe (red) along with their actual response (black) w.r.t coordinate systems located in the hip joints of the legs. . . 186

6.6 Desired response of knee angles (red) and actual response of knee joint (black). 187

6.7 Desired response of hip angles (red) and actual response of hip joint (black). . 188

6.8 PWM commands of each leg's knee motor (black) and the respective predefined PWM limits (red). 189

6.9 PWM commands of each leg's hip motor (black) and the respective predefined PWM limits (red). 190

6.10 Velocity estimation of each leg's knee joint (black) and the respective predefined motor speed limits (red). 191

6.11 Velocity estimation of each leg's hip joint (black) and the respective predefined motor speed limits (red). 192

List of tables

2.1 Hard real-time versus soft real-time systems [18]. 40

2.2 Comparison of the Topic, Server, and Action [7]. 65

3.1 Commercial versus Open-Source EMs [16, Chapter 18]. 92

3.2 EtherLab versus SOEM. 93

3.3 Application Interface Timing Comparison [2]. 103

4.1 EtherCAT Laelaps II Motion Control Output variables. 126

4.2 EtherCAT Laelaps II Motion Control Input variables. 127

4.3 ROS API of the software project. 134

6.1 Trotting Experiment parameters. 183

6.2 Parameters independent of EtherCAT application. 184

6.3 Parameters independent of EtherCAT application. 184

6.4 Configurations tested. 185

6.5 Frequency Experiment Results. 193

List of code blocks

5.1 The call to `mlockall()`. 140

5.2 The `EthercatSlave` class definition. 142

5.3 The `ecrt_slave_config_dc()` function declaration. 142

5.4 The `EthercatCommunicator` class definition. 143

5.5 The `EthercatCommunicator::init` method. 146

5.6 The `EthercatCommunicator::start` method. 147

5.7 The `EthercatCommunicator::run` method. 148

5.8 The `EthercatCommunicator::publish_raw_data` method. 153

5.9 The `PDOOutListener` class definition. 155

5.10 The `PDOInPublisher` class definition. 156

5.11 The `PDOInPublisher::pdo_raw_callback` method. 157

5.12 The `utilities::copy_process_data_buffer_to_buf` function. 159

5.13 Command for unzipping the kernel compressed archive. 161

5.14 Commands for patching the kernel. 161

5.15 The configuration options for building the kernel with PREEMPT_RT patch. 162

5.16 Steps for building the kernel with the PREEMPT_RT patch. 163

5.17 The `install_etherlab_patched.sh` script. 165

5.18 The `Makefile` for building EtherLab. 166

6.1 The `change_permissions_ether_ros.sh` script. 176

6.2 The `reinstall_e1000e_wo_throttling.sh` script. 177

6.3 The `make_rt_task_ether_ros.sh` script. 180

1 The final script for performing extra real-time optimizations. 213

List of Acronyms

API Application Programming Interface. iii, iv, xix, 5, 9–11, 20, 46, 93, 95, 102, 103, 105, 107, 109–111, 113, 114, 116, 117, 119, 121, 134, 140, 142, 143, 196

CLI Command Line Interface. 9, 11, 110, 111, 113, 114, 180

DC Distributed Clocks. xvi, 79–81, 83–89, 130, 138, 143, 145, 146, 148, 152

DMA Direct Memory Access. 118, 119, 121

ESC EtherCAT Slave Controller. xvii, 72, 76, 79, 80, 86, 128–131

FMMU Field Memory Management Unit. xvi, 77, 98, 99

FOSS Free and Open Source Software. 32, 36

IRQ Interrupt Request. 21, 49, 120, 121, 178, 197

ISR Interrupt Service Routine. 47–49, 86, 178

MCU Micro Controller Unit. xvii, 1, 24, 128–131, 196

MII Media Independent Inter face. 87

NIC Network Interface Controller. 96, 101, 118, 119, 121, 122, 177

NMI Non-maskable Interrupt. 174

PDI Process Data Image. 98, 99, 128

PDOs Process Data Objects. 10–12, 77, 98, 102, 111–116, 128, 138, 143, 145, 147, 148, 152, 154–156, 158, 160, 194

RTOS Real-Time Operating System. iii, iv, 5, 31–33, 36, 41, 45, 46, 91, 94

SDO Service Data Object. 98, 102

SPI Serial Peripheral Interface. 128, 130

ΕΛ/ΛΑΚ Ελεύθερο Λογισμικό/Λογισμικό Ανοιχτού Κώδικα. 2

Εισαγωγή

Σκοπός & Κίνητρο

Απαιτήσεις για τετράποδα ρομπότ όπως η υψηλή ταχύτητα, η μεγάλη επιτάχυνση και η ικανότητα να κάνουν κλειστές στροφές, επιβάλλουν σκληρούς περιορισμούς πραγματικού χρόνου στις μονάδες επεξεργασίας τους. Με ένα σύστημα κατανεμημένου ελέγχου, η χρήση του EtherCAT για το σχεδιασμό ενός σκληρού συστήματος πραγματικού χρόνου που αποτελείται από δικτυωμένες μονάδες επεξεργασίας αποτελεί μια ικανοποιητική επιλογή, αν και προκύπτει η ανάγκη προγραμματισμού σε πραγματικό χρόνο των κόμβων επικοινωνίας (master / slaves). Συνήθως, οι EtherCAT slaves (μονάδες επεξεργασίας για τον έλεγχο των ποδιών ή Μονάδες Μικροελεγκτή (MCU)) σε αυτό το είδος της διαμόρφωσης, έχουν ένα πολύ συγκεκριμένο καθήκον (έλεγχο των ποδιών) και αποτελούνται από, ειδικά σχεδιασμένους σύμφωνα με το εγχειρίδιο χρήσης του κατασκευαστή και ολοκληρωμένους στο υλισμικό EtherCAT Slave Controllers, επομένως δεν υπάρχει ανάγκη για επιπλέον εργασία σε ότι αφορά το θέμα του πραγματικού χρόνου. Ωστόσο, ο EtherCAT master μπορεί να υλοποιηθεί ως λύση λογισμικού, οπότε μια ανάγκη εμφανίζεται σε αυτή την περίπτωση για μια λύση πραγματικού χρόνου. Έτσι, ο πρωταρχικός στόχος αυτής της διπλωματικής είναι η σχεδίαση και η υλοποίηση μιας εφαρμογής που να χρησιμοποιεί έναν EtherCAT master πραγματικού χρόνου στο ROS σε GNU / Linux.

Ορισμένα από τα πλεονεκτήματα στην προσέγγιση αυτής της διπλωματικής είναι τα εξής:

- **Χρήση του EtherCAT:** Ως πρωτόκολλο επικοινωνίας δικτύου σε πραγματικό χρόνο, το EtherCAT διαθέτει μια μεγάλη κοινότητα χρηστών. Τα τελευταία χρόνια, έχει γίνει δημοφιλές στη ρομποτική κοινότητα και στα εργαστήρια ρομποτικής λόγω των πλεονεκτημάτων του.

- **Ενσωμάτωση στο ROS:** Στη ρομποτική, το λειτουργικό σύστημα για ρομπότ (ROS) αποτελεί ένα καθιερωμένο πλαίσιο. Το λογισμικό που ενσωματώνεται σε αυτό, έχει σημαντικά οφέλη, όπως έτοιμες βιβλιοθήκες, ταχεία πρωτοτυποποίηση, επεκτασιμότητα, δομοστοιχειωτό σχεδιασμό, τυποποίηση και υποστήριξη από την κοινότητα.

- **Λογισμικό στο GNU / Linux:** Δεν υπάρχουν πολλά να πούμε για τα πλεονεκτήματα

της ανάπτυξης λογισμικού στο GNU / Linux. Το γεγονός ότι μέχρι σήμερα αποτελεί ένα από τα μεγαλύτερα έργα Ελεύθερο Λογισμικό/Λογισμικό Ανοιχτού Κώδικα (ΕΛ/ΛΑΚ) παρέχει αξεπέραστα πλεονεκτήματα όπως δωρεάν πηγαίο κώδικα (GNU Public License) και μεγάλη κοινότητα χρηστών και προγραμματιστών.

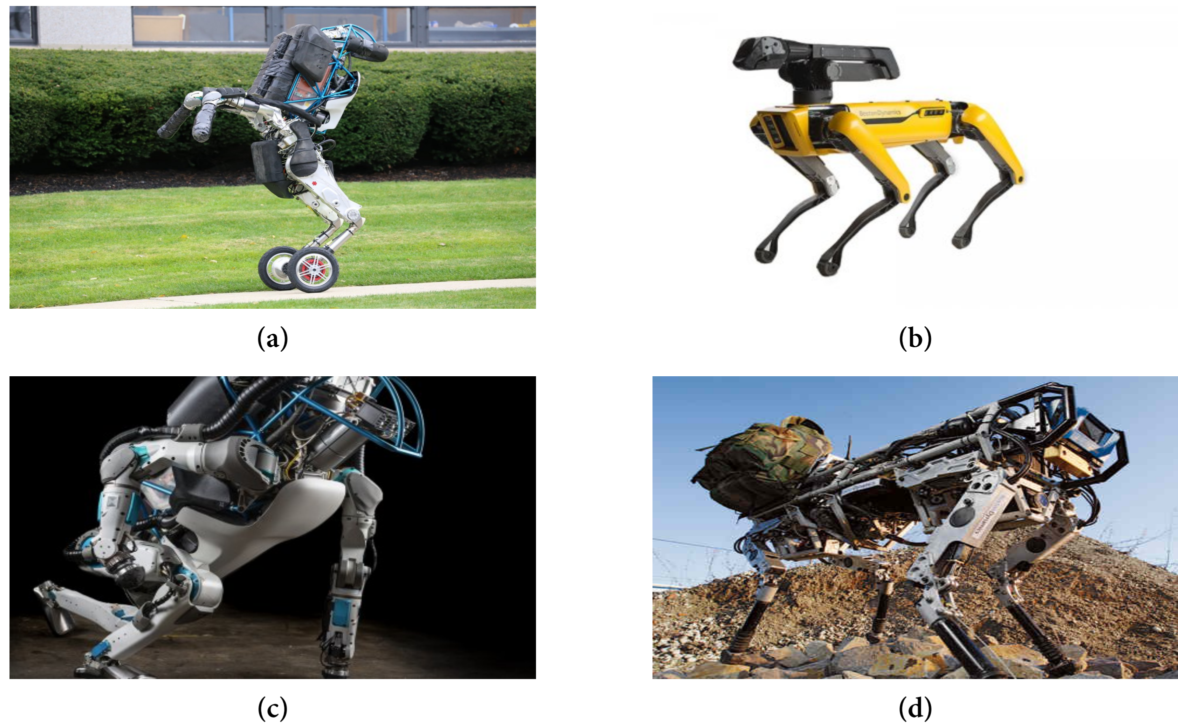
- **Λύση πραγματικού χρόνου:** Η προτεινόμενη προσέγγιση σχεδίασης χρησιμοποιεί ένα από τα μεγαλύτερα έργα στον κόσμο GNU / Linux πραγματικού χρόνου, συγκεκριμένα το patch PREMPT-RT. Τα οφέλη περιλαμβάνουν χαμηλό κόστος συντήρησης, σταθερότητα και μεγάλη κοινότητα για υποστήριξη και ανάπτυξη.

Υπάρχουσες Προσεγγίσεις

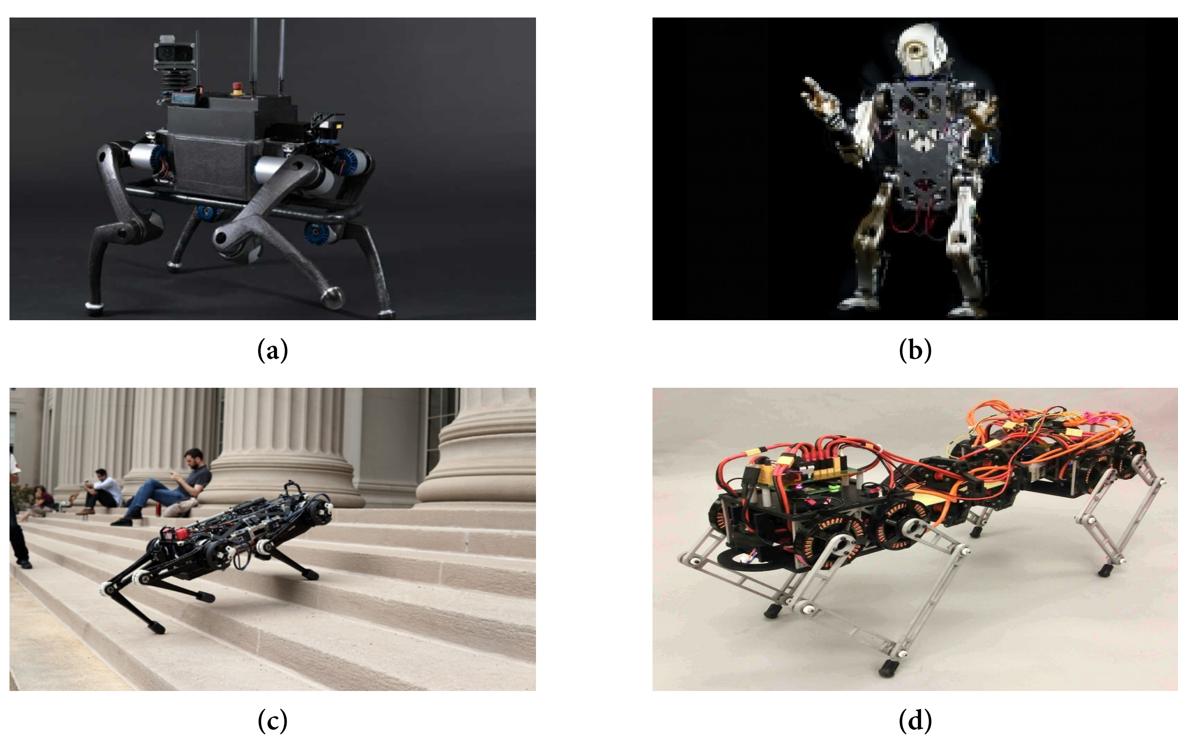
Έρευνα στα *ρομπότ με πόδια* έχει διεξαχθεί για πολλά θέματα, από το σχεδιασμό τους έως τη δυνατότητα ελέγχου τους, εξετάζοντας την ικανότητα τους για αυτόνομες, ημιαυτόνομες ή τηλεχειριζόμενες επιχειρήσεις σε δύσκολα εδάφη όπου τα οχήματα με τροχούς φτάνουν στα όριά τους. Τα μελλοντικά τετράποδα ρομπότ αναμένεται να λειτουργούν σε άκρως δυναμικούς, μη δομημένους υπαίθριους χώρους όπου θα περιηγούνται σε δύσπρόσιτα περιβάλλοντα, όπως καταρρέοντα κτίρια, καταστροφές, δάση, βουνά και εργοτάξια. Τα καθήκοντά τους θα κυμαίνονται από τη μετάδοση αναγνώσεων αισθητήρων στον απομακρυσμένο χειριστή (π.χ. κάμερες, LIDAR, υπέρυθρες ακτινοβολίες και επίπεδα ακτινοβολίας) μέχρι τη μεταφορά μεγάλων ωφέλιμων φορτίων όπως εργαλείων ή δομικών υλικών. Τα υπάρχοντα ρομπότ με πόδια αιχμής περιλαμβάνουν τα Handle Σχήμα 1(a), SpotMini Σχήμα 1(b), Atlas Σχήμα 1(c) και BigDog Σχήμα 1(d), σχεδιασμένα και κατασκευασμένα από την Boston Dynamics[1].

Τέλος, όσον αφορά τον ερευνητικό κόσμο, το ANYmal ρομπότ Σχήμα 2(a) από το Institute of Robotics and Intelligent Systems στο πανεπιστήμιο ETH της Ζυρίχης[2], το Hermes Σχήμα 2(b) και Cheetah Σχήμα 2(c), ρομπότ από το Biomimetic Robotics Lab στο MIT[3] και το Inu Σχήμα 2(d) ρομπότ από το KOD*LAB στο UPenn[4] είναι χαρακτηριστικά παραδείγματα ρομπότ με πόδια που αναπτύσσονται σε πανεπιστήμια.

[1] `https://www.bostondynamics.com`
[2] `http://www.rsl.ethz.ch/robots-media/anymal.html`
[3] `http://biomimetics.mit.edu`
[4] `https://kodlab.seas.upenn.edu`



Σχήμα 1: *Ρομπότ με πόδια της Boston Dynamics:* ***(a)*** *Handle,* ***(b)*** *SpotMini,* ***(c)*** *Atlas και* ***(d)*** *BigDog.*



Σχήμα 2: *Ρομπότ αιχμής με πόδια, που βρίσκονται σε ερευνητικά ιδρύματα:* ***(a)*** *ANYmal,* ***(b)*** *Hermes,* ***(c)*** *Cheetah και* ***(d)*** *Inu.*

Υπόβαθρο

Συστήματα Πραγματικού Χρόνου

> *Ένα σύστημα πραγματικού χρόνου είναι ένα σύστημα που πρέπει να ικανοποιεί ρητούς (περιορισμένους) περιορισμούς χρόνου απόκρισης ειδάλλως κινδυνεύει με σοβαρές συνέπειες, συμπεριλαμβανομένης της αποτυχίας [19].*

Συνεπώς, η ορθότητα της απόκρισης ενός τέτοιου συστήματος εξαρτάται όχι μόνο από το λογικό αποτέλεσμα αλλά και από τον χρόνο που παραδόθηκε. Τα συστήματα πραγματικού χρόνου διακρίνονται σε τρεις κατηγορίες [19]:

- Σκληρά (Hard): Σε σκληρά συστήματα πραγματικού χρόνου, η μη τήρηση μιας μόνο προθεσμίας οδηγεί σε πλήρη και καταστροφική αποτυχία του συστήματος.
- Σταθερά (Firm): Σε σταθερά συστήματα πραγματικού χρόνου, η μη τήρηση ορισμένων προθεσμιών δεν θα οδηγήσει σε πλήρη αποτυχία, αλλά η μη τήρηση περισσότερων από μερικές, οδηγεί σε πλήρη και καταστροφική αποτυχία του συστήματος.
- Μαλακά (Soft): Σε μαλακά συστήματα πραγματικού χρόνου, η απόδοση υποβαθμίζεται από την αδυναμία ικανοποίησης των περιορισμών του χρόνου απόκρισης.

GNU / Linux και Πραγματικός Χρόνος

Το GNU / Linux [20] σχεδιάστηκε και κατασκευάστηκε ως λειτουργικό σύστημα πολλαπλών χρηστών γενικού σκοπού, βασισμένο στο UNIX. Οι στόχοι ενός συστήματος πολλαπλών χρηστών είναι γενικά σε αντίθεση με τους στόχους της λειτουργίας πραγματικού χρόνου. Τα λειτουργικά συστήματα γενικού σκοπού ρυθμίζονται για να μεγιστοποιήσουν τη μέση απόδοση, μερικές φορές σε βάρος της καθυστέρησης, ενώ τα λειτουργικά συστήματα πραγματικού χρόνου επιχειρούν να τοποθετήσουν ένα άνω όριο στην καθυστέρηση, μερικές φορές σε βάρος της μέσης απόδοσης. Γενικά, υιοθετήθηκαν δύο σημαντικές προσεγγίσεις στο GNU / Linux όσον αφορά τον πραγματικό χρόνο:

- Η Co-Kernel προσέγγιση: Η παλαιότερη λύση που βρέθηκε για το Linux πραγματικού χρόνου ήταν η τοποθέτηση ενός μικρού πυρήνα πραγματικού χρόνου που τρέχει δίπλα-δίπλα με το Linux στο ίδιο υλισμικό. Σε αυτή την προσέγγιση συμπεριλαμβάνονται οι προσπάθειες των RTAI και Xenomai. Σε αυτήν την περίπτωση, όλες οι διακοπές

συσκευών (device interrupts) πρέπει να περάσουν από τον co-kernel προτού υποβληθούν σε επεξεργασία από τον κανονικό πυρήνα, έτσι ώστε το Linux να μην μπορεί ποτέ να τους αναβάλει, εξασφαλίζοντας έτσι προβλέψιμο χρόνο απόκρισης στην πλευρά του πραγματικού χρόνου. Επίσης, σε αυτή την περίπτωση, απαιτούνται συνήθως συγκεκριμένα API για την ανάπτυξη μιας εφαρμογής σε πραγματικό χρόνο.

- Η προσέγγιση πλήρους preemptible πυρήνα (Fully Preemptible Kernel): Αυτή η προσέγγιση ασχολείται με τη μετατροπή του Linux σε ένα πλήρες Λειτουργικό Σύστημα Πραγματικού Χρόνου (ΛΣΠΧ / RTOS). Αυτό συνεπάγεται ότι γίνονται αλλαγές στον πυρήνα του Linux που επιτρέπουν την εκτέλεση διαδικασιών σε πραγματικό χρόνο χωρίς να υπάρχει παρεμβολή από απρόβλεπτες ή unbounded δραστηριότητες από διαδικασίες που δεν είναι πραγματικού χρόνου. Το Real-Time Linux (RTL) Collaborative Project (RTL)[5] είναι η πιο σχετική λύση ανοιχτού κώδικα για αυτήν την επιλογή [21]. Το έργο βασίζεται στην επέκταση PREEMPT_RT και στοχεύει στη δημιουργία ενός προβλέψιμου και ντετερμινιστικού περιβάλλοντος που μετατρέπει τον πυρήνα του Linux σε μια βιώσιμη πλατφόρμα πραγματικού χρόνου. Ο τελικός στόχος του έργου RTL είναι να περάσει την επέκταση PREEMPT_RT στον mainline πυρήνα. Η σημασία αυτής της προσπάθειας δεν σχετίζεται με τη δημιουργία ενός RTOS που βασίζεται στο Linux (αυτό έχει επιχειρηθεί ήδη αρκετές φορές), αλλά με την παροχή στον ίδιο τον πυρήνα του Linux, δυνατοτήτων πραγματικού χρόνου. Το κύριο όφελος είναι η δυνατότητα χρήσης των τυποποιημένων εργαλείων και βιβλιοθηκών του Linux χωρίς την ανάγκη ειδικών ΔΠΕ[6] πραγματικού χρόνου. Επίσης, το GNU / Linux χρησιμοποιείται και υποστηρίζεται ευρέως, γεγονός που βοηθά να διατηρηθεί το λειτουργικό σύστημα ενημερωμένο με νέες τεχνολογίες και χαρακτηριστικά, κάτι που αποτελεί συχνά πρόβλημα σε μικρότερα έργα λόγω περιορισμών πόρων.

Έχοντας αυτά υπόψη, η επέκταση PREEMPT_RT επιλέχθηκε ως ο καλύτερος υποψήφιος για την ανάπτυξη αυτής της εφαρμογής πραγματικού χρόνου. Αξίζει να σημειωθεί ότι, όπως και η περίφημη συζήτηση Torvalds / Tanenbaum σχετικά με την απαξίωση των μονολιθικών πυρήνων [22], στο GNU / Linux υπήρξε μακρά σειρά συζητήσεων σχετικά με διάφορες πτυχές των επιλογών σχεδιασμού πυρήνα του Linux. Ένα από τα πιο αμφιλεγόμενα θέματα ήταν η ερώτηση σχετικά με τον τρόπο προσθήκης επεκτάσεων πραγματικού χρόνου στον πυρήνα Linux [23].

---

[5] `https://wiki.linuxfoundation.org/realtime/rtl/start`

[6] Διεπαφή Προγραμματισμού Εφαρμογών (API)

Λειτουργικό Σύστημα για Ρομπότ (ROS)

> *Το ROS είναι ένα μετα-λειτουργικό σύστημα ανοιχτού κώδικα για το ρομπότ σας. Παρέχει τις υπηρεσίες που θα περιμένατε από ένα λειτουργικό σύστημα, συμπεριλαμβανομένης της αφαίρεσης υλικού, του ελέγχου των συσκευών χαμηλού επιπέδου, της εφαρμογής κοινώς χρησιμοποιούμενων λειτουργιών, της μετάδοσης μηνυμάτων μεταξύ των διαδικασιών και της διαχείρισης των πακέτων. Παρέχει επίσης εργαλεία και βιβλιοθήκες για την απόκτηση, κατασκευή, συγγραφή και εκτέλεση κώδικα σε πολλούς υπολογιστές*[7].

Το ROS σημαίνει λειτουργικό σύστημα για ρομπότ [7], οπότε θα περίμενε κανείς ότι το ROS είναι ένα ακόμα παραδοσιακό λειτουργικό σύστημα που στοχεύει σε συγκεκριμένες ρομποτικές πλατφόρμες. Αυτό όμως δεν ισχύει και η συντομογραφία δεν βοηθά στην επίλυση αυτής της σύγχυσης. Ένας ακριβέστερος ορισμός είναι ότι το ROS είναι ένα μετα-λειτουργικό σύστημα. Ο όρος αυτός περιγράφει ένα σύστημα που παρέχει λειτουργίες όπως η διαχείριση διαδικασιών, ο προγραμματισμός, η παρακολούθηση, η διαχείριση μνήμης, ο χειρισμός σφαλμάτων, οι πρωταρχικές μορφές επικοινωνίας και η λειτουργικότητα, χρησιμοποιώντας ένα επίπεδο εικονικοποίησης μεταξύ εφαρμογών και κατανεμημένων υπολογιστικών πλατφορμών, ενώ τρέχει πάνω από ένα παραδοσιακό λειτουργικό σύστημα. Αυτός ο τύπος λογισμικού ονομάζεται επίσης middleware ή πλαίσιο λογισμικού.

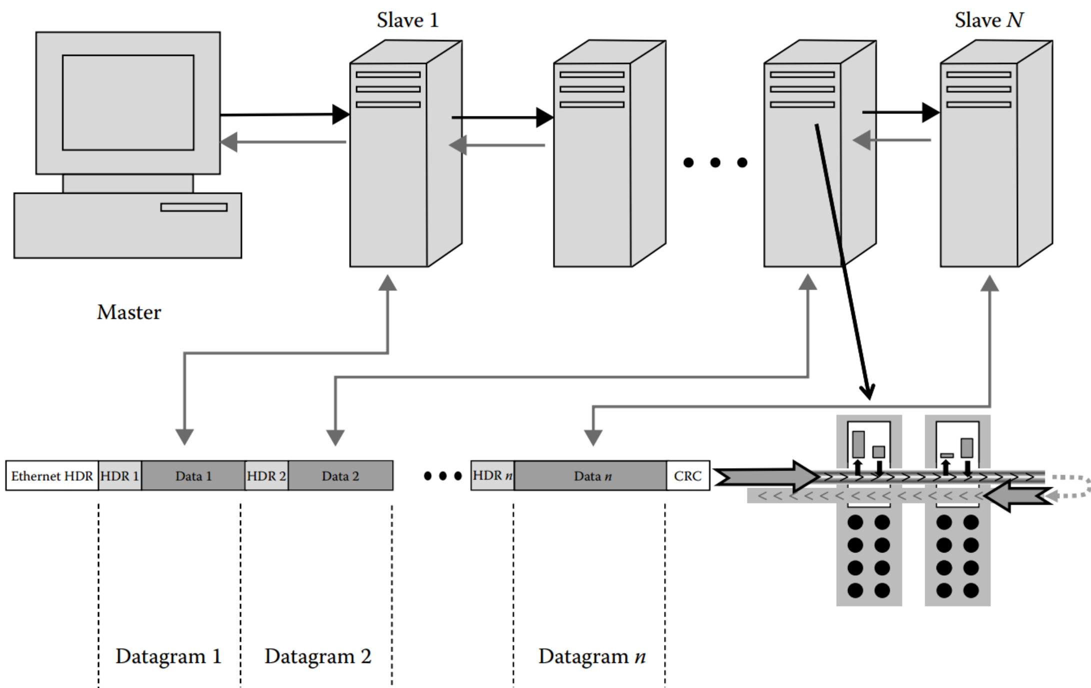
Το πρωτόκολλο EtherCAT

Το πρωτόκολλο EtherCAT βασίζεται σε μια προσέγγιση master / slave και στηρίζεται σε μια τοπολογία δακτυλίων στο φυσικό επίπεδο [17]. Μόνο ένας master επιτρέπεται στο δίκτυο (μπορούν να συνδεθούν πολλαπλοί master μέσω διακόπτη, αλλά μόνο ένας μπορεί να υπάρχει σε κάθε υποδίκτυο που ο διακόπτης ορίζει) και αυτό είναι κατάλληλο, για παράδειγμα, για να συνδέεται μια μονάδα ελέγχου (π.χ. PLC) με αποκεντρωμένα περιφερειακά (αισθητήρες, ενεργοποιητές, μηχανισμοί κίνησης κ.λπ.). Χρησιμοποιώντας κατάλληλες πύλες, το EtherCAT μπορεί να διαλειτουργεί τόσο με συμβατικά πρωτόκολλα δικτύωσης υπολογιστών (TCP / IP στοίβα) όσο και με άλλες λύσεις Ethernet (RTE) πραγματικού χρόνου, όπως EtherNet / IP ή/και PROFINET.

Ο κύριος κόμβος έχει τον πλήρη έλεγχο της κυκλοφορίας που ανταλλάσσεται μέσω του δικτύου EtherCAT. Συγκεκριμένα, είναι η μόνη συσκευή που μπορεί να αναλάβει την πρωτο-

[7] `https://www.ros.org/`

βουλία στην επικοινωνία. Ως εκ τούτου, είναι υπεύθυνη για την έναρξη όλων των ανταλλαγών δεδομένων με τις υποτελείς μονάδες. Κάθε υποτελής μονάδα επεξεργάζεται το ληφθέν πλαίσιο (frame) για να εξάγει / εισάγει δεδομένα από / μέσα της. Στη συνέχεια, το πλαίσιο (frame) μεταφέρεται στον επόμενο υποτελή κόμβο του δακτυλίου, όπως απεικονίζεται στο Σχήμα 3.



Σχήμα 3: *Μία τυπική EtherCAT τοπολογία, με την "on-the-fly" επεξεργασία πλαισίων (frames) EtherCAT [1, Κεφάλαιο 38].*

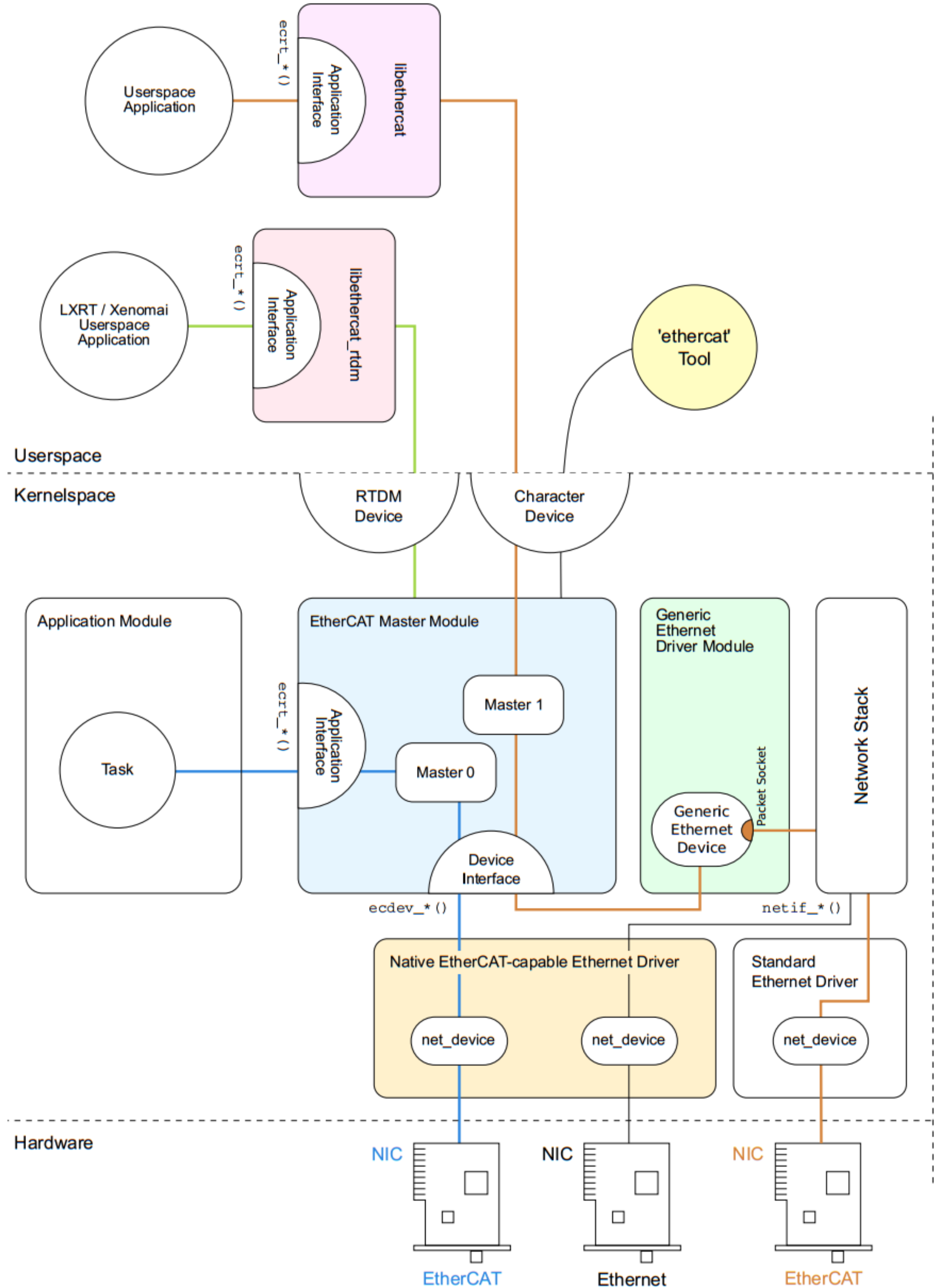
Ο EtherLab Master

Δεδομένου ότι το λογισμικό EtherLab έχει επιλεγεί ως ο EtherCAT master που θα επικοινωνεί η προς ανάπτυξη εφαρμογή, η αρχιτεκτονική του παρουσιάζεται συνοπτικά στο Σχήμα 4.

Τα συστατικά του περιβάλλοντος του master περιγράφονται παρακάτω:

- **Μονάδα Master**: Μονάδα πυρήνα που περιέχει ένα ή περισσότερα EtherCAT master στιγμιότυπα, τη διασύνδεση EtherCAT συσκευών και την διεπαφή εφαρμογής.

- **Μονάδες Συσκευών**: Μονάδες οδηγού συσκευής Ethernet με δυνατότητα υποστήριξης του πρωτοκόλλου EtherCAT, που προσφέρουν τις συσκευές τους στον EtherCAT master μέσω της EtherCAT Διεπαφής Συσκευής. Αυτά τα τροποποιημένα προγράμματα οδήγησης δικτύου μπορούν να χειριστούν συσκευές δικτύου που χρησιμοποιούνται για λειτουργία σε δίκτυο EtherCAT και "κανονικές" Ethernet συσκευές παράλ-



Σχήμα 4: *Συνολική Αρχιτεκτονική του EtherLab[2].*

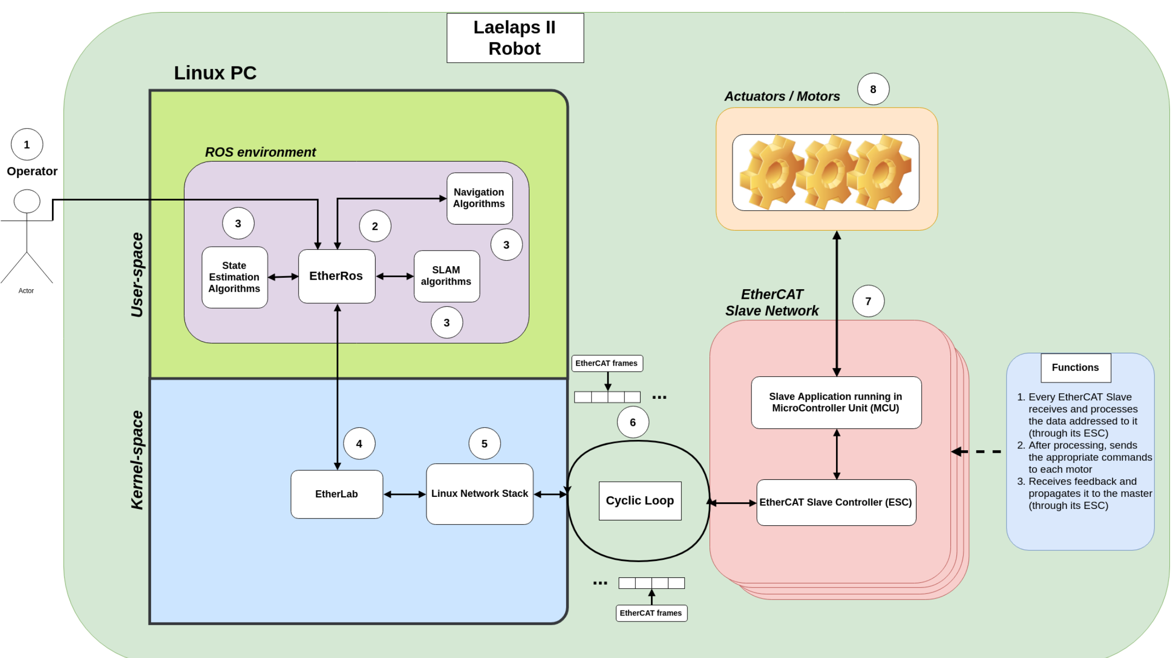
ληλα. Ένας master μπορεί να δεχτεί μια συγκεκριμένη συσκευή και έπειτα μπορεί να στείλει και να λάβει πλαίσια (frames) EtherCAT. Οι συσκευές Ethernet που απορρίφθηκαν από τον EtherCAT master είναι συνδεδεμένες στη στοίβα δικτύου του πυρήνα όπως συνήθως.

- **Εφαρμογή:** Ένα πρόγραμμα που χρησιμοποιεί τον EtherCAT master (συνήθως για κυ-

κλική ανταλλαγή δεδομένων διεργασίας με EtherCAT slaves). Αυτά τα προγράμματα δεν αποτελούν μέρος του EtherCAT master κώδικα, αλλά πρέπει να δημιουργούνται ή να γράφονται από το χρήστη. Μια εφαρμογή μπορεί να ζητήσει ένα master μέσω της Διεπαφής Εφαρμογής. Αν αυτό επιτύχει, έχει τον έλεγχο του master: Μπορεί να παρέχει δεδομένα διαμόρφωσης διαύλου και ανταλλαγής δεδομένων. Οι εφαρμογές μπορούν να είναι μονάδες πυρήνα (που χρησιμοποιούν απευθείας τη Διεπαφή Εφαρμογής πυρήνα, kernelspace) ή προγράμματα χώρου χρήστη (userspace), που χρησιμοποιούν τη Διεπαφή Εφαρμογής μέσω της βιβλιοθήκης EtherCAT ή της βιβλιοθήκης RTDM, όπως φαίνεται και στο Σχήμα 4.

Σχεδιασμός & Υλοποίηση

Στο Σχήμα 5 παρουσιάζεται ένα βασικό διάγραμμα ανάπτυξης μαζί με τα βασικά στοιχεία και τις συνδέσεις τους. Αυτό το διάγραμμα περιγράφει διαισθητικά τα στοιχεία του συνολικού συστήματος και παρέχει μια γενική εικόνα του ρομπότ με τον χειριστή του.

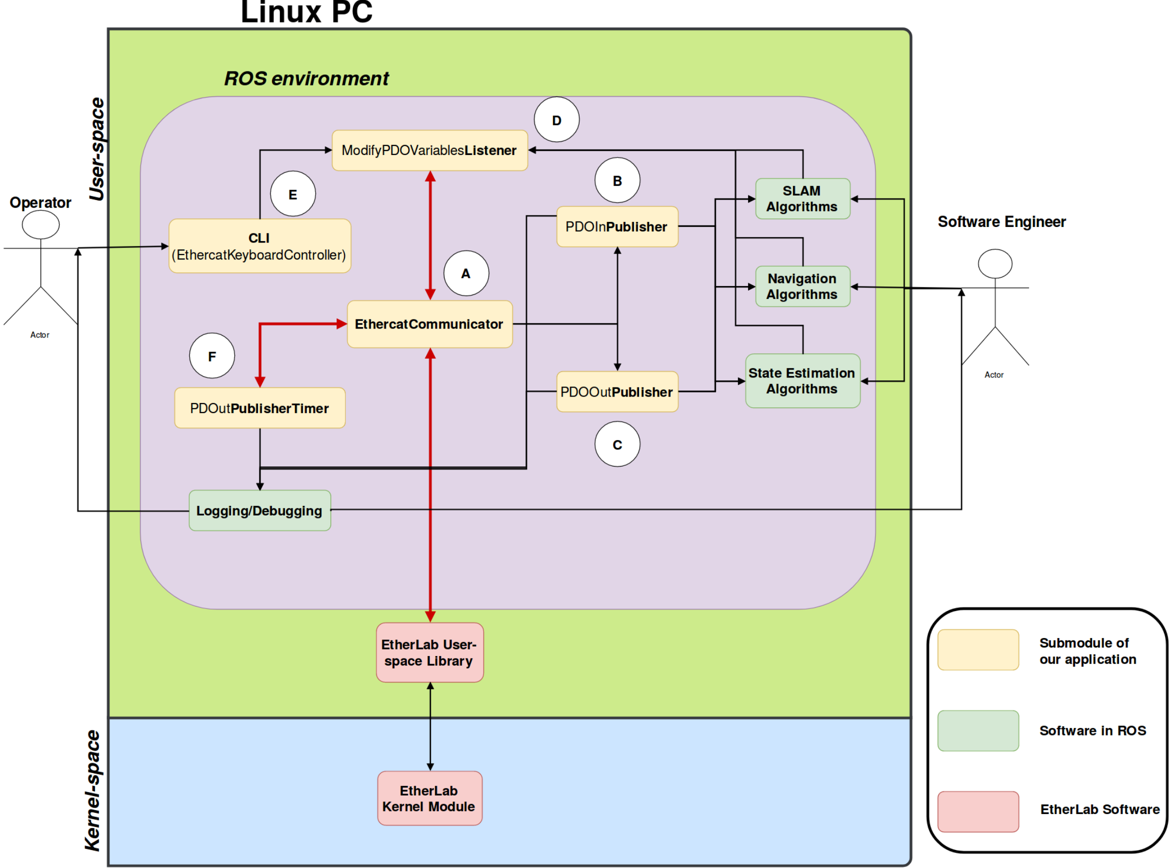


Σχήμα 5: *Συνολική Αρχιτεκτονική του Συστήματος.*

Συνιστώσα Λογισμικού

Η συνιστώσα λογισμικού του έργου, στα πλαίσια της διπλωματικής, παρουσιάζεται στο (2) (βλ. Σχήμα 5) μαζί με τις υψηλότερου επιπέδου συνδέσεις της. Το λογισμικό αυτό προσφέρει ένα ROS API για ROS κόμβους που αναπτύχθηκαν από άλλους μηχανικούς λογισμικού, προσφέρει Διεπαφή Γραμμής Εντολών (CLI) στον χειριστή και επικοινωνεί με τη μονάδα πυ-

ρήνα EtherLab για την επίτευξη της EtherCAT επικοινωνίας. Αυτή η συνολική συμπεριφορά επιτυγχάνεται μέσω διαφορετικών υπομονάδων, που απεικονίζονται στο Σχήμα 6.



Σχήμα 6: *Εσωτερική Αρχιτεκτονική της Μονάδας Λογισμικού.*

Στο (A) στο Σχήμα 6, απεικονίζεται η βασική υπομονάδα του λογισμικού, που έχει το όνομα EtherCAT Communicator. Αυτή η υπομονάδα στην βάση της αποτελείται από ένα νήμα (thread) που εκτελείται σε πραγματικό χρόνο και καλεί το API της EtherLab βιβλιοθήκης για χώρο χρήστη, το οποίο με τη σειρά του κάνει μια κλήση συστήματος στη μονάδα πυρήνα EtherLab που επικοινωνεί με τους EtherCAT slaves. Αυτή η υπομονάδα χρησιμοποιεί τη βιβλιοθήκη pthread για τη δημιουργία ενός νήματος πραγματικού χρόνου και για τη χρήση ενός pthread spinlock. Εφαρμόζει μια μηχανή καταστάσεων, την οποία υλοποιεί σε ένα πλαίσιο πραγματικού χρόνου στη συχνότητα βρόχου ελέγχου EtherCAT ($\geq$ 2 KHz).

Στο (B) (βλ. Σχήμα 6), επισημαίνεται η υπομονάδα του Input Process Data Objects (PDOs) Publisher. Αυτό το μέρος του λογισμικού λαμβάνει τα αντικείμενα δεδομένων εισαγωγής (τις μεταβλητές EtherCAT που μεταβάλλουν οι slaves και μεταβιβάζουν στον master) από το δίκτυο EtherCAT μέσω του EtherCAT Communicator και τις δημοσιεύει σε ένα **topic** στο ROS, στην συχνότητα βρόχου ελέγχου EtherCAT ($\geq$ 2 KHz). Ακόλουθα, οι κόμβοι ROS

που εφαρμόζουν αλγόριθμους ρομποτικής όπως SLAM, πλοήγηση και εκτίμηση κατάστασης, μπορούν να λάβουν αυτά τα δεδομένα και να τα επεξεργαστούν αναλόγως.

Στο (C) (βλ. Σχήμα 6), επισημαίνεται η υπομονάδα του Output Process Data Objects (PDOs) Publisher. Αυτό το μέρος του λογισμικού λαμβάνει τα αντικείμενα δεδομένων διεργασίας εξόδου (τις μεταβλητές EtherCAT που μεταβάλλονται από τους κόμβους του ROS ή τον χειριστή και μεταβιβάζονται στο master Ethernet σε δίκτυο EtherCAT μέσω του EtherCAT Communicator και εκδίδει σε ένα **topic** στο ROS, στη συχνότητα βρόχου ελέγχου EtherCAT ($\geq$ 2KHz). Συνεπώς, οι κόμβοι ROS που εφαρμόζουν αλγόριθμους ρομποτικής όπως SLAM, πλοήγηση και εκτίμηση κατάστασης, μπορούν να λάβουν αυτά τα δεδομένα και να τα επεξεργαστούν αναλόγως. Ανακύπτει ένα ερώτημα σχετικά με το γιατί αυτά τα Output PDOs θα πρέπει να δημοσιεύονται στο οικοσύστημα ROS, αφού πιθανότατα μεταβάλλονται από έναν κόμβο στο πλαίσιο του ROS. Η απάντηση είναι ότι αυτά τα δεδομένα θα μπορούσαν να ενδιαφέρουν περισσότερους από έναν κόμβους ROS (εκτός από πιθανώς αυτόν που τα αλλάζει), έτσι ώστε οι άλλοι κόμβοι να έχουν πρόσβαση σε αυτές τις αλλαγές. Ένας άλλος λόγος είναι ότι τα δεδομένα θα μπορούσαν να αλλάζουν από τον χειριστή, όπως προαναφέρθηκε, έτσι ένας κόμβος ROS να μπορεί να γνωρίζει τις αλλαγές με την εγγραφή στο προαναφερθέν topic. Παρ' όλα αυτά, αυτό το υποσύνολο δημιουργήθηκε για να παρέχει πληρότητα μέσω του ROS API, ωστόσο αν το κόστος που εισάγεται είναι υπερβολικό, αυτό το υποσύνολο θα μπορούσε να απενεργοποιηθεί σε μελλοντικές εκδόσεις.

Στο (D) (βλ. Σχήμα 6), απεικονίζεται η υπομονάδα του Output Process Data Objects (PDOs) Listener. Αυτό το μέρος του λογισμικού ακούει σε ένα ROS **topic**, λαμβάνει τα (τροποποιημένα) αντικείμενα δεδομένων διεργασίας εξόδου (τις μεταβλητές EtherCAT που αλλάζουν από τον κύριο κόμβο και διαβιβάζονται στους EtherCAT υποτελείς κόμβους) απευθείας από το οικοσύστημα ROS ή έμμεσα από το δημιουργημένο CLI και τα διαβιβάζει στον EtherCAT Communicator για να σταλούν στο δίκτυο. Το (D) ολοκληρώνει έναν πρώτο κλειστό βρόχο ανατροφοδότησης που αποτελείται από το άμεσο οικοσύστημα ROS (άλλοι ROS κόμβοι που υλοποιούνται), το δίκτυο EtherCAT και την εφαρμογή, επιτρέποντας την επικοινωνία μεταξύ όλων αυτών των στοιχείων.

Στο (E) (βλ. Σχήμα 6), εμφανίζεται η υπομονάδα της διεπαφής γραμμής εντολών (CLI). Αυτό το μέρος του λογισμικού διευκολύνει τον χρήστη που είναι υπεύθυνος για τη συνολική λειτουργία και τη διαχείριση των λειτουργιών του ρομπότ (επίσης γνωστός ως Χειριστής / Operator), να αλληλεπιδρά με έναν απλό και αποτελεσματικό τρόπο με το δίκτυο των EtherCAT υποτελών κόμβων και να ελέγχει αποτελεσματικά τις συγχρονισμένες κινήσεις

των ποδιών. Επιπλέον, ο EtherCAT Communicator ενεργοποιείται ή απενεργοποιείται μέσω αυτής της υπομονάδας και επίσης οι μεταβλητές EtherCAT που στέλνει ο κύριος κόμβος (Output PDOs), μεταβάλλονται μέσω αυτής της υπομονάδας από τον Χειριστή / Operator.

Στο (F) (βλ. Σχήμα 6), παρουσιάζεται η υπομονάδα του Output Process Data Objects (PDOs) Publisher Timer (χρονοδιακόπτης). Αυτό το μέρος του λογισμικού σε συγκεκριμένα χρονικά διαστήματα (για αυτό το λόγο ονομάζεται χρονοδιακόπτης) αντιγράφει τα δεδομένα διεργασίας που αποστέλλονται από την αντίστοιχη δομή προσωρινής αποθήκευσης (buffer) και τα δημοσιεύει σε ένα ROS **topic**. Με αυτές τις δημοσιευμένες πληροφορίες, πραγματοποιείται μια έμμεση καταγραφή που αποτελεί μια γρήγορη εκκίνηση για την αποσφαλμάτωση της συμπεριφοράς της μονάδας λογισμικού. Ως εκ τούτου, το (F) ολοκληρώνει ένα δεύτερο κλειστό κύκλωμα ανάδρασης, αποτελούμενο από τους χρήστες που διαχειρίζονται τη λειτουργία του ρομπότ, το οικοσύστημα ROS, τη μονάδα λογισμικού και το δίκτυο EtherCAT. Ωστόσο, αυτός ο δεύτερος κλειστός βρόχος είναι σίγουρα πιο χαλαρός και έμμεσος από τον πρώτο, υπό την έννοια ότι υπάρχει ο ανθρώπινος παράγοντας στη μέση, πράγμα που σημαίνει ότι πρέπει να υπάρξει διαχείριση και παρακολούθηση από ένα χρήστη ώστε να αναλάβει δράση και να κλείσει αυτόν το βρόχο.

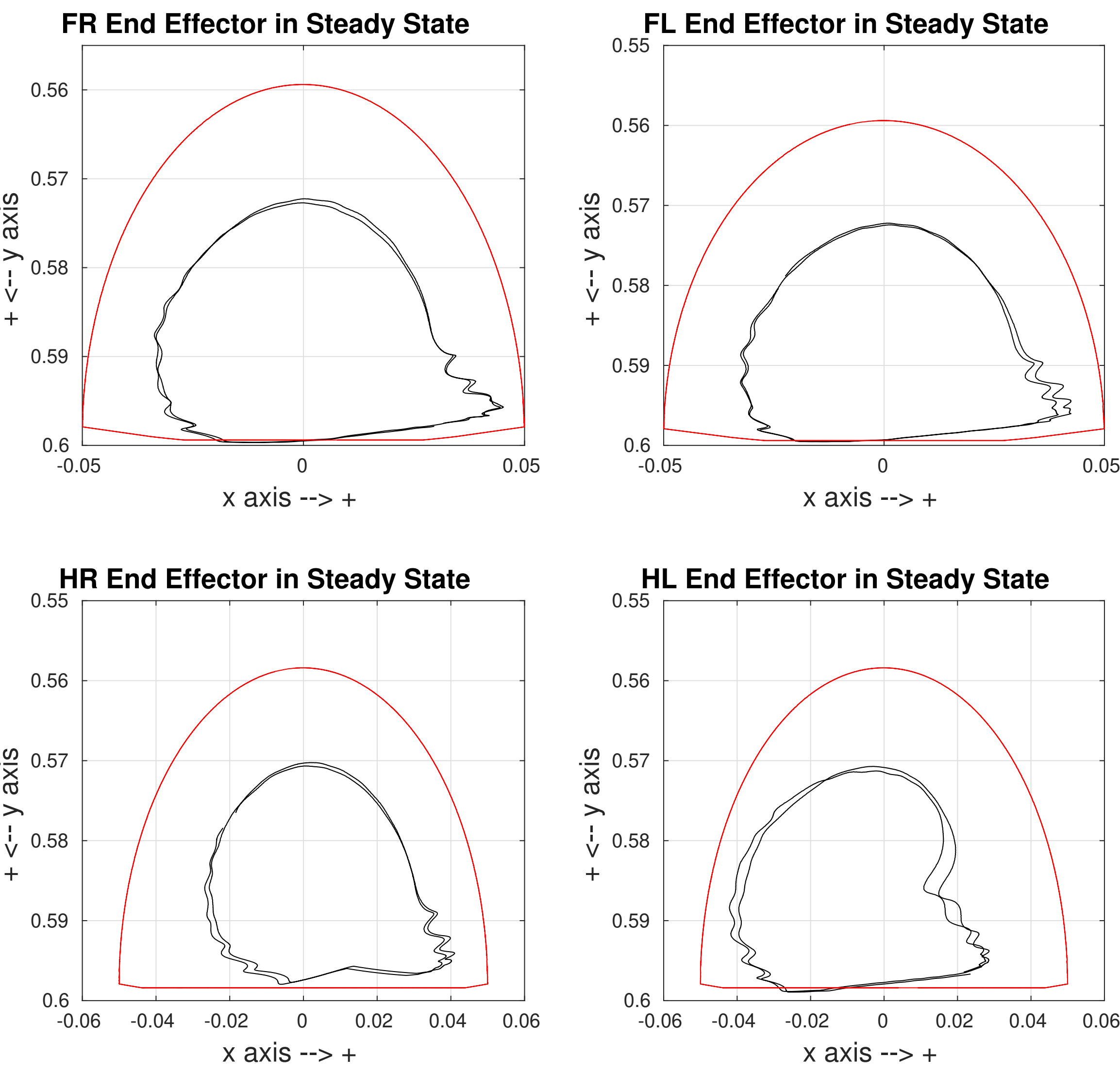
Πειραματική Αξιολόγηση

Αποτελέσματα

Για την αξιολόγηση της απόδοσης ολόκληρου του συστήματος, πραγματοποιήθηκε μια σειρά πειραμάτων. Σε αυτή τη σειρά πειραμάτων, ορίζεται μία επιθυμητή ελλειπτική τροχιά για το άκρο του κάθε ποδιού μέσω της αναπτυγμένης εφαρμογής μαζί με τα κέρδη ελέγχου και τις παραμέτρους του συστήματος. Τα δεδομένα καταγράφονται χρησιμοποιώντας το `rosbag` και επεξεργάζονται και απεικονίζονται με τη χρήση ενός προσαρμοσμένου Matlab script. Αξίζει να σημειωθεί ότι σε κάθε slave εφαρμόζεται ένας ελεγκτής PIV (Proportional - Integral - Velocity) (περισσότερες πληροφορίες στο [17]), έτσι ο master δεν επηρεάζει τη διαδικασία ελέγχου αλλά απλώς παρέχει σε κάθε slave τις απαραίτητες παραμέτρους μέσω EtherCAT.

Κατά τη διάρκεια της φάσης μόνιμης κατάστασης του πειράματος, όπου και οι παράμετροι `a_ellipse100` και `b_ellipse100` έχουν φτάσει στην τελική τους τιμή, το άκρο (End Effector) κάθε ποδιού εκτελεί μια συγκεκριμένη διαδρομή που προσπαθεί να συγκλίνει με την επιθυμητή ελλειπτική τροχιά. Η επιθυμητή ελλειπτική διαδρομή / τροχιά του άκρου κάθε ποδιού (κόκκινο) μαζί με την πραγματική απόκριση κάθε ποδιού (μαύρο) στο χώρο εργασίας

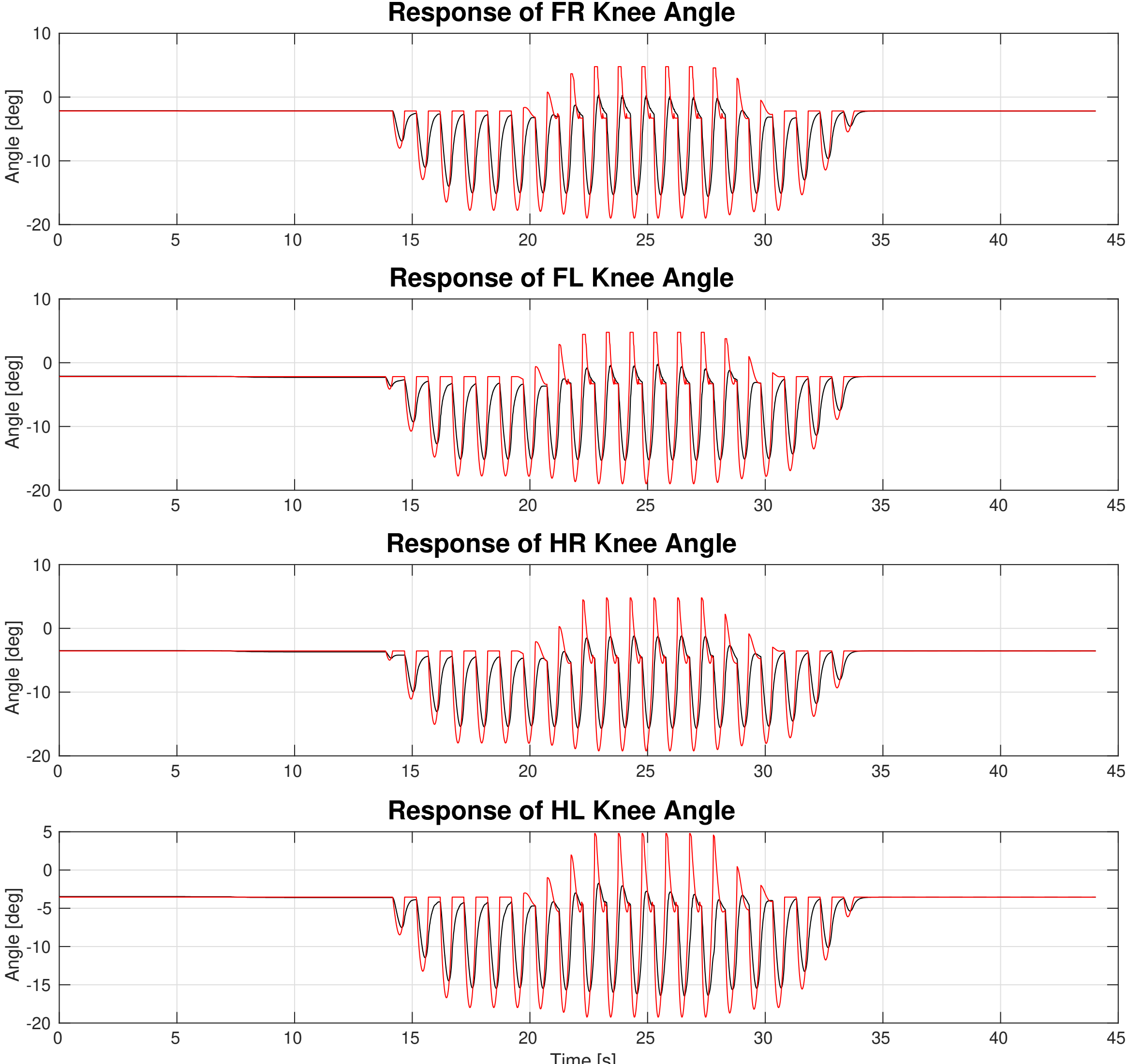
τους, σε σχέση με τα συστήματα συντεταγμένων που βρίσκονται στις αρθρώσεις ισχίων των ποδιών, εμφανίζονται στο Σχήμα 7. Αυτό το σχήμα διευκρινίζει το γεγονός ότι τα σφάλματα μόνιμης κατάστασης στις αρθρώσεις ισχίου και γονάτου μεταφράζονται ως σφάλματα στην τοποθέτηση του άκρου. Αξίζει να σημειωθεί ότι λόγω του εδάφους και των χαμηλών τιμών των Κερδών Ελέγχου, οι επιθυμητές ελλειπτικές τροχιές δεν παρακολουθούνται στενά στην μόνιμη κατάσταση και απαιτείται καλύτερη ρύθμιση αυτών των κερδών, ειδικά για τα οπίσθια πόδια.



Σχήμα 7: *Επιθυμητή ελλειπτική τροχιά όλων των άκρων των ποδιών (κόκκινο) μαζί με την πραγματική τους απόκριση (μαύρα) σε σχέση με τα συστήματα αναφοράς που βρίσκονται στις αρθρώσεις ισχίων των ποδιών.*

Το Σχήμα 8 εμφανίζει την επιθυμητή τιμή γωνίας άρθρωσης γόνατος κάθε ποδιού (κόκκινο) και την πραγματική απόκριση κάθε αντίστοιχης γωνίας άρθρωσης γόνατος (μαύρο) σε όλο το πείραμα. Τόσο η μεταβατική κατάσταση όσο και η μόνιμη κατάσταση απεικονίζονται. Οι μονάδες όλων των μεταβλητών είναι μοίρες και όπως μπορεί να παρατηρηθεί σε αυτά τα

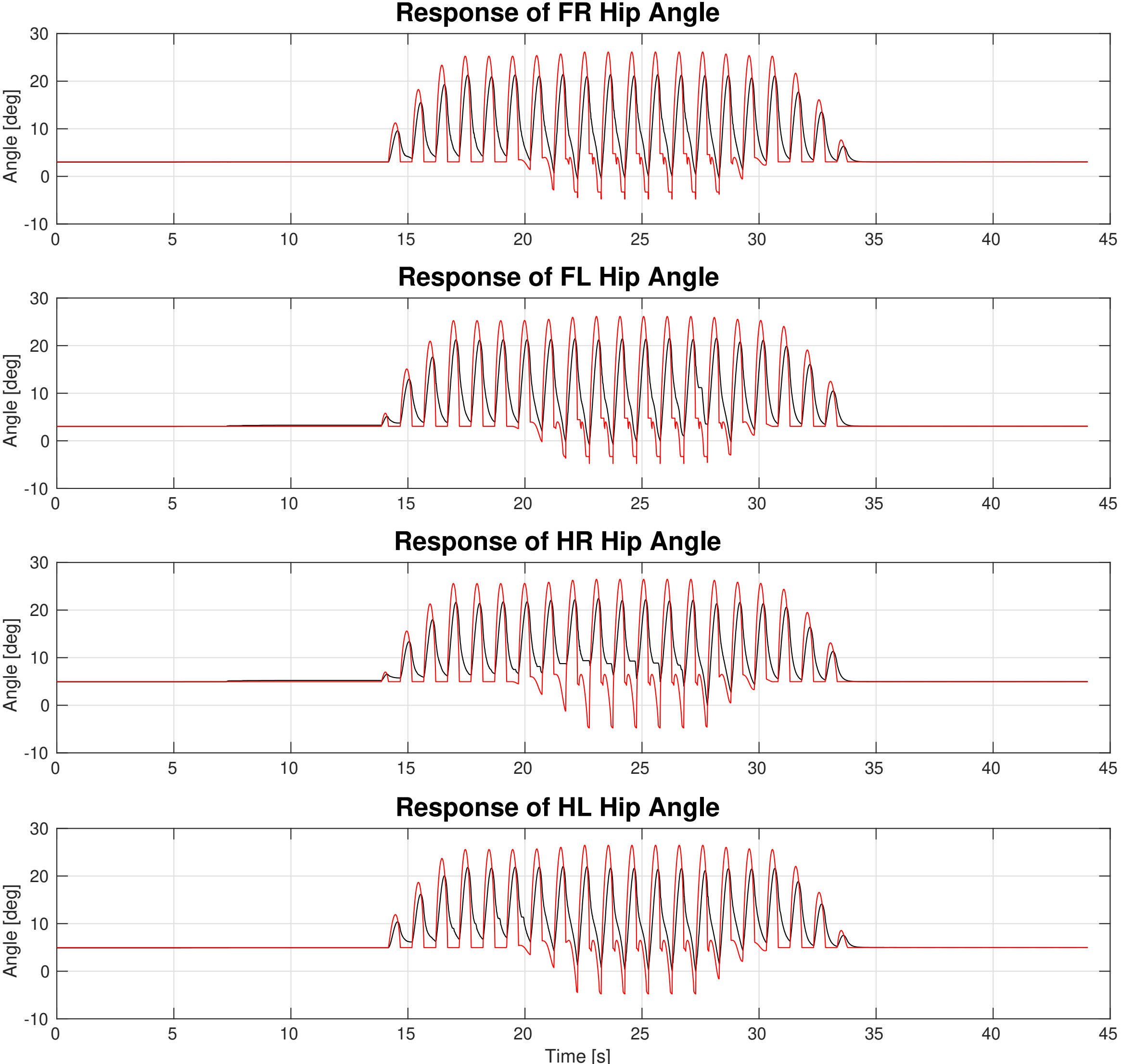
σχήματα, οι επιθυμητές τιμές παρακολουθούνται στενά από όλα τα πόδια, ωστόσο υπάρχει αρκετό περιθώριο βελτίωσης που μπορεί να επιτευχθεί με μια συνετή ρύθμιση των κερδών ελέγχου ή με ρύθμιση των ροπών για τους κινητήρες του γονάτου.



Σχήμα 8: *Επιθυμητή απόκριση των γωνιών των γονάτων (κόκκινο) και πραγματική απόκριση των αρθρώσεων των γονάτων (μαύρο).*

Κατά παρόμοιο τρόπο, το Σχήμα 9 περιγράφει την επιθυμητή τιμή της γωνίας άρθρωσης του ισχίου κάθε ποδιού (κόκκινο) και την πραγματική απόκριση κάθε αντίστοιχου ισχίου άρθρωσης (μαύρο) καθ' όλη τη διάρκεια του πειράματος. Τόσο η μεταβατική κατάσταση όσο και η μόνιμη κατάσταση απεικονίζονται. Οι μονάδες όλων των μεταβλητών είναι μοίρες και όπως μπορεί να παρατηρηθεί σε αυτά τα σχήματα, οι επιθυμητές τιμές παρακολουθούνται στενά από όλα τα πόδια, ωστόσο υπάρχει αρκετό περιθώριο βελτίωσης (ακόμη περισσότερο από τους κινητήρες γόνατος) που μπορεί να επιτευχθεί με σωστή ρύθμιση των κερδών ελέγχου για τους κινητήρες ισχίου. Επειδή οι ίδιες τιμές κέρδους ελέγχου χρησιμοποιήθηκαν και για

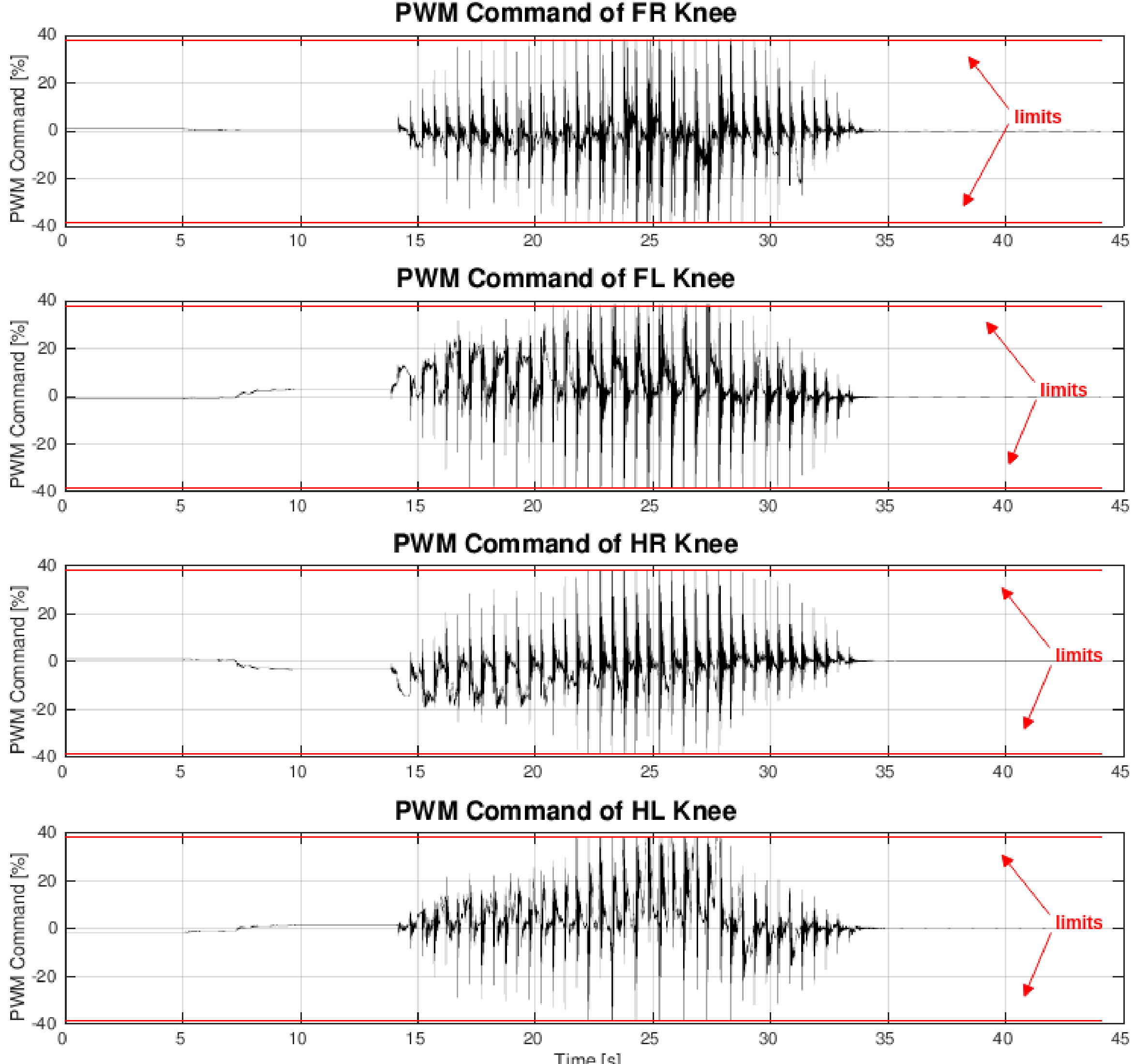
τους δύο κινητήρες (με και χωρίς ψήκτρες) είναι απολύτως κατανοητό γιατί οι δύο αυτές αρθρώσεις δεν έχουν την ίδια απόκριση όσον αφορά τα σφάλματα. Επιπλέον, αξίζει να αναφερθεί ότι η άρθρωση ισχίου εκτελεί ευρύτερη κίνηση, η οποία αποτελεί έναν ακόμη λόγο για τον οποίο τα προκύπτοντα σφάλματα είναι μεγαλύτερα σε σύγκριση με τις αρθρώσεις γόνατος.



Σχήμα 9: *Επιθυμητή απόκριση των γωνιών των ισχίων (κόκκινο) και πραγματική απόκριση των αρθρώσεων των ισχίων (μαύρο).*

Το Σχήμα 10 απεικονίζει τις εντολές PWM [%] του κινητήρα γόνατος κάθε ποδιού (μαύρο) με το αντίστοιχο προκαθορισμένο όριο (κόκκινο). Αυτές οι εντολές είναι η έξοδος του ελεγκτή PIV του γονάτου και μεταφράζονται απευθείας σε εντολές ροπής δεδομένου ότι εφαρμόζεται μια αρχιτεκτονική ελέγχου ρεύματος. Όπως μπορούμε να παρατηρήσουμε, οι εντολές και στα δύο οπίσθια πόδια είναι πάντοτε εντός του εύρους ορίων, επομένως δεν υπάρχει λόγος τροποποίησης τους. Όμοια, στα δύο μπροστά πόδια, αν και έχουν φθάσει τα όρια πολλές φο-

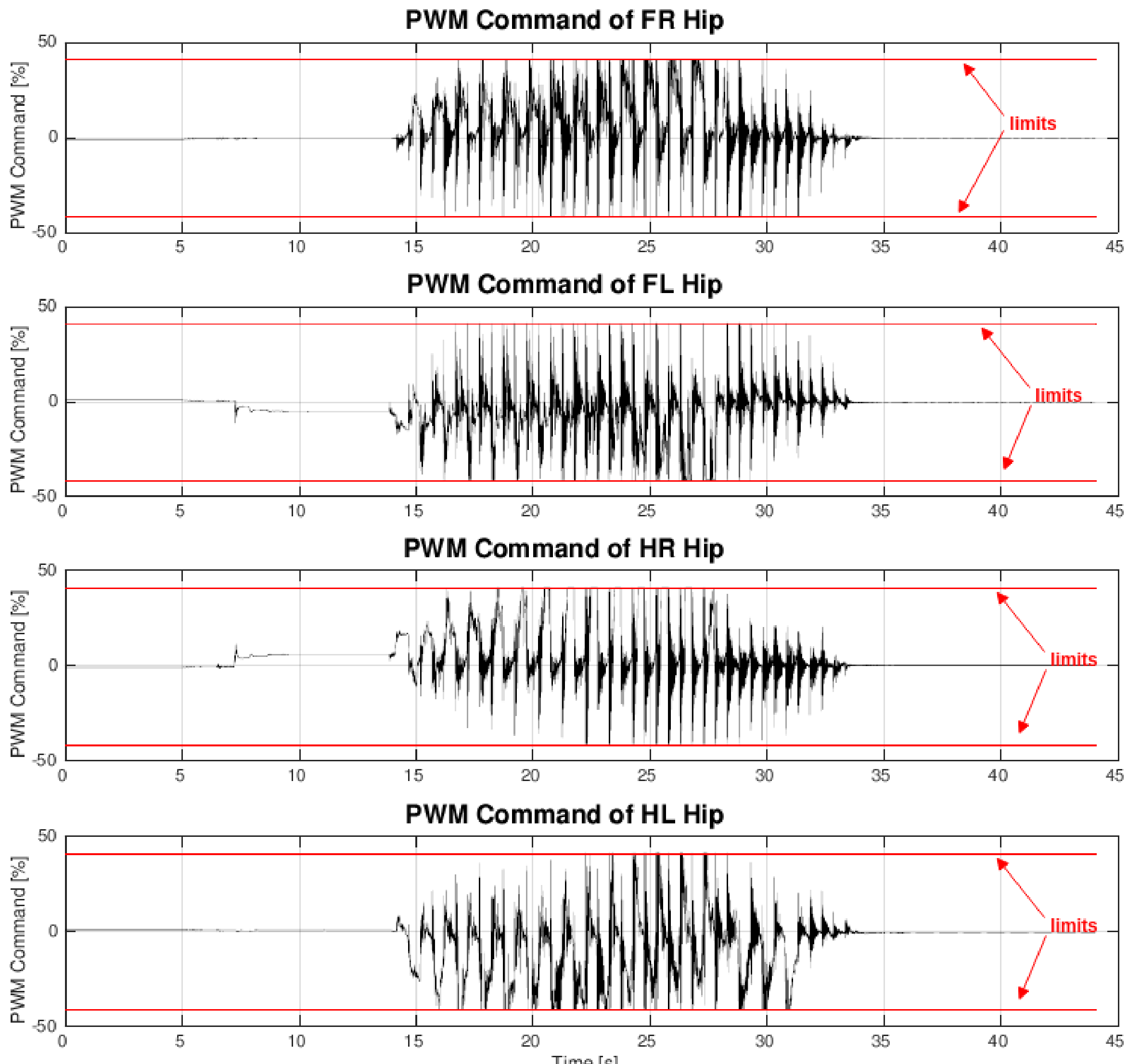
ρές, λόγω του γεγονότος ότι συνέβη μόνο για σύντομα χρονικά διαστήματα, δεν απαιτείται καμία επιπλέον ενέργεια.



Σχήμα 10: *Εντολές PWM του κινητήρα γονάτου κάθε ποδιού (μαύρο) και τα αντίστοιχα προκαθορισμένα όρια PWM (κόκκινο).*

Ομοίως, το Σχήμα 11 απεικονίζει τις εντολές PWM [%] του κινητήρα ισχίου κάθε ποδιού (μαύρο) με το αντίστοιχο προκαθορισμένο όριο (κόκκινο). Αυτές οι εντολές είναι η έξοδος του ελεγκτή PIV του ισχίου αυτή τη φορά και μεταφράζονται απευθείας σε εντολές ροπής, δεδομένου ότι εφαρμόζεται μια αρχιτεκτονική ελέγχου ρεύματος. Όπως μπορεί να παρατηρηθεί, τα όρια PWM του ισχίου προσεγγίζονται επανειλημμένα, ειδικά στα οπίσθια πόδια, με αποτέλεσμα να πρέπει να ληφθεί υπόψη μια αύξηση του επιτρεπόμενου εύρους.

Το Σχήμα 12 παρουσιάζει την εκτίμηση ταχύτητας της άρθρωσης του γόνατος κάθε ποδιού (μαύρο) και τα αντίστοιχα όρια ταχύτητας του κινητήρα (κόκκινο) όπως καθορίζονται από

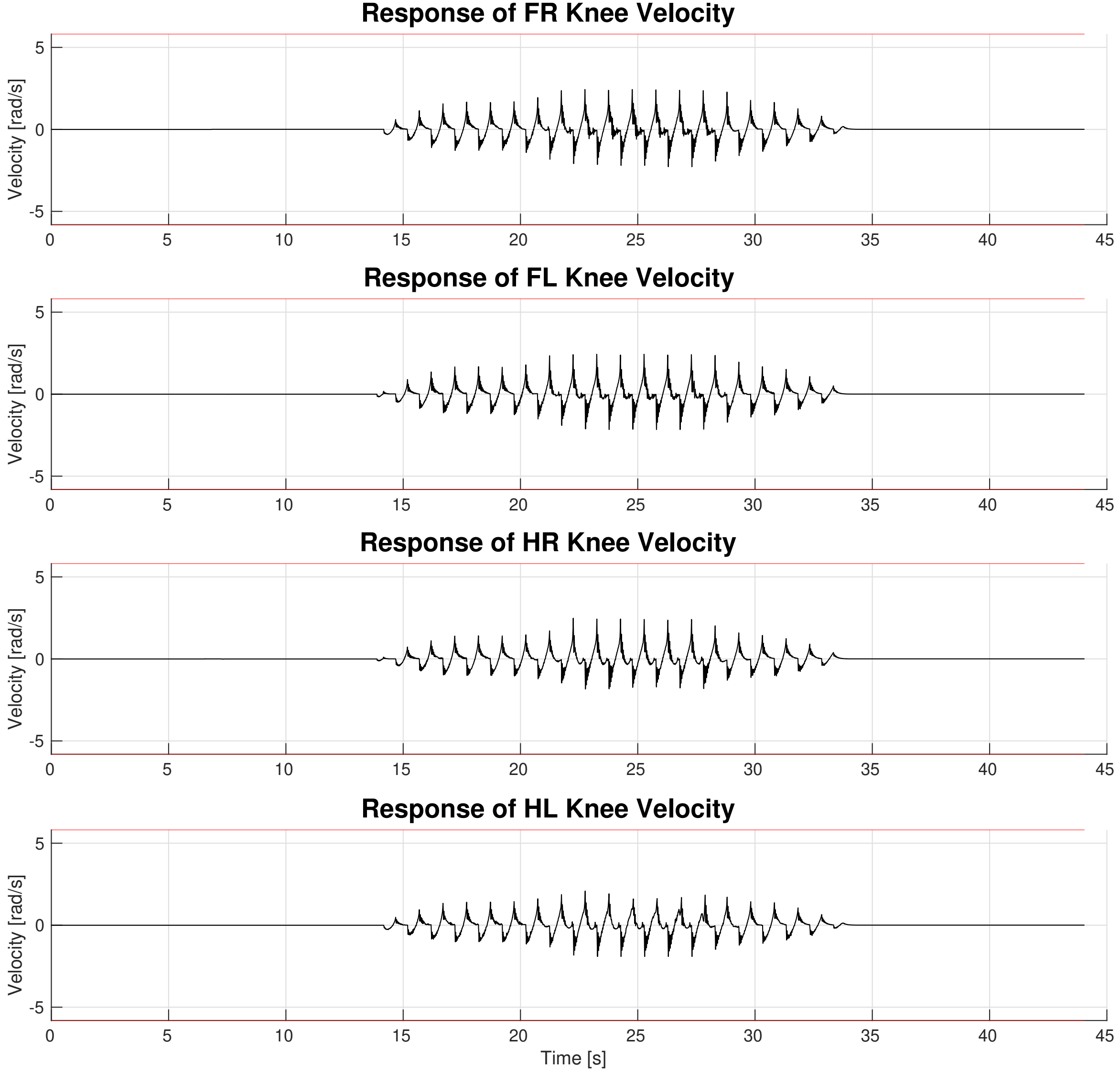


Σχήμα 11: *Εντολές PWM του κινητήρα ισχίου κάθε ποδιού (μαύρο) και τα αντίστοιχα προκαθορισμένα όρια PWM (κόκκινο).*

τον κατασκευαστή. Όπως μπορεί να παρατηρηθεί από το ακόλουθο σχήμα, οι ταχύτητες κάθε κινητήρα γόνατος είναι πάντοτε εντός της επιτρεπόμενης περιοχής. Επομένως, δεν υπάρχει καμία ανησυχία σχετικά με το σύστημα ταχύτητας που θα μπορούσε να δικαιολογήσει τη μείωση των ορίων PWM του γόνατος.

Τέλος, κατά παρόμοιο τρόπο, το Σχήμα 13 απεικονίζει την εκτίμηση ταχύτητας της άρθρωσης του ισχίου κάθε ποδιού (μαύρο) και τα αντίστοιχα όρια ταχύτητας του κινητήρα (κόκκινο) όπως καθορίζονται από τον κατασκευαστή. Για άλλη μια φορά, οι ταχύτητες κάθε κινητήρα ισχίου είναι πάντοτε εντός του επιτρεπόμενου εύρους, οπότε δεν υπάρχει λόγος να μειωθούν τα όρια PWM του ισχίου.

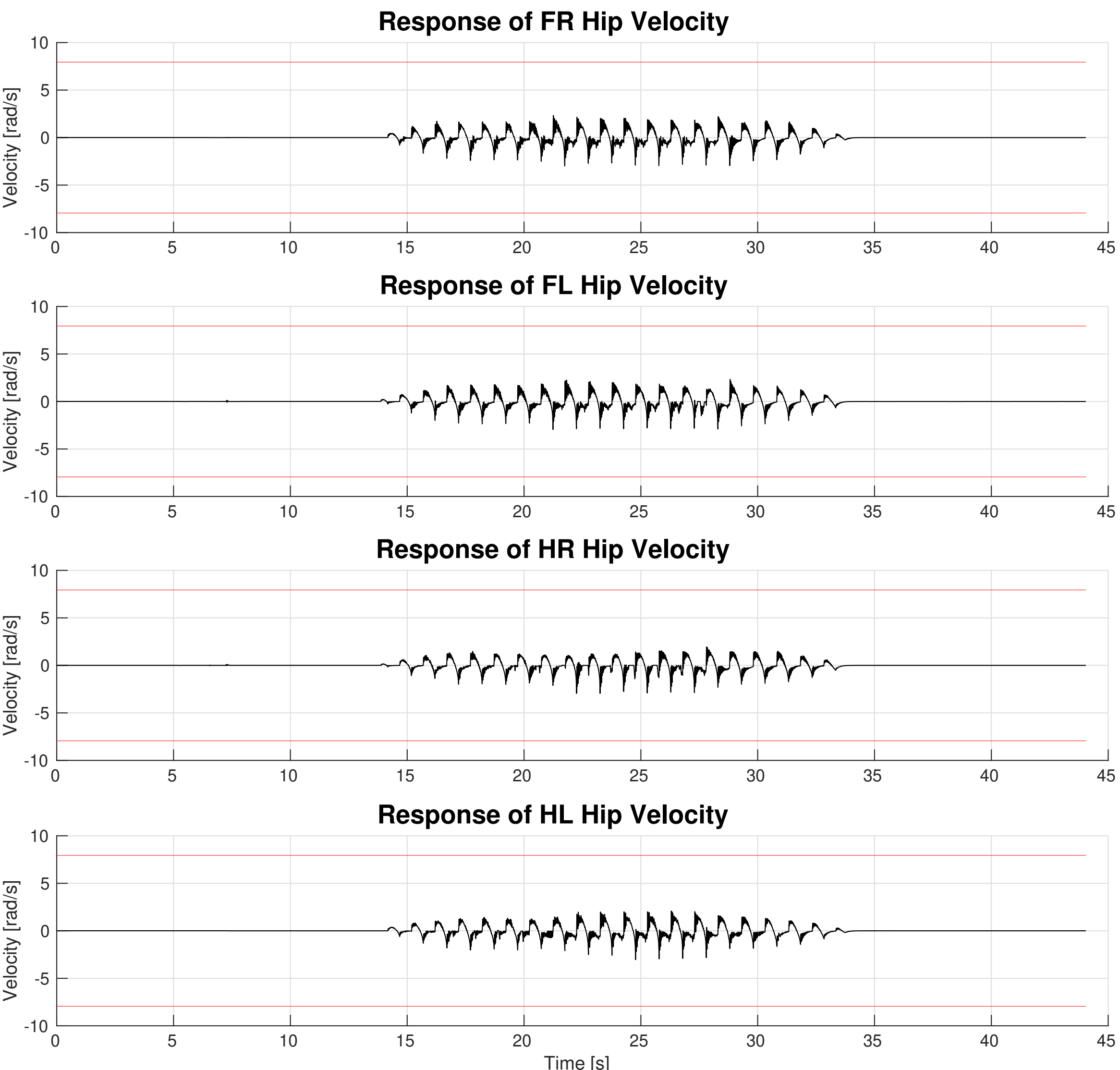
Γενικά, η συνολική εικόνα αυτών των αποτελεσμάτων δείχνει ότι τα πειράματα για τα πόδια



Σχήμα 12: *Εκτίμηση ταχύτητας της άρθρωσης γονάτου κάθε ποδιού (μαύρο) και τα αντίστοιχα προκαθορισμένα όρια ταχύτητας του κινητήρα (κόκκινο).*

στο Laelaps II ήταν επιτυχημένα, αν και συνιστώνται μικρές τροποποιήσεις στα κέρδη και στις αναλογίες ελέγχου (PWM εντολές) για τη βέλτιστη απόδοση στο βάδισμα, οι οποίες θα πρέπει να μελετηθούν περαιτέρω.

Από την προοπτική του EtherCAT master, πρέπει να σημειωθεί ότι το `ether_ros` λειτούργησε όπως αναμενόταν, χωρίς να υπάρχουν skipped πακέτα από το EtherLab, πράγμα που σημαίνει ότι οι περιορισμοί σε πραγματικό χρόνο, όπως αναλύονται στις λειτουργικές απαιτήσεις, έγιναν σεβαστοί . Ακόμη, είναι αξιοσημείωτο το γεγονός ότι **επιτεύχθηκε** $2.5\ kHz$ **συχνότητα βρόχου EtherCAT**. Τέλος, όσον αφορά το περιβάλλον ROS, τα μηνύματα λήφθηκαν με επιτυχία στα κατάλληλα θέματα εγκαίρως.



Σχήμα 13: *Εκτίμηση ταχύτητας της άρθρωσης ισχίου κάθε ποδιού (μαύρο) και τα αντίστοιχα προκαθορισμένα όρια ταχύτητας του κινητήρα (κόκκινο).*

Επίλογος

Συμπεράσματα

Συνολικά, οι απαιτήσεις που είχαν διατυπωθεί, ικανοποιήθηκαν με επιτυχία. Σύμφωνα με τα αποτελέσματα της πειραματικής αξιολόγησης, ο σχεδιασμός και η ανάπτυξη που παρουσιάζεται σε αυτή την Διπλωματική Εργασία πέτυχαν να συνδυάσουν τις τεχνολογίες EtherCAT και ROS υπό περιορισμούς σε πραγματικό χρόνο και να παράγουν το αποτέλεσμα ενός τετράποδου ρομπότ, δηλαδή του Laelaps II.

Λεπτομερέστερα, οι δυνατότητες πραγματικού χρόνου που προσφέρονται από την επέκταση

PREEMPT_RT αποδείχτηκαν επαρκείς για τον έλεγχο κίνησης του Laelaps II και ο συνδυασμός της επέκτασης μαζί με το EtherLab αποδείχθηκε άξιος αντικαταστάτης της προσέγγισης Windows / TwinCAT. Όσον αφορά την επέκταση PREEMPT_RT, παρόλο που καταναλώθηκε αρκετός χρόνος ανάπτυξης για λεπτομερή ρύθμιση του πυρήνα του συστήματος και του κώδικα της εφαρμογής, προκειμένου να βελτιστοποιηθεί η καθυστέρηση (latency) του master, το κόστος αυτό θεωρείται πολύ μικρότερο από άλλες προσεγγίσεις όπως το Xenomai και το RTAI, που μπορεί να προσφέρουν καλύτερες επιδόσεις, αλλά έχουν περισσότερα έξοδα συντήρησης και ανάπτυξης. Όσον αφορά το EtherLab, η απόφαση να υιοθετηθεί αυτή η προσέγγιση σχεδίασης αντί για το SOEM, αποδείχθηκε σοφή κατά τη διαδικασία ανάπτυξης και επικύρωσης. Αν και είχε μια απότομη καμπύλη μάθησης για την κατανόηση του τρόπου ανάπτυξης κώδικα που χρησιμοποιεί το API του, η τεκμηρίωση ήταν εξαιρετική και διευκόλυνε τη διαδικασία ανάπτυξης. Επίσης, το EtherLab έδειξε τη δύναμή του στη διαδικασία εντοπισμού σφαλμάτων, καθώς προσέφερε μηχανισμούς για την άμεση εξέταση κάθε πτυχής του δικτύου EtherCAT.

Τέλος, εκτός από τις δυνατότητες πραγματικού χρόνου, η αναπτυγμένη εφαρμογή προσφέρει διαλειτουργικότητα με το περιβάλλον ROS, μέσω του ROS API. Αυτό το βήμα ανοίγει πολλές δυνατότητες, λαμβάνοντας υπόψη το μέγεθος του οικοσυστήματος ROS και την ποικιλία των εφαρμογών που αναπτύσσονται σε αυτό. Οι μελλοντικοί κόμβοι ROS θα έχουν τη δυνατότητα να επικοινωνούν με τους κωδικοποιητές και τους κινητήρες του Laelaps II και να δημιουργούν προφίλ συγχρονισμένων κινήσεων των ποδιών. Αυτά τα προφίλ θα μπορούσαν να ξεκινήσουν απλά, όπως το βάδισμα, που μελετήθηκε σε αυτή την διπλωματική, και να συνεχίζουν με εξαιρετικά πολύπλοκες κινήσεις, όπως καλπασμό και τρέξιμο ή συνδυασμούς αυτών. Αυτό το χαρακτηριστικό δεν πρέπει να παραμεληθεί: η *ROS*-οποίηση του Laelaps II είναι ένα τεράστιο βήμα προς την δομοστοιχειωτή σχεδίαση (modularity) του λογισμικού και τη μείωση της ανάπτυξης και της διατήρησης, που είναι σημαντικοί παράγοντες τόσο για τον ακαδημαϊκό χώρο όσο και για τη βιομηχανία.

Μελλοντικές Δυνατότητες

Παρόλο που η τρέχουσα εφαρμογή ελέγχου κίνησης μέσω EtherCAT στο Laelaps II έχει δοκιμαστεί και έχει αποδειχθεί ότι είναι πλήρως λειτουργική τόσο σε επίπεδο λογισμικού όσο και υλικού, πολλές πτυχές μπορούν να βελτιωθούν στο μέλλον για να επιτευχθεί μεγαλύτερη ευρωστία.

Πρώτον, η αναπτυγμένη εφαρμογή μπορεί να επεκταθεί για να υποστηρίξει διαφορετικά

φορτία δεδομένων για EtherCAT slaves και αυτόματη διαμόρφωση μιας νέας εφαρμογής EtherCAT χωρίς χειροκίνητη διαμόρφωση στον πηγαίο κώδικα `ether_ros`.

Μια πρόσθετη ιδέα είναι η διεξαγωγή πειραμάτων για τον εντοπισμό των καθυστερήσεων σε κάθε πτυχή του συστήματος. Ο χρόνος του βρόχου EtherCAT καταναλώνεται μεταξύ του δικτύου, των υποτελών κόμβων και του κύριου κόμβου και θα ήταν χρήσιμο να γνωρίζουμε τον χρόνο που καταναλώνει κάθε στοιχείο του συστήματος. Σε αυτή την κατεύθυνση, εργαλεία εντοπισμού[8] στον πυρήνα θα μπορούσαν να χρησιμοποιηθούν για τον εντοπισμό ποια διαδικασία εκτελείται από ποια CPU, πόσο χρόνο χρειάζεται για να εκτελεστεί κλπ. Με αυτόν τον τρόπο μπορεί να εντοπιστεί η καθυστέρηση της εφαρμογής, του πυρήνα και του δικτύου EtherCAT. Η καθυστέρηση του δικτύου EtherCAT μπορεί να ανιχνευθεί με τη μέτρηση των διαστημάτων μεταξύ δύο διαδοχικών διακοπών του Ethernet IRQ που είναι αφιερωμένο στο δίκτυο EtherCAT. Αυτή η καθυστέρηση μπορεί εύκολα (αλλά όχι με μεγάλη ακρίβεια) να παρακολουθείται από τη χρήση του Wireshark. Η καθυστέρηση του πυρήνα αποτελείται από καθυστερήσεις που εισάγονται από την μονάδα EtherLab, τον χρονοδρομολογητή (scheduler) και άλλες διακοπές που δεν σχετίζονται με το δίκτυο EtherCAT (π.χ. χρονοδιακόπτες, IPI).

Τέλος, αν το τρέχον σύστημα ελέγχου αλλάξει και γίνει κεντρικό, το `ether_ros` θα πρέπει επίσης να αλλάξει. Μια κεντρικοποιημένη προσέγγιση σημαίνει μεγαλύτερο όγκο δεδομένων προς το `ether_ros` και η αναπτυγμένη εφαρμογή δεν έχει βελτιστοποιηθεί για αυτό το είδος περιπτώσεων. Για να λειτουργήσει αυτή η προσέγγιση, η εφαρμογή πρέπει να αλλάξει σημαντικά, βελτιστοποιώντας την αλληλεπίδραση με το περιβάλλον ROS, όπως τον αριθμό των topics που θα χρησιμοποιηθούν, τις συγκεκριμένες ουρές επανάκλησης (callback queues) κ.α.

[8]`http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html`

1 Introduction

Only those who will risk going too far can possibly find out how far one can go.

T.S. Eliot

In this chapter, we first provide a quick overview of the problem and the proposed solution. Then, existing solutions and their shortcomings are described. Next, the benefits of the proposed solution are briefly mentioned. Finally, the structure of the document is presented.

1.1 Problem Statement

In industrial robotic systems, there was always a need for a feature-rich network communication, between actuators, sensors and the processing unit(s), and usually this was handled with wired means. For this reason a variety of protocols in both hardware and software were developed. In addition, the time-dependent nature of the processes of those systems demanded the employment of real-time solutions to the network and computations.

ROS (Robot Operating System) is a meta-operating system, a standard in modern robotics, which is used for software development of robotics applications. It greatly reduces development and maintenance time and offers modularity (in hardware and software). Its community is targeted on GNU/Linux.

Fieldbus Systems is a family of industrial computer network protocols used for real-time control, which had an enormous influence on the flexibility and performance of industrial automation systems in all application areas.

EtherCAT is an Ethernet-based fieldbus system, and is suitable for both hard and soft real-time computing requirements in automation technology. Its features include short data update times with low communication jitter and reduced hardware costs, due to utilization of the low cost Ethernet technology, thanks to the latter's long term usage in computer networks in the past decades. These features render EtherCAT a proper network solution for real-time constrained robotic systems, especially for systems like quadruped or biped robots.

Quadruped robots tend to be designed with properties such as high speed, rapid acceleration and ability to make tight turns, thus requiring hard real-time constraints from their onboard processing units. With a distributed control scheme at hand, employment of EtherCAT for the design of a hard real-time system composed of networked processing units, is a satisfactory option, although the need for real-time programming the communication nodes (master/slaves) arises. Usually, the EtherCAT slaves (processing units for controlling the legs or MCUs in this kind of configuration, have a very specific task, i.e. that of controlling the legs. They comprise of, specifically designed, hardware-integrated EtherCAT Slave Controllers, therefore there is no need for extra work in the real-time aspect. However, the EtherCAT master can be implemented as a software solution, so a need appears to real-time program it. Thus, the primary objective of this thesis is the design and implementation of a real-time EtherCAT master in ROS on GNU/Linux.

In the following chapters, we thoroughly describe the design and implementation of a real-time EtherCAT master in the ROS framework and present the obtained results.

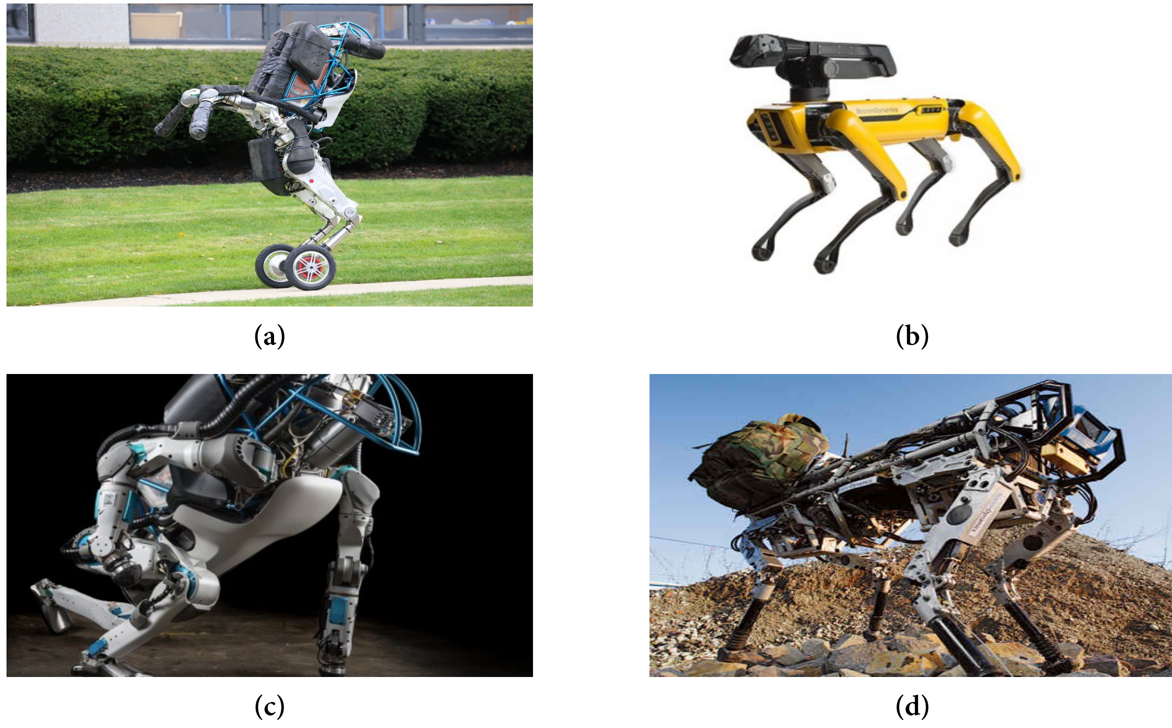
1.2 Literature Review

In this section we briefly describe other approaches, which have similarities with our own.

1.2.1 Legged Robots Overview

Research in *Legged Robots* has been conducted for numerous matters, from their design to their controllability, examining their capability for autonomous, semi-autonomous, or remotely-controlled operations in challenging terrains, where wheeled and tracked vehicles reach their limits. Future quadruped robots are expected to operate in highly dynamic, unstructured outdoor environments, where they will navigate inside challenging environments, such as collapsed buildings, disaster sites, forests, mountain farms, and construction sites. Their tasks will range from transmitting sensor readings to the remote operator (e.g., cameras, LI-

DAR, infrared, and radiation levels) to carrying heavy payloads such as tools or building materials. State of the art legged robots include Handle in Figure 1.1(a), SpotMini in Figure 1.1(b), Atlas in Figure 1.1(c) and BigDog in Figure 1.1(d), designed and manufactured by Boston Dynamics[1].



(a) (b) (c) (d)

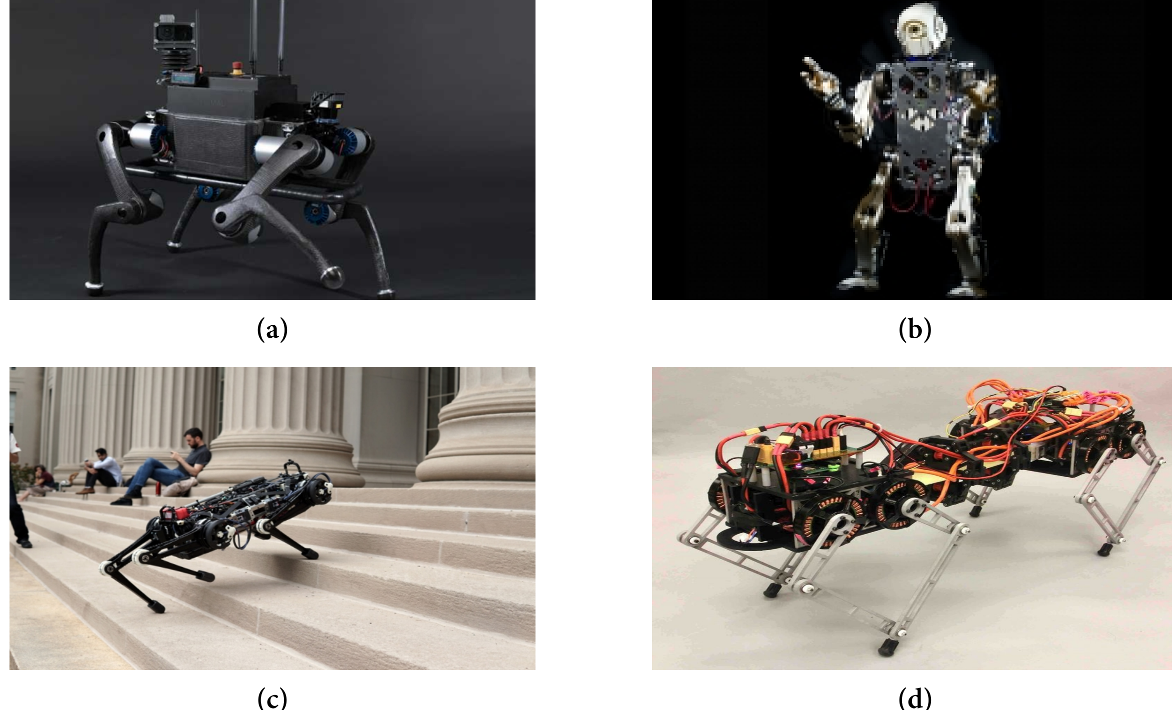
Figure 1.1: *Boston Dynamics legged robots:* ***(a)*** *Handle,* ***(b)*** *SpotMini,* ***(c)*** *Atlas,* ***(d)*** *BigDog.*

ANYmal robot in Figure 1.2(a) from the Institute of Robotics and Intelligent Systems of ETH Zurich university[2], MIT's Hermes in Figure 1.2(b) and Cheetah in Figure 1.2(c), robots by the Biomimetic Robotics Lab[3], and Upenn's Inu in Figure 1.2(d), robot by KOD*LAB[4], are characteristic examples of legged robots developed at universities.

1.2.2 Fieldbus Systems Overview

The advent of fieldbus systems in automation industry in the late 80's and early 90's, revolutionized it in a unique way. Prior to their arrival, the traditional method in industrial automation for connecting multiple computational units was parallel wiring [24], where all components were wired individually. However, the number of connections increased with the increasing degree of automation, which led to a high wiring expenditure. Therefore the

[1] `https://www.bostondynamics.com`
[2] `http://www.rsl.ethz.ch/robots-media/anymal.html`
[3] `http://biomimetics.mit.edu`
[4] `https://kodlab.seas.upenn.edu`



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

Figure 1.2: *State of the Art legged robots:* ***(a)*** *ANYmal,* ***(b)*** *Hermes,* ***(c)*** *Cheetah and* ***(d)*** *Inu.*

cheap fieldbus systems were a rather necessary solution, providing cheap and faster communication in the industrial networks.

The fieldbus systems are nowadays indispensable within industry. As a fixed component of complex machinery and installations, they are primarily used in manufacturing automation. However, the fieldbus is also used in process and building automation, as well as in automotive engineering.

Sensors and actuators (so-called "field devices") as well as motors, switches, drives, or lamps are connected with programmable logic controllers (PLCs) / master and process controllers with the help of wire-bound and serial fieldbuses. As such, the fieldbus supports the rapid exchange of data between individual system components even over great distances. Strong external loads cannot influence the robust digital signal transmission system. Fieldbus communicates only via a single cable, allowing considerable decrease of the wiring, compared to parallel wiring systems.

Fieldbus systems function in master-slave operation. The master controls the processes, while the slave stations work on the individual partial tasks.

Fieldbus systems can differ in their topology (star, line, tree or ring), in their transmission

medium, and in their transmission protocols (message-oriented procedure or summation frame procedure). The individual fieldbuses also differ in regard to the reachable cable length, the maximum number of data bytes per telegram and the function scope. As such, additional functions such as the alarm handling, diagnosis, and lateral traffic between individual bus participants are not possible for every fieldbus.

Popular examples of fieldbus systems:

- Interbus: The interbus with transmission rates of up to 2 Mbps is characterised by high transmission security and a short, constant cycle time. It is divided into subsystems and consists of the remote bus, the installation remote bus and the local bus arranged in a ring topology. The remote bus serves to connect up to 254 subscribers which are located at large distances from each other, while the local bus connects subscribers, that are located close to each other, to the system.
- Profibus: The PROFIBUS is used in manufacturing engineering and automation. It has an unlimited number of subscribers and data transmission rates between 9.6 kbps and 500 kbps. In master-slave operation, the Token passing [25] access procedure is used. Here, slaves may only access the profibus upon the master's request.

In addition, the utilization of the inexpensive Ethernet technology in the industrial automation, has produced many systems and protocols which are categorized as fieldbus systems. These systems have the same type of operation (master-slave), but are connected over Ethernet. As such, these Industrial Ethernet systems combine two valuable traits: standarization to the type of operation and support by the fieldbus systems community, and standarization and support by the computer networking society. Since modern machines and systems must perform increasingly complex tasks, data networks are growing larger. This is where real-time capable Ethernet networks come into play, because they provide a consistent flow of data from the control level down to the field level. Today, Industrial Ethernet is being promoted with several different proprietary designs [26, 27]. More than 20 different protocols compete in this rapidly growing market, each offering adaptations to meet different real-time and cost challenges, such as:

- EtherNet/IP (IP stands for "Industrial Protocol"): The EtherNet/IP is an industrial network protocol that adapts the Common Industrial Protocol to standard Ethernet. It is one of the leading industrial protocols in the United States and is widely used in a range of industries including factory, hybrid and process automation. An active star topology

is characteristic for the Ethernet/IP protocol, where individual devices are connected via a point-to-point connection, which is done via a switch. This has the advantage that due to the star topology, operation of devices with transfer rates from Mbps to Gbps can be activated in the same network. In addition, Ethernet/IP enables problem-free functioning of twisted pair and glass fiber cables. Not least, data collisions with simultaneous utilization of real-time applications are avoided with the help of the switch.

- Profinet: Profinet (acronym for Process Field Net) is an industry technical standard for data communication over Industrial Ethernet. It is designed for data collection from, and control of, equipment in industrial systems, under tight time constraints (on the order of 1ms or less). It is the open industrial Ethernet standard promoted by Profibus International (PI). This group claims that more than 2 million Profinet devices are currently installed in plant environments; more Profinet than Profibus engineers were certified in 2012.

- Sercos III [28]: Sercos III is the third generation of the Sercos interface, a globally standardized open digital interface for the communication between industrial controls, motion devices, input/output devices (I/O), and Standard Ethernet nodes. Sercos III merges the hard real-time aspects of the Sercos interface with Ethernet. It is based on and conforms to the Ethernet standard (IEEE 802.3 and ISO/IEC 8802-3). Sercos III features include cyclic update to devices at rates as low as 31.25 µs and support of up to 511 Slave devices on one network.

- EtherCAT: EtherCAT (Ethernet for Control Automation Technology) is a fieldbus system based on Ethernet, invented by Beckhoff Automation. The protocol is standardized in IEC 61158 and is suitable for both hard and soft real-time computing requirements in automation technology. The goal during development of EtherCAT was to apply Ethernet for automation applications requiring short update times (also called cycle times; ≤ 100 µs) with low communication jitter (for precise synchronization purposes; $\leq 1\mu\ s$) and reduced hardware costs. The entire *process data* communication takes place in the slave controller. Normal network update rates range from 1 to 30 kHz. However, EtherCAT can also be used with slower cycle times.

Concerning the Laelaps II quadruped, EtherCAT was selected, because of its high performance in terms of bandwidth and speed, its determinism, and its convenient slave-synchronization capabilities. In addition, there is no need to set device addresses, and its diagnostic

capabilities make the process of finding the sources of malfunctions and troubleshooting substantially easier.

1.2.3 EtherCAT Robotic Applications Overview

EtherCAT technology in robot applications has become increasingly popular in the last decade mainly due to the low cycle time, achieved reduced wiring and its modularity. Herein, characteristic examples in the different robotic application fields are presented.

In the industrial manufacturing sector, KUKA Robotics[5] has developed a modular EtherCAT controller (KR C4 Controller - Figure 1.3(a)) to control the developed industrial robotic arms of the company in several different tailor-made automation solutions. NexCom[6] has developed a wide range of EtherCAT based robotic solutions such as MiniBOT Robot in Figure 1.3(b) for educational purposes too, offering a broad selection of master controllers, robot arms, drives and motors, I/Os, industrial cameras etc.

(a) (b)

Figure 1.3: *(a) KR C4 Controller with robotic arm by KUKA and (b) MiniBOT Robot by NexCom.*

In the field of haptic – soft robotics and manipulation robotics, Shadow Robot Company exploited EtherCAT technology to develop a truly anthropomorphic hand. The Shadow Dexterous Hand[7] (Figure 1.4), has 20 actuated degrees of freedom, absolute position and force sensors, and ultra sensitive touch sensors on the fingertips, providing high precision.

In the field of legged robotics, PAL Robotics[8] has designed TALOS (Figure 1.5(a)), a fully electrical humanoid biped robot that uses torque control in every joint and EtherCAT to tackle complex industrial tasks. Talos is capable of 6 kg payload in each arm. Similarly, the Depart-

[5]`https://www.kuka.com`
[6]`http://www.nexcom.com`
[7]`https://www.shadowrobot.com`
[8]`https://pal-robotics.com`

Figure 1.4: *Shadow Dexterous Hand by Shadow Rob Company*

ment of Advanced Robotics of the Italian Istitute of Technology (IIT)[9] has exploited EtherCAT to design and build HyQ2Max quadruped robot [29] (Figure 1.5(b)) which mimics the robustness and versatility of animals in challenging terrains. In Switcherland, the robotics company ANYbotics, a spinoff of the famous Robotic Systems Lab in ETH Zurich, along with the lab have utilized EtherCAT in the design and control of ANYmal quadruped robot [30] (Figure 1.5(c)).

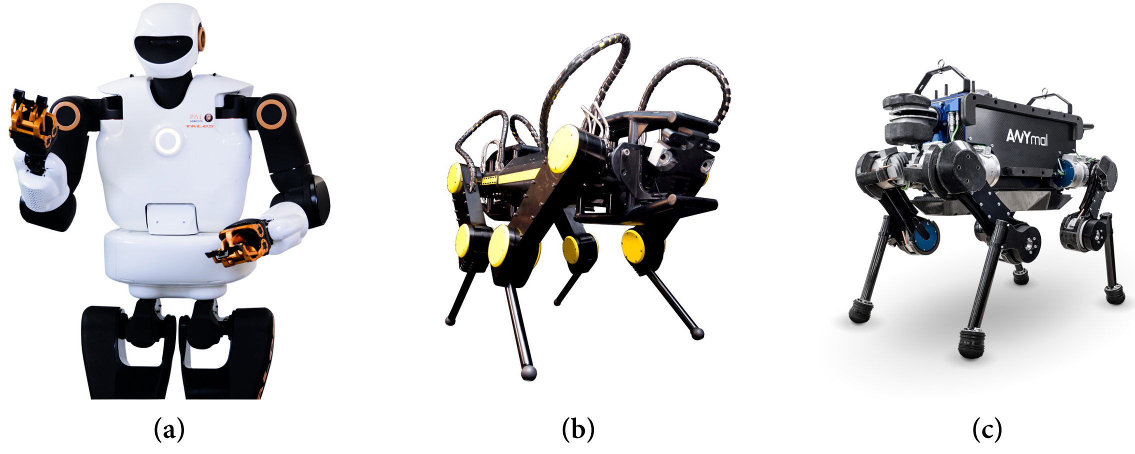


Figure 1.5: ***(a)*** *Talos biped robot by PAL Robotics,* ***(b)*** *HyQ2Max quadruped robot by IIT and* ***(c)*** *ANYmal robot from ETH.*

1.2.4 Real-time Systems Overview

Real-time systems [31] are computing systems that must react within precise time constraints to events in the environment. As a consequence, the correct behavior of these systems depends not only on the value of the computation but also on the time at which the results are produced [32]. A reaction that occurs too late could be useless or even dangerous. Examples

[9] https://www.iit.it

of applications that require real-time computing include the following [31]:

- Chemical and nuclear plant control.
- Control of complex production processes.
- Railway switching systems.
- Automotive applications.
- Flight control systems.
- Environmental acquisition and monitoring.
- Telecommunication systems.
- Medical systems.
- Industrial automation.
- Robotics.
- Military systems.
- Space missions.
- Consumer electronic devices.
- Multimedia systems.
- Smart toys.
- Virtual reality.

In the computer engineering world, as the transition from single process handling to multi process handling took place, the need for an Operating System to manage these processes was apparent. Similarly in the real-time systems world, the need for multi-tasking and scheduling of multiple prioritized tasks made the shift to *Real-Time Operating Systems* inevitable.

1.2.5 Real-Time Operating Systems Overview

With the use of an Operating System in real-time systems, there were new and more complex applications and solutions on the field. Nowadays, many Real-Time Operating Systems are available in the community. Some of the most popular proprietary RTOSes include [31]:

- VxWorks (by Wind River)[10]: First released in 1987, VxWorks is designed for use in embedded systems requiring real-time, deterministic performance and safety and security certification. It is targeted to industries, such as aerospace and defense, medical devices, industrial equipment, robotics, energy, transportation, network infrastructure, automotive, and consumer electronics.
- ENEA OSE (by ENEA)[11]: Enea OSE is a robust, high-performance, Real-Time Operating System optimized for multi-processor systems requiring deterministic real-time

[10] `https://www.windriver.com`
[11] `https://www.enea.com`

behavior and high availability. It decreases development time, enhances reliability and reduces lifetime maintenance costs for a wide range of systems, from wireless devices and automobiles, to medical instruments and telecom infrastructure.

- Windows CE (by Microsoft)[12]: Windows CE is an operating system developed by Microsoft and designed for small footprint devices or embedded systems.Some of the devices that run Windows CE include industrial controllers, point of sale terminals, cameras, Internet appliances, cable set-top boxes and communications hubs.
- QNX Neutrino[13]: QNX Neutrino is a Real-Time Operating System used for mission-critical applications, from medical instruments and Internet routers to telematics devices, process control applications, and air traffic control systems.
- Integrity (by Green Hills)[14]: Integrity is a Real-Time Operating System which is built around a partitioning architecture to provide embedded systems with reliability, security, and real-time performance.

Through the years, intensive research in the field of real-time systems has been conducted, producing many open-source real-time research kernels, including [31]: CHAOS [33], MARS [34], Spring [35], ARTS [36], RK [37], TIMIX [38], MARUTI [39], HARTOS [40], YARTOS [41], HARTIK [42], Erika Enterprise[15], Shark[16], Marte OS[17] and FreeRTOS[18]. Recently the FreeRTOS kernel became an AWS[19] open source project.

Most of the aforementioned kernels didn't evolve to a commercial product, but they were useful for experimenting novel approaches, some of which are to be integrated in next-generation operating systems [31].

GNU/Linux as a Free and Open Source Software (FOSS) project has a large active developer and user community, offering security, maintainability, reliability, stability, frequent updates and other features. However, it is a general purpose operating system originally designed to be used in server or desktop environments. For this reason, not much attention has been given to real-time issues. As a result, a latency of hundreds of milliseconds can be experi-

[12] https://www.microsoft.com
[13] https://blackberry.qnx.com/
[14] https://www.ghs.com
[15] http://www.erika-enterprise.com
[16] http://shark.sssup.it
[17] https://marte.unican.es
[18] https://www.freertos.org
[19] Amazon Web Services

enced in real-time activities. This makes common GNU/Linux distributions not suitable for hard real-time applications with tight timing constraints. On the other hand, making Linux a Real-Time Operating System would enable the full-power of a real operating system for real-time applications, including a broad range of open source drivers and development tools. For this reason, a considerable amount of effort has been given during the last years to provide GNU/Linux with real-time features. Such efforts include (but are not limited to) [31]:

- RTLinux: RTLinux has been the first real-time extension for Linux, created by Victor Yodaiken. Wind River Systems acquired the product in February 2007 and made a version available as Wind River Real-Time Core for Wind River Linux. As of August 2011, Wind River has discontinued the Wind River Real-Time Core product line, effectively ending commercial support for the RTLinux product. Currently, the version distributed by Wind River as Wind River Linux could be considered as the continued RTLinux project.

- RTAI: RTAI[20] (Real Time Application Interface) started as a modification of RTLinux by Paolo Mantegazza at the Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, Italy. Through the years, the original idea of RTLinux has been considerably changed and enhanced. RTAI is now a community project, and the source code is released as open source.

- Xenomai: The Xenomai[21] project was launched in August 2001. In 2003 it merged with the RTAI project to produce a production-grade real-time free software platform for Linux called RTAI/fusion, on top of Xenomai's abstract Real-Time Operating System (RTOS) core. Eventually, the RTAI/fusion effort became independent from RTAI in 2005 as the Xenomai project.

- PREEMPT_RT: PREEMPT_RT[22] is a kernel patch to make a Linux system more predictable and deterministic. It was originally created and developed by Ingo Molnar, a major contributor to the Linux kernel. Currently, Real-Time Linux, an Open Source project sponsored by The Linux Foundation[23], was formed to coordinate efforts to mainline Preempt RT and assist maintainers in continuing development work, long-term support and future research of RT.

[20] http://www.rtai.org
[21] http://www.xenomai.org
[22] https://wiki.linuxfoundation.org/realtime/start
[23] https://www.linuxfoundation.org

- SCHED_DEADLINE: SCHED_DEADLINE[24] is a Linux kernel patch developed by Evidence s.r.l. in the context of the ACTORS[25] European project. It adds a deadline-based scheduler with resource reservations in the standard Linux kernel.

- Linux/RK: In Linux/RK, the Linux kernel has been directly modified [43, 44] to introduce real-time features. Linux/RK is developed by the Real-time and Multimedia Systems Laboratory led by Dr. Raj Rajkumar at Carnegie Mellon University.

- LITMUS^{RT}: LITMUS^{RT}[26] is a real-time extension of the Linux kernel with a focus on multiprocessor real-time scheduling and synchronization. The Linux kernel is modified to support the sporadic task model, modular scheduler plugins, and reservation-based scheduling. Clustered, partitioned, and global schedulers are included, and semi-partitioned scheduling is supported as well. LITMUS^{RT} has been continuously maintained by Björn Brandenburg since 2006, and actively developed until 2017.

1.2.6 ROS 2 Overview

1.2.6.1 History of ROS

ROS [45] started as the development environment for the Willow Garage PR2 robot. The primary goal was to provide the software tools that users would need to undertake novel research and development projects with the PR2. At the same time, it was desired that ROS to be useful on other robots. So a lot of effort was given into defining levels of abstraction (usually through message interfaces) that would allow much of the software to be reused.

Still, it's characteristics include [45]:

- No real-time requirements (or, any real-time requirements would be met in a special-purpose manner).

- Excellent network connectivity (either wired or close-proximity high-bandwidth wireless).

- Applications in research, mostly academia.

- Maximum flexibility, with nothing prescribed or proscribed.

[24] `http://www.evidence.eu.com/sched_deadline.html`
[25] `http://www.actors-project.eu/`
[26] `https://www.litmus-rt.org/`

ROS satisfied the PR2 use case, but also was useful on a variety of other robots. Today ROS is used not only on the PR2 and robots that are similar to the PR2, but also on wheeled robots of all sizes, legged robots, industrial arms, outdoor ground vehicles (including self-driving cars), aerial vehicles, surface vehicles, and more.

In addition, ROS is being adopted beyond the academic research that was the initial focus. ROS-based products are coming to market, including manufacturing robots, agricultural robots, commercial cleaning robots, and others. Government agencies are also looking closely at ROS for use in their field systems. For instance, NASA is expected to be running ROS on the Robonaut 2 that is deployed to the International Space Station.

With all these new uses of ROS, the platform is extended. While it is holding up well, there is a belief that the needs of a now-broader ROS community can be met, by tackling their new use cases head-on, hence the effort for ROS 2.

1.2.6.2 New Use Cases

The following use cases are of specific interest, for the ongoing and future growth of the ROS community, in which there wasn't much consideration at the beginning of the project [45]:

- Teams of multiple robots: while it is possible to build multi-robot systems using ROS today, there is no standard approach, and they are all somewhat of a hack on top of the single-master structure of ROS.
- Small embedded platforms: small computers, including "bare-metal" micro controllers, are wanted to be first-class participants in the ROS environment, instead of being segregated from ROS by a device driver.
- Real-time systems: real-time control directly in ROS is a common request, including inter-process and inter-machine communication (assuming appropriate operating system and/or hardware support).
- Non-ideal networks: ROS is expected to behave as well as possible when network connectivity degrades due to loss and/or delay, from poor-quality WiFi to ground-to-space communication links.
- Prescribed patterns for building and structuring systems: while the flexibility of ROS is maintained, there is a need to provide clear patterns and supporting tools for features such as life cycle management and static configurations for deployment.

At the beginning of the ROS project, the above mentioned use cases weren't the norm, therefore they weren't treated in a canonical frame and the development on these areas was dependent in the way robotics engineers skipped the norm to satisfy their needs. However, with ROS 2[27] there seems to be a different point of view. Specifically in the real-time systems aspect, there is an exemplary use case [46, 21] which demonstrates a real-time robotic system, consisting of real-time computing nodes (with Real-Time Operating Systems) and real-time communication between them (via TSN protocols), all integrated in the ROS 2 environment. The results look very promising, however the technology needs to mature and gain sufficient support from the robotics community.

1.3 Benefits

In this section we highlight the value of our contribution by mentioning the benefits of its design, as follows:

- **EtherCAT utilization:** As a real-time network communications protocol, EtherCAT has a large community of users. In the last years, it has become popular in the robotics community and in robotics labs for its benefits.
- **Integration in ROS:** In robotics, the Robot Operating System has become a standard framework. Software that integrates with it, has profound benefits, like off-the-shelf libraries, rapid prototyping, modularity, standardization and community support.
- **Software on GNU/Linux:** There isn't much to say about the benefits of developing software on GNU/Linux. The fact that it's one of the biggest FOSS projects till today provides unsurpassed benefits: free code (GNU Public Licence), large user and developer community.
- **Real-time Solution:** The proposed design approach utilizes one of the biggest projects in the real-time GNU/Linux world, namely the PREMPT-RT patch. Again benefits here include low maintenance cost, stability and great community for support and development.

[27] `https://github.com/ros2/ros2`

1.4 Thesis Structure

This thesis is organized as follows: In **Chapter 2** and **Chapter 3**, we present the theoretical background and concepts that our work is based on. In **Chapter 4** we analyse the architecture of our solution and the design decisions made from a higher-level perspective. In **Chapter 5** we demonstrate the main points of our implementation and refer to the problems we faced during the development process, the proposed workarounds, the optimizations and the testing. In **Chapter 6** we present the experimental evaluation of our solution. Finally, in **Chapter 7** we sum up with concluding remarks, suggested future improvements and alternative approaches.

2 Background in Real-Time & ROS

We are not makers of history. We are made by history.

Martin Luther King, Jr.

In the two following chapters, the key theoretical elements for the understanding of this project, are provided. First, several fundamental principles concerning real-time systems are explained. Next specific areas considering the real-time modification of GNU/Linux, are analyzed. This chapter concludes with a brief presentation of ROS, the framework in which the developed project has been written.

2.1 Real-time Systems Concepts

Real-time systems had an astounding impact in the industrial automation field. From avionics and nuclear plants, to robotics and automotive industry, the need for deterministic systems is unquestionable. First, a definition of these systems, along with their basic characteristics are the following:

2.1.1 General Concepts

> *A real-time system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure [19].*

Consequently, the correctness of the system's response depends not only on the logical result but also on the time it was delivered. A real-time system can be distinguished in three

categories [19]:

- Hard: In hard real-time systems, failure to meet a single deadline leads to complete and catastrophic system failure.
- Firm: In firm real-time systems failure to meet a few deadlines will not lead to total failure, but missing more than a few leads to complete and catastrophic system failure.
- Soft: In soft real-time systems performance is degraded by failure to meet response-time constraints.

Characteristic differences between hard real-time systems and soft real-time systems are illustrated in the following Table 2.1:

Characteristic	Hard real-time	Soft real-time (on-line)
Response time	Hard-required	Soft-desired
Peak-load performance	Predictable	Degraded
Control of pace	Environment	Computer
Safety	Often critical	Non-critical
Size of data files	Small/medium	Large
Redundancy type	Active	Checkpoint–recovery
Data integrity	Short-term	Long-term
Error detection	Autonomous	User assisted

Table 2.1: *Hard real-time versus soft real-time systems [18].*

Examples of hard real-time systems include power plant control systems, railway switching systems, medical systems (e.g. pacemakers), military systems, avionics and electronic engines. Examples of firm real-time systems include most professional and industrial robot control systems such as the control loops of collaborative robot arms, aerial robot autopilots and mobile robots, including self-driving vehicles [46]. Examples of soft real-time systems include live audio-video systems and telepresence robots [46]. An artistic illustration of the above concepts is presented in Figure 2.1.

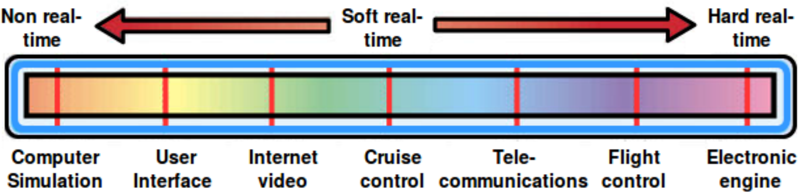


Figure 2.1: *Spectrum of real-time systems.*

Other examples of categories of real-time systems, include fail-safe, fail-operational, guaranteed-response, best-effort and more. More information on the matter can be found in [18].

2.2 Real-time Task Scheduling

Note: This section is largely based on [3, 31].

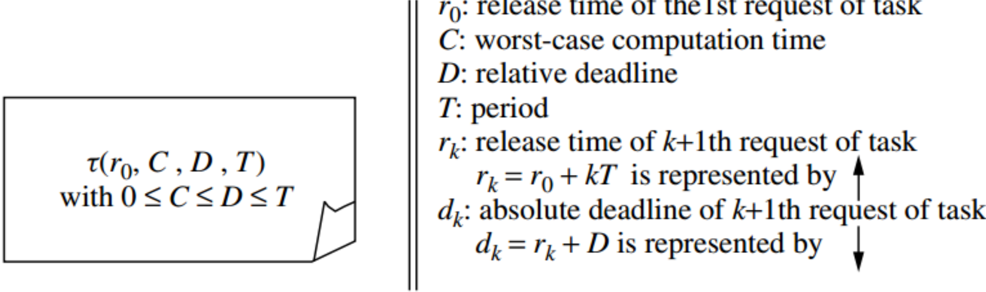
In a conventional *operating system*, the goal of a scheduler is to optimize a metric (average latency, average throughput, minimum latency, maximum throughput etc), by assigning work to resources. However, in a Real-Time Operating System, tasks have timing constraints and their execution is bounded to a maximum delay that has to be respected [3]. The objective of scheduling in this case, is not only to optimize a metric, but also to allow tasks to meet these timing constraints when the application runs in nominal mode.

Real-time tasks are the basic software activities that are scheduled; they may be periodic or aperiodic, and have soft, firm or hard real-time constraints [3]. The basic parameters of a real-time task are depicted in a task model and are presented in Figure 2.2:

- r: The task's release time (or arrival time), i.e. the triggering time of the task execution request.
- C: The task's worst-case computation time, i.e. the time the task is fully allocated to the processor.
- D: The task's relative deadline, i.e. the time the task has in order to finish, before it misses it's deadline.
- T: The task's period (this parameter is valid only for periodic tasks).

For a hard real-time task, the relative deadline allows computation of the absolute deadline $d = r + D$. Violation of the absolute deadline causes failure.

For the aperiodic tasks, the parameter T doesn't exist [3]. These four parameters (r, C, D, T) are sufficient for modelling a periodic task. Each time a task is ready to run, it releases a periodic request. After the first release time, the next release times (also called request times, arrival times or ready times) are $r_k = r_0 + k_T$, where r_0 is the first release and r_k the $k + 1$th release. Consequently, the next absolute deadlines are $d_k = r_k + D$. A common scenario for a real-time task, is to have parameters $D = T$, which implies that the periodic task has a relative deadline equal to period. The task parameters should always follow this rule: $0 < C \leq D \leq T$. The precision on defining the above parameters affects also the quality of scheduling, therefore their definition is an important aspect of real-time design. If the duration of operations



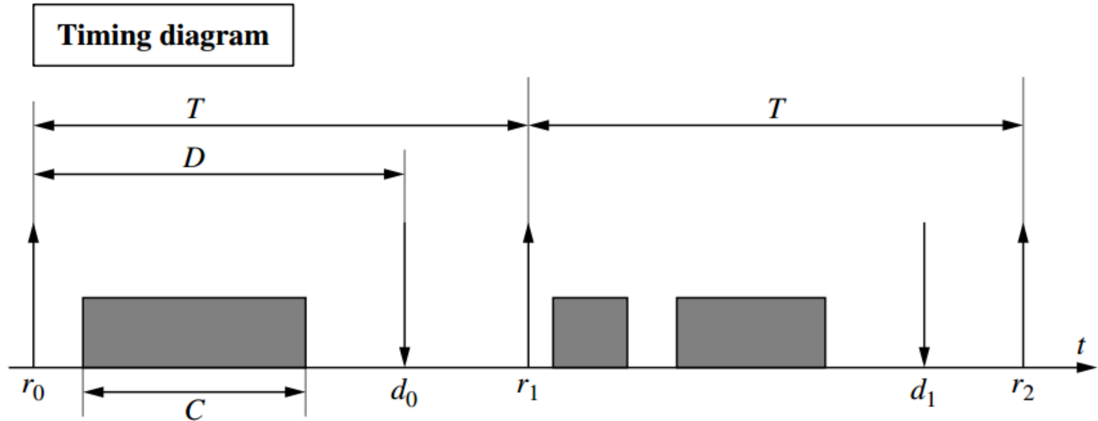


Figure 2.2: *A task model [3].*

like task switching, system calls, interrupt processing and scheduler execution cannot be neglected, they have to be added to the task computation times. As a result, a deterministic behavior is required for the kernel, which should guarantee maximum values for these operations. Other useful parameters derived from the four previous parameters (r, C, D, T) are:

- $u = C/T$: The processor utilization factor of a task; the inequality $u \leq 1$ must hold.
- $ch = C/D$: The processor load factor; the inequality $ch \leq 1$ must hold.

It should be noted that usually, the problem of *timing constraints* is not the only one that has to be addressed [31]. Other typical constraints include *precedence constraints* (there is a time dependence between two tasks) and *resource constraints* (software structures may require mutual exclusion).

In general, to define a scheduling problem three sets need to be specified: a set of n tasks

$\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, a set of m processors $P = \{P_1, P_2, ..., P_m\}$ and a set of s types of resources $R = \{R_1, R_2, ..., R_s\}$ [31]. In this context, scheduling means assigning processors from P and resources from R to tasks from Γ in order to complete all tasks under the specified constraints [47]. This problem, in its general form, has been proved to be NP-complete [48] and computationally intractable. In order to reduce the complexity of constructing a feasible schedule, typical approaches found in literatures include: simplification of the computer architecture (i.e. by considering single processor systems), adoption of a preemptive model, usage of fixed priorities, removal of precedence and/or resource constraints, homogeneity in task sets (only periodic or only aperiodic activities), just to name a few [31]. Based on the assumptions made on the system or on the tasks, the various scheduling algorithms are classified as follows [31]:

- Preemptive versus Non-preemptive: The running task can / cannot be interrupted at any time to assign the processor to another active task.
- Static versus Dynamic: Scheduling decisions are based on fixed / dynamic parameters.
- Off-line versus Online: A scheduling algorithm is used off-line / online if it is executed before tasks activation / executed at runtime.
- Optimal versus Heuristic: An algorithm is said to be optimal if it minimizes a given cost function defined over the task set. An algorithm is said to be heuristic if it is guided by a heuristic function in taking its scheduling decisions. The heuristic algorithm doesn't guarantee optimality.

Considering these classifications, some of the most popular uniprocessor periodic task scheduling algorithms used in real-time operating systems, include:

- Rate Monotonic (RM): This algorithm assigns priorities to periodic tasks according to their periods [31]. This means that tasks with shorter periods will have higher priorities. Since periods are constant, RM is a fixed-priority assignment: a priority P_i is assigned to the task before execution and does not change over time [31]. In addition, RM is preemptive: the currently executing task is preempted by a newly arrived task with shorter period [31]. The fixed-priority assignment makes the RM algorithm easy to use and simple to understand and implement. However this simplicity comes with a cost. In [49] it is shown that RM is optimal among all fixed-priority assignments in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM. The authors in [49] also calculated the least upper bound of the

processor utilization factor for a generic set of n periodic tasks. The schedulability test for RM is [3, 50]:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

where C_i is the computation time, T_i is the release period (with deadline one period later), and n is the number of processes to be scheduled. For example, $U \leq 0.8284$ for two processes. If the number of processes tends towards infinity, this expression tends towards [50]:

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147\ldots$$

A study [3] using stochastic methods showed that for random task sets, the processor utilization has an asymptotic bound of 88% [51], however this bound depends on knowing the exact task parameters (periods, deadlines) which cannot be guaranteed for all task sets. The feasibility analysis of the RM algorithm can also be performed using a different approach, called the Hyperbolic Bound [52, 53]. The test has the same complexity as the original bound in [49], but it is more permissive, as it accepts task sets that would be rejected using the original approach. More information regarding this algorithm can be found in [31, 3] and [54, Chapter 2].

- Earliest Deadline First (EDF): This algorithm assigns priority to tasks according to their absolute deadline: the task with the earliest deadline will be executed at the highest priority [31, 3]. As pointed in [55], EDF is optimal among all online algorithms, meaning that if a task set is not schedulable by EDF, then it cannot be scheduled by any other algorithm. There is a necessary and sufficient schedulability condition for periodic tasks with deadlines equal to periods, scheduled under EDF [3]:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.1}$$

The previous inequality shows that a set of periodic tasks with deadlines equal to periods is schedulable with the EDF algorithm if and only if the processor utilization factor is less than or equal to 1. A hybrid task set (with periodic and aperiodic tasks) is schedulable with the EDF algorithm if (sufficient condition):

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \tag{2.2}$$

A necessary condition is given by Equation 2.1. Note that Equation 2.1 provides a nec-

essary and sufficient condition to verify the feasibility of the schedule [54, Chapter 2]. Thus, if it is not satisfied, no algorithm can produce a feasible schedule for that task set. The EDF algorithm does not make any assumption about the periodicity of the tasks; hence it can be used for scheduling periodic as well as aperiodic tasks [3]. The dynamic priority assignment allows EDF to exploit the full CPU capacity, reaching up to 100% of processor utilization [54, Chapter 2]. When the task set has a utilization factor less than one, the residual fraction can be efficiently used to handle aperiodic requests activated by external events. A benefit for using a fixed-priority algorithm like RM, compared to a dynamic-priority like EDF, is its simple implementation and support by the hardware and the RTOSes. EDF [54, Chapter 2] is superior in many aspects [56], generating a lower number of context switches, thus causing less runtime overhead. Furthermore, using a suitable kernel mechanism for time representation [57], EDF can be implemented effectively in microprocessors for increased system utilization and timely execution of hybrid task sets [58]. Finally, it is worth mentioning that [31] an interesting feature of EDF during permanent overloads is that it automatically performs a period rescaling, so tasks start behaving as they were executing at a lower rate, proven in [59], while under fixed priority scheduling, a permanent overload condition causes a complete blocking of the lower priority tasks.

In conclusion, it is worth mentioning that according to [3] there doesn't exist an optimal online scheduling algorithm for multiple processors [60]. Therefore the guaranteed optimality of a real-time scheduling algorithm in uniprocessor systems, i.e. EDF, doesn't hold in the multiprocessor systems. Another notable fact is that usually in a realistic scenario, the task set is not homogeneous, meaning that the above mentioned algorithms must be enhanced with other approaches to handle non-homogeneous (hybrid) task sets.

2.3 Real-time GNU/Linux

Note: This section is largely based on [5].

GNU/Linux [20] was developed to be a general-purpose operating system based on Unix, supporting multiple users. However, the objectives of such a system don't line up with the requirements of real-time tasks and operations. The main objective of general-purpose operating systems is the maximization of average throughput, at the expense of latency, while the main objective of Real-Time Operating Systems is to place an upper bound on latency, at the expense of average throughput. In general, two major real-time approaches were adopted in

GNU/Linux [20, 5]:

- The Co-Kernel Approach: In this approach, a real-time kernel is placed side-by-side with Linux on the same hardware. In this approach belong the efforts by RTAI and Xenomai. In this case, all device interrupts are processed by the co-kernel prior to being processed by the standard kernel, in order for Linux not to postpone them. In this way, deterministic response time is ensured on the real-time side. Also, usually specific APIs are needed in order to develop a real-time application in systems following this approach.
- The Fully Preemptible Kernel Approach: In this approach, the main objective is to convert Linux itself into a full RTOS. This means that the Linux kernel's internals are changed, in order to allow real-time processes to run uninterrupted, without unpredictable or unbounded activities caused by non real-time processes. The Real-Time Linux (RTL) Collaborative Project[1] is the most relevant open-source solution for this option [21]. The RTL project is based on the PREEMPT_RT patch and aims to create a predictable and deterministic environment turning the Linux kernel into a viable real-time platform. The ultimate goal of the RTL project is to mainline the PREEMPT_RT patch. The objective of this project is not to create an RTOS based on GNU/Linux, but to provide real-time capabilities to the Linux kernel. The benefit of this approach, is the utilization of existing Linux standard tools and libraries without the need for compatibility with specific real-time APIs. Moreover, GNU/Linux has a strong community of users and developers, which provides frequent OS updates with new technologies and features [21]. For smaller projects this can be an issue, due to resource limitations [21].

With that in mind, the PREEMPT_RT patch was selected as the best candidate for the development of the real-time application, in the context of this thesis. It is worth mentioning that, like the famous Torvalds/Tanenbaum debate about the obsolescence of monolithic kernels [22], in GNU/Linux there was a long series of debates about various aspects of Linux kernel design choices. One of the most controversial topics was the question on how to add real-time extensions to the Linux kernel [23].

2.3.1 The PREEMPT_RT Patch

A few years ago, a great endeavour started in the Linux community. Its ultimate goal was to convert the Linux kernel into a Real-Time Operating System (RTOS), without the need of a

[1] https://wiki.linuxfoundation.org/realtime/rtl/start

microkernel [5]. For achieving this objective, structural changes to the kernel's internals were necessary. For instance, the ISRs should not unconditionally preempt processes running on CPUs and unbounded priority inversion should not be allowed.

Ingo Molnar, a major contributor to the Linux kernel, started his own patch (one among many efforts) against the mainline kernel in order to add real-time features [5]. He wanted to enhance the Linux kernel with real-time features that would improve the user's experience [5]. Molnar started his RT patch and several other kernel developers joined his project. The project matured and became a real-time alternative. Features developed in the patch have been mainlined, including high-resolution timers, kernel lock validation, generic interrupts for all architectures, robust futexes, and priority inheritance. Currently, the project is continued under the context of Real-Time Linux Collaborative Project. Head of maintaining the latest release is Thomas Gleixner and head of maintaining past releases is Steven Rostedt. Some of these features are described below.

2.3.1.1 Interrupts As Threads

In the Linux kernel, when a device performs an asynchronous event, it sends an interrupt signal that preempts the CPU to perform the Interrupt Service Routine (ISR), for the device that issued the interrupt [5]. Then, the ISR is executed at a higher priority than any user task, and with interrupts disabled (or masked off) on the CPU. Thus, the ISR can be preempted only by another interrupt, and only if the ISR re-enables the interrupts. Interrupt work is normally divided into two parts: top half and bottom half. The top half is implemented by the interrupt handler. The bottom half is implemented by softirqs, tasklets or work queues initiated from the top half, or by the interrupt thread in case of threaded interrupt. A device driver puts as little work as possible into the ISR and pushes other work to a tasklet, a softirq or a work queue. These methods are analyzed below:

- *kernel thread*: A *kernel thread* is a thread residing in the Linux kernel. It can be awakened by an ISR to handle any work left, so that the ISR can return quickly and allow the process which was preempted, to resume. A kernel thread is similar to other threads in Linux. For instance, it can be scheduled, have its priority changed or pinned to specific CPUs, just to name a few operations.

- *softirq*: A *softirq* is a service routine that is performed after the return of an ISR and before resuming the process that was interrupted [5, 61]. If too much work has been queued in softirq context, the kernel wakes up a kernel thread (*ksoftirqd*) to finish it.

There's been debate in the Linux kernel community as to what qualifies as "too much work" [5, 62].

- *tasklet*: A *tasklet* has similarities with a softirq, in the sense that it also occurs after an ISR and before resuming the interrupted process. A tasklet can run on only one CPU at a time, while a softirq can run simultaneously on two separate CPUs [5, 62]. Tasklets are implemented internally, by a softirq. The softirq function that implements tasklets, merely ensures that two tasklet functions are not running at the same time [5]. Consequently, tasklets are also executed by a *ksoftirqd* thread [5]. In a *ksoftirqd* thread, softirqs are serviced in the order depicted in Figure 2.3.

- A *work queue* queues up work to be run in a *worker kernel thread*. *Works* are placed in the work queue to be executed sequentially and the worker kernel thread provides asynchronous execution of works from it. The work queue works in a FIFO manner, which means that the worker thread calls the works in turn [5]. Work performed in a work queue can block or be preempted, which may be desirable in situations where resources are requested but are not available [63]. Their simplicity are a reason for utilizing them rather than creating custom kernel threads.

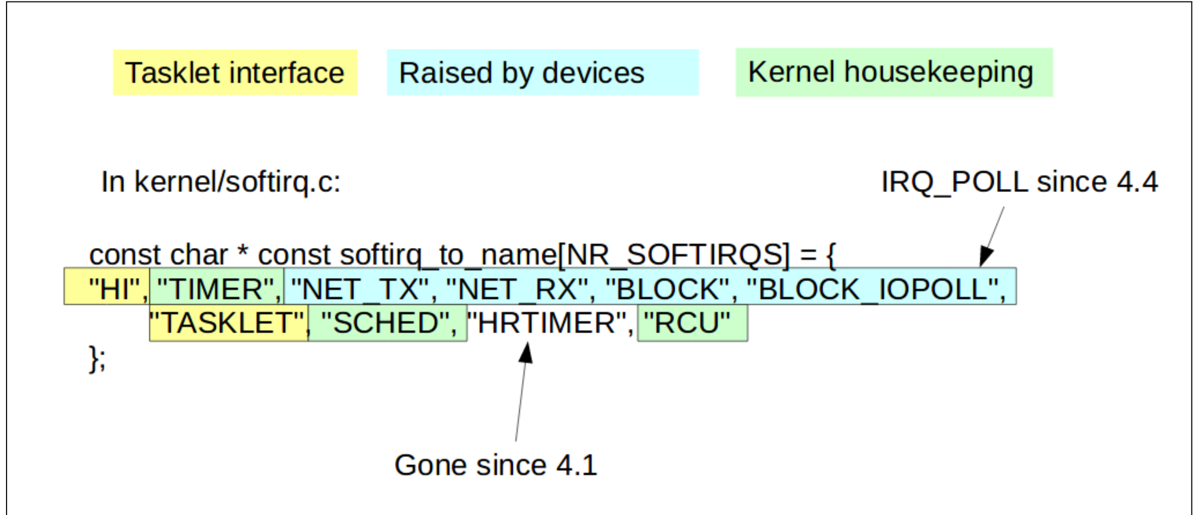


Figure 2.3: *Priority order of execution in ksoftirqd thread [4].*

In the Linux kernel, ISRs, softirqs and tasklets compose the highest-priority entities. Therefore when they are executed, they preempt the process which is running on the CPU. This behavior however introduces high latencies in the system, thus the RT patch transforms all of them into kernel threads.

Hard IRQs As Threads: A hard Interrupt Request (IRQ) mainly consists of an Interrupt Service Routine (ISR). It starts when the interrupt preempts the CPU and lasts until the ISR returns the CPU back to normal processing [5]. If an ISR preempts a high-priority process in order to service a lower-priority work, interrupt inversion happens. In Figure 2.4, the latency introduced by an ISR, which preempts a high-priority process, is illustrated [5]. The latency includes the two arrows (the context switch latency), in addition to the ISR running time [5].

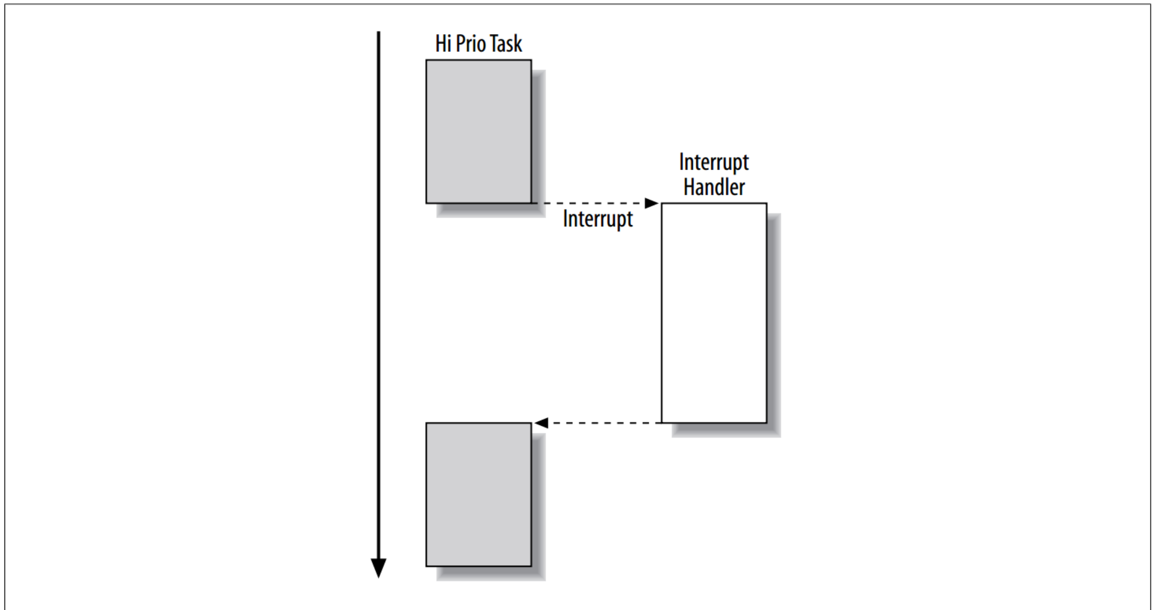


Figure 2.4: *Interrupt inversion [5].*

It is evident that the hardware interrupt has to preempt the CPU. However, the RT patch minimizes the time of the interrupt inversion, by converting the interrupt handlers into kernel threads [5]. In this way, when the interrupt is triggered, the ISR merely wakes up a kernel thread that will run the registered function by the driver, instead of the ISR running the interrupt handler itself [5]. This threaded interrupt handling by the RT patch, is illustrated in Figure 2.5.

With the threaded interrupt handling, the preemption of the CPU when an interrupt is triggered is still unavoidable. However, with this handling, there are only two actions performed before giving the CPU to the previously running task; the interrupt lines are masked and the interrupt service kernel thread is awakened [5]. Thus, if the awakened thread has higher priority than the preempted task, then it will preempt again the previously preempted task [5]. Otherwise, the previously preempted task will continue to run [5].

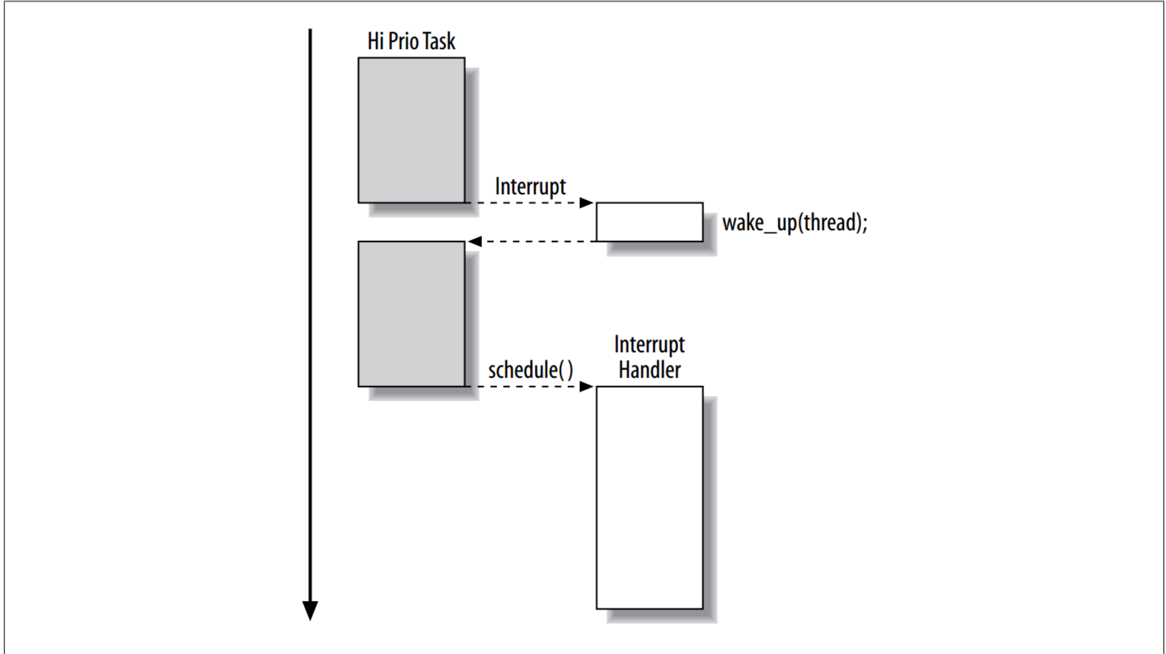


Figure 2.5: *Threaded interrupt handling [5].*

Softirqs As Threads: There are two places where software interrupts are run and preempt the current thread, as illustrated in Figure 2.6 [61, 5]. One of them is at the end of the processing for a hardware interrupt; it is common for interrupt handlers to raise softirqs, so it makes sense (for latency and optimal cache use) to process them as soon as hardware interrupts can be re-enabled [61]. The second option is when kernel code re-enables softirq processing [61]. The final result is that the accumulated softirq work (which can be substantial) is executed in random intervals and preempts the process which happens to be running at the wrong time; this was a major latency issue that needed to be addressed.

Until 3.0 kernel, the real-time patches have traditionally pushed all softirq processing into separate threads, each with its own priority [5, 61]. This allowed, for example, the priority of network softirq handling to be raised on systems where networking needed real-time response; similarly, it could be lowered on systems where response to network events was less critical [5]. However, the process of tuning the priorities of these threads could be a hard task. Since 3.6.1-rt1 patch, the handling of softirqs has changed again [61]. When a thread raises a softirq, the specific interrupt in question (i.e. network receive processing) is remembered by the kernel [61]. When the thread exits the context where software interrupts are disabled, that particular softirq (and no others) will be run. This has the effect of minimizing softirq latency (since softirqs are run as soon as possible) [61]. Equally important is the fact that it also ties processing of softirqs to the processes that generate them [61]. For instance, a process rais-

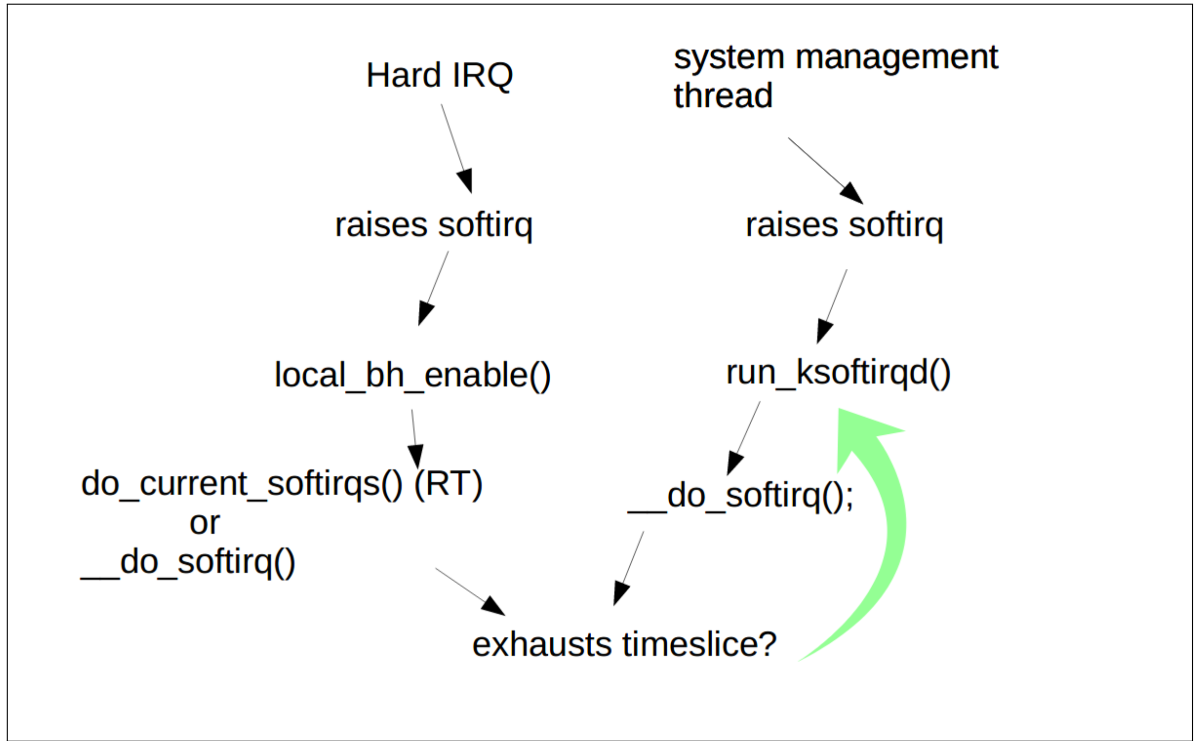


Figure 2.6: *Two paths by which softirqs run [4].*

ing network-related softirqs will not be obliged to process another process's timers. Thus, the work is kept local, non-deterministic behaviors caused by running another process's softirqs are avoided, and softirq processing is run with the priority of the process which created the work [61]. In conclusion, the PREEMPT_RT patch handles differently from the mainline kernel the time the softirq runs, after there was a hardware interrupt (the first path), yet the path triggered by the *ksoftirqd* re-enabling softirq processing (the second path) is mostly the same between the patch and mainline.

2.3.1.2 Priority Inheritance

Since 2.6.18 kernel, priority inheritance is part of the mainline Linux kernel [5]. The first entities in the RT patch incorporating the priority inheritance scheme, were the userland fast mutexes (futex). Nowadays, the futex priority inheritance algorithm is the one used for internal locks in the RT patch [5]. It should be noted that priority inversion isn't a problem, unless it is unbounded, which means that the time the process with higher priority must wait for the blocked resource, is not predictable [5].

The classic example of unbounded priority inversion consists of three processes, A, B, and C, where A has the highest priority and C has the lowest. C starts first, acquires a lock and then is preempted by A. A is trying to take the same lock that C has, but must block and wait for C

to release it. A gives the CPU back to C so that C can finish its work that needed the lock. But B comes along, preempts C, and runs for some unpredicted amount of time. Consequently, B is not only preempting the lower-priority process C but also the higher-priority process A, since A was waiting on C. This is unbounded priority inversion, and it is illustrated in Figure 2.7 [5].

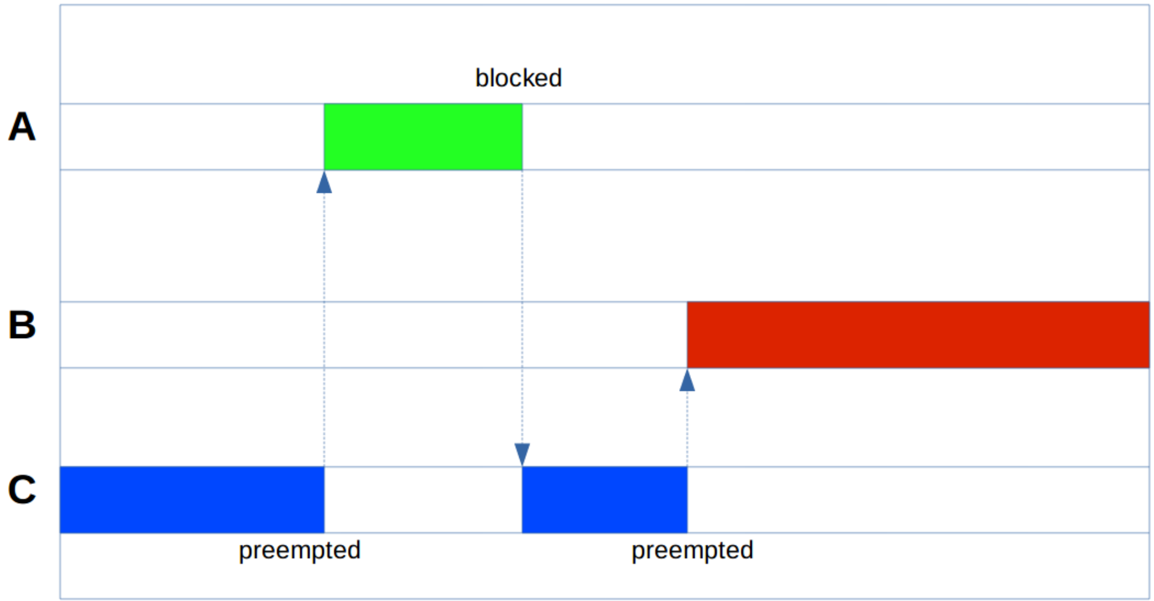


Figure 2.7: *A priority inversion example [6].*

Generally, there are various methods to address priority inversion. The RT patch utilizes the *priority inheritance* approach [5]. In the classic example illustrated in Figure 2.7, the priority inheritance approach works this way: C starts first, acquires a lock and then is again preempted by A, then A tries to take the same lock that C has but must block and wait for C to release it. At this point, the priority inheritance algorithm takes place: C gets the maximum priority of the processes waiting for the lock C has, so in this example the priority of A. Consequently, B cannot preempt C, when it wakes up and tries to. Then, C finishes its work that needed the lock, releases the lock, then A acquires it (as the process with the highest priority), does its work, sleeps and then B runs as expected. The priority inheritance approach is illustrated in Figure 2.8.

The priority inheritance algorithm, was first utilized in futexes, solving the problem of unbounded priority inversions [5]. The futex is a way to perform locking in user-space without the need to enter the kernel, apart from cases of contention. It is similar to a mutex, except it doesn't make unnecessary system calls [5]. The futex, using shared memory and atomic operations (supported by hardware), acquires and releases mutex locks without the overhead of a

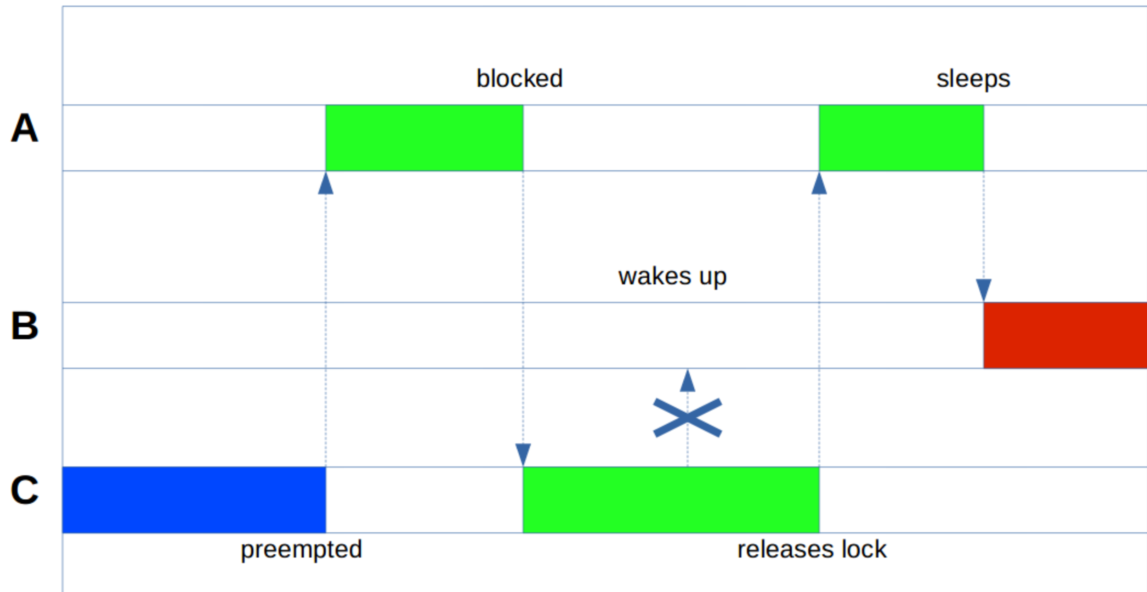


Figure 2.8: *A priority inheritance example [6].*

system call (in user-space) [5]. When contention takes place, for instance when a thread waits for a mutex and needs to sleep, the thread releasing the mutex, notices the thread waiting for the mutex and makes a system call to wake up the sleeping thread [5].

In order to solve the problem of a blocked orphaned futex (the owner of the futex didn't release it and was terminated), robust futex[2] was invented. In a nutshell, the robust futex has a layout which the kernel reads and knows what futexes a thread may have on exit [64]. Thus, when a thread terminates, the kernel can unlock its locked futexes and signal the waiting threads to wake up [5]. In this way, applications don't need to bother whether a thread holding a futex dies and locks up the rest of the application [5].

Futexes can be used among processes, apart from threads, provided they have a piece of shared memory [5]. POSIX mutexes implement futexes in the latest distributions. More information regarding futexes, can be found in [64, 65, 66, 67].

2.3.1.3 High-Resolution Timers

The most essential from all the characteristics that distinguish a real-time system, is the ability to trigger an event at a specific time (otherwise the term *real-time system* has lost its meaning) [5]. Until kernel 2.6, the smallest unit which represented time was the *jiffy* and the `HZ` global variable, represented the hertz of jiffies [5]. The timers used, would create interrupts based on an amount of jiffies, which in turn were depending on the value of `HZ`. For instance, if

[2] `https://lwn.net/Articles/177111/`

the HZ frequency was 1000, a jiffy had a resolution of $1\ ms$ $(1/1000)$ and therefore a timing event could be scheduled at minimum after $1\ ms$. Moreover, when a jiffy in time would pass, a timer interrupt was needed to update the jiffy variable [5].

In addition, the timer accounting was done in a *timer wheel*. The timing events would be recorded into the timer wheel. The timer wheel consisted of layers of "buckets". In the first layer, each bucket represented a future jiffy, with the first layer having 256 buckets. For instance, if an application needed to be notified 20 jiffies into the future, that event would be recorded into the 20th bucket of the first layer. If the event would be more than 256 jiffies into the future, it would go to the next layer of buckets, where each bucket represented 256 jiffies. If the event would be more than $65,536$ (256×256) jiffies, it would be placed to the third layer of buckets. When the time would reach the last bucket of the first layer, the events on the second layer would need to be rehashed into the first layer. The rehashing operation required the interrupts to be disabled and had $O(n)$ complexity (where n is the number of items in the bucket for rehashing) [5].

Thomas Gleixner, a major contributor to the RT patch, tried to solve this issue with a new design of timer infrastructure called *hrtimers* [5]. He realized that the timers placed into the timer wheel belonged to two distinct types: *action timers* and *timeout timers*. Action timers are timers that are expected to expire [5]. Applications use action timers regularly, in order to be notified for events. If the action timer is placed in the upper layers of the timer wheel, it will get rehashed again and again until it reaches the first layer. The complexity of adding / removing a timer has $O(1)$ for the timer wheel, however the rehashing has $O(n)$ complexity. Therefore, the timer wheel isn't efficient for action timers, since these timers require more rehashings than additions / deletions.

On the other hand, timeout timers are ideal for the timer wheel [5]. Timeout timers are timers that fire if an event was missed. For example, the network stack in Linux uses many timeout timers. For instance, a timeout timer will fire when a packet didn't arrive in time, thus telling the kernel that another acknowledgment should be sent [5]. These timers are added / removed constantly, therefore these operations should have as low overhead as possible. Consequently, the timer wheel is a good match for the timeout timers [5].

As a result, hrtimers handled the action timers and the timeout timers remained in the timer wheel [5]. The hrtimer infrastructure uses a red / black tree instead of hashes. Thus, the complexity for adding / removing nodes becomes $O(logn)$. The first node of the tree can be

found in $O(1)$ time, with the help of hooks introduced by the algorithm on the tree. However, the major advantage of using this tree, is that the nodes in it are sorted, which translates to no cost of rehashing, on the contrary to the timer wheel. The hrtimer infrastructure was mainlined in Linux 2.6.18. After many improvements, the $O(1)$ scheduler eventually was replaced by the Completely Fair Scheduler (CFS), which is the default process scheduler used in GNU/Linux ever since kernel 2.6.23 [68]. More information regarding the hrtimers, can be found in [69, 70, 5].

Detailed overview of characteristics and features of the PREEMPT_RT patch can be found in [71, 72, 73, 5], [74, Chapter 17] and [75, Chapter 16].

2.4 Real-time Scheduling in GNU/Linux

Note: This section is largely based on [76].

In Section 2.2, the real-time scheduling algorithms were briefly introduced. In this section, this information is specialized in the context of GNU/Linux, as the ground base of the followed implementation scheme.

As it has been stated in Subsubsection 2.3.1.3, since Linux 2.6.23, the default scheduler is the Completely Fair Scheduler (CFS), which replaced the earlier *O(1)* scheduler. The scheduler is the kernel component that decides which runnable thread will be executed by the CPU next.

In this section, the reader is assumed to be familiar with basic knowledge of the default process priorities and simple system calls for changing scheduling policies, like *nice()*. If this is not the case, a concise explanation of these matters can be found in [76]. The behavior of the Linux scheduler with respect to a process depends on the process's *scheduling policy*, also called the *scheduling class* [76]. Apart from the normal policies, GNU/Linux provides also three real-time scheduling policies, since kernel 3.14.

A preprocessor macro from the header <*sched.h*>[3] represents each policy: the macros are `SCHED_FIFO`, `SCHED_RR`, `SCHED_DEADLINE`, `SCHED_BATCH`, `SCHED_NORMAL` and `SCHED_IDLE`. A static priority is assigned to every process, not to be confused with the `nice` value. For normal (non real-time) processes, this priority is 0 [76]. For the real-time processes except the deadline-classed ones, its range is [1 - 99]. Since the deadline class is a dynamic priority policy, a static priority cannot apply to it, and therefore for consistency reasons this priority

[3] `https://github.com/torvalds/linux/blob/master/include/uapi/linux/sched.h`

is 0 for the deadline-classed processes.

The Linux scheduler always selects the highest-priority process to run (i.e. the one with the largest numerical static priority value) [76]. For example, if a process with a priority of 41 becomes runnable and a process is running with a static priority of 40, then the scheduler will immediately preempt the running process and switch to the newly runnable process [76]. In the same manner, if a process is running with a priority of 40, and a process with a priority of 39 becomes runnable, the scheduler will not run it until the process with priority 40 somehow blocks (i.e. I/O or sleeps or waits for an event). Since normal processes have a static priority of 0, any real-time process that becomes runnable will always preempt a normal process and then run [76].

Moreover, the deadline-classed processes, since they have dynamic priorities, have higher priority even than real-time classed processes like FIFO and RR. Although this section should discuss only the real-time scheduling policies, the non real-time scheduling classes are briefly described for completion.

2.4.1 The first in, first out policy

The *first in, first out (FIFO)* class is a real-time policy without timeslices. A FIFO-classed process will continue running as long as no higher-priority real-time process becomes runnable [76]. The FIFO class is represented by the macro `SCHED_FIFO`.

One of the characteristic features of the FIFO class, is its lack of timeslices, which distinguishes this class from the RR class (see Subsection 2.4.2). Since a FIFO-classed process has real-time policy, once it becomes runnable, it will immediately preempt a normal process. Generally, a runnable FIFO-classed process will always run if it's the process with the highest priority. However there are cases in which this doesn't happen, for instance if this process blocks, yields the processor in which it's running or a real-time process with higher priority becomes runnable [76]. The FIFO class can implement the Rate Monotonic (RM) algorithm briefly introduced in Section 2.2. This is accomplished by assigning to each real-time FIFO-classed process a static scheduling policy, inversely proportionate to its period time[4]. More information regarding the FIFO class can be found in [76].

[4] https://elinux.org/images/f/fe/Using_SCHED_DEADLINE.pdf

2.4.2 The round-robin policy

The *round-robin (RR)* class is almost the same with the FIFO class, except that it imposes additional rules in the case of processes with the same priority. This class is represented by the `SCHED_RR` macro. The distinctive feature the RR class has, is the timeslice. When an RR-classed process exhausts its timeslice, another process with the same priority is scheduled. In this way, RR-classed processes of a given priority are scheduled round-robin among themselves. If there is only one process at a given priority, the RR class is identical to the FIFO class. In such a case, when its timeslice expires, the process simply resumes execution [76].

The decision whether to use `SCHED_FIFO` or `SCHED_RR` is entirely dependent on the intra-priority process behavior. The RR class's timeslices are relevant only among same-priority processes. FIFO-classed processes will run uninterruptible, while RR-classed processes with the same priority will schedule among themselves. A lower-priority process will never run if a higher-priority process exists, whichever policy is chosen [76].

2.4.3 The deadline policy

The *deadline* class is inherently different from the other two real-time classes (FIFO & RR). The static priorities of the processes in this class are `0`, since the algorithm this class represents, assigns *dynamic* priorities to processes. The implemented algorithm is **EDF** (briefly described in Section 2.2), complemented by *Constant Bandwidth Server* (CBS) [77, 78, 79] along with *Greedy Reclamation of Unused Bandwidth* (GRUB) algorithms [80, 81, 82, 83].

The CBS algorithm assigns scheduling deadlines to tasks so that each task runs for at most its runtime every period, avoiding any interference between different tasks (temporal isolation). The GRUB algorithm allows tasks to consume more than their reserved runtime, up to a maximum fraction of the CPU time (so minimum spare CPU time exists for execution of other tasks), provided this doesn't break the guarantees of other tasks. This class is represented by the `SCHED_DEADLINE` macro. It was developed by Evidence s.r.l.[5] in collaboration with ReTiS Lab of Scuola Sant'Anna within the ACTORS[6] European project and it was incorporated in the mainline kernel, since version 3.14 [31].

In a nutshell, a process in this class is defined with three scheduling parameters (defined in nanoseconds):

[5] `http://www.evidence.eu.com/sched_deadline.html`
[6] `http://www.actors-project.eu/`

- *Runtime*
- *Deadline*
- *Period*

These parameters (*Runtime, Deadline, Period*) do not necessarily correspond to the parameters defined in Subsection 2.1.1 (r, C, D, T); common practice is to set Runtime to something bigger than the average computation time (or worst-case execution time for hard real-time tasks) C, Deadline to the relative deadline D, and Period to the period of the task, T. Thus, for this scheduling class, the scheduling parameters are presented in Figure 2.9:

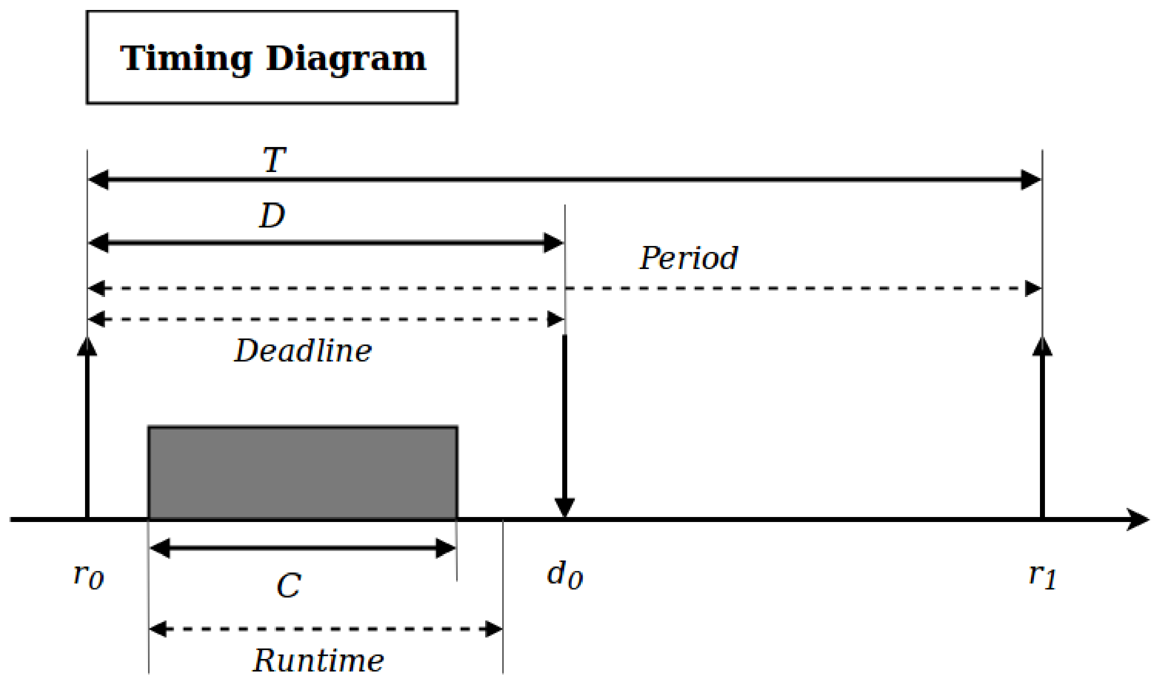


Figure 2.9: *The usual task model of a real-time task defined with the Linux deadline-class parameters.*

If the Period is specified as 0, then it is made the same as Deadline. In general the kernel requires the following inequalities to hold:

$$Runtime \leq Deadline \leq Period$$

In addition, under the current implementation, all of the parameter values must be at least 1024, which is just over one microsecond (there cannot be a resolution of less than a microsecond in these parameters), and less than 2^{63}. If any of the above mentioned checks fails, the process will receive an error by the kernel [84].

To ensure deadline scheduling guarantees, the kernel must prevent situations where the set of deadline-classed processes is not feasible (schedulable) within the given constraints. The kernel thus performs an admittance test when setting or changing Deadline policy and attributes. This admission test calculates whether the change is feasible and if it is not, the process will receive an error by the kernel [84].

For example, it is required (but not necessarily sufficient) for the total utilization to be less than or equal to the total number of CPUs available, where that process's utilization is its Runtime divided by its Period (since each process can maximally run for Runtime per Period). More information about this class can be found in [85, 86, 87, 88, 89, 90, 91, 92, 84], and in the material below[7] ,[8].

2.4.4 The normal policy

The normal policy is the standard scheduling policy and the default non real-time class [76]. This policy is represented by `SCHED_NORMAL`. All normal-classed processes have a static priority of 0 (unrelated with their nice value). Consequently, any runnable real-time (FIFO, RR, Deadline) classed process will preempt a running normal-classed process. Processes with normal policy, are scheduled based on their nice value [76].

2.4.5 The batch policy

This policy is represented by `SCHED_BATCH`. It's the complete opposite of the real-time policies: processes in this class will run only when there are no other runnable processes on the system, even if every other process has exhausted its timeslice [76]. This behavior is different from the behavior of processes with the largest nice values (i.e. the lowest-priority processes) in that eventually such processes will run, as the higher priority processes will eventually exhaust their timeslices [76].

2.4.6 The idle policy

This policy is represented by `SCHED_IDLE`. It is a policy for scheduling low priority jobs. All idle-classed processes have a static priority of 0 (unrelated with their nice value). This policy is intended for running processes at extremely low priority (lower even than a $+19$ nice value with the normal or batch policies) [84]. It was mainlined in kernel version 2.6.23.

[7]https://ti.tuwien.ac.at/ecs/teaching/courses/brds/slides-1/rt-linux
[8]http://retis.santannapisa.it/ luca/TuToR/

More technical information on the scheduling policies in Linux, can be found in [84, 93].

2.5 Robot Operating System (ROS)

Note: This section is largely based on [7]. A basic definition of *ROS*, is provided in the ROS Wiki:

> *ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers*[9]*.*

In other words, ROS is a robot software platform that provides numerous development tools and libraries for easy development of robot applications [7].

ROS is a meta-operating system [7]. This term describes a system that provides functionalities like process management, scheduling, monitoring, memory management, error handling, communication primitives and operability by utilizing a virtualization layer among applications and distributed computing platforms, while running on top of a traditional operating system [7]. This type of software is also called middleware or software framework.

ROS is officially supported to run on top of Ubuntu or Debian [7]. In addition, it has its own application package management system and package conflict resolution. That said, the versatility and plentitude of the different robot application programs developed and maintained by the ROS community, have created an ecosystem that provides distributed packages peer-reviewed and peer-developed. In Figure 2.10, ROS as a meta-operating system is illustrated, controlling robots and sensors with a hardware abstraction layer and offering the tools and libraries for developing robot applications based on existing traditional operating systems [7].

2.5.1 Components of ROS

As shown in Figure 2.11[10], ROS consists of many components including [7]:

[9] `https://www.ros.org/`
[10] `https://wiki.ros.org/APIs`

Figure 2.10: *The ROS Meta-Operating System [7].*

- A client library layer to support various programming languages.
- A hardware interface layer for hardware control (also called Hardware Abstraction Layer or HAL).
- A communication layer enabling data transmission between different components in the ROS environment.
- The Robotics Application Framework which aids at building Robotics Applications.
- The Robotics Applications, service applications which aid in developing custom applications in ROS.
- Simulation tools which can simulate robots in virtual environments.
- Software Development Tools which facilitate the development and debugging of applications.

2.5.2 Basic ROS Terminology

In this subsection fundamental ROS terms are briefly described[11].

Master: The master[12] is the main component of the ROS environment. It behaves like a name

[11] `https://wiki.ros.org/ROS/Concepts`
[12] `https://wiki.ros.org/Master`

Figure 2.11: *ROS Components [7].*

server for the node to node connections and communications. It is responsible for book-keeping the address, name, status and other information of topics, services, nodes and actions. Without the master, the connection among nodes and the communication via mechanisms like topics, services and actions, is impossible. The master communicates with slaves using the XML-RPC (XML-Remote Procedure Call) protocol [94]. This is an HTTP-based protocol that does not maintain connectivity, allowing it to be lightweight, therefore making it suitable for robotic applications. It can be scaled to tenths or even hundreds of nodes [7].

Node: A node[13] is in ROS what is a process in a traditional UNIX operating system; a process that performs computation. It is typical in ROS, every node to have a specific task to accomplish. For example in a robot control system, one node will control a laser range-finder, one will control the wheel motors, another will perform localization and so on. Their use is important in ROS, as they provide modularity and fault tolerance. If one fails, the others will continue to work. A master is also a node. However, if the master crashes, the ROS environment can not work properly, as previously described. Upon startup, a node registers information such as name, message type, URI address and port number of the node. The registered node can act as a publisher, subscriber, server, client, action server, action client or a mixture of the above based on the registered information. Lastly, nodes can exchange messages using topics and services [7].

[13] `https://wiki.ros.org/Nodes`

Package: ROS software is organized in packages[14]. A package may contain multiple ROS nodes, custom (independent of ROS) libraries, third-party developed libraries, configuration files, which constitute a coherent module [7]. The goal of these packages is to provide this useful functionality in a modular manner so that software can be easily reused. Packages should have enough functionality in them, making them reusable by other software but not too much making them heavyweight.

Message: The nodes exchange data via messages[15]. A message is a simple data structure, comprising of typed fields. Standard primitive types (integer, boolean, string, floating point, etc.) are supported, as are arrays of primitive types and there is no limitation in the number of fields defined [7].

Topic: Topics[16] are named buses over which nodes exchange messages [7]. The standard process for using topics is as follows: the publisher node first registers its topic with the master and then starts publishing messages on a topic. Subscriber nodes that want to receive the topic, request from the master to subscribe them to the topic with the specific name. The specified name plays an important role in this process, as there cannot be multiple topics with the same name. Based on this information, the subscriber node directly connects to the publisher node to exchange messages using topics [7].

Publish and Publisher: The term *publish* means the action of transmitting relative to the topic messages [7]. The publisher node communicates with the master and registers its information and topic. Then, it sends a message to connected subscriber nodes that are interested in the same topic [7]. The publisher is declared in a node. A node can have many publishers that publish to the same (or different) topic [7].

Subscribe and Subscriber: The term *subscribe* means the action of receiving relative to the topic messages [7]. The subscriber node communicates with the master and registers its information and topic [7]. Then, receives information from the master related to the publisher that publishes to the relative topic [7]. Based on the received publisher information, the subscriber node directly connects to the publisher node and receives messages from the connected publisher node [7]. The subscriber is declared in a node. A node can have many subscribers that subscribe to the same (or different) topic [7].

[14] `https://wiki.ros.org/Packages`
[15] `https://wiki.ros.org/Messages`
[16] `https://wiki.ros.org/Topics`

Service: A service[17] provides a *synchronous bidirectional* communication between the service client, which requests a service, and the service server, which is responsible for responding to requests [7].

Service Server: A *service server* receives a request as an input and transmits a response as an output [7]. Both request and response are in the form of messages [7].

Service Client: A *service client* requests a service to the server and receives a response [7]. Both request and response are in the form of messages [7].

Action: The action[18] is another message communication method used for *asynchronous bidirectional* communication [7]. Action is used where there is some time for providing a response after receiving a request and intermediate feedback responses are provided until the result is returned [7]. The main difference with services is that actions are representing asynchronous events and processes, whilst services are more close to the traditional definition of a server, communicating in a synchronous manner.

Action Server: An *action server* receives a goal from an action client and responds with a result and/or feedback [7]. The process the server follows can be programmatically defined. The goal, result and feedback are all in the form of messages.

Action Client: An *action client* transmits a goal to a server and receives a result and/or feedback [7]. The goal, result and feedback are all in the form of messages.

Parameter: A parameter[19] in ROS refers to parameters used by nodes [7]. The parameters have default values, which can be modified if necessary [7]. These parameters are stored in the memory of the *parameter server* node, and are retrieved or modified with communication with this server, via the master [7]. Since the concept of parameters is not designed for high-performance or real-time performance, it is best used for static, non-binary data such as configuration parameters.

Parameter Server:

The Parameter Server is loaded in the master, and is responsible for storing parameters, which nodes use (read or modify) [7].

[17] `https://wiki.ros.org/Services`
[18] `https://wiki.ros.org/actionlib`
[19] `https://wiki.ros.org/ParameterServer`

2.5.3 Message Communication in ROS

The message communication mechanisms in ROS are presented here with more details, since they will be useful for understanding the design decisions made in this work. The different message communication primitives are illustrated in Figure 2.12 and a summary of their differences is presented in Table 2.2.

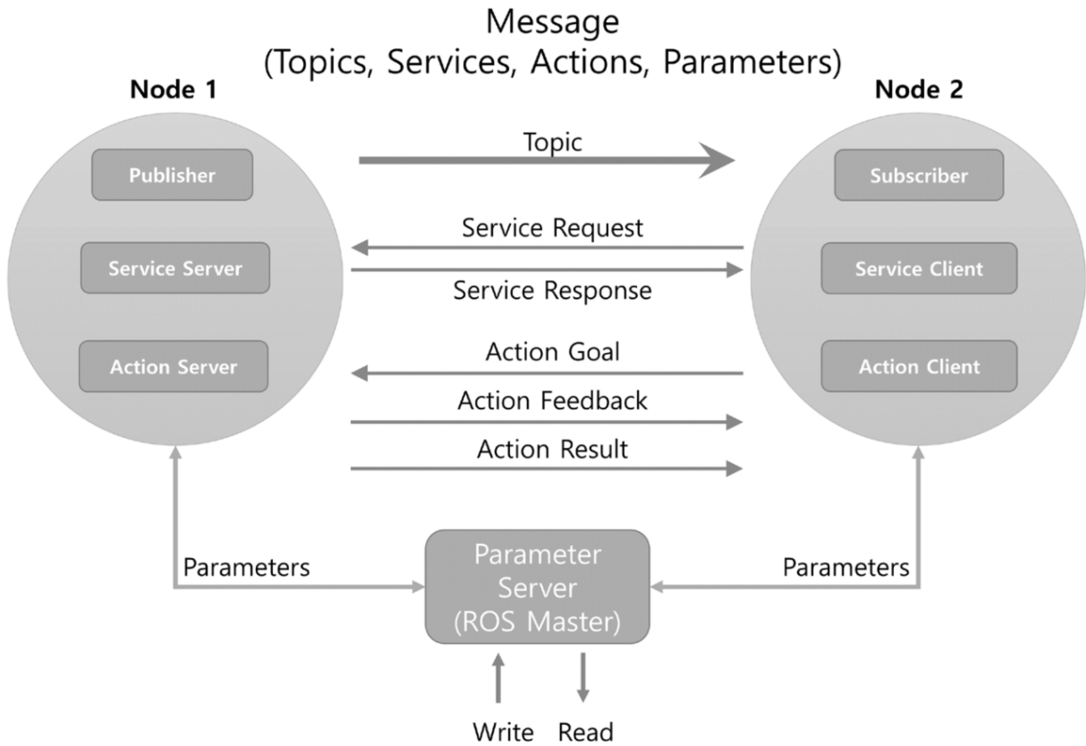


Figure 2.12: *Message Communication between Nodes [7].*

Table 2.2: *Comparison of the Topic, Server, and Action [7].*

Type	Features	Direction	Description
Topic	Asynchronous	Unidirectional	Used when exchanging data continuously
Service	Synchronous	Bi-directional	Used when request processing requests and responds current states
Action	Asynchronous	Bi-directional	Used when it is difficult to use the service due to long response times after the request or when an intermediate feedback value is needed

2.5.3.1 Topic

The topic message communication uses the same type of message for both publisher and subscriber as shown in Figure 2.13 [7]. The publisher node registers its information and topic to

the master and publishes its messages. The subscriber node receives the information of the publisher node corresponding to the specific topic name registered in the master. Based on this information, the subscriber node directly connects to the publisher node to receive the messages published [7].

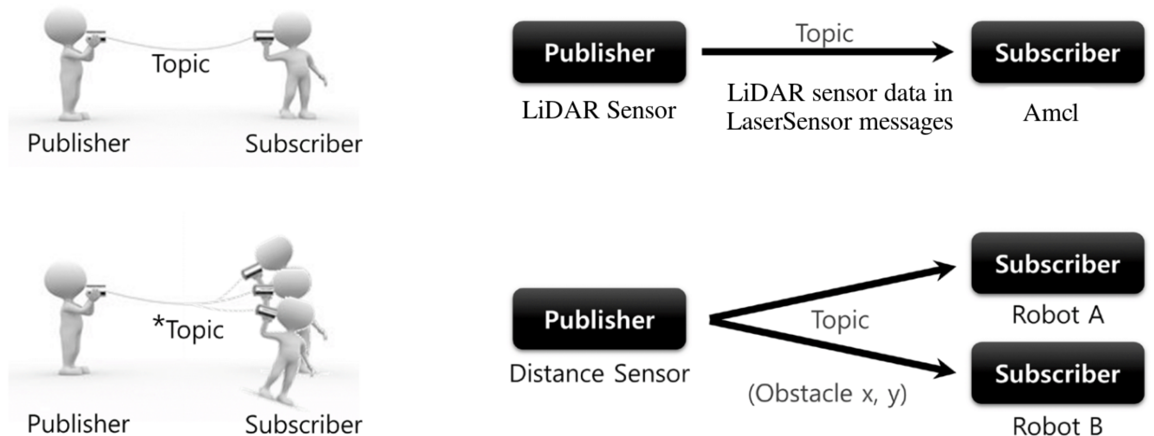


Figure 2.13: *Topic Message Communication [7].*

For example, the node responsible for controlling the LiDAR sensor[20], retrieves the scans and publishes them in the form of messages (in this case the LaserScan type of messages[21]), typically in the topic */scan*. Then the node that wants these messages, e.g. for localization like amcl[22], will subscribe to the topic and after it receives the proper information from the master, it connects to the publisher node and receives the messages.

Since topics are unidirectional and remain connected (TCP connection is used under the hood) to continuously send or receive messages, they are useful in situations which require publishing messages periodically [7]. A message from a publisher can be received my many subscribers and vice versa [7]. Connections with multiple publishers / subscribers can be created too [7].

2.5.3.2 Service

The service message communication is a *synchronous* and *bidirectional* communication between the service client, requesting a service, and the service server, responding to the request as shown in Figure 2.14 [7]. The *topic*, is an *asynchronous* method which is advantageous on periodical data transmission since it is unidirectional and creates and keeps a connection [7].

[20] https://en.wikipedia.org/wiki/Lidar
[21] https://docs.ros.org/melodic/api/sensor_msgs/html/msg/LaserScan.html
[22] https://wiki.ros.org/amcl

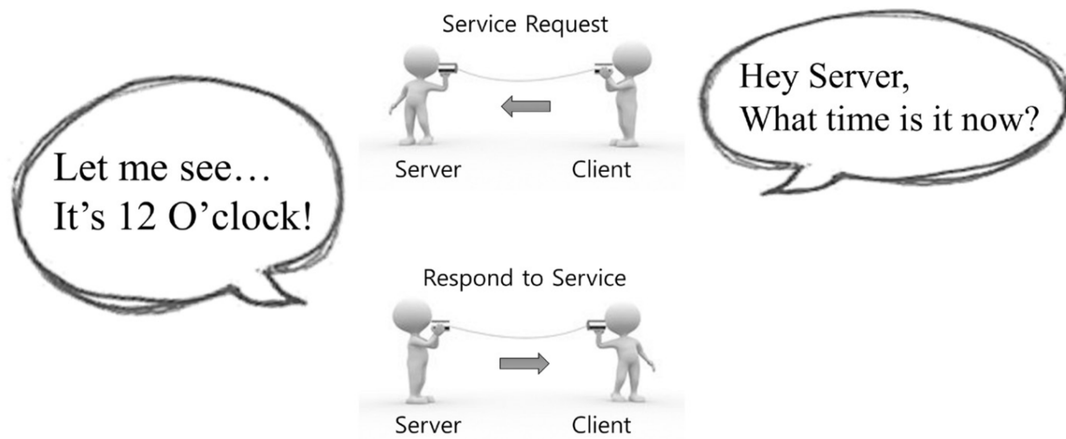


Figure 2.14: *Service Message Communication [7].*

On the other hand, in ROS there is a need for ***synchronous*** communication which uses request and response, much like a traditional DNS server and client [7]. ROS satisfies this need by providing a synchronized message communication method called *service* [7]. A service consists of a service server that responds to a received request and a service client that sends requests and receives responses. A service implements one time message communication, which differentiates it from a topic [7]. Consequently, when the request and response of a service are completed, the connection between the two nodes is lost [7].

A typical scenario for using services, is a ROS node that wants to start another node, so it sends a service request to a service server (similar to a daemon in a UNIX OS) which is active and in its turn wakes up the requested node. As an example, a client sends a request for the current time to a server, as shown in Figure 2.14 [7]. Then, the server will check the time and respond to the client [7]. After the bidirectional communication, connection is lost [7]. Usually there shouldn't be a delay in the server's response. The decision for utilizing the service mechanism over other mechanisms, leads to deciding whether there should be synchronous communication between two nodes in the context of the well-known client-server communication model.

2.5.3.3 Action

The action message communication is an *asynchronous* and *bidirectional* communication between the action client requesting a goal and the action server responding to the goal as shown in Figure 2.15.

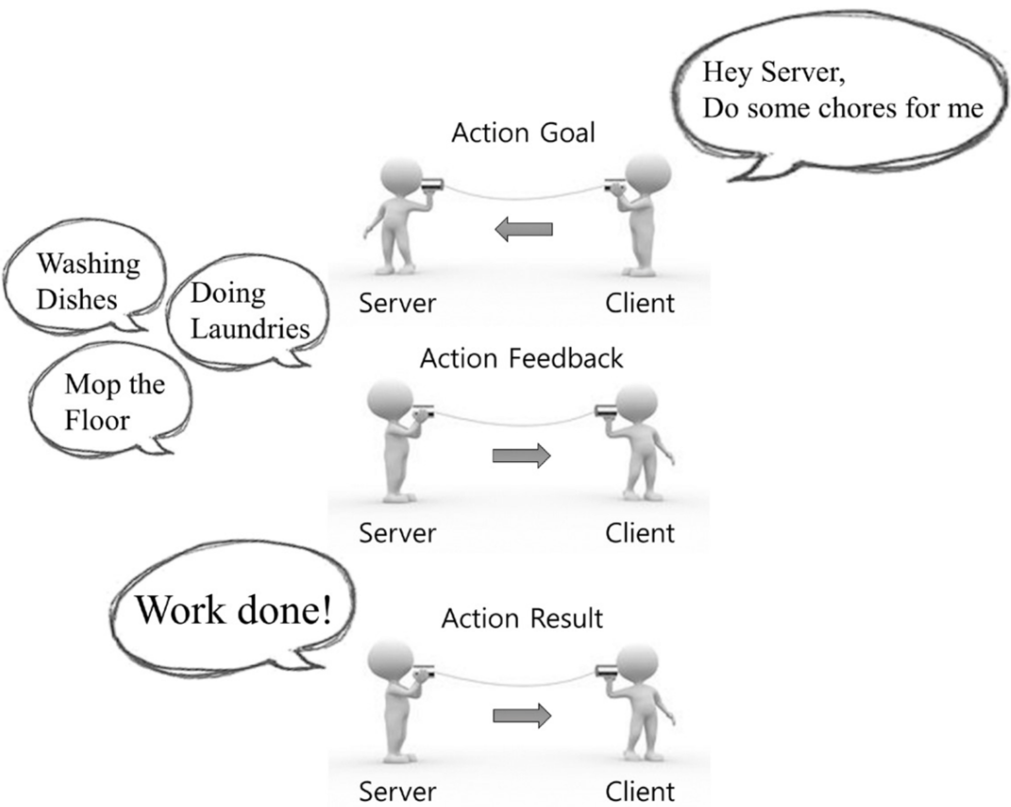


Figure 2.15: *Action Message Communication [7].*

Actions are used when a requested goal takes a long time to be completed, therefore progress feedback is necessary. It is similar to a service, but the key difference between them is the ***asynchronicity*** which characterizes the actions. The message transmission method is similar to the asynchronous topic [7].

For example, as shown in Figure 2.15, a client sets home-cleaning tasks as a goal to the server. Then, the server informs the user of the progress of these tasks in the form of feedback, and finally sends the final result to the client [7]. Unlike the service, the action is often used to command complex robot tasks such as canceling transmitted goal while the operation is in progress. In addition, a typical scenario in which actions are used in ROS is the package responsible for moving the robot, namely the *move_base*[23], which provides a node that implements an action server: it accepts a new goal in the form of a new desired pose of the robot, sends feedback of the current pose of the robot and returns the result pose if the goal pose was accomplished.

Nodes in ROS can have multiple publishers, subcribers, service clients / servers, action clients / servers and communicate with other nodes [7]. In order for the nodes to exchange mes-

[23] `https://wiki.ros.org/move_base`

sages among themselves, the master is necessary for establishing a connection, as shown in Figure 2.16 [7].

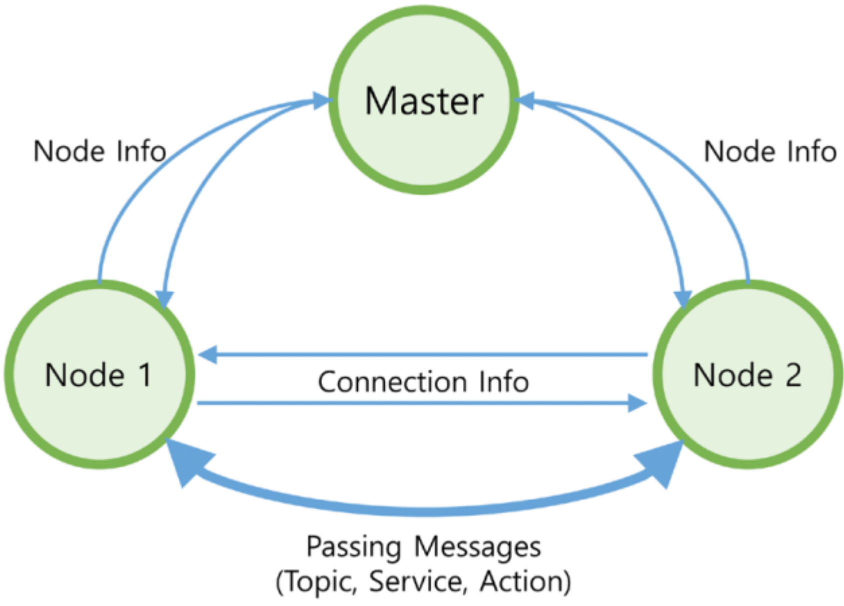


Figure 2.16: *Message Communication [7].*

A master behaves similarly to a name server as it keeps names, URI addresses, port numbers and parameters of all the nodes, topics, services and actions [7]. Nodes register their own information to the master as soon as they are launched, and receive relative information from the master for other nodes [7]. Then, each node connects to each other to perform message communication [7].

3 Background in EtherCAT

Technology is a useful servant but a dangerous master.

Christian Lous Lange

In this chapter, the key theoretical elements for the understanding of this work are provided. At first, the architecture and functionality of EtherCAT, a central component of this project, is described. This is followed by an analysis of EtherCAT masters in GNU/Linux, focusing on their virtues and drawbacks.

3.1 EtherCAT Technology

Note: This section is largely based on [16, 17].

3.1.1 EtherCAT characteristics

Ethernet for Control Automation Technology (EtherCAT)[1] belongs to the Ethernet based fieldbus systems category. Apart from leveraging Ethernet technology, its main features include short cycle times and low communication jitters [1, Chapter 38].

EtherCAT networks adopt the master/slave approach and form *ring topologies*[2] at the physical level [16, Chapter 18]. The master/slave approach implies that there is only one master in an EtherCAT network [16, Chapter 18]. A characteristic example of EtherCAT's utilization is the connection of control units (e.g PLCs) to decentralized peripherals (e.g. sensors, actu-

[1] `https://www.ethercat.org/default.htm`

[2] `https://en.wikipedia.org/w/index.php?title=Ring_network&oldid=887240057`

ators) [16, Chapter 18]. Another feature of EtherCAT is its interoperability with traditional (e.g. TCP/IP stack) as well as other real-time Ethernet (RTE) protocols, like Ethernet/IP and PROFINET [16, Chapter 18].

In an EtherCAT network, the EtherCAT traffic is controlled by the master node [16, Chapter 18]. The master initializes the network for data transmission, by preparing the data exchanges with the slaves [16, Chapter 18]. Each slave processes the received frame in order to extract/insert data from/into it [16, Chapter 18]. Then, the frame is forwarded to the next slave in the ring, as illustrated in Figure 3.1 [16, Chapter 18].

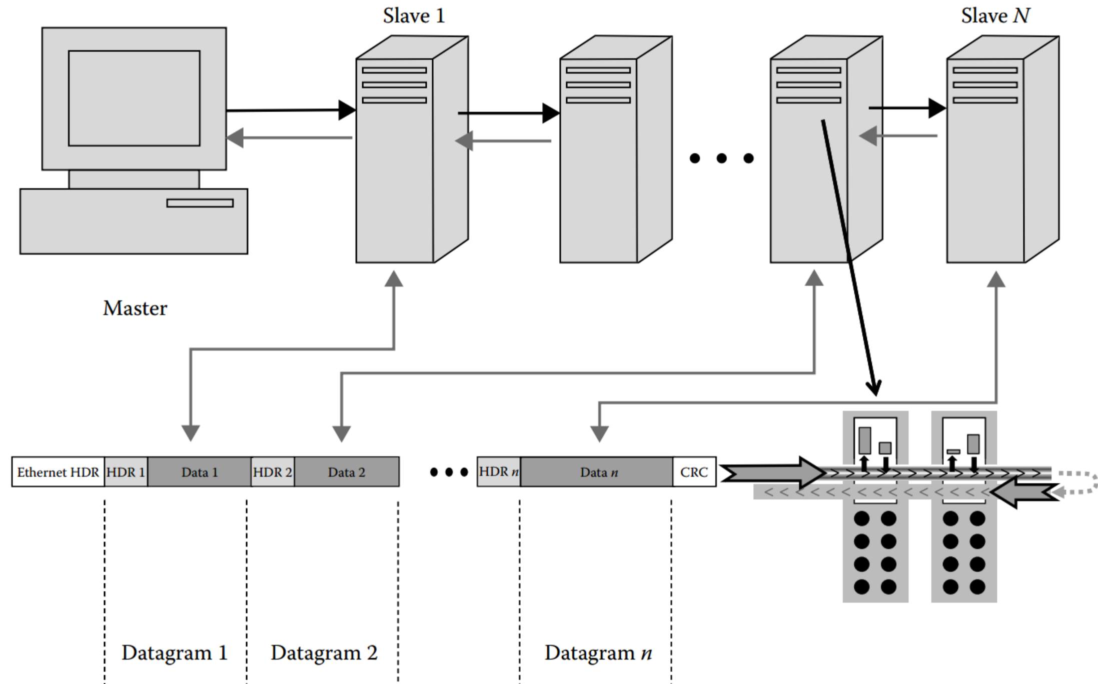


Figure 3.1: *EtherCAT typical topology, with the on-the-fly frame processing [1, Chapter 38].*

3.1.2 Physical Layer

A distinctive feature of EtherCAT, is its famous processing *on-the-fly*, done in the data link layer of the slaves. This feature ensures high performance, however, in order to achieve this, tasks like frame processing and relaying need to happen in parallel in hardware. Consequently, specialized hardware called EtherCAT Slave Controllers (ESCs) is used on the slave's side. The communication is accomplished by the frame passing through a slave with only a minimum delay, and while passing, the slave hardware (ESC) reads the data that is addressed to it and writes a response [16, Chapter 18]. The frame continues to the next slave which reads and writes in the same way, and so on until the frame has passed the last slave [16,

Chapter 18]. At this point the frame turns around and takes the same way back as it came[3] (*ring topology*) [16, Chapter 18]. When received, the master reads the entire frame and takes actions according to the slaves' information [16, Chapter 18].

The EtherCAT protocol can be used in many network topologies (e.g. star), although the one-frame/many-slaves concept requires the topology to be reducible to a logical line (e.g. a simple line, or a more complex tree) [16, Chapter 18]. The key to this concept is that a frame can only travel one way through all slaves, in a well-defined order. Apart from Ethernet, EtherCAT supports EBUS as a physical layer, however, in this thesis the Ethernet physical layer is used. More information regarding the Physical Layer of EtherCAT can be found in [16, Chapter 18].

3.1.3 Data Link Layer

The design of the Data Link Layer of EtherCAT aimed to leverage the available Ethernet bandwith as well as to achieve qualitative communication between master and slaves.

Note: The terms *octet* and *byte* are used interchangeably. The reason there are two terms with the same meaning (an entity with 8 bits of data), is that the former is clearly describing an entity with 8 bits of data, while the latter historically has been used to describe entities with variable amount of bits[4].

3.1.3.1 Frame Format

In an EtherCAT network, the propagated frames, are standard Ethernet frames with EtherCAT frames encapsulated in the data field (payload). As a result, the following Ethernet fields are also included (Figure 3.2):

- Preamble (8 bytes)[5].
- Destination and source MAC addresses (6 bytes each).
- EtherType (2 bytes, set to **0x88A4** to distinguish them from non-EtherCAT frames).
- Frame check sequence (FCS, 32 bits).

[3] although through a different wire, in the case of *full-duplex* Ethernet technology.

[4] `https://en.wikipedia.org/w/index.php?title=Byte&oldid=896613432`

[5] "An Ethernet frame starts with a seven-octet preamble and one-octet *start frame delimiter* (SFD). The preamble consists of a 56-bit (seven-byte) pattern of alternating 1 and 0 bits, allowing devices on the network to easily synchronize their receiver clocks, providing bit-level synchronization. It is followed by the SFD to provide byte-level synchronization and to mark a new incoming frame." [95].

- Inter-frame gap.

The EtherCAT frame, encapsulated in the payload of the Ethernet frame, contains:

- An EtherCAT frame header (2 bytes).
- One or more EtherCAT datagrams.

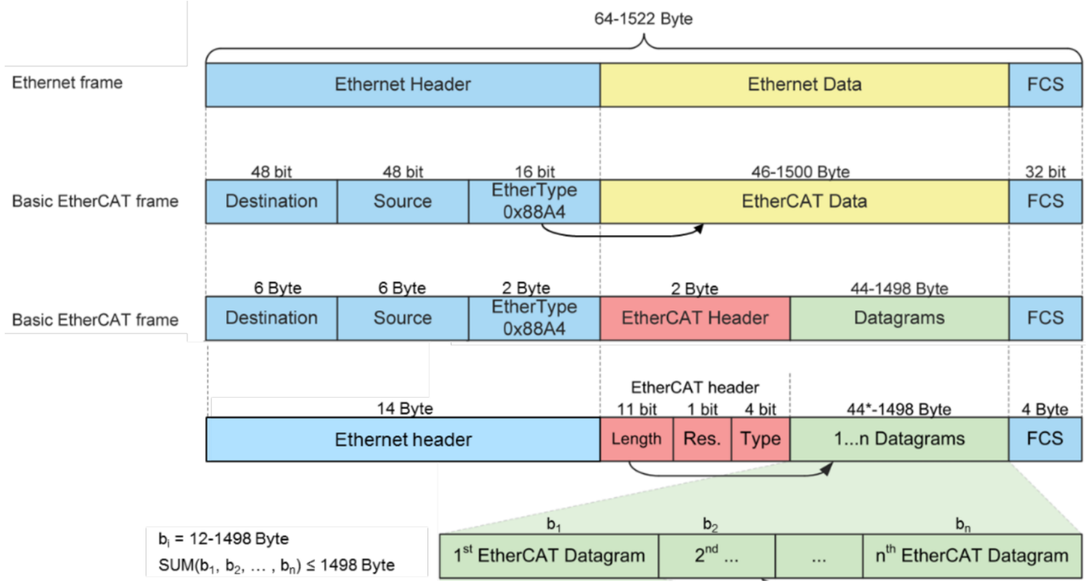


Figure 3.2: *EtherCAT Frame Structure [8].*

The EtherCAT frames are concatenated, without intermediate gaps between them [16, Chapter 18]. With the last EtherCAT frame, the payload of the Ethernet frame is completed, unless its total size is 63 octets or less [16, Chapter 18]. In such a case, the payload is padded with extra 0 bits, so as to have 64 octets, as required by the Ethernet specifications [16, Chapter 18]. The last filed of the Ethernet frame is CRC, which is necessary for checking the integrity of the frame (from the master and the slaves) [16, Chapter 18].

3.1.3.2 EtherCAT datagram Format

As shown in Figure 3.3, each EtherCAT datagram consists of the following fields:

- The *Datagram Header*, which has valuable information for the EtherCAT datagram, including:
  - The type of the service command (*Cmd*).
  - The address of the slave, to which the datagram is targeted to (*Address*).
  - The length of the EtherCAT datagram field *Data* (*Len*).

   – A bit showing if there are more EtherCAT datagrams after the current datagram (M).

- The *Data* field, which can have variable-sized data (0 to 1486 bytes) and includes the information to be exchanged.

- The *working counter* (WKC), which is used for checking if a command has been successfully executed by the relevant slaves.

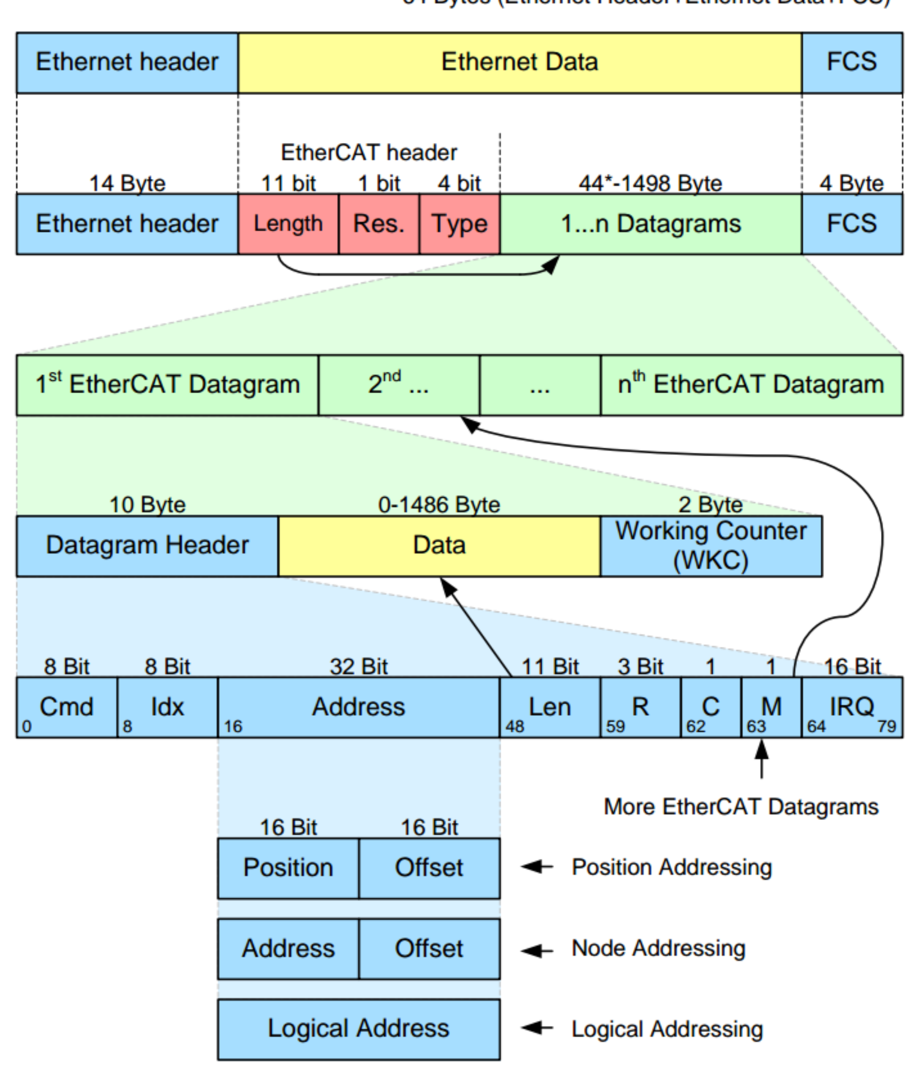


Figure 3.3: *EtherCAT datagram structure [9].*

Regarding the *Cmd* field, there exist different types of commands, which can be used to carry out highly optimized read and write operations on slaves [16, Chapter 18]. Generally speaking, they can be grouped according to the access type [16, Chapter 18]:

- Read (RD) is used by the master to read memory areas or registers from slave devices

[16, Chapter 18].

- Write (WR) is used by the master to write to memory areas or registers of slave devices [16, Chapter 18].
- Read/Write (RW) is used by the master to carry out both a read and a write operation at the same time; in this case, reading is performed by the slave before writing [16, Chapter 18].
- Read/Multiple Write (RMW) is used by the master to carry out a read operation to the addressed slave and a write operation to all the other slaves on the network [16, Chapter 18]. This type of command isn't so common.

Further details on the service commands and the EtherCAT datagram internals can be found in [16, Chapter 18],[1, Chapter 38] and [9].

3.1.3.3 SyncManager

In order to exchange data, the master and the application running on the slave access the ESC's memory [16, Chapter 18]. As a result, concurrency problems may arise if simultaneous access is performed without restriction [16, Chapter 18]. To solve this problem, EtherCAT provides the mechanism of SyncManagers [16, Chapter 18]. They are implemented in the ESC hardware and are configured by the master [16, Chapter 18].

Both the communication direction and the communication mode can be chosen [16, Chapter 18]. Each SyncManager uses a buffer in the local memory area for exchanging data and transparently controls all accesses to the buffer [16, Chapter 18]. The buffer changes take effect immediately after the reception of the end of the frame [16, Chapter 18].

In a nutshell, SyncManagers support two communication modes:

1. **Buffered Mode** (or ***Process Data***): This mode is real-time capable [16, Chapter 18]. In this mode, the producer and the master operations are independent, which means that each entity can access the buffer any time without causing concurrency problems [16, Chapter 18]. The consumer is always provided with the newest data [16, Chapter 18]. In case data are written to the buffer faster than they are read, old data are discarded [16, Chapter 18]. Cyclic process data exchange constitutes the main use of this mode [16, Chapter 18]. This mode is also known as 3-buffer mode, since three buffers of identical size are used [16, Chapter 18]. The first is used by the producer (for writing),

the second by the consumer (for reading) and the third is used for intermediate storage [16, Chapter 18].

2. **Mailbox Mode**: The Mailbox is used for sending larger pieces of data [16, Chapter 18]. They are guaranteed to reach their destination, however real-time guarantees cannot be given [16, Chapter 18]. In this mode, a handshake mechanism takes place prior to data exchange, in order to prevent concurrency issues [16, Chapter 18]. For each mailbox, one buffer is used [16, Chapter 18]. The mailbox mode is typically used for *application layer* (AL) protocols, where the time required to exchange information is usually not very important [16, Chapter 18].

More information for the SyncManagers can be found in [16, Chapter 18] and [1, Chapter 38].

3.1.4 Application Layer (AL)

The Application Layer of EtherCAT is implemented as a state machine, in which the states describe the behavior of the device and the transitions between states are triggered by events [16, Chapter 18]. In each state, different functions are called in the EtherCAT slave [16, Chapter 18]. Similarly, in each state different commands should be sent to the slave by the master [16, Chapter 18].

The state machine is controlled and monitored using some registers included in the slave [16, Chapter 18]. The master controls the state transitions by writing to the *AL control register*, thus creating the corresponding events [16, Chapter 18]. In turn, the slave updates information about its current state by writing in the *AL status register* [16, Chapter 18]. In this way, error notification is performed via error codes written in this register [16, Chapter 18]. As Figure 3.4 shows, an EtherCAT slave supports four basic states and one optional:

- Init: EtherCAT slaves enter this state at power-on. In this situation, the master initializes the SyncManager channels for mailbox communications [16, Chapter 18].
- Preoperational: In this state, mailbox communications are enabled but process data communications are not [16, Chapter 18]. The EM initializes the SyncManager channels for process data, the Field Memory Management Unit (FMMU)s and the Process Data Objects (PDOs) mapping mechanism, if supported [16, Chapter 18].
- Safe operational: In this state, mailbox and process data communications are enabled, but the slave outputs are kept in a safe state, while inputs are updated cyclically [16,

Chapter 18].

- Operational: In this state, slaves can transfer data between the network and their I/O logic. Mailbox and process data communications are completely enabled. The operational state is the normal working condition for slaves after completing the bootstrap phase [16, Chapter 18].

- Bootstrap (optional): The bootstrap state is mainly aimed at downloading the device firmware [16, Chapter 18]. In the bootstrap state, mailboxes are active but restricted to file access via EtherCAT services [16, Chapter 18].

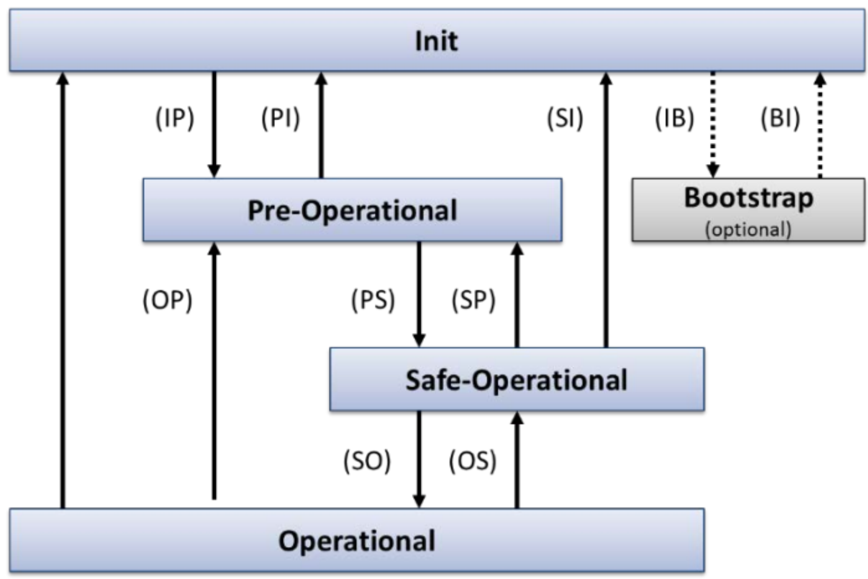


Figure 3.4: *EtherCAT Slave State Machine [8].*

3.1.4.1 Application Protocols

An additive feature of EtherCAT is the support of multiple standard application protocols [16, Chapter 18]. Supported solutions include [16, Chapter 18]:

- *CANopen over EtherCAT* (CoE): This option offers a way to access a CANopen object dictionary (OD) and to exchange CANopen messages according to event-driven mechanisms.

- *Ethernet over EtherCAT* (EoE): This option allows tunneling of standard Ethernet Frames in EtherCAT networks.

- *File access over EtherCAT* (FoE): This option enables the download/upload of firmware and other files.

- *Servo drive profile over EtherCAT* (SoE): This option enables the SERCOS device profile to be used, which is suitable for demanding drive technology.

3.1.5 Distributed Clocks

The task of synchronizing multiple clocks in a distributed system isn't found only in automation systems, but also in many computer and network systems. There are few methods to synchronize slave nodes over a network. One of them is the IEEE 1588 Precision Time Protocol [96] (since 2002), a technology for sharing clocks between distributed systems. IEEE 1588 provides a distributed time base used to timestamp data with sub-microsecond precision and was designed to satisfy the needs of specific markets, which weren't served by either of the two dominant protocols, NTP and GPS [96].

The EtherCAT Distributed Clocks (DC) uses the same concept of distributed time base. Since DC refers to the ESC internal clocks, slave synchronization between slaves corresponding to DC is done in hardware and thus guaranteed to much better than $1\ \mu s$[97].

The clock synchronization process consists of the following three main actions:

i. Propagation delay measurement: At certain time intervals, the master sends a synchronization datagram to the slaves. In this datagram, each slave writes the time measurement of its local clock [16, Chapter 18]. After receiving all the timestamps, the master computes the propagation delay for each segment of the network, while taking into account the EtherCAT network topology [16, Chapter 18].

ii. Offset compensation: Since the local clock of each slave is a free-running counter, usually it doesn't have the same value as the reference clock [16, Chapter 18]. In order to compensate this offset, the master computes the offset between the local clock of each slave and the reference clock. Then, the master writes each offset to a specific register of each slave [16, Chapter 18]. When this step is finished, all devices (master and slaves) share the same absolute system time [16, Chapter 18].

iii. Drift compensation: After the two previous actions are performed, the drift of every local clock is compensated by a *time control loop* (TCL) [16, Chapter 18]. This mechanism corrects the local clock of each device by regularly measuring its difference with the reference clock [16, Chapter 18]. This algorithm has been evaluated and alternative approaches are presented in [11, 97, 98, 99].

3.1.5.1 Propagation Delay Measurement

In each slave, there exist frame processing/forwarding delays, related to internal and communication medium mechanisms [16, Chapter 18]. As a result, the propagation delay introduced between the reference node and each slave should be measured with caution [16, Chapter 18]. The process is the following [16, Chapter 18]:

- The master sends a datagram to all slaves.
- Each slave writes its local clock's value when the first bit of this datagram is received.
- This operation is performed to each port of the slave device, on both the processing and forwarding paths.
- The master receives the timestamps and computes the path delays, taking into consideration the network topology.

More details on this action can be found in [16, Chapter 18].

3.1.5.2 Offset Compensation

When the system starts, the local clock on each device of the EtherCAT network will probably have a different value from the reference clock [16, Chapter 18]. Thus an offset compensation is necessary [16, Chapter 18]. After the propagation delay measurement has finished, the master can compute the offset of each local clock from the reference clock, by examining the previously received timestamps [16, Chapter 18]. Then, this offset is written into a *system time offset* register of each slave and is used to adjust the local time [16, Chapter 18]. Therefore, when the initialization has finished, each slave supporting DC can compute the absolute system time independently, by using the local time and the offset values [16, Chapter 18]. An illustrated example with one slave is presented in Figure 3.5.

3.1.5.3 Drift Compensation

The last action of the DC synchronization process is the compensation of oscillator drifts [16, Chapter 18]. There is a natural drift between the local clock of each device and the reference clock, due to variations between the crystal oscillators used in each device (two clocks are never identical, even from the same manufacturer) [16, Chapter 18]. This drift is corrected by a TCL algorithm implemented into each ESC and shown in Figure 3.6 [16, Chapter 18].

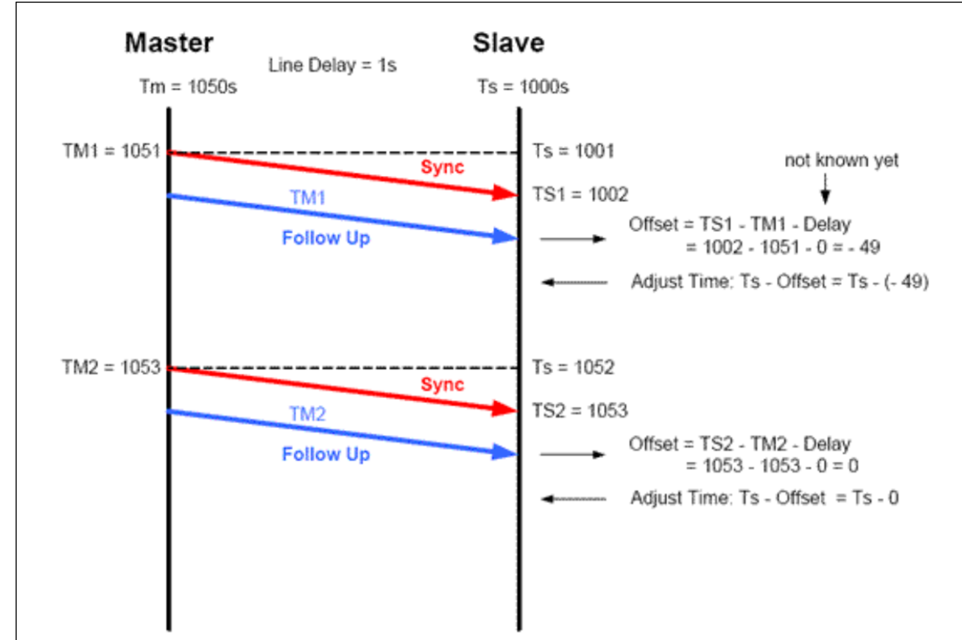


Figure 3.5: *Offset measurement in the DC mechanism [10].*

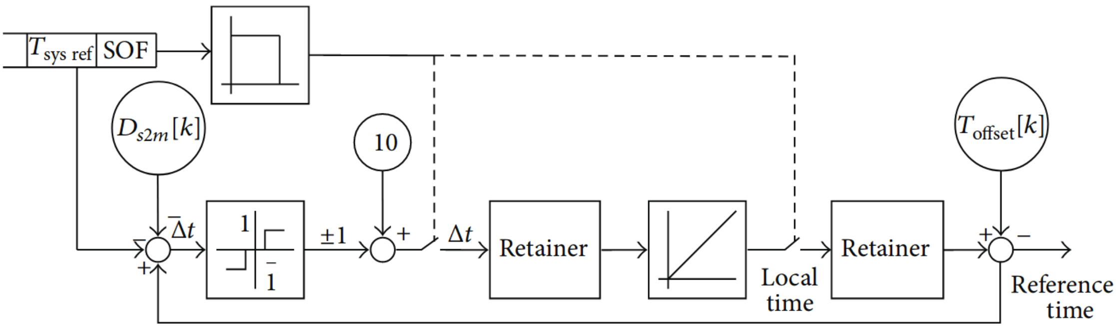


Figure 3.6: *Concept of the TCL algorithm [11].*

3.1.6 Synchronization in the Slaves

A core feature of EtherCAT is its ability to enable automatic synchronization between the master and the slaves in the network, providing an absolute system time to which all devices adjust [17]. Another important characteristic is the flexibility of each slave, to define its own synchronization mode (i.e. not all slaves support DC Synchronization Mode), independently of the other slaves in the network [17]. At application level, the master and the slaves perform software loops (Figure 3.7).

The applications on the master and the slaves exchange process data in predefined time intervals (Figure 3.8) [12]. These intervals can be arbitrarily short, provided that the applications have time to execute their loops [12].

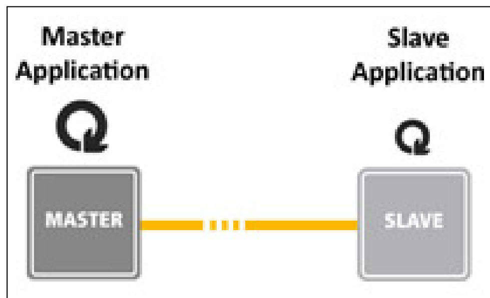


Figure 3.7: *EtherCAT Application Level [12].*

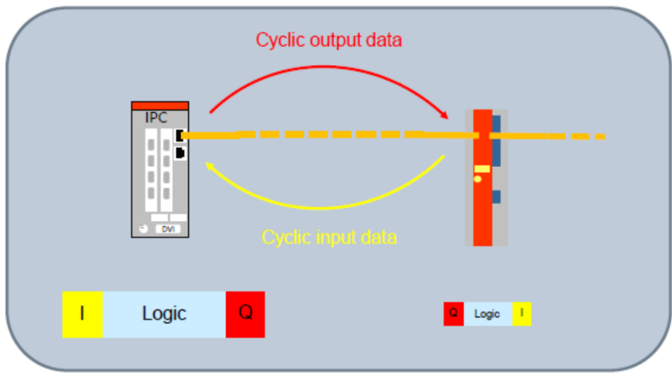


Figure 3.8: *EtherCAT process data exchange [12].*

The applications on the master and the slaves synchronize by defining time relationships between the start time of their cyclic loops, as shown in Figure 3.9 [12].

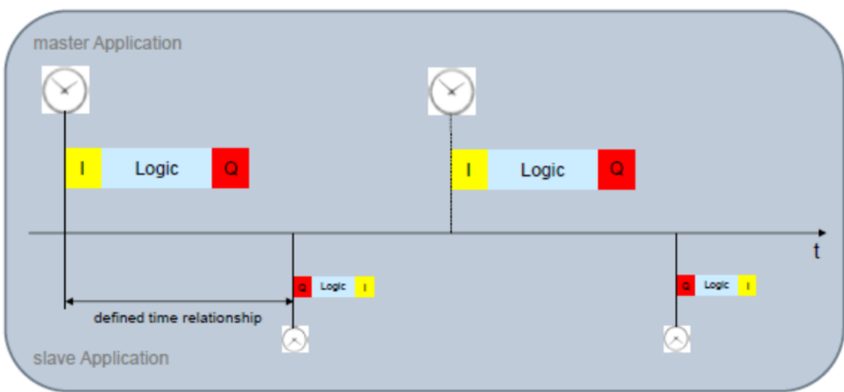


Figure 3.9: *Time between Master and Slave Application [12].*

In EtherCAT there are three main time relationships defined for each slave application with respect to the master cycle (**Synchronization Modes**) [12]:

- ***Free Run*** (no synchronization): Process data handling in the slave is triggered by an internal event. There is no time relationship with the master cycle.
- ***SM Synchronous*** (Sync Manager): Process data handling in the slave is triggered by a

hardware interrupt event generated when the cyclic frame carrying the process data is received.

- ***DC Synchronous*** (Distributed Clocks): Process data handling in the slave is triggered by a hardware interrupt event based on the Distributed Clocks and on the absolute system time.

3.1.6.1 Free Run Mode

When a slave operates in Free Run Mode (Figure 3.10), the execution of the local application is triggered by an internal time source [12]. This mode has the following characteristics [13] (Figure 3.11):

- The cyclic frames and local application don't have a time relationship.
- Time offset among different "Free Run" slaves is not defined.
- Intended for I/O (input/Output) devices handling slow-varying signals.

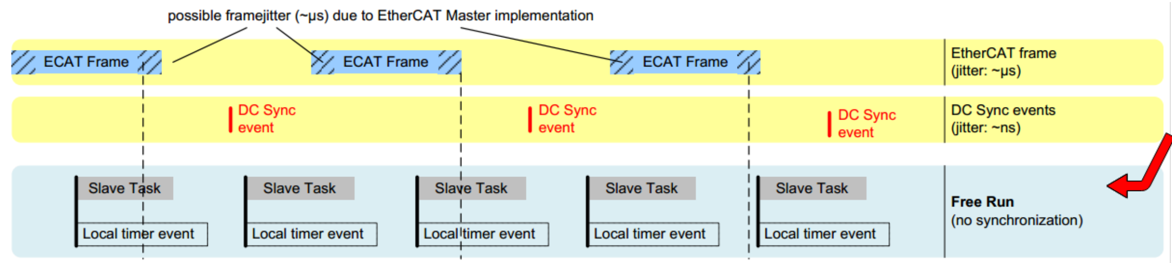


Figure 3.10: *Slave in Free Run mode [12].*

3.1.6.2 SM Synchronous Mode

When a slave operates in SM Synchronous Mode (Figure 3.12), the process data handling is triggered by a hardware interrupt when the cyclic frames are received (Figure 3.13) [12]. The master provides a timer variable for each slave in order for the entire network to synchronize to the reference clock [12]. This becomes a necessity to quadruped control applications, where each slave should adhere to a reference clock in order to create gaiting sequences [17].

Synchronization inaccuracies may occur if a network is configured in SM Synchronous mode which may affect the efficiency of the synchronization [12]. Main reasons are [12]:

- The cyclic frames have jitter due to the master. This jitter is propagated to the slaves' applications.

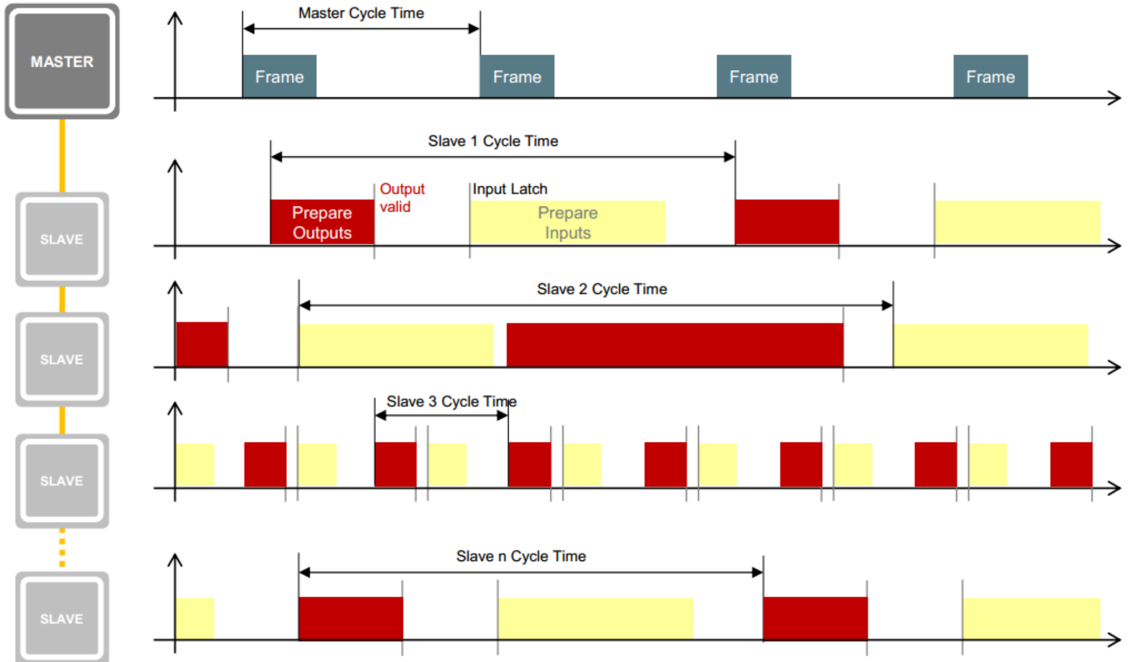


Figure 3.11: *EtherCAT network in Free Run mode [13].*

- The last slaves receive the cyclic frames later than the first due to propagation delays, regardless of the master's jitter.

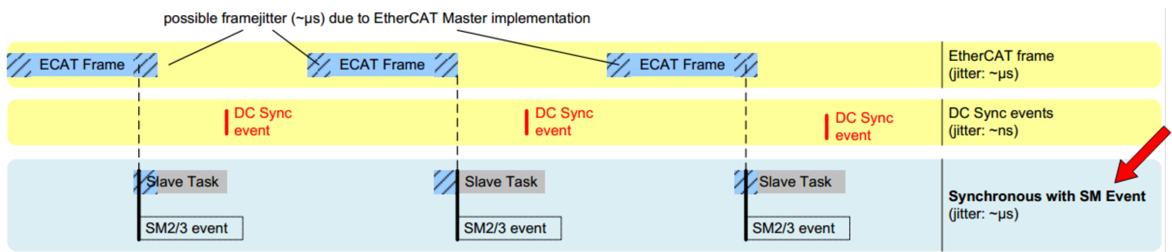


Figure 3.12: *Slave in SM Synchronous mode [12].*

3.1.6.3 DC Synchronous Mode

Using the DC mechanism, as described in Subsection 3.1.5, the devices in a network can be synchronized, allowing distributed applications to synchronize as well (Figure 3.14).

The DC mechanism has many features, including [9]:

- Synchronization of the clock of each device in the EtherCAT network.
- Generation of synchronous output signals (SyncSignals).
- Precise timestamping of input events (LatchSignals).
- Generation of synchronous interrupts.

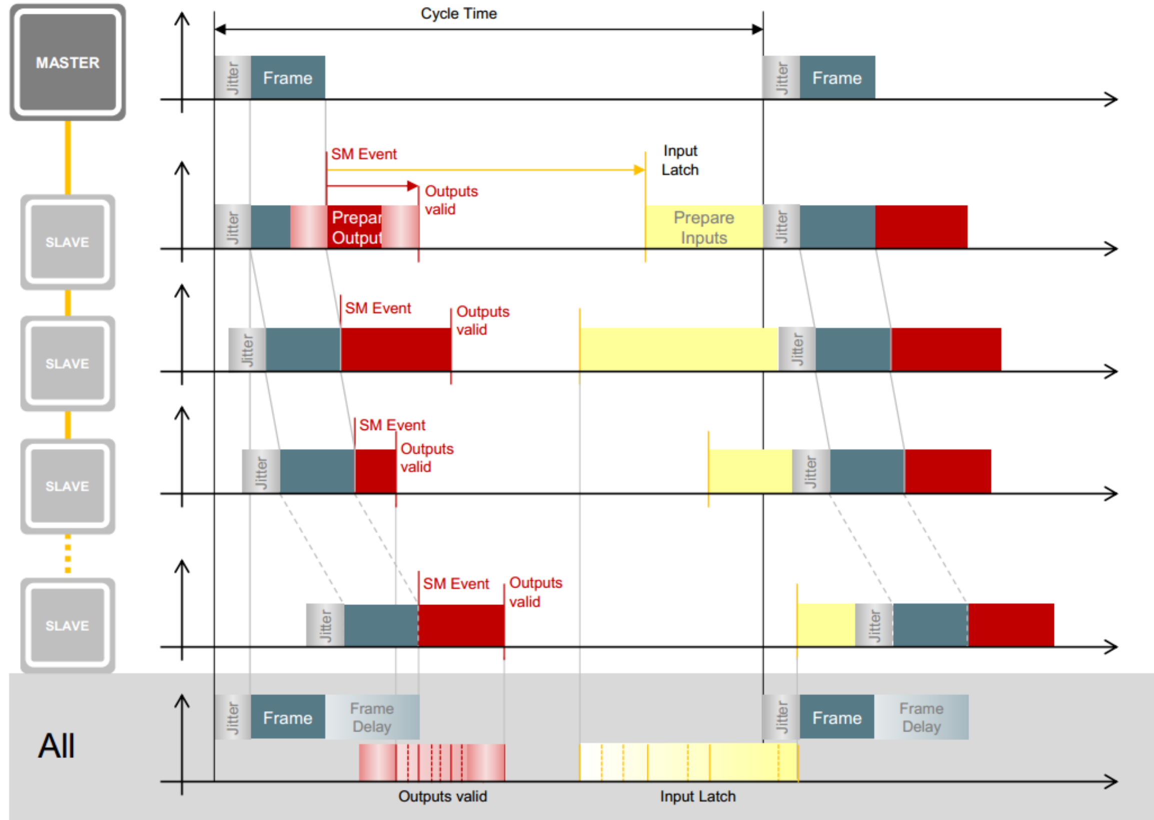


Figure 3.13: *EtherCAT network in SM Synchronous mode [13].*

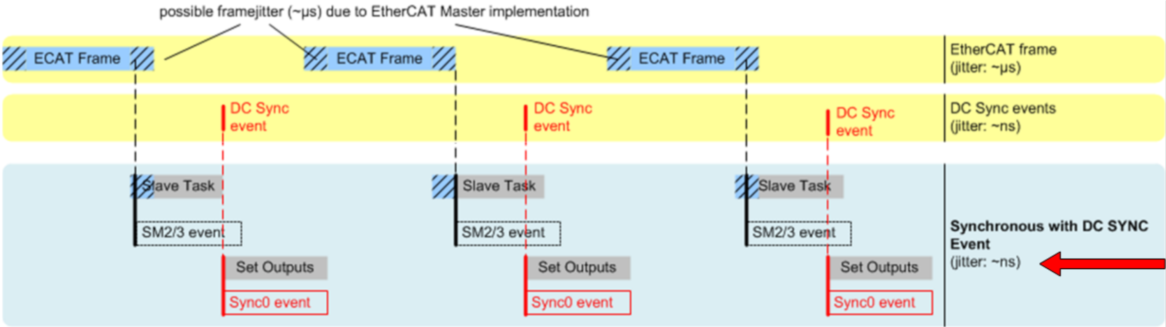


Figure 3.14: *Slave in DC Synchronous mode [12].*

- Synchronous digital output updates.
- Synchronous digital input sampling.

The DC mechanism operates above the Data Link layer of EtherCAT [17]. It's not supported by all slaves, however DC-enabled and non DC-enabled slaves can typically operate together in the same network [17]. It should be mentioned that the DC mechanism depends on specific feaures of EtherCAT, such as its ring topology, datagram processing "on the fly" and hardware timestamping, thus it's not a general-purpose synchronization mechanism [17].

When a slave operates in DC Synchronous Mode, the process data handling is triggered by the hardware SYNC events generated in the slave based on the DC System Time (Figure 3.15). Each generated interrupt signal is serviced by an Interrupt Service Routine (ISR), which is triggered simultaneously in each slave of the network. This ensures intrinsic synchronization among the slaves without using any timer variables. This mode in order to work, requires the cycle frame time to be large enough to allow all ISRs to be triggered in each slave. In case this isn't satisfied, lost frames interfere with the internal synchronization of the slaves, causing communication errors. The advantages of DC Synchronous Mode are:

- The hardware SYNC events (interrupt signals) are generated in each slave automatically by the EtherCAT Slave Controller (ESC). The ESC should be configured to operate in DC Synch mode (specified in the ENI file).
- The process data handling in each slave is not affected by the master's jitter or propagation delays.

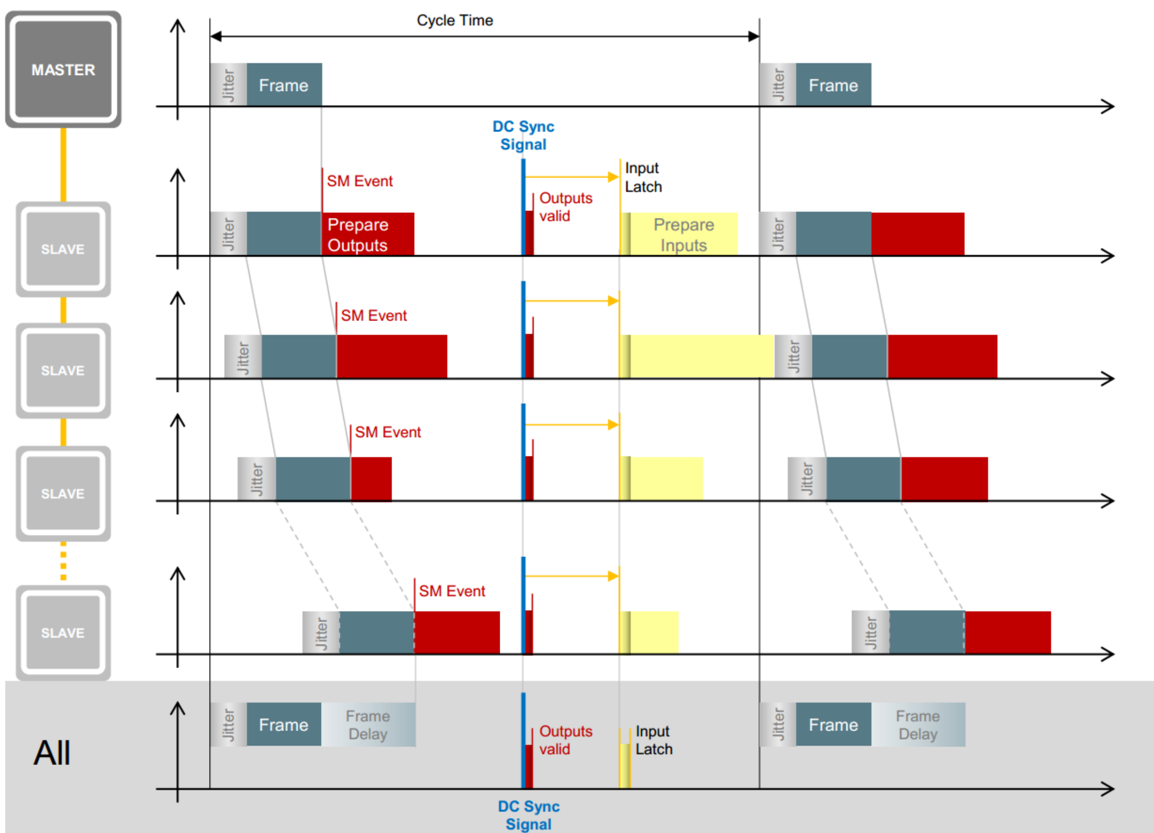


Figure 3.15: *EtherCAT network in DC Synchronous mode [13].*

3.1.6.4 SYNC Shift Times

The application on each slave operating in SM or DC Synchronous mode, needs to be shifted with respect to the application on the master, in order for the slave application to receive the incoming data before it starts a new loop (Figure 3.16).

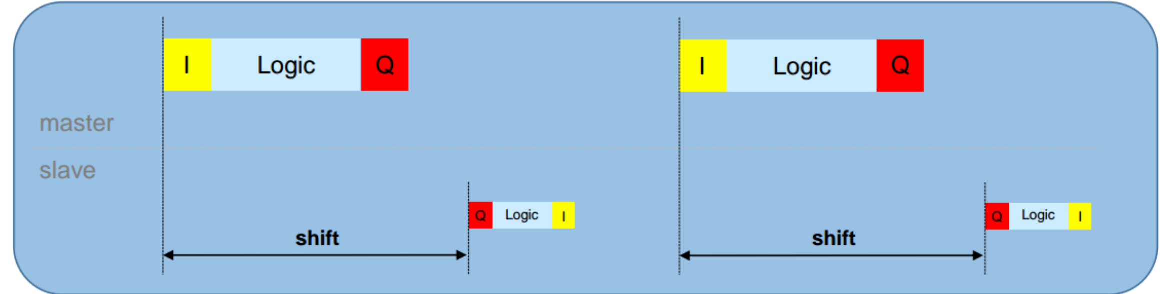


Figure 3.16: *EtherCAT shift time [14].*

In SM Synchronous mode, the slave application is triggered by the incoming frame, thus no extra configuration parameter is needed.

In DC Synchronous mode, the shift between the SM interrupt and the master cycle is set by the master on system start-up. The shift's value is reconfigurable. Setting the time shift in DC Synchronous mode requires caution. The shift's value should guarantee the SYNC event in the slave to be generated after the cyclic frame is received by all slaves in the network and before the next cyclic frame is received by the slave. In the same time, its value shouldn't be affected by communication jitter, propagation delays or the number of slaves (Figure 3.17). Therefore, there doesn't exist a single correct value, but an interval of acceptable values.

The SYNC shift has a lower bound, which consists of the following factors:

- Hardware delay introduced by the slaves internally:
  - 1 µs for every slave of the network with Media Independent Inter face (MII) Ports.
  - 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 each meter of cable in the EtherCAT network [100].

3.1.7 Synchronization in the Master

Note: This section is largely based on [15].

The master operates in two synchronization modes [15]:

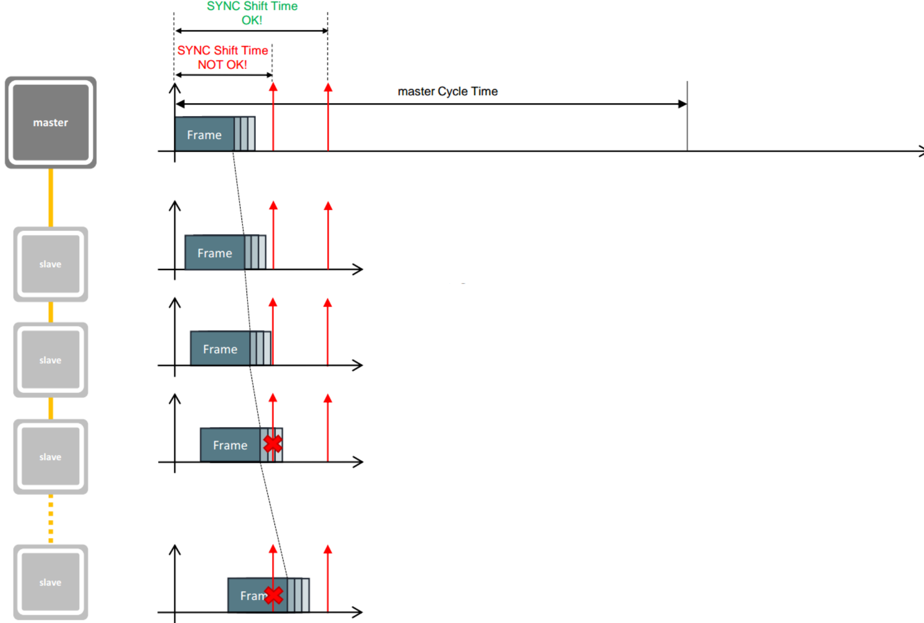


Figure 3.17: *Acceptable vs wrong shift times [14].*

- The Cyclic Mode.
- The DC Mode.

3.1.7.1 Cyclic Mode

Slaves operating in Free Run or SM Synchronous mode are compatible with the *Cyclic Mode*. The master sends the process data frames on certain time intervals [15]. These intervals are controlled by a local timer in the master [15].

3.1.7.2 DC Mode

In *DC Mode*, the master sends the process data frames periodically, similarly to the Cyclic Mode. The difference is that the local clock in the master (and the local clock in each slave) is synchronized with the master clock [15]. In DC Mode, all DC-enabled slaves and the master are synchronized to the DC Base Time, a virtual time which has a fixed time relationship with the reference time (produced by the master clock) [15]. In Figure 3.18, the synchronization between the local clock of the master and the DC Base Time is shown [15].

Note: The master clock is the EtherCAT reference clock, to which all the devices (including the master's local clock) are adjusted. This clock is usually the local clock of the slave closer to the master.

More and technical information can be found in [15].

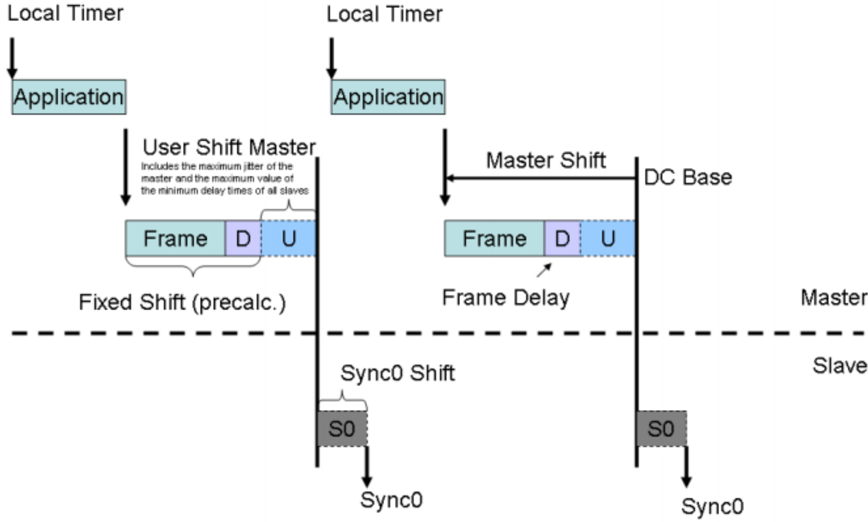


Figure 3.18: *Master synchronized to DC Base [15].*

3.2 EtherCAT Masters

Note: This section is largely based on [16, Chapter 18].

Since the EtherCAT technology is thoroughly described, the next spot of attention goes to the EtherCAT Masters. In this section, the EtherCAT Masters are introduced and a brief comparison takes place. Finally, the preferable EtherLab is briefly introduced.

3.2.1 EtherCAT Masters Overview

EtherCAT Masters (EMs) can be implemented in software [16, Chapter 18]. EM facilitates the use of a control application in the master, which reads and writes process data from/into the slaves' memory [16, Chapter 18]. Typical characteristics of an EM include network monitoring, fault detection and recovery, automatic discovery of the network topology, slaves synchronization and configuration at system start-up [16, Chapter 18].

However, the most critical task of an EM is the control application running in the master [16, Chapter 18]. This application usually realizes a control loop, which starts after the configuration of the slaves has finished [16, Chapter 18]. Thus, the level of determinism of the EM, affects directly the real-time performance of the running control application [16, Chapter 18].

3.2.1.1 Control Loop

The control application usually realizes a basic control loop, as shown with pseudo-code in Figure 3.19 [16, Chapter 18]. The control loop is a cyclic task with a period (T_{CYCLE}) [16,

Chapter 18]. This task includes writing/reading process data to/from slaves [16, Chapter 18].

```
while (TRUE) {
      wait(time = time + TCYCLE);
      data = receive_frame();
      new_data = execute_control_application(data);
      send_frame(new_data);
}
```

R
E
S
NET
T_{CYCLE}

Figure 3.19: *Pseudo-code of a typical EM control loop [16, Chapter 18].*

In a nutshell, the EtherCAT frames received through the EM's network interface are passed to the application through a call to the `receive_frame` function [16, Chapter 18]. In this function call, certain sub-tasks are implied such as extraction of datagrams from the frame, extraction of process data from each datagram, their concatenation and storage into the corresponding variables (`data` in the figure) [16, Chapter 18]. Then the main control algoirthm is executed by calling the execute_control_application function [16, Chapter 18]. When the function finishes, new data (`new_data` in the figure) have been produced and are ready to be transmitted [16, Chapter 18]. These data are passed as arguments to the `send_frame` function, which ensures the data are sent to the EtherCAT slaves, after making the necessary actions such as concatenating the data into datagrams, coalescing the datagrams into a single frame and sending the frame to the EtherCAT network [16, Chapter 18]. After the frame is sent, the application needs to wait time equal to the period, hence the call to the wait function (`wait` in the figure) with argument a time interval equal to T_{CYCLE} [16, Chapter 18].

Iterations of the control loop are shown in the timing diagram in Figure 3.20. The t_{CPU} in

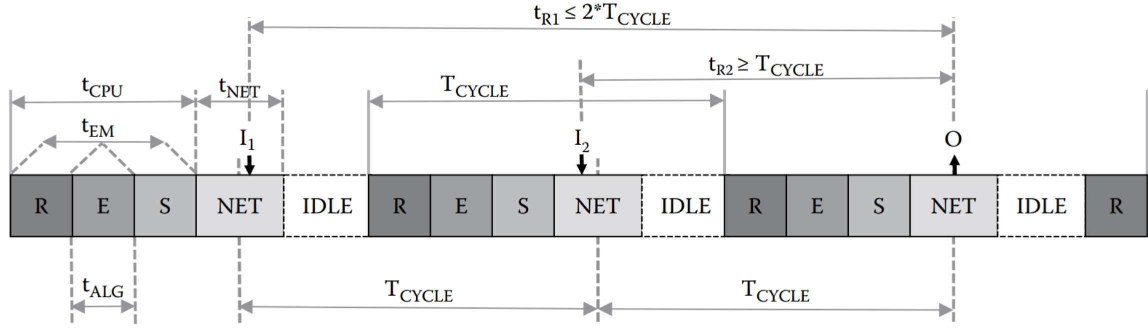


Figure 3.20: *EtherCAT control loop timing diagram [16, Chapter 18].*

Figure 3.20 corresponds to the time spent by the application, to occupy a CPU in the master

and execute the control loop (i.e. the calls to the `receive_frame`, `execute_control_-application` and `send_frame` functions) [16, Chapter 18]. The t_{NET} represents the time needed for the EtherCAT frame to traverse all the slaves in the network and return back to the EM [16, Chapter 18]. The time needed to store the frame received from the network interface into the EM memory is also included in t_{NET} [16, Chapter 18]. The time interval needed for the input data of each slave to be copied from its memory to EtherCAT datagrams is also included in t_{NET} [16, Chapter 18]. Similarly, the time interval needed for the output data of each slave to be copied from the EtherCAT datagrams to its memory is also included in $t_{NET}[16, Chapter\ 18]$.

Since EtherCAT provides determinism, the t_{NET} interval has small variations across iterations of the control loop. In practice, t_{NET}, which depends on the frame size and other parameters, can be computed analytically [100].

However, the t_{CPU} interval is not deterministic [16, Chapter 18]. This interval consists of two time intervals [16, Chapter 18]:

- t_{ALG}: This interval represents the time the control algorithm spends in the CPU. The implementation of the algorithm in code affects directly the t_{ALG}. Usually, t_{ALG} has an upper bound, which can be found analytically or experimentally.

- t_{EM}: This interval includes the latency introduced by the protocol stack to send and receive EtherCAT frames and operating system latencies caused by interacting tasks, context switching, and scheduling. Unfortunately, it's not known a priori whether t_{EM} has an upper bound.

Consequently, the inequality $t_{CPU} + t_{NET} \leq T_{CYCLE}$ might not hold under certain circumstances [16, Chapter 18]. To solve the unbounded intervals of t_{EM}, EM implementations usually leverage hard Real-Time Operating Systems, network drivers and protocol stacks optimized for latency [16, Chapter 18]. For instance, in [101] the performance of an EM using hard Real-Time Operating Systems and protocol stacks is evaluated [16, Chapter 18]. It shows that jitters on the order of $10\mu s$ can be experienced for a $1ms$ cycle time [16, Chapter 18].

On the contrary, EM implementations based on non Real-Time Operating Systems and protocol stacks are typically inappropriate for applications requiring the real-time features of EtherCAT [16, Chapter 18]. These implementations usually mean that t_{EM} is not bounded, thus delays larger than $1ms$ are expected [16, Chapter 18]. In such cases, the performance

using EtherCAT could be worse than non real-time Ethernet protocols like EtherNet/IP [16, Chapter 18].

3.2.1.2 Commercial versus Open-Source implementations

The most important requirement for EM is determinism [16, Chapter 18]. The EtherCAT network provides determinism to the t_{NET} time interval, which is achieved through specific slave hardware [16, Chapter 18]. However the determinism of the master is equally important, as the actions summarized in Figure 3.19 need to be completed in a single iteration of the control loop, in order to ensure deterministic cycle times [16, Chapter 18].

Although EMs based on mixed hardware/software have been proposed in the literature (with Field Programmable Gate Arrays in [102]), this section focuses on EMs implemented purely in software [16, Chapter 18].

Two distinct categories exist for EM implementations: commercial and open-source. Their differences are summarized in Table 3.1. A more detailed comparison between the two categories can also be found in [103].

Table 3.1: *Commercial versus Open-Source EMs [16, Chapter 18].*

Criteria	Commercial	Open-Source
Cost	− Usually expensive	+ Less expensive (or free)
Customization	− Not always	+ Customizable code and (usually) better performance
Usability	+ Easy-to-use	– Not always easy-to-use
Standardization	+ Programming languages compliant with IEC 61131-3	– Nonstandard programming languages (e.g. C/C++)
Documentation	+ Detailed	– Sometimes poor
Hardware support	+ Variety of devices	– Limited
Features	+ Full	– Some EtherCAT features maybe not implemented
Customer Support	+ Advanced	– Sometimes incomplete support by the community
Reliability	+ QoS guarantees	– Instability issues

The biggest strength of open-source EMs is that they are usually free and easily customizable [16, Chapter 18]. Consequently, if a large number of complex and custom control applications needs to be developed, then the open-source category makes a good match [16, Chapter 18].

Although not free, the commercial EMs provide other features, like usability through Graphical User Interfaces (GUIs), reliability, full support, standardization and fully featured editions

[16, Chapter 18]. These features facilitate the learning process and provide short *time-to-market* solutions [16, Chapter 18].

Among the commercial implementations, Beckhoff TwinCAT[6] is one of the most popular EMs, developed by the company which introduced EtherCAT [16, Chapter 18]. Other popular commercial implementations include the NI EM and KPA EM[7], produced by National Instrument and Koenig-pa GmbH, respectively [16, Chapter 18].

In the open-source implementations, the most popular EM is EtherLab [16, Chapter 18]. It's developed by Ingenieurgemeinschaft IgH[8] and it's free [16, Chapter 18]. SOEM is another popular open-source solution and it's also free [16, Chapter 18]. However, SOEM is not used in applications requiring determinism and low cycle loop period T_{CYCLE}, since its implementation is not real-time (more on this below) [16, Chapter 18] [103].

3.2.1.3 Comparison of EtherCAT Masters in GNU/Linux

In GNU/Linux the two most popular open-source EMs are IgH Master (or EtherLab) and SOEM. These EMs are licensed under LGPLv2 and GPLv2 respectively. A concise comparison is presented in Table 3.2.

Table 3.2: *EtherLab versus SOEM.*

	EtherLab	**SOEM**
Learning Curve (Installation, Configuration and API)	Steep	Low
Documentation	Excellent (since v.1.5)	Poor
Mailing List Support	Yes	Yes
Portability	No	Yes
Integrability of EtherCAT (EM Type)	Full (Type A)	Some EtherCAT features not implemented (Type B)
user-space / kernel-space	kernel-space with user-space API	user-space
Licence	LGPLv2	GPLv2

SOEM is a small library running in user-space. It is lightweight and easy to familiarize with (setting it up and getting started). The documentation of the project is quite poor but the mailing list is supportive not only to beginners but also to experienced users and developers. Since it's a user-space library, it can be deployed in other OSes as well (Windows, MacOS,

[6] `https://www.beckhoff.com/twincat/`
[7] `https://koenig-pa.de/products/ethercat/kpa-ethercat-master`
[8] `https://www.etherlab.org/en/ethercat/index.php`

RTOSes). Since SOEM is a library, the user needs to create a custom application to provide means for:

- Reading and writing process data to be sent/received by SOEM.
- Detecting and managing errors reported by SOEM.

After creating the process data to be transmitted, SOEM communicates with the vanilla Linux Network Stack in order to pass the process data to the network driver, which in turn will send them across the EtherCAT network. After the system call, SOEM has no control over the latency introduced by the OS (network driver, scheduler etc). More information for the SOEM project can be found in it's GitHub repository[9], it's official homepage[10] and it's index page[11].

On the other hand, the IgH Master is a full-featured EM, highly configurable and flexible. It is more complex to set it up and get started. The documentation provided is excellent and the mailing list is also quite active.

A fundamental feature of the SOEM software is that it resides in user-space. With that in mind, it can not employ features a kernel module may have, like talking directly with the network driver, and be informed when each datagram is sent and delivered. SOEM can only communicate with the vanilla Linux Network Stack (not optimized for latency) and suffer the extra performance loss which OS's latency (context switching and copying of process data) introduces. This drawback can be considered a major one, as far as latency and determinism of cycle time are considered.

On the contrary, the architecture of IgH Master is quite perspicacious as it's implemented as a kernel module [2]. Kernel code can have real-time characteristics, i.e. lower latency than user-space code [2]. The main task of an EM is to service the control loop, which requires cyclic work to be done [2]. Cyclic work in the kernel is typically triggered by timer interrupts [2]. Thus, the execution time of a function processing timer interrupts is less in kernel-space than in user-space [2]. In kernel-space, context switches to other user-space processes isn't necessary [2]. Another reason favoring kernel-space, is that the master needs to directly communicate with the Ethernet hardware. This is suitable to be done in the kernel also (through network device drivers) [2].

[9] https://github.com/OpenEtherCATsociety/SOEM
[10] https://openethercatsociety.github.io
[11] https://openethercatsociety.github.io/doc/soem/index.html

3.2.2 The IgH EtherCAT Master for GNU/Linux (EtherLab)

Note: This section is largely based on [2].

Since the EtherLab software has been selected for development, it's necessary to present it's core features and functionalities for consistency.

3.2.2.1 Features

A summary of the basic features of EtherLab is illustrated below. More details for these features along with the installation instructions of EtherLab can be found in [2, 104]. Some EtherLab's features include [2]:

- It's a kernel module for Linux 2.6 / 3.x / 4.x.
- It's implemented according to IEC 61158-12 specifications [105, 106].
- Includes EtherCAT-capable native drivers for some Ethernet chips as well as a generic driver for all chips supported by the Linux kernel.
- It supports multiple EtherCAT masters running in parallel.
- It supports many Linux real-time extensions (Xenomai, RTAI, PREEMPT_RT) through its independent architecture.
- It provides an Application Programming Interface (API) for applications, that want to use EtherCAT functionality.
- It introduces *Domains*, which allow grouping of process data transfers with different slave groups and task periods.
- Supports Distributed Clocks.
- Supports CANopen over EtherCAT (COE), Ethernet over EtherCAT (EoE), Vendor-specific over EtherCAT (VoE), File Access over EtherCAT (FoE) and Servo Profile over EtherCAT (SoE) protocols.
- Includes a user-space command-line tool *ethercat*.

3.2.2.2 Architecture

The architecture of the EM is presented in Figure 3.21 [2]. The components of the master environment are briefly described [2]:

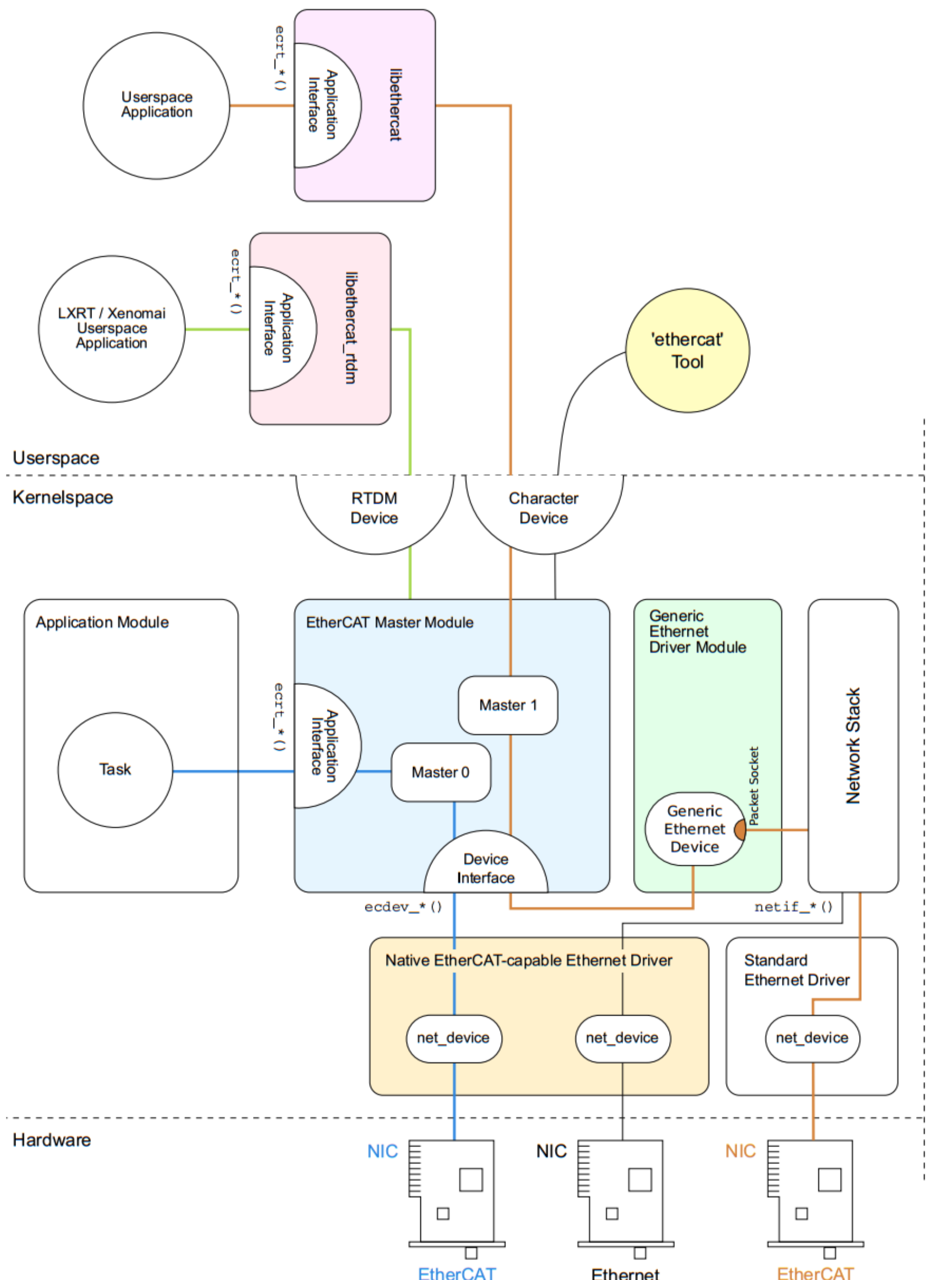


Figure 3.21: *EtherLab Master Architecture [2].*

- **Master Module**: This is a kernel module which contains at least one EtherCAT master instance, the EtherCAT Device Interface and the Application Interface.
- **Device Modules**: These are the intermediates between the Master Module and the Network Interface Controller (NIC). There are two categories; In the first belong the EtherCAT-capable Ethernet device driver modules, which offer their devices to the EtherCAT master via the EtherCAT Device Interface. These modified network drivers can handle network devices used for EtherCAT operation and standard Ethernet traffic at the same time. In the second category belong the Standard Ethernet device drivers

that aren't modified by EtherLab. They are connected to the kernel's network stack as usual. In this case, the Master Module communicates with the Linux Network Stack with its Generic Ethernet driver module. After accepting a device, the master is able to send and receive EtherCAT frames.

- **Application**: This is a program which is written by the user. It interfaces with the master in order to exchange process data with the EtherCAT slaves in the network. An application can request a master through the application interface. The application can reside in kernel-space and interface with the master via the kernel application interface, or reside in user-space and interface with the master via the EtherCAT or the RTDM library.

The EtherCAT master kernel module *ec_master* can contain multiple master instances [2]. Each master waits for certain Ethernet device(s) identified by its MAC address(es) [2]. For instance, the master module can be loaded with two masters, each one assigned to a specific MAC address as shown in Figure 3.22 [2].

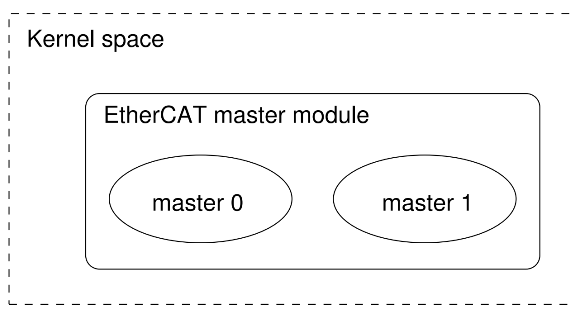


Figure 3.22: *Multiple masters in one module [2].*

Every EtherCAT master provided by the master module (*ec_master*) transitions between various phases, shown in Figure 3.23 [2]:

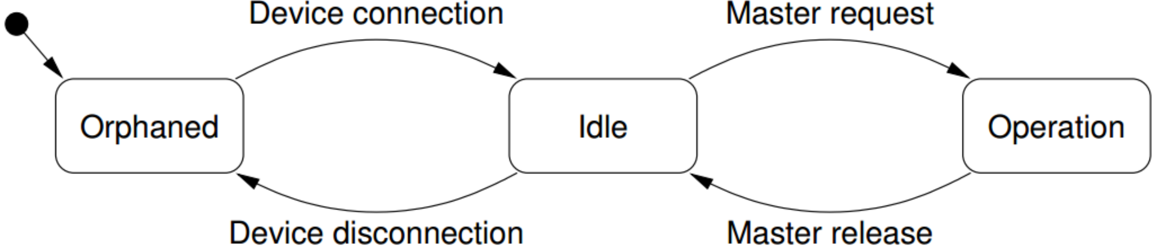


Figure 3.23: *Master phases and transitions [2].*

- **Orphaned phase**: In this phase the master waits for its Ethernet device(s) to connect. There is no bus configuration available.

- **Idle phase**: In this phase the master has accepted all available Ethernet devices, but is not requested by any application yet. Thus, the master merely runs its state machine. This involves automatically scanning the bus for slaves and executing pending operations from the user-space interface (for example SDO access). Again, there is no Process Data exchange since the bus communication isn't configured yet.

- **Operation phase**: In this phase the master is requested by an application which provides a bus configuration and exchanges Process Data.

For consistency, some useful common terms are presented below [2]:

- **Process Data Image**: The logical entities which are exchanged between master and slaves are called Process Data Objects (PDOs). They are encapsulated in EtherCAT datagrams, before the master sends them to the EtherCAT network. These entities are presented by the slaves and change their inputs and outputs. The available PDOs can be specified by reading the slaves' E^2PROM. The application registers the PDOs' entries for exchange during cyclic operation. The sum of all registered PDO entries defines the *Process Data Image*, which is exchanged via datagrams with *logical* memory access (e.g. LWR, LRD or LRW).

- **Process Data Domains**: The Process Data Image can be easily managed by introducing *domains*, which allow grouped PDOs exchange. Domains are necessary for Process Data exchange, so there has to be at least one. There is no upper limit for the number of domains, but each domain occupies one Field Memory Management Unit (FMMU) in each slave involved, so the maximum number of domains is limited by the slaves.

- **FMMU Configuration**: An application registers PDO entries for exchange. Every PDO entry occupies an area of the slave's physical memory, which is protected by a SyncManager for synchronized access. The SyncManager needs to have its last byte to be accessed, in order to react on a datagram accessing its memory. Thus, the whole synchronized memory area needs to be included into the Process Data Image. Figure 3.24 presents how FMMUs are configured to map physical memory to logical Process Data Images.

3.2.2.3 Application Interface

The Application Interface provides functions and data structures for applications to use an EtherCAT master. Every application uses the master in two steps [2]:

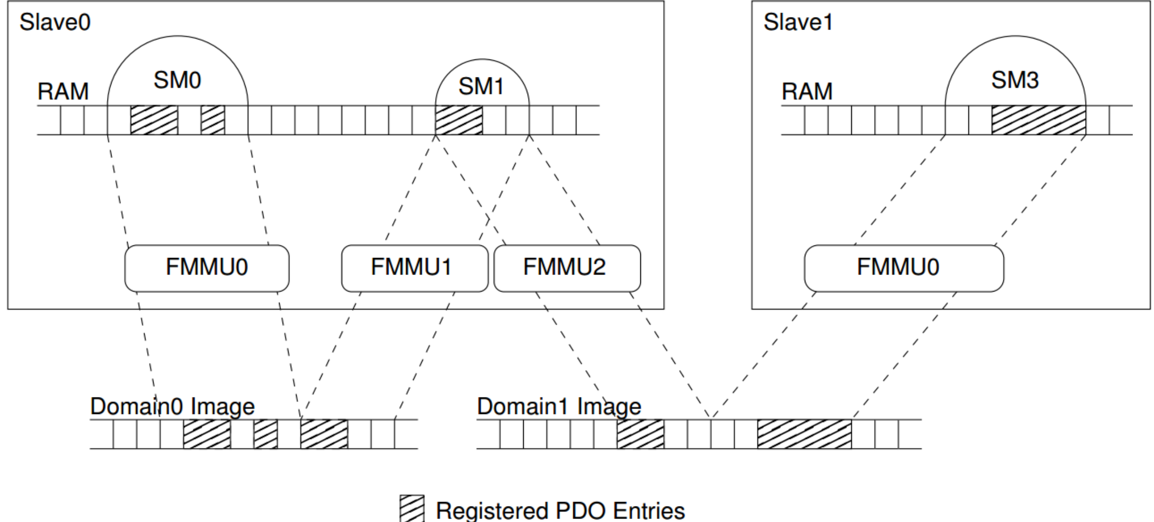


Figure 3.24: *Field Memory Management Unit (FMMU) Configuration [2].*

- **Configuration**: The application requests the master and configures it. For instance, the application can create domains, configure slaves and register PDO entries in each domain.

- **Operation**: The application runs its control loop and exchanges process data.

For the user's convenience, there are a few example applications in the *examples/* subdirectory of the master code which are documented in the source code [2].

The application configures the bus through the application interface [2]. Figure 3.25 depicts the objects, which can be configured by the application [2].

Slave Configuration: The application has to inform the master about the expected bus topology [2]. This is done by creating *slave configurations*, which provide information related to internal characteristics and position of the slaves in the network [2]. When the application creates a slave configuration, it provides the slave's bus position, vendor id and product code [2]. The master in turn, checks whether a slave with the given vendor id, product code and position exists [2]. If the information is correct, the slave configuration is linked to the real slave on the bus and the slave is configured with the information provided by the application [2].

Cyclic Operation: After the application has finished the configuration step, the master needs to be *activated* [2]. During activation, the master calculates the Process Data Image (PDI) and applies the bus configuration for the first time [2]. After the master is activated, the bus configuration cannot be changed and the application proceeds to the Operation step [2].

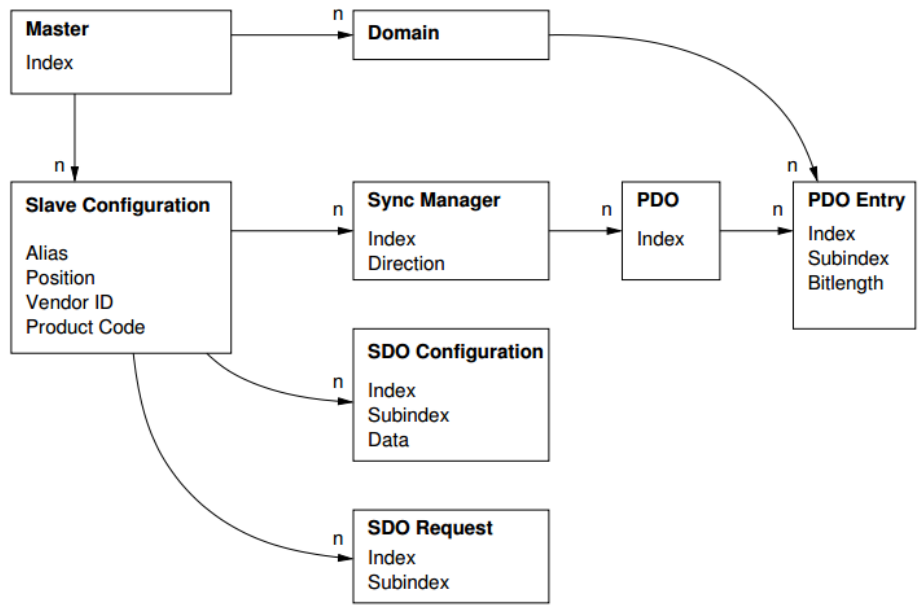


Figure 3.25: *Master Configuration [2].*

More details regarding the Application Interface of the master, can be found in [2].

3.2.2.4 Ethernet Devices

Note: The term *device* means an Ethernet network interface hardware.

EtherLab supports two types of devices [2]:

- **Native Ethernet Device Drivers**: These are native EtherCAT-capable device driver modules, which handle Ethernet hardware. They are used by the master to connect to the EtherCAT network and need to be able to accept Ethernet devices either for EtherCAT operations (real-time) or for standard Ethernet traffic using the Linux Network Stack. More information regarding the basic structure of a standard Ethernet device driver can be found in [107]. Its advantages include [2]:
  - There is only one networking driver used for EtherCAT and non-EtherCAT devices.
  - The modifications are based on existing Ethernet device driver, which is functioning properly and without any issues.
  - The master can achieve *bare-metal* performance, since for EtherCAT operation, the traffic doesn't traverse the non deterministic Linux Network Stack.

  However, it has the following disadvantages [2]:

  - The modified driver becomes more complicated, as it must handle EtherCAT and non-EtherCAT devices.
  - There are additional case differentiations in the driver code.
  - Changes and bug fixes on the standard drivers have to be ported to the EtherCAT-capable versions regularly.
  - A modified EtherCAT-capable version of the original Ethernet driver needs to exist for the Linux system's components, thus this type has limited availability.

- **Generic Ethernet Device Driver**: This type uses the Linux Network Stack to connect to the Network Interface Controller (NIC). It's available since master version 1.5. Its advantages include [2]:
  - It's not limited to specific drivers, versions and vendors, thus all Linux Ethernet drivers are supported.
  - The Linux Ethernet drivers are used without any modification.

  However, it has the following disadvantages [2]:
  - It doesn't support real-time extensions like RTAI, because the Linux Network Stack code uses dynamic memory allocations, which could cause the system to freeze in realtime context [2].
  - The performance is worse than the native type, since the EtherCAT frames need to traverse the whole Linux Network Stack.

Since there are ways of providing the Linux kernel wih real-time capabilities (e.g. with PREEMPT_RT, more on this in the following chapters) [108], the master in such setting could operate without native drivers, by using the Linux Network Stack instead. Figure 3.21 shows the *Generic Ethernet Driver Module*, that connects to standard Ethernet device drivers via the Linux Network Stack. This kernel module is named *ec_generic* and can be loaded the same way as a native EtherCAT-capable Ethernet driver. After it's loaded, the module makes available all the Linux-compatible devices to the EtherCAT master. As soon as the master accepts a device, the module creates a packet socket[12] with its `socket_type` set to `SOCK_RAW`, bound to this device. Subsequent calls to the device interface will interface with this socket.

[12] `https://linux.die.net/man/3/socket`

3.2.2.5 User-space Interfaces

Since the master is a kernel module, it is useful to have user-space interfaces, which facilitate master accessibility from user-space, finer monitoring, bus visualization and online parameter modification. These interfaces are implemented via a character device[13] and a user-space library.

Command-line Tool: The user-space tool, developed in the context of EtherLab, provides a quick visualization of the whole status of the EtherCAT network with simple commands. The user of this program can:

- Display the Bus Configuration.
- Print the current slaves on the bus along with some useful information for each one.
- Set a Master's Debug Level.
- List Sync Managers, PDOs assignment and mapping of slaves.
- Show configured Domains.
- Show master and Ethernet devices' information.
- Request from the slaves to reach new application layer states.
- Output a slave's register contents.
- Write Service Data Object (SDO) entries to a slave.

User-space Library: The native application interface resides in kernel-space and hence is only accessible from there. To make the application interface available from user-space programs, the user-space library *libethercat* has been created, which is linked to programs under the terms and conditions of the LGPL, version 2[14]. The kernel-space API is mapped to user-space, through an `ioctl()` interface. The kernel code interfaces directly the kernel API. Since the user-space API calls an `ioctl()` interface before reaching the kernel API, a small delay is introduced. For performance reasons, the actual domain process data are not copied every time between kernel and user memory, but are memory-mapped to the user-space application. As soon as the master is configured and activated, it maps the whole process data memory (all the

[13] `https://en.wikipedia.org/w/index.php?title=Device_file&oldid=894614419`
[14] `https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`

domains) to user-space. As a result, the user-space application accesses directly the process data without additional delay.

Timing Aspects: An interesting aspect is the time differences between the user-space library calls and the kernel API calls. Table 3.3 shows the call times and standard deviancies of typical (and time-critical) API functions measured on an Intel Pentium 4 M CPU with 2.2 GHz and a standard 2.6.26 kernel.

	Kernel-space		**User-space**	
Function	$\mu(t)$	$\sigma(t)$	$\mu(t)$	$\sigma(t)$
`ecrt_master_receive()`	$1.1\mu s$	$0.3\mu s$	$2.2\mu s$	$0.5\mu s$
`ecrt_domain_process()`	$< 0.1\mu s$	$< 0.1\mu s$	$1\mu s$	$0.2\mu s$
`ecrt_domain_queue()`	$< 0.1\mu s$	$< 0.1\mu s$	$1\mu s$	$0.1\mu s$
`ecrt_master_send()`	$1.8\mu s$	$0.2\mu s$	$2.5\mu s$	$0.5\mu s$

Table 3.3: *Application Interface Timing Comparison [2].*

The test results show that, for this configuration, the user-space API introduces about 1 μs additional delay for each function, compared to the kernel API.

Kernel/User API Differences: The only difference between the two APIs is the inability of the user-space API to provide external memory for domains. The reason is that the process data memory is managed internally by the library functions, since it's mapped to user-space.

4 Requirements Analysis & Technical Specifications

If you can't explain it simply, you
don't understand it well enough.

Popular quote

In this chapter a thorough analysis of the fundamental requirements of this work is provided. In addition, all the technical specifications of the system, which have been translated to the implementation of the solution are illustrated. First, the performance of the final system is specified for this project. Finally, the design choices adopted, the modeling, along with the overall system architecture and the API are presented.

4.1 Requirements Analysis

Every software project should have specific requirements to comply with. In the following section the project's requirements analysis is presented.

4.1.1 Laelaps II

Laelaps II[1](shown in Figure 4.1)introduces some improvements over its previous version, Laelaps I[2], in both mechanical and electrical properties. In the motion control scope, the main features of the robot have changed, including the leg design, the actuator-related char-

[1]http://nereus.mech.ntua.gr/legged/?page_id=161
[2]http://excellence.minedu.gov.gr/thales/en/thalesprojects/379424

acteristics and the power supply systems [17].

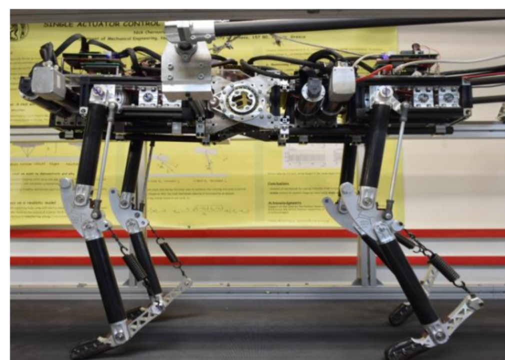


Figure 4.1: *Laelaps II.*

Laelaps II's enhancements, compared to Laelaps I, include [17]:

- New legs are designed and manufactured with lightweight carbon fiber tubes and custom aluminum parts.
- The PCIe/104 tower, which was used for centralized control is replaced with four identical EtherCAT Slave towers. Each of them controls the motion of one leg, with parameters provided by the EtherCAT master (decentralized control scheme).
- Driver extension boards are upgraded.
- The front parts of the body are reallocated, enabling the four legs to be symmetrically distributed.

Most of the electrical upgrades are thoroughly described in [109, Chapter 4] and the interested reader is referred to it for more details.

4.1.2 User Categories

The system to be developed will support the following user categories:

- **Software Developer**: The Software Developer creates software in the ROS environment, which inter-operates with the system to be developed.
- **Operator**: The Operator commands and monitors the overall system and tunes the necessary parameters to achieve the desired performance and metrics.

4.1.3 Functional Requirements

The EtherCAT network of Laelaps II, was thoroughly evaluated and tested with much success in Windows [17]. However the robotics (localization, navigation, state estimation, perception) algorithms, developed in the ROS environment for reasons described in Section 2.5, require GNU/Linux (preferably the Ubuntu distribution) as the host OS. The system to be developed should:

- Run in the GNU/Linux OS (preferably the Ubuntu distribution).
- Employ EtherCAT as the communication protocol and technology between master and slaves.
- Reside in the ROS environment.
- Have firm real-time characteristics. That means that the communication between master and slaves should be deterministic. This requirement, stands in the middle of functional and non-functional requirements, however due to the importance of determinism on the project, it has been categorized as functional.

4.1.4 Non-functional Requirements

The non-functional requirements complete the requirements of this work. These requirements explain how the system should behave. In this context, the system to be developed should:

- Have APIs for **interoperability** with the other robotics algorithms residing in ROS.
- Be **extensible**, meaning if the EtherCAT application data changes, an experienced developer should be able to make the required changes in < 20 person-hours.
- Have EtherCAT control loop time $T_{CYCLE} < 500\mu s$ (defined in Subsection 3.2.1), which translates to a **control loop frequency** > **2 kHz**.
- Have **adaptability**, meaning that the Mean Time to Change (MTTC) operability with different EtherCAT master under GNU/Linux will require < 1 person-month for a senior system developer[3].

[3] http://users.csc.calpoly.edu/~jdalbey/SWE/QA/QualityAttributesStearns.html

- Offer **installability**; A non-experienced user can install and operate the program without assistance of any kind.
- Offer **maintainability**; A software developer with 1 year of experience will be able to correct any known defect in < 2 person-days[4].
- Provide **understandability**; A novice user can learn to operate major use cases without outside assistance.
- Provide thorough **documentation** of every aspect of it.
- Provide **robustness**; erroneous data inputs should be answered with error messages.
- Provide mechanisms for **safety**; For critical situations there will be a software "panic button". This should stop instantly the motion control of the robot.
- Be **open-source**; All the source code will be available at the CSL-EP Bitbucket repository.

4.2 Technical Specifications

In this section the design along with the system's architecture are illustrated.

4.2.1 Design Choices

The requirements specified in the previous section need to be translated in the technical specifications of the project. The design choices made, played a catalytic role in this procedure, and are described below.

To realize the employment of an EtherCAT master in GNU/Linux, two choices exist as analyzed and compared in Subsection 3.2.1. Only the EtherLab EM meets the specified requirements and therefore this EM is chosen. More details on the features and architecture of EtherLab can be found in Subsection 3.2.2 and in [2].

In order to meet the functional requirement for firm real-time characteristics, the developed application should run in an OS that is modified to be hard real-time capable, as previously stated and explained in Subsection 3.2.1. For this requirement there are a handful of options to consider like RTAI, Xenomai and PREEMPT_RT. RTAI has been a rather popular library in the embedded linux world, however in the last decade its popularity has declined due to the

[4] http://users.csc.calpoly.edu/~jdalbey/SWE/QA/QualityAttributesStearns.html

fact that if one would like to write code i.e. a real-time thread, it should reside in kernel-space. This has the benefit of very low latency, yet lacks in maintainability, installability and configurability, not to mention the profound risk of freezing the kernel due to some buggy code. Therefore this library cannot be chosen. After experimenting with Xenomai and PREEMPT_-RT for a small while, the latter is chosen as the most non-invasive and suitable solution for our case. Recall that maintainability and installability are key-factors and non-functional requirements for the project and these requirements were taken into account (augmented with the reasons illustrated in Section 2.3) and led to the decision for PREEMPT_RT. More details on the features and architecture of PREEMPT_RT can be found in Subsection 2.3.1.

The requirement for interoperability should be satisfied from this project and this comes down to deciding what kind of API to use. This API should be created in the terms of ROS (analogous to a REST API in a web application). In the ROS context, this translates to choosing message communication mechanisms for interaction and data exchange between nodes. For further details and definitions on the ROS context, the interested reader is referred to Subsection 2.5.2 and for details and differences on the various communication mechanisms in ROS, the reader is referred to Subsection 2.5.3. Furthermore for interoperability reasons, the language of choice for the project is C++. This decision is based partly on the knowledge that with C++, the EtherLab user-space C API can be integrated seamlessly in the project, but also that C++ is one of the mainstream languages supported by ROS and has a good balance between low-level tweaking and programming, and expressiveness and abstraction, distinctive features of the high-level languages.

Interoperability is required among ROS nodes which implement higher-level locomotion, control and localization algorithms. These nodes must send commands to the motors and receive feedback from the encoders quite frequently, therefore the mechanism selected for implementing the ROS API of the project is the **topics** mechanism. This conclusion was not reached at the beginning of the project, but after experimentation with the **services** mechanism. Topics provide a more throughput-friendly message communication among nodes. Since the nodes communicating with the project's node will require frequent exchange of data messages, this is the most appropriate design choice.

Another important design decision should be made on the synchronization primitive for the application's threads. In the program there should be two fundamental threads. The first will run in real-time context and execute the cyclic loop in a deterministic manner and the second accepts the input commands from the ROS environment and writes synchronously to

a common buffer, shared with the real-time thread. This synchronization should be handled with extreme caution, since the real-time thread shouldn't go for sleep, rather busy-wait on a locking mechanism. This mechanism is decided to be a **POSIX spin lock**, which offers the busy-waiting part and is easy to use, since implementation of the POSIX threads (spin locks API is part of pthreads API) API are instantly available on Linux.

Last but not least, the most appropriate library for threads should be picked out for our case. There are three favorable candidates: The C++ Thread core library, the Boost Thread library and the POSIX Thread library. After a brief research on these libraries, it is concluded that the most useful, rich and appropriate for this project is the Posix Thread library. A key part of the conclusion is the familiarity of the author with the pthread library. Also, major role to this conclusion have characteristics of this library like richness and simplicity of its API and assignment of system-wise attributes, compared with the other libraries. However, a known trade-off arises, the overhead of wrapping objects and functions of the pthread C library in C++ classes and methods.

4.2.2 System Architecture

In this section the overall System Architecture is presented and it's key components are analyzed. In Figure 4.2 a basic deployment diagram enhanced with the fundamental components and their connections is illustrated. This diagram describes in an intuitive manner the overall system's components, and provides a systemic view of the robot with it's operator. Its components are described in detail below.

4.2.2.1 The Operator Interface

The Operator, in (1), communicates with the software component, starts/stops the EtherLab operation phase and changes values of EtherCAT variables related to the motion of the quadruped robot's legs. For deeper understanding of the user interface created, facilitating the operation of the robot, a use case diagram is illustrated in Figure 4.3.

Apart from starting/stopping and restarting the communication with the EtherCAT slaves, the most important command offered through the Command Line Interface (CLI) is the ability to change the values of the EtherCAT variables, independently of the state of the communication with the slaves (online or offline). Lastly, the Operator has the ability to change EtherCAT variables for one slave or for all of them.

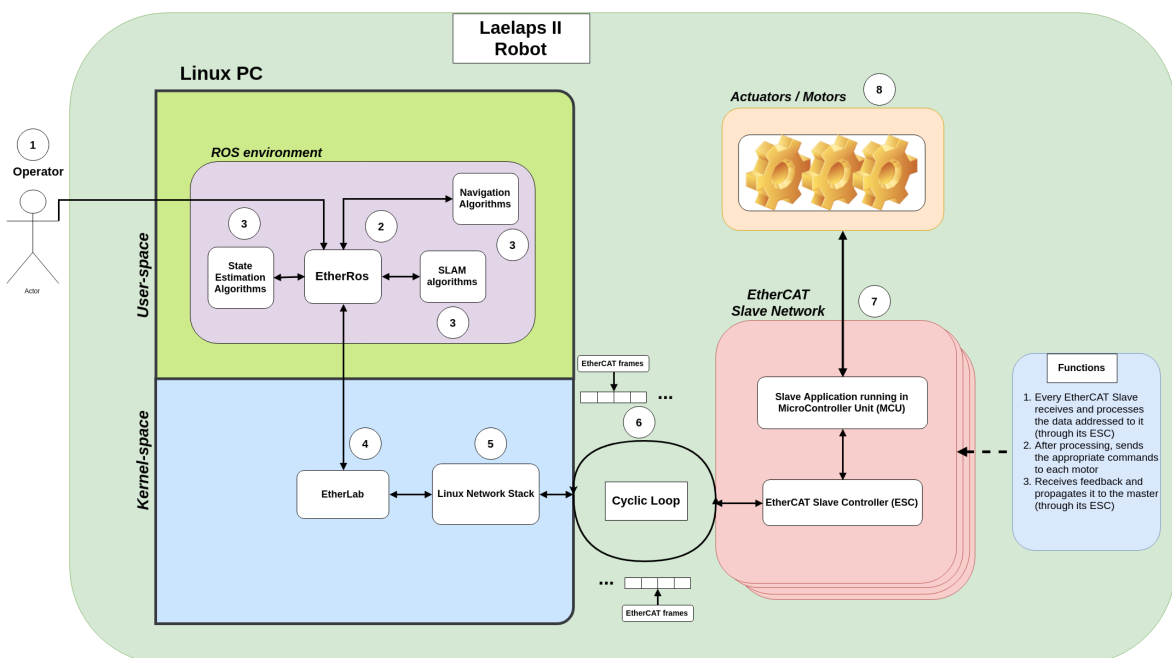


Figure 4.2: *Overall System Architecture.*

4.2.2.2 Project's software component

The software component of this project is presented in (2) along with it's higher-level connections. The software offers a ROS API (more on that in Subsection 4.2.3) to ROS nodes developed by other software engineers, offers a CLI to the Operator and interfaces with the EtherLab kernel module to achieve EtherCAT communication. This overall behaviour is accomplished through different submodules, illustrated in Figure 4.4.

In (A) the EtherCAT Communicator submodule is pictured. This submodule consists of a thread running in real-time context and calls the EtherLab User-space Library API, which in turn makes a system call to the EtherLab Kernel Module that communicates with the EtherCAT slaves. This module uses the pthread library for creating a real-time thread and for utilizing the pthread spinlock. It realizes the state machine described in Subsubsection 3.2.1.1, implementing it in real-time context at the EtherCAT control loop frequency ($\geq$ 2kHz).

In (B) the submodule of the Input Process Data Objects (PDOs) Publisher is highlighted. This part of the presented software project receives the input Process Data Objects (PDOs) (the EtherCAT variables which the slaves change and pass to the master) from the EtherCAT network via the EtherCAT Communicator, and publishes them in a **topic** in ROS, at the EtherCAT control loop frequency ($\geq$ 2kHz). Consequently, the ROS nodes implementing robotics algorithms such as SLAM, navigation and state estimation, can receive these data

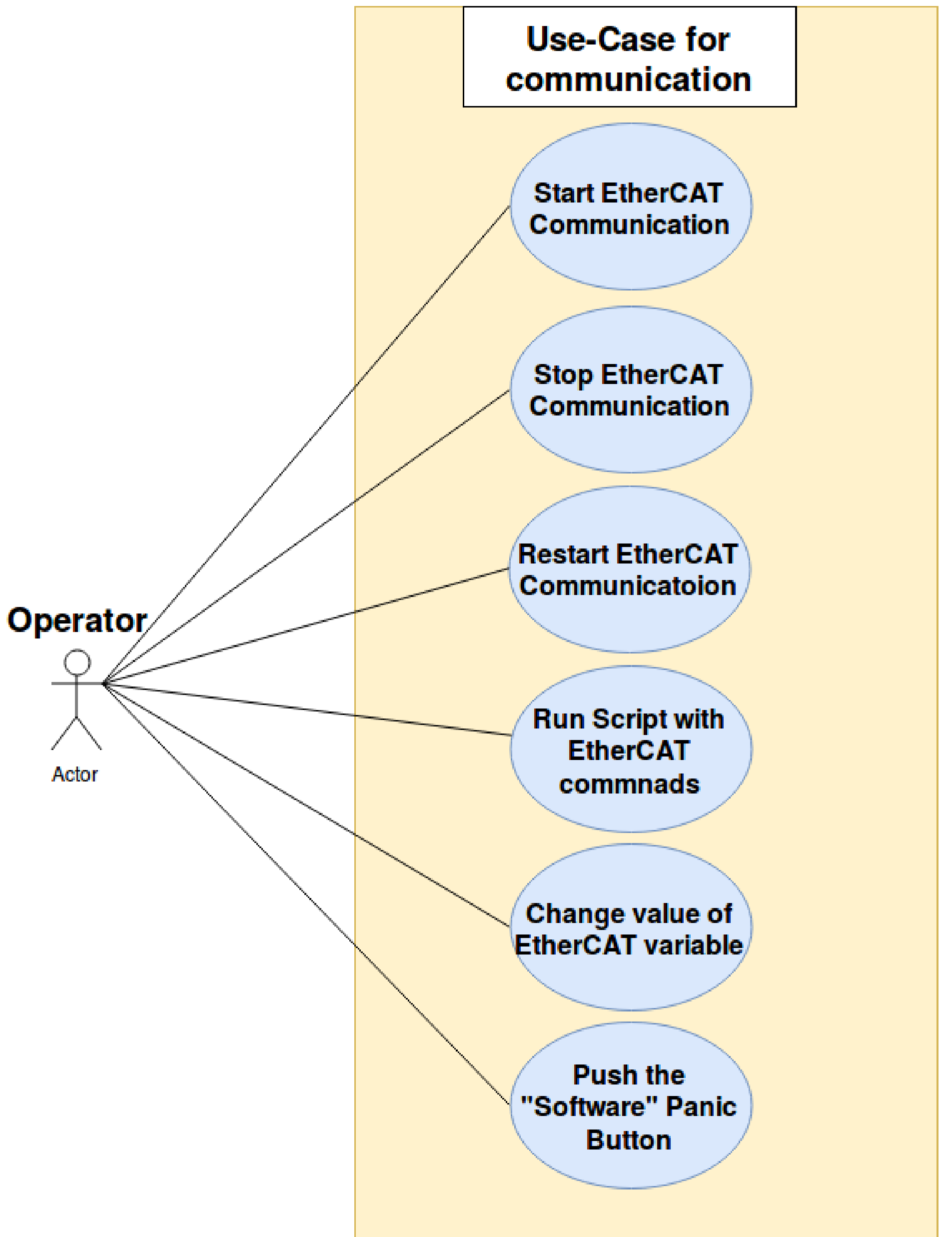


Figure 4.3: *A Use-Case Diagram for the Operator.*

and process them accordingly.

In (C) the Output Process Data Objects (PDOs) Publisher is highlighted. This submodule receives the output Process Data Objects[5] from the EtherCAT network via the EtherCAT Communicator, and publishes them in a **topic** in ROS, at the EtherCAT control loop frequency ($\geq$ 2kHz). Consequently, the ROS nodes implementing robotics algorithms such as SLAM, navigation and state estimation, can receive these data and process them accordingly. A question here arises, as to why should these Output PDOs be published to the entire ROS

[5]The EtherCAT variables which nodes in ROS or the Operator change and pass them to master in order to be sent to the EtherCAT network.

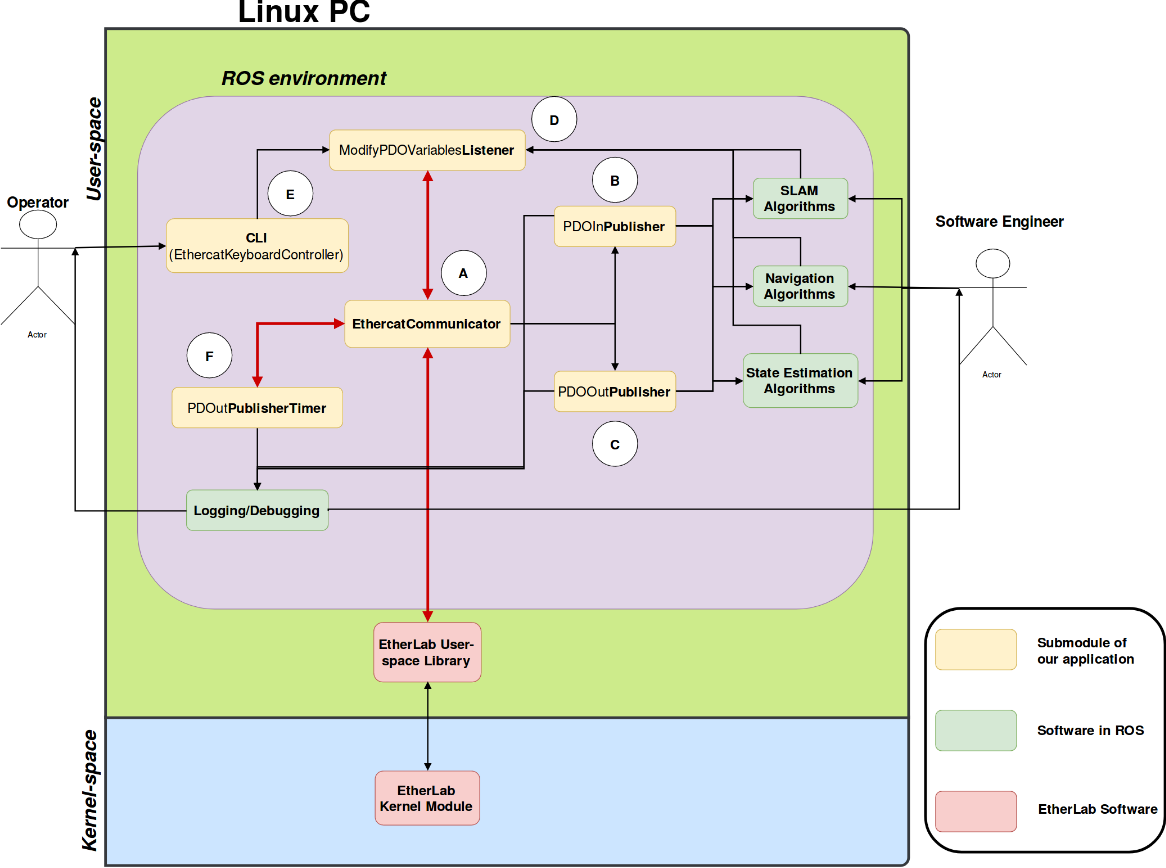


Figure 4.4: *Internal architecture of the software project.*

environment, since probably they are changed by a node in the ROS framework. The answer is that these data could interest more than one ROS nodes (apart from the one changing them), so other nodes can access these changes. Another possibility is that the data could be changed by the Operator, as previously mentioned, so a ROS node should be able to know the changes by subscribing to the aforementioned topic. Nevertheless, this submodule was created to provide completeness in its ROS API. If the overhead introduced is extravagant, this submodule could be disabled in future versions.

In (D) the submodule of the Output Process Data Objects (PDOs) Listener is presented. This submodule listens to a ROS **topic**, receives the (modified) output Process Data Objects[6] directly from the ROS ecosystem or indirectly from the created CLI, and passes them to the EtherCAT Communicator in order to be sent to the network. The (D) submodule concludes a first closed feedback loop, consisting of the ROS ecosystem (other ROS nodes implemented), the EtherCAT network and the application, allowing communication among all these components.

[6]The EtherCAT variables which are changed from the master side and passed to the EtherCAT slaves.

The Command Line Interface (CLI), in (E), facilitates the user who is responsible for the overall operation and administration of the robot functionalities (the Operator), to interact with a simple and effective manner with the EtherCAT slaves network and control the synchronized moves of the legs effectively. This submodule offers to the Operator the functionalities described previously in (1). Furthermore, the Operator can activate / deactivate the EtherCAT Communicator through the CLI, and the EtherCAT variables that the master sends (the Output PDOs) can be altered via this submodule from the Operator.

In (F), the submodule of the Output Process Data Objects (PDOs) Publisher Timer is presented. This part of the project copies at certain intervals in time (for this reason it's called a Timer) the process data to be sent from the corresponding buffer and publishes them to a ROS **topic**. With this published information, indirect logging takes place which constitutes a quick start for debugging the behavior of the software component. Therefore submodule (F) concludes a second closed feedback loop, consisting of the users administering the operation of the robot, the ROS ecosystem, the software module and the EtherCAT network. However, this second closed loop is definitely more indirect from the first one, in the sense that there is the human factor in the middle, therefore there must be administration and monitoring from a user to take action in order to close this loop.

The submodules in (A), (D) and (F) are working simultaneously on the same critical data. For this reason, a synchronization scheme is introduced as previously analyzed, with the bold red lines illustrating this need for synchronization. The chosen synchronization mechanism is a pthread spinlock, for reasons previously stated. The scheme is depicted in Figure 4.5.

The main shared resources in the following description are the spinlock `lock`, the buffer `process_data_buf` and the buffer `domain1_pd`. In a nutshell, submodule (A) realizes the fundamental state machine described in Subsubsection 3.2.1.1. This includes setting the real-time attributes of the thread before starting the cyclic loop, waiting for a fixed time interval and receiving the PDOs of the frame from the EtherCAT network, via calling a user-space API of EtherLab (more on that in Chapter 5), in the `domain1_pd` buffer.

It should be noted, that the `domain1_pd` buffer is shared between the application and the EtherLab kernel module, as aforementioned in Subsubsection 3.2.2.5. This means that simultaneous access to this buffer shouldn't be allowed, since it is not safe. Although submodule (A) wouldn't ever access the shared buffer at the same time with the EtherLab module (the submodule always uses the user-space API -calls EtherLab- **after** accessing the shared buffer),

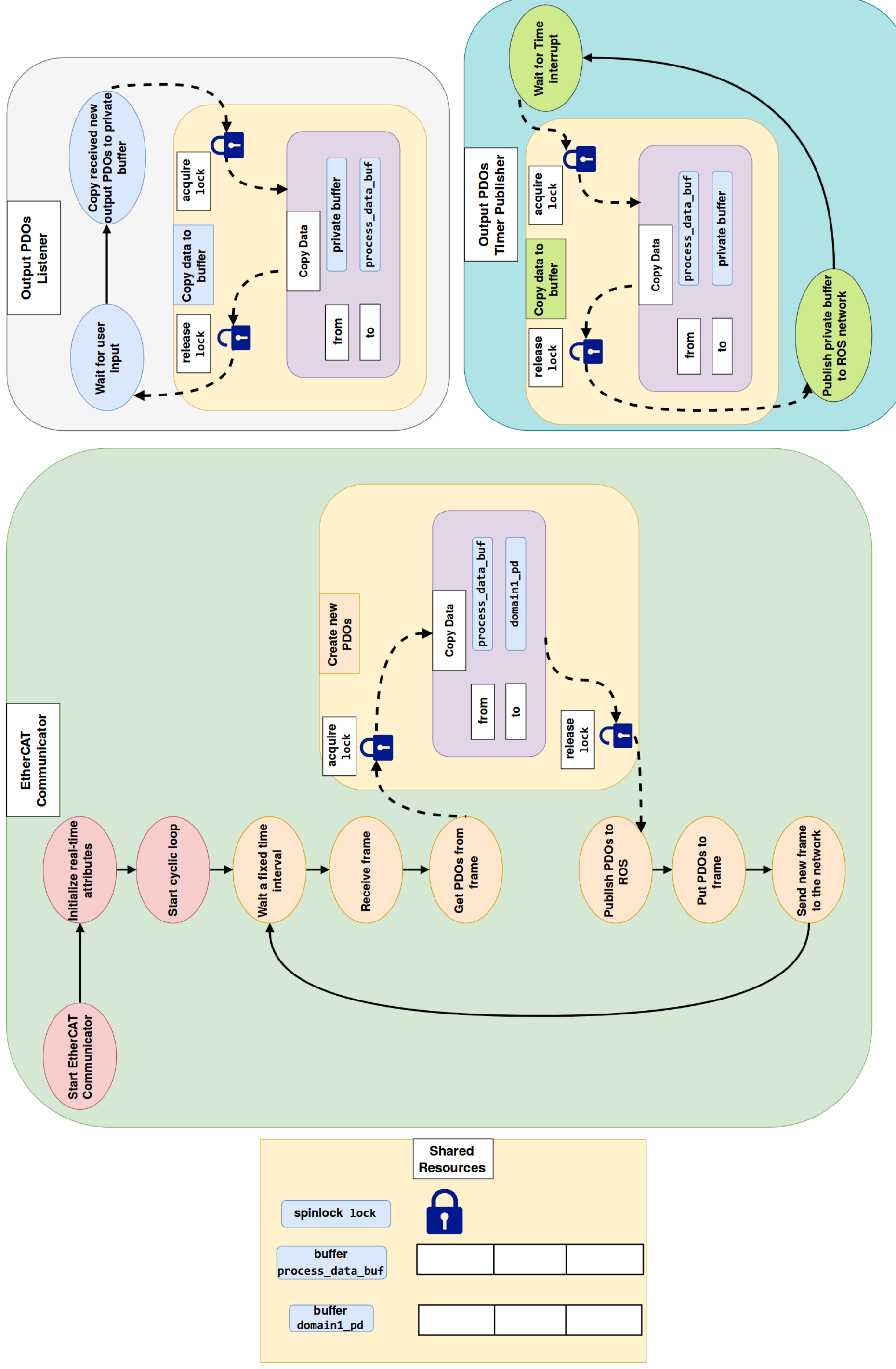


Figure 4.5: *Synchronization scheme followed in the software project.*

it is expected from the application to be instantly responsive to new inputs, i.e. to manage new incoming output PDOs from anywhere anytime. This expectation could not be met with just one buffer, since the application has limited time slice in the overall control loop time interval to access the buffer, while the incoming traffic could arrive in any moment in the control loop. Consequently, a neat solution to this problem was devised, by introducing a second buffer,

namely the `process_data_buf`. This buffer is shared by the application's submodules and isn't accessed by EtherLab. Therefore a synchronization mechanism is provided, namely spin-lock `lock`, in order for the submodules to access safely the same buffer `process_data_buf`.

That said, submodule (A) copies safely the new output PDOs (only these PDOs are changed from the master's side) from the `process_data_buf` to the `domain1_pd` buffer. With this action, it has created the new PDOs ready to be sent. Then, it sends all the PDOs to (B) and (C), in order to be published in the corresponding topics. Next, it calls the user-space API of EtherLab to send the `domain1_pd` buffer with the new PDOs to the EtherCAT network. Finally it starts over the entire process.

Submodule (D) is responsible for taking the user input (output PDOs of EtherCAT application) into a private buffer and safely copy it into the `process_data_buf`. Finally submodule (F) copies safely the buffer `process_data_buf` to a private one and subsequently publishes it's contents to a corresponding ROS topic, at fixed time intervals (order of seconds).

4.2.2.3 Other ROS nodes

The other ROS modules inter-operating with the developed application (using the ROS API), are depicted in (3), in Figure 4.2. The kind of algorithms depending on the developed software application, are the ones that communicate with the legs and synchronize them. In this context, when the robot operates, these components should be necessary for an autonomous operation:

- High-Level Control & Motion Planning Algorithms: These algorithms are the bare-bones of motion, since they allow the robot to move correctly based on the developed control algorithms. This family of algorithms sends motion commands to the four legs, based on an analytical model of the robot. For example, if we wish the robot to move to a certain location, this kind of algorithms should compute the corresponding velocities and accelerations and pass them to each leg controller (see also (6)). In order to pass these parameters, they should use the project's ROS API to communicate with the EtherCAT slaves.

- State Estimation Algorithms: These algorithms are critical in the functionality of a robot, as they estimate the pose[7] of the robot at fixed time intervals. They estimate the robot's pose in an unknown world, therefore should these components not work,

[7] Here by pose we mean the vector consisting of the position $\{x, y, z\}$ and orientation $\{\epsilon, \eta\}$ (represented by a unit quaternion).

it would be devastating for the accomplishment of tasks and operations. This set of algorithms in order to function properly needs to use the project's ROS API, extract the information for the position and velocities of the motors and determine, through a motion and sensors model, the current pose of the robot with accuracy.

- SLAM Algorithms: The Simultaneous Localization and Mapping (SLAM) algorithms are an essential part of every robot software. With the aid of such algorithms, the robot can localize itself based on a map of the environment, which is simultaneously updated dynamically. In order for these algorithms to properly work, they need to have an odometry estimation, which can be inferred by the angles and angular velocities of the motors of the legs, through the project's ROS API, or by other external means.

4.2.2.4 EtherLab

The EtherLab software is shown in (4), in Figure 4.2. The project's software utilizes EtherLab to communicate with the EtherCAT Slave Network. More information on its internal architecture can be found in Subsection 3.2.2 and in [2]. This module has a user-space library, and it's API is used by the project's software, as already shown in Figure 4.4.

4.2.2.5 Linux Network Stack

The Linux Network Stack is illustrated in (5), in Figure 4.2. To acquire a complete picture of the overall system architecture, it is convenient to present the relationship of the Linux Network Stack with the EtherLab module and the EtherCAT network. Detailed information about the internal architecture of the Linux Network Stack can be found in [110, 111, 112, 113, 114, 115]. For simplicity and consistency, the analysis follows two common paths: the sending and the receiving one.

Sending Path: The path the code and the data follow in order for the application to transmit an EtherCAT frame, is illustrated in Figure 4.6.

Depending on the type of the device chosen to run EtherLab (Native or Generic), the sending path differs significantly. However the first steps are the same: The application calls `ecrt_master_send()` of the user-space API of EtherLab, which makes an `ioctl()` call with the `EC_IOCTL_SEND` option and eventually calls the kernel-space `ecrt_master_send()` function of the module. There some sanity checks are performed as well as internal bookkeeping and some statistics are kept. Then, the frame is ready to leave the master and head to slaves.

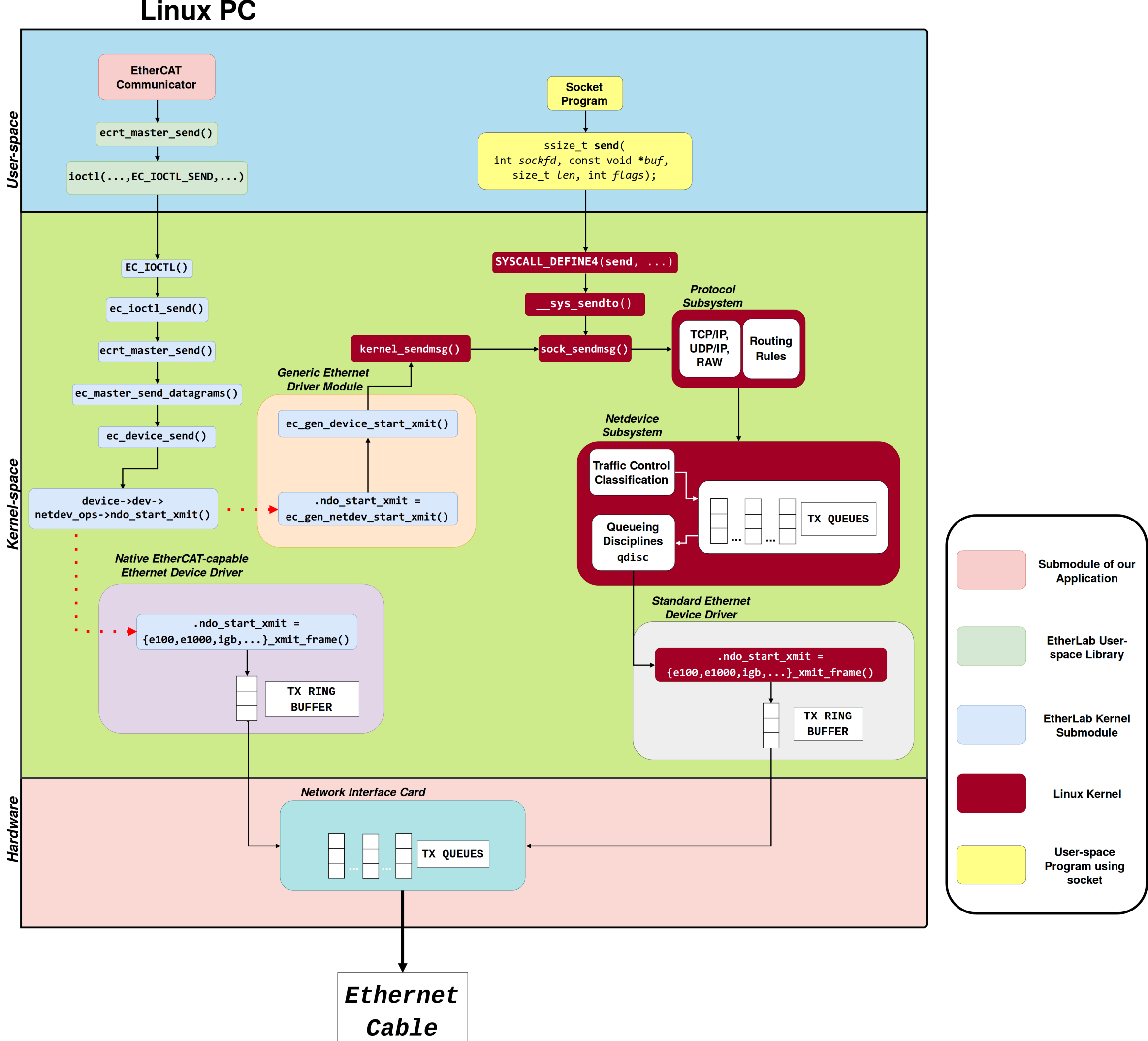


Figure 4.6: *Sending Path anatomy.*

This call is made from the `ndo_start_xmit()`[8] function pointer. Note that, depending on how the EtherLab module has been compiled and built in the system, there will be different functions registered to be called from the above function pointer.

From this point on, the two sending paths are separated (red dashed lines).

If the device chosen is a Native EtherCAT-capable one, then the call continues to it's registered function for `ndo_start_xmit()` which passes the frame to the hardware specific code and copies the data to a TX ring buffer in RAM (some NICs are "multiqueue": they can DMA many Buffers from/to RAM), signals the NIC for DMA and then NIC is ready for transmission.

However, if the device chosen is a generic one, which means that the driver used has no mod-

[8] `device->dev->netdev_ops->ndo_start_xmit()`

ifications from the EtherLab code, then the path followed is a typical one: Calling the socket kernel API through the wrapper `kernel_sendmsg()` with a `RAW` defined socket. After this call, the rest is left to the Linux kernel. In addition, for comparison reasons, a call from the `send()` system call is included, which is called from a typical network program that sends data to a network through a socket.

The short version of the rest of the path followed is described below. The `kernel_sendmsg()` calls `sock_sendmsg()` which after some internal wrappings, gets into the Protocol Subsystem. There, the data will pass all the IP-related internal layers and are encapsulated into a packet. Since the IP source and destination field is not defined in the EtherCAT data (unlike a typical IP packet), the data probably won't be routed. Then, the data continue into the net-device subsystem in which traffic control code will classify with the aid of the *Transmit Packet Steering* (XPS) algorithm, into which *TX queue* the data will be put (if there are many). Then the *Queueing Discipline* of the specific TX queue will be applied. This task will run in a *softirq context* (from the `ksoftirqd` thread of a specific CPU), while so far the path was created in a *process context*. Finally the packet will be checked if it needs segmentation, will be handed to the packet taps (like PCAP, the library Wireshark or tcpdump use to capture filters in POSIX-compliant systems) and the standard driver's `ops` are used to pass the data down to the NIC by calling the registered function for `ndo_start_xmit()`. After that, the typical procedure follows: The registered function for `ndo_start_xmit()` passes the data to the hardware specific code, the now EtherCAT frame is checked for fragmentation, is copied to a a TX ring buffer in RAM, the NIC is signaled for DMA and then NIC is ready for transmission. Keep in mind that in spite of coloring the Native Driver submodules blue (which contribute to the transmission), the Native Driver is a standard driver with some minor modifications from the EtherLab code.

Receiving Path: The path which the code and the data follow in order for the application to receive an EtherCAT frame, is illustrated in Figure 4.7.

Depending on the type of device chosen to run EtherLab (Native or Generic), the receiving path is altered significantly too. However the first steps are the same: The application calls `ecrt_master_receive()` of the user-space API of EtherLab, which makes an `ioctl()` call with the `EC_IOCTL_RECEIVE` option and eventually calls the kernel-space `ecrt_master_receive()` function of the module. From there and after some sanity checks, internal bookkeeping and some statistics, the `device->poll(device->dev)` function pointer calls the registered function for polling. Note that, depending on how the EtherLab module has been

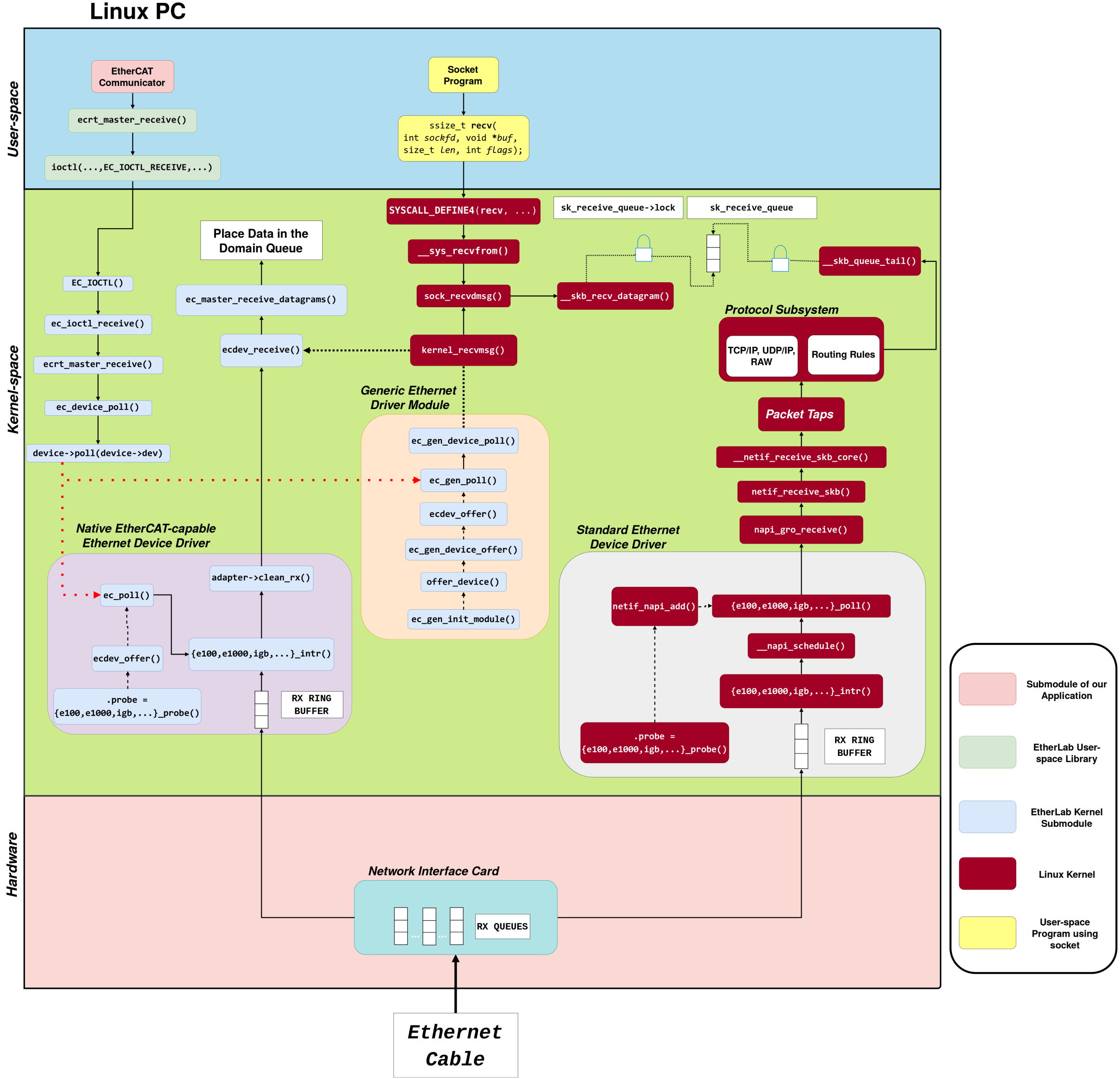


Figure 4.7: *Receiving Path anatomy.*

built and initialized in the system, there will be different functions registered to be called from the above function pointer.

From this point on, the two receiving paths are separated (red dashed lines), but they both merge with the call to `ecdev_receive()`, after each one completing it's own journey.

If the device chosen is a Native EtherCAT-capable one, then the call continues to the registered function for polling `ecdev_receive()` (EtherLab code inside the standard driver). This function has been registered when the EtherCAT native driver module was inserted ("probed") into the kernel (path shown with black dashed lines). Next, the function calls "manually" the, lightly modified by EtherLab, IRQ handler of the driver, which is short-circuited to a call to `adapter->clean_rx()` function handler which finally leads to the fi-

nal goal of `ecdev_receive()`. Note here that the IRQ handler is modified so that if it is awaken by the EtherLab code, it doesn't call any actual poll functions, as a standard Ethernet driver would do normally (see later). The `ecdev_receive()` receives the raw (EtherCAT) frames and passes them to `ec_master_receive_datagrams()`, which extracts the necessary EtherCAT datagrams and places them in the Domain Queue for further processing.

However, if the device chosen is a generic one, which means that the driver used has no modifications from the EtherLab code (the standard Ethernet driver is used), then the path followed is a typical one; The registered poll function that the `device->poll(device->dev)` points to, is the `ec_gen_poll()` (registered when the EtherLab module is initialized in the kernel with the generic device, path shown with black dashed lines), which leads to `ec_gen_device_poll()` which does two things: First calls the socket kernel API through the wrapper `kernel_recvmsg()` with a `RAW` defined socket. After this call and when `kernel_recvmsg()` returns, it merges with the path followed by Native, which means calling the `ecdev_receive()`, which receives the raw (EtherCAT) frames and passes them to `ec_master_receive_datagrams()`, which extracts the necessary EtherCAT datagrams and places them in the Domain Queue for further processing. Meanwhile, in the call of the wrapper `kernel_recvmsg()`, the path continues to the Linux kernel. In addition, for comparison reasons, a call from the `recv()` system call is included, which is called from a typical network program that receives data to a network through a socket.

The receive path in the Linux kernel code breaks into two parts. The first consists of the frames received from the Ethernet cable while the second part consists of the calls either from the system call `recv()` or from the EtherLab module with a generic device, starting from their common call of `sock_recvmsg()`. The meeting point is the queue `sk_receive_queue`, which both parts access safely through a shared `sk_receive_queue->lock`. The first part runs in *application* context, while the second runs in *interrupt* and later *softirq* context.

The first part is described below, through a brief summary. The NIC receives the Ethernet frame, DMAs the frame into an RX ring buffer in RAM and raises an interrupt in the kernel. The registered IRQ handler of the standard driver is executed, the IRQ is cleared on the NIC, so that it can generate IRQs for new packet arrivals, NAPI softirq poll loop is started with a call to `__napi_schedule`, which triggers the `ksoftirqd` thread to run the corresponding handler of the pending `softirq`, which eventually calls the NAPI[9] poll function

[9]New API (NAPI) was introduced in the kernel as a solution to the issue of driven down CPUs caused by the frequency of the interrupts from NICs.

registered from the driver (when "probed" in the kernel, path shown with black dashed lines) to do the further processing. Next, the driver's poll function harvests packets from the RX ring buffer in RAM and hands them over to `napi_gro_receive`, where they are checked for Generic Receive Offloading (GRO). After GRO the path continues to `netif_receive_skb()`, which after Receive Packet Steering (RPS) handling eventually leads to `netif_receive_skb_core()`, which after delivering data to any taps (like PCAP), passes the data on to the registered protocol layer handlers. Finally, the data after passing through the protocol stacks, netfilter, routing optimizations and berkeley packet filters, are eventually placed in the queue `sk_receive_queue` with the use of the `__skb_queue_tail()`, through the shared `sk_receive_queue->lock`.

The second part is far more brief than the first one; The call from `kernel_recvmsg()` leads after passing through some protocol specific layers (like `UDP`, `TCP` and ... `RAW` too) to `sock_recvmsg()`, which calls the function `__skb_recv_datagram()`. This function reads safely from the queue `sk_receive_queue`, through the shared lock `sk_receive_queue->lock`.

Finally, after `kernel_recvmsg()`, the function `ec_gen_device_poll()` calls `ecdev_receive()` and eventually the data end up in the Domain Queue for further processing.

4.2.2.6 Cyclic loop exchange

The cyclic loop created from exchange of EtherCAT frames is pictured in ⑥, in Figure 4.2. The EtherCAT protocol, network topology, frames and cyclic loop are thoroughly described in Section 3.1 and Section 3.2. For the quadruped robot, the EtherCAT slaves network topology forms a daisy chain, in the sense that the EtherCAT frame leaves the Ethernet NIC of the master, passes all the slaves, and then takes the same way back as it came, through the same cable (ring topology). There is no switch intervened in the current EtherCAT network topology. The Ethernet cable is connected to one port of the NIC of the master, while the slaves have two connected Ethernet ports each, except from the last one, which has also one port.

Prior to introducing the EtherCAT process data or variables shared among the master and the slaves, it is of utmost importance to underline the leg's model which is followed and the reasons for choosing these process data. Each leg of the quadruped robot consists of three links, as shown in Figure 4.8. However, since the attached spring is stiff, it can be safely assumed that it comprises of two links (the upper is actual while the lower is virtual) [17].

In Figure 4.9, the motion planning and control parameters of each leg are presented [17].

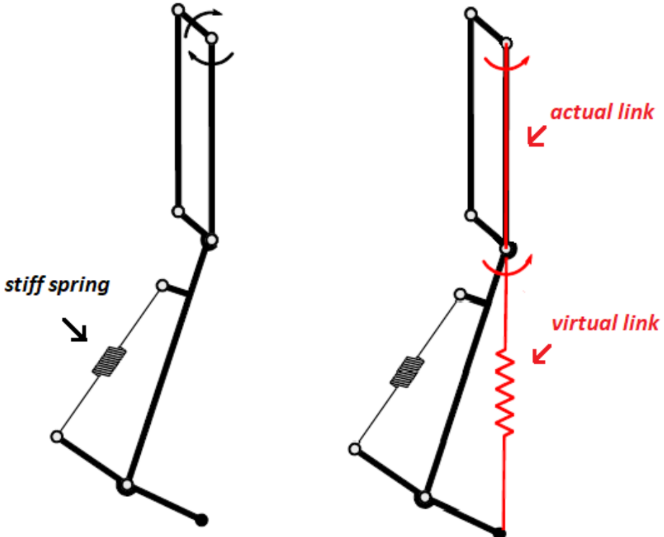


Figure 4.8: *Actual and virtual links of Laelaps II legs [17].*

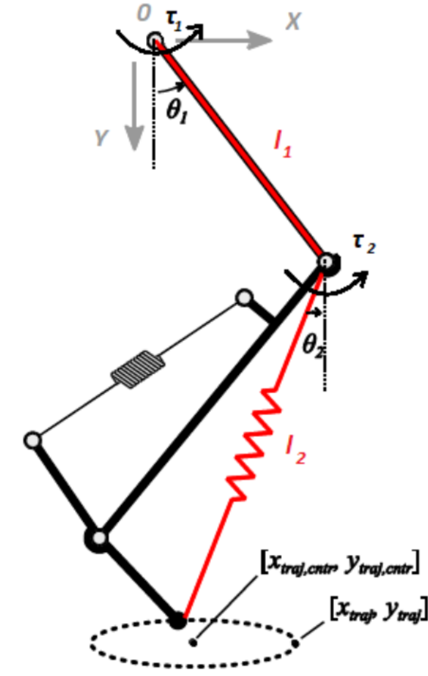


Figure 4.9: *The leg's model [17].*

The kinematics equations regarding the leg shown in Figure 4.9 are [17]:

Forward Kinematics:

$$\begin{aligned} x_E &= l_1 sin\theta_1 + l_2 sin\theta_2 \\ y_E &= l_1 cos\theta_1 + l_2 cos\theta_2 \end{aligned} \tag{4.1}$$

Inverse Kinematics:

Applying the law of cosines, the former (4.1) becomes [17]:

$$\begin{aligned}
&\phi = \theta_2 - \theta_1 \\
&x_E^2 + y_E^2 = l_1^2 + l_2^2 - 2l_1l_2cos(\pi - \phi) = l_1^2 + l_2^2 + 2l_1l_2cos\phi \\
&cos\phi = \frac{x_E^2 + y_E^2 - (l_1^2 + l_2^2)}{2l_1l_2} \\
&sin\phi = -\sqrt{1 - cos^2\phi} \\
&\phi = atan2(sin\phi, cos\phi)
\end{aligned} \tag{4.2}$$

Thus, solving (4.2) for θ_1 and θ_2 [17]:

$$\begin{aligned}
&\theta_2 = \frac{\pi}{2} - atan2(y_E, x_E) + atan2(l_1sin\phi, l_2 + l_1cos\phi) \\
&\theta_1 = \theta_2 - atan2(sin\phi, cos\phi)
\end{aligned} \tag{4.3}$$

Regarding the **leg's workspace**, the maximum/minimum lengths of the leg (knee joint at end-stop) are [17]:

$$\begin{aligned}
&l_{eff,max} = l_1 + l_2 = 250 + 350 = 600mm \\
&l_{eff,min} = \sqrt{l_1^2 + l_2^2} = \sqrt{250^2 + 350^2} = 430,12mm, \theta_2 = 90^\circ
\end{aligned} \tag{4.4}$$

A visualization of the leg's workspace is shown in Figure 4.10 [17].

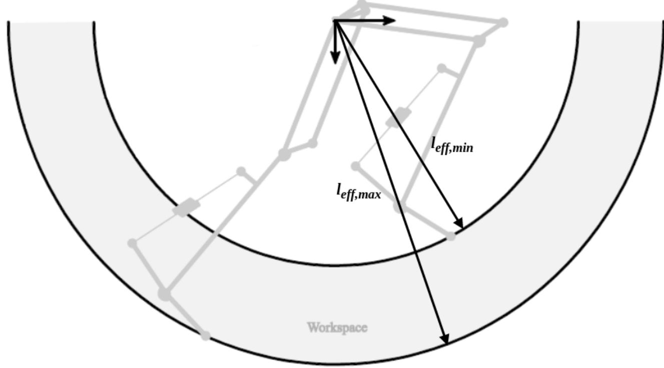


Figure 4.10: *The leg's workspace [17].*

Note: The *EtherCAT application* is the one which is implemented with the EtherCAT protocol and refers to data exchange via the Input and Output EtherCAT variables. When the *developed application* is mentioned, this refers to the developed application in the context of this thesis (which eventually runs the EtherCAT application).

Each EtherCAT Slave enables each leg to form semi-elliptical trajectories, with all the parameters of these trajectories controlled by the master, as introduced in [116]. Thus, only the slaves perform the necessary computations for the motion control of each leg [17]. The master merely provides the necessary parameters for the desired elliptical trajectory, listed in the *TrajectoryParameters* Record in Table 4.1 [17]. The computed ellipse is defined in (4.5) w.r.t. point 0 (hip axis) defined in Figure 4.9 [17]. The ellipse needs to be always inside the limits of the leg's workspace [17]. Consequently, each slave's firmware is specifically programmed to disregard parameters which produce invalid ellipses [17]. The firmware will continue to serve the last (x_{traj}, y_{traj}) point until a new valid point is passed to it [17].

$$\begin{aligned} x_{traj} &= x_{traj,cntr} + \alpha cos(\omega_{traj}t + \phi) \\ y_{traj} &= y_{traj,cntr} + bsin(\omega_{traj}t + \phi) \end{aligned} \tag{4.5}$$

To model the impedance of the treadmill's floor, a flattening parameter has been added on the y semi-minor axis (b), altering the shape of the elliptical trajectory [17]. The different positions along this semi-elliptical trajectory are shown in Figure 4.11 [17].

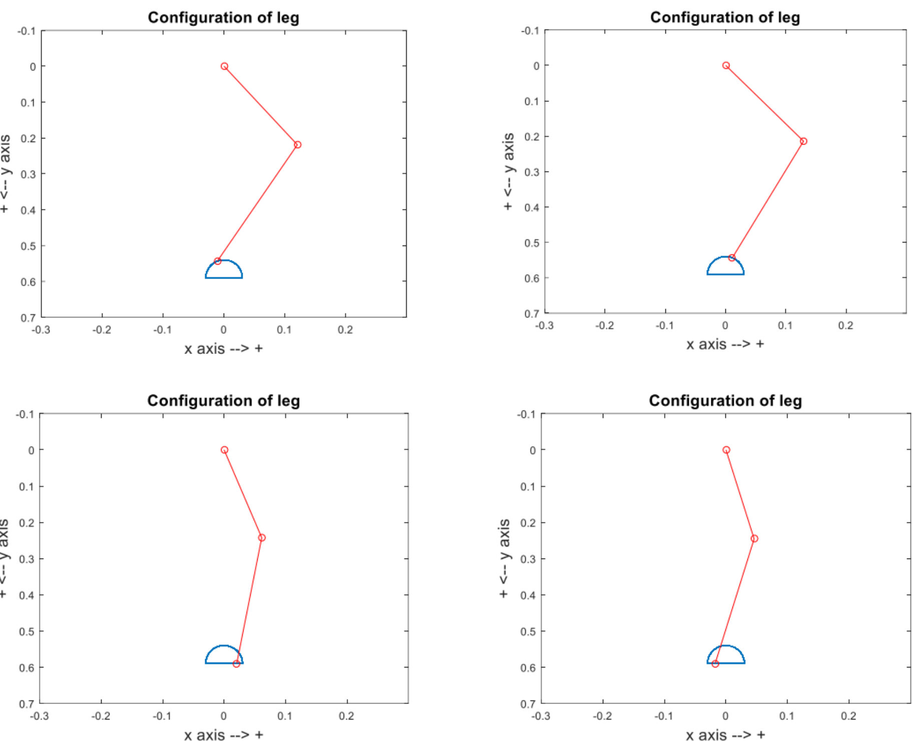


Figure 4.11: *Different positions along the semi-elliptical trajectory [17].*

From the aforementioned analysis in (4.5), the necessary parameters for the semi-elliptical trajectories are realized. These include the centre of the trajectory in the x axis ($x_{traj,cntr}$),

the centre of the trajectory in the y axis ($y_{traj,cntr}$), the semi-major x axis (α), the semi-minor y axis (b), the flattening parameter for the creation of semi-ellipses ($flattness_param$), and finally the trajectory frequency (ω_{traj}). These parameters are of utmost importance for the creation of the correct semi-ellipses by the slaves, via the correct commands of the master. However for consistency reasons, the whole set of the EtherCAT input and output variables shared among master and slaves with comments for each one, along with the most important ones aforementioned, are presented in Table 4.1 and Table 4.2.

Note: The Output EtherCAT variables are variables the master sends to the slaves. Thus, they are **written** by the master and **read** by the slaves. The Input EtherCAT variables are variables the master receives from the slaves. Thus, they are **written** by the slaves and **read** by the master.

Table 4.1: *EtherCAT Laelaps II Motion Control Output variables.*

Index	Subindex	Data Type	Name	Comments
0x7000	Record		Buttons	
	0x01	`bool`	`State_Machine`	State Machine variable
	0x02	`bool`	`Initialize_clock`	not used
	0x03	`bool`	`Initialize_-angles`	not used
	0x04	`bool`	`Inverse_-Kinematics`	not used
	0x05	`bool`	`Blue_LED`	light Blue LED
	0x06	`bool`	`Red_LED`	light Red LED
	0x07	`bool`	`Button1`	not used
	0x08	`bool`	`Button2`	not used
	0x09	`int8`	`Transition_time`	Time for smooth transition functions (sec)
0x7010	Record		Desired_x_value	
	0x01	`int32`	`Desired_x_value`	Not read by slave (for future use)
0x7012	Record		TargetMode	
	0x01	`uint16`	`FilterBandwidth`	First order lag filter frequency (Hz)
0x7014	Record		Desired_y_value	

	0x01	int32	`Desired_y_value`	Not read by slave (for future use)
0x7020	Record		ControlGains	PIV Gains
	0x01	int16	`Kp100_knee`	Proportional gain of knee motor / 100
	0x02	int16	`Kd1000_knee`	Velocity gain of knee motor / 1000
	0x03	int16	`Ki100_knee`	Integral gain of knee motor / 100
	0x04	int16	`Kp100_hip`	Proportional gain of hip motor / 100
	0x05	int16	`Kd1000_hip`	Velocity gain of hip motor / 1000
	0x06	int16	`Ki100_hip`	Integral gain of hip motor / 100
0x7030	Record		TrajectoryParameters	Semi-elliptical trajectory parameters
	0x01	int16	`x_cntr_traj1000`	x centre of the ellipsis (mm)
	0x02	int16	`y_cntr_traj1000`	y centre of the ellipsis (mm)
	0x03	int16	`a_ellipse100`	semi-major x axis (cm)
	0x04	int16	`b_ellipse100`	semi-minor y axis (cm)
	0x05	int16	`traj_freq100`	Trajectory's frequency (Hz) / 100
	0x06	int16	`phase_deg`	Trajectory's initial phase (deg)
	0x07	int16	`FlatnessParam100`	Flatness parameter of y axis / 100

Table 4.2: *EtherCAT Laelaps II Motion Control Input variables.*

Index	**Subindex**	**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		FeedbackTime	
	0x01	uint16	`Time`	Time variable from slave device (sec)
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	PIV Gains
	0x01	int32	velocity_knee1000	Rotational speed of knee (rad/s) * 1000
	0x02	int32	velocity_hip1000	Rotational speed of hip (rad/s) * 1000

4.2.2.7 EtherCAT Slave Network

The EtherCAT Slave Network is depicted in (7), in Figure 4.2. Every component of this network comprises of internal hardware and software architecture, which is briefly introduced. More information for the hardware and the software architecture of these components, can be found in [17, 109].

As far as the software architecture of the EtherCAT slaves is concerned, the handling of the Process Data Objects (PDOs) in the EtherCAT slave can be separated in two main steps as depicted in Figure 4.12 [17]:

- Low level *on-the-fly* data exchange: The ESC reads/writes data from/to the EtherCAT frame and stores/reads the data to/from the internal DPRAM.
- The slave application performs further data processing.

In each slave, a microcontroller is responsible for the entire application layer. As outlined in Figure 4.13, the EtherCAT slave software stack consists of three main parts [17]:

- Process Data Image (PDI) and Hardware abstraction which is hardware specific and needs to be implemented according to the platform/PDI. In the Slave Application, Serial Peripheral Interface (SPI) plays this role which is the means of communication between the MCU and the EtherCAT Slave Controller (see below).

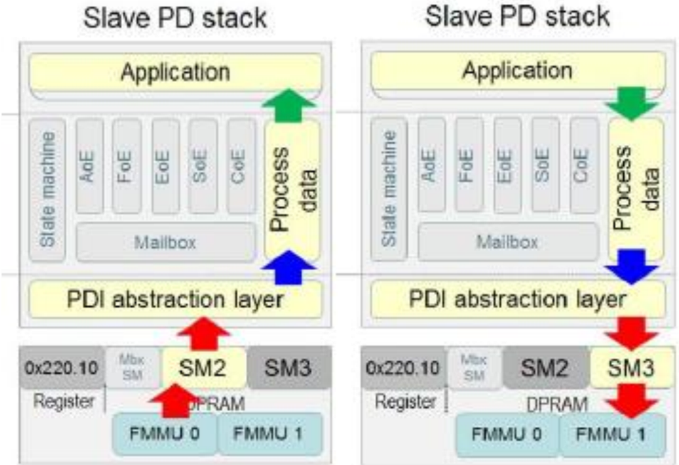


Figure 4.12: *EtherCAT Process Data handling in the slaves [17].*

- Generic EtherCAT stack that corresponds to all those functionalities which are not hardware and application specific for a slave, such us the full EtherCAT state machine, mailbox communication and generic process data exchange.
- User application which implements the slave specific functions i.e. motor control.

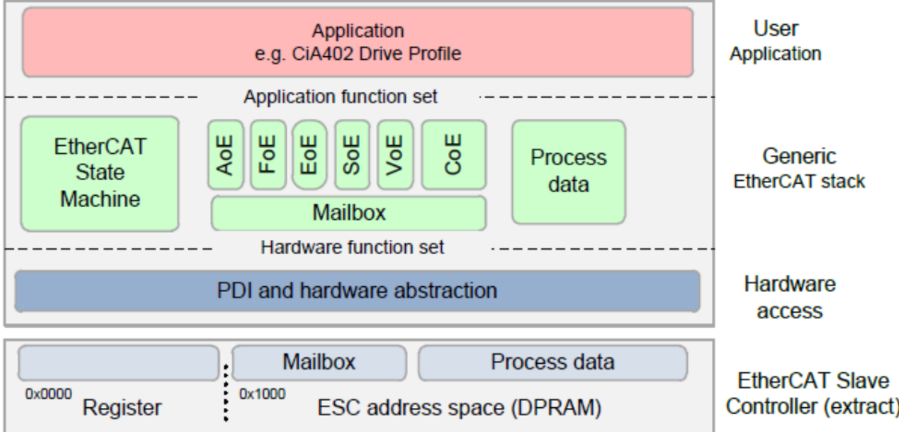


Figure 4.13: *EtherCAT slave software architecture [17].*

As far as the slave hardware architecture is concerned, this comprises of three hardware components [17]:

- An EtherCAT Slave Controller (ESC) which handles the EtherCAT protocol in real-time by processing the EtherCAT frames *on-the-fly* and provides the interface for data exchange between a master and a slave. The ESC is responsible for the realization of the *Physical* and *Data Link* Layers.
- A host Micro Controller Unit (MCU) realizing the *Application Layer* including the

Hardware Access, the Generic EtherCAT stack and User Application structures as presented in Figure 4.13.

- A custom printed circuit board connecting these two devices (the green board in the following figures).

For Laelaps II needs, the C2000 Delfino **MCU** F28379D LaunchPad Development Kit by Texas Instruments (TI) (Figure 4.14) has been selected, as a low cost and powerful MCU, to become the host microcontroller of all EtherCAT slaves [17].

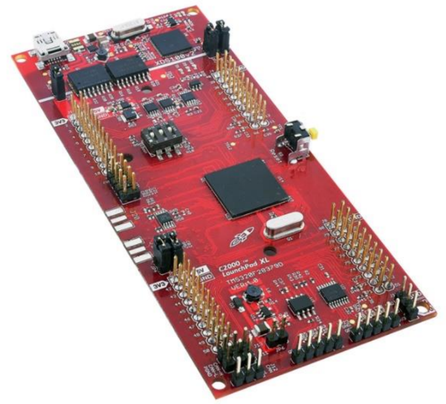


Figure 4.14: *The EtherCAT slave MCU [17].*

Regarding the EtherCAT Slave Controller (ESC), the FB1111-0141 (SPI) **ESC** by Beckhoff (Figure 4.15), has been selected [17]. It's a flexible ESC which communicates with the MCU via the Serial Peripheral Interface (SPI) protocol and operates in DC Synchronous mode triggered by three external interrupt signals [17].

Figure 4.15: *The EtherCAT slave ESC [17].*

Each leg of Laelaps II is being controlled by one *EtherCAT Control Tower Assembly* which realizes an EtherCAT slave in the configured network. Thus, four identical assemblies are constructed and used to control Laelaps II [17]. Figure 4.16 shows the final version of the

EtherCAT Control Tower Assembly [17]. Except for the aforementioned components, the assembly includes [17]:

- A TMS320F28379D Extension board interfacing with all necessary peripherals (ePWM, eQEP etc.) for two motors presented in Section 4.4 of [109].
- A voltage regulator (DC - DC converter, Step – Down 5V 2A USB[10]) supplying the logic power to the whole assembly.
- A plexiglass supporting base for mounting purposes on the Laelaps body.

In Figure 4.16 also notice the custom printed circuit board for connecting the MCU with the ESC in green color.

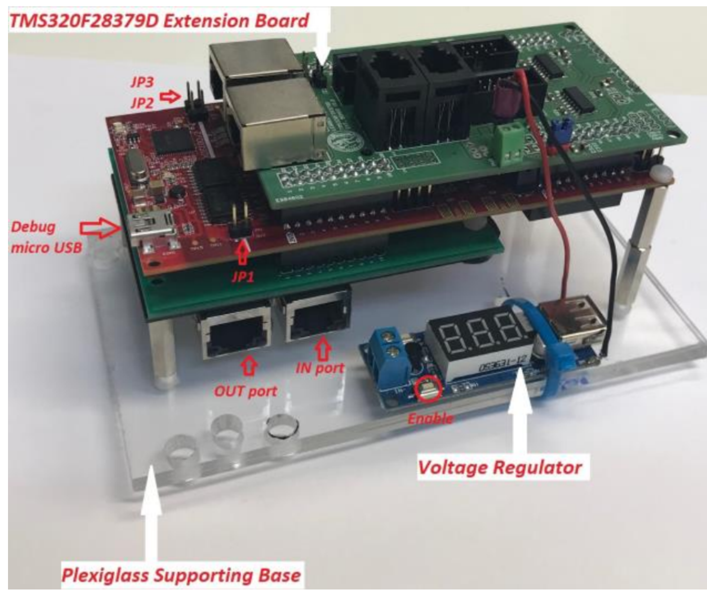


Figure 4.16: *EtherCAT Control Tower Assembly [17].*

Finally, the entire EtherCAT Control Tower Assembly mounted on Laelaps II robot is shown in Figure 4.17 [17]. All four slave devices are connected to the EtherCAT network. The first is on the Hind Right Leg and the last on the Fore Right Leg [17].

4.2.2.8 Electrical & Actuation systems

The actuation system of Laelaps II is pictured in (8), in Figure 4.2. Prior to introducing the actuation systems chosen in Laelaps II, it's worth to mention briefly the electrical system, in order to acquire a general grasp of the overall architecture. For further information and details on the electrical system of Laelaps II, the reader is referred to the exhaustive and excellent

[10]https://grobotronics.com/dc-dc-step-down-5v-2a.html

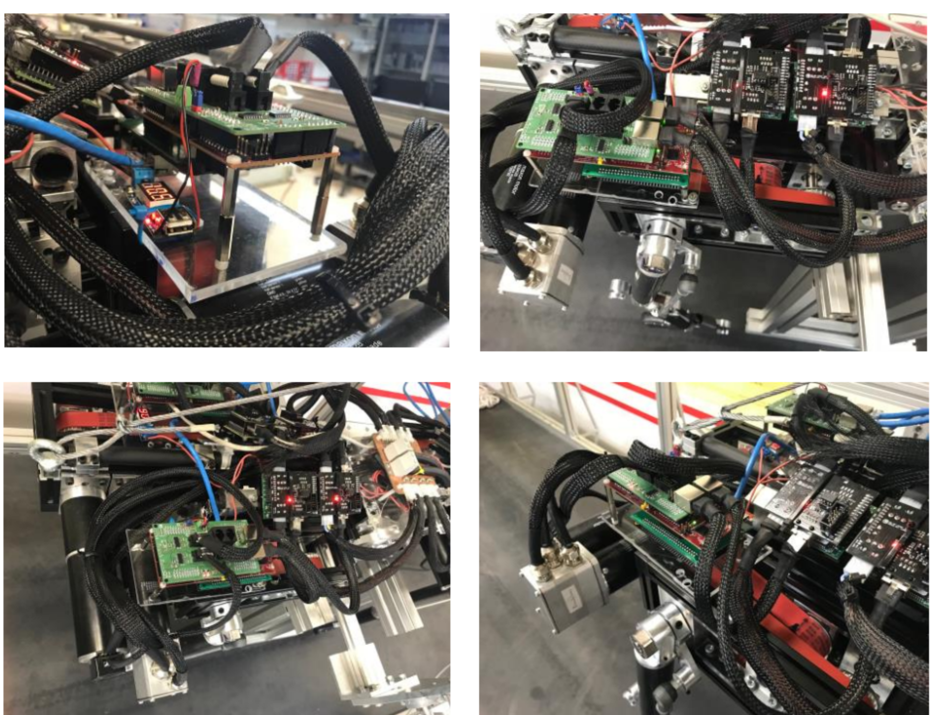


Figure 4.17: *EtherCAT Control Tower Assembly on Laelaps II [17].*

approach in [109, Chapter 4].

Electrical system

The main electrical components of Laelaps II are [17]:

- The High Power Distribution board which provides high power to all drivers.
- The Logic Power supply system with voltage regulators (5V) supplying all EtherCAT towers.
- Eight motor driver boards (amplifiers) configured for current control. Four of the drivers are connected to brushed motors, which drive the knee of each leg and the rest are connected to brushless motors, which control the hip motion.
- Four EtherCAT Control Tower Assembly slaves (introduced briefly above), connected to the motor drivers and the encoders of each leg.

It should be noted that each set of EtherCAT tower and connected drivers controls the leg of the opposite side (left $\rightarrow$ right), because of the way the motors are mounted to the body [17]. For example, the EtherCAT Control Tower Assembly and the two motor drivers shown in Figure 4.18, control the motion of the Fore Right Leg and **NOT** the Fore Left Leg which is

visible in the same figure [17]. This detail is important when operating the software on the master's as well on the slaves' side since the users should not be confused which EtherCAT Control Tower Assembly corresponds to which Laelaps leg [17].

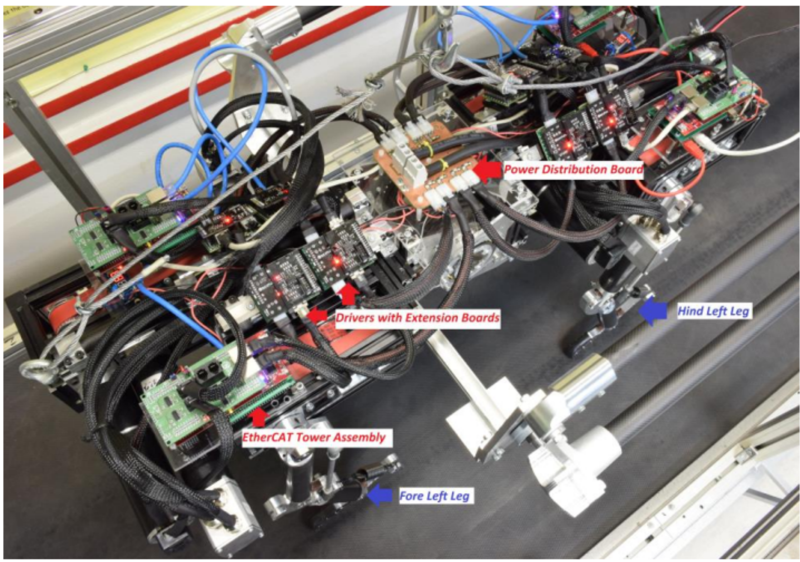


Figure 4.18: *Electrical System of Laelaps II [17].*

Actuation system

In Laelaps II different combinations of motors and gearheads are used for driving its knee and hip joints, but in both cases, a pulley with a specific gear ratio (48/26) is mounted to reduce the rotational speed of the motor and increase the output torque [17]. All motors and gearheads are purchased from Maxon Motors[11] [17]. For the hip joints, *EC 45, Ø45 mm, brushless motors, 250 Watt*[12] are used along with the *Planetary Gearhead GP 52 C Ø52 mm, 4–30 Nm*[13] with a gear ratio of 343/8 [17]. For the knee joints, *RE 50, Ø50 mm, Graphite Brushes motors, 200 Watt*[14] are used along with the *Planetary Gearhead GP 52 C Ø52 mm, 4–30 Nm*[15] with a gear ratio of 637/12 [17]. More information on the actuation system, the type of control scheme and the electronics created for controlling the motors, can be found in [17, 109].

[11] `https://www.maxonmotor.com`
[12] Motor Datasheet
[13] Gearhead Datasheet
[14] Motor Datasheet
[15] Gearhead Datasheet

4.2.3 Application Programming Interface

In this subsection the Application Programming Interface (API) provided by the project's software to other ROS software, is presented. As already highlighted, this API should be created in the context of a ROS API (much like a REST API for web applications), since the software should interoperate with other ROS nodes. The ROS API is presented in Table 4.3.

Table 4.3: *ROS API of the software project.*

Package Name:	ether_ros	
Node type:	ether_ros	
Node name:	ether_comm	
Publishers:		
	Topic Name:	/pdo_raw
	Message Name:	PDORaw.msg
	Message Type:	Header header uint8[] pdo_in_raw uint8[] pdo_out_raw
	Topic Name:	/pdo_in_slave_x , x $\in [0-3]$
	Message Name:	PDOIn.msg
	Message Type:	Header header int16 hip_angle int16 desired_hip_angle uint16 time int16 knee_angle int16 desired_knee_angle int16 PWM10000_knee int16 PWM10000_hip int32 velocity_knee1000 int32 velocity_hip1000
	Message Type:	
	Topic Name:	/pdo_out
	Message Name:	PDOOut.msg

	Message Type:	`Header header`
		`uint8 slave_id`
		`bool state_machine`
		`bool initialize_clock`
		`bool initialize_angles`
		`bool inverse_kinematics`
		`bool blue_led`
		`bool red_led`
		`bool button_1`
		`bool button_2`
		`int8 sync`
		`int32 desired_x_value`
		`uint16 filter_bandwidth`
		`int32 desired_y_value`
		`int16 kp_100_knee`
		`int16 kd_1000_knee`
		`int16 ki_100_knee`
		`int16 kp_100_hip`
		`int16 kd_1000_hip`
		`int16 ki_100_hip`
		`int16 x_cntr_traj1000`
		`int16 y_cntr_traj1000`
		`int16 a_ellipse100`
		`int16 b_ellipse100`
		`int16 traj_freq100`
		`int16 phase_deg`
		`int16 flatness_param100`

	Topic Name: Message Name: Message Type:	**`/pdo_out_timer`** **`PDOOut.msg`** The same as in **`/pdo_out`**
Subscribers:		
	Topic Name: Message Name:	**`/pdo_listener`** **`ModifyPDOVariables.msg`**

	Message Type:	uint8 slave_id uint8 index uint8 subindex bool bool_value uint8 uint8_value int8 int8_value uint16 uint16_value int16 int16_value uint32 uint32_value int32 int32_value uint64 uint64_value int64 int64_value string type
	Topic Name:	**/pdo_raw** **2 subscribers:** **1 in PDOInPublisher** **1 in PDOOutPublisher**
	Message Name:	**PDORaw.msg**
	Message Type:	Header header uint8[] pdo_in_raw uint8[] pdo_out_raw
Services:		
	Service Name:	**/ethercat_communicatord**
	Service Type:	**EthercatCommd.srv**
	Request type:	string mode
	Response type:	string success
Actions:		
	Action Name:	-
	Request type:	-
	Feedback type:	-
	Response type:	-
Parameters:		
	File:	config/ethercat_slaves.yaml

	Name	Default value	Comments
	/ethercat_-slaves/(front / back)_(left / right)_-leg/vendor_id	0x00000A12	The Vendor ID is a specific ID given to a vendor, member of the ETG (EtherCAT Technology Group). For NTUA CSL this is the default value. It's necessary to be specified, otherwise the slave won't be configured by EtherLab
	/ethercat_-slaves/(front / back)_(left / right)_leg/alias	0	A zero alias means to use simple position addressing. For more information see the documentation in [2]
	/ethercat_-slaves/(front / back)_(left / right)_-leg/position	[0-3]	The position of the slave(s) in the network
	/ethercat_-slaves/(front / back)_(left / right)_-leg/product_code	0x00a986fd	A specific number which is stored in the slave's E^2PROM and is flashed when downloading the firmware to the slave. Defined also in the ESI file of the slave.

	/ethercat_slaves/(front / back)_(left / right)_leg/assign_activate	0x0700	A specific value found in the ESI file of the slave and necessary for tuning the slave in DC Synchronization mode
	/ethercat_slaves/(front / back)_(left / right)_leg/input_port	0x6010	The port for the input PDOs. Defined in the ESI.
	/ethercat_slaves/(front / back)_(left / right)_leg/output_port	0x7000	The port for the output PDOs. Defined in the ESI.
	/ethercat_slaves/period_ns	400000	The period the control loop is working on. It's defined in nanoseconds. Should it be changed, it should with caution.
	/ethercat_slaves/run_time	360000	The run time of the experiments. Defined in seconds. It's set for 100 hours :)
	/ethercat_slaves/sync0_shift	55000	This parameter is very crucial to understand. Refer to Chapter 5 for more details.

5 Implementation

Nothing ever comes to one, that is worth having, except as a result of hard work.

Booker T. Washington

In this chapter, a detailed description is given of the implementation of the software project, which resides in the ROS environment, utilizes EtherLab and complies to real-time requirements, as previously analyzed in Chapter 4. Furthermore, the rationale behind each optimization applied upon the initial approach, is presented. Then, the installation and configuration process, along the path of a fully operational testing environment, are outlined. Finally, the methods and tools used to ensure the correctness of the code, are descrided in detail.

In this chapter, only selected segments of code are included and discussed, in order to aid the reader's understanding of the implementation. The complete source code is available on `https://github.com/mikekaram/ether_ros`.

5.1 Software Implementation

In this section, a list of the key classes, functions and structures that were implemented, are provided, along with a brief explanation. This list refers to the final, *optimized* implementation.

`ether_ros`: This is the main source file of the EtherROS project (also the name of the ROS package). Its main purpose is to initialize all the services, topics, data handlers and threads for

the operation of the program. Initially it requests and configures a master using the EtherLab API. After the correct configuration of the master, it moves to acquiring a *domain* for the process data to be used (*domains* allow grouping of process data transfers with different slave groups and task periods). In the quadruped's case, the EtherCAT slaves should be in the same slave group and have the same task periods, therefore only one domain is created. Then it continues by initializing the EtherCAT slaves by calling their `init()` method, the EtherCAT Communicator, the Input PDO Publisher, the Output PDO Publisher, the Output PDO Listener, the Output PDO Publisher Timer and the EtherCAT Communicator Daemon service. Finally it opens the log file, if there are statistics for logging and calls the famous `ros::spin()` function, which spins a thread for handling all the registered message communications (topics, services, actions) by calling the corresponding registered handler.

Note: prior to the request for a master, the program locks the memory pages it will use in advance with the following code in `main()`:

```
1  ...
2  if (mlockall(MCL_CURRENT | MCL_FUTURE) == -1)
3  {
4     ROS_FATAL("mlockall failed");
5     exit(1);
6  }
7  ...
```

Listing 5.1: *The call to `mlockall()`.*

Linux processes access memory by using virtual addresses [63, Chapter 4]. Each virtual address translates into a physical address with the help of translation tables in the hardware [63, Chapter 4]. As all processes don't need all their allocated memory at the same time, it's possible to address more virtual memory than available physical memory [63, Chapter 4].

Allocating memory by default will only reserve a virtual memory range [63, Chapter 4]. When the first memory access to this newly allocated virtual memory occurs, this causes a page fault, which triggers a hardware interrupt [63, Chapter 4]. This interrupt indicates that the translation table does not contain the addressed virtual memory [63, Chapter 4]. The page fault interrupt will be handled by the Linux kernel, which will provide the virtual-to-physical memory mapping [63, Chapter 4]. Then the program execution continues [63, Chapter 4].

Most hardware architectures use a cache called translation lookaside buffer (TLB) as the translation table [63, Chapter 4]. The TLB cache is used to speed up virtual-to-physical memory translations [63, Chapter 4]. If the looked-up address is in the TLB (TLB hit), then the translation is done instantly. Otherwise (TLB miss) the address should be searched in the *Page Table* which introduces extra latency [63, Chapter 4].

Virtual memory makes it possible for Linux to have memory content stored in a disk and the data to be copied from the disk to physical memory when they are needed by the process [63, Chapter 4]. This is called *demand paging* and could cause unbounded latency [63, Chapter 4]. Thus, an application designed as real-time, such as the one developed, needs to disable demand paging by using the `mlockall()` function call: `mlockall(MCL_CURRENT | MCL_FUTURE)` [63, Chapter 4].

The `MCL_CURRENT` flag makes sure that all pages which are currently mapped into the address space of the process are locked and the TLB contains the needed virtual-to-physical memory mapping [63, Chapter 4]. This includes code, global variables, shared libraries, shared memory, stack and heap [63, Chapter 4]. The `MCL_FUTURE` flag makes sure that all pages which will become mapped into the address space of the process in the future are locked. means that updates to the TLB and initialization of the physical memory are performed during future allocations, not when accessing the memory [63, Chapter 4]. Future allocations can be stack growth, heap growth, new memory mapped files or shared memory regions, `shm_open()`, `malloc()`, or similar calls like `mmap()` [63, Chapter 4].

When the `mlockall()` system call is used, it's important to be called at the proper time [63, Chapter 4]. For instance, a call to `malloc()` after `mlockall()` is called, can still show large latency variation since the TLB is updated within this function call instead of when accessing the memory [63, Chapter 4]. Not to mention that a `malloc()` could request more virtual memory from the kernel [63, Chapter 4]. Thus, all needed dynamic memory should be allocated at the start of the real-time process, to avoid this extra latency [63, Chapter 4].

`EtherCAT slave`: This class represents the EtherCAT slaves communicating with this software. Its main purpose is to act as a placeholder for all the slave-oriented functions and variables. This class could easily be represented as a struct, however it was foreseen that this class can have many methods acting on it's member variables and it should be more appropriate to see it as an object, in which operations are performed on. The number of the EthercatSlave class objects equals to the number of the EtherCAT slaves in the network. The objects of this

class are instantiated from the `ether_ros` `main()`, as described above.

```
class EthercatSlave
{
  private:
    int vendor_id_;
    std::string slave_id_;
    int product_code_;
    int assign_activate_;
    int position_;
    int alias_;
    int input_port_;
    int output_port_;
    ec_slave_config_t *ighm_slave_; //pointer to the basic slave struct in
        EtherLab
    int pdo_in_;
    int pdo_out_;
    int32_t sync0_shift_;

  public:
    void init(std::string slave, ros::NodeHandle &n);
    int get_pdo_out();
    int get_pdo_in();
    ec_slave_config_t *get_slave_config();
};
```

Listing 5.2: *The `EthercatSlave` class definition.*

As seen by the class definition, most of the private variables of this class, are configuration parameters specified in the Subsection 4.2.3, needed for the correct configuration of the EtherCAT slaves and are parsed from the `ethercat_slaves.yaml` into the object's private variables. The class definition, ends with the initialization method and the methods for getting the parameters and storing them into the object's private variables. It's worth noting that the initialization method ends with a call to the `ecrt_slave_config_dc()`. This function is part of the EtherLab API and is used for configuring distributed clocks in a slave. It's declaration is the following:

```
void ecrt_slave_config_dc(
  ec_slave_config_t *sc, /**< Slave configuration. */
  uint16_t assign_activate, /**< AssignActivate word. */
  uint32_t sync0_cycle, /**< SYNC0 cycle time [ns]. */
  int32_t sync0_shift, /**< SYNC0 shift time [ns]. */
  uint32_t sync1_cycle, /**< SYNC1 cycle time [ns]. */
  int32_t sync1_shift /**< SYNC1 shift time [ns]. */
  );
```

Listing 5.3: *The `ecrt_slave_config_dc()` function declaration.*

This function sets the *AssignActivate* word (`assign_activate` argument) and the cycle and shift times for the sync signals. The *AssignActivate* word is vendor-specific and can be taken from the XML device description file.

In the DC synchronization mode, the synchronization signals need a shift in order to fire at the same time, after the SM events have finished (written input PDOs and read output PDOs), in every slave. This is necessary for all slaves to have synchronously valid Outputs. However, because of network delays and the master jitter, the SM events in the last slave require more time to trigger than in the other slaves, therefore this shift needs to be adequately large to avoid firing before every SM event has finished and small enough to avoid firing after the next cycle has started. All this information is described extensively in Subsection 3.1.6. The `sync0_shift` is a critical parameter and there doesn't exist an optimal value for this parameter. However, since there were many successful experiments carried out in [17], which used the TwinCAT's value, namely 55 μs, this value was chosen for this project also.

`EtherCAT Communicator`: This class represents the central thread, on which every component of the project depends on and is related to. Its core functionality has been summarized in Subsubsection 4.2.2.2 and has been illustrated in Figure 4.5. This class essentially represents a real-time thread which communicates with the EtherCAT network via the EtherLab API. The class definition is presented below:

```
class EthercatCommunicator
{
private:
  pthread_attr_t current_thattr_;
  struct sched_param sched_param_;
```

```
6    static int cleanup_pop_arg_;
7    //cleanup_pop_arg_ is used only for future references. No actual usage
        in our application.
8    //Serves as an argument to the cleanup_handler.
9    static pthread_t communicator_thread_;
10   static ros::Publisher pdo_raw_pub_;
11   static bool running_thread_;
12   static uint64_t dc_start_time_ns_;
13   static uint64_t dc_time_ns_;
14   static int64_t system_time_base_;
15
16 #ifdef SYNC_MASTER_TO_REF
17   static uint8_t dc_started_;
18   static int32_t dc_diff_ns_;
19   static int32_t prev_dc_diff_ns_;
20   static int64_t dc_diff_total_ns_;
21   static int64_t dc_delta_total_ns_;
22   static int dc_filter_idx_;
23   static int64_t dc_adjust_ns_;
24 #endif
25   static void *run(void *arg);
26   static void cleanup_handler(void *arg);
27   static void copy_data_to_domain_buf();
28   static void publish_raw_data();
29   static void sync_distributed_clocks(void);
30   static void update_master_clock(void);
31   static uint64_t system_time_ns(void);
32
33 public:
34   static bool has_running_thread();
35   void init(ros::NodeHandle &n);
36   void start();
37   void stop();
38 };
```

Listing 5.4: *The* `EthercatCommunicator` *class definition.*

One can observe in the private variables the scheduling related variables, the `pthread_t` object which realizes the implemented pthread used in the program, a variable for handling the publishing of the Process Data Objects (PDOs) (more on this later), some variables for operating the fundamental control loop and finally some variables under the preprocessor if, `#ifdef SYNC_MASTER_TO_REF`.

This macro is used to distinguish two operating modes the program can operate in, which can be implied by the brief description of DC Mode of synchronization in Subsubsection 3.1.7.2, however they are clarified here[1]:

- In the first operating mode, the EtherCAT master provides the master clock (`SYNC_-REF_TO_MASTER` defined) and in this case the synchronization in the DC mode from the EtherCAT master's side, functions in the folowing way:
  - The EtherCAT master computer is used as the DC master for the entire system. `ecrt_master_application_time()` is called in every cycle from `ether_ros` to tell the EtherLab master what the current PC time is.
  - Then `ecrt_master_sync_reference_clock()` is called in order to tell to the slave DC master to synchronize to the EtherLab master's time.
  - Finally `ecrt_master_sync_slave_clocks()` to tell all other DC slaves to sync to the slave DC master.
- In the second operating mode, the slave DC master provides the master clock (`SYNC_-MASTER_TO_REF` defined) and in this case the synchronization in the DC mode from the EtherCAT master's side functions in the folowing way:
  - `ether_ros` gets the slave DC master's time using `ecrt_master_reference_-clock_time()` and synchronizes the EtherLab master's cycle and time to it.
  - Then `ecrt_master_sync_slave_clocks()` is called in order to tell to all the other DC slaves to synchronize to the slave DC master.
  - Finally `ecrt_master_application_time()` is called with the next cycles master time.

Note: With the second option there is a need to adjust the EtherCAT master PC's time by the drift time from the slave DC master time and adjust the real-time cycle to it. This is done in

[1]More on the matter in `http://lists.etherlab.org/pipermail/etherlab-users/2016/003013.html`

`ether_ros` with a call to the `update_master_clock()` method of the EthercatCommunicator class. This method will be discussed later on when it's called in the code.

Note: The second option appears to be better, it does not introduce jitter compared to the first option. This is the default option used in TwinCAT.

After this brief introduction on the two synchronization modes in the DC mode, it's useful to examine further the code of some key methods of this class:

`EthercatCommunicator::init()`

```
1  void EthercatCommunicator::init(ros::NodeHandle &n)
2  {
3      ...
4
5      if (pthread_attr_init(&current_thattr_))
6      {
7          ROS_FATAL("Attribute init\n");
8          exit(1);
9      }
10     if (pthread_attr_setdetachstate(&current_thattr_,
           PTHREAD_CREATE_JOINABLE))
11     {
12         ROS_FATAL("Attribute set detach state\n");
13         exit(1);
14     }
15     if (pthread_attr_setinheritsched(&current_thattr_,
           PTHREAD_EXPLICIT_SCHED))
16     {
17         ROS_FATAL("Attribute set inherit schedule\n");
18         exit(1);
19     }
20     /*
21         Use the SCHED_FIFO for now. It should be tested later if there is
               a better scheduler (see: SCHED_DEADLINE, EDF + CBS)
22     */
23     if (pthread_attr_setschedpolicy(&current_thattr_, SCHED_FIFO))
24     {
```

```
25          ROS_FATAL("Attribute set schedule policy\n");
26          exit(1);
27      }
28      ret = pthread_attr_setschedparam(&current_thattr_, &sched_param_);
29
30      if (ret != 0) handle_error_en(ret, "pthread_attr_setschedparam");
31      ...
32
33      //Create ROS publisher for the Ethercat RAW data
34      pdo_raw_pub_ = n.advertise<ether_ros::PDORaw>("pdo_raw", 1000);
35
36
37  }
```

Listing 5.5: *The `EthercatCommunicator::init` method.*

This method is called from `main()`; it's main purpose is to initialize the attributes of the real-time thread which handles the sending and receiving of Process Data Objects (PDOs) from the EtherCAT network. The most important of them are the scheduling policy (set with a call to `pthread_attr_setschedpolicy()`) and the scheduling parameters (set with a call to `pthread_attr_setschedparam()`). With them, the scheduling policy as well as some scheduling parameters related to this policy are defined for this thread and are provided to the Linux scheduler. For now the FIFO real-time scheduling policy (`SCHED_FIFO`), described in Subsection 2.4.1, is used. After the initialization of the attributes, the ROS topic `/pdo_raw` of the ROS publisher `pdo_raw_pub_` is advertised (more on this later).

`EthercatCommunicator::start()`

```
1   void EthercatCommunicator::start()
2   {
3       int ret;
4
5       ret = ecrt_master_select_reference_clock(master,
            ethercat_slaves[0].slave.get_slave_config());
6       ...
7       ROS_INFO("Activating master...\n");
8       if (ecrt_master_activate(master))
```

```
9       {
10          ROS_FATAL("Failed to activate master.\n");
11          exit(1);
12      }
13      domain1_pd = NULL;
14      if (!(domain1_pd = ecrt_domain_data(domain1)))
15      {
16          ROS_FATAL("Failed to set domain data.\n");
17          exit(1);
18      }
19      running_thread_ = true;
20
21      ret = pthread_create(&communicator_thread_, &current_thattr_,
            &EthercatCommunicator::run, NULL);
22      ...
23      ROS_INFO("Starting cyclic thread.\n");
24  }
```

Listing 5.6: *The `EthercatCommunicator::start` method.*

This method is called from the `EthercatCommunicatord` service when the user sends the start command for EtherCAT communication. This method starts by selecting the first EtherCAT slave in the network to be the reference clock (slave DC master), to which every node of the network will synchronize. Then, the function `ecrt_master_activate()` is called, which activates the master by requesting EtherLab to switch to Operational Mode (which will send a request to the EtherCAT slaves to reach *Operational* State, as shown in Subsection 3.1.4). After this call, the domain of the Process Data Objects (PDOs) is created by calling `ecrt_domain_data()` and finally the pthread is created. This thread will run a function which will have as arguments, the `run()` method to run and the attributes initialized in `init()`.

`EthercatCommunicator::run()`

```
1   void *EthercatCommunicator::run(void *arg)
2   {
3       ...
4       pthread_cleanup_push(EthercatCommunicator::cleanup_handler, NULL);
```

```
 5 #ifdef FIFO_SCHEDULING
 6     CPU_SET(3, &cpuset_);
 7 
 8     // set pthread affinity to CPU 3
 9     if (pthread_setaffinity_np(communicator_thread_, sizeof(cpuset_),
           &cpuset_))
10     {
11         ROS_FATAL("Set pthread affinity, not portable\n");
12         exit(1);
13     }
14 #endif
15     // get current time
16     clock_gettime(CLOCK_TO_USE, &wakeup_time);
17     clock_gettime(CLOCK_TO_USE, &break_time);
18     break_time = utilities::timespec_add(break_time, offset_time);
19     //PTHREAD_CANCEL_DEFERRED is the default but nevertheless
20     pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
21 /****************************************************/
22     do
23     {
24         // check if there is a request for cancel
25         pthread_testcancel();
26 
27         //set the cancel state to DISABLE
28         ret = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
29         ...
30 
31         wakeup_time = utilities::timespec_add(wakeup_time, cycletime);
32         clock_nanosleep(CLOCK_TO_USE, TIMER_ABSTIME, &wakeup_time, NULL);
33 #ifdef TIMING_SAMPLING
34         clock_gettime(CLOCK_TO_USE, &start_time);
35         create_statistics(&start_time);
36         last_start_time = start_time;
37 #endif
38 
39         // receive EtherCAT frame
40         ecrt_master_receive(master);
```

```
41
42      // receive process data
43      ecrt_domain_process(domain1);
44
45      // check the state of the domain
46      utilities::check_domain1_state();
47
48      // get statistics if the flags are enabled
49      if (!counter) //if counter is 0
50      {
51          // get statistics at 10 Hz
52          initialize_statistics_metrics();
53
54          // check for master state (optional)
55          utilities::check_master_state();
56      }
57      else counter--;
58
59      // move the data from process_data_buf to domain1_pd buf carefuly
60      utilities::copy_process_data_buffer_to_buf(domain1_pd);
61
62      //queue the EtherCAT data to domain buffer
63      ecrt_domain_queue(domain1);
64
65      // sync distributed clock just before master_send to set most
            accurate master clock time.
66      EthercatCommunicator::sync_distributed_clocks();
67
68      // send EtherCAT frame
69      ecrt_master_send(master);
70
71      // send the raw data to the raw data topic
72      EthercatCommunicator::publish_raw_data();
73
74      // update the master clock with the drift, if SYNC_MASTER_TO_REF
            defined
75      EthercatCommunicator::update_master_clock();
```

```
76
77          // set the cancel state to ENABLE
78          int ret = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
79          ...
80
81  #ifdef TIMING_SAMPLING
82          clock_gettime(CLOCK_TO_USE, &end_time);
83  #endif
84          clock_gettime(CLOCK_TO_USE, &current_time);
85      } while (DIFF_NS(current_time, break_time) > 0);
86  /*****************************************************/
87  #ifdef TIMING_SAMPLING
88      // write the statistics to file
89      log_statistics_to_file();
90  #endif
91      ...
92      running_thread_ = false;
93      exit(0);
94  }
```

Listing 5.7: *The* `EthercatCommunicator::run` *method.*

This method is executed by the pthread created in `start()`. It's the core method of `EthercatCommunicator` and implements the pipeline shown in Figure 4.5.

A quick description follows: The method starts by declaring the `cleanup_handler` to be used. This handler will be called when the thread is cancelled (i.e. when the EtherCAT Communicator is stopped). This handler could provide cleaning service, like freeing up memory used from dynamic data structures.

Generally, stopping a thread externally is a difficult task to accomplish. One of the mechanisms that the pthread library provides is the one that cancels a thread asynchronously without using custom shared variables, signals or other ways of message passing. That said, at the beginning of the control loop, there is a call to `pthread_setcancelstate()` with the flag `PTHREAD_CANCEL_DISABLE` (so that the loop will get to run uninterrupted) and at the end of the loop there is a call to `pthread_setcancelstate()` with the flag `PTHREAD_CANCEL_ENABLE`. Prior to the first call to `pthread_setcancelstate()` the cancel state is tested with a

call to `pthread_testcancel()`, hence if there is a request for cancellation, it won't interrupt the execution of the control loop and the thread will recognize the request in the beginning of the loop and exit. In this manner, there is a clean and straight way of thread cancellation without customized solutions.

Returning to the same point after the declaration of the `cleanup_handler`, if `FIFO_SCHEDULING` is defined (for now FIFO is the only policy well supported), the thread running `run()` is pinned to a specific CPU, namely CPU 3, by calling `pthread_setaffinity_np()`. More on the CPU pinning will be discussed in the Subsection 5.3.1.

Next, the code continues to the control loop, after computing some time parameters necessary for running the loop. In the control loop, after `pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, ...)` there is a call to `clock_nanosleep()` which makes sure that the thread will sleep for a fixed time interval (defined in nanoseconds). This time interval is defined in the `ethercat_slaves.yaml` file.

Next, after creating the statistics metrics (if `LOGGING` is defined) the core functions are performed. Namely, the thread tells the EtherLab master to receive the EtherCAT frame by calling `ecrt_master_receive()` and then it requires to receive its Process Data Objects (PDOs) defined in the domain used, by calling `ecrt_domain_process()`. Then, it continues to check the state of the domain, by checking if anything changed in the Working Counter. The Working Counter is related to the EtherCAT commands used (refer to Subsubsection 3.1.3.2), like LRW, and it's value corresponds to the number and the kind of commands actually being carried out. After this check, it creates a new sample with timing metrics (latency, execution, period) for logging (if `LOGGING` is defined).

The method then proceeds by moving the data from the `process_data_buf` buffer (which is filled by `PDOOutListener class`) to the `domain1_pd` buffer. The `domain1_pd` is the buffer used by EtherLab to get the process data from the network and to send the new process data to the network. The synchronization process was briefly introduced in Figure 4.5 and it is discussed further in `copy_process_data_buffer_to_buf()`.

After the new process data have been copied to the `domain1_pd` buffer, they are ready to be sent. Indeed, the method continues by calling `ecrt_domain_queue()` to queue the new Process Data Objects (PDOs) to EtherLab's internal domain queue. Then synchronization of the distributed clocks is performed by calling `sync_distributed_clocks()`. The operations of this method were discussed above, when the two synchronization methods in DC

mode were described. The two methods are both supported, however the `SYNC_REF_TO_-MASTER` is the default, since by testing both of them, the performance of the master was the same.

Next, `ecrt_master_send()` is called in order to send the new process data to the network via EtherLab.

Note: The process of sending and receiving the EtherCAT frames by EtherLab was thoroughly described in Figure 4.6 and in Figure 4.7 respectively.

After sending the process data to the EtherCAT network, publishing these data to the ROS network takes place by calling `publish_raw_data()`. More details on how this is done are presented in the description of this method. Finally, `update_master_clock()` is called for updating the master clock with the time drift, if `SYNC_MASTER_TO_REF` is defined, and the current time is compared with the time for breaking the loop in the `while()` command. This concludes the control loop pipeline in the `run()` method, which is run by the real-time pthread. After the control loop and if `LOGGING` is defined, the statistics metrics are written to the log file, and the `run()` method exits.

`EthercatCommunicator::publish_raw_data()`: This method is called from the `run()` method and it's purpose is to publish the process data, as soon as they have been received from the EtherCAT network, to the ROS network. However these data are buffer data and don't make sense because they are not formatted to the EtherCAT variables used (defined in Subsection 4.2.3). Consequently, these unformatted data are not sent directly to the ROS network. First these data are sent to a ROS node, which will format them to EtherCAT variables and then publish them to the ROS network. This solution was chosen, since communicating with the EtherCAT network and formatting the data with the specific EtherCAT variables used, can be decoupled. Hence, the real-time communication which does not depend on the type of variables used, can remain unchanged. In addition, if the EtherCAT variables used change in the future, only the formatters will have to change, since they depend on the type of EtherCAT variables used and as a result, software modularity is achieved. The method is presented below:

```
1 void EthercatCommunicator::publish_raw_data()
2 {
3     // Create raw data vectors
4     std::vector<uint8_t> input_data_raw, output_data_raw;
```

```
5      std::vector<uint8_t> input_vec, output_vec;
6      unsigned char *raw_data_pointer;
7
8      // Create input data raw string
9      for (int i = 0; i < master_info.slave_count; i++)
10     {
11         raw_data_pointer = (unsigned char *)domain1_pd +
               ethercat_slaves[i].slave.get_pdo_in();
12         input_vec.insert(std::end(input_vec), raw_data_pointer,
               raw_data_pointer + num_process_data_in);
13     }
14     input_data_raw.insert(std::end(input_data_raw), std::begin(input_vec),
           std::end(input_vec));
15
16     // Create output data raw string
17     for (int i = 0; i < master_info.slave_count; i++)
18     {
19         raw_data_pointer = (unsigned char *)domain1_pd +
               ethercat_slaves[i].slave.get_pdo_out();
20         output_vec.insert(std::end(output_vec), raw_data_pointer,
               raw_data_pointer + num_process_data_out);
21     }
22     output_data_raw.insert(std::end(output_data_raw),
           std::begin(output_vec), std::end(output_vec));
23
24     // Send both strings to the topic
25     ether_ros::PDORaw raw_data;
26     raw_data.pdo_in_raw = input_data_raw;
27     raw_data.pdo_out_raw = output_data_raw;
28     pdo_raw_pub_.publish(raw_data);
29 }
```

Listing 5.8: *The `EthercatCommunicator::publish_raw_data` method.*

The method starts by wrapping the data in the `domain1_pd` buffer (`unsigned char *` $\rightarrow$ `std::vector<uint8_t>`). This wrapping is necessary in order to publish these data through a topic in ROS, which natively supports C++. The input Process Data Objects are copied in the

input_data_raw vector and the output Process Data Objects are copied in the output_-data_raw vector. Then a ether_ros::PDORaw message is created, and the two "raw" vectors are wrapped into the raw_data message field. Finally this message is published to the ROS network through the pdo_raw_pub_ publisher's publish method.

Output PDO Listener: This class is responsible for receiving the input from the users, namely the output Process Data Objects (PDOs), and safely fill the shared process_data_-buf buffer. Later on, the data of this buffer will be copied from the real-time pthread running the run() method, safely into the domain1_pd buffer, in order to be sent to the EtherCAT network. The synchronization scheme for safe operations on the shared process_data_buf buffer was briefly described in Figure 4.5 and is further discussed in copy_process_data_-buffer_to_buf(). The definition of the class is the following:

```
1  class PDOOutListener
2  {
3    private:
4      ros::Subscriber pdo_out_listener_;
5      std::map<std::string, int> int_type_map_ = {
6          {"bool", 0},
7          {"uint8", 1},
8          {"int8", 2},
9          {"uint16", 3},
10         {"int16", 4},
11         {"uint32", 5},
12         {"int32", 6},
13         {"uint64", 7},
14         {"int64", 8}
15     };
16
17   public:
18     void init(ros::NodeHandle & n);
19     void pdo_out_callback(const ether_ros::ModifyPDOVariables::ConstPtr
           &new_var);
20     void modify_pdo_variable(int slave_id, const
           ether_ros::ModifyPDOVariables::ConstPtr &new_var);
21 };
```

Listing 5.9: *The PDOOutListener class definition.*

The definition of the private variables starts with the definition of the ROS subscriber `pdo_out_listener_`, which subscribes to `/pdo_listener` topic. Then the `int_type_map_` is defined, which maps data types like `bool` and `uint8` coming as strings to integers from 0 to 8. This was done in this way, in order to allocate the correct amount of memory, in order for these data to be copied to the `process_data_buf` buffer. In the public scope, the public methods of the class are declared. In the `init()` method the ROS subscriber subscribes to the topic aforementioned. In the `pdo_out_callback()` method, the new output Process Data Objects are received and `modify_pdo_variable()` is called to process them. The processing doesn't take place on the `pdo_out_callback()` method, since `slave_id`, a message field in `ether_ros::ModifyPDOVariables`, can have value between 0 and 255. The 255 is reserved for multicasting the change to an EtherCAT variable, to all EtherCAT slaves. In this case, the `modify_pdo_variable()` method is called for every slave.

`Input PDO Publisher`: This class is responsible for formatting the input Process Data Objects, received from the aforementioned ROS publisher `pdo_raw_pub_`, into Input EtherCAT variables (defined in Table 4.2) and publishing them to the ROS network. The definition of the class is the following:

```
class PDOInPublisher
{
    private:
      ros::Subscriber pdo_raw_sub_;
      ros::Publisher * pdo_in_pub_;

    public:
      void init(ros::NodeHandle &n);
      void pdo_raw_callback(const ether_ros::PDORaw::ConstPtr &pdo_raw);
};
```

Listing 5.10: *The `PDOInPublisher` class definition.*

In the private scope, the `pdo_raw_sub_` ROS subscriber and the `pdo_in_pub_` ROS publisher are defined. The former is used for subscribing to the `/pdo_raw` topic to receive the "raw" Process Data Objects (PDOs) and the latter is used for publishing the newly formatted Input EtherCAT variables to topics, one for each existing EtherCAT slave. In the public scope, the `init()` method is used for initializing the two private variables and the `pdo_raw_call-`

`back()` is further discussed below:

`PDOInPublisher::pdo_raw_callback()`

```
1  void PDOInPublisher::pdo_raw_callback(const ether_ros::PDORaw::ConstPtr
       &pdo_raw)
2  {
3      std::vector<uint8_t> pdo_in_raw = pdo_raw->pdo_in_raw;
4      uint8_t *data_ptr;
5      size_t pos;
6      for (int i = 0; i < master_info.slave_count; i++)
7      {
8          pos = i * num_process_data_in; //The size of every entry is
               num_process_data_in
9          data_ptr = (uint8_t * ) & pdo_in_raw[pos];
10         ether_ros::PDOIn pdo_in;
11         using namespace utilities;
12
13         // change the following code to match your needs
14         /*
15             Insert code here ...
16         */
17
18         pdo_in.hip_angle = process_input_int16(data_ptr, 0);
19         pdo_in.desired_hip_angle = process_input_int16(data_ptr, 2);
20         pdo_in.time = process_input_uint16(data_ptr, 4);
21         pdo_in.knee_angle = process_input_int16(data_ptr, 6);
22         pdo_in.desired_knee_angle = process_input_int16(data_ptr, 8);
23         pdo_in.PWM10000_knee = process_input_int16(data_ptr, 10);
24         pdo_in.PWM10000_hip = process_input_int16(data_ptr, 12);
25         pdo_in.velocity_knee1000 = process_input_int32(data_ptr, 14);
26         pdo_in.velocity_hip1000 = process_input_int32(data_ptr, 18);
27
28         /*
29             .....
30         */
31         pdo_in_pub_[i].publish(pdo_in);
32     }
```

```
33 }
```

Listing 5.11: *The PDOInPublisher::pdo_raw_callback method.*

The method starts by receiving the "raw" input process data. Then a new ROS message of type `ether_ros::PDOIn` is created for every EtherCAT slave on the network, namely `pdo_in`. Next, this message is filled with the Input EtherCAT variables (which are extracted from the input process data buffer `data_ptr` by choosing the type of the variable and it's position inside the buffer). Finally, the message is published to the topic of the corresponding slave.

Note: The topics have names of the form `/pdo_in_slave_x`, where x is the `slave_id` of each slave.

Note: The position of the input EtherCAT variables inside the buffer can be computed, by measuring the bytes prior to the variables, from the output of the command `$ ethercat pdos` in a terminal.

`Output PDO Publisher`: This class is responsible for formatting the output Process Data Objects (PDOs), received from the aforementioned ROS publisher `pdo_raw_pub_`, into Output EtherCAT variables (defined in Table 4.1) and for publishing them to the ROS network. This class is very similar with the `PDOInPublisher` class discussed above, thus further description isn't necessary.

`Output PDO Publisher Timer`: This class was created for debugging and logging purposes. As far as it's structure is concerned, it's very similar to the `PDOOutPublisher` class. The only difference is that the callback is not triggered by a subscriber listening to a topic and receiving the PDOs, but by a software timer. In the callback, the output Process Data Objects are copied safely from the `process_data_buf` buffer (see `utilities::copy_process_data_buffer_to_buf()`) to a private buffer, then they are formatted and published to the `/pdo_out_timer` topic in ROS.

`Services`: This is a complementary source file to the project, which acts as a placeholder for all the services used. For now, there's only one service used, namely `EthercatCommunicatord`, which is a daemon for starting, stopping and restarting the `EthercatCommunicator`. When there is a start command from the user, this service calls the `EthercatCommunicator::start()` method, which eventually starts the real-time pthread with the `EthercatCommunicator::run()` method. Initially, the `PDOOutListener` was not implemented as a

subscriber multiplexing different types of variables, yet was split into different services for the different types of variables. This however changed, since services, as message communication means, have latency and cannot acheive high data throughput. Therefore the `EthercatCommunicatord` service is the only one remaining in the file.

`Utilities`: This is a source file with utility functions, which are needed from core methods and functions of the project. The most used function under `utilities` namespace is `utilities::copy_process_data_buffer_to_buf()`, which is presented and discussed below:

`utilities::copy_process_data_buffer_to_buf()`

```
1  void copy_process_data_buffer_to_buf(uint8_t * buffer)
2  {
3     pthread_spin_lock(&lock);
4     for (int i = 0; i < master_info.slave_count; i++)
5     {
6        memcpy((buffer + ethercat_slaves[i].slave.get_pdo_out()),
7               (process_data_buf + ethercat_slaves[i].slave.get_pdo_out()),
8               (size_t)(ethercat_slaves[i].slave.get_pdo_in() -
                   ethercat_slaves[i].slave.get_pdo_out())
9           );
10    }
11    /*
12    buffer + ethercat_slaves[i].slave.get_pdo_out()) ----> the starting
         address of the slave's output pdos in the buffer
13
14    process_data_buf + ethercat_slaves[i].slave.get_pdo_out()) ----> the
         starting address of the slave's output pdos in the process_data_buf
15
16    (size_t)(ethercat_slaves[i].slave.get_pdo_in() -
         ethercat_slaves[i].slave.get_pdo_out() ----> size of output pdos
         of the slave
17
18    */
19    pthread_spin_unlock(&lock);
20 }
```

Listing 5.12: *The* `utilities::copy_process_data_buffer_to_buf` *function.*

This function is finally presented after many previous references. It starts with locking the critical section for copying the data from one buffer to the other, by calling `pthread_spin_lock()` on the `lock` global variable. Then the output Process Data Objects are copied from the `process_data_buf` buffer to the buffer given as parameter in the function call. Finally the function before exiting calls `pthread_spin_unlock()` to unlock the `lock` variable, since it exitted the critical section.

5.2 Installation Process

In this section, the process of installing the environment needed for deploying the aforementioned software, is described.

5.2.1 The Preempt_RT Patch

The first step to set up the environment, is to patch the Linux kernel to be used, with the PREEMPT_RT patch. For now, the latest supported version of Linux kernel for deploying the environment with all the necessary software, is 4.9. For some reason, EtherLab cannot be built with kernel versions higher than 4.9. In the following procedure, Ubuntu 16.04 is used as the host GNU/Linux distribution. If there is a different choice of GNU/Linux distribution or Ubuntu version, the following steps can be easily translated and implemented in that too.

The following steps are based on a post in the Ubuntu fora[2]:

5.2.1.1 Step 0 - Making a working directory

First, a working directory should be created, e.g. `/˜kernel`:

```
1   # Make working directory
2 $ mkdir ~/kernel && cd ~/kernel
```

[2] `https://ubuntuforums.org/showthread.php?t=2273355`

5.2.1.2 Step 1 - Downloading the Linux kernel and the patch

In the rt project in the Linux kernel archives[3] the 4.9 RT patch can be found and be downloaded. Up to now, the most recent RT patch for kernel 4.9 is `patch-4.9.146-rt125.patch.gz`. This can be downloaded with the following command:

```
wget https://www.kernel.org/pub/linux/kernel/projects/rt/4.9/patch-4.9-
.146-rt125.patch.gz
```

Then, in the Linux kernel archives[4], the kernel that matches the above patch can be found and be downloaded:

```
wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.9.146.tar.gz
```

5.2.1.3 Step 2 - Unzipping the kernel

Next, the compressed archive containing the kernel needs to be unzipped. This is done with the following command:

```
1   # x - extract
2   # z - pipe through gunzip
3   # v - verbose (text output)
4   # f - from file
5  $ tar -xzvf linux-4.9.146.tar.gz
```

Listing 5.13: *Command for unzipping the kernel compressed archive.*

5.2.1.4 Step 3 - Patching the kernel

Then, the kernel needs to be patched. The code steps are shown below:

```
1   # Move to kernel source directory
2  $ cd linux-4.9.146
3
4   # c - pipe file contents to stdout
5   # d - decompress
6  $ gzip -cd ../patch-4.9.146-rt125.patch.gz | patch -p1 --verbose
```

Listing 5.14: *Commands for patching the kernel.*

[3] https://www.kernel.org/pub/linux/kernel/projects/rt/
[4] https://www.kernel.org/pub/linux/kernel/

5.2.1.5 Step 4 - Enabling Real-time attributes

In order to build the kernel the `libncurses-dev` package should be installed:

```
$ sudo apt-get install libncurses-dev build-essential libssl-dev git
    bison flex libelf-dev
```

The next step should create a graphical menu in the terminal which can be scrolled through.

```
$ make menuconfig
```

This line will create a menu like the one pictured in Figure 5.1. In this figure, the option of PREEMPT_RT is selected.

Figure 5.1: *The PREEMPT_RT kernel configuration option using* `menuconfig`*.*

In the graphical menu of `make menuconfig`, extra configuration parameters can be specified, derived from [63, Chapter 3], namely:

```

# y = YES & n = NO

# see below for a detailed description on how to enable this
    configuration. Also see the above figure.
CONFIG_PREEMPT_RT_FULL=y

CONFIG_CPU_FREQ=n
```

```
8
9  CONFIG_CPU_IDLE=n
10
11 CONFIG_NO_HZ_FULL=y # see Configuration section. You might not need this
     configuration after all, if the application is multithreaded.
12
13 CONFIG_RCU_NOCB_CPU=y # see section of Configuration. You might not need
     this configuration after all, if the application is multithreaded.
```

Listing 5.15: *The configuration options for building the kernel with PREEMPT_RT patch.*

The button presses are enclosed in []s, except for menu scrolling with the up and down arrows. Comments are preceeded by # and are ment for clarification.

Tip: While in the menuconfig, one can type "/" and then search the place of a configuration parameter. One can exit with `[ESC]`.

```
1  ##Graphical Menu##
2
3  Processor type and features ---> [Enter]
4  Preemption Model (Voluntary Kernel Preemption (Desktop)) [Enter]
5  Fully Preemptible Kernel (RT) [Enter] #Select
6
7  [Esc][Esc]
8
9  Kernel hacking --> [Enter]
10 Memory Debugging [Enter]
11 Check for stack overflows #Already deselected - do not select
12
13 [Esc][Esc]
14
15 [Right Arrow][Right Arrow]
16
17 <Save> [Enter]
18
19 .config
20
21 <Okay> [Enter]
```

```
22
23  <Exit> [Enter]
24
25  [Esc][Esc]
26
27  [Right Arrow]
28  <Exit> [Enter]
```

Listing 5.16: *Steps for building the kernel with the PREEMPT_RT patch.*

5.2.1.6 Step 5 - Compiling the kernel

This step takes more than 5 minutes in a typical workstation (i5/i7 CPUs).

```
1 $ make -j4
```

The `-j4` referes to the number of jobs to be spawned for parallel processing. If there is only a single core processor, this option can be omitted. It's common practice to use the number of cores after `-j`. There isn't a proof that this is the best option. If little is known about the processors, `lscpu` can be used in a terminal to determine the number of cores.

5.2.1.7 Step 6 - Making modules & installing

This step will take some time, but not as long as the previous step:

```
1 $ sudo make modules_install -j4
2 $ sudo make install -j4
```

5.2.1.8 Step 7 - Verifying and updating

One could verify that `initrd.img-4.9.146-rt125`, `vmlinuz-4.9.146-rt125`, and `config-4.9.146-rt125` exist. They should have been created in the previous step:

```
1 $ cd /boot
2 $ ls
```

Then grub should be updated; this will allow the selection of the new kernel on bootup.

```
1 $ sudo update-grub
```

It should be verified that there exists a menu entry containing the text "menuentry 'Ubuntu, with Linux 4.9.146-rt125'". One can replace `vim` with `gedit` or any other text editor of her choice, however this file should not be edited.

```
1 $ vim /boot/grub/grub.cfg
```

If one would like to make this kernel the new default (optional), this can be done by editing the `/etc/default/grub` file. More information can be found in the Ubuntu Help page[5].

5.2.1.9 Step 8 - Rebooting

Then, the PC should be rebooted and when the grub menu appears during boot, the newly created RT kernel should be selected.

```
1 $ sudo reboot
```

Once rebooted, one can verify that everything was successful by running:

```
1 $ uname -a
```

The output should like the one below:

```
1 Linux pc_name 4.9.146-rt125 #1 SMP PREEMPT RT ...
```

5.2.2 EtherLab

After building the PREEMPT_RT Linux kernel and selecting it during boot, the EtherLab kernel module should be installed. The procedure is the following: After cloning the repository of the project, `cd` into the `etherlab` directory of the project and run the `install_etherlab_patched.sh` script. This script is specifically written for automatic installation of EtherLab and is presented below:

```
1  # install the necessary packages for building EtherLab
2  sudo apt-get install autoconf automake libtool mercurial
3
4  # hg clone might fail because there is no user registered. In this case
       uncomment and run the following line:
```

[5] https://help.ubuntu.com/community/Grub2/Setup

```
5  # echo -e '[extensions] \n mq = \n [ui] \n username = Foo Bar
       <foobar@mail.com>' > ~/.hgrc
6
7  # clone the EtherLab repository
8  hg clone -u 33b922ec1871 http://hg.code.sf.net/p/etherlabmaster/code
       ethercat-1.5.2-merc
9
10 # clone the patches
11 hg clone http://hg.code.sf.net/u/uecasm/etherlab-patches
       ethercat-1.5.2-merc/.hg/patches
12 cd ethercat-1.5.2-merc
13
14 # apply the patches
15 hg qpush -a
16 cd ..; make ethercatMasterInstallWithAutoStart
17 rm -rf ethercat-1.5.2-merc
```

Listing 5.17: *The `install_etherlab_patched.sh` script.*

After patching the EtherLab source code, in line 16 a `make` command is issued with `ethercatMasterInstallWithAutoStart` argument. The *Makefile* corresponding to the above `make` command is based in a *Makefile* available from [104] and has been modified for the project's needs. It is presented below:

```
1  SHELL := /bin/bash
2  ethercatMasterVersion:=1.5.2-merc_unofficial_patch
3  # ethercatMasterZip:=ethercat-$(ethercatMasterVersion).tar.bz2
4  ethercatMasterDirectory:=ethercat-1.5.2-merc
5  udevRulesFile:=99-EtherCAT.rules
6  ethercatUserGroup:=$(shell whoami)
7
8  $(udevRulesFile):
9    @echo "Generating udev rules file"
10   @echo "KERNEL==\"EtherCAT[0-9]*\", MODE=\"0664\",
       GROUP=\"$(ethercatUserGroup)\"">$(udevRulesFile)
11
12 ethercatMaster: $(udevRulesFile)
```

```
13   # tar -xvf $(ethercatMasterZip)
14   cd $(ethercatMasterDirectory);\
15   ./bootstrap;\
16   ./configure --enable-generic --disable-8139too --enable-e1000e
         --with-e1000e-kernel=4.9 --enable-hrtimer --enable-cycles;\
17   make all modules;
18
19 ethercatMasterInstall: ethercatMaster
20   cd $(ethercatMasterDirectory);\
21   sudo make modules_install install;\
22   sudo depmod;\
23   sudo mv ../$(udevRulesFile) /etc/udev/rules.d/$(udevRulesFile);\
24   sudo ln -s /opt/etherlab/etc/init.d/ethercat /etc/init.d/ethercat;\
25   sudo mkdir -p /etc/sysconfig/;\
26   sudo cp /opt/etherlab/etc/sysconfig/ethercat /etc/sysconfig/ethercat;\
27   sudo sed -i 's/DEVICE_MODULES=\"\"/DEVICE_MODULES=\"generic\"/g'
         /etc/sysconfig/ethercat;\
28   sudo ln -s /opt/etherlab/bin/ethercat /usr/bin/ethercat;\
29   interfaces=`ifconfig | grep -e "^e[tn][a-z0-9]*" -o`;\
30   for i in $$interfaces;do lastInterface=$$i; done;\
31   interfaceMAC=`ifconfig $$lastInterface | ...
32   grep "[0-9A-Fa-f]\{2\}:[0-9A-Fa-f]\{2\}:[0-9A-Fa-f]\{2\}:
33   [0-9A-Fa-f]\{2\}:[0-9A-Fa-f]\{2\}:[0-9A-Fa-f]\{2\}" -o`;\
34   sudo sed -i
         "s/MASTER0_DEVICE=\"\"/MASTER0_DEVICE=\"$$interfaceMAC\"/g"
         /etc/sysconfig/ethercat;\
35
36 ethercatMasterInstallWithAutoStart: ethercatMasterInstall
37   sudo update-rc.d ethercat defaults;\
38   sudo /etc/init.d/ethercat start;
39
40 clean:
41   @echo Removing compiled installation files
42   @rm -f -r $(ethercatMasterDirectory) $(udevRulesFile)
```

Listing 5.18: *The `Makefile` for building EtherLab.*

It should be noted that in order to build EtherLab, root permissions are needed. With this *Makefile*, EtherLab is built with both native and generic driver options. If a different native driver from *e1000e* is used, the line 17 in the Makefile should be changed and updated with the correct configuration option and the correct driver version. In an EtherLab's web page[6] and in [2, Chapter 9], the supported hardware and the options the command `configure` takes, are shown respectively. If the hardware at hand is not supported, then the configuration options related to the native drivers should be removed and EtherLab should be built only with the generic driver option.

5.3 Configuration & Optimization

Note: This section is largely based on [63].

In this section, the steps for configuring and optimizing the aforementioned installed system are presented, in order to meet the real-time requirements described in Section 4.1.

5.3.1 Isolating the Application

If there is a need for real-time performance on *single-core* systems it is necessary to adapt the entire system, e.g. using the PREEMPT_RT patch or an RTOS [63, Chapter 2]. This is not always necessary in a *multi-core* system [63, Chapter 2]. Recently added features in the Linux kernel make it possible to aggressively migrate sources of kernel-introduced jitter away from selected CPUs [63, Chapter 2]. This provides bare-metal-like performance on the CPUs where sources of jitter have been removed, thus creating a real-time environment for an application running in Linux user-space [63, Chapter 2].

On a default setup, this is not possible since the Linux kernel needs to do some regular housekeeping [63, Chapter 2]. It is possible to move much of this housekeeping to some dedicated CPUs, provided there is a multicore system [63, Chapter 2]. That leaves the other CPUs relatively untouched by the Linux kernel, unless a user-space task triggers some kernel activity [63, Chapter 2]. The application that executes in this bare-metal environment should avoid using libc calls and Linux system calls [63, Chapter 2].

When using this method correctly, it is possible to enhance throughput and real-time performance by reducing the overhead of interrupt handling [63, Chapter 2]. This is beneficial e.g. for applications that require very high throughput, and for device drivers that handle fre-

[6] `http://www.etherlab.org/en/ethercat/hardware.php`

quent interrupts, such as 10Gb Ethernet drivers [63, Chapter 2]. The basic approach followed is descibed briefly here; For elaborate information on the matter, the reader is referred to the excellent guide in [63, Chapters 2 and 3]. The first step for CPU isolation in Linux, is to define different *cgroups* (non real-time and real-time) in the *cpuset* cgroup [63, Chapter 3].

Note: After this definition, the two distinct cgroups need to be associated with a *NUMA* node, even if the memory architecture of the system isn't NUMA-enabled.

Load balancing, i.e. task migration, is a default activity in the Linux kernel that introduces non-deterministic jitter. It is therefore necessary to disable load balancing in the real-time cpuset. This also means that it is necessary to specify the correct affinity for the threads that should execute within the real-time CPUs. Next, the general purpose tasks are moved to the general non real-time partition, however this is not possible for every task, since some tasks need to execute on all available CPUs. All future child tasks that are created from the non real-time partition will also be placed in this partition.

After the general purpose tasks are migrated, it is the interrupts' turn. Some interrupts are not CPU-bound. Unwanted interrupts introduce jitter and can have serious negative impact on real-time performance. They should be handled on the general purpose CPUs whenever possible. The affinity of these interrupts can be controlled using the `/proc` file system. Typical interrupts that should be moved are: timer interrupts, network related interrupts and serial interface interrupts.

On the other hand, if there are any interrupts that are part of the real-time application, they should be configured to fire in the real-time partition. Regarding the network interrupts, Linux can route the packets on different CPUs in an SMP system and the handling can create timers on the specific CPUs, for example the ARP timer management, based on `neigh_-timer`. There are a couple of solutions that can be adopted to minimize the effect of rerouting packets on different CPUs, like migrating all the timers on the non-realtime partition if possible or specifying the affinity of network queues on some architectures. The developed application needs the packets from the EtherCAT network to be received only in the real-time partition thus the affinity of the network queues (for the XPS and RPS algorithms, see also Subsubsection 4.2.2.5) should be set to the CPUs related to the real-time partition.

Finally, the pid of the application which will run in a real-time context, should be moved to the real-time partition and also pinned to a specific CPU (if there are many CPUs in the real-time partition). This has been described briefly in `pthread_setaffinity_np()` function

call. In this manner, the application will be isolated (even from general purpose interrupts) and pinned to a specific CPU, thus achieving real-time and bare-to-metal performance.

Note: If EtherLab is configured and run with the native driver option, then as previously described in Subsubsection 4.2.2.5), there is no need to set affinity of the network queues, since the native EtherCAT-capable driver is accessed without traversing the Linux Network Stack first. However, if EtherLab is configured and run with the generic driver option, then this affinity should be set.

Note: There is also a kernel boot parameter that achieves isolation of CPUs, useful for isolating the real-time domain from load balancing at system start-up: `isolcpus=1,2,3,4,...`.

Note: Of course the developed application can not be pinned to a specific CPU belonging to the real-time partition, if the application is not in the real-time cgroup. Thus, the application needs to be moved to the real-time cgroup prior to pinning it on a specific CPU.

5.3.2 Full Dynamic Ticks

Ticks are used to balance CPU execution time between several tasks running on the same CPU [63, Chapter 2]. They are interrupts generated by a hardware timer and occur at regular intervals determined by the `CONFIG_HZ` kernel configuration, which for most architectures can be configured when compiling the kernel [63, Chapter 2]. The tick interrupt is a per-CPU interrupt [63, Chapter 2].

The full dynamic ticks (`CONFIG_NO_HZ_FULL` kernel configuration) adaptively try to shutdown the tick whenever possible, even when the CPU is running tasks [63, Chapter 3]. To achieve full dynamic ticks on a CPU, the application running on this CPU must comply to some requirements [63, Chapter 3]. First, only one thread should run on each CPU [63, Chapter 3]. The application should not use any POSIX timers, directly or indirectly [63, Chapter 3]. This usually excludes any kernel calls that will access the network, but also excludes a number of other kernel calls [63, Chapter 3]. Keeping the kernel calls to a minimum will maximize the likelihood of achieving full dynamic ticks [63, Chapter 3]. *Since the developed application is a multithreaded one, full dynamic ticks option is not encouraged.*

To enable full dynamic ticks to specific CPUs (the kernel configuration must be enabled), the following boot parameters should be used: `nohz_full=1,2,3,4,...` [63, Chapter 3].

5.3.3 Optimizing the Partitioned System

If the above subsections do not offer enough real-time properties, then this subsection provides some more hints for optimization.

5.3.3.1 Optimizing Power Saving

Power saving can be handled in Linux with various techniques. Here two of them are briefly described:

- **Dynamic Frequency Scaling:** When little CPU-bound work is performed, the CPU frequency can be reduced as a way to reduce power consumption [63, Chapter 2]. This is called *dynamic frequency scaling* [63, Chapter 2]. This option is enabled at compile time by the configuration parameter `CONFIG_CPU_FREQ` [63, Chapter 2]. If enabled, the system will include functionality, called a *governor*, for controlling the frequency [63, Chapter 2]. There are several governors optimized for different types of systems [63, Chapter 2]. The decision to use dynamic frequency scaling in a real-time system depends on the time that is needed to increase the frequency and that time's relation to the latency requirements [63, Chapter 2].

- **CPU Power States:** When the CPU is idle (i.e. no tasks are ready to run on this CPU) the CPU can be put in *sleep state* (C state) [63, Chapter 2]. A sleep state means that the CPU does not do any execution, while still ready to respond on certain events, e.g. an external interrupt [63, Chapter 2]. CPUs usually have a range of power modes [63, Chapter 2]. Deeper sleep means lower power consumption at the price of increased wake-up time [63, Chapter 2]. As with dynamic frequency scaling, the transition between the power states is controlled by a governor [63, Chapter 2]. To configure the kernel to enter sleeping state when idle, the compile-time configuration parameter `CONFIG_CPU_IDLE` is used [63, Chapter 2].

Power saving techniques interact poorly with real-time requirements [63, Chapter 2]. The reason is that exiting a power saving state cannot be done instantly, e.g. $200\mu\ s$ wake-up latency from sleep mode C3 and $3\mu\ s$ from C1 on an Intel i5 - 2GHz [63, Chapter 2]. This may not be a problem in e.g. a soft real-time system where the accepted latency is longer than the wake-up time or in a multicore system where power saving techniques may be used in a subset of the cores [63, Chapter 2]. However, it is recommended the power saving mechanisms to be disabled on system start-up, using the following kernel configuration parameters [63,

Chapter 2]:

- Disabled frequency scaling by setting `CONFIG_CPU_FREQ=n`.
- Disabled transitions to low-power states by setting `CONFIG_CPU_IDLE=n`.

5.3.3.2 Disabling power management

The CPU frequency governor causes jitter because it is periodically monitoring the CPUs [63, Chapter 3]. The actual activity of changing the frequency can also have a serious impact [63, Chapter 3]. The frequency governor is disabled, as described previously, with the following configuration: `CONFIG_CPU_FREQ=n [63, Chapter 3]`.

However, an alternative is, at runtime, to change the governor policy (of a specific real-time CPU) to `performance`. The advantage in this approach, is that each CPU can have different power policy [63, Chapter 3]. Yet, it should be noted that this could damage the hardware because of overheating and research should be conducted as to what works for the specific hardware [63, Chapter 3].

5.3.3.3 Optimizing Real-Time Throttling

If only real-time tasks were runnable on a CPU, they would consume all CPU power if the scheduling principles were followed [63, Chapter 2]. Sometimes that is the wanted behavior, but it would also allow that bugs in real-time threads completely block the system [63, Chapter 2]. To prevent this from happening, the *real-time throttling* mechanism makes it possible to limit the amount of CPU power that the real-time threads can consume [63, Chapter 2].

The mechanism is controlled by two parameters: *rt_period* and *rt_runtime* [63, Chapter 2]. The total execution time for all real-time threads cannot exceed *rt_runtime* during each *rt_period* [63, Chapter 2]. As a special case, *rt_runtime* can be set to -1 to disable the real-time throttling [63, Chapter 2].

The throttling mechanism allows the real-time tasks to consume *rt_runtime* times the number of CPUs for every *rt_period* of elapsed time [63, Chapter 2]. Consequently, a real-time task can utilize 100% of a single CPU as long as the total utilization does not exceed the limit. The default settings *rt_period=1000000 μs* ($1s$) and *rt_runtime=950000 μs* ($0.95s$) give a limit of 95% CPU utilization [63, Chapter 2]. The parameters are associated with two files in the `/proc` file system [63, Chapter 2]:

- `/proc/sys/kernel/sched_rt_period_us`
- `/proc/sys/kernel/sched_rt_runtime_us`

In the generic case, execution of the real-time tasks may be blocked for a time equal to the difference between *rt_runtime* and *rt_period* [63, Chapter 3]. This situation should however be quite rare since it requires that there are real-time tasks (i.e. tasks scheduled with real-time policies) that are ready to run on all CPUs. This condition should be rare since real-time systems are typically designed to have an average real-time load of significantly less than 100% [63, Chapter 3]. Consequently, it is recommended to keep the real-time throttling enabled [63, Chapter 3]. For systems that do not have any real-time tasks, the real-time throttling will never be activated and the settings will not have any impact [63, Chapter 3]. An alternative when using CPU isolation is to avoid using real-time classes, since the CPU is supposed to run a single task anyway [63, Chapter 3]. In this case, real-time throttling should not be activated [63, Chapter 3].

Note: If the system is configured with `CONFIG_NO_HZ_FULL` and a real-time process executes on a `CONFIG_NO_HZ_FULL` CPU, real-time throttling will cause the kernel to schedule extra ticks [63, Chapter 3].

5.3.3.4 Time Stamp Counter (tsc timer - x86 only)

The time stamp counter is a per-CPU counter that produces time stamps [63, Chapter 3]. Since the counters may drift, Linux will periodically check that they are synchronized [63, Chapter 3]. But this periodicity means that the tick might appear despite using full dynamic ticks [63, Chapter 3]. By telling Linux that the counters are reliable, Linux will no longer perform the periodic synchronization [63, Chapter 3]. The side effect of this is that the counters may start to drift, something that can be visible in trace logs for example [63, Chapter 3]. The boot parameter for making the tsc timers reliable is: `tsc=reliable` [63, Chapter 3].

5.3.3.5 Delay vmstat timer

This timer is used for collecting virtual memory statistics [63, Chapter 3]. The statistics are updated at an interval specified as seconds in `/proc/sys/vm/stat_interval` [63, Chapter 3]. The amount of jitter can be reduced by writing a large value to this file [63, Chapter 3].

5.3.3.6 Machine check - x86 only

The x86 architecture has a periodic check for corrected machine check errors (MCE) [63, Chapter 3]. The periodic machine check requires a timer that causes unwanted jitter [63, Chapter 3]. The periodic check should be turned off on the real-time CPUs [63, Chapter 3].

5.3.3.7 Disabling the watchdog timer

The watchdog timer is used to detect and recover from software faults and requires a regular timer interrupt [63, Chapter 3]. This interrupt is a jitter source that can be removed, at the obvious cost of less error detections [63, Chapter 3].

5.3.3.8 Disabling the NMI Watchdog - x86 only

The NMIs are hardware interrupts which are fired when there are non-recoverable hardware errors. Thus, the debugging feature for catching hardware hangings and cause a kernel panic (the NMI Watchdog) can also be disabled [63, Chapter 3]. On some systems it can generate a lot of interrupts, causing a noticeable increase in power usage [63, Chapter 3].

5.3.3.9 Memory Overcommit

By default, the Linux kernel allows applications to allocate (but not use) more memory than is actually available in the system [63, Chapter 2]. This feature is called *memory overcommit* [63, Chapter 2]. The idea is to provide a more efficient memory usage since processes typically ask for more memory than they will actually need [63, Chapter 2]. However, overcommitting also means there is a risk that processes will try to utilize more memory than there is available [63, Chapter 2]. If this happens, the kernel invokes the Out-Of-Memory Killer (OOM killer) [63, Chapter 2]. The OOM killer scans through the tasklist and selects a task to kill to reclaim memory, based on a set of heuristics [63, Chapter 2].

When an out-of-memory situation occurs, the whole system may become unresponsive for a significant amount of time, or even end up in a deadlock [63, Chapter 2]. Thus, for embedded and real-time critical systems, the allocation policy should be changed so that memory overcommit is not allowed [63, Chapter 2]. In this mode, `malloc()` will fail if an application tries to allocate more memory than is strictly available, and the OOM killer is avoided [63, Chapter 2]. More information on the matter can be found in the man page for proc (5)[7] and Linux kernel supported overcommit handling modes.

[7] `http://man7.org/linux/man-pages/man5/proc.5.html`

6 Experimental Evaluation

> If something can go wrong, it will go wrong.
>
> *Murphy's General Law*

In this chapter, the evaluation process and the experimental results are presented. The tools, methodology, and environment pertaining to the evaluation process are described. The chapter concludes with the presentation of experimental results along with a brief description.

6.1 Tools, Methodology & Environment

In this section the tools and methodology employed for building and launching the developed application are described. Next, the configured and monitoring environment are presented. The setup described in this chapter was configured with the following tools:

- A PC/104 computer by RTD Embedded Technologies, Inc[1], see Figure 6.1.
- Ubuntu 16.04 with kernel 4.9.115-rt93 patched with PREEMPT_RT (described in Chapter 5).
- IgH Master (EtherLab) version 1.5.2.
- Intel Network Interface Controller 82574L with `e1000e` driver.

[1] `https://www.rtdusa.com/home.htm`

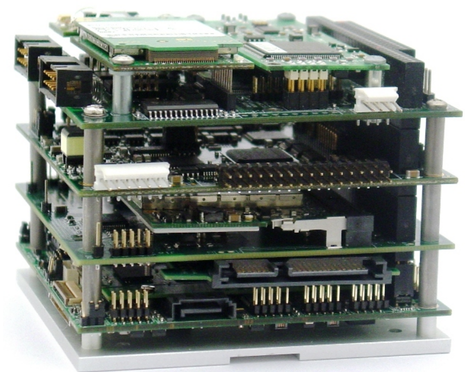


Figure 6.1: *The PC/104 computer.*

6.1.1 Building the application

After building the environment (described in Chapter 5) as a placeholder for the application to run, the next step is to build the application and launch it in this environment.

The application developed is wrapped into a ROS package. Therefore the only thing required for building the application is to run:

```
1 $ catkin_make
```

This is a command for issuing the build of the ROS packages residing in the ROS workspace.

After successfully building the application, there is an extra step before launching it. Since the developed application needs to communicate with EtherLab (a kernel module), it needs *root* privileges. For this reason, the following script was written, which should be executed after building the application:

```
1 #!/bin/bash
2 cd ~/catkin_ws/devel/lib/ether_ros
3 chown root:root ether_ros
4 chmod a+rx ether_ros
5 chmod u+s ether_ros
```

Listing 6.1: *The* `change_permissions_ether_ros.sh` *script.*

6.1.2 Starting the EtherLab module

Before launching the application, EtherLab should be properly configured and initialized. If EtherLab was built with the script presented in Chapter 5, then probably EtherLab has started already (the script builds it with the `ethercatMasterInstallWithAutoStart` option, which auto-starts EtherLab on system start). However, in case there exist many NICs in a system, it should be defined with which NIC should EtherLab communicate. This is done with a sysconfig file (located in `etc/sysconfig/ethercat`), which is read when EtherLab is initialized and described in [2, Chapter 7]. The `MASTER0_DEVICE` parameter should be filled with the corresponding NIC. The corresponding MAC address can be easily found with the `ifconfig` command in a terminal.

Important note: The Ethernet driver modules for EtherCAT operation should be defined in the sysconfig file also. In case the EtherLab module is intended to be used with the native driver option, then the `DEVICE_MODULES` parameter must be filled with the corresponding EtherCAT native driver's name (e.g. `e100`, `e1000`, `e1000e`, `igb`). If the generic driver option is used, then the parameter should be filled with `generic`. However, since the application will use the Linux Network Stack, in the generic case, before (re)starting EtherLab, the following script should be run as a further optimization:

```
1 #!/bin/bash
2 rmmod e1000e
3 modprobe e1000e InterruptThrottleRate=0 RxIntDelay=0 TxIntDelay=0
```

Listing 6.2: *The `reinstall_e1000e_wo_throttling.sh` script.*

This script basically reinstalls the `e1000e` (used in this setup) but with some optimizations applied. The most basic is the parameter `InterruptThrottleRate` set to 0, with which the driver places no limit to the amount of interrupts per second, the adapter will generate for incoming packets. More information can be found in [117]. This is not necessary for the native option, since the default driver won't be used by EtherLab.

After this configuration, EtherLab can be initialized. This is done with the following command:

```
1 $ sudo /etc/init.d/ethercat restart # other options are: start, stop and
    more
```

Important note: After EtherLab initialization has finished, the optimizations shown in Section 5.3 should be applied. It's critical to apply these optimizations after (re)installing the Ethernet driver to be used, since the priority of the IRQ thread regarding the Ethernet driver's ISR should be changed after (re)installing the Ethernet driver.

6.1.3 Slaves Initialization

Before launching the application it is necessary to initialize the EtherCAT slaves to the correct configuration. The slaves of Laelaps II are initialized by placing manually all the legs in the position depicted in Figure 4.8 and by pressing the Reset button (shown in Figure 6.2) of every Delfino Launchpad [17].

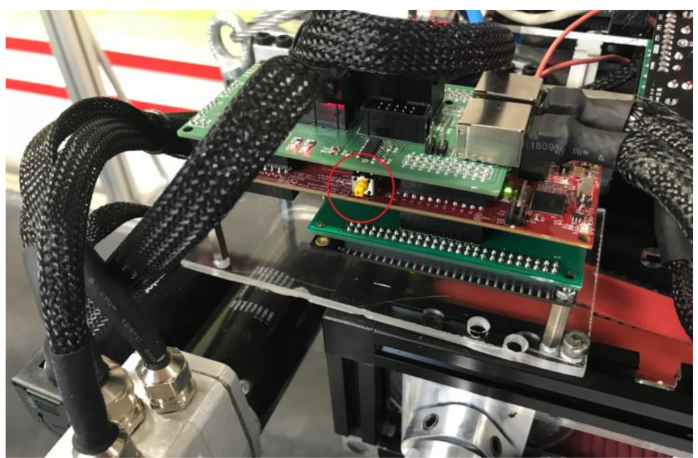


Figure 6.2: *Reset button to initialize legs' pose [17].*

Before continuing to the next step, all wires, drivers and extension boards should be checked and be properly mounted on the quadruped robot [17]. The current EtherCAT application (as developed in [17]) comprises of two states: the *Operational* state and the *Configurational* state [17]. In the Operational state, the Output variables are processed by the slaves and the Input variables are returned to the master, while in the Configurational state, the Output variables are not processed by the slaves, thus the Input variables returned should be disregarded [17]. Therefore prior to performing any experiments on Laelaps II, the EtherCAT variable *State Machine*[2] should be set to Configurational State (0) for each slave, before enabling the High Voltage Power Supply [17]. Figure 6.3 illustrates the experimental setup of Laelaps II on the treadmill, ready to perform the desired task [17].

The *State Machine* diagram of Laelaps is illustrated in Figure 6.4. When parameters to all EtherCAT Output variables have been set and the *State Machine* is switched to Operational

[2]`bool State_Machine`

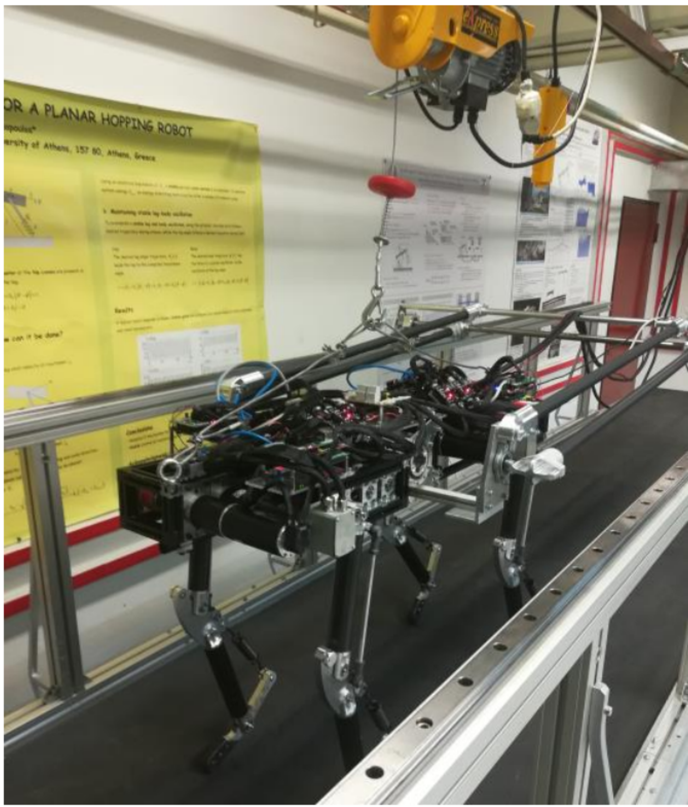


Figure 6.3: *Laelaps II on treadmill ready to perform experiments [17].*

State (1), Laelaps executes the desired movement.

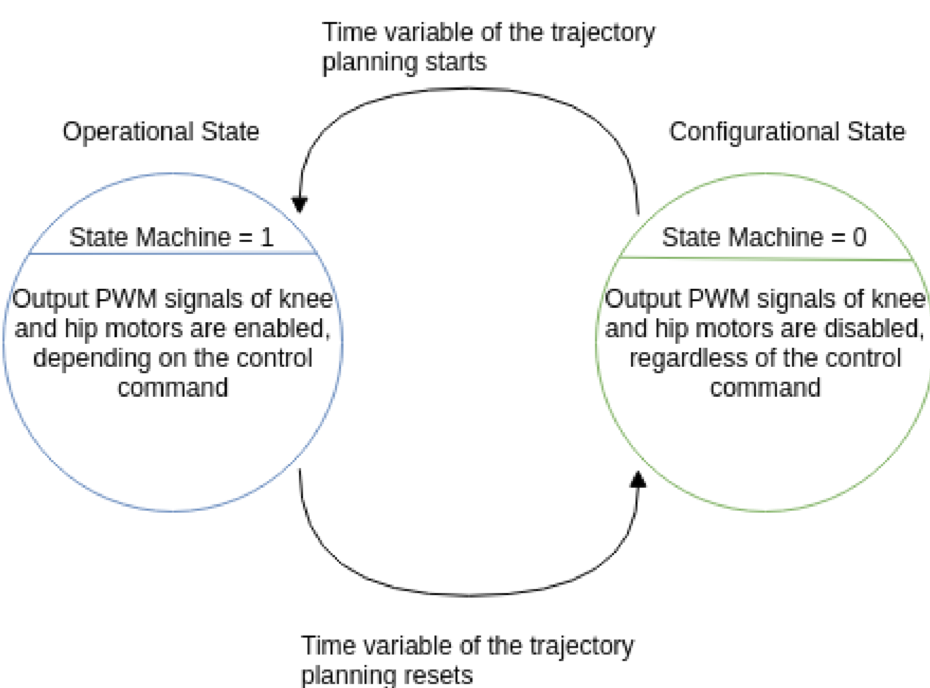


Figure 6.4: *Laelaps' State Machine [17].*

6.1.4 Launching the application

After starting the EtherLab module and initializing the EtherCAT slaves, the application is launched. This is done by running the following command in a terminal:

```
1 $ roslaunch ether_ros ether_ros.launch
```

The launch file used above simply creates a ROS node from the developed application and launches it in the ROS environment. The Operator should run the python file `ethercat_keyboard_controller.py` by executing:

```
1 $ rosrun ether_ros ethercat_keyboard_controller.py
```

This python file creates a custom Command Line Interface (CLI) which gives functionality to the user, as described in Subsubsection 4.2.2.2. With this, the user can send commands to the EtherCAT network. The Output variables are summarized in Subsection 4.2.3, but also can be viewed using the following command:

```
1   $ ethercat pdos
```

Before sending the "start" command (and after launching the ROS node), the user should run the following script:

```
1 #!/bin/bash
2 ether_ros_pid=$(ps -e | grep "ether_ros" | grep -o -E '[0-9]+' | head -n
     1)
3 sudo echo $ether_ros_pid > /sys/fs/cgroup/cpuset/rt/tasks
```

Listing 6.3: *The `make_rt_task_ether_ros.sh` script.*

This script will make sure that the *pid* of the ROS node process will be on the pids of the real-time cgroup, as created in Section 5.3. In this way, the application can request to be pinned on a specific CPU (namely 3), as shown in `pthread_setaffinity_np()` function call, and the request will be satisfied, since it belongs to the real-time cgroup. Otherwise, the request will not be accepted and an error code will be returned.

By completing the aforementioned steps, it's time for the user to give the "start" command. This is done in the custom terminal with: `!start`.

6.1.5 Monitoring

After launching the application and sending the "start" command, the real-time communication begins. However, somehow the process should be monitored, in order for the user to stay informed with the current situation of the overall system. The following steps aim to this direction.

6.1.5.1 Ring Buffer

The messages from the kernel need somehow to be monitored. EtherLab logs information regarding skipped packets in the kernel logs, therefore if some packets are skipped, they can be monitored with these logs. The kernel messages are written into a ring buffer. The contents of the ring buffer can be monitored through the `dmesg` command. Another useful command is the following:

```
1 $ tail -f /var/log/kern.log
```

The `tail` command reveals only the latest messages, and the `kern.log` contains only the kernel's messages of any log level.

6.1.5.2 rqt

The *rqt*[3] is a useful tool in the ROS environment for monitoring the status of the ROS ecosystem. With this tool, topics and nodes along with their connections can be observed easily. Lastly, one useful plugin to visualize online data from multiple topics, is the *rqt_multiplot*[4].

6.2 Experiments & Results

This section presents the results of experiments with Laelaps II and the developed real-time application in low frequency.

Since successful experiments have been conducted with a similar system (the only difference is the EtherCAT master used) [17], the experiments carried out in this section, correspond to the parameters shown in [17].

[3]http://wiki.ros.org/rqt
[4]http://wiki.ros.org/rqt_multiplot

6.2.1 Experiments

In the context of this thesis, two experiments have been conducted. The first is related to the trotting ability of Laelaps II. Its objective is to evaluate the ability of the legs to synchronize properly and achieve the desired trotting movement. The second is related to the control loop frequencies of the EtherCAT network. Its objective is to compare the two types of EtherLab drivers and measure their highest achieved EtherCAT control loop frequency.

6.2.1.1 Trotting Experiment

In this experiment, the developed application provides the parameters of the desired elliptical trajectory for the toe of each leg along with other parameters of the system. The data are logged using `rosbag` and post processed and plotted using a Matlab script. It is worth mentioning that a PIV (Proportional – Integral – Velocity) controller is implemented in each slave (more information in [17]), thus the master does not affect the control algorithms running in the slaves, merely supplies each slave with the necessary parameters via EtherCAT.

For this experiment, a table describing the parameters used is provided along with figures. The figures present:

- The desired elliptical trajectory of all toes (red) along with their actual response (black) w.r.t coordinate systems located at the hip joints of the legs.
- The desired response of both knee and hip angles (red) of every leg with their respective actual response of each knee and hip joint (black).
- The PWM commands of each knee and hip motor (black) which is the output of the PIV controllers with their respective predefined PWM limits (red). These values represent the continuous current limits of both motors. More information on the selected limits can be found in [17].
- The velocity estimation of each knee and hip joint (black) and the respective predefined motor speed limits (red).

In this experiment, Laelaps II is initially in a standing position with all four legs configured with the parameters shown in Table 6.1. The parameters `a_ellipse100` and `b_ellipse100` are set to 0 at the beginning of the experiments, therefore the elliptical trajectory is just a point. After the recording (with *rosbag*) begins, `b_ellipse100` parameter (which corresponds to the clearance from the ground) is increased to 4 cm linearly with time (the rate of this increase

depends on the value of `Transition_time` variable, which was set to 3 seconds throughout the experiment) to all slaves simultaneously, and similarly `a_ellipse100` variable (which corresponds to the step length) is linearly increased to 5 cm. Laelaps starts trotting slowly and accelerates to reach a constant forward velocity. After several steps, the parameters are again changed to their initial values (first `a_ellipse100` and then `b_ellipse100`), Laelaps decelerates and eventually stops walking and remains still. The recording is terminated and all data are saved and post processed in Matlab.

The experiment's parameters are depicted in Table 6.1 and in Table 6.2:

Table 6.1: *Trotting Experiment parameters.*

Parameters		FL Leg	FR Leg	HL Leg	HR Leg
Trajectory Parameters	`x_cntr_traj1000` (mm)	0	0	0	0
	`y_cntr_traj1000` (mm)	599	599	598	598
	`a_ellipse100` (cm)	5	5	5	5
	`b_ellipse100` (cm)	4	4	4	4
	`traj_freq100` (Hz / 100)	100	100	100	100
	`phase_deg` (deg)	180	0	0	180
	`Transition_time` (sec)	3	3	3	3
	`FlatnessParam100`	0	0	0	0
Control Gains of Knee	`Kp100_knee`	8000	8000	8000	8000
	`Kd1000_knee`	50	50	50	50
	`Ki100_knee`	0	0	0	0
Control Gains of Hip	`Kp100_hip`	8000	8000	8000	8000
	`Kd1000_hip`	50	50	50	50
	`Ki100_hip`	0	0	0	0
PWM max values (%)	Knee	38.25	38.25	38.25	38.25
	Hip	41.17	41.17	41.17	41.17
Filter Bandwidth Frequency	`FilterBandwidth` (Hz)	20	20	20	20
Control Loop Frequency (kHz)		10	10	10	10

Table 6.2: *Parameters independent of EtherCAT application.*

Parameters	Values
Loop Frequency of EtherCAT	2.5 kHz
Shift Time of Sync0 Interrupt	55 μs
Voltage Supply (System)	40.34 V
Max Value of Current (System)	50.11 A

6.2.1.2 Frequency Experiment

In this experiment, the developed application provides the parameters of the desired elliptical trajectory for the toe of each leg along with other parameters of the system, however High Power is not provided to the robot, hence the robot doesn't move. Therefore, the developed application and the EtherCAT application are running in the same way as in the trotting experiment, but there is no actuation from the motors.

For this experiment, the table describing the parameters used is the same as in the trotting experiment, shown in Table 6.1 and Table 6.2, although the *Loop Frequency of EtherCAT* is changed.

In this experiment, Laelaps II is in a standing position with all four legs configured with the parameters shown in Table 6.1. The parameters `a_ellipse100` and `b_ellipse100` are set to 0 throughout the experiment. In each round of the experiment a configuration (as shown in Table 6.4) is tested with a specific EtherCAT Loop Frequency and the kernel logs are checked for skipped EtherCAT frames. If there are skipped frames, this means that the frequency provided is not achievable by the according configuration. The experiment's parameters are depicted in Table 6.3 and in Table 6.4. In each first round the EtherCAT Loop Frequency is set to $2.5kHz$ and in each following round the EtherCAT Loop Time $(1/EtherCATLoopFrequency)$ is decreased by $50\mu\, s$.

Table 6.3: *Parameters independent of EtherCAT application.*

Parameters	Values
Time of each round	10 minutes
Shift Time of Sync0 Interrupt	55 μs

Voltage Supply (System)	40.34 V
Max Value of Current (System)	50.11 A

Table 6.4: *Configurations tested.*

EtherLab Configuration	**Optimizations**
EtherLab with Generic Driver	No
EtherLab with Generic Driver	Yes
EtherLab with Native Driver	No

6.2.2 Results

6.2.2.1 Trotting Experiment Results

During the steady state phase of the experiment, where both the `a_ellipse100` and `b_ellipse100` parameters have reached their final value, the toe (End Effector) of every leg performs a specific path trying to converge with the desired elliptical trajectory. The desired elliptical trajectory of each toe (red) along with the actual response of every leg (black) in their workspace, with respect to the coordinate systems located in the hip joints of the legs (O point in Figure 4.9), are shown in Figure 6.5.

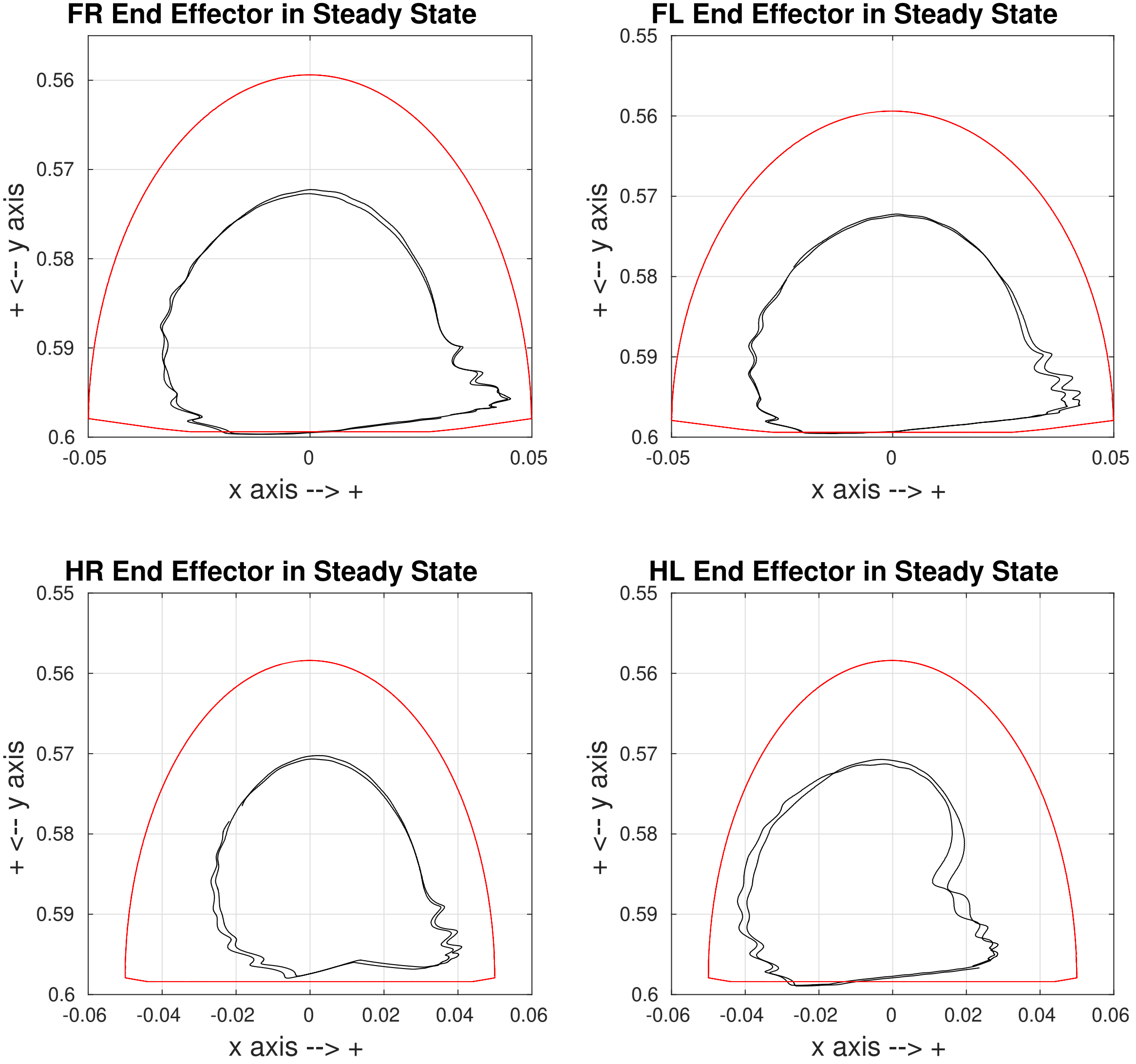


Figure 6.5: *Desired elliptical trajectory of all legs toe (red) along with their actual response (black) w.r.t coordinate systems located in the hip joints of the legs.*

This figure clarifies the fact that steady state errors in the hip and knee joints are adjourned as errors to the positioning of the toe. It is worth mentioning that due to the ground and the low values of the Control Gains, the desired elliptical trajectories are not tracked closely at the "steady" state and a better tuning of these gains is required, especially for the hind legs. Furthermore, the gains for the I term of the control scheme were 0, so proper tuning of these gains is required too.

Figure 6.6 displays the desired value of each knee joint (red) and the actual response of this joint (black) throughout the experiment.

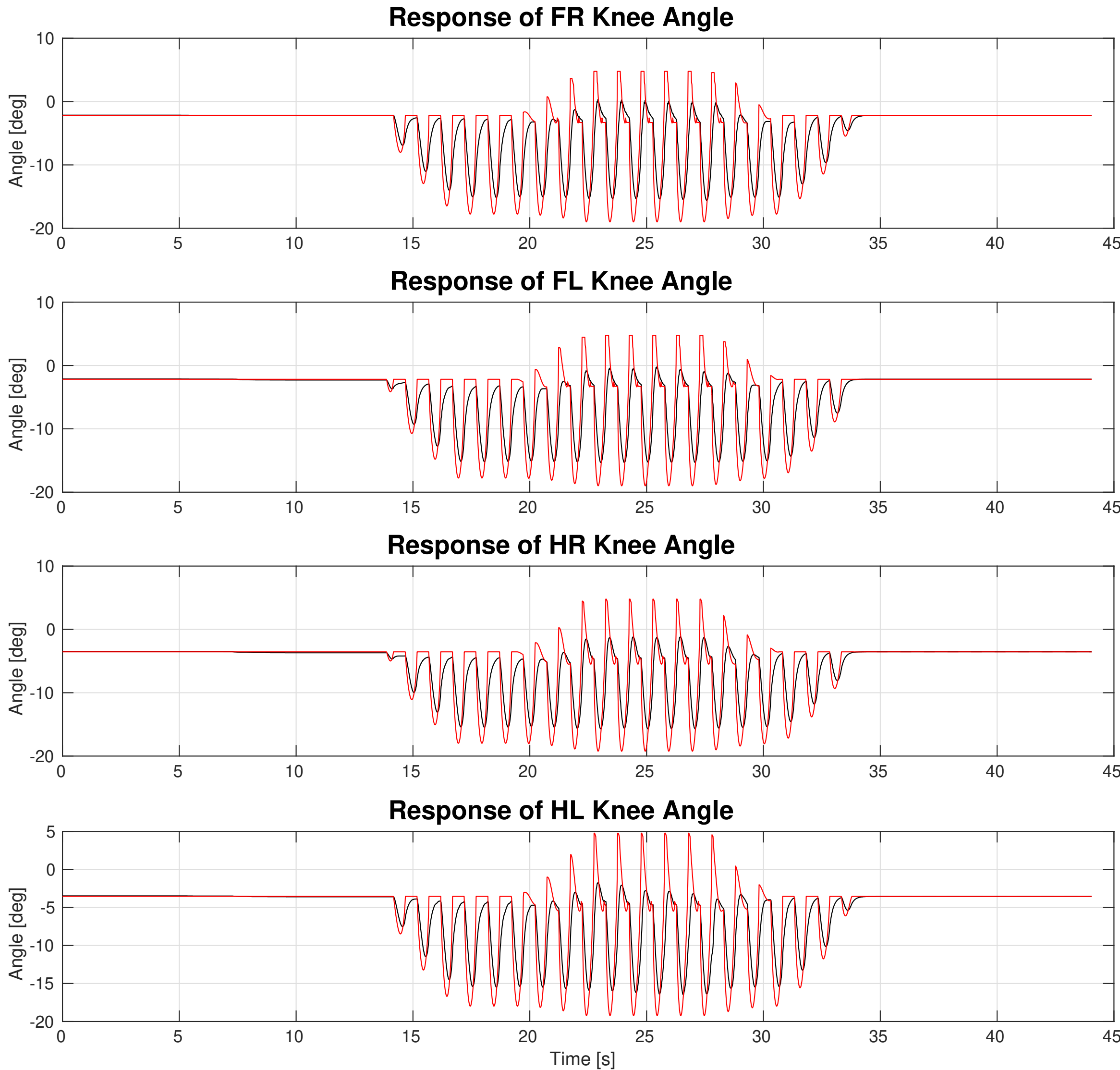


Figure 6.6: *Desired response of knee angles (red) and actual response of knee joint (black).*

Both the transition and the steady state phase are illustrated. The desired values are closely tracked by all legs, yet there is plenty of room for improvement which can be achieved by a judicious regulation of the control gains for the knee motors.

In a similar manner, Figure 6.7 describes the desired value of each hip joint angle (red) and the actual response of every hip joint (black) throughout the experiment.

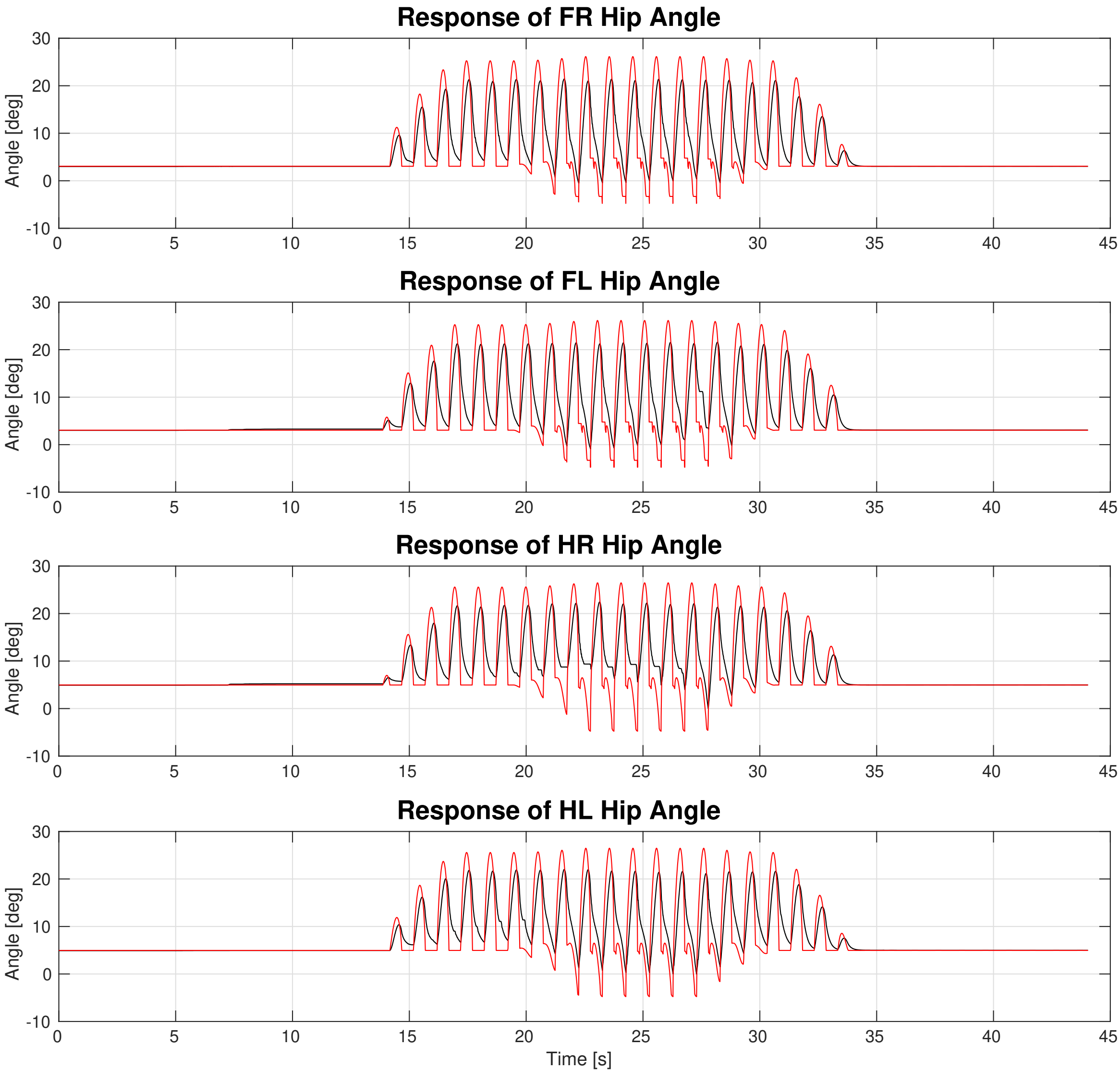


Figure 6.7: *Desired response of hip angles (red) and actual response of hip joint (black).*

Both the transition and the steady state phase are illustrated. The desired values are closely tracked by all legs, yet there is plenty of room for improvement (even more than the knee motors) which can be achieved by proper regulation of the control gains for the hip motors. Since identical control gain values were used for both motors (brushed and brushless) it is totally understandable why these two joints don't have an identical response as far as errors are concerned. Moreover, it is worth mentioning that the hip joint performs a wider movement which is another reason why the resulting errors are larger compared to the knee joints. Another reason explaining the large errors, is the absence of an I term or a feedforward term, which could support the robot weight.

Figure 6.8 depicts the PWM commands [%] of each knee motor (black) with its respective predefined limit (red). These commands are the output of the knee's PIV controller and are

directly translated in torque commands since a current control architecture is implemented. As it can be observed, the commands in both hind legs are always within the limit. Concerning the two forelegs, although the limits are reached several times, due to the fact that it happened only for short intervals, no extra action is needed.

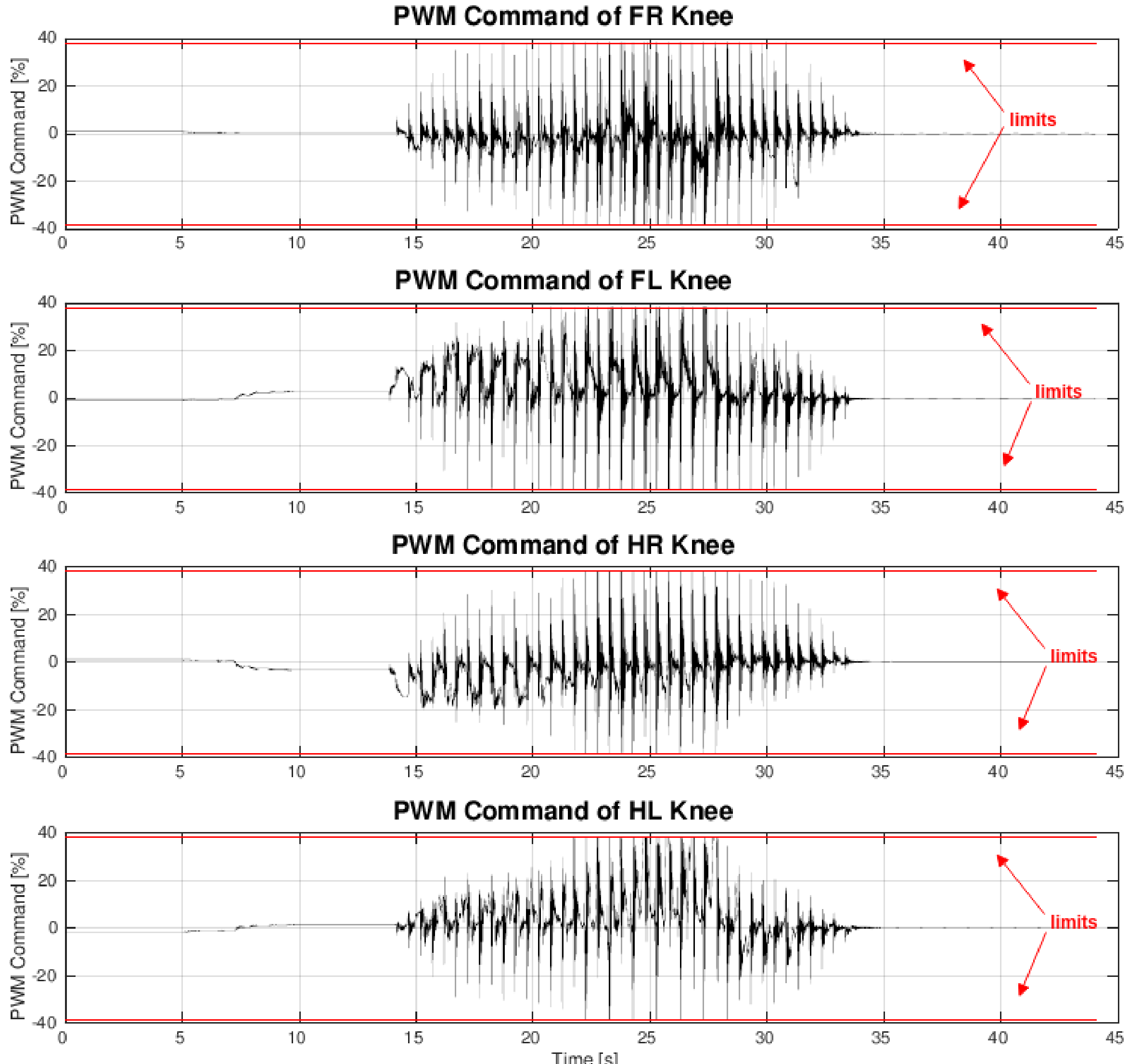


Figure 6.8: *PWM commands of each leg's knee motor (black) and the respective predefined PWM limits (red).*

Similarly, Figure 6.9 depicts the PWM commands [%] of each leg's hip motor (black) with its respective predefined limit (red). These commands are the output of the hip's PIV controller, and are directly translated in torque commands since a current control architecture is implemented. As it can be observed, hip PWM limits are recurrently reached, especially in the hind legs, thus an increase of the allowed range should be considered.

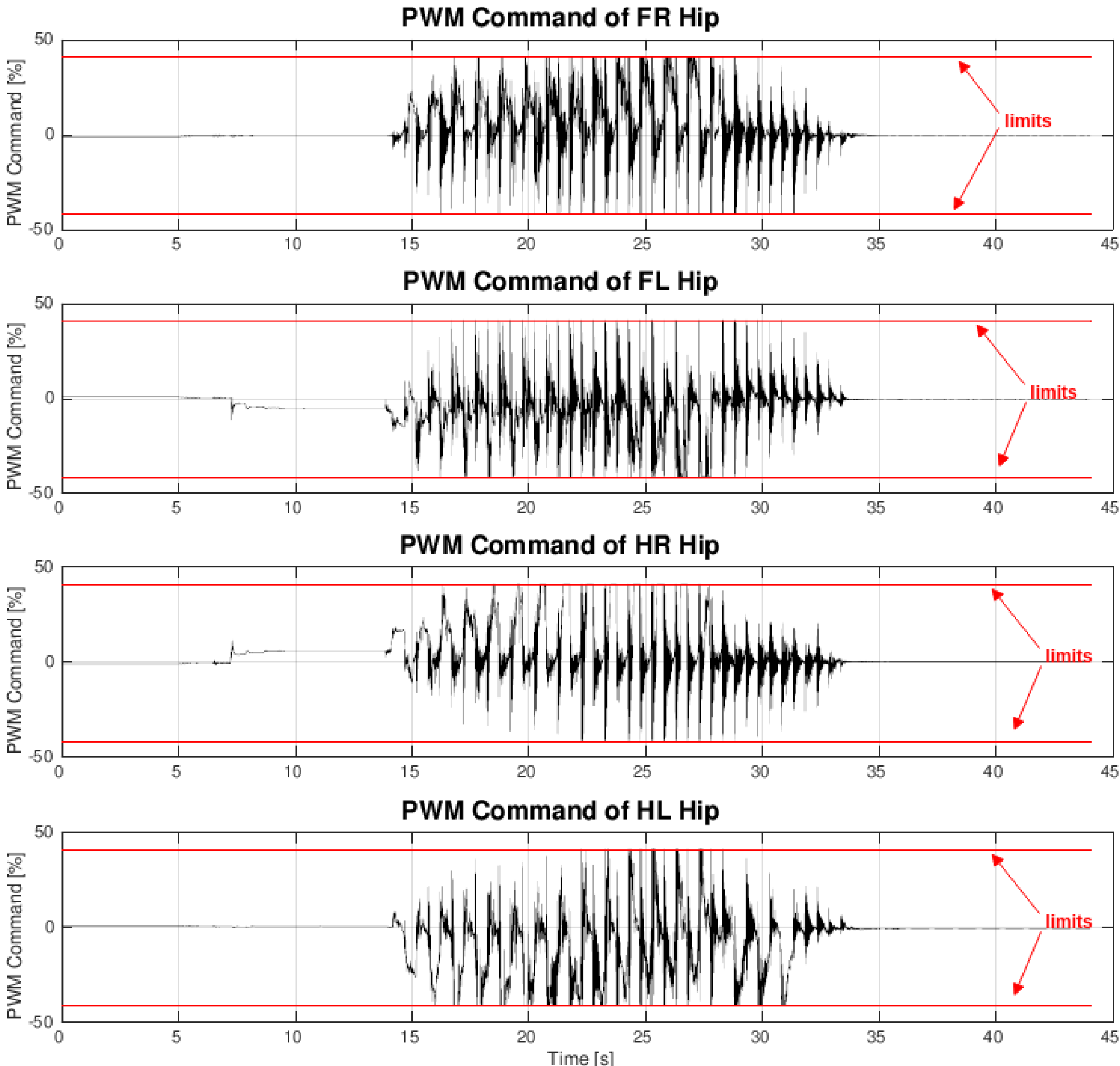


Figure 6.9: *PWM commands of each leg's hip motor (black) and the respective predefined PWM limits (red).*

Figure 6.10 presents the velocity estimation of each leg's knee joint (black) and the respective motor speed limits (red) as specified by the manufacturer (briefly described in Subsection 4.2.2). As it can be noticed from Figure 6.10, the velocities of all knee motors are always within the allowed range.

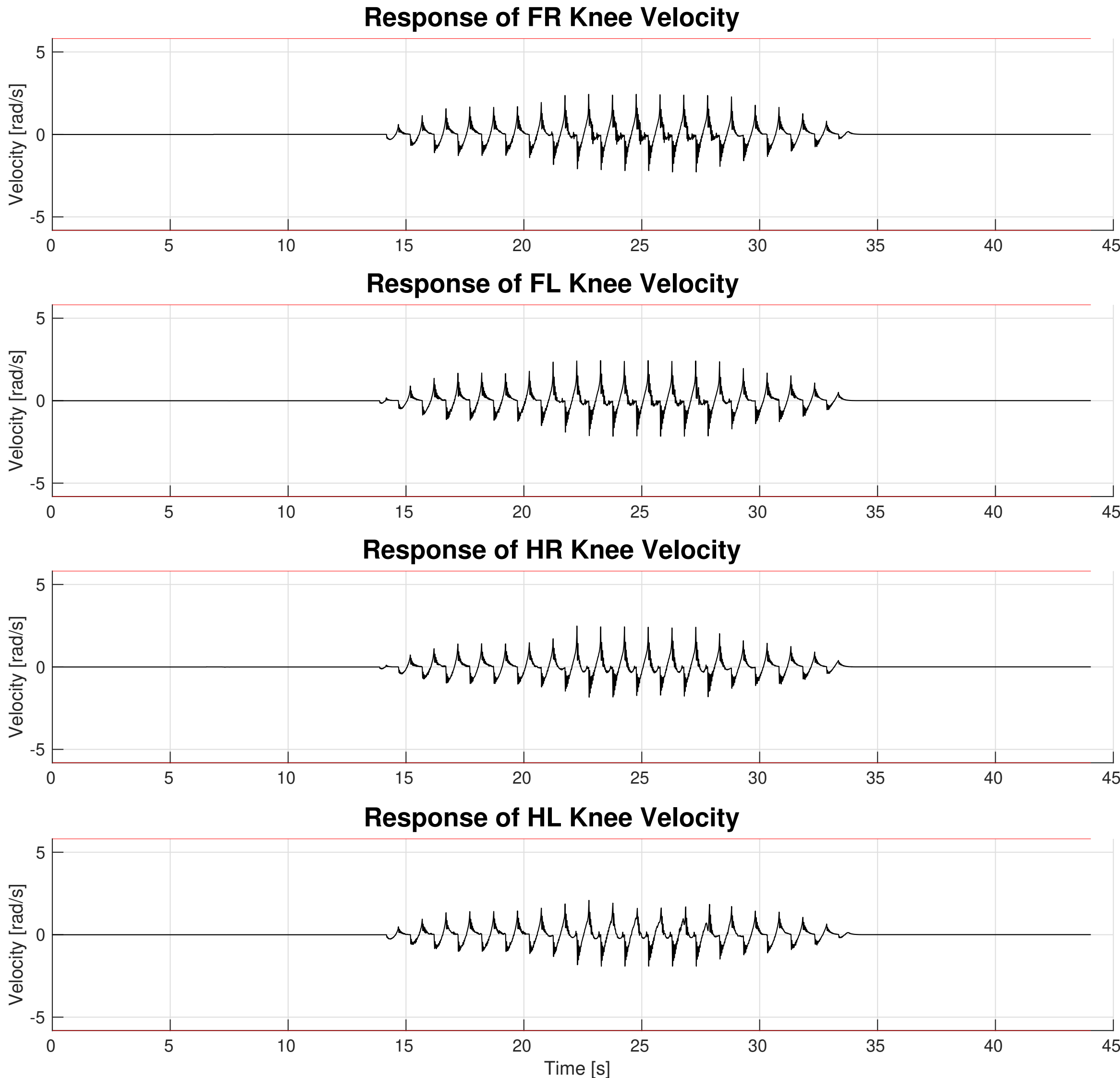


Figure 6.10: *Velocity estimation of each leg's knee joint (black) and the respective predefined motor speed limits (red).*

Finally, Figure 6.11 illustrates the velocity estimation of each hip joint (black) and the respective motor speed limits (red) as specified by the manufacturer (briefly described in Subsection 4.2.2). Once again, the velocities of every hip motor are always within the allowed range, thus there is no need to consider reducing hip PWM limits.

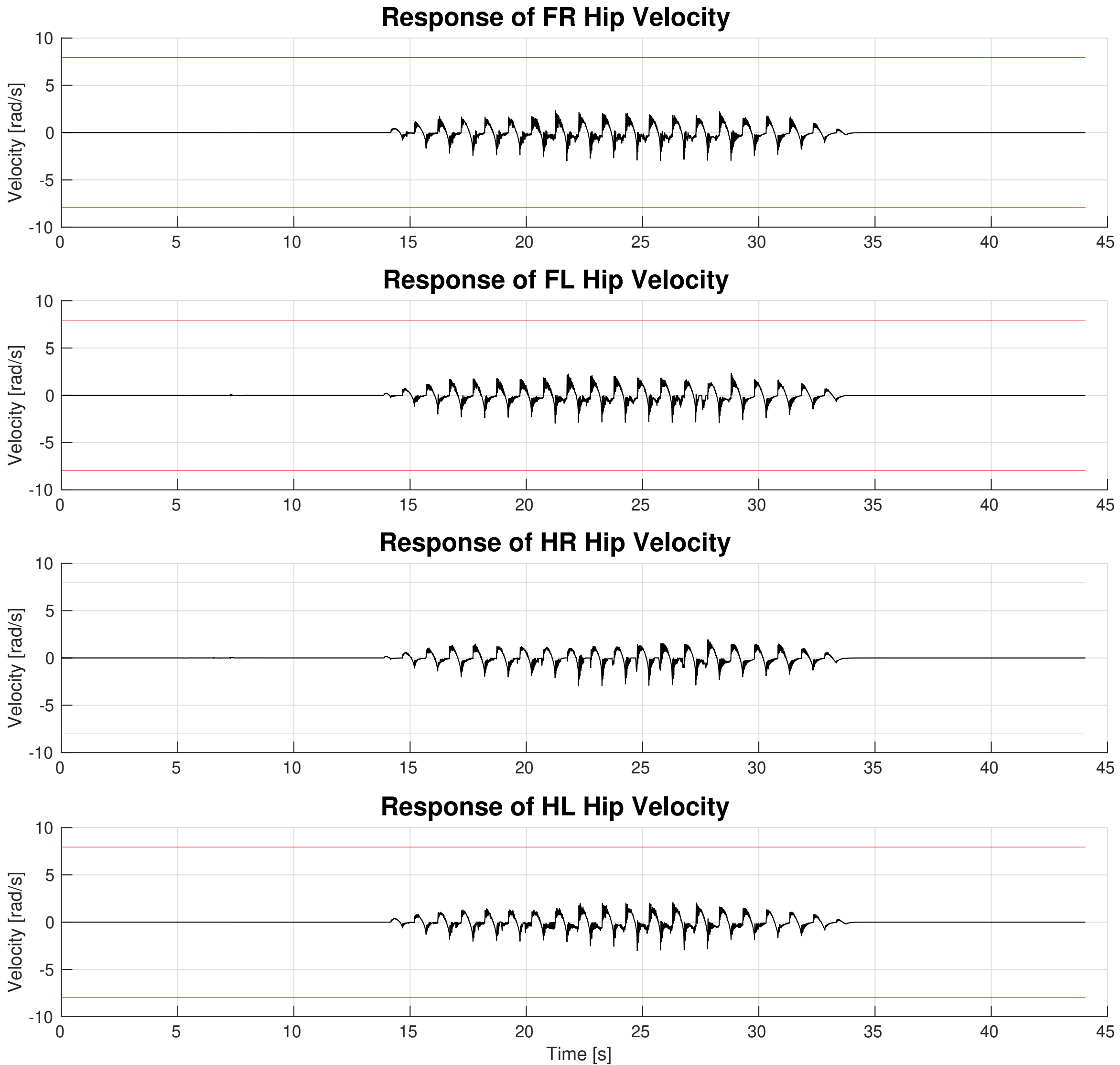


Figure 6.11: *Velocity estimation of each leg's hip joint (black) and the respective predefined motor speed limits (red).*

In this section, the presented results depict a successful trotting experiment on Laelaps II. Minor modifications on control gains and ratios (PWM commands) are recommended for improved performance on the gait, which should be further investigated. From the EtherCAT master perspective, it should be noted that `ether_ros` worked as expected, without skipped packets from EtherLab, in spite of using the generic driver. This means that the real-time constraints, as analyzed in the functional requirements, were respected, with $2.5\ kHz$ **EtherCAT loop frequency achieved**. Finally, regarding the ROS environment, the messages were received successfully on the appropriate topics on time.

6.2.2.2 Frequency Experiment Results

As it can be observed from the following Table 6.5, the native driver is superior to the generic by a large margin (close to 120 μs). The apparent reason as illustrated and explained in the previous chapters, is the use of the Linux Network Stack by the generic driver, which adds extra latency resulting to skipped packets from a Loop Time and below. The native driver may be superior to the generic, however it has limited availability. If the native driver is not available for the existing system, further development needs to be done for creating one. Thus a well-known tradeoff arises once again, between availability and performance. It should be noted that when the master (using EtherLab with native driver) requested from the EtherCAT slaves to switch to Operation State and the Loop Time was equal or below 220 μs, the slaves refused to switch. Therefore for the specific EtherCAT payload (240 bytes) and for the specific slaves used, the threshold of the Loop Time is 230 μs.

Table 6.5: *Frequency Experiment Results.*

Configuration	**Optimizations**	**EtherCAT Loop Time (μs)**	**Number of skipped packets**
EtherLab with Generic	No	400	0
EtherLab with Generic	No	350	≈ 100
EtherLab with Generic	Yes	400	0
EtherLab with Generic	Yes	350	0
EtherLab with Generic	Yes	300	≈ 100
EtherLab with Native	No	400	0
EtherLab with Native	No	350	0
EtherLab with Native	No	300	0

EtherLab with Native	No	250	0
EtherLab with Native	No	240	0
EtherLab with Native	No	230	0

Nevertheless, the EtherCAT application could run in a $4.348\ kHz$ **EtherCAT Control Loop frequency**, a frequency far beyond the requested one. From this experiment, it was also measured that the time the EtherCAT frame (240 bytes) needed to traverse the EtherCAT network and return to the master has a median of $110\ \mu s$ and a variance of $4\ \mu s$. Also the time needed by the developed application to receive an EtherCAT frame, create the new PDOs and send the new frame to the network was less than $30\ \mu s$.

7 Conclusions & Future Work

> We are made wise not by the recollection of our past, but by the responsibility for our future.
>
> *George Bernard Shaw*

In this final chapter, an overall assessment of the developed project is drawn and conclusions are presented regarding the results and the technologies used. Finally, a few directions for further improvement (considered worthy of investigation) and future work are outlined.

7.1 Concluding Remarks

All in all, the requirements formulated in Chapter 4 were managed and met succesfully. According to the experimental evaluation results, the design and development of this thesis achieved to combine the technologies of EtherCAT and ROS under real-time constraints and produce the outcome of a trotting quadruped robot, namely Laelaps II.

In more detail, the real-time capabilities offered by the PREEMPT_RT patch proved to be highly sufficient for the motion control of Laelaps II and the combination of the patch along with EtherLab proved to be worthy superseder of the Windows / TwinCAT approach. Regarding the PREEMPT_RT patch, although a fair amount of development time was consumed on tweaking the system's kernel and the application's code in order to optimize the master's latency, this cost is considered to be far smaller than other approaches like Xenomai and RTAI,

which may offer better performance, yet trading off maintainability and development costs. As far as EtherLab is concerned, the design decision to adopt this approach instead of SOEM, proved to be wise during the development and the validation process. Although a hard and steep learning curve was involved for understanding the way of developing code that utilizes it's API, the documentation was excellent and facilitated the process of development. Also, EtherLab showed it's strength in the debugging process, since it offered mechanisms to examine instantly every aspect of the EtherCAT network.

In addition, on top of the real-time capabilities, the developed application offers inter-operability with the ROS environment, through it's ROS API. This step opens a lot of possibilities, considering the size of the ROS ecosystem and the diversity of the currently developed applications in it. The future ROS nodes will have the ability to communicate with the encoders and motors of Laelaps II and orchestrate profiles of synchronized motions of the legs. These profiles could start with simple ones like trotting, studied in this thesis, and continue with highly complex like galloping and running or combinations of them. This feature shouldn't be neglected; the *ROS*-ification of Laelaps II is a huge step towards software modularity and reduction of development and maintaining costs, important factors for both academia and industry.

Last but not least, the EtherCAT communication protocol proved to be highly efficient and useful throughout the experimental validation. Depending on the data payload which is intimately connected to the size of the EtherCAT frame, this technology can reach really low cycle times and guarantee proper communication between a master and several slaves exploiting only a few really affordable devices (MCUs and ESCs). To put this information in context, the total purchasing cost of all the required components for the control architecture of Laelaps II is almost 10% of the previous version (Laelaps I). Conclusively, the decision to switch towards EtherCAT technology was judicious and wise due to its alleviating functionalities, especially in the motion control area. The new decentralized architecture will certainly enable Laelaps II to perfom higher frequency motions and reach its maximum velocity, with only minor upgrades on its mechanical system, enabling future projects in the CSL-EP laboratory to easily adopt this scheme.

7.2 Future Work

Although the current implementation of motion control via EtherCAT on Laelaps II has been tested and has been proven to be fully functional at the software and hardware level, several

aspects can be improved in the future to achieve greater robustness.

Firstly, the developed application can be extended to support different payloads for EtherCAT slaves and automatic configuration of a new EtherCAT application without manual configuration on the `ether_ros` source code.

An additional idea, is to conduct experiments for tracing the latencies in every aspect of the system. The EtherCAT loop time is consumed among the network, the slaves and the master, and it would be useful to know the amount of time each component of the system consumes. To this direction, tracing tools[1] in the kernel could be utilized in order to trace which process is run by which CPU, how much time does it take to run etc. In this way, the latency of the application, the kernel and the EtherCAT network can be traced. The EtherCAT network's latency can be traced by measuring the intervals between two consecutive interrupts of the Ethernet IRQ dedicated to the EtherCAT network. This latency can be easily (but not so accurately) monitored by the use of Wireshark too. The kernel latency consists of latencies introduced by the EtherLab module, the scheduler and other interrupts not related with the EtherCAT network (e.g. timers, IPIs).

Another possible direction, is experimentation with a different scheduling policy and comparison with the currently used `SCHED_FIFO`. A suitable candidate could be `SCHED_DEADLINE` policy, however extra technical effort is required for integrating this policy to the existing project's workflow.

Moreover, should the current control scheme change and become centralized, the `ether_ros` should change too. A centralized approach means more throughput towards `ether_ros` and the developed application is not optimized for this kind of case. For this approach to work, the application needs to be changed significantly, by optimizing the interaction with the ROS environment, like the number of threads to be used, specific callback queues etc.

Finally, when the thesis was halfway finished, new approaches of motion control based on open-source standards emerged. These approaches are based on ROS 2 (version "crystal" at the time of this writting), which is yet at an early stage of maturity, but is thought to be the successor of ROS. These approaches deal with real-time constraints with the offered inherent features of ROS 2. Currently, the approach of controlling a robotic arm without the need for a fieldbus system is proposed [46, 21, 118, 119].

[1] `http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html`

Although EtherCAT offers unquestionably many benefits, the need for programming both the master and the slaves persists. On the contrary, this approach proposes a motion control of a robotic arm, by adopting ROS 2, PREEMPT_RT patch, H-ROS, HRIM [120] and Ethernet enhanced with Time Sensitive Networking (TSN) standards [119], which offers no extra development cost for programming the "slaves". In fact, in this approach there is no master and slaves, merely a computer and the objects (sensors, actuators) publishing and listening to specific topics, which the computer can take advantage of. This is achieved by erasing a variable from the existing equation, namely EtherCAT, and replacing it with deterministic Ethernet under TSN standards (IEEE 802.1 AS) [119]. There is a caveat however; the currently supported motors are only of a specific brand[2], therefore the shift is not necessarily applicable to the CSL-EP laboratory's setting, yet this approach deserves some research.

[2]https://acutronicrobotics.com/docs/products/actuators/modular_motors/hans/specification

Bibliography

[1] B. Wiliamowski and J. Irwin, *Industrial Communications Systems: The Industrial Electronics Handbook*. CRC Press, 2011.

[2] F. Pose, *IgH EtherCAT Master 1.5.2 Documentation*, Ingenieurgemeinschaft IgH, 10 2017, [Accessed 18-March-2019]. [Online]. Available: https://www.etherlab.org/download/ethercat/ethercat-1.5.2.pdf

[3] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri, *Scheduling in Real-Time Systems*. John Wiley & Sons, November 2002, pp. 8–33, 93–96. [Online]. Available: https://doi.org/10.1002/0470856343

[4] Alison Chaiken, "IRQs: the Hard, the Soft, the Threaded and the Preemptible," October 2016, [Accessed 14-March-2019]. [Online]. Available: https://events.static.linuxfound.org/sites/events/files/slides/Chaiken_ELCE2016.pdf

[5] Yaghmour, Karim and Masters, Jonathan and Ben-, Gilad, *Building Embedded Linux Systems, 2nd Edition*, 2nd ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2008.

[6] Steven Rostedt, "Real-Time is coming to Linux: What does that mean for you?" October 2018, [Accessed 14-March-2019]. [Online]. Available: https://events.linuxfoundation.org/wp-content/uploads/2017/12/elc-eu-2018-rt-what-does-it-mean_Steven-Rostedt.pdf

[7] Y. Pyo, H. Cho, L. Jung, and D. Lim, *ROS Robot Programming (English)*. ROBOTIS, 12 2017, pp. 10–15, 41–63.

[8] *EtherCAT and EtherCAT-P Slave Implementation Guide*, 3rd ed., Beckhoff Automation GmbH, 3 2018, [Accessed 18-March-2019]. [Online]. Available: https://www.ethercat.org/download/documents/ETG2200_V3i0i4_G_R_SlaveImplementationGuide.pdf

[9] *EtherCAT Slave Controller Hardware Data Sheet Section I: Technology*, 2nd ed., Beckhoff Automation GmbH, 7 2014, [Accessed 18-March-2019]. [Online]. Available: https://download.beckhoff.com/download/document/io/ethercat-development-products/ethercat_esc_datasheet_sec1_technology_2i3.pdf

[10] Elmo Motion Control Ltd., "Multi-Axis Position Control by EtherCAT Real-Time Networking," january 2012, [Accessed 18-March-2019]. [Online]. Available: https://products4engineers.nl/images/default/vbVUrK-pdf.pdf

[11] J. Liu, X. Li, M. Liu, X. Cui, and D. Xu, "A new design of clock synchronization algorithm," *Advances in Mechanical Engineering*, vol. 6, 2014. [Online]. Available: https://journals.sagepub.com/doi/pdf/10.1155/2014/958686

[12] *Synchronization Modes*, EtherCAT Technology Group, 2019, etherCAT Technology Group (ETG) Knowledge Base. Only available for ETG Members. [Accessed 18-March-2019].

[13] EtherCAT Technology Group, "EtherCAT Synchronization," 5 2014, etherCAT Technology Group (ETG) Knowledge Base. Only available for ETG Members. [Accessed 18-March-2019].

[14] ——, "How-To Configure DC SYNC Shift Times," 12 2017, etherCAT Technology Group (ETG) Knowledge Base. Only available for ETG Members. [Accessed 18-March-2019].

[15] *EtherCAT Protocol Enhancements*, EtherCAT Technology Group, 12 2015, etherCAT Technology Group (ETG) Knowledge Base. Only available for ETG Members. [Accessed 18-March-2019].

[16] R. Zurawski, *Industrial Communication Technology Handbook*, 2nd ed. CRC Press, 2017.

[17] S. Athiniotis, "Firmware design for microcontrollers on ethercat network for quadruped robot motion control," *Master's thesis* , School of Mechanical Engineering, National Technical University of Athens, 2018.

[18] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed., ser. Real-Time Systems Series. Springer, 2011, pp. 13–17. [Online]. Available: https://doi.org/10.1007/978-1-4419-8237-7

[19] P. A. Laplante, *Real-Time Systems Design and Analysis*, 3rd ed. John Wiley & Sons, April 2004, pp. 4–6. [Online]. Available: https://doi.org/10.1002/0471648299

[20] D. Abbott, *Linux for embedded and real-time applications*, 4th ed. Elsevier, 2017, pp. 258–270.

[21] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Real-time linux communications: an evaluation of the linux communication stack for real-time robotic applications," *arXiv preprint arXiv:1808.10821*, 2018.

[22] Wikipedia contributors, "Tanenbaum-Torvalds debate," 2019, [Accessed 14-March-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Tanenbaum%E2%80%93Torvalds_debate&oldid=884077588

[23] Thomas Gleixner, "Realtime Linux: academia v. reality," July 2010, [Accessed 14-March-2019]. [Online]. Available: https://lwn.net/Articles/397422/

[24] KUNBUS GmbH, "Fieldbus Basics," 2018, [Accessed 4-December-2018]. [Online]. Available: https://www.kunbus.com/fieldbus-basics.html

[25] Wikipedia contributors, "Token passing," 2019, [Accessed 14-March-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Token_passing&oldid=789850568

[26] A. Hansson, "Industrial Ethernet is now bigger than fieldbuses," 2018, [Accessed 5-December-2018]. [Online]. Available: https://www.automationworld.com/article/industrial-ethernet-now-bigger-fieldbuses

[27] J. Pinto, "From Fieldbus to Industrial Ethernet," 2018, [Accessed 4-December-2018]. [Online]. Available: https://www.automationworld.com/article/technologies/networking-connectivity/ethernet-tcp-ip/fieldbus-industrial-ethernet

[28] Wikipedia contributors, "SERCOS III," 2018, [Accessed 4-December-2018]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=SERCOS_III&oldid=855929777

[29] C. Semini, V. Barasuol, J. Goldsmith, M. Frigerio, M. Focchi, Y. Gao, and D. G. Caldwell, “Design of the hydraulically actuated, torque-controlled quadruped robot hyq2max,” *IEEE/ASME Transactions on Mechatronics*, vol. 22, no. 2, April 2017, pp. 635–646.

[30] M. Hutter, C. Gehring, A. Lauber, F. Gunther, C. D. Bellicoso, V. Tsounis, P. Fankhauser, R. Diethelm, S. Bachmann, M. Bloesch, H. Kolvenbach, M. Bjelonic, L. Isler, and K. Meyer, “ANYmal - toward legged robots for harsh environments,” *Advanced Robotics*, vol. 31, no. 17, 2017, pp. 918–931. [Online]. Available: https://doi.org/10.1080/01691864.2017.1378591

[31] Giorgio C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed., ser. Real-Time Systems Series. Springer, 2011, vol. 24, pp. 1, 34–39, 86–118, 428–456. [Online]. Available: https://doi.org/10.1007/978-1-4614-0676-1

[32] J. A. Stankovic and K. Ramamritham, Eds., *Tutorial: Hard Real-time Systems*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1989.

[33] K. Schwan, P. Gopinath, and W. Bo, “CHAOS-kernel support for objects in the real-time domain,” *IEEE Transactions on Computers*, vol. C-36, no. 8, Aug 1987, pp. 904–916.

[34] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, “Distributed fault-tolerant real-time systems: the mars approach,” *IEEE Micro*, vol. 9, no. 1, Feb 1989, pp. 25–40.

[35] J. A. Stankovic and K. Ramamritham, “The spring kernel: a new paradigm for real-time systems,” *IEEE Software*, vol. 8, no. 3, May 1991, pp. 62–72.

[36] H. Tokuda and C. W. Mercer, “ARTS: A distributed real-time kernel,” *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 3, Jul. 1989, pp. 29–53. [Online]. Available: http://doi.acm.org/10.1145/71021.71023

[37] I. Lee, R. King, and R. Paul, “RK: A real-time kernel for a distributed system with predictable response,” *Technical Reports (CIS)*, p. 714, 1988.

[38] I. Lee and R. King, “Timix: a distributed real-time kernel for multi-sensor robots,” in *Proceedings. 1988 IEEE International Conference on Robotics and Automation*, April 1988, vol. 3, pp. 1587–1589.

[39] S. T. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala, "The MARUTI hard real-time operating system," in *[1989] Proceedings. The Fourth Israel Conference on Computer Systems and Software Engineering*, June 1989, pp. 5–15.

[40] D. D. Kandlur, D. L. Kiskis, and K. G. Shin, "HARTOS: A distributed real-time operating system," *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 3, Jul. 1989, pp. 72–89. [Online]. Available: http://doi.acm.org/10.1145/71021.71025

[41] K. Jeffay, D. Stone, and D. Poirier, "Yartos: Kernel support for efficient, predictable real-time systems," *IFAC Proceedings Volumes*, vol. 24, no. 2, 1991, pp. 7 – 12. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474667017512604

[42] G. C. Buttazzo and M. D. Natale, "HARTIK: a hard real-time kernel for programming robot tasks with explicit time constraints and guaranteed execution," in *Proceedings IEEE International Conference on Robotics and Automation*, vol. 2, May 1993, pp. 404–409.

[43] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*, IEEE, 1999, p. 111.

[44] S. Oikawa, "Linux/RK: A portable resource kernel in linux," in *In 19th IEEE Real-Time Systems Sumposium*, 1998.

[45] B. Gerkey, "Why ROS 2?" 2017, [Accessed 6-December-2018]. [Online]. Available: https://design.ros2.org/articles/why_ros2.html

[46] C. S. V. Gutiérrez, L. Usategui San Juan, I. Zamalloa Ugarte, and V. Mayoral Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications," *ArXiv e-prints*, p. arXiv:1809.02595, Sep. 2018.

[47] J. Blazewicz, K. H. Ecker, G. Schmidt, and J. Weglarz, *Scheduling in computer and manufacturing systems*. Springer Science & Business Media, 2012.

[48] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[49] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, 1973, pp. 46–61.

[50] Wikipedia contributors, "Rate-monotonic scheduling," 2019, [Accessed 14-March-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Rate-monotonic_scheduling&oldid=884167834

[51] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *[1989] Proceedings. Real-Time Systems Symposium*, Dec 1989, pp. 166–171.

[52] E. Bini, G. Buttazzo, and G. Buttazzo, "A hyperbolic bound for the rate monotonic algorithm," in *Proceedings 13th Euromicro Conference on Real-Time Systems*, June 2001, pp. 59–66.

[53] E. Bini, G. C. Buttazzo, and G. M. Buttazzo, "Rate monotonic analysis: the hyperbolic bound," *IEEE Transactions on Computers*, vol. 52, no. 7, July 2003, pp. 933–942.

[54] I. Lee, J. Y.-T. Leung, and S. H. Son, *Handbook of Real-Time and Embedded Systems*, 1st ed. Chapman & Hall/CRC, 2007.

[55] M. Dertouzos, "Control robotics: The procedural control of physical processeds," in *Proc. IFIP congress*, 1974, pp. 807–813.

[56] G. C. Buttazzo, "Rate monotonic vs. edf: judgment day," *Real-Time Systems*, vol. 29, no. 1, 2005, pp. 5–26.

[57] A. Carlini and G. C. Buttazzo, "An efficient time representation for real-time embedded systems," in *Proceedings of the 2003 ACM symposium on Applied computing*. ACM, 2003, pp. 705–712.

[58] G. Buttazzo, P. Gai *et al.*, "Efficient edf implementation for small embedded systems," in *Proc. International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2006.

[59] A. Cervin, "Integrated control and real-time scheduling," Ph.D. dissertation, Lund University, 2003.

[60] S. Sahni, "Preemptive scheduling with due dates," *Operations Research*, vol. 27, no. 5, 1979, pp. 925–934.

[61] Jonathan Corbet, "Software interrupts and realtime," October 2012, [Accessed 14-March-2019]. [Online]. Available: https://lwn.net/Articles/520076/

[62] M. Wilcox and H.-P. Company, "I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers," in *Linux Conference of Australia 2003, linux. conf. au.*, 01 2003.

[63] T. L. et al, *Linux Real-Time Manual*, Enea AB, September 2015, [Accessed 19-March-2019]. [Online]. Available: http://linuxrealtime.org/index.php/Main_Page

[64] Rostedt, Steven et all, "RT-mutex implementation design," June 2017, linux kernel documentation, [Accessed 14-March-2019]. [Online]. Available: https://www.kernel.org/doc/Documentation/locking/rt-mutex-design.txt

[65] ——, "RT-mutex subsystem with PI support," linux kernel documentation, [Accessed 14-March-2019]. [Online]. Available: https://www.kernel.org/doc/Documentation/locking/rt-mutex.txt

[66] ——, "Lightweight PI-futexes," linux kernel documentation, [Accessed 14-March-2019]. [Online]. Available: https://www.kernel.org/doc/Documentation/pi-futex.txt

[67] Jonathan Corbet, "Priority inheritance in the kernel," April 2006, [Accessed 14-March-2019]. [Online]. Available: https://lwn.net/Articles/178253

[68] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010, pp. 41–50.

[69] T. Gleixner and D. Niehaus, "Hrtimers and beyond: Transforming the linux time subsystems," in *Proceedings of the Linux symposium*, vol. 1. Citeseer, 2006, pp. 333–346.

[70] Molnar, Ingo and Gleixner, Thomas, "High resolution timers and dynamic ticks design notes," linux kernel documentation, [Accessed 14-March-2019]. [Online]. Available: https://www.kernel.org/doc/Documentation/timers/highres.txt

[71] Paul McKenney, "A realtime preemption overview," August 2005, [Accessed 14-March-2019]. [Online]. Available: https://lwn.net/Articles/146861/

[72] S.-T. Dietrich and D. Walker, "The Evolution of Real-Time Linux," in *7th RTL Workshop*. Citeseer, 2005.

[73] S. Rostedt and D. V. Hart, "Internals of the rt patch," in *Proceedings of the Linux symposium*, vol. 2, 2007, pp. 161–172.

[74] C. Hallinan, *Embedded Linux Primer: A Practical Real-World Approach*. Pearson Education India, 2011.

[75] C. Simmonds, *Mastering Embedded Linux Programming*. Packt Publishing Ltd, 2017.

[76] R. Love, *Linux system programming: talking directly to the kernel and C library*. "O'Reilly Media, Inc.", 2013, pp. 177–210.

[77] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, Dec 1998, pp. 4–13.

[78] L. Abeni, S. Superiore, and S. Anna, "Server mechanisms for multimedia applications," 1998.

[79] T. Cucinotta and F. Checconi, "The IRMOS realtime scheduler," August 2010, [Accessed 14-March-2019]. [Online]. Available: https://lwn.net/Articles/398470/

[80] G. Lipari and S. Baruah, "Greedy reclamation of unused bandwidth in constant-bandwidth servers," in *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*. IEEE, 2000, pp. 193–200.

[81] L. Abeni, J. Lelli, C. Scordino, and L. Palopoli, "Greedy cpu reclaiming for sched_deadline," in *Proceedings of the Real-Time Linux Workshop (RTLWS), Dusseldorf, Germany*, 2014.

[82] L. Abeni, G. Lipari, A. Parri, and Y. Sun, "Multicore cpu reclaiming: parallel or sequential?" in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 1877–1884.

[83] Luca Abeni, "CPU reclaiming for SCHED_DEADLINE," December 2016, [Accessed 14-March-2019]. [Online]. Available: https://lwn.net/Articles/710360/

[84] Michael Kerrisk et all, *SCHED(7)*, Linux man pages project, [Accessed 14-March-2019]. [Online]. Available: http://man7.org/linux/man-pages/man7/sched.7.html

[85] Wikipedia contributors, "SCHED DEADLINE," 2019, [Accessed 14-March-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=SCHED_DEADLINE&oldid=877700934e

[86] Automotive Linux Wiki, "SCHED DEADLINE," [Accessed 14-March-2019]. [Online]. Available: https://wiki.automotivelinux.org/sched_deadline

[87] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, 2016, pp. 821–839.

[88] Jonathan Corbet, "Deadline scheduling for Linux," October 2009, [Accessed 14-March-2019]. [Online]. Available: https://lwn.net/Articles/356576/

[89] Jonathan Corbet, "Deadline scheduling: coming soon?" December 2013, [Accessed 14-March-2019]. [Online]. Available: https://lwn.net/Articles/575497/

[90] Jonathan Corbet, "Adding periods to SCHED_DEADLINE," July 2010, [Accessed 14-March-2019]. [Online]. Available: https://lwn.net/Articles/396634/

[91] Linux Kernel Contributors, "Deadline Task Scheduling," linux kernel documentation, [Accessed 14-March-2019]. [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt

[92] Steven Rostedt, "Understanding SCHED_DEADLINE," May 2017, [Accessed 14-March-2019]. [Online]. Available: http://events17.linuxfoundation.org/sites/events/files/slides/oss-tokyo-using-sched-deadline-2017.pdf

[93] Michael Kerrisk et all, *SCHED_SETATTR(2)*, Linux man pages project, [Accessed 14-March-2019]. [Online]. Available: http://man7.org/linux/man-pages/man2/sched_setattr.2.html

[94] Wikipedia contributors, "XML-RPC," 2019, [Accessed 3-January-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=XML-RPC&oldid=859831408

[95] Wikipedia contributors, "Ethernet frame," 2019, [Accessed 15-April-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Ethernet_frame&oldid=889578680

[96] Wikipedia contributors, "Precision Time Protocol," 2019, [Accessed 18-March-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Precision_Time_Protocol&oldid=888012740

[97] G. Cena, I. C. Bertolotti, S. Scanzio, A. Valenzano, and C. Zunino, "Evaluation of EtherCAT distributed clock performance," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 1, 2012, pp. 20–29.

[98] ——, "On the accuracy of the distributed clock mechanism in EtherCAT," in *IEEE International Workshop on Factory Communication Systems Proceedings*. IEEE, 2010, pp. 43–52.

[99] S.-M. Park, H. Kim, H.-W. Kim, C. N. Cho, and J.-Y. Choi, "Synchronization improvement of distributed clocks in EtherCAT networks," *IEEE Communications Letters*, vol. 21, no. 6, 2017, pp. 1277–1280.

[100] G. Cena, S. Scanzio, A. Valenzano, and C. Zunino, "Performance analysis of switched EtherCAT networks," in *IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 2010, pp. 1–4.

[101] M. Cereia, I. C. Bertolotti, and S. Scanzio, "Performance of a real-time EtherCAT master under Linux," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, 2011, pp. 679–687.

[102] Maruyama, Tatsuya and Yamada, Tsutomu, "Hardware acceleration architecture for EtherCAT master controller," in *9th IEEE International Workshop on Factory Communication Systems*. IEEE, 2012, pp. 223–232.

[103] S. Scanzio, "SoftPLC-Based Control: A Comparison between Commercial and Open-Source EtherCAT Technologies," in *Handbook of Research on Industrial Informatics and Manufacturing Intelligence: Innovations and Solutions*. IGI Global, 2012, pp. 440–463.

[104] "Getting started with IgH EtherCAT Master for Linux," Synapticon GmbH, tutorial for installing IgH EtherCAT master. [Accessed 18-March-2019]. [Online]. Available: https://doc.synapticon.com/tutorials/getting_started_igh_ethercat_master/installing_igh_ethercat_master

[105] "Industrial communication networks - Fieldbus specifications - Part 4-12: Data-link layer protocol specification - Type 12 elements," International Electrotechnical Commission, Geneva, CH, Standard, 8 2014.

[106] "Industrial communication networks - Fieldbus specifications - Part 6-12: Application layer protocol specification - Type 12 elements," International Electrotechnical Commission, Geneva, CH, Standard, 8 2014.

[107] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers: Where the Kernel Meets the Hardware*, 3rd ed. " O'Reilly Media, Inc.", 2005.

[108] *HOWTO setup Linux with PREEMPT_RT properly*, The Linux Foundation, 6 2017, [Accessed 18-March-2019]. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/preemptrt_setup

[109] G. Bolanakis, "Design and Implementation of a Quadruped Robot Electronic System," *Master's thesis* , School of Electrical and Computer Engineering in National Technical University of Athens, 9 2018. [Online]. Available: http://dspace.lib.ntua.gr/handle/123456789/47768

[110] *Monitoring and Tuning the Linux Networking Stack: Receiving Data*, packagecloud, 6 2016, [Accessed 18-March-2019]. [Online]. Available: https://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/

[111] *Monitoring and Tuning the Linux Networking Stack: Receiving Data Illustrated*, packagecloud, 10 2016, [Accessed 18-March-2019]. [Online]. Available: https://blog.packagecloud.io/eng/2016/10/11/monitoring-tuning-linux-networking-stack-receiving-data-illustrated/

[112] *Monitoring and Tuning the Linux Networking Stack: Sending Data*, packagecloud, 2 2017, [Accessed 18-March-2019]. [Online]. Available: https://blog.packagecloud.io/eng/2017/02/06/monitoring-tuning-linux-networking-stack-sending-data/

[113] Arnout Vandecappelle, *kernel-flow*, The Linux Foundation, 1 2018, [Accessed 19-March-2019]. [Online]. Available: https://wiki.linuxfoundation.org/networking/kernel_flow

[114] Linus Torvalds, main Github repository of Linux Kernel source code. [Accessed 18-March-2019]. [Online]. Available: https://github.com/torvalds/linux

[115] C. Benvenuti, *Understanding Linux Network Internals: Guided Tour to Networking on Linux*. " O'Reilly Media, Inc.", 2006.

[116] K. Machairas and E. Papadopoulos, "An active compliance controller for quadruped trotting," in *24th Mediterranean Conference on Control and Automation (MED)*. IEEE, 2016, pp. 743–748.

[117] "Linux Driver for Intel(R) Ethernet Network Connection," Linux kernel documentation, [Accessed 3-April-2019]. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/e1000e.txt

[118] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, I. M. Goenaga, L. A. Kirschgens, and V. M. Vilches, "Time synchronization in modular collaborative robots," *arXiv preprint arXiv:1809.07295*, 2018.

[119] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Time-sensitive networking for robotics," *arXiv preprint arXiv:1804.07643*, 2018.

[120] I. Zamalloa, I. Muguruza, A. Hernández, R. Kojcev, and V. Mayoral, "An information model for modular robots: the Hardware Robot Information Model (HRIM)," *arXiv preprint arXiv:1802.01459*, 2018.

Appendices

1 Appendix A

1.1 Final script

The system is booted into the operating system with the PREEMPT_RT patch and with the following boot parameters added: isolcpus=2,3 nohz_full=2,3 tsc=reliable. The final script presented here, is applicable in a system with four physical CPUs and a x86 architecture. If this is not the case, changes to this script should be applied.

The final script developed for the aforementioned optimizations (some of them, not all), is presented below:

```
1
2   #!/bin/bash
3
4   ## Create CPU isolation with cgroups (Probably already done with
        isolcpus boot parameter)
5
6   #enable the creation of cpuset folder
7   mount -t tmpfs none /sys/fs/cgroup
8   #create the cpuset folder and mount the cgroup filesystem
9   mkdir /sys/fs/cgroup/cpuset/
10  mount -t cgroup -o cpuset none /sys/fs/cgroup/cpuset/
11  #create the partitions
12  mkdir /sys/fs/cgroup/cpuset/rt
13  mkdir /sys/fs/cgroup/cpuset/nrt
14
15  # add the general purpose CPUs to the nRT set:
16  echo 0,1 > /sys/fs/cgroup/cpuset/nrt/cpuset.cpus
17
18  # add the real-time CPUs to the RT set:
19  echo 2,3 > /sys/fs/cgroup/cpuset/rt/cpuset.cpus
20
21  # make the CPUs in the RT set exclusive, i.e. do not let tasks in other
        sets use them:
22  echo 1 > /sys/fs/cgroup/cpuset/rt/cpuset.cpu_exclusive
23
```

```
24
25 ## Restart real-time CPUs with CPU hotplug
26 # Restart is not needed, because the CPUs are isolated from boot.
27
28 ## Not NUMA-enabled configuration
29
30 # Associate the nRT set with NUMA node 0:
31 echo 0 > /sys/fs/cgroup/cpuset/nrt/cpuset.mems
32
33 # Associate the RT set with NUMA node 0:
34 echo 0 > /sys/fs/cgroup/cpuset/rt/cpuset.mems
35
36
37 ## Configure load balancing
38
39
40 # Disable load balancing in the root cpuset. This is necessary for
       settings in the child cpusets to take effect:
41 echo 0 > /sys/fs/cgroup/cpuset/cpuset.sched_load_balance
42
43 # Then disable load balancing in the RT cpuset:
44 echo 0 > /sys/fs/cgroup/cpuset/rt/cpuset.sched_load_balance
45
46 # Finally enable load balancing in the nRT cpuset:
47 echo 1 > /sys/fs/cgroup/cpuset/nrt/cpuset.sched_load_balance
48
49 # Also kill the irq_balance process of Linux
50 pkill -9 irqbalance
51
52 ## Move general purpose tasks to the general GP partition
53
54 # For each task in the root cpuset, run the following command, where
       each pid of task should occur on its own line: echo pid_of_task >
       /sys/fs/cgroup/cpuset/nrt/tasks
55
56 IFS=$'\r\n' GLOBIGNORE='*' command eval 'cpuset_pids=($(cat
       /sys/fs/cgroup/cpuset/tasks))'
```

```
57 for i in "${cpuset_pids[@]}";
58 do
59 echo $i; echo $i > /sys/fs/cgroup/cpuset/nrt/tasks;
60 done
61
62 ## Move IRQs to the general purpose CPUs
63
64 # Some interrupts are not CPU-bound. Unwanted interrupts introduce
     jitter and can have serious negative impact on real-time
     performance. They should be handled on the general purpose CPUs
     whenever possible. The affinity of these interrupts can be
     controlled using the /proc file system.
65 # First set the default affinity to CPU0 or CPU1 to make sure that new
     interrupts 'wont be handled by the real-time CPUs. The set {CPU0,
     CPU1} is represented as a bitmask set to 3, (20 + 21)..
66 echo 3 > /proc/irq/default_smp_affinity
67
68 # Move IRQs to the nRT partition
69 # echo 3 > /proc/irq/<irq>/smp_affinity
70 # Interrupts that can not be moved will be printed to stderr. When it
     is known what interrupts can not be moved, consult the hardware and
     driver documentation to see if this will be an issue. It might be
     possible to disable the device that causes the interrupt.
71
72 # Typical interrupts that should and can be moved are: certain timer
     interrupts, network related interrupts and serial interface
     interrupts. If there are any interrupts that are part of the
     real-time application, they should of course be configured to fire
     in the real-time partition.
73
74 cd /proc/irq
75 irq_array=($(ls -d */ | cut -f1 -d'/'))
76 for i in "${irq_array[@]}";
77 do
78 echo $i; echo 3 > /proc/irq/$i/smp_affinity;
79 done
80
```

```
81  ## Network queues affinity
82
83  # Linux can route the packets on different CPUs in an SMP system. Also
        this handling can create timers on the specific CPUs, an example is
        the ARP timer management, based on neigh_timer. There are a couple
        of solutions that can be adopted to minimize the effect of rerouting
        packets on different CPUs, like migrating all the timers on the
        non-realtime partition if possible, specifying the affinity of
        network queues on some architectures.
84
85  # If the application needs the packets to be received only in the nRT
        or RT partition then the affinity should be set as follows:
86
87  # echo <NRT cpus mask> > /sys/class/net/<non EtherCAT
        interface>/queues/<queue>/<x/r>ps_cpus
88  # echo <RT cpus mask> > /sys/class/net/<EtherCAT
        interface>/queues/<queue>/<x/r>ps_cpus
89  echo 8 > /sys/class/net/enp5s0/queues/rx-0/rps_cpus
90  echo 8 > /sys/class/net/enp5s0/queues/tx-0/xps_cpus
91
92  echo 3 > /sys/class/net/enp6s0/queues/rx-0/rps_cpus
93  echo 3 > /sys/class/net/enp6s0/queues/tx-0/xps_cpus
94
95  ## Execute a task in the real-time partition
96
97  # Now it is possible to run a real-time task in the real-time partition:
98  # echo pid_of_task > /sys/fs/cgroup/cpusets/rt/tasks
99
100 # Since we have an RT partition with more than one CPU we might want to
        choose a specific CPU to run on. Change the task affinity to only
        include CPU3 in the real-time partition. This is done in the code,
        so no need to be done externally.
101
102
103 ## Time Stamp Counter (tsc - x86 only)
104
105 # The time stamp counter is a per-CPU counter for producing time
```

```
    stamps. Since the counters might drift a bit, Linux will
    periodically check that they are synchronized. But this periodicity
    means that the tick might appear despite using full dynamic ticks.
106
107 # By telling Linux that the counters are reliable, Linux will no longer
    perform the periodic synchronization. The side effect of this is
    that the counters may start to drift, something that can be visible
    in trace logs for example.
108
109 # Here is an example of how to use it as a boot parameter:
110
111 # isolcpus=2,3 nohz_full=2,3 tsc=reliable
112
113
114 ## Delay vmstat timer
115
116 # It is used for collecting virtual memory statistics.The statistics
    are updated at an interval specified as seconds in
    /proc/sys/vm/stat_interval. The amount of jitter can be reduced by
    writing a large value to this file. However, that will not solve the
    issue with worst-case latency.
117
118 # Example (10000 seconds):
119 echo 10000 > /proc/sys/vm/stat_interval
120
121 # BDI writeback affinity
122
123 # It is possible to configure the affinity of the block device
    writeback flusher threads. Since block I/O can have a serious
    negative impact on real-time performance, it should be moved to the
    general purpose partition. Disable NUMA affinity for the writeback
    threads
124 echo 0 > /sys/bus/workqueue/devices/writeback/numa
125
126 # Set the affinity to only include the general purpose CPUs (CPU0 and
    CPU1).
127 echo 3 > /sys/bus/workqueue/devices/writeback/cpumask
```

```
128
129 ## Real-time throttling in partitioned system
130
131 # Real-time throttling (RTT) is a kernel feature that limits the amount
      of CPU time given to Linux tasks with real-time priority. If any
      process that executes on an isolated CPU runs with real-time
      priority, the CPU will get interrupts with the interval specified in
      /proc/sys/kernel/sched_rt_period_us. If the system is configured
      with CONFIG_NO_HZ_FULL and a real-time process executes on a
      CONFIG_NO_HZ_FULL CPU, note that real-time throttling will cause the
      kernel to schedule extra ticks. See Section 2.3, Real-Time
      Throttling and Section 3.2.4, Optimize Real-Time Throttling for more
      information.
132
133 # Disable real-time throttling by the following command:
134 echo -1 > /proc/sys/kernel/sched_rt_runtime_us
135
136 ## Machine check - x86 only
137
138 # The x86 architecture has a periodic check for corrected machine check
      errors (MCE). The periodic machine check requires a timer that
      causes unwanted jitter. The periodic check can be disabled. Note
      that this might lead to that silently corrected MCEs goes unlogged.
      Turn it off on the RT CPUs. For each CPU in the real-time partition,
      do the following:
139
140 echo 0 > /sys/devices/system/machinecheck/machinecheck2/check_interval
141 echo 0 > /sys/devices/system/machinecheck/machinecheck3/check_interval
142
143 # It has been observed that it is enough to disable this for CPU0 only;
      it will then be disabled on all CPUs.
144
145 ## Disabling the NMI Watchdog - x86 only
146
147 # Disable the debugging feature for catching hardware hangings and
      cause a kernel panic. On some systems it can generate a lot of
      interrupts, causing a noticeable increase in power usage:
```

```
148
149 echo 0 > /proc/sys/kernel/nmi_watchdog
150
151
152 ## Increase flush time to disk
153
154 # To make write-backs of dirty memory pages occur less often than the
        default, you can do the following:
155
156 echo 1500 > /proc/sys/vm/dirty_writeback_centisecs
157
158 ## Disable tick maximum deferment
159
160 # To have the full tickless configuration, this patch should be
        included. This allows the tick interval to be maximized by setting
        sched_tick_max_deferment variable in the /proc filesystem. To
        disable the maximum deferment, it should be set to -1.
161
162 echo -1 > /sys/kernel/debug/sched_tick_max_deferment
163
164 ## Disable Memory Overcommit
165
166 echo 2 > /proc/sys/vm/overcommit_memory
167
168 #### Pending ####
169 ## Change the real-time priority of: EtherCAT IRQs, ksoftirqd thread
        for CPU3.
```

Listing 1: *The final script for performing extra real-time optimizations.*