NATIONAL TECHNICAL UNIVERSITY OF ATHENS
School of Mechanical Engineering
Department of Mechanical Design and Control Systems
Control Systems Lab

Diploma Thesis

Firmware design for microcontrollers on EtherCAT network for quadruped robot motion control

Stamatios Athiniotis

Supervising Professor: E.G. Papadopoulos

ATHENS 2018

Abstract

Modern motion control requires large data and resources from different devices, sensors and libraries, which must be recognizable and within reach, to be viable. EtherCAT is the fastest industrial Ethernet technology setting new standards for real-time performance and topology flexibility. When using an EtherCAT infrastructure for motion control, all hardware accessories and necessary motors - cables are easily connected to the system with minimum wiring compared to traditional methods. Thus, reducing clutter within the working area, the possibility of an accident that may be caused by loose cables and other loose fittings within the room decreases, making motion control far more manageable. Motion control solutions based on the EtherCAT framework are purpose-built for enhanced performance and come with immense advantages since machine automation becomes more cost effective with much better turn-around times and outputs.

In this thesis, the development of an EtherCAT network of microcontrollers is introduced, describing the configuration process of a slave and all the implementation prerequisites. For the purpose of implementing EtherCAT technology, TwinCAT XAE (Visual Studio) is exploited to realize the master node in a Windows Operating System (OS). On the other hand, to materialize the slave nodes of the network, the LaunchXL – F28379D launchpad by Texas Instruments is used as the host Micro Controller Unit (MCU) along with FB1111-0141 by Beckhoff as the EtherCAT Slave Controller (ESC). The provided software (available for download) which is thoroughly explained within the second chapter, is fully operational and contains all required EtherCAT stack to implement an analogous network. It is assembled in such a way that no specific User Application is coded, therefore users may define the functionality of each node according to their needs.

Subsequently, in the third chapter, the motion control via EtherCAT of Laelaps II quadruped robot is explained along with a description of its leg architecture and electrical system details. Each slave connected to the robot's network (EtherCAT Control Tower Assembly) controls the motion of one leg, including the hip and knee motors, using elliptical shaped trajectories for its End Effector. Furthermore, the TwinCAT Scope View Tool is adumbrated enabling the data logging of EtherCAT variables for their post processing in Matlab.

Finally, in the fourth chapter, the experimental validation of the exploited decentralized control theory of Laelaps II is appended, illustrating the response of all four legs in a fundamental test. The experimental process is described and a table containing all used parameters is provided before presenting the resulting figures of each joint. The overall procedure proves that the running stack is judiciously assembled, fully functional and prudent to be tested in higher velocities of the robot in the future.

Περίληψη

Ο σύγχρονος έλεγχος κίνησης απαιτεί μεγάλα δεδομένα και πόρους από διαφορετικές συσκευές, αισθητήρες και βιβλιοθήκες, οι οποίοι πρέπει να είναι αναγνωρίσιμοι και διαθέσιμοι προκειμένου να είναι βιώσιμοι. Το EtherCAT είναι η ταχύτερη βιομηχανική τεχνολογία Ethernet που θέτει νέα πρότυπα για την επικοινωνία σε πραγματικό χρόνο και την ευελιξία της τοπολογίας. Όταν χρησιμοποιείται το EtherCAT για έλεγχο κίνησης, όλα τα εξαρτήματα υλικού και οι απαραίτητοι κινητήρες - καλώδια συνδέονται εύκολα στο σύστημα με ελάχιστη καλωδίωση σε σύγκριση με τις παραδοσιακές μεθόδους. Έτσι μειώνοντας την ακαταστασία μέσα στην περιοχή εργασίας, μειώνεται η πιθανότητα ενός ατυχήματος που μπορεί να προκληθεί από χαλαρά καλώδια και άλλα χαλαρά εξαρτήματα μέσα στο χώρο εργασίας, κάνοντας τον έλεγχο κίνησης πολύ πιο εύχρηστο και διαχειρίσιμο. Οι λύσεις ελέγχου κινήσεων βασισμένες στο πλαίσιο EtherCAT είναι κατασκευασμένες με σκοπό τη βελτίωση της απόδοσης και διαθέτουν τεράστια πλεονεκτήματα, καθώς η αυτοματοποίηση του συστήματος καθίσταται πιο αποδοτική από πλευράς κόστους με πολύ καλύτερους χρόνους εκτέλεσης.

Στην παρούσα εργασία παρουσιάζεται η ανάπτυξη ενός δικτύου EtherCAT από μικροελεγκτές, περιγράφοντας τη διαδικασία διαμόρφωσης ενός slave και όλες τις προϋποθέσεις υλοποίησης. Για την πραγμάτωση της τεχνολογίας EtherCAT, το πρόγραμμα TwinCAT XAE (Visual Studio) χρησιμοποιείται για την υλοποίηση του master κόμβου σε λειτουργικό σύστημα Windows (OS). Από την άλλη πλευρά, για την υλοποίηση των slave κόμβων του δικτύου, η πλακέτα LaunchXL-F28379D από την Texas Instruments χρησιμοποιείται ως ο μικροελεγκτής που «φιλοξενεί» το σύστημα, μαζί με το FB1111-0141 από την Beckhoff που διαδραματίζει το ρόλο του EtherCAT Slave Controller (ESC). Το προσφερόμενο λογισμικό, το οποίο αναλύεται διεξοδικά στο δεύτερο κεφάλαιο της εργασίας, είναι πλήρως λειτουργικό και περιέχει όλον τον απαραίτητο EtherCAT κώδικα για την υλοποίηση ενός ανάλογου δικτύου. Έχει προγραμματιστεί με τέτοιο τρόπο ώστε οι χρήστες να μπορούν να ορίζουν τη λειτουργία του κάθε κόμβου slave ανάλογα με τις ανάγκες τους.

Στο τρίτο κεφάλαιο, επεξηγείται ο έλεγχος κίνησης μέσω του EtherCAT του τετράποδου ρομπότ Laelaps II, μαζί με μια περιγραφή της αρχιτεκτονικής του ποδιού και των λεπτομερειών του ηλεκτρικού συστήματος. Κάθε κόμβος slave που συνδέεται με το δίκτυο του ρομπότ (EtherCAT Control Tower Assembly) ελέγχει την κίνηση ενός ποδιού, συμπεριλαμβανομένων των κινητήρων ισχίου και γονάτου, χρησιμοποιώντας ελλειπτικά διαμορφωμένες τροχιές για το πέλμα του (ΤΣΔ). Επιπλέον, περιγράφεται το εργαλείο TwinCAT Scope View που επιτρέπει την καταγραφή και αποθήκευση των μεταβλητών EtherCAT για την μεταγενέστερη επεξεργασία τους στη Matlab.

Τέλος, στο τέταρτο κεφάλαιο παρουσιάζεται η πειραματική επικύρωση της θεωρίας αποκεντρωμένου ελέγχου του Laelaps II, που απεικονίζει την απόκριση και των τεσσάρων ποδιών σε ένα θεμελιώδες πείραμα. Η πειραματική διαδικασία περιγράφεται και παρέχεται ένας πίνακας που περιέχει όλες τις χρησιμοποιούμενες παραμέτρους του πειράματος πριν παρουσιάστουν οι αποκρίσεις των αρθρώσεων. Η συνολική διαδικασία αποδεικνύει ότι ο χρησιμοποιούμενος κώδικας είναι εύστοχα προγραμματισμένος, πλήρως λειτουργικός και ικανός για να δοκιμαστεί σε υψηλότερες ταχύτητες του ρομπότ στο μέλλον.

Acknowledgements

I would like to thank my supervisor Professor E. Papadopoulos for his guidance, his advice and work ethic which inspired and motivated me throughout the elaboration of this thesis and significantly contributed to its quality.

Furthermore, I would like to thank the PhD candidates of the CSL lab and good friends of mine K. Machairas, K. Koutsoukis and T. Mastrogeorgiou for introducing me to the fascinating world of robotics and of course for their incessant and valuable assistance in the last three years that I have been a member of the CSL – EP laboratory. I would also like to personally thank and acknowledge the work of Aristotelis Papatheodorou for selecting EtherCAT components and introducing the team to EtherCAT technology, George Bolanakis for his Software and Hardware support and John Valvis for the leg design and the treadmill support mechanism.

Finally, I would like to thank all those people who supported and encouraged me throughout all my undergraduate years and made all this possible: my family, my friends (MB), classmates and professors.

Dedicated to my Family,
Alex, Anthony
and Sam

Table of Contents

Abstract **3**

Περίληψη **5**

Acknowledgements **7**

Table of Contents **11**

List of Figures **14**

List of Tables **20**

1 Introduction **22**

1.1 Motivation 22

1.2 Literature review 22

1.3 Thesis Outline 26

2 EtherCAT Communication and Implementation **28**

2.1 EtherCAT Technology 28

2.1.1 Introduction 28

2.1.2 Physical Layer 28

2.1.3 Data Link Layer 31

2.1.4 Application Layer 35

2.2 EtherCAT Synchronization 37

2.2.1 Synchronization Overview 37

2.2.2 Free Run Mode 38

2.2.3 SM-Synchronous Mode 39

2.2.4 DC-Synchronous Mode 40

2.3 Process Data Handling 43

2.4 EtherCAT Application Guide 45

2.4.1 EtherCAT Code Structure Overview 45

2.4.2 Hardware and Software Requirements 46

2.4.3 EtherCAT Application Solution Guide 52

3 Motion Control of Laelaps II via EtherCAT **69**

3.1 Laelaps II robot description and motion planning 69

3.1.1 Leg design and motion planning 70

3.1.2 Electrical system 73

3.2 Motion Control of Laelaps via EtherCAT Solution Guide 75

4 Laelaps II Locomotion Experiments **96**

4.1 Trotting Experiment 1 96

5 Conclusion and Future Work **105**

5.1 Conclusion 105

5.2 Future Work 105

References **107**

6 Appendix A **109**

6.1 Download and Install Code Composer Studio, C2000ware & ControlSuite 109

6.2 Download and Install Slave Stack Code Tool 110
6.3 Generate Slave Stack Code for C28x architecture microcontrollers 110
6.4 Download and Install TwinCAT 3 Software 115
6.5 Import CCS project into Code Composer Studio 118
6.6 Define and Select Target Configuration 120
6.7 Add and remove EtherCAT Input and Output variables 122
6.8 TwinCAT in Run Mode 130
6.9 Add Watch Expression in CCS Debug 133
6.10 Laelaps II motors and gearheads 134
6.11 Matlab PIV controller simulation 134

7 Appendix B 136

7.1 Matlab Leg Modelling Code 136
7.2 Matlab Post Process Code 137
7.3 Matlab PIV Controller Simulation 145

List of Figures

Figure 1-1. Boston Dynamics legged robots: (a) Handle (b) SpotMini (c) Atlas (d) BigDog. 23
Figure 1-2. State of the Art legged robots: (a) ANYmal (b) Hermes (c) Cheetah (d) Inu. 23
Figure 1-3. (a) KR C4 Controller with robotic arm by KUKA and (b) MiniBOT Robot by NexCom. .25
Figure 1-4. Shadow Dexterous Hand by Shadow Rob Company. 26
Figure 1-5. (a) Talos biped robot by PAL Robotics and (b) HyQ2Max quadruped robot by IIT. 26
Figure 2-1. EtherCAT Topology. 28
Figure 2-2. A typical EtherCAT network..30
Figure 2-3. EtherCAT topology with branches. 30
Figure 2-4. EtherCAT Frame Structure with EtherCAT Datagrams (a) directly in the data field of the Ethernet frame (b) within the data section of a datagram, by means of the User Datagram Protocol (UDP)...33
Figure 2-5. EtherCAT Datagram (or DLPDU) structure. 33
Figure 2-6. EtherCAT Application Layer State Machine. 36
Figure 2-7. EtherCAT Application Level..37
Figure 2-8. EtherCAT process data exchange. 37
Figure 2-9. EtherCAT Synchronization. 38
Figure 2-10. Slave in Free Run mode. 38
Figure 2-11. EtherCAT network in Free Run mode. 39
Figure 2-12. Slave in SM Synchronous mode. 39
Figure 2-13. EtherCAT network in SM Synchronous mode. 40
Figure 2-14. Slave in DC Synchronous mode. 41
Figure 2-15. EtherCAT network in DC Synchronous mode. 42
Figure 2-16. EtherCAT shift times...42
Figure 2-17. EtherCAT time shifts...43
Figure 2-18. EtherCAT Process Data handling. 43
Figure 2-19. Process data handling generic functions..44
Figure 2-20. Free Run mode process data handling and sequence. 44
Figure 2-21. SM Synchronous mode process data handling and sequence. 44
Figure 2-22. DC Synchronous mode process data handling and sequence. 45
Figure 2-23. EtherCAT Slave Architecture. 46
Figure 2-24. EtherCAT Architecture..46
Figure 2-25. EtherCAT Slave MCU..47
Figure 2-26. EtherCAT Slave Controller. 48
Figure 2-27. Overview of FB1111-0141 features. 48
Figure 2-28. FB1111-0141 Library (a) Device (b) Package. 50
Figure 2-29. Delfino Launchpad Library (a) Device (b) Package. 50
Figure 2-30. Schematic of EtherCAT Slave PCB. 50
Figure 2-31. (a) Top View and (b) Bottom View of EtherCAT Slave PCB. 51
Figure 2-32. EtherCAT Application slave assembly. 51
Figure 2-33. CCS with imported project. 52

Figure 2-34. Basic SSC execution structure. 53
Figure 2-35. APPL_GenerateMapping() function. 54
Figure 2-36. APPL_InputMapping() function. 55
Figure 2-37. APPL_OutputMapping() function. 55
Figure 2-38. APPL_Application() function. 56
Figure 2-39. Select Build Configuration. 57
Figure 2-40. Build and Debug CCS Project. 57
Figure 2-41. CPU selection. 58
Figure 2-42. CCS Debug window. 58
Figure 2-43. EtherCAT Assembly. 59
Figure 2-44. TwinCAT new project. 59
Figure 2-45. TwinCAT EtherCAT Application project. 59
Figure 2-46. TwinCAT Solution Explorer. 60
Figure 2-47. EtherCAT Master realization. 60
Figure 2-48. Reload Device Descriptions. 61
Figure 2-49. Definitions of TwinCAT buttons. 61
Figure 2-50. TwinCAT scan for slaves. 62
Figure 2-51. Update EEPROM of ESC's memory. 62
Figure 2-52. XML selection. 63
Figure 2-53. Remove EtherCAT slave. 63
Figure 2-54. EtherCAT Application in TwinCAT. 64
Figure 2-55. EtherCAT Slave's State Machine. 64
Figure 2-56. Online Write of Blue_LED. 65
Figure 2-57. SyncManager/Sync0/Sync1 Mode. 65
Figure 2-58. DC Sync mode of EtherCAT Application. 66
Figure 2-59. Timers and Flags to watch. 66
Figure 2-60. EtherCAT Application frame disintegrated. 67
Figure 2-61. EtherCAT frame description. 67
Figure 2-62. Example of EtherCAT network with 4 configured slaves. 68
Figure 3-1. Laelaps I. 69
Figure 3-2. Laelaps II. 70
Figure 3-3. Actual and virtual links of Laelaps II legs. 70
Figure 3-4. Leg model. 71
Figure 3-5. Leg's workspace. 72
Figure 3-6. Visualization of legs motion in Matlab. 73
Figure 3-7. Electrical System of Laelaps. 74
Figure 3-8. EtherCAT Control Tower Assembly. 74
Figure 3-9. EtherCAT Control Tower Assembly on Laelaps II. 75
Figure 3-10. APPL_GenerateMapping() function. 77
Figure 3-11. APPL_InputMapping() function. 78
Figure 3-12. APPL_OutputMapping() function. 79
Figure 3-13. APPL_Application() function. 80

Figure 3-14. DCL_runPID_C1() block diagram. 81
Figure 3-15. Epwm1_isr() function. 82
Figure 3-16. Epwm2_isr() function. 83
Figure 3-17. Position & Rotational Speed calculation. 85
Figure 3-18. Modified linker command file to enable DCL functions. 86
Figure 3-19. EtherCAT Control Tower Assembly wired. 87
Figure 3-20. XML selection. 88
Figure 3-21. Add Scope Measurement. 89
Figure 3-22. Add PLC Task. 90
Figure 3-23. Add Global Variable List. 90
Figure 3-24. Global Variable List. 91
Figure 3-25. MAIN (PRG) list. 91
Figure 3-26. Build Solution. 92
Figure 3-27. Linking PLC variables. 92
Figure 3-28. Start PLC task. 93
Figure 3-29. Adding variables to the Scope View. 93
Figure 3-30. TwinCAT Scope Record. 93
Figure 3-31. Save and Export Recording. 94
Figure 3-32. Reset button to initialize legs poise. 94
Figure 3-33. Laelaps II on treadmill ready to perform experiments. 95
Figure 3-34. Laelaps' State Machine. 95
Figure 4-1. 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. 98
Figure 4-2. Desired response of knee angles (red) and actual response of knee joint (black). 99
Figure 4-3. Desired response of hip angles (red) and actual response of hip joint (black). 100
Figure 4-4. PWM commands of each leg’s knee motor (black) and the respective predefined PWM limits (red). 101
Figure 4-5. PWM commands of each leg’s hip motor (black) and the respective predefined PWM limits (red). 102
Figure 4-6. Velocity estimation of each leg’s knee joint (black) and the respective predefined motor speed limits (red). 103
Figure 4-7. Velocity estimation of each leg’s hip joint (black) and the respective predefined motor speed limits (red). 104
Figure 6-1. Code Composer Studio Installation. 109
Figure 6-2. Control Suite Installation. 109
Figure 6-3. C2000ware Installation. 109
Figure 6-4. SSC Tool Download. 110
Figure 6-5. SSC Tool Installation. 110
Figure 6-6. ControlSuite EtherCAT Demo Tool. 111
Figure 6-7. SSC Tool Create new project. 111
Figure 6-8. Importing ESI description file. 111
Figure 6-9. Slave Stack Code - New Project. 112
Figure 6-10. SSC Tool Configuration Options. 112
Figure 6-11. Importing project confirmation. 113

Figure 6-12. SSC Tool slave information. 113
Figure 6-13. Create new Slave Files. 114
Figure 6-14. Create EtherCAT stack and xml file. 114
Figure 6-15. EtherCAT project files. 114
Figure 6-16. CCS project tree. 115
Figure 6-17. TwinCAT 3 Download scheme. 115
Figure 6-18. TwinCAT 3 Installation 116
Figure 6-19. TcSwitchRuntime Installation. 116
Figure 6-20. TcSwitchRuntime Activation. 116
Figure 6-21. TwinCAT Verification. 117
Figure 6-22. Real Time Ethernet Adapter Installation. 117
Figure 6-23. Code Composer Studio starting page. 118
Figure 6-24. CCS Import project. 118
Figure 6-25. CCS project import selection. 119
Figure 6-26. CCS browse and import. 119
Figure 6-27. Project imported CCS window. 120
Figure 6-28. Select View Target Configuration. 120
Figure 6-29. New Target Configuration. 120
Figure 6-30. Name Target Configuration. 121
Figure 6-31. Select Connection and Device. 121
Figure 6-32. Link ccxml file to Project. 122
Figure 6-33. Add initialization definition of new variables. 122
Figure 6-34. Update pOutputSize value. 123
Figure 6-35. Update APPL_OutputMapping() function with new variables. 123
Figure 6-36. New variables object address definition. 124
Figure 6-37. Update SyncManager assignment. 124
Figure 6-38. Object record definition of "Additions". 125
Figure 6-39. Update of ApplicationObjDic[]. 125
Figure 6-40. DT1603 DataType definition. 126
Figure 6-41. DT1C12 and DT1C12ARR DataType definition. 126
Figure 6-42. DT7030 DataType definition. 127
Figure 6-43. Object definition of #x1603. 128
Figure 6-44. Updated Object definition of #x1C12. 128
Figure 6-45. Object definition of #x7030. 129
Figure 6-46. Update Sync Manager Output size. 129
Figure 6-47. RxPdo definition of #x1603. 130
Figure 6-48. Slave device in DC Sync Mode. 130
Figure 6-49. Select minimum EtherCAT cycle time. 131
Figure 6-50. Create I/O Task with Image and Define cycle time. 131
Figure 6-51. Create cyclic Output variable. 131
Figure 6-52. Link software to hardware variable. 132
Figure 6-53. Activate Configuration and switch to Run Mode. 132

Figure 6-54. Enter Security Code. 132
Figure 6-55. Add Expression Tab. 133
Figure 6-56. Add new expression. 133
Figure 6-57. Select variable to inspect..134
Figure 6-58. Block diagram of one actuated degree of freedom of Laelaps II leg............................134
Figure 6-59. Block diagram of the Matlab PIV controller. 135

List of Tables

Table 2-1. Pin Connection. 49
Table 2-2. EtherCAT Application Process Data Interface. 54
Table 2-3. Generic functions execution time. 66
Table 2-4. EtherCAT frame components. 67
Table 3-1. EtherCAT Laelaps Motion Control Output variables. 76
Table 3-2. EtherCAT Laelaps Motion Control Input variables. 77
Table 3-3. Benchmark parameters in Laelaps II experiment. 84
Table 3-4. Velocity calculation range. 86
Table 4-1. Trotting Experiment 1 97

1 Introduction

1.1 Motivation

The purpose of this thesis is to replace the centralized control scheme exploited in Laelaps I quadruped robot of the CLS – EP laboratory (Figure 3-1) with an advanced decentralized scheme using microcontrollers connected via a state of the art communication protocol, to be used in the next version of the robot, Laelaps II. The main reasons which led to this endeavor were that the former architecture (centralized) consisted of a high-cost, hard to replace or to extend, heavy and bulky central control tower (PCIe/104) with 4 interface control card layers to handle all communications. This scheme reached low control loop frequencies and it was burdened with the whole computational payload of the robot employing a non-real-time Linux-ROS (Robot Operating System) implementation; one computer was responsible for controlling the eight motorized joints of Laelaps (two for each leg, knee and hip). Consequently, programming this computational system was unduly demanding, and particularly strenuous to make modifications. Most importantly, in case of any hardware dysfunctionality or damage, the whole system of Laelaps would be disabled, since the really high cost did not allow easy procurance of spare parts.

To overcome these limitations, in this thesis, a new, low-cost, efficient and powerful real-time architecture was designed and implemented for Laelaps II, the new version of this quadruped robot (Figure 3-2). An important question that arose – debatable amongst most engineers nowadays – concerned the protocol that should be used to connect the microcontrollers destined to control Laelaps II; Fieldbus or Industrial Ethernet. After a meticulous validation of those options, it was decided that the most suitable and convenient protocol to employ was an Industrial Ethernet solution, and particularly EtherCAT (Ethernet for Control Automation Technology).

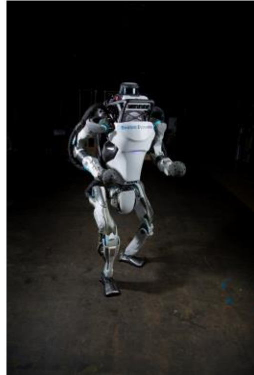
1.2 Literature review

Legged Robot Applications

Legged robots have the potential to become a new generation of rough terrain vehicles that are capable of autonomous, semi-autonomous, or remotely-controlled operations in challenging terrains where wheeled and tracked vehicles reach their limits. In the future, legged vehicles will assist or replace humans in dangerous and dirty tasks. Quadruped robots are expected to operate in highly dynamic, unstructured outdoor areas where they will navigate inside challenging environments, such as collapsed buildings, disaster (natural and man-made) sites, forests, mountain farms, and construction sites. Their tasks will range from providing sensor streams to the remote operator (e.g., cameras, LIDAR, infrared, and radiation levels) to carrying heavy payloads such as tools or building materials. Some of the State of the Art legged robots include Handle (a), SpotMini (b), Atlas (c) and BigDog (d) shown in Figure 1-1, designed and manufactured by Boston Dynamics [17] .

(a)

(b)



(c)

(d)

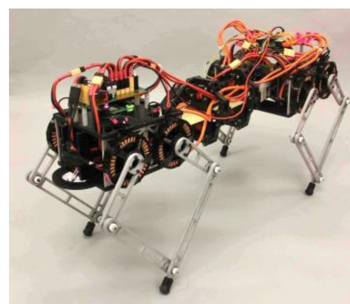
Figure 1-1. Boston Dynamics legged robots: (a) Handle (b) SpotMini (c) Atlas (d) BigDog.

ANYmal robot (a) from the Institute of Robotics and Intelligent Systems of ETH Zurich university [18] , MIT's Hermes (b) and Cheetah (c) robots by the Biomimetic Robotics Lab [19] and Upenn's Inu (d) robot by KOD*LAB [20] are also characteristic examples of legged robots developed within universities, as illustrated in Figure 1-2.

(a)

(b)

(c)



(d)

Figure 1-2. State of the Art legged robots: (a) ANYmal (b) Hermes (c) Cheetah (d) Inu.

Fieldbus and Industrial Ethernet

The first step in industrial automation for connecting multiple computational units was parallel wiring, where all participants were wired individually. However, the number of subscribers increased with the increasing degree of automation, which led to a high wiring expenditure. Now, parallel wiring has been widely replaced by cheaper and faster fieldbus systems and the Ethernet-based communication networks.

The Fieldbus systems [47] created in the 1980s 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. Even strong external loads cannot influence the robust digital signal transmission system. As the fieldbus communicates only via a cable, it has been possible to decrease the wiring considerably in comparison to parallel wiring.

A fieldbus functions in the so-called master-slave operation. While the master is in charge of control of the processes, the slave stations work the individual partial tasks. Fieldbuses differ according to their topology (star, line, tree or ring), their transmission medium, and – depending on the type – different transmission protocols (message-oriented procedure or summation frame procedure). The individual fieldbuses also differ in regard to the reachable cable length, the max. 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 each fieldbus. The most widespread examples of fieldbus technology are:

- Interbus: with transmission rates of up to 2 Mbps is characterised by especially 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. As the names already suggest, the remote bus serves to connect up to 254 subscribers which are located at large distances from each other. On the other hand, the local bus connects subscribers that are located close to each other to the system.
- 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. It has a hierarchical structure with the sensors/actuators levels, field levels and the process level. In master-slave operation, the token passing access procedure is used. Here, slaves may only access the profibus upon the master’s request.
- Foundation H1: is a bi-directional communications protocol (31.25 kbit/s) used for communications among field devices and to the control system. It utilizes either twisted pair, or fiber media to communicate between multiple nodes (devices) and the controller. The controller requires only one communication point to communicate with up to 32 nodes, this is a significant improvement over the standard 4-20 mA communication method which requires a separate connection point for each communication device on the controller system

On the contrary, Industrial Ethernet [48] is without a doubt very well established in automation technology, although traditional fieldbus technology still has a long way to go before reaching retirement. Since modern machines and systems must perform increasingly complex tasks, data networks are growing

ever 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 [49] . More than 20 different protocols compete in various segments of this rapidly growing market, each offering adaptations to meet different real-time and cost challenges, such as:

- Profinet 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.
- EtherCAT (Ethernet for Control Automation Technology) originally developed by Beckhoff, provides real-time performance and supports various topologies with twisted pair and fiber optic media.
- EtherNet/IP (IP for “industrial protocol”) is supported by Rockwell Automation-affiliated organizations, ControlNet International (CI) and Open DeviceNet Vendors Association (ODVA).
- Modbus-TCP allows the widely used Modbus protocol to be carried over standard Ethernet networks on TCP/IP.
- Ethernet Powerlink combines CANopen and offers deterministic real-time operation.

For application in Laelaps II quadruped, EtherCAT was selected as the technology mostly used in robotics nowadays, because of its high performance in terms of bandwidth and speed, its high determinism, its convenient slave-synchronization capabilities and the reduced equipment costs. In addition, in EtherCAT there is no need to set device addresses, and also its diagnostic capabilities make the process of finding the sources of malfunctions and troubleshooting substantially easier.

EtherCAT in robotic applications

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

In the industrial manufacturing sector, KUKA Robotics [21] 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 [22] has developed a wide range of EtherCAT based robotic solutions such as MiniBOT Robot (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)

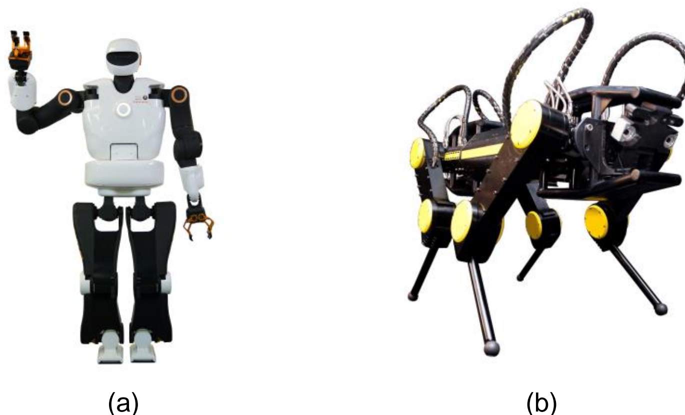
(b)

Figure 1-3. (a) KR C4 Controller with robotic arm by KUKA and (b) MiniBOT Robot by NexCom.

In the haptic – soft robotics and manipulation field, Shadow Robot Company [23] exploited EtherCAT technology to develop a truly anthropomorphic hand, Shadow Dexterous Hand (Figure 1-4), with 20 actuated degrees of freedom, absolute position and force sensors, and ultra sensitive touch sensors on the fingertips, providing unique capabilities for problems that require high precision.

Figure 1-4. Shadow Dexterous Hand by Shadow Rob Company.

In the field of legged robotics, PAL Robotics [24] has designed TALOS (Figure 1-5 (a)), a fully electrical humanoid biped robot that uses torque control in all its joints and EtherCAT to tackle complex industrial tasks with 6 Kg payload capability in each arm. Similarly, the Department of Advanced Robotics of the Italian Istitute of Technology (IIT) [25] has exploited EtherCAT to design and build HyQ2Max quadruped robot (Figure 1-5 (b)) which mimics the robustness and versatility of animals in challenging terrains.



(a) (b)

Figure 1-5. (a) Talos biped robot by PAL Robotics and (b) HyQ2Max quadruped robot by IIT.

1.3 Thesis Outline

Along with the first introductory chapter, where the motivation of this work and the literature review is presented, the thesis is structured in five chapters.

In the second chapter, a brief description of EtherCAT techonology is presented along with the three synchronization solutions that can be configured. Then, the process data handling is explained, followed by a detailed solution reference guide describing the configuration procedure of an EtherCAT network of microcontroller units (MCUs) with a user defined application.

In the third chapter, a short description of Laelaps II quadruped is introduced, followed by an extensive solution reference guide explaining the implementation process of Laelaps II control architecture. The firmware developed for the slaves is described and the most significant points of interest are mentioned, providing the ability to future users to manipulate and expand the application according to their needs.

In the fourth chapter, an experimental validation of the robot is presented containing the results of locomotion experiments executed with Laelaps II using a PC running TwinCAT XAE as the EtherCAT master node. A description of the testing procedure is entailed along with a table containing the definition of all parameters used in the experiments.

In the last chapter, the conclusions of the thesis are summarized and future work is suggested.

2 EtherCAT Communication and Implementation

2.1 EtherCAT Technology

2.1.1 Introduction

Ethernet for Control Automation Technology or EtherCAT [26] is a high performance Ethernet Based fieldbus system. The main reason for its development was the adoption of Ethernet in automation applications, where short cycle times and low communication jitters are required [4] .

EtherCAT is based on a master-slave approach and relies on a ring topology at the physical level. Only one master is allowed in the network, and this is suitable, for instance, to connect a control unit (e.g., a programmable logic controller [PLC]) to decentralized peripherals (sensors, actuators, drives, microcontrollers etc.). By using suitable gateways, EtherCAT can interoperate with both conventional Transmission Control Protocol (TCP)/Internet Protocol (IP)-based networks (intranets) and other realtime Ethernet (RTE) solutions, such as EtherNet/IP and PROFINET.

The master node is in complete control of the traffic exchanged over the EtherCAT network. In particular, it is the only device that can take the initiative in the communication; hence, it is responsible for initiating all data exchanges with the slaves. Each slave processes the received frame (patch of information in a specific format) in order to extract from and insert data into it (Figure 2-1). Then, the frame is forwarded to the next slave in the ring.

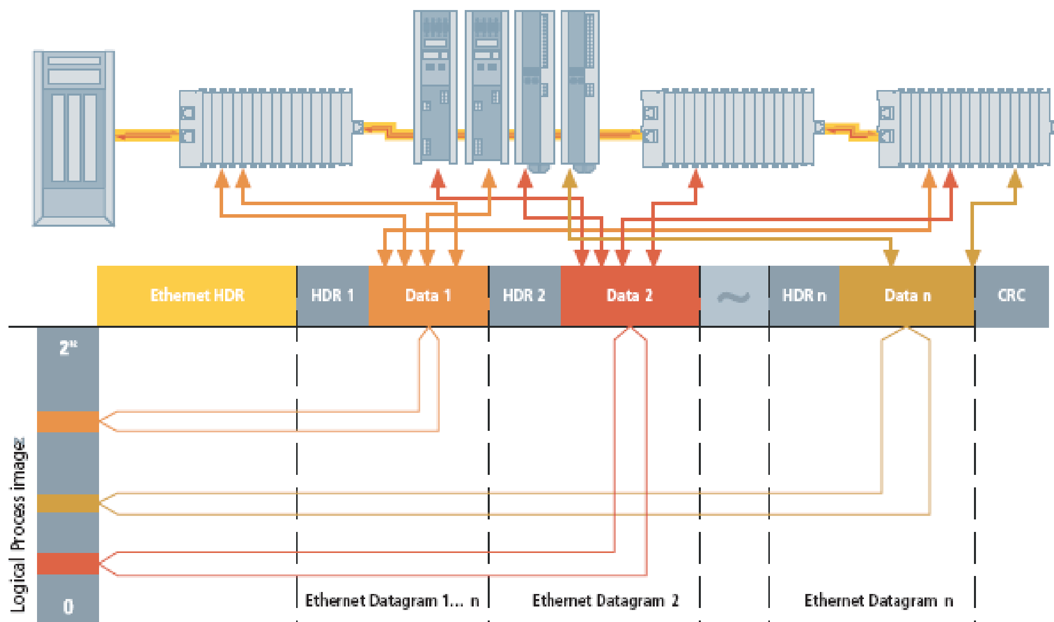


Figure 2-1. EtherCAT Topology.

2.1.2 Physical Layer

Unlike Ethernet switches and bridges, slaves do not manage frames according to a conventional store-and-forward approach, which implies receiving the frame, decoding the related protocol control information, and

sending the message out. Every frame, instead, is processed on-the-fly by the slave data link layer therefore achieveing higher cycle frame speed. In order to ensure high performance, frame processing and relaying take place at the same time so that these operations have to be carried out in hardware. This explains why specialized components are used for slaves, which are known as EtherCAT Slave Controllers (ESCs).

Communication Support

The physical layer of EtherCAT relies on the proven fast Ethernet transmission technology, which enables high data rates. Although the use of switches is not recommended to ensure real-time behavior, the EtherCAT network architecture is quite flexible and can also stretch over wide areas.

Though EtherCAT is correctly listed among the industrial Ethernet solutions available today, it actually supports two different types of physical layers, namely, Ethernet and EBUS. In our application that will be described thoroughly later on, the Ethernet physical layer is used.

Ethernet relies on the conventional 100 Mb/s full-duplex Ethernet technology and it is typically used for connecting the master to the network segment to which slaves are attached. Indeed, the entire EtherCAT segment is seen by the master as a single, large Ethernet device which concurrently receives and sends Ethernet frames by exploiting full-duplex transmissions. However, this device does not consist of a single Ethernet controller but includes a (possibly very large) number of EtherCAT slaves connected so as to form a ring topology. The transmission medium, in this case, consists of a Cat 5 twisted pair (either shielded or unshielded, depending on the amount of electromagnetic interference) although higher category cables are also allowed. Both classic RJ45 (8P8C) and circular M12 D-code connectors can be used.

EBUS can be used only as a backplane bus and is not intended for wire connections. EBUS, in fact, was mainly conceived to interconnect modules in modular devices. Unlike other fieldbuses that enable a modular design for devices, the sequence of logical bits that EtherCAT transmits over EBUS is exactly the same as for Ethernet. This means that switching from Ethernet to EBUS (and vice versa) can be done quickly, efficiently, and inexpensively (in practice, only transceivers have to be replaced). It is an inexpensive physical layer that features reduced pass-through delays inside the slaves.Typically, frames experience delays on the order of 120 ÷ 500 ns when propagating through EBUS interfaces, whereas longer latencies (about 1 μs) are introduced by Ethernet interfaces.

Network Topology

The star topology, commonly used for switched Ethernet networks implies significant cabling and infrastructure costs; hence, line or tree topologies, which are commonly used in EtherCAT applications, are usually preferable in factory and automation networks.

Typically, slave devices in EtherCAT segments are connected in linear structures and exploit a daisy chain wiring scheme (Figure 2-2). Every slave is provided with (at least) two Ethernet ports, to connect downstream and upstream devices. The last slave in the segment performs a loopback function and returns the frame in the opposite direction to the master without any additional wiring. The master is the headend of the structure and requires one Ethernet port only. Each slave relays all frames it receives to the next device in the EtherCAT segment.

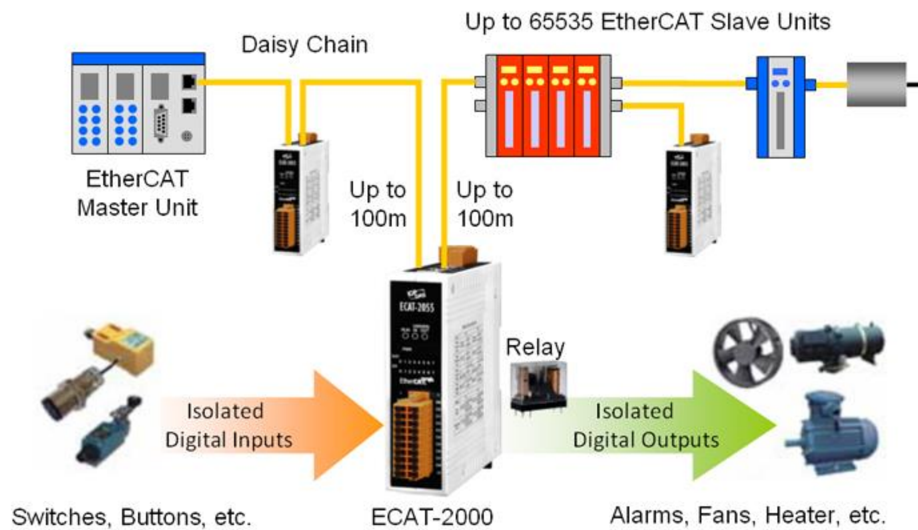


Figure 2-2. A typical EtherCAT network.

Because of the daisy chain connection and loopback, all the slaves in the segment form an open ring (line). The master transmits frames at one of the ends of this open ring and receives them at the other end, after they have been processed by every slave. This means that on the whole the physical topology of an EtherCAT network is actually a ring. Thanks to the full-duplex capabilities of Ethernet, which uses two pairs of wires housed in the same cable to carry out communications in both directions simultaneously, the resulting topology resembles, nevertheless, a physical line, as in most legacy fieldbuses. The reduction of wiring complexity helps in making the network deployment easier and lowers the installation costs at the same time. In principle, branches as shown in Figure 2-3 can be introduced anywhere in an EtherCAT segment, by using devices equipped with three or more ports (EtherCAT couplers).

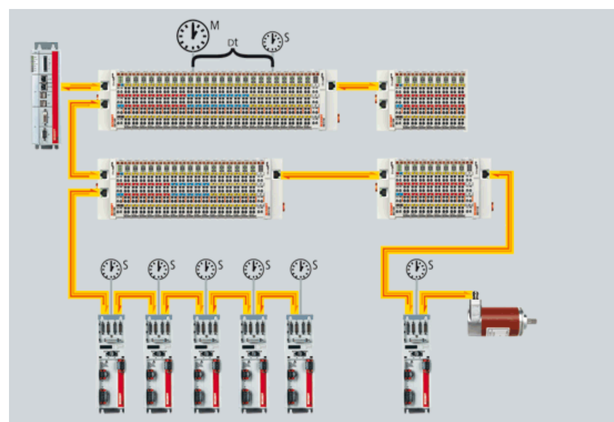


Figure 2-3. EtherCAT topology with branches.

This kind of devices are provided, for example, with two Ethernet ports and one EBUS interface for direct connection of input/output (I/O) modules (also known as EtherCAT terminals) and can be used to enhance the basic line structure setting arbitrarily complex tree networks topologies. It is worth noting that, in this case, every slave device located at the end of a branch has to close the ring on its own using the loopback function.

The maximum number of addressable devices in EtherCAT is quite large, since 216 nodes for each segment are allowed. The limit depends on the data link layer, and in particular on the address field, which is encoded in 16 bits. In the same way, the maximum network extension is usually able to satisfy the

requirements of most real applications. The limitation, in this case, is mainly due to the maximum distance allowed between any two adjacent nodes (i.e., the length of the cable), which in turn depends on the underlying transmission support (up to 10 m for EBUS and up to 100 m for Ethernet connections). This means that the whole network size is practically unlimited; in theory, up to 216 devices can be connected in daisy chain using 100 m Ethernet cable segments.

Device Architecture

EtherCAT masters (EMs) rely on standard communication hardware (full-duplex Ethernet network interface controllers) and dedicated software, with open-source solutions based on Linux-like operating systems also available. On the contrary, purposely designed hardware components (ESCs-EtherCat Slave Controllers) are indispencable for slave configuration.

In order to reduce the implementation costs, frame processing by ESCs occurs in one direction only, which is known as the *processing path*. The reverse direction, known as *forwarding path*, is needed to propagate frames in the ring back to master. From a logical point of view, ESCs exhibit an active behavior only on the processing path (frame modifications on the forwarding path are not allowed), but at the physical layer, they behave as repeaters in both directions. Consequently, they are able to regenerate electrical signals so that network equipment like stand-alone repeaters is no longer necessary. This also reduces connection costs and complexity in large installations. Most ESCs are internally equipped with two or more ports, depending on device complexity. For example, the Beckhoff ET1100 is provided with four separate ports, which can be individually configured to operate as either EBUS or MII. Port 0 is the upstream port whereas the others are used for downstream connections and to forward signals. Each port implements two functions, called auto-forwarder and loopback. The auto-forwarder block performs frame checks, such as CRC error detection at the physical level and manages the error count. It is also responsible for taking timestamps on frame receptions, a mechanism which is needed, for instance, by the clock synchronization protocol. The loopback function, instead, forwards frames to the next logical port if the related link is not available. In this way, the ring is automatically closed in the case of faults affecting either devices or links.

2.1.3 Data Link Layer

The data link protocol of EtherCAT was designed to maximize the utilization of the Ethernet bandwidth and to grant a very high communication efficiency. As mentioned earlier, the access mechanism of EtherCAT is based on a master/slave approach, where the master node (typically the control unit, e.g., a PLC) sends Ethernet frames to slave nodes. Slaves, in their turn, either extract data from the frame payload or insert information by overwriting part(s) of the payload itself.

Frame Format

Messages sent over the network are standard Ethernet frames, with EtherCAT frames (also known as *Type 12 frames*) encapsulated in the *data field* (payload). Consequently, they include the conventional fields (Figure 2-4):

- preamble (8 bytes)
- 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)
- interframe gap

An EtherCAT frame, in turn, contains:

- A frame header (2 bytes)
- One or more EtherCAT datagrams, also known as Type 12 Data Link Protocol Data Units (DLPDU) according to the data link layer standard specifications

In this way, the large data field made available by conventional Ethernet can be better exploited to increase the communication efficiency. DLPDUs are packed together, one after the other, without intermediate gaps. The payload of the Ethernet frame ends with the last DLPDU, unless its overall size is 63 octets or less. In this case, the frame is padded to 64 octets in length, as required by the Ethernet specifications. The standard Ethernet CRC closes the frame and is used by each device (either master or slave) to check the integrity of the message. Thanks to the EtherType field, EtherCAT can coexist, in theory, with other Ethernet protocols.

Note 1: The octet is a unit of digital information in computing and telecommunications that consists of eight bits. The term is often used when the term byte might be ambiguous, as the byte has historically been used for storage units of a variety of sizes.

Note 2: A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data.

Each DLPDU corresponds to a separate EtherCAT command and consists of three sections: header, data, and counter field. Commands are used to perform data exchanges: basically they are issued by the master for reading or writing specific memory areas in the slave devices. Ethernet frames, and in particular the DLPDUs they embed, are processed in sequence by the slaves. Each slave recognizes its commands of interest and executes them while the frames are passing through. Because of the physical ring topology, a frame is returned to the master after being processed by all the slaves. This procedure exploits the full-duplex mode of Ethernet, which means that the two communication directions can work independently. Several DLPDUs can be embedded in the same Ethernet frame, each one addressing different devices and/or memory areas. As shown in Figure 2-4, DLPDUs are transported either

(a) directly in the data field of the Ethernet frame (used in our application) or
(b) within the data section of a datagram, by means of the User Datagram Protocol (UDP).

The first variant (a) is limited to a single subnetwork, since Ethernet frames are not relayed by routers. Usually, this is not a limitation for machine control applications. Direct Ethernet encapsulation is by far the most widespread EtherCAT solution at the shop floor of factory automation systems. In theory, multiple EtherCAT segments can be connected to a single master through one or more switches, and the MAC address of the first node in each segment is used for addressing the segment itself. However, this approach can affect the real-time properties of the communication.

On the one hand, the second variant (b), which relies on UDP and the IP, implies lightly larger overheads (because of the IP and UDP headers) and is also limited by switches, which can easily add nondeterministic characteristics to the communication [27] . On the other hand, this solution also enables IP routing; hence, it is suitable for applications having loose timing requirements, such as in process automation. Any standard UDP/IP implementation can be used in this case on the master side.

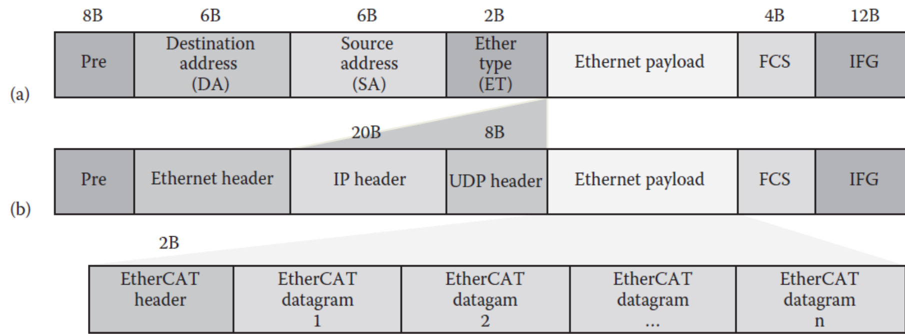


Figure 2-4. EtherCAT Frame Structure with EtherCAT Datagrams (a) directly in the data field of the Ethernet frame (b) within the data section of a datagram, by means of the User Datagram Protocol (UDP).

EtherCAT Datagram (or DLPDU) Format

As shown in Figure 2-5, each DLPDU (or EtherCAT Datagrams) consists of a number of fields. The initial fields (up to IRQ included) can be assumed to belong to the header part, which has a fixed size (10 bytes). The variable-sized data area is placed immediately after the header and includes the information to be exchanged, often referred to as *data link service data unit* (DLSDU). The last field in the frame is the *working counter* (WKC), used mainly for checking whether a command has been successfully executed by the relevant slaves.

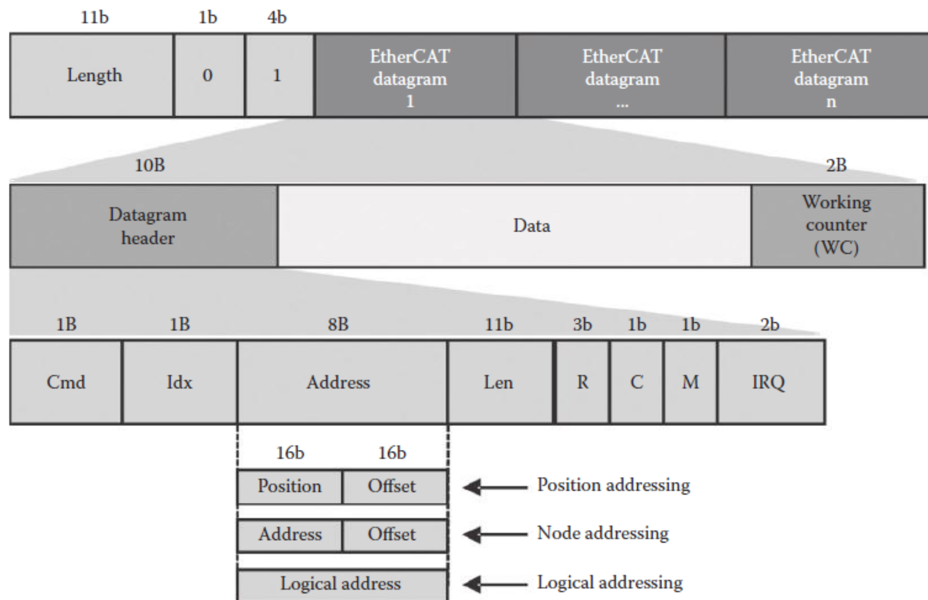


Figure 2-5. EtherCAT Datagram (or DLPDU) structure.

The service command (1 byte) is encoded in the CMD parameter. Different types of command exist, which can be used to carry out highly optimized read and write operations on slave devices [4] . Generally speaking, they can be grouped according to the access type:

- Read (RD) is used by the master to read memory areas or registers from slave devices.
- Write (WR) is used by the master to write to memory areas or registers of slave devices.

- 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.
- Read/Multiple Write (RMW) is a quite peculiar service, where the addressed slave carries out a read operation while all other slaves are performing a write action.

SyncManager

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

EtherCAT provides a mechanism for slave memory access control, which is based on SyncManagers, and was designed bearing in mind concurrency issues. SyncManagers are implemented in hardware in the ESC and enable consistent and secure data exchanges between the EM and the local application, together with the interrupt generation to notify both sides of changes. SyncManagers are configured by the EM. The communication direction can be selected, as well as the communication mode. Each SyncManager uses a buffer in the local memory area for exchanging data and transparently controls all accesses to the buffer. The buffer must be accessed beginning with the start address; otherwise, the access is denied. Once access to the start location is granted, the whole buffer can be accessed, either as a whole or in a number of strokes. Accessing the last location also concludes the whole operation. Buffer changes caused by the master are accepted by the SyncManager only if the frame FCS is correct. This also means that such buffer changes take effect immediately after the reception of the end of the frame.

SyncManagers support two communication modes:

1. Buffered mode: In this case, the interaction between the producer and the consumer of data is uncorrelated, and each entity can access the buffer at any time. The consumer is always provided with the newest data. In the case data are written into the buffer faster than they are read out, old data are simply discarded. The buffered mode is typically used for cyclic process data. This mechanism is also known as 3-buffer mode, because the SyncManager manages three buffers of identical size (denoted as 0, 1, and 2). One buffer is allocated to the producer (for writing), another buffer to the consumer (for reading), and a third buffer helps as intermediate storage. Reading or writing the last byte of the buffer results in an automatic buffer exchange. It is worth noting that both the EM and the local application must always refer to buffer 0 when accessing memory. It is up to the SyncManager redirecting accesses to the right buffer.

2. Mailbox mode: In this case, a handshake mechanism is implemented for data exchanges, which prevents buffer overwriting and ensures that no data will be lost. Just one buffer is allocated for each mailbox; moreover, reading and writing are enabled alternatively. The mechanism implemented by mailboxes is straightforward. At first the producer writes to the mailbox buffer. When done, the SyncManager locks it for writing and enables read access to the consumer. Only when the consumer has finished reading data out of the buffer, the producer is granted write access again. At the same time, the mailbox turns to the locked state

for the consumer. The mailbox mode is typically used for *application layer* (AL) protocols, where the time taken to exchange information typically is not very relevant.

2.1.4 Application Layer

The AL of EtherCAT implements a state machine, which describes the behavior of a device by means of its states and events that trigger transitions between states. In particular, the state machine is responsible for coordinating master and slave applications during the start-up and operational phases. Depending on the current state, different functions are enabled in the EtherCAT slave. Different commands have to be sent to the device in each state by the EM, in particular, during the boot sequence of the slave. Commands are acknowledged by the local application after the involved operations have been completed. Unsolicited changes of the local application state are also possible. Moreover, simpler devices, which do not include a microcontroller, can be configured to follow the state machine logic through an emulation mechanism. In this case, any state change has to be accepted and acknowledged.

The state machine is controlled and monitored using some registers included in the slave. The master controls the state transitions by writing to the AL control register. In turn, the slave updates information about its current state by writing in the AL status register, which is also used for error notification by means of suitable error codes written in the register itself. As Figure 2-6 shows, an EtherCAT slave supports four basic states and, possibly, one optional state:

- *Init*: EtherCAT slaves enter this state at power-on. In this situation, the master initializes the SyncManager channels for mailbox communications.
- *Preoperational*: Mailbox communications are enabled in the preoperational state, but process data communications are not. The EM initializes the SyncManager channels for process data, the Fieldbus Memory Management Unit (FMMUs) and the Process Data Objects (PDOs) mapping mechanism, if supported.
- *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.
- *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 sequence.
- *Bootstrap* (optional): The bootstrap state is mainly aimed at downloading the device firmware. In the bootstrap state, mailboxes are active but restricted to file access via EtherCAT services.

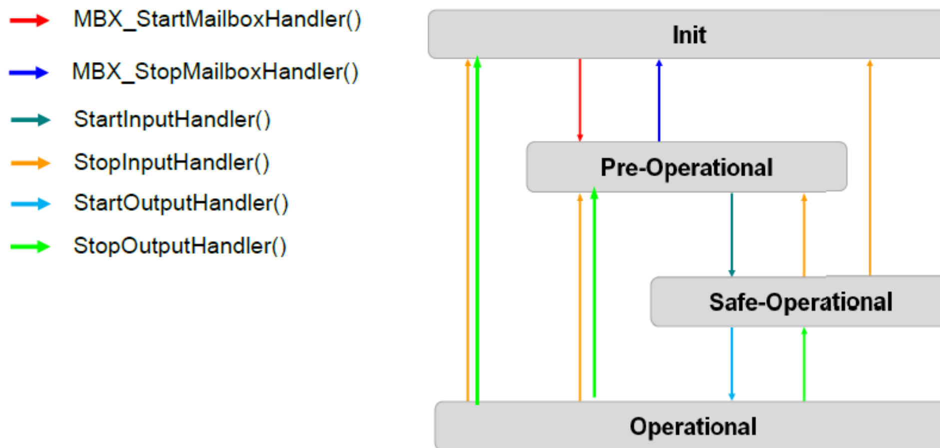


Figure 2-6. EtherCAT Application Layer State Machine.

Application Protocols

An important characteristic of EtherCAT is its ability to support multiprotocol higher-level communications using standardized mailboxes. This aspect is particularly appealing when options are offered for popular solutions such as the following:

- *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 to tunneling standard Ethernet Frames in EtherCAT.
- *File access over EtherCAT* **(FoE)**: This option enables the download/upload of firmware and other files.
- *Servo drive profile over EtherCAT* **(SoE)**: This option is useful to grant access to the device profile of SERCOS.

Supporting popular communication protocols helps in improving compatibility and efficiency of data exchanges between new and old components in automation systems; to this purpose, EtherCAT makes use of well-known and established technologies. For instance, the *CoE* protocol enables the adoption of the complete CANopen profile family in EtherCAT networks. Besides this feature, the service data object (SDO) transport protocol allows the transmission of objects of any size and is equivalent to its CANopen counterpart so that it is possible to reuse existing protocol stacks. Data are organized in process data objects (PDOs), which are transferred using the efficient support of EtherCAT. Moreover, an enhanced mode is defined that overcomes the 8 byte limitation of CAN and enables the readability of the whole object list.

Another appealing feature an industrial Ethernet solution should provide is the support to standard IP-based communication protocols (i.e., TCP/IP and UDP/IP) and all higher-level protocols that rely on them, such as HTTP, FTP, SNMP, etc. To this purpose, the *EoE* feature exploits a mechanism where Ethernet datagrams are tunneled and reassembled in a device, before being relayed as complete Ethernet frames. This procedure has no impact on the achievable cycle time, because the size of fragments can be optimized according to the available bandwidth.

The *FoE* is an EtherCAT service that can be used to download a file from a client to a server or to upload it in the opposite direction. The protocol is similar to the trivial file transfer protocol (TFTP), and both sides are

allowed to initiate a read or write request via the corresponding command. This service is typically used to update the device firmware.

Finally, the *Servo drive profile over EtherCAT* (SoE) service enables the use of the SERCOS device profile and is suitable for demanding applications that rely on popular drive technology.

2.2 EtherCAT Synchronization

2.2.1 Synchronization Overview

One of the most significant and crucial features of EtherCAT technology is the fact that it enables automatic synchronization between the master and all the slaves connected to the network, providing a universal clock that all components adhere to. Moreover, another facilitating feature of EtherCAT technology is its flexibility on accounts that each slave can be configured to its own synchronization mode without being affected by the rest slaves that are connected to the network. At application level, both the master and the slave device consist of cyclically executed software code (Figure 2-7).

Figure 2-7. EtherCAT Application Level.

The master and each slave application cyclically exchange process data in both directions in predefined cycle times (Figure 2-8). These intervals can be really short as long as both the master and slave application have enough time to execute their own stack.

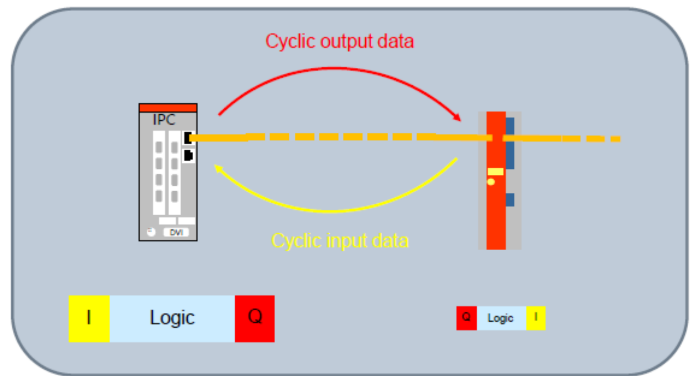


Figure 2-8. EtherCAT process data exchange.

Synchronizing master and slave applications basically means defining a **time relationship** between the start time of the cyclic code handling process in master and slave alike as shown in Figure 2-9.

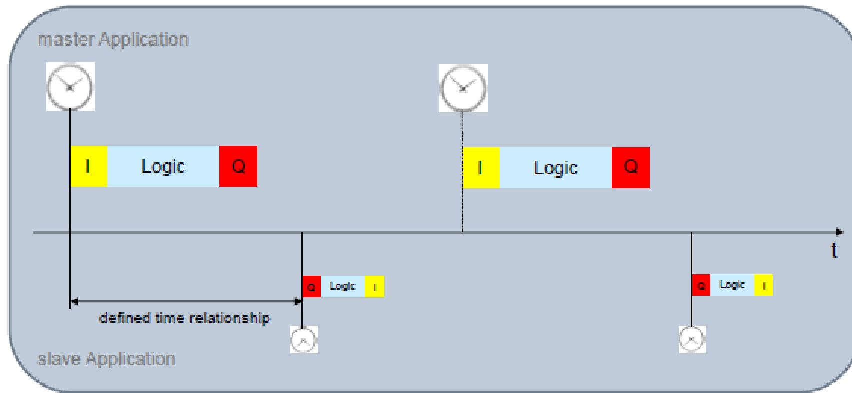


Figure 2-9. EtherCAT Synchronization.

EtherCat defines **three** main time relationships of each slave application with respect to the master cycle (**Synchronization Modes**):

- **Free Run** (no synchronization): process data handling in the slave is initiated by an internal event having no defined time relation with the master cycle.
- **SM-Synchronous (Sync Manager)**: process data handling in the slave is initiated 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 initiated by a hardware interrupt event based on the Distributed Clocks and on the corresponding System Time.

2.2.2 Free Run Mode

In Free Run Mode (Figure 2-10), the process data handling in the slave is triggered by an internal event and the communication scheme is described by the following characteristics and Figure 2-11:

- No defined relationship between cyclic frames and local application
- Time offset among different "Free Run" slaves is undefined
- Intended for I/O (input/Output) devices handling slow-varying signals

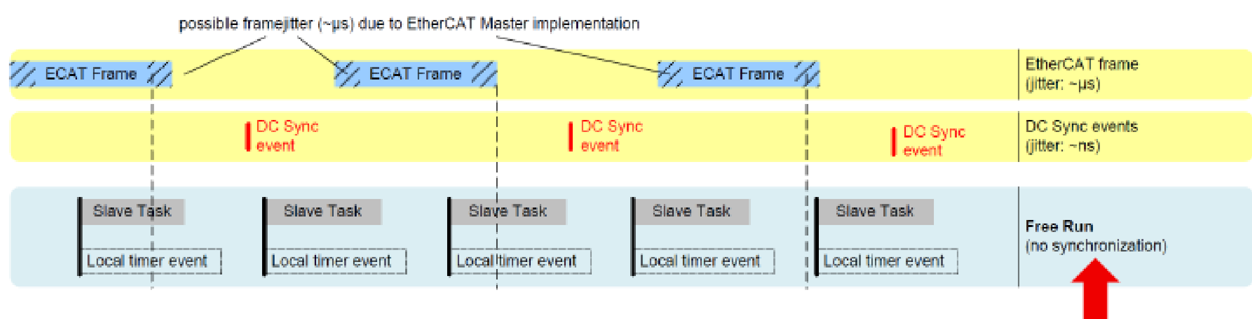


Figure 2-10. Slave in Free Run mode.

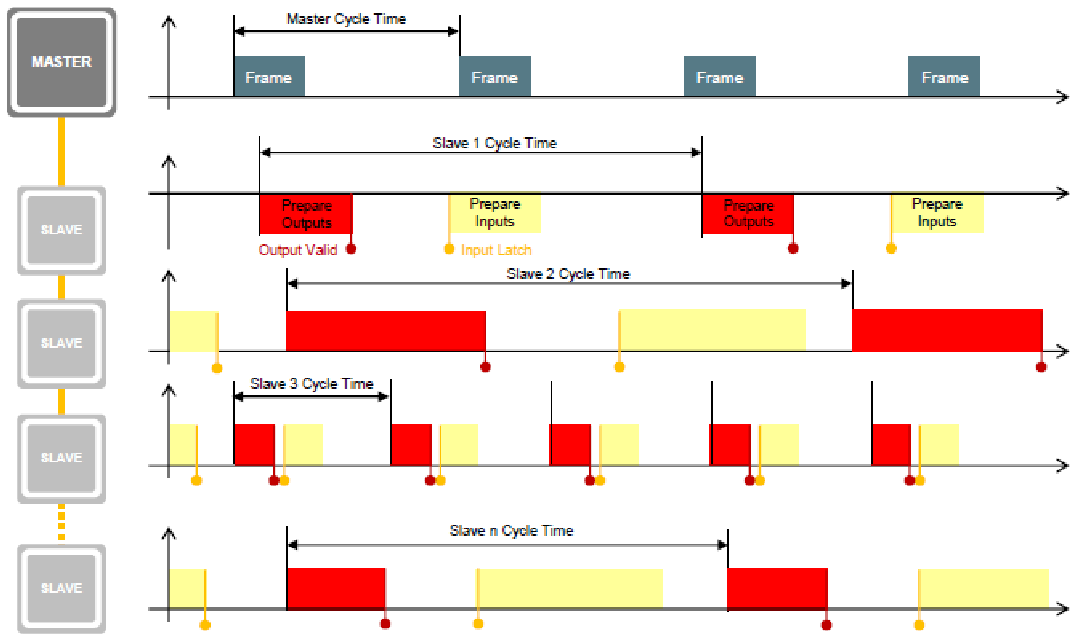


Figure 2-11. EtherCAT network in Free Run mode.

2.2.3 SM-Synchronous Mode

In SM Synchronous Mode (Figure 2-12), the process data handling in the slave is triggered by one interrupt signal when the cyclic frames are received as shown in Figure 2-13. Hence, the master of the EtherCAT network is obliged to provide a timer variable to each slave so that they can all be synchronized to a universal clock. This requirement becomes a necessity in decentralized robotic applications where each slave must obey a universal clock in order to produce smooth and continuous locomotions.

There are several possible causes of synchronizations inaccuracies if a network is configured in SM Synchronous mode which may affect the efficiency of the synchronization:

- Cyclic frames are received by slaves with the same jitter which affects the master in sending them
- Even with no jitter, due to finite hardware propagation delays the last slaves will receive the cyclic frames later with respect to the first ones

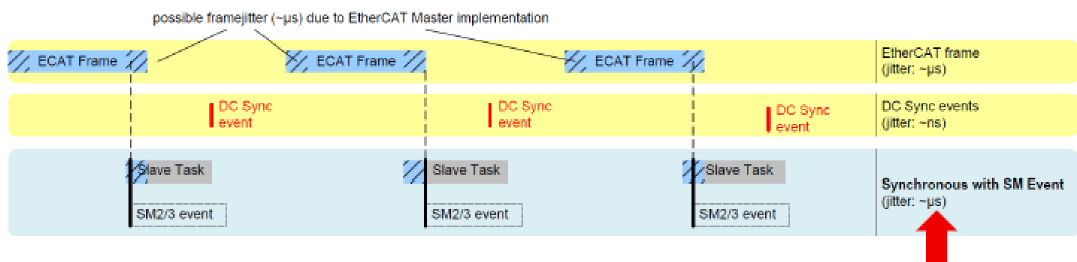


Figure 2-12. Slave in SM Synchronous mode.

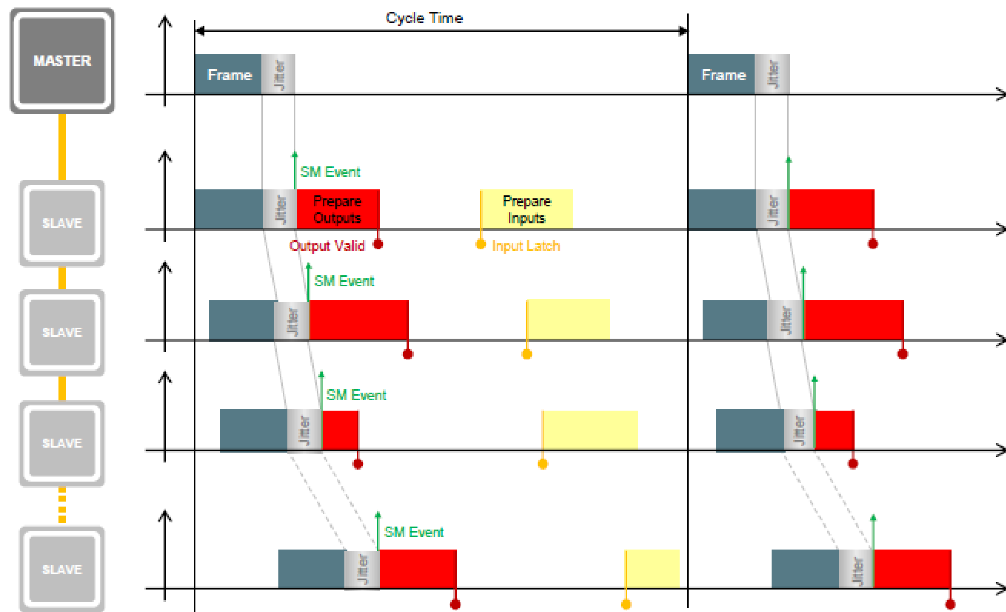


Figure 2-13. EtherCAT network in SM Synchronous mode.

2.2.4 DC-Synchronous Mode

An important mechanism included in the EtherCAT specification is the distributed clocks (DC) synchronization protocol, which enables all slave devices to share the same system time with high precision and accuracy. Synchronization errors are typically well below 1 μs; in this way, all devices can be synchronized, and consequently, distributed applications are synchronized as well (Figure 2-14). Possibly, the master can also be synchronized, even though this option requires additional capabilities.

Main Features

The DC mechanism provides a number of features that are very useful for distributed control applications, the most important being

- Synchronization of the slaves (and the master) clocks
- Generation of synchronous output signals (*SyncSignals*)
- Precise timestamping of input events (*LatchSignals*)
- Generation of synchronous interrupts
- Synchronous digital output updates
- Synchronous digital input sampling

DC is placed above the EtherCAT data link protocol, and its implementation is not mandatory. For this reason, both DC-enabled and non-DC-enabled devices can quietly coexist in the same network. It is worth noting that DC is not a general-purpose synchronization protocol, since it relies on specific features of EtherCAT, such as its ring topology, on-the-fly datagram processing, and hardware timestamping capabilities.

DC Mechanism

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

1. *Propagation delay measurement*: The master sends a synchronization DLPDU at certain time intervals, and each slave stores the time of its local clock; after collecting all timestamps, the master, which is aware of the network topology, computes the propagation delay for each segment.
2. *Offset compensation*: Because the local time of each device is a free-running counter, which typically does not have the same value as the reference clock, the master computes the offset between the reference and local clocks separately for each DC-enabled slave. Then, the offset is written to a specific register of the slave, in order to compensate differences individually. At the end of this step, all devices share the same absolute system time.
3. *Drift compensation*: After propagation delays have been measured and the offsets between clocks compensated, the drift of every local clock is corrected through a time control loop (TCL). This mechanism readjusts the local clock by regularly measuring its difference with the reference clock.

In DC Synchronous Mode, the process data handling in the slave is triggered by the hardware SYNC events generated in the slave based on the DC System Time as shown in Figure 2-15. These interrupt signals, the number of which may vary from one to three depending on the requirements of the application, ensure that the Interrupt Service Routine (ISR) configured for each of these channels will be triggered simultaneously in every slave connected to the network, therefore providing intrinsic synchronization among the slave devices without any timer variable required. However, developers should be really cautious when defining the cycle frame time owing to the fact that this interval should be wide enough to allow all ISRs to be excuted by every slave. In any other case, recurrent lost frames might interfere with the internal synchronization of the slave devices, causing errors in communication level.

The most significant advantages of DC Synchronous mode is that:

- Hardware SYNC events (interrupt signals) are generated within each slave automatically by the EtherCAT Slave Controller who must be configured to operate in DC Synch mode (specified in the ENI file)
- The triggering event in each slave is not affected by master jitter or propagation delays

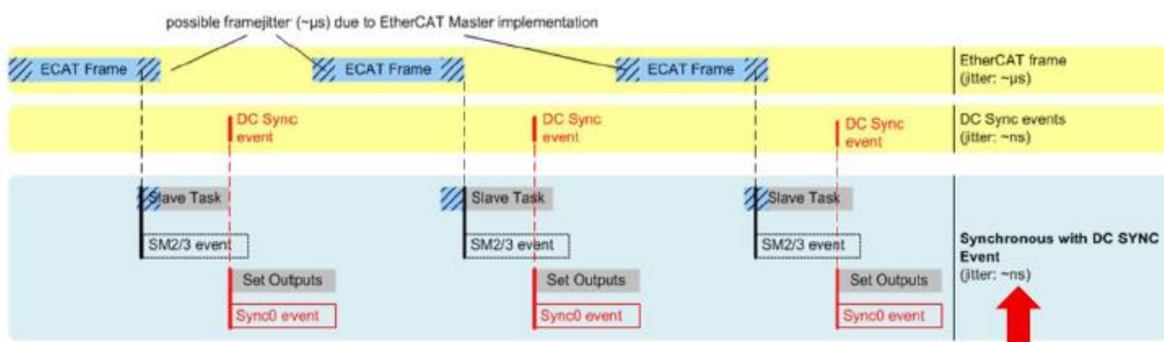


Figure 2-14. Slave in DC Synchronous mode.

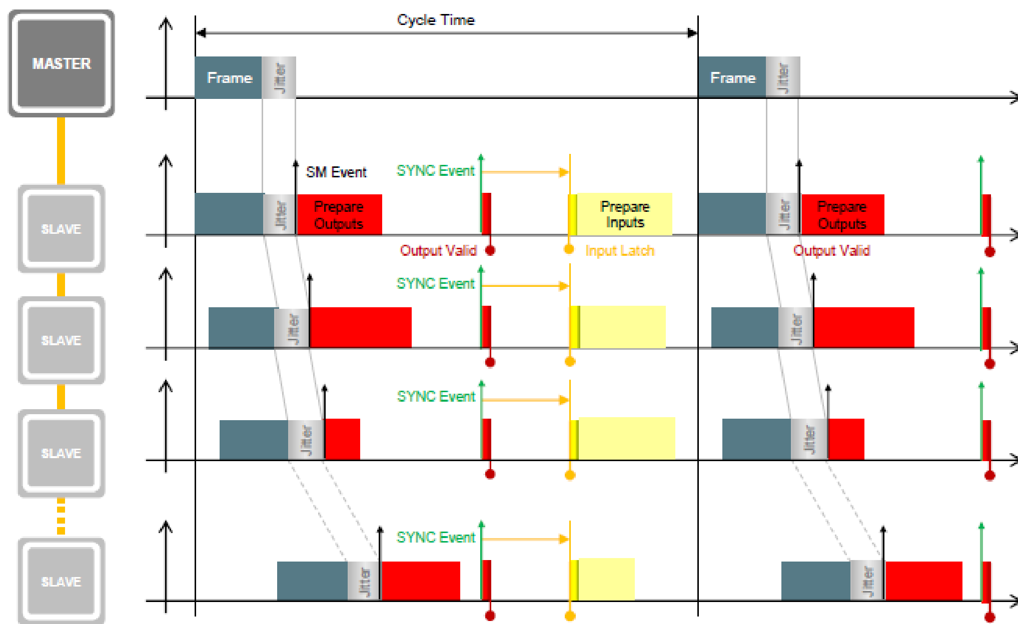


Figure 2-15. EtherCAT network in DC Synchronous mode.

In **SM** and **DC Synchronous** mode, a certain shift is always needed between the master and slave application times, in order to enable the communication partner to receive the data before its cycle begins (Figure 2-16).

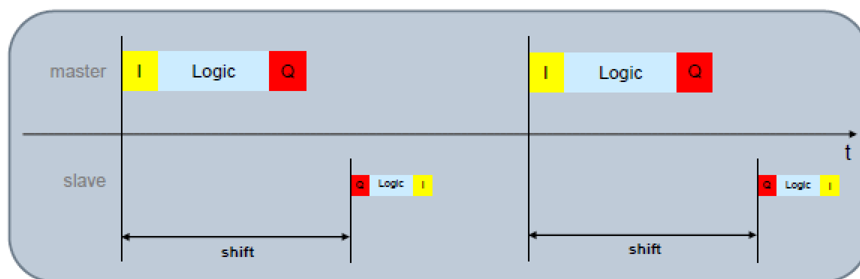


Figure 2-16. EtherCAT shift times.

In **SM-Synchronous** mode, the shift is set by the synchronization mode itself (no parameter configuration needed), as the slave application is directly triggered by the cyclic frame. On the other hand, in **DC-Synchronous** mode, the shift between the SM interrupt and the master cycle is set by the master during the start-up phase, and can be changed by users if needed. A proper setting of the **time shift** in DC-Synchronous mode shall guarantee that the SYNC event within the slave is generated after the cyclic frame delivering outputs was received by every slave and before the next cyclic frame collecting inputs is received by the slave despite communication jitter, propagation delays and number of slaves (Figure 2-17). Admittedly, there is not only one correct value, yet an entire interval of possible values for the time shifts.

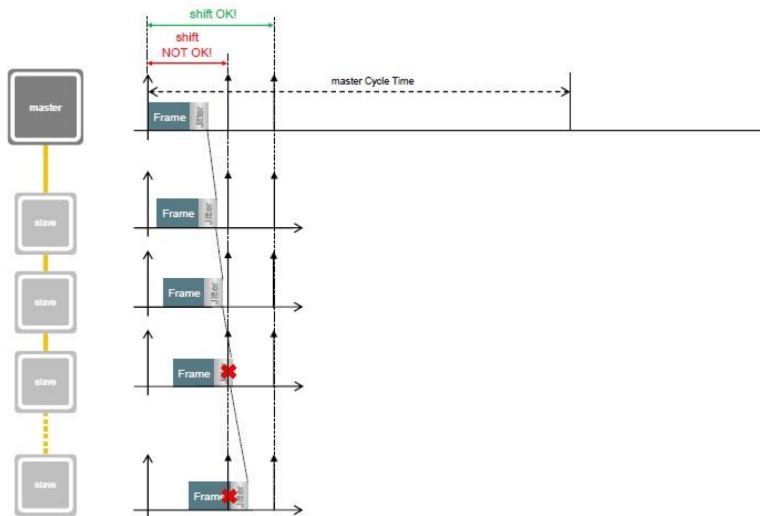


Figure 2-17. EtherCAT time shifts.

An estimation of the minimum value for the **SYNC Shift for Outputs** can be obtained as algebraic sum of the following contributions:

- Hardware delay introduced by the slaves internally:
  - 1 µs for every slave of the network with MII Ports
  - 3 µs for every slave of the network with only EBUS Ports
- Hardware delay introduced by the cables which is 5,3 ns for every meter of the length of the copper cables in the network

2.3 Process Data Handling

The EtherCAT slave process data communication can be separated in two main steps as depicted in Figure 2-18:

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

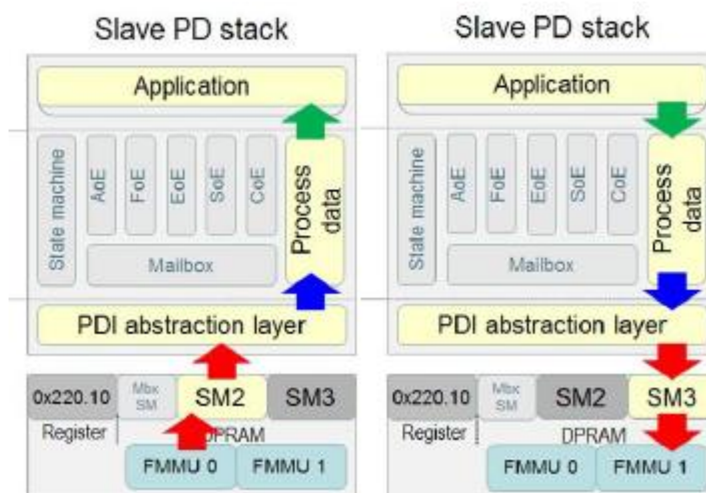


Figure 2-18. EtherCAT Process Data handling.

The process data handling in the SSC is managed in three functions of the generic stack as depicted in Figure 2-19. Each of these functions triggers the corresponding application specific functions.

1. *PDO_OutputMapping()*: handles the data from the master to the slave
2. *ECAT_Application()*: contains the slave application written by the user
3. *PDO_InputMapping()*: handles the data from the slave to the master

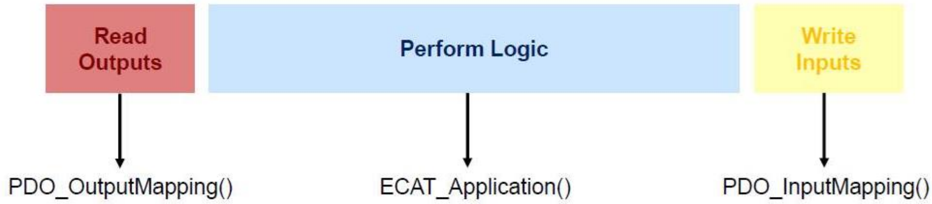


Figure 2-19. Process data handling generic functions.

The handling and sequence of the application for the different synchronization modes are described in the following figures. As mentioned above, in Free Run Mode there is no synchronization and all three functions are cyclically executed in turn as shown in Figure 2-20. All c files listed in the *File* collumn are part of the generic EtherCAT slave stack.

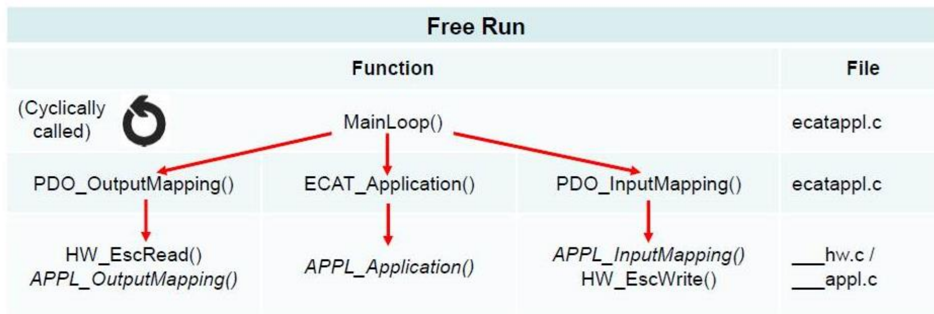


Figure 2-20. Free Run mode process data handling and sequence.

In SM Synchronous Mode, a hardware interrupt (IRQ – Interrupt Request) signal (triggered internally on each slave every time a frame is received) triggers the PDI_Isr function which executes the three generic functions in turn as shown Figure 2-21.

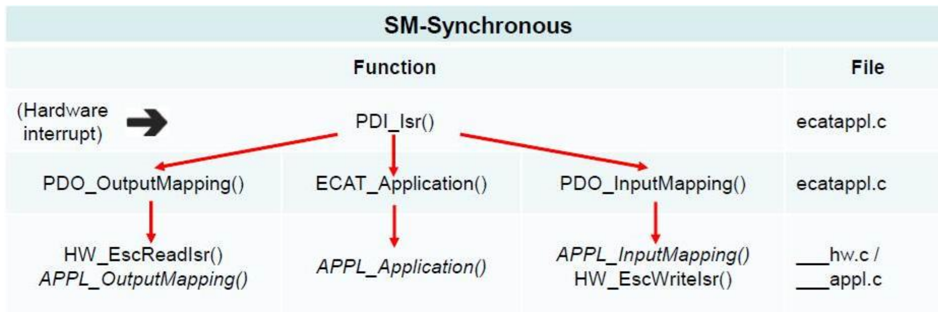


Figure 2-21. SM Synchronous mode process data handling and sequence.

Finally, in DC Synchronous Mode, a hardware interrupt (IRQ – Interrupt Request) signal (triggered internally on each slave every time a frame is received) triggers the Sync0_Isr function which executes the three generic functions in turn as shown Figure 2-22. The aforementioned scheme is identical with SM

Synchronous Mode's process data handling and sequence. However, in DC Synch mode, developers may enable two extra interrupt signals (Al_EVENT_ENABLED and SYNC1) and achieve the highest intrinsic synchronization among the slaves owing to the fact that all three generic functions are executed simultaneously in all slave devices.

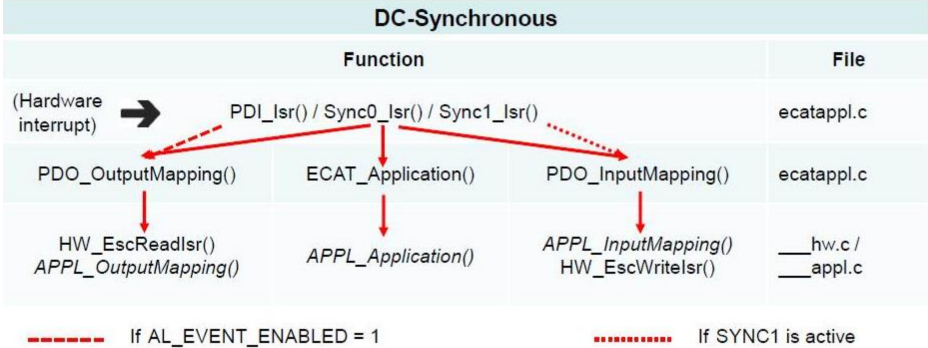


Figure 2-22. DC Synchronous mode process data handling and sequence.

2.4 EtherCAT Application Guide

In this section, the process of configuring an EtherCAT network of microcontrollers with a generic user's application will be thoroughly described. The master device (Personal Computer) will exchange a number of different dummy variables with all connected slave devices (MCUs) in DC Synchronous mode which guarantees the most efficient synchronization. This extensive approach will cover all aspects of EtherCAT architecture, both from master and slave side, providing all the necessary information to build a user defined EtherCAT application quickly and efficiently.

In addition, all the hardware and software components that constitute the application will be described, explaining the main reasons for selecting them, portray the main idea of the running stack, explain the process of adding/removing I/O variables to/from the network and most significantly how to assemble a User Application. These key features form the most essential characteristics of EtherCAT technology when designing a custom application.

2.4.1 EtherCAT Code Structure Overview

A microcontroller in each slave is responsible for the entire application layer. As adumbrated by Figure 2-23, the EtherCAT slave stack consists of three main parts:

- *PDI and Hardware abstraction* which is hardware specific and needs to be implemented according to the platform/PDI. In our application, SPI (Serial Peripheral Interface) plays this role which is the means of communication between the MCU and the EtherCAT Slave Controller.
- *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 such as motor control.

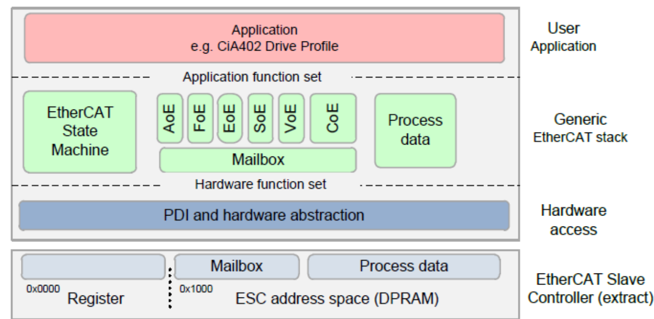


Figure 2-23. EtherCAT Slave Architecture.

2.4.2 Hardware and Software Requirements

The basic EtherCAT system configuration is shown in Figure 2-24. The EtherCAT master uses a standard Ethernet port and network configuration information stored in the EtherCAT Network Information file (ENI). The ENI is created based on EtherCAT Slave Information files (ESI) which are provided by the vendors for every device. Slaves are connected via Ethernet cables and different topology types are possible for EtherCAT networks although, as previously mentioned, the most efficient one is the physical line or ring topology which exploits the minimum wiring scheme.

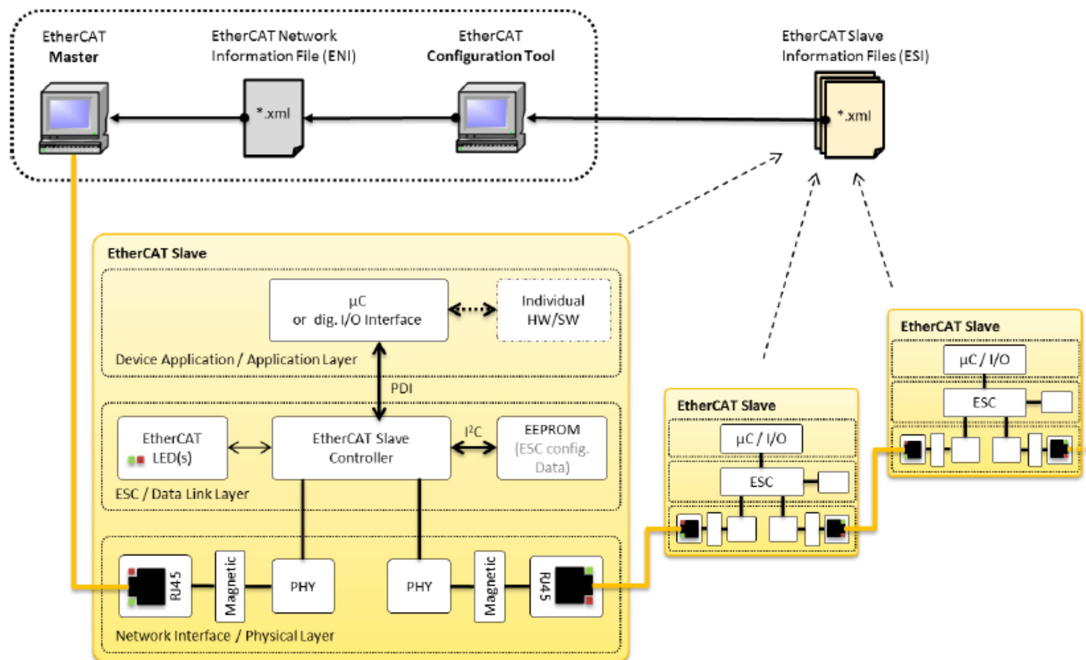


Figure 2-24. EtherCAT Architecture.

As shown in Figure 2-24, an EtherCAT network requires several physical components realizing the Physical Layer, the ESC/Data Link Layer and the Application Layer. In our application some of the components were purchased while others were designed and built in-house in order to meet our needs and specifications.

EtherCAT Master Requirements

TwinCAT 3 Engineering software tool running in Windows 10 (through Visual Studio) on a personal computer (PC) will materialize the EtherCAT Master (EM) device of our network. In fact, this is the only software requirement needed to implement an EM and configure all different kinds of slaves, writing their EEPROM accordingly. Developers should follow the steps described in Download and Install TwinCAT 3 Software to download and install the application on their computer.

The only hardware requirement for an EtherCAT master is a standard Network Interface Controller (NIC, 100 MBit/s Full duplex). TwinCAT Runtime will be able to download the required drivers and switch to RUN Mode only if the PC has compatible network adapters for Real Time Ethernet communication (find all compatible devices as specified by Beckhoff in [28]). Moreover, 100 Mb/s full-duplex Ethernet cable (Cat 5 twisted pair or higher, shielded or unshielded depending on the electromagnetic interference) must be used to guarantee proper cyclic communication.

EtherCAT Slave Requirements

As far as the slave device is concerned, three hardware components are used, namely:

- an *EtherCAT Slave Controller* (ESC) which handles the EtherCAT protocol in real-time by processing the EtherCAT frames on the fly and providing the interface for data exchange between a master and a slave – responsible for the realization of the *Physical* and *Data Link Layers,*
- a *host Microcontroller Unit* (MCU) realizing the *Application Layer* including the Hardware Access, the Generic EtherCAT stack and User Application structures as adumbrated by Figure 2-23, and
- a custom printed circuit board connecting these two devices.

In this application, the C2000 Delfino MCU F28379D LaunchPad Development Kit by Texas Instruments (TI), Figure 2-25, was selected as the host microcontroller of all EtherCAT slaves. The most significant advantages are that it is a low cost, powerful MCU (featuring a TMS320F28379D Dual Core Microprocessor), suitable for motion control of several motors. Also importantly, it is very well documented for using it in EtherCAT applications, since it is similar to the TMDSECATCNCD379D hardware kit used in the respective TI's EtherCAT Solution Reference Guide (reference).

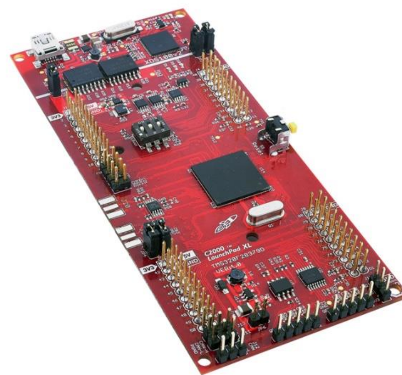


Figure 2-25. EtherCAT Slave MCU.

As mentioned above, in order to implement EtherCAT communication, developers must design or purchase an EtherCAT Slave Controller. In our case, the *FB1111-0141 (SPI) ESC* by Beckhoff (Figure 2-26), was selected as a highly flexible ESC that can communicate with the MCU via Serial Peripheral Interface (SPI) protocol and operate in DC Synchronous mode triggered by three external interrupt signals. An overview of the features of the selected ESC is shown in Figure 2-27.

Figure 2-26. EtherCAT Slave Controller.

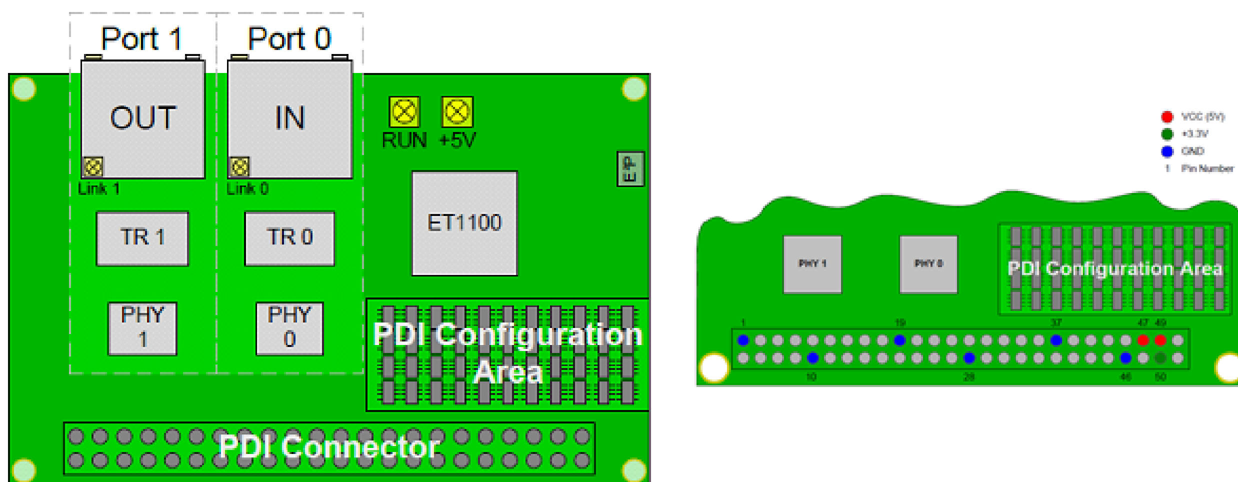


Figure 2-27. Overview of FB1111-0141 features.

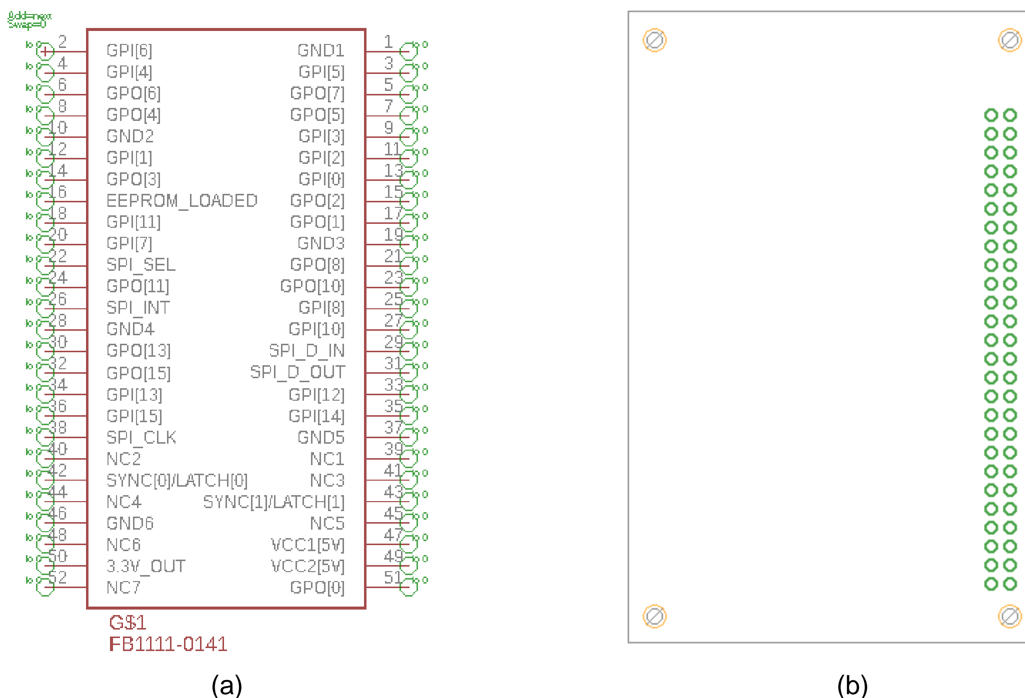
To connect together the MCU and the ESC, it is imperative to build an intermediate board connecting the desired I/Os of these two devices. The desired wiring scheme is adumbrated in Table 2-1. The host MCU pins were selected for SPI-A configuration, however users can select whichever SPI port is more convenient when designing their own application. Note that when designing the PCB, all GND pins should be wired together to avoid jittering or external interference.

Table 2-1. **Pin Connection.**

FB1111–0141 (SPI)		*LauncXL F28379D*		
Pin No	*Property*	*GPIO*	*Pin No*	*Property*
1	Ground (GND)	-	GND	Ground
29	SPI_D_IN (MOSI)	58	15	SPIA: Slave In-Master Out
31	SPI_D_OUT (MISO)	59	14	SPIA: Slave Out-Master In
38	SPI_CLK	60	7	SPIA: Clock
22	SPI_SEL	61	19	SPIA: Slave Select Pin
16	EEPROM_LOADED	124	13	Indicates Loaded EEPROM
26	SPI_INT (IRQ)	125	12	Interrupt IRQ
42	SYNC0 / LATCH0	19	3	SYNC 0 Interrupt signal
43	SYNC1 / LATCH1	18	4	SYNC 1 Interrupt signal
47, 49	V_{cc} (5V supply)	-	5 V	Power Supply

To connect the pins described in Table 2-1, a custom PCB was designed in Autodesk Eagle, such that it can be mounted at the bottom of the host MCU (Delfino) in order to free the top plane for a second PCB handling the control-related peripherals (this is extensively described in the following chapter).

Firstly, the packages of the Delfino Launchpad and the FB1111-0141 ESC were drawn; Figure 2-28 shows the designed *Device* (a) and *Package* (b) of the ESC, while Figure 2-29 shows the *Device* (a) and *Package* (b) of the Delfino Launchpad. It is worth mentioning that it was not necessary to design the whole Delfino Launchpad library due to the fact that only the upper part pins (1-40) were exploited by our application.



(a) (b)

Figure 2-28. FB1111-0141 Library (a) Device (b) Package.

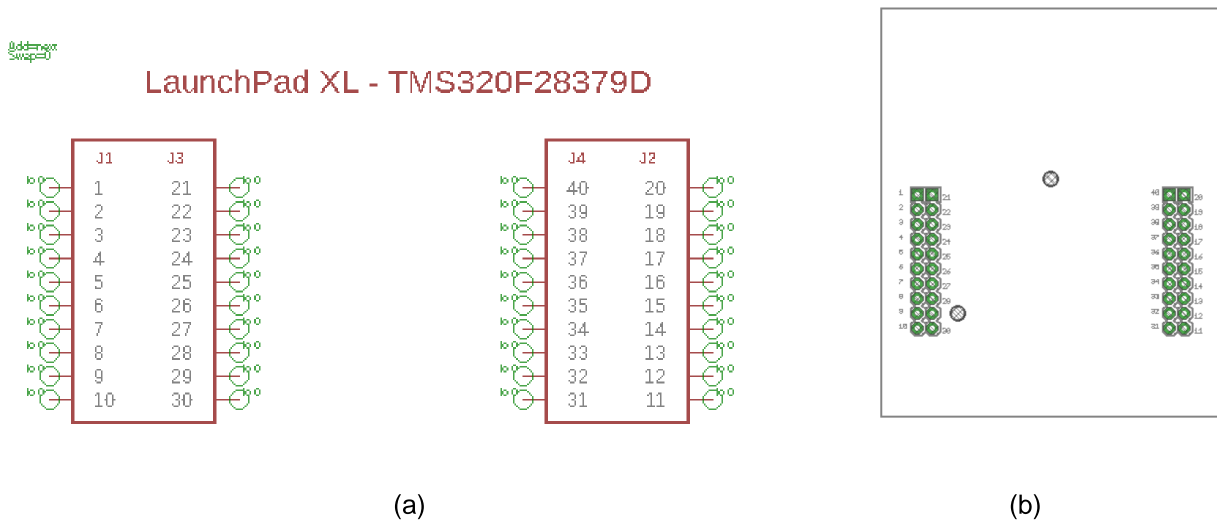


Figure 2-29. Delfino Launchpad Library (a) Device (b) Package.

Secondly, a schematic was drawn to properly connect the desired input and output pins of the aforementioned *Devices* based on Table 2-1 as shown in Figure 2-30. It is obvious that all Ground pins were connected to GND in order to meet Beeckhoff's specifications regarding the FB1111-0141 ESC.

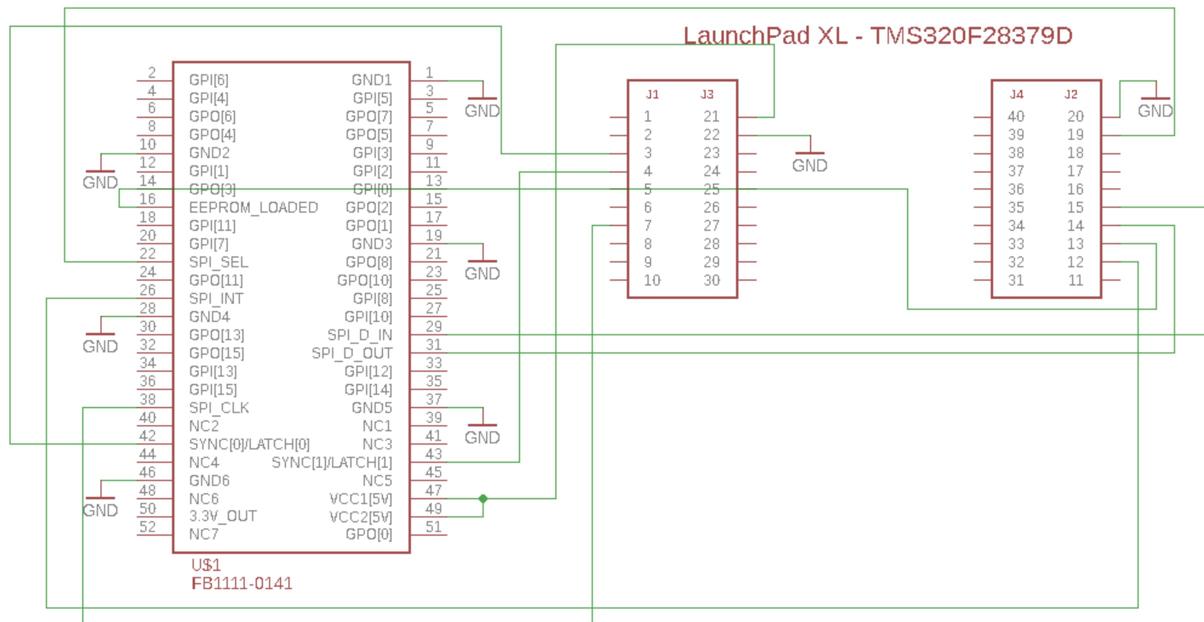
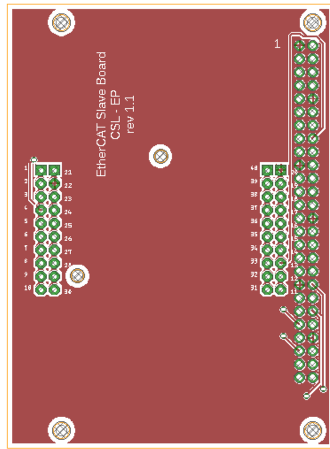
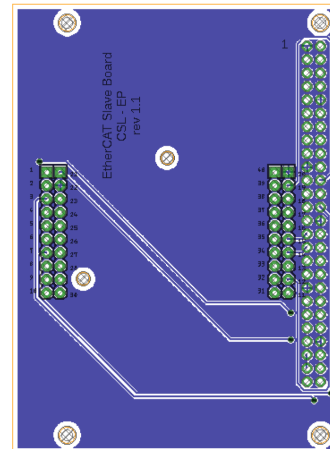


Figure 2-30. Schematic of EtherCAT Slave PCB.

The third and final step was to design the actual board; Figure 2-31 (a) depicts the Top View of the PCB, whilst Figure 2-31 (b) adumbrates the Bottom View as illustrated by *Eagle* Software Tool. Developers my download all the necessary Eagle files from [50] to reproduce or upgrade the design.

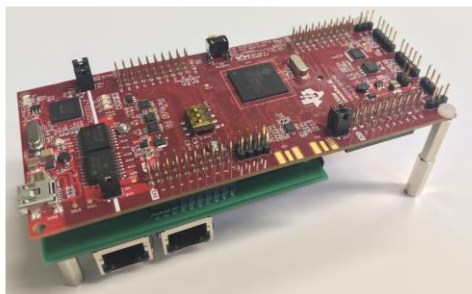


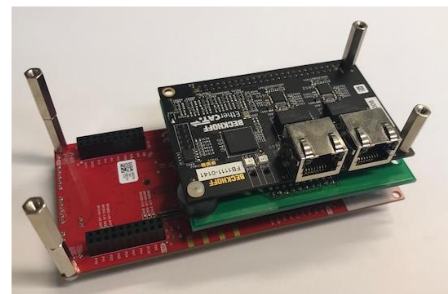


(a) (b)

Figure 2-31. (a) Top View and (b) Bottom View of EtherCAT Slave PCB.

Figure 2-32 depicts the final EtherCAT slave assembly with all the aforementioned components.





(a) (b)

Figure 2-32. EtherCAT Application slave assembly.

As far as the software requirements of the EtherCAT slave are concerned, Code Composer Studio (CCS) was selected to program the slave devices. CCS is an Integrated Development Environment (IDE) to develop applications for Texas Instruments (TI) embedded processors. In addition, for the selected MCU, TI provides highly useful software, namely the ControlSuite and the C2000ware packages, which entail numerous examples. Developers should follow the steps described in Download and Install Code Composer Studio, C2000ware & ControlSuite to download and install these tools on their computer.

Moreover, as specified above, a Configuration Tool is needed to generate a network description, the so called EtherCAT Network Information file (ENI, XML file based on a pre-defined file schema). This is based on the information provided by the EtherCAT Slave Information files (ESI, device description in XML format) and/or the online information provided by the slaves in their EEPROM and their object dictionaries. The ENI file describes the network topology, the initialization commands for each device and the commands which have to be sent cyclically. The ENI file is provided to the master, which sends commands according to this file.

This software tool is provided by the *EtherCAT Technology Group (ETG)*, namely *Slave Stack Code Tool (SSC Tool)* and only ETG members with a valid *Vendor ID* can download and exploit its features. Members of the *Control Systems Lab – Evangelos Papadopoulos* (CSL-EP) may contact Professor E. Papadopoulos to retrieve the CSL-EP credentials in order to download and install the SSC Tool. Future developers should take

into account that all the necessary stack of this application - tutorial has already been generated using the SSC Tool.

However, in case it is required to execute the entire procedure from scratch, one should download the SSC Tool as described in Download and Install Slave Stack Code Tool and emulate the actions described in Generate Slave Stack Code for C28x architecture microcontrollers. In any other case of exploiting the already generated stack projects, SSC Tool is totally unnecessary.

2.4.3 EtherCAT Application Solution Guide

Importing the Project

After installing all the necessary software components and gathering all the necessary hardware components (mentioned above), developers are in position of implementing EtherCAT communication in a few easy steps. The instructions below describe the process of configuring one EtherCAT slave in the network at the beginning and how it can be accomplished-extended for more slaves later.

1. Navigate to the following link [29] and download *EtherCAT Application* repository which includes a CCS project and an xml ENI file.

2. Import the *EtherCAT Application* CCS project into Code Composer Studio by following the instructions of Import CCS project into Code Composer Studio

3. *Specify* and *Link* the desired Target Configuration of the development by following the instructions of Define and Select Target Configuration. This procedure is of utmost importance on accounts that the binary program that will be downloaded to your launchpad must be generated for the specific MCU of your application.

4. The Project Explorer window and EtherCAT_Application project tree should now look like Figure 2-33. Highlighted is the name of the project and the selected *Build Configuration* (_1_LAUNCHXL_F2837xD_SPIA_RAM).

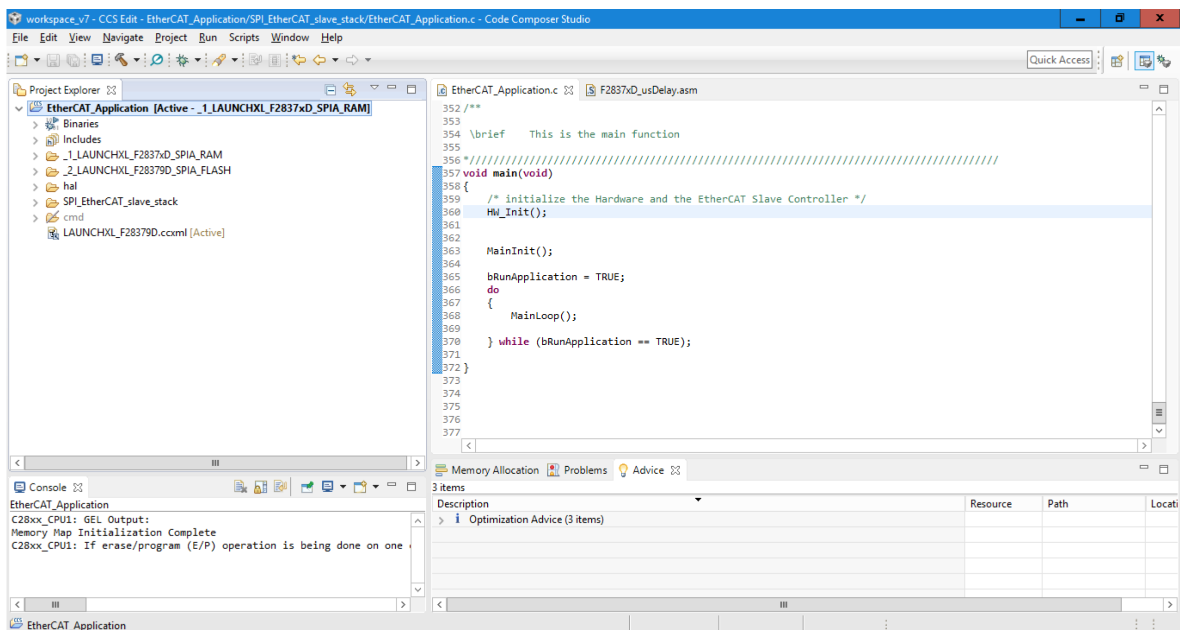


Figure 2-33. CCS with imported project.

Project Overview

The project files are separated into two main folders:

- *SPI_EtherCAT_slave_stack* which contains all the files that realize the *Generic EtherCAT Stack Layer* and the *User Application* (see EtherCAT Code Structure Overview).
- *hal* which contains all the necessary files that initialize and configure the MCU's functionalities (ex clocks, GPIO's,ctimers, communication protocols etc) and materialize the *PDI and Hardware Abstraction Layer* (SPI functions to communicate with the ESC).

Expanding these two folders in the Project Explorer tree, developers can locate all these files and manipulate them in order to cover the needs of their application. The Slave Stack Code execution consists of an initialization phase (executed only once) and a cyclic phase (executed continuously without interruptions) as shown in Figure 2-34 where the MainLoop() function contains the main cycle of the Slave's firmware, which always runs when the slave is properly configured. The *main(void)* function of the project where the aforementioned stack is executed can be located in *SPI_EtherCAT_slave_stack* > *EtherCAT_Application.c*. This is the most significant .c file of EtherCAT's stack due to the fact that it also contains the three process data handling generic functions which define the nature of the project (see Process Data Handling).

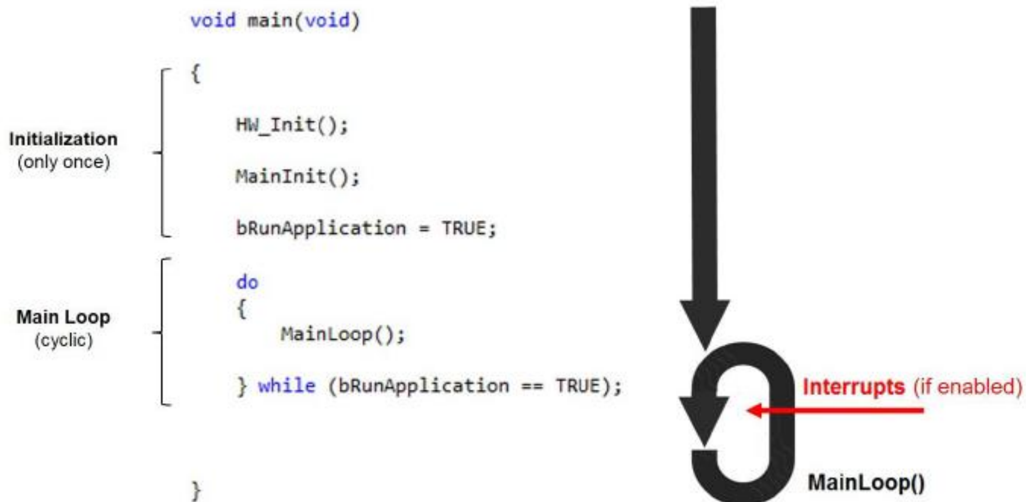


Figure 2-34. Basic SSC execution structure.

The matching between the process data handling generic functions and our project's is:

- *PDO_OutputMapping() → APPL_OutputMapping()*
- *ECAT_Application() → APPL_Application()*
- *PDO_InputMapping() → Appl_InputMapping()*

In Table 2-2, developers can monitor the Input and Output variables of our project as handled by EtherCAT communication. Each variable has a specific type (ex BOOL, INT), belongs to a Record (general address) containing more variables of identical or different type [5] and a unique name within the Record. Output variables are those who are controlled and determined by the master node during the execution of the stack and their Index always begin with 0x70, while Input variables are designated by each slave their Index always begins with 0x60.

Table 2-2. EtherCAT Application Process Data Interface.

Index	*Sub Index*	*Data Type*	*Name*
0x7000	Record		Buttons
	0x01	BOOL	Button1
	0x02	BOOL	Button2
	0x03	BOOL	Button3
	0x04	BOOL	Button4
	0x05	BOOL	Blue_LED
	0x06	BOOL	Red_LED
	0x07	BOOL	Button7
	0x08	BOOL	Button8
	0x09	INT8	Sync
0x7010	Record		Output1INT32
	0x01	INT32	OutINT32Var1
0x7012	Record		OutputUINT16
	0x01	UINT16	OutUINT16Var1
0x7014	Record		Output2INT32
	0x01	INT32	OutINT32Var2
0x7020	Record		OutputINT16
	0x01	INT16	OutINT16Var1
	0x02	INT16	OutINT16Var2
	0x03	INT16	OutINT16Var3
	0x04	INT16	OutINT16Var4
	0x05	INT16	OutINT16Var5
	0x06	INT16	OutINT16Var6

Index	*Sub Index*	*Data Type*	*Name*
0x6010	Record		Input1INT32
	0x01	INT32	InINT32Var1
0x6012	Record		InputUINT16
	0x01	UINT16	InUINT16Var1
0x6014	Record		Input2INT32
	0x01	INT32	InINT32Var2
0x6020	Record		InputINT16
	0x01	INT16	InINT16Var1
	0x02	INT16	InINT16Var2
0x6030	Record		Input3INT32
	0x01	INT32	InINT32Var3
	0x02	INT32	InINT32Var4

The most crucial and interesting functions implemented in *EtherCAT_Application.c* are:

- *APPL_GenerateMapping():* which sends the Input and Output process data size in bytes as calculated from Table 2-2 by the sum of all configured variables. As shown in Figure 2-35, these sizes for our application is 22 bytes for the Input variables and 24 for the Output variables. No additional steps are required by the developers for this function.

```
200 ////////////////////////////////////////////////////////////////////////////////////////
201 /**
202 \return     0(ALSTATUSCODE_NOERROR), NOERROR_INWORK
203 \param      pInputSize  pointer to save the input process data length
204 \param      pOutputSize  pointer to save the output process data length
205
206 \brief    This function sends the process data sizes
207 *////////////////////////////////////////////////////////////////////////////////////////
208 UINT16 APPL_GenerateMapping(UINT16 *pInputSize,UINT16 *pOutputSize)
209 {
210     *pInputSize = 22;
211     *pOutputSize = 24;
212     return ALSTATUSCODE_NOERROR;
213 }
214
215 ////////////////////////////////////////////////////////////////////////////////////////
```

Figure 2-35. APPL_GenerateMapping() function.

- *APPL_InputMapping():* which copies the Input variables from the local memory of the slave (Delfino MCU) to the ESC memory in order to send them to the Master device (Figure 2-36). No additional steps are required by the developers for this function.

```
215 ////////////////////////////////////////////////////////////////////////////////////////
216 /**
217 \param      pData  pointer to input process data
218
219 \brief      This function copies the inputs from the local memory to the ESC memory
220 *////////////////////////////////////////////////////////////////////////////////////////
221 void APPL_InputMapping(UINT16* pData)
222 {
223     uint16_t *pTmpData = (uint16_t *)pData;
224
225     memcpy(pTmpData, &Input1INT320x6010.InINT32Var1,SIZEOF(Input1INT320x6010.InINT32Var1));
226     pTmpData += 2;
227     memcpy(pTmpData, &InputUINT160x6012.InUINT16Var1,SIZEOF(InputUINT160x6012.InUINT16Var1));
228     pTmpData ++;
229     memcpy(pTmpData, &Input2INT320x6014.InINT32Var2,SIZEOF(Input2INT320x6014.InINT32Var2));
230     pTmpData += 2;
231     memcpy(pTmpData, &InputINT160x6020.InINT16Var1,SIZEOF(InputINT160x6020.InINT16Var1));
232     pTmpData ++;
233     memcpy(pTmpData, &InputINT160x6020.InINT16Var2,SIZEOF(InputINT160x6020.InINT16Var2));
234     pTmpData ++;
235     memcpy(pTmpData, &Input3INT320x6030.InINT32Var3,SIZEOF(Input3INT320x6030.InINT32Var3));
236     pTmpData += 2;
237     memcpy(pTmpData, &Input3INT320x6030.InINT32Var4,SIZEOF(Input3INT320x6030.InINT32Var4));
238 }
239
240 ////////////////////////////////////////////////////////////////////////////////////////
```

Figure 2-36. APPL_InputMapping() function.

- *APPL_OutputMapping():* which copies the Output variables from the ESC memory to the local memory of the Delfino MCU slave to update their values within the application. No additional steps are required by the developers for this function.

```
241 ////////////////////////////////////////////////////////////////////////////////////////
242 /**
243 \param      pData  pointer to output process data
244
245 \brief    This function copies the outputs from the ESC memory to the local memory
246 *////////////////////////////////////////////////////////////////////////////////////////
247 void APPL_OutputMapping(UINT16* pData)
248 {
249     uint16_t *pTmpData = (uint16_t *)pData;// allow byte processing
250     uint16_t data = 0;
251
252     /* RxPDO */
253     data = (*(volatile uint16_t *)pTmpData);
254     (Buttons0x7000.Button1) = data & 0x1;
255     data = data >> 1;
256     (Buttons0x7000.Button2) = data  & 0x1;
257     data = data >> 1;
258     (Buttons0x7000.Button3) = data  & 0x1;
259     data = data >> 1;
260     (Buttons0x7000.Button4) = data  & 0x1;
261     data = data >> 1;
262     (Buttons0x7000.Blue_LED) = data  & 0x1;
263     data = data >> 1;
264     (Buttons0x7000.Red_LED) = data  & 0x1;
265     data = data >> 1;
266     (Buttons0x7000.Button7) = data  & 0x1;
267     data = data >> 1;
268     (Buttons0x7000.Button8) = data  & 0x1;
269     data = data >> 1;
270     (Buttons0x7000.Sync) = data  & 0xFF;
271     pTmpData++;
272
273     memcpy(&Output1INT320x7010.OutINT32Var1,pTmpData,SIZEOF(Output1INT320x7010.OutINT32Var1));
274     pTmpData += 2;
275     memcpy(&OutputUINT160x7012.OutUINT16Var1,pTmpData,SIZEOF(OutputUINT160x7012.OutUINT16Var1));
276     pTmpData ++;
277     memcpy(&Output2INT320x7014.OutINT32Var2,pTmpData,SIZEOF(Output2INT320x7014.OutINT32Var2));
278     pTmpData += 2;
279
280     memcpy(&OutputINT160x7020.OutINT16Var1,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var1));
281     pTmpData ++;
282     memcpy(&OutputINT160x7020.OutINT16Var2,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var2));
283     pTmpData ++;
284     memcpy(&OutputINT160x7020.OutINT16Var3,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var3));
285     pTmpData ++;
286     memcpy(&OutputINT160x7020.OutINT16Var4,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var4));
287     pTmpData ++;
288     memcpy(&OutputINT160x7020.OutINT16Var5,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var5));
289     pTmpData ++;
290     memcpy(&OutputINT160x7020.OutINT16Var6,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var6));
291 }
292
293 ////////////////////////////////////////////////////////////////////////////////////////
```

Figure 2-37. APPL_OutputMapping() function.

- *APPL_Application():* which contains the User Application. Within this function, developers are free to decide what they desire to do with the predefined Input and Output variables of EtherCAT Application project as long as they are cautious not to mix variables of different types. The circled lines of Figure 2-38, as stated by the comment as well, will flash the build in LEDs of Delfino MCU through *Blue_LED* and *Red_LED* variables. Users may follow the comments to configure their own application handling the process data.

```
293 /////////////////////////////////////////////////////////////////////////////////////////
294 /**
295 \brief    This function will be called from the synchronisation ISR or from the MainLoop
296 if no synchronization is supported executing the User Application
297 *////////////////////////////////////////////////////////////////////////////////////////
298 void APPL_Application(void)
299 {
300 /*
301  * EtherCAT Output Variables (Outputs FROM the master)
302  * Uncomment the following lines and use the Output variables of the master as you desire in your application.
303  * For example -> int Flag = Buttons0x7000.Button1;
304  * With this line in your code, the value of "Flag" variable will be determined by EtherCAT Master's output variable "Button1"
305  */
306     // = Buttons0x7000.Button1;
307     // = Buttons0x7000.Button2;
308     // = Buttons0x7000.Button3;
309     // = Buttons0x7000.Button4;
310     // = Buttons0x7000.Button7;
311     // = Buttons0x7000.Button8;
312
313     // = Output1INT320x7010.OutINT32Var1;
314     // = OutputUINT160x7012.OutUINT16Var1;
315     // = Output2INT320x7014.OutINT32Var2;
316
317     // = OutputINT160x7020.OutINT16Var1;
318     // = OutputINT160x7020.OutINT16Var2;
319     // = OutputINT160x7020.OutINT16Var3;
320     // = OutputINT160x7020.OutINT16Var4;
321     // = OutputINT160x7020.OutINT16Var5;
322     // = OutputINT160x7020.OutINT16Var6;
323
324 // Flash the built in LEDs of the Delfino microcontroller by manipulating the "Blue LED" and "Red LED" EtherCAT variables using TwinCAT
325     GPIO_WritePin(31, !Buttons0x7000.Blue_LED); // Turn on/off blue LED (GPIO31) depending on output of Blue_LED for debug purpose
326     GPIO_WritePin(34, !Buttons0x7000.Red_LED); // Turn on/off red LED (GPIO31) depending on output of Red_LED for debug purpose
327 /*
328  * EtherCAT Input Variables (Inputs TO the master)
329  * Uncomment the following lines and use the Input variables to the master as you desire in your application.
330  * For example -> InputINT160x6020.InINT16Var1 = OutputINT160x7020.OutINT16Var1 + OutputINT160x7020.OutINT16Var2;
331  * With this line in your code, the value of "InputINT160x6020.InINT16Var1" EtherCAT input variable will be determined
332  * the sum of "OutputINT160x7020.OutINT16Var1" and "OutputINT160x7020.OutINT16Var2" EtherCAT output variables using TwinCAT
333  */
334
335     //Input1INT320x6010.InINT32Var1 = ;
336     //InputUINT160x6012.InUINT16Var1 = ;
337     //Input2INT320x6014.InINT32Var2 = ;
338
339     //InputINT160x6020.InINT16Var1 = ;
340     //InputINT160x6020.InINT16Var2 = ;
341
342     //Input3INT320x6030.InINT32Var4 = ;
343     //Input3INT320x6030.InINT32Var3 = ;
344
345 }
346
347 /////////////////////////////////////////////////////////////////////////////////////////
```

Figure 2-38. APPL_Application() function.

Although the procedure of adding or removing EtherCAT variables is both painstaking and time consuming, if developers decide that they want to proceed accordingly, they should follow the instructions of Add and remove EtherCAT Input and Output variables.

Configuring the Project

1. Select the desired *Build Configuration* of the imported project. EtherCAT_Application project contains two separate Build Configurations one for RAM and one for Flash. These two projects are identical as far as functionality is concerned. However, Build Configuration number 1 is intended for RAM which means that the Delfino MCU will execute the downloaded project only until it is powered-off. On the other hand, Build Configuration number 2 is intended for FLASH which means that the project will be saved in Flash memory and the MCU will “remember” it even after rebooting. In general, RAM configuration is intended for testing purposes, while FLASH configuration when a project is properly functioning. In order to choose the desired

Build Configuration, right click on the project's name, click Build Configurations > Set active and the desired configuration as depicted in Figure 2-39.

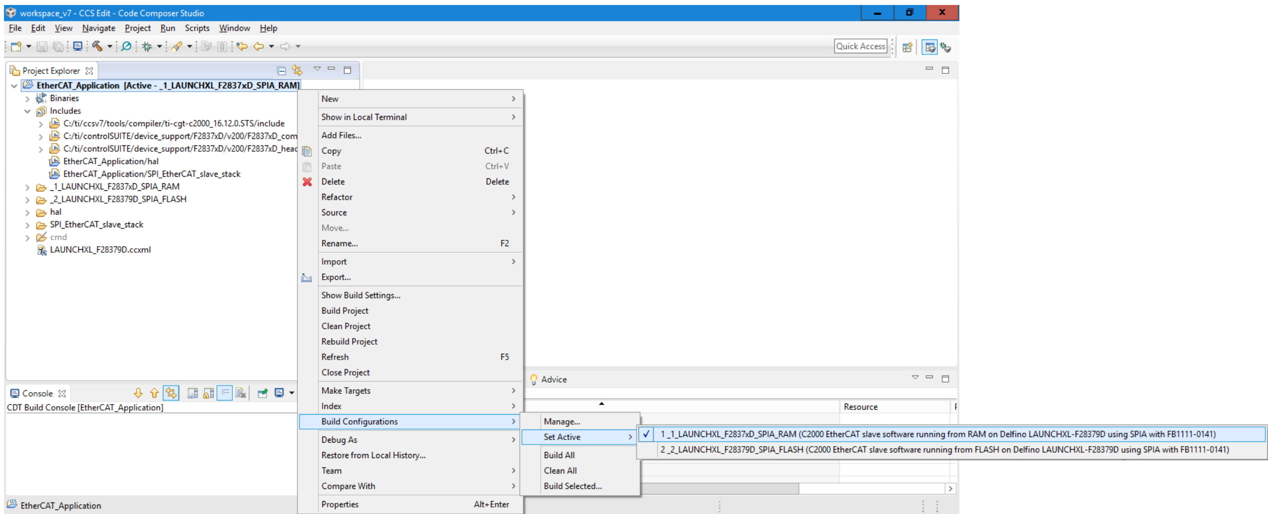


Figure 2-39. Select Build Configuration.

2. Connect the launchpad to the PC using the microUSB cable and *Build* the project. If no error occurs, then download the project into the launchpad by clicking the *Debug* button as shown in Figure 2-40. Take into consideration the fact that if the Delfino MCU is powered through the USB port, jumpers JP1, JP2 and JP3 must all be mounted.

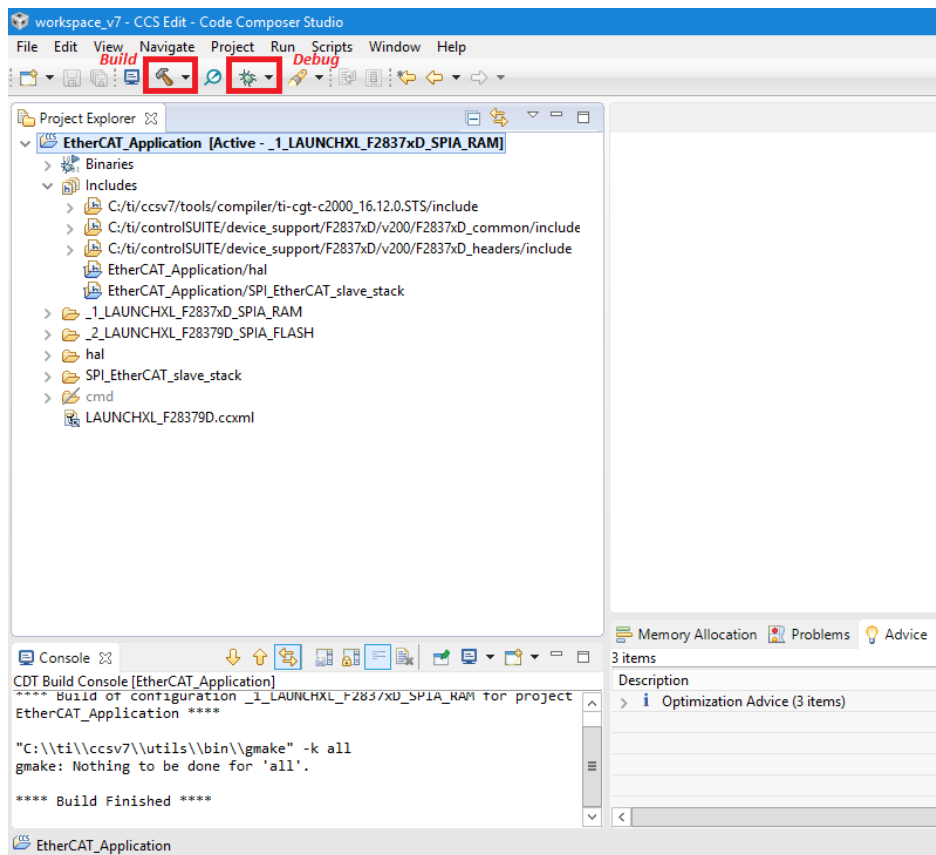


Figure 2-40. Build and Debug CCS Project.

3. The launchpad consists of two cores, thus select CPU1 and click OK from the pop up window as illustrated in Figure 2-41.

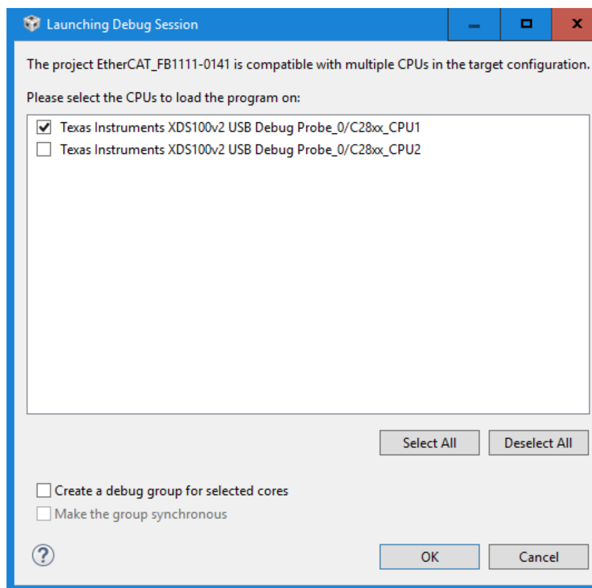


Figure 2-41. CPU selection.

4. Code Composer Studio will then automatically turn into CCS Debug mode. When the project is downloaded (obviously, downloading the Flash project lasts a lot longer than RAM) click the Resume button as adumbrated in Figure 2-42 and the slave side is now up and running. The *Terminate* button will not stop the execution of the program in the MCU yet it will only end the debug session. For the time being DO NOT *Terminate* the debug session to make sure that the slave is properly running in DC mode.

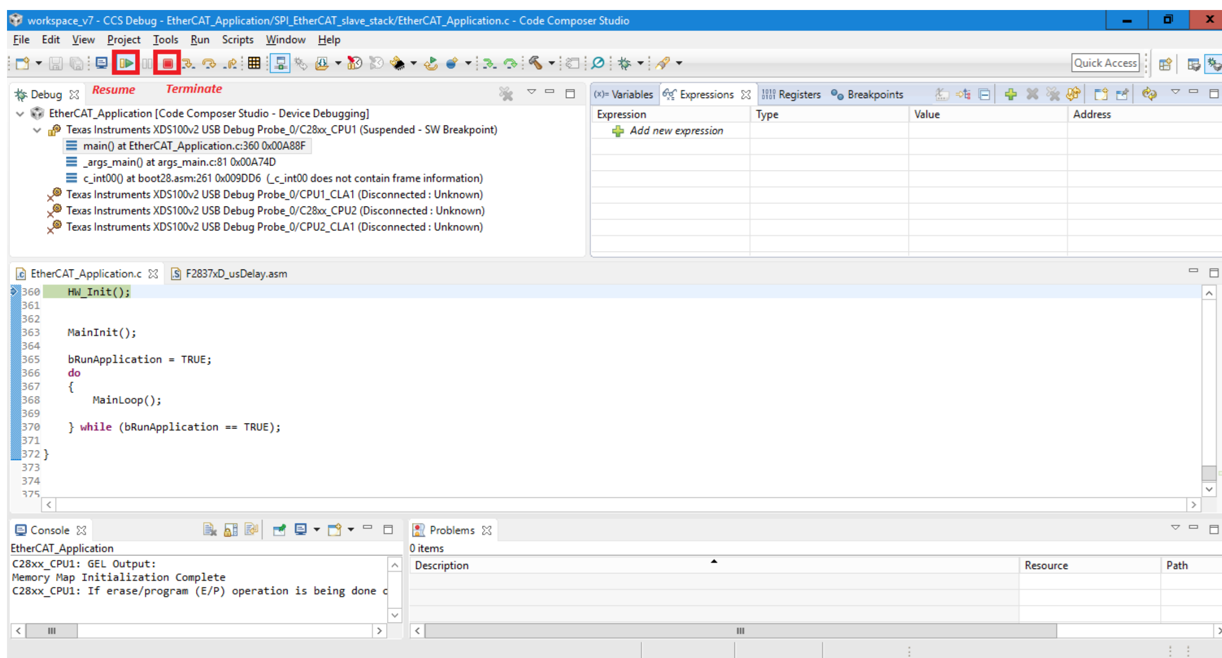


Figure 2-42. CCS Debug window.

5. Connect the *IN* port of the FB1111-0141 ESC, using a suitable Ethernet cable (see EtherCAT Master Requirements) with the PC as illustrated in Figure 2-43 and start TwinCAT XAE (VS 2013).

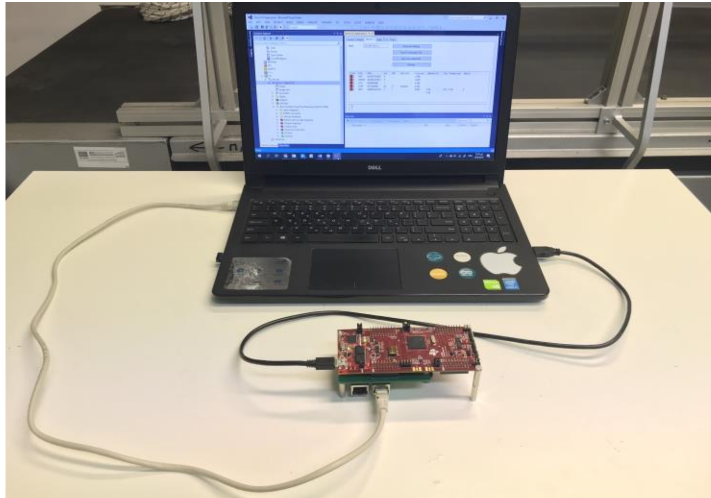


Figure 2-43. EtherCAT Assembly.

6. Select *File* > *New* > *Project* as shown in Figure 2-44.

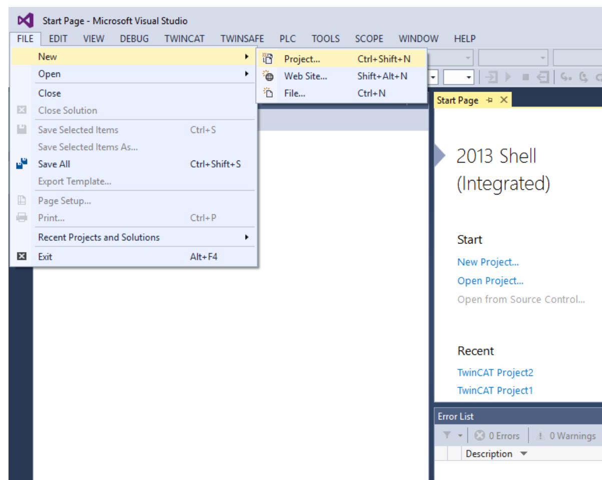


Figure 2-44. TwinCAT new project.

7. From the *TwinCAT Projects tab* select *TwinCAT XAE Project (XML format)*, name the project (ex EtherCAT Application) and click *OK* as illustrated in Figure 2-45.

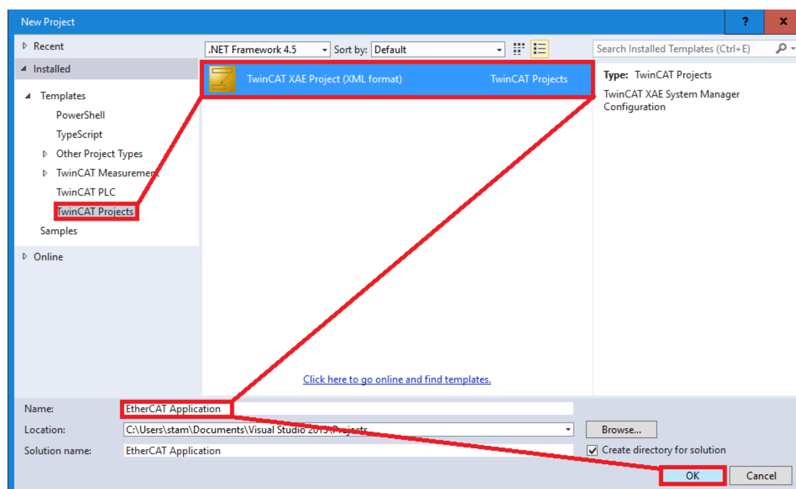


Figure 2-45. TwinCAT EtherCAT Application project.

8. From the *Solution Explorer* window expand the *I/O* element, right click on *Devices* and select *Add New Item* (Figure 2-46).

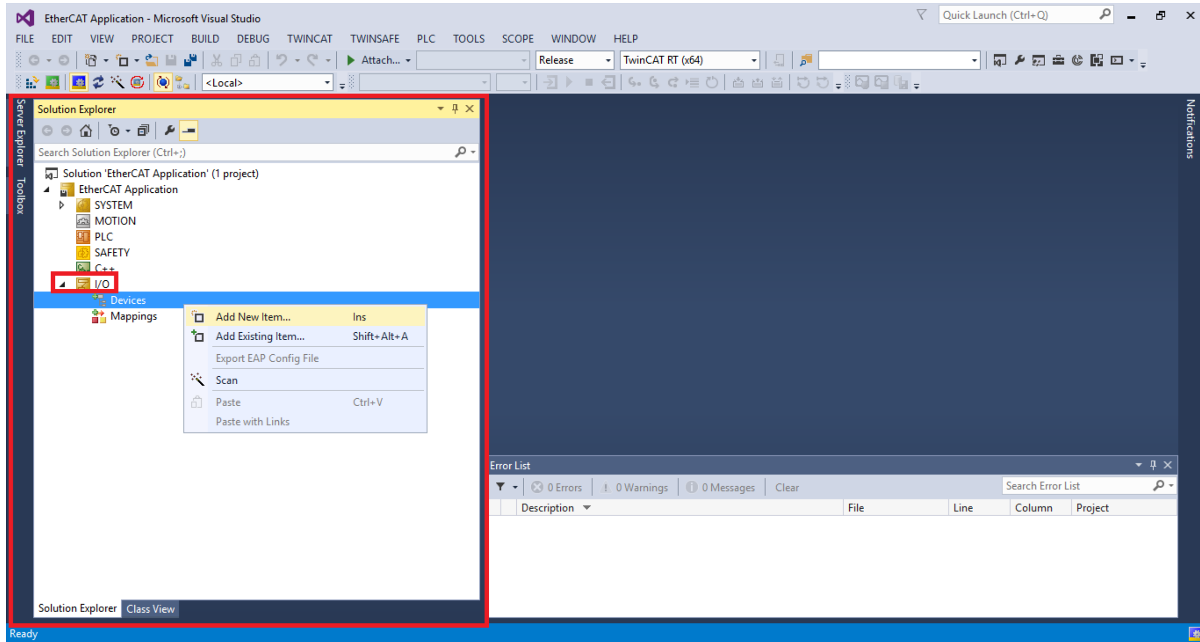


Figure 2-46. TwinCAT Solution Explorer.

9. Select *EtherCAT Master* and click *OK* (Figure 2-47). This way, your PC is now configured as an EtherCAT Master device.

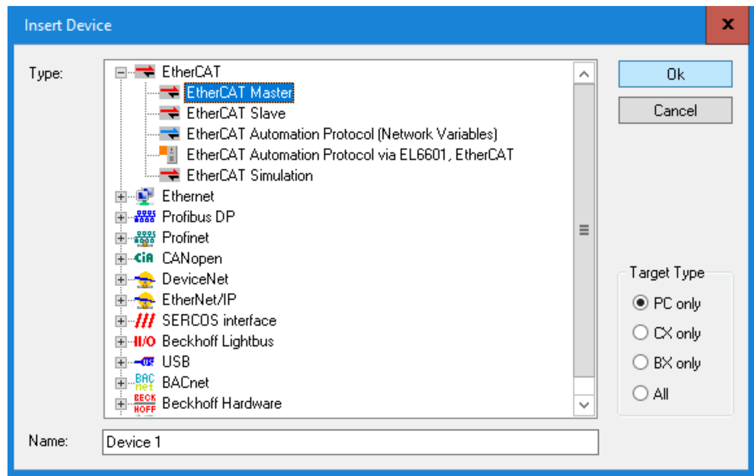


Figure 2-47. EtherCAT Master realization.

10. Navigate to the location of the downloaded folder from Bitbucket, copy *EtherCAT Application (SPI).xml* file and paste it at the following location *C:\TwinCAT\3.1\Config\Io\EtherCAT* owing to the fact that in order to be recognized by the TwinCAT development environment, ESI descriptions of slave devices shall be saved in the default directory.

11. In Visual Studio, select *TWINCAT* > *EtherCAT Devices* > *Reload Device Descriptions* (Figure 2-48) on accounts that if the content of the TwinCAT default folder is changed (new files are added, old files are deleted, files overwritten, content of one or more files is changed), the ESI database must be reloaded in order to make the changes available.

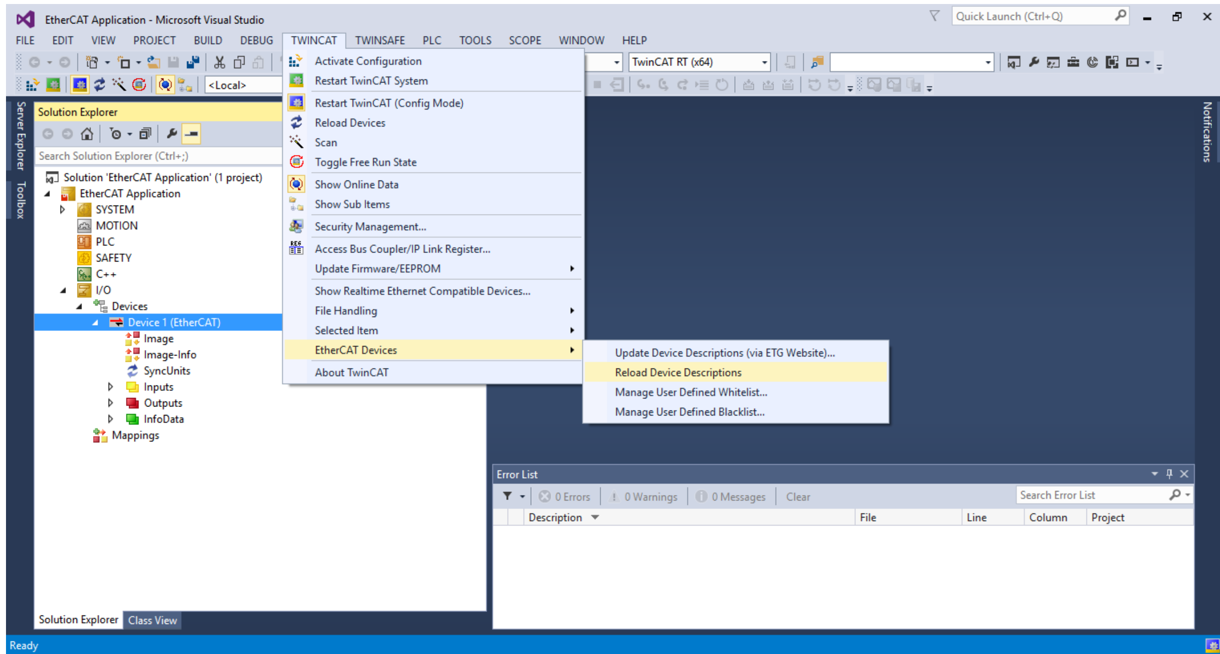


Figure 2-48. Reload Device Descriptions.

12. The definitions of the TwinCAT buttons can be viewed in Figure 2-49. These functionalities will be exploited extensively in the following steps.

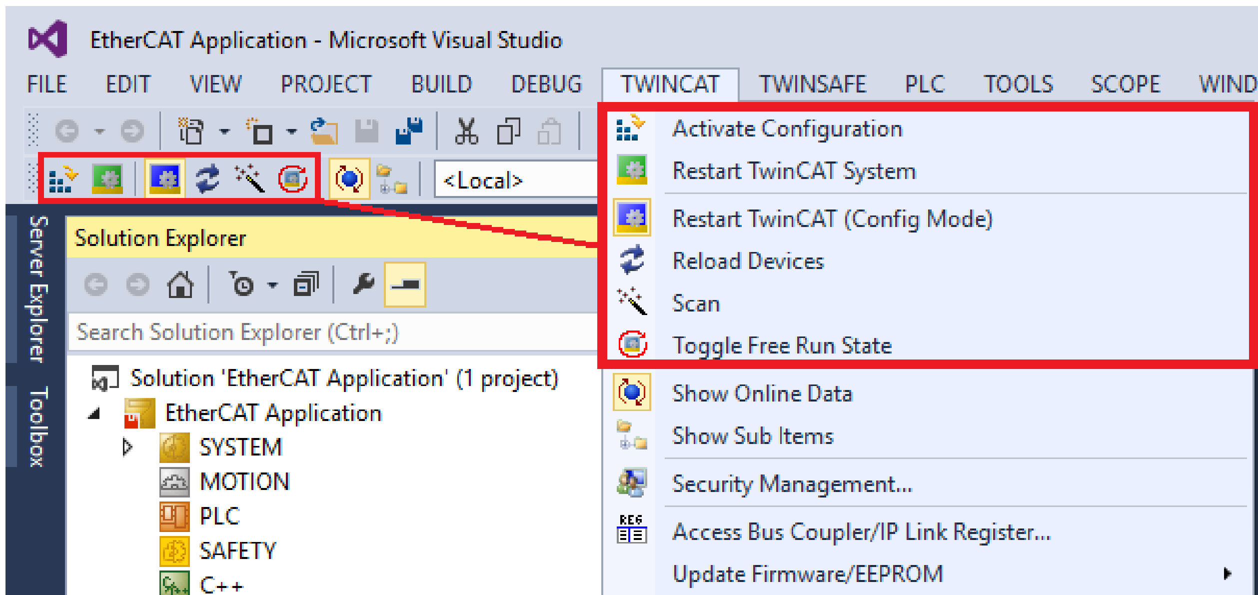


Figure 2-49. Definitions of TwinCAT buttons.

13. Click once on the *Device 1 (EtherCAT)* item of the *Solution Explorer* tree and the *Scan* button (Figure 2-49) will instantly become clickable. Select the *Scan* feature (automatic scan for slave devices) and select *No* in the pop up window asking whether to *Activate Free Run*. A slave (TwinCAT defines slave devices as *Boxes*) must be now visible on the *Solution Explorer* tree within the *Device 1 (EtherCAT)* master device with the last configuration that was written on the ESC's EEPROM as shown in Figure 2-50.

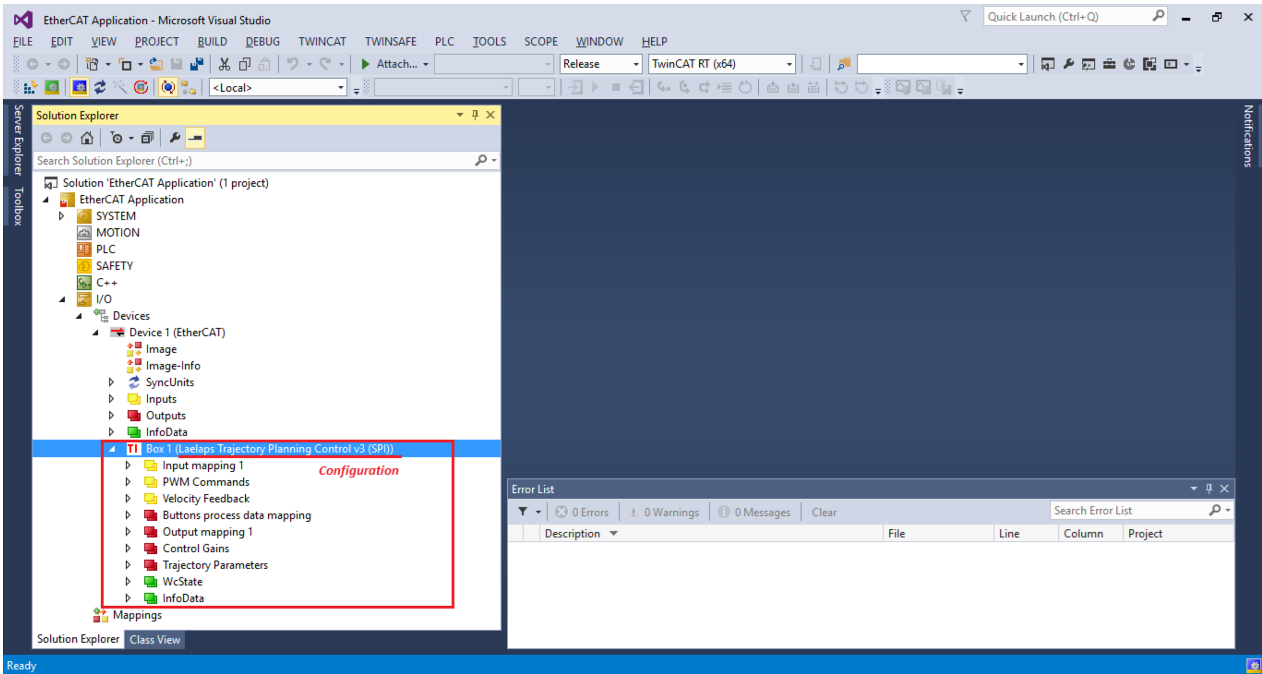


Figure 2-50. TwinCAT scan for slaves.

14. In order to write the EEPROM of the ESC with our project's description file (xml) so that it matches with the configuration of the Delfino MCU, double click on *Device 1 (EtherCAT)*, select the *Online* tab of the emerged window, right click on *Box 1* item and choose *EEPROM Update*.(Figure 2-51).

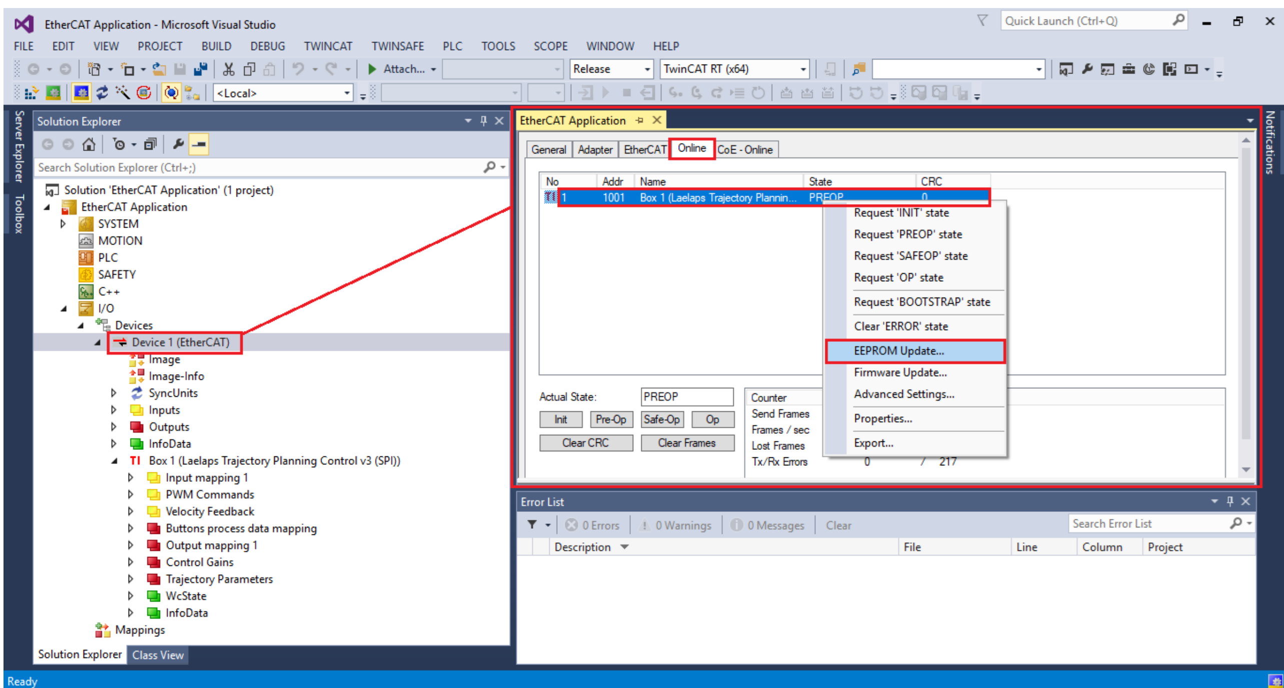


Figure 2-51. Update EEPROM of ESC's memory.

15. Select *EtherCAT Application (SPI) (11110141 /1)* and click *OK* (Figure 2-52). The Number *11110141* is the Product Code and *1* is the Revision Number of the XMl file.

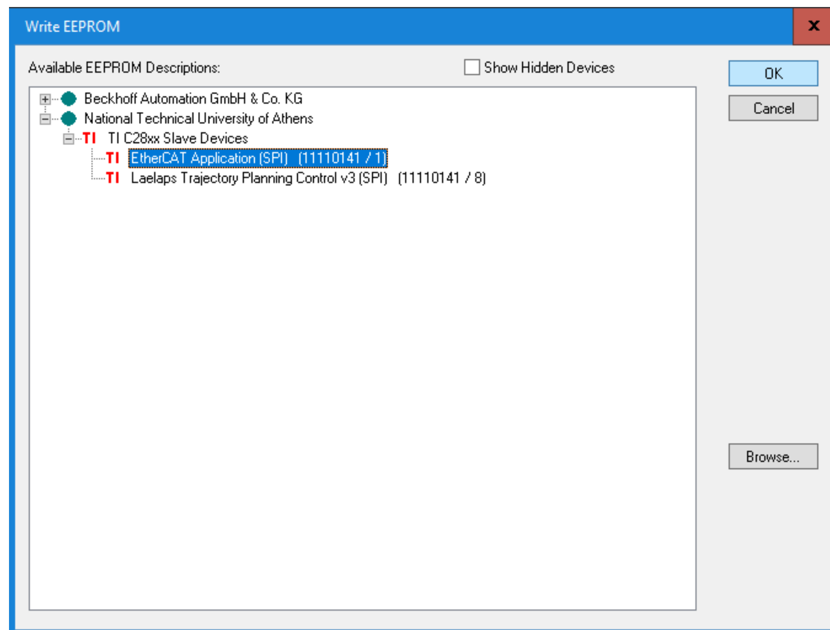


Figure 2-52. XML selection.

16. Now the slave's EEPROM memory is correctly configured, yet TwinCAT cannot automatically recognize this transition. In order to do so, on the *Solution Explorer* window, right click on *Box 1*, select *Remove* and then *OK* (Figure 2-53).

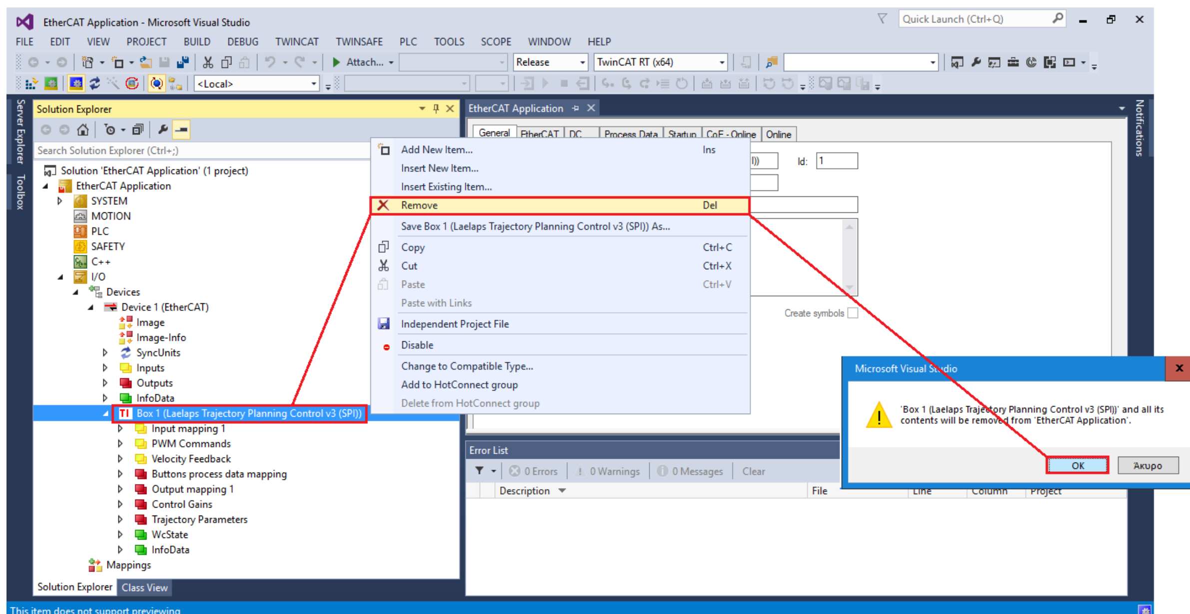


Figure 2-53. Remove EtherCAT slave.

17. *Scan* the network again as indicated above and monitor *Box 1* with the desired configuration and EtherCAT variables (Table 2-2) by expanding the respective tree items as illustrated in Figure 2-54. One can also rename the slave by clicking on the *Box 1* and altering the *Name* item of the *General* tab (ex EtherCAT Application Slave Device). The application is now completed and ready to use. One can test the project in Free Run Mode (no synchronization) to ensure that everything is executed as intended and try the DC Synchronous mode later on.

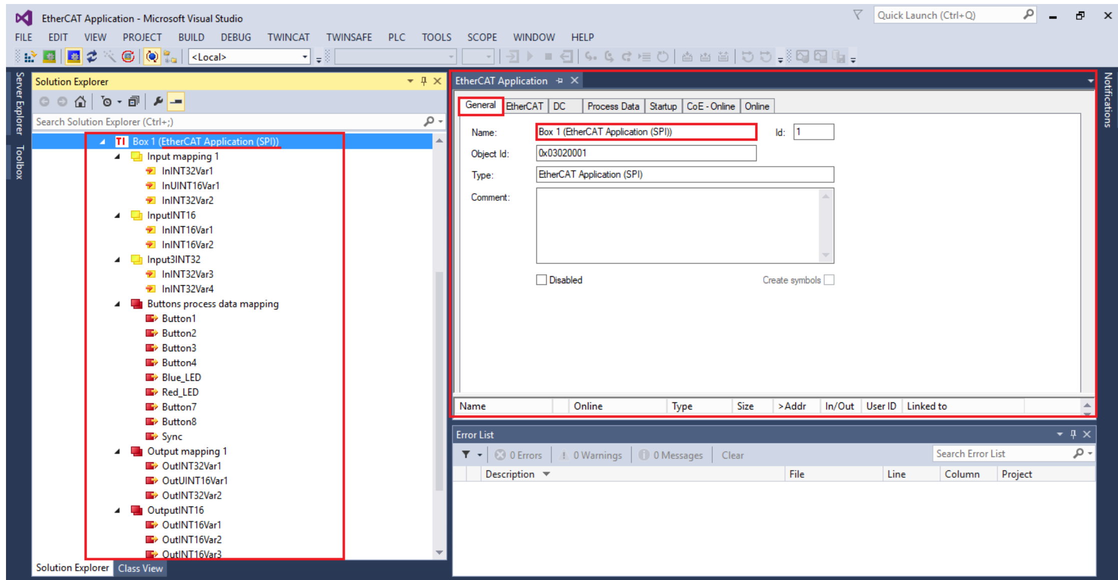


Figure 2-54. EtherCAT Application in TwinCAT.

18. Activate *Free Run Mode* by clicking *Toggle Free Run State* button (Figure 2-49) and ensure that the slave device has swiched to *Operation State* by checking the *RUN* LED of the FB1111-0141 ESC (must be ON) and the *Current State* of the *Online tab* (Figure 2-55) which must be *OP* (Operational).

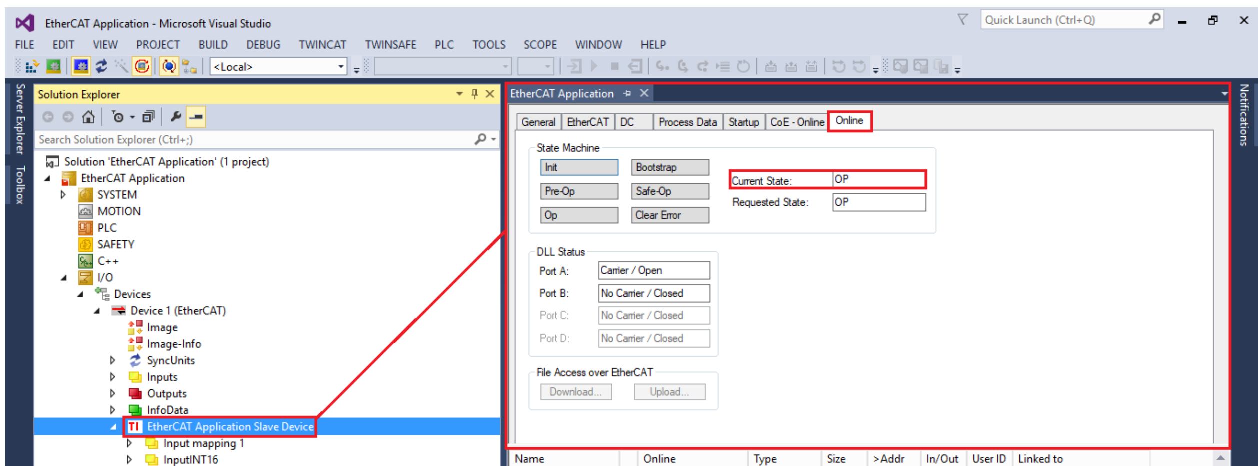


Figure 2-55. EtherCAT Slave's State Machine.

Testing the Project

EtherCAT Application is now running in Free Run Mode and users are now in position of Online Writing the Output variables which are being sent to the Slave Device and inspect the Input variables that originate from the slave device depending on the User Application that they have coded. The only predefined feature of the project is the blue and red LEDs of the Delfino MCU that are linked to the Blue_LED and Red_LED output variables of the Buttons record.

In order to execute an Online Write and turn ON the blue LED for example, expand the project tree of *EtherCAT Application Slave Device*, click on the *Blue_LED* boolean variable of the *Buttons* output record, select *Write* on the *Online tab*, type *1* in the *Dec* field and select *OK* (Figure 2-56). The exact same process can be followed to determine the value of any other variable from the Output variables list.

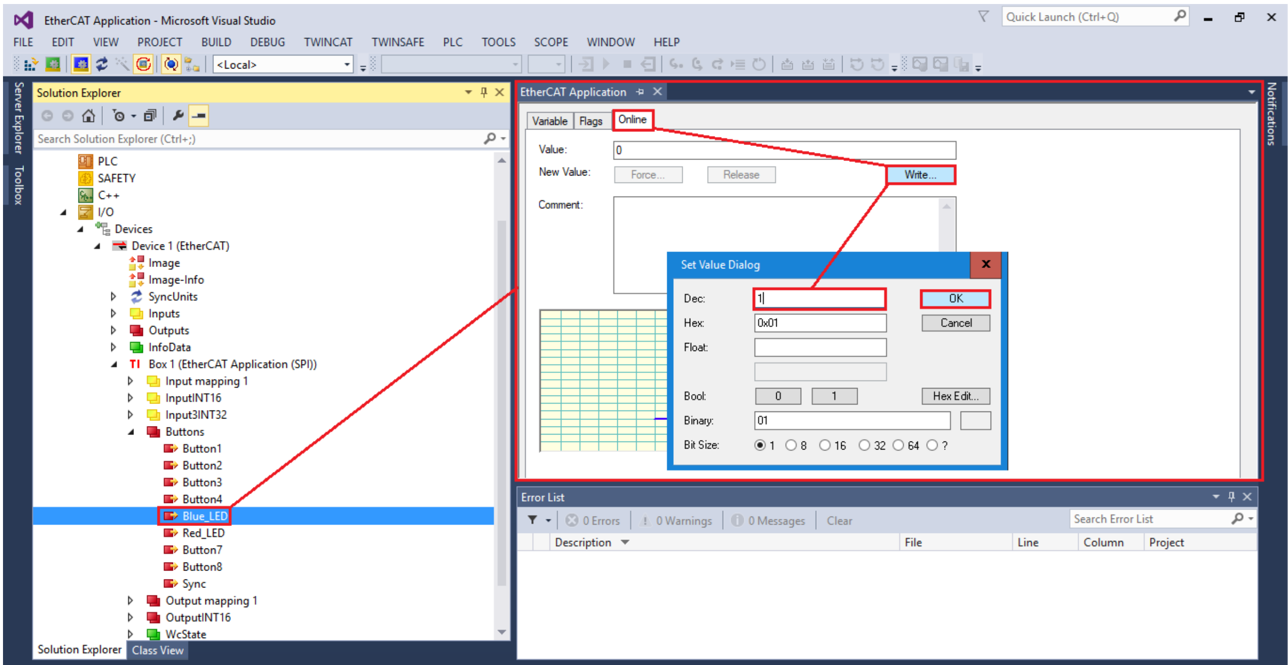


Figure 2-56. Online Write of Blue_LED.

Obviously, Input variables cannot be determined by the master since they are configured by the application running in the slave device. However, users may inspect the value of input variables by navigating to the *Online* tab of the desired one as depicted in Figure 2-56 for the *Blue_LED* output variable.

Project in DC – Sync mode

EtherCAT Application project is configured to exploit all three interrupt channels (physical signals) for synchronization (*PDI_IRQ, SYNC0 and SYNC1*) when operating in DC-synchronous mode and more specifically the **SyncManager/Sync0/Sync1** mode adumbrated in Figure 2-57.

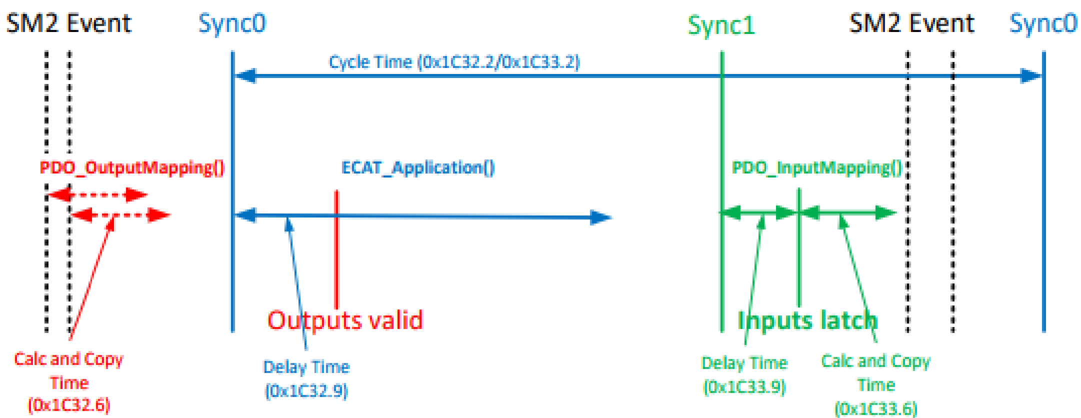


Figure 2-57. SyncManager/Sync0/Sync1 Mode.

The output process data mapping is triggered by the SM2 event, the ECAT_Application is trigerred by Sync0 and the input latch by SYNC1 (Figure 2-58). In EtherCAT Application we have specified all the Delay Times to be zero in order to minimize the execution time of the slave's project and achieve higher cycle times.

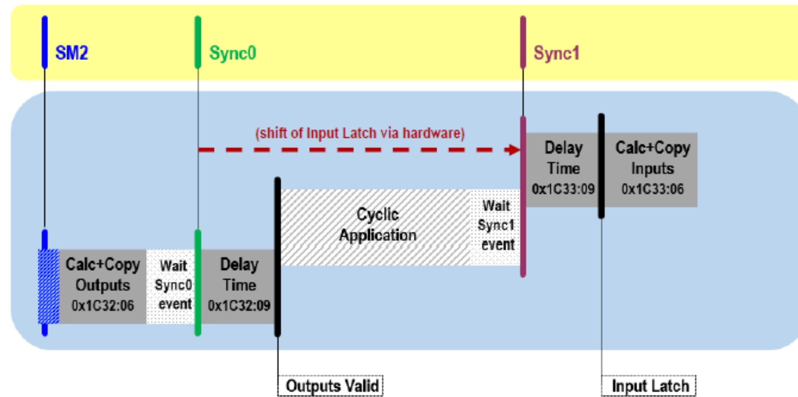


Figure 2-58. DC Sync mode of EtherCAT Application.

In order to force the slave to operate in DC – Sync mode and achieve the ultimate synchronization, follow the instructions of TwinCAT in Run Mode. Timers and flags have been placed within the stack to make sure that the slave is Operating in DC mode and calculate the execution time of each of the aforementioned EtherCAT functions that determiningly define the minimum cycle time of the frame. Follow the instructions of Add Watch Expression in CCS Debug to add the variables of Figure 2-59.

Expressions | (x)= Variables | Registers | Breakpoints

Expression	Type
(x)= PDI_Isr_Output	int
(x)= timer_PDI_Isr_Output	unsigned long
(x)= SYNC0_Isr_Appl	int
(x)= timer_SYNC0_Isr_Appl	unsigned long
(x)= SYNC1_Isr_Input	int
(x)= timer_SYNC1_Isr_Input	unsigned long

Figure 2-59. Timers and Flags to watch.

If all integer type variables are 1, then the slave is properly functioning in DC mode and the values of the unsigned long type variables (timer_) indicate the required execution time in micro seconds listed in Table 2-3.

Table 2-3. Generic functions execution time.

EtherCAT slave stack function	Execution time (μs)
PDO_OutputMapping	67
ECAT_Application	1
PDO_InputMapping	17
Minimum Cycle Time	85

The minimum cycle time of our application with the given process data exchange is the sum of the execution times of the above three functions. The configured cycle time in TwinCAT must exceed this value to switch in Operational mode. However, taking into consideration that we covet to achieve the fastest communication, we shouldn’t diverge from the minimum cycle time to a great extend but test multiple cycle times close to this value until the slave becomes operational with no lost frames.

Another interesting thing to consider regarding EtherCAT when deciding on the frame cycle time is the time needed by the master to transmit a frame on the network or more explicitly the time the network card needs (running at 100Mbit/s) in order to physically transmit the corresponding frame on the cable, in every

cycle. This feature can be calculated directly from TwinCAT and displayed by the *Size/Duration* column of *EtherCAT* tab (Figure 2-60). The five predefined Datagrams of the project, depending on the selected synchronization mode and EtherCAT's addressing, are also highlighted in the same figure.

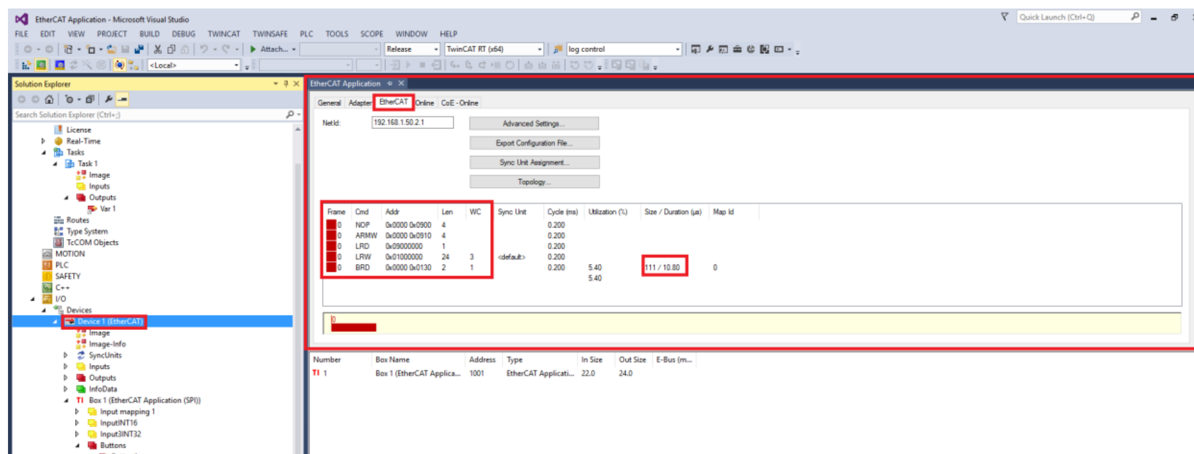


Figure 2-60. EtherCAT Application frame disintegrated.

In our project, this transmition time can be disintegrated into the components described in Figure 2-61 and Table 2-4.

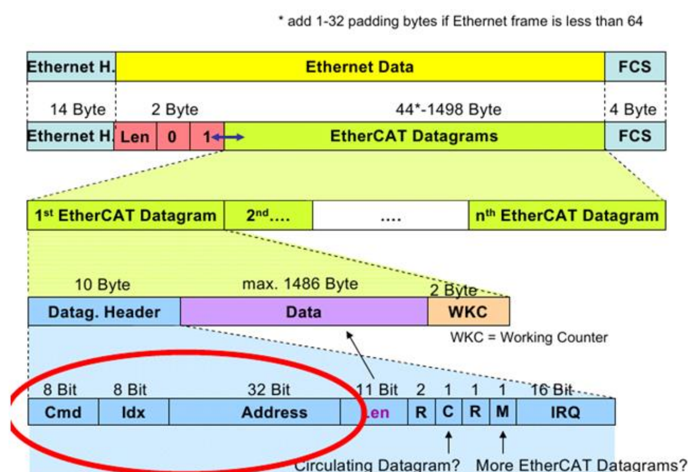


Figure 2-61. EtherCAT frame description.

Table 2-4. EtherCAT frame components.

Frame Component	Bytes	Description
Ethernet Header	14	fixed
EtherCAT Header	2	fixed
Datagram Headers	10 * 5 = 50	10*number of Datagrams
Datagram Data	4+4+1+24+2=35	Sum of bytes
Datagram WKCs	2 * 5 = 10	2*number of Datagrams
Ethernet FCS	4	fixed
Intergap + Preamble+SOF	12+7+1=20	fixed
Total	135	1080 bits (@100Mb/s) = 10.8 μs

Since the total time needed by the master (10.8 μs) is almost eight times less than the minimum execution time of slave (85 us), the frame cycle time principally depends on the latter. Taking the above into consideration, the minimum frame cycle time that enabled the slave device to successfully switch to *Operational Mode* for the given configuration and process data without having any lost frames at all was **200 μs.** In lower cycle times, a considerable number of frames were lost during the communication and the robustness of the the project was doubtful. However, we need to take into consideration the fact that only integer multiples of Twincat's base unit (50 μs) were allowed to be tested for frame cycle time. In your application, this cycle time might be even greater owing to the fact that the execution time of the ECAT_Application will increase.

The procedure described above can easily be extended to the desired number of slaves by simply downloading the same project to all Delfino MCU as already specified, connecting the slaves following the wiring scheme of Figure 2-24 and scanning the new EtherCAT network. Subsequently, emulate the steps of TwinCAT in Run Mode to configure all slaves to operate in DC – Sync mode (to achieve ultimate synchronization among the slave devices and test your application in realtime. This process is also explained in the next chapter where four EtherCAT slaves are connected to the network yet an example of TwinCAT window with four configured slaves can ve viewed in Figure 2-62.

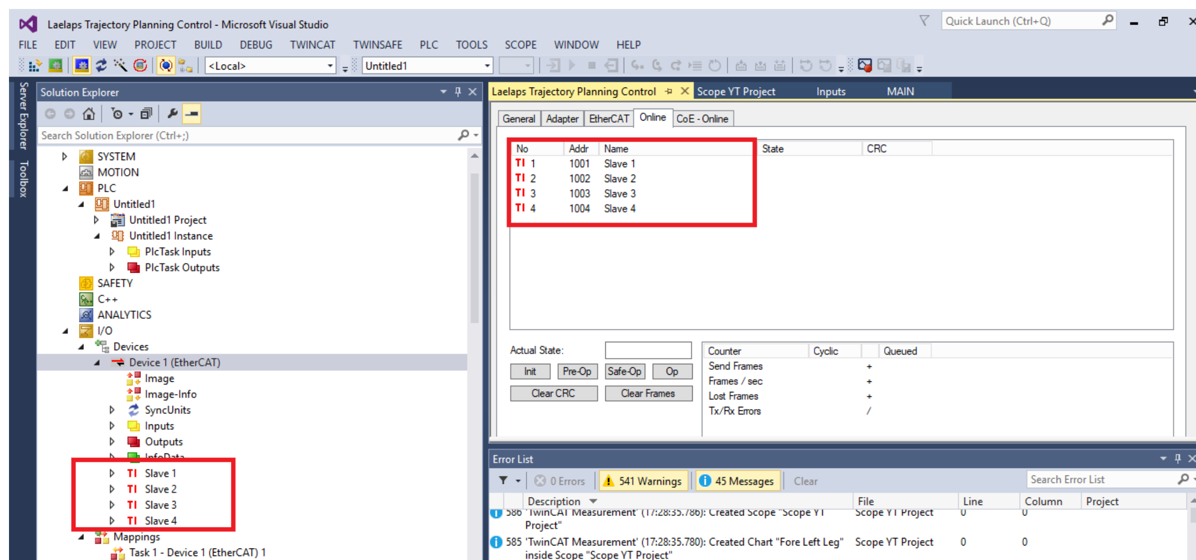


Figure 2-62. Example of EtherCAT network with 4 configured slaves.

3 Motion Control of Laelaps II via EtherCAT

As stated in previous chapters, the purpose of moving towards the EtherCAT technology is to implement a decentralized control architecture on Laelaps II quadruped robot [30] where each slave device controls the motion of one leg. This chapter presents the firmware running in Laelaps II motion control MCUs and the configuration procedure of the EtherCAT master to handle all four slave devices and save the necessary data for post processing using *TwinCAT's Scope View* tool.

The main focus is on designing a network of MCUs responsible for motion planning, control and synchronization of the legs using trajectories at the toes and an indirect force control based on [31] . The motion parameters are handled by the EtherCAT Master communicating with four Delfino MCUs, leading the robot to several walking and running gaits. Finally, a detailed guide is given for building from scratch all the required software and hardware components used in this chapter.

3.1 Laelaps II robot description and motion planning

In [32] and [33] one may find all available details of Laelaps I (Figure 3-1) as far as mechanical and electrical design, functionalities and programming scheme are concerned. Moreover, the experimental validation of the centralized motion control theory is presented and conclusions are drawn for this initial architecture.

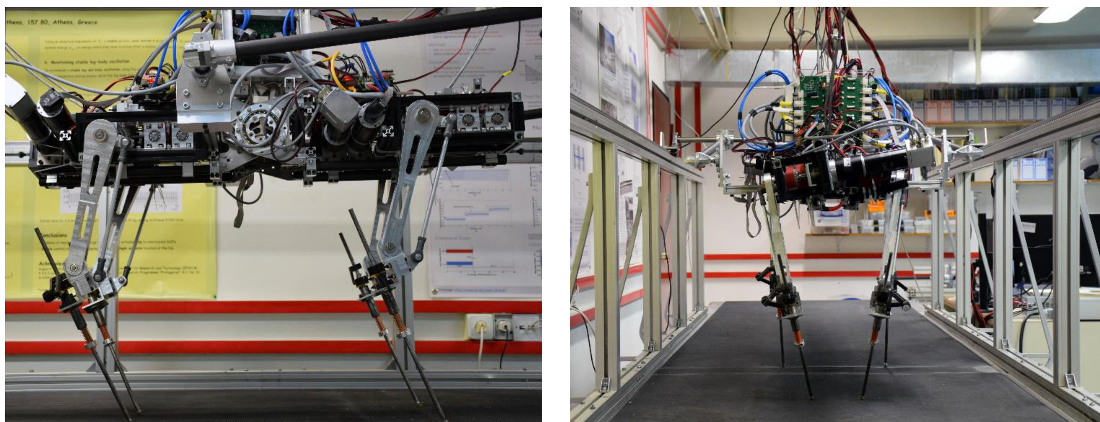


Figure 3-1. Laelaps I.

Laelaps II (Figure 3-2) has certain improvements that distinguish it from its previous version Laelaps I regarding both mechanical and electrical properties. This chapter presents the main features of the robot that intersect with motion control; more specifically the leg design, the actuator-related characteristics and the power supply systems.

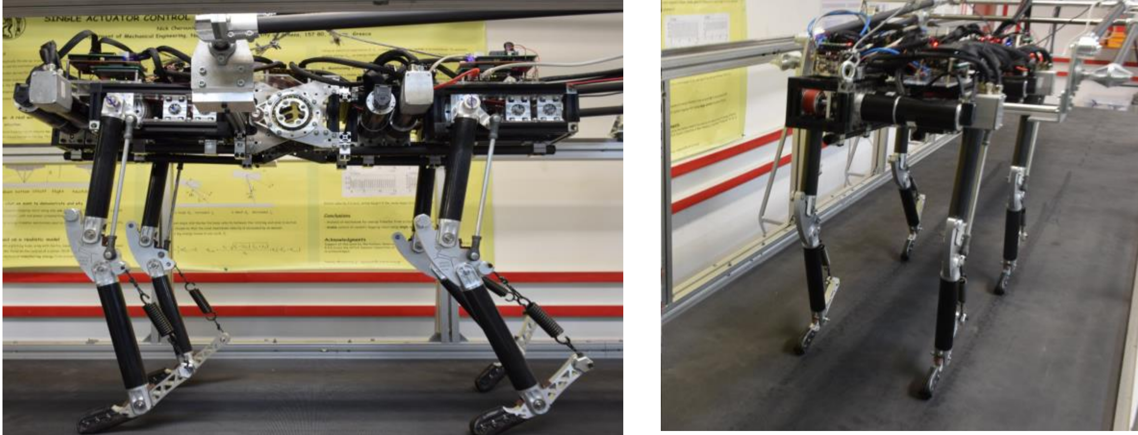


Figure 3-2. Laelaps II.

Laelaps II, compared to the first quadruped, has several enhacements including:

- New leg design, fibraticed with lightweight carbon fiber tubes and custom alluminium parts
- Replacement of the PCIe/104 tower, which was used as the central control unit of all motors with four identical EtherCAT Slave towers with each controlling the motion of one leg based on parameters designated by an EtherCAT Master (decentralized control)
- Upgraded driver extension boards to deal with issues encountered in the former design
- Reallocation of the front parts of the body so that all fours legs are symmetrically distributed

Most of the electrical upgrades are thoroughly described in [5] and readers are encouraged to refer to its fourth chapter for more details.

3.1.1 Leg design and motion planning

Although each leg clearly consists of three links (Figure 3-3), due to the fact that the attached spring is highly stiff, we consent that it comprises of two links (upper actual, lower virtual).

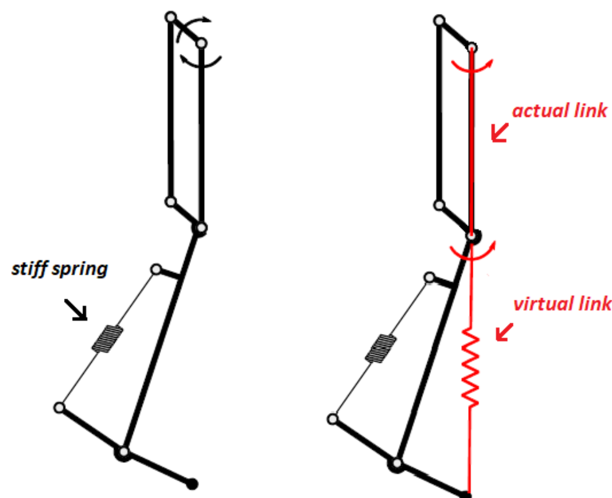


Figure 3-3. Actual and virtual links of Laelaps II legs.

Figure 3-4 illustrates the motion planning and control parameters of the leg which will be used in our project.

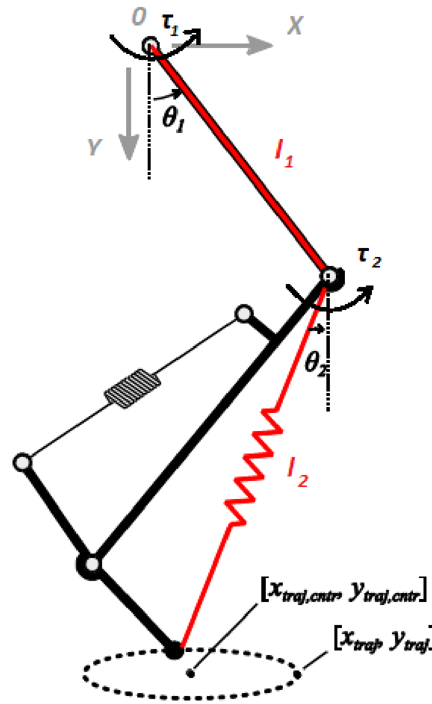


Figure 3-4. Leg model.

Forward Kinematics

$$
\begin{aligned}
x_E &= l_1 \sin\theta_1 + l_2 \sin\theta_2 \\
y_E &= l_1 \cos\theta_1 + l_2 \cos\theta_2
\end{aligned}
\tag{3-1}
$$

Inverse Kinematics

Using the law of cosines:

$$
\begin{aligned}
&\varphi = \theta_2 - \theta_1 \\
&{x_E}^2 + {y_E}^2 = l_1^2 + l_2^2 - 2 l_1 l_2 \cos(\pi - \varphi) = l_1^2 + l_2^2 + 2 l_1 l_2 \cos\varphi \\
&\cos\varphi = \frac{{x_E}^2 + {y_E}^2 - ({}_1^2 + l_2^2)}{2 l_1 l_2} \\
&\sin\varphi = -\sqrt{1 - \cos^2\varphi} \\
&\varphi = a\tan 2(\sin\varphi, \cos\varphi)
\end{aligned}
\tag{3-2}
$$

Finally,

$$
\begin{aligned}
\theta_2 &= \frac{\pi}{2} - a\tan 2(y_E, x_E) + a\tan 2(l_1 \sin\varphi, l_2 + l_1 \cos\varphi) \\
\theta_1 &= \theta_2 - a\tan 2(\sin\varphi, \cos\varphi)
\end{aligned}
\tag{3-3}
$$

Leg's Workspace

The maximum effective length of the leg (knee joint at end-stop) is given by,

$$l_{eff,\max} = l_1 + l_2 = 250 + 350 = 600mm \quad (3\text{-}4)$$

The minimum effective length of the leg (knee joint at end-stop) is given by,

$$l_{eff,\min} = \sqrt{{l_1}^2 + {l_2}^2} = \sqrt{250^2 + 350^2} = 430.1163mm \quad (3\text{-}5)$$

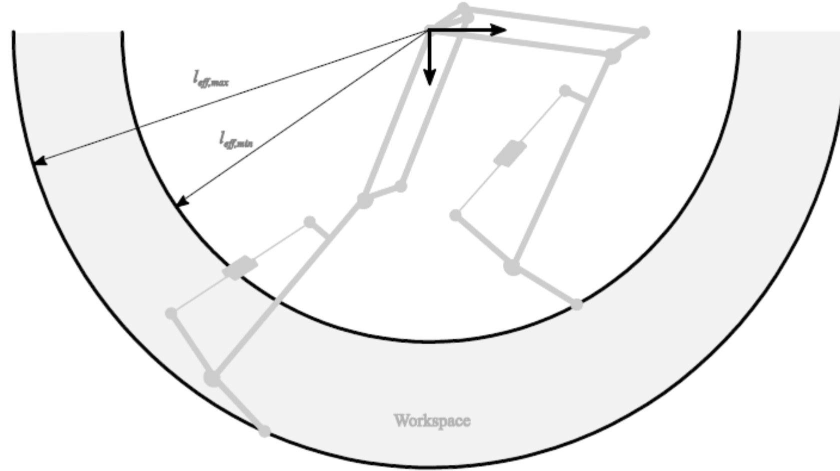


Figure 3-5. Leg's workspace.

Trajectory Planning

The firmware of each slave is specifically configured to enable each leg to move along (semi)elliptical trajectories with all the parameters controlled by the master, along the lines of [31] . Hence, all slaves exclusively handle the calculation payload for the motion control of each leg. The EtherCAT Master just determines a list of necessary parameters for the desired elliptical trajectory which are listed in the *TrajectoryParameters* Record of Table 3-1. The elliptical shape is defined by (3-6) w.r.t. point 0 (hip axis) defined in Figure 3-4, and must always be within the limits of the leg's workspace. Therefore, the current project running in all slave devices is programmed in such way that it forbids any leg to move outside its predefined workspace, yet it will stay at the last acceptable (x_{traj},y_{traj}) point until a new allowed one is passed to the Inverse Kinematics code implementation.

$$\begin{aligned} x_{traj} &= x_{traj,cntr} + a\cos(\omega_{traj}t + \varphi) \\ y_{traj} &= y_{traj,cntr} + b\sin(\omega_{traj}t + \varphi) \end{aligned} \quad (3\text{-}6)$$

To model the impedance of the treadmill's floor, a flattening parameters has been added on the y axis amplitude (b), altering the shape of the elliptical trajectory as shown in Figure 3-6.For more information regarding the aforementioned leg design, motion planning and control, refer to [34] .

Visualization in Matlab

For testing purposes and evaluation, a Matlab script was written to visualize the leg motion. This helps in Laelaps experiments in order to accurately define the elliptical parameters for each slave and avoid errors regarding motion planning. Snapshots from the execution of the code are illustrated in Figure 3-6. Developers may find and use this testing tool at [35] and Matlab Leg Modelling Code.

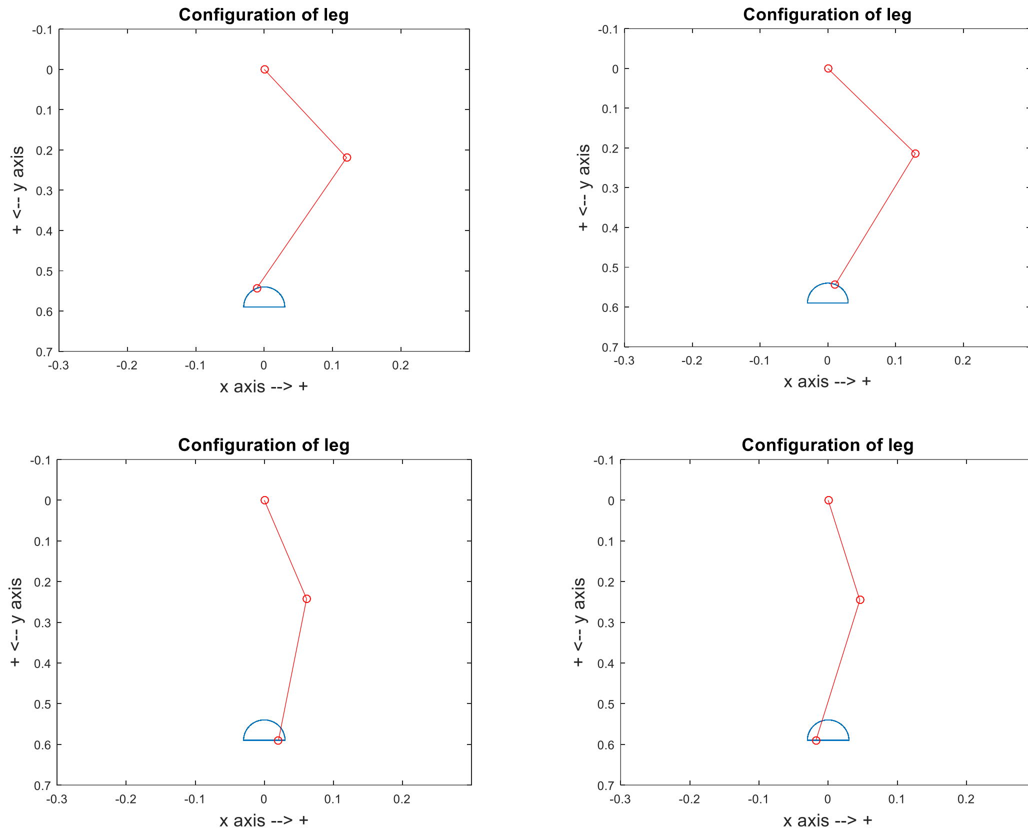


Figure 3-6. Visualization of legs motion in Matlab.

3.1.2 Electrical system

As mentioned above, the electrical system of Laelaps is exhaustively described in Chapter 4 of [5] . Therefore in this thesis, the approach is EtherCAT slave oriented and towards preparing Laelaps II for experiments. However, a fundamental overview of the electrical scheme is included to understand the general concept.

The main electrical components are:

- 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.
- 8 motor driver boards (amplifiers) (along with their designated extension boards mounted on top) 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.
- 4 EtherCAT Control Tower Assembly slaves connected to the motor drivers and the encoders of each leg.

It is worth mentioning that due to the mounting of the motors onto the body, each set of EtherCAT tower and connected drivers controls the leg of the other side (left → right). For example, the indicated *EtherCAT Control Tower Assembly* and *Drivers with Extension Boards* of Figure 3-7 control the motion of the Fore Right Leg and NOT the Fore Left Leg which is visible in the same figure. This detail is of utmost importance when downloading a project to the Delfino boards since users must not confuse the *Build Configuration* with the side of Laelaps legs.

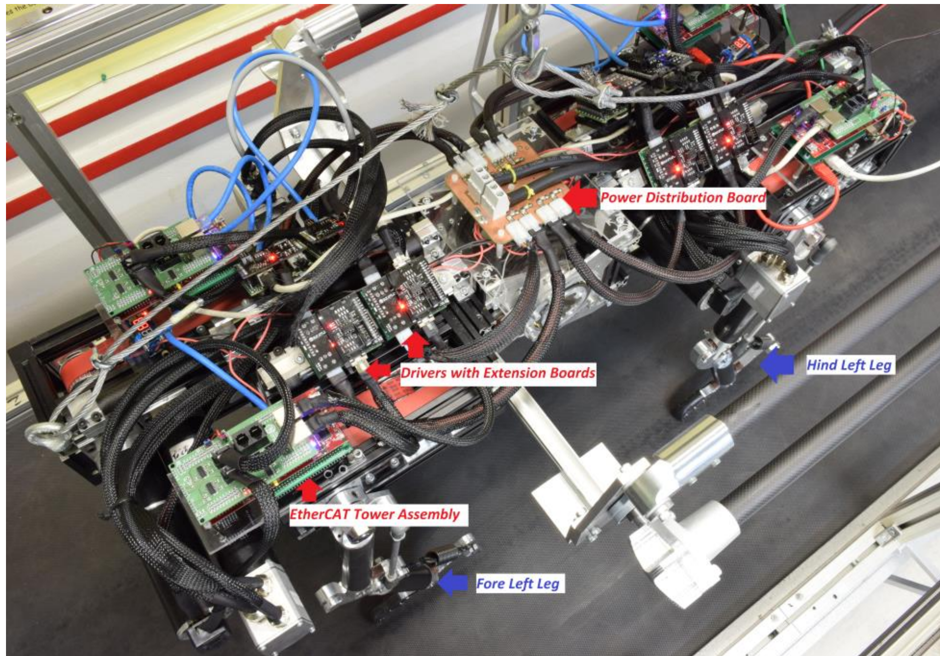


Figure 3-7. Electrical System of Laelaps.

EtherCAT Tower Assembly

As specified above, each leg of Laelaps is being controlled by one *EtherCAT Control Tower Assembly* which plays the role of an EtherCAT slave in the configured network. Hence, four identical assemblies needed to be constructed to control Laelaps II. Figure 3-8 shows the final version of the EtherCAT Control Tower Assembly that was used throughout the trotting experiments. Except for the components described in EtherCAT Slave Requirements and illustrated in Figure 2-32, the assembly also includes:

- a *TMS320F28379D Extension board* interfacing with all necessary peripherals (ePWM, eQEP etc.) for two motors presented in section 4.4.3 of [5]
- a voltage regulator (DC - DC converter, Step – Down 5V 2A USB [36]) supplying the logic power to the whole assembly
- a plexiglass supporting base for mounting purposes on the Laelaps body

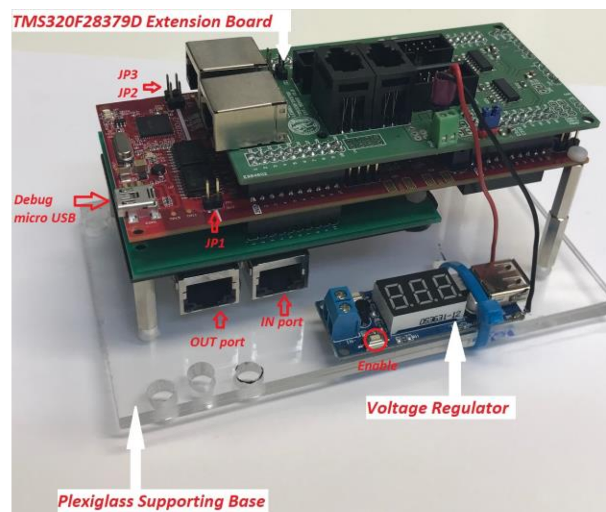


Figure 3-8. EtherCAT Control Tower Assembly.

Figure 3-9 shows the entire EtherCAT Control Tower Assembly mounted on Laelaps II robot. All four slave devices are connected to the EtherCAT network as shown in Figure 2-24 starting from the Hind Right Leg and ending with the Fore Right Leg.

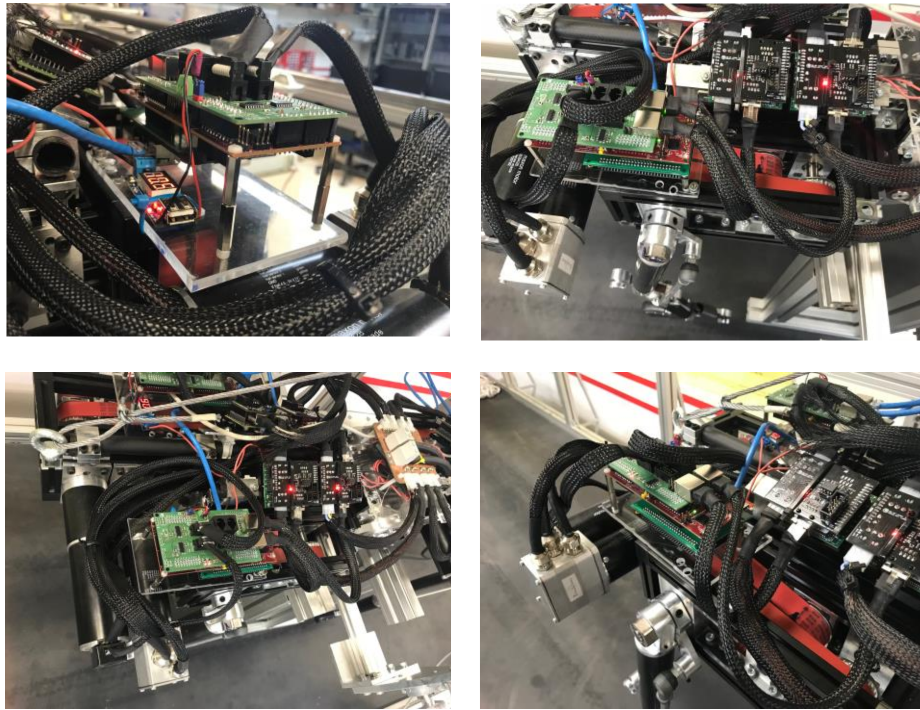


Figure 3-9. EtherCAT Control Tower Assembly on Laelaps II.

3.2 Motion Control of Laelaps via EtherCAT Solution Guide

This section describes the configuration process of the complete EtherCAT network. Developers are strongly advised to go through and emulate the instructions of EtherCAT Application Solution Guide before moving to this chapter to make sure that they have fully comprehended EtherCAT technology and its implementation process.

Importing the Project

1. Navigate to the following link [35] and download *EtherCAT Laelaps Motion Control* repository including a CCS project and an xml ENI file.

2. Import the *EtherCAT Laelaps Motion Control* CCS project into Code Composer Studio by following the instructions of Import CCS project into Code Composer Studio.

3. *Specify* and *Link* the desired Target Configuration of the development by following the instructions of Define and Select Target Configuration.

Firmware Structure

The project files are again separated into two main folders:

- *SPI_EtherCAT_slave_stack* which contains all the files that realize the *Generic EtherCAT Stack Layer* and the *User Application* (identical code structure with EtherCAT Application Solution Guide but different EtherCAT variables and control application).

- *hal* which contains all the necessary files that initialize and configure the MCU functionalities and control peripherals (ePWM, eQEP, DCL control, GPIO's etc) and materialize the *PDI and Hardware Abstraction Layer* (SPI functions to communicate with the ESC).

The main motor control features annexed in this project are thoroughly described in Chapter 5 of [5] including detailed information regarding the initialization and configuration procedure. This work will illustrate the modifications that needed to be implemented in order to enable extra functionalities required in this project, reduce the execution time and configure the control application.

In Table 3-1, developers can monitor the Output variables of the project as handled by the EtherCAT communication.

Table 3-1. EtherCAT Laelaps Motion Control Output variables.

Index	*Sub Index*	*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 [s]
0x7010	Record		Desired_x_value	
	0x01	INT32	Desired_x_value	Not read by SPI (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 SPI (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	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	Amplitude of x axis[cm]
	0x04	INT16	b_ellipse100	Amplitude of y axis[cm]
	0x05	INT16	traj_freq100	Trajectory's frequency [Hz] / 100
	0x06	INT16	phase_deg	Trajectory's initial phase of [deg]
	0x07	INT16	FlatnessParam100	Flatness parameter of y axis / 100

In order to reduce the EtherCAT frame payload, most of the variables are configured for their minimum unit values (ex highest practical precision required for the (x,y) center of the planning ellipsis w.r.t pint 0 –hip axis- is milimeters) in order to avoid using REAL type variables (64 bit). For example, for an Online Write of the output variable *Kp100_knee* with the value 10, it will be divided by 100 (as specified by Table 3-1) and translated into 0.1 for the Proportional gain of the knee motor within the stack. In Table 3-2, developers can monitor the Input variables of the project as handled by the EtherCAT communication.

Table 3-2. EtherCAT Laelaps Motion Control Input variables.

Index	*Sub Index*	*Data Type*	*Name*	*Comments*
0x6010	Record		hip_angle	
	0x01	INT16	hip_angle	Rotational angle of hip [deg] * 100
	0x02	INT16	Desired_hip_angle	Desired rotation angle of hip [deg] * 100
0x6012	Record		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	
	0x01	INT32	velocity_knee1000	Rotational speed of knee [rad/s] * 1000
	0x02	INT32	velocity_hip1000	Rotational speed of hip [rad/s] * 1000

The most significant functions implemented in *EtherCAT_Laelaps_Motion_Control.c* are:

- *APPL_GenerateMapping():* sends the Input and Output size of the process data interface - Figure 3-10.

```
250 ////////////////////////////////////////////////////////////////////////////////////////
251 /**
252 \return     0(ALSTATUSCODE_NOERROR), NOERROR_INWORK
253 \param      pInputSize  pointer to save the input process data length
254 \param      pOutputSize  pointer to save the output process data length
255
256 \brief    This function sends the process data sizes
257 *////////////////////////////////////////////////////////////////////////////////////////
258 UINT16 APPL_GenerateMapping(UINT16 *pInputSize,UINT16 *pOutputSize)
259 {
260     *pInputSize = 22;
261     *pOutputSize = 38;
262     return ALSTATUSCODE_NOERROR;
263 }
264
265 ////////////////////////////////////////////////////////////////////////////////////////
```

Figure 3-10. APPL_GenerateMapping() function.

- *APPL_InputMapping():* copies the Input variables from the local memory of the slave (Delfino MCU) to the ESC memory - Figure 3-11.

```
265 ////////////////////////////////////////////////////////////////////////////////////////
266 /**
267 \param      pData  pointer to input process data
268
269 \brief      This function copies the Input variables from the local memory to the ESC memory
270 *////////////////////////////////////////////////////////////////////////////////////////
271 void APPL_InputMapping(UINT16* pData)
272 {
273     uint16_t *pTmpData = (uint16_t *)pData;
274       /* TxPDO 1*/
275     memcpy(pTmpData, &hip_angle0x6010.hip_angle,SIZEOF(hip_angle0x6010.hip_angle));
276     pTmpData ++;
277     memcpy(pTmpData, &hip_angle0x6010.Desired_hip_angle,SIZEOF(hip_angle0x6010.Desired_hip_angle));
278     pTmpData ++;
279     memcpy(pTmpData, &FeedbackTime0x6012.Time,SIZEOF(FeedbackTime0x6012.Time));
280     pTmpData ++;
281     memcpy(pTmpData, &knee_angle0x6014.knee_angle,SIZEOF(knee_angle0x6014.knee_angle));
282     pTmpData ++;
283     memcpy(pTmpData, &knee_angle0x6014.Desired_knee_angle,SIZEOF(knee_angle0x6014.Desired_knee_angle));
284     pTmpData ++;
285     memcpy(pTmpData, &Commands0x6020.PWM10000_knee,SIZEOF(Commands0x6020.PWM10000_knee));
286     pTmpData ++;
287     memcpy(pTmpData, &Commands0x6020.PWM10000_hip,SIZEOF(Commands0x6020.PWM10000_hip));
288     pTmpData ++;
289     memcpy(pTmpData, &Velocity0x6030.velocity_knee1000,SIZEOF(Velocity0x6030.velocity_knee1000));
290     pTmpData += 2;
291     memcpy(pTmpData, &Velocity0x6030.velocity_hip1000,SIZEOF(Velocity0x6030.velocity_knee1000));
292 }
293
294 ////////////////////////////////////////////////////////////////////////////////////////
```

Figure 3-11. **APPL_InputMapping() function**.

- *APPL_OutputMapping():* copies the Output variables from the ESC memory to the local memory of the Delfino MCU slave to update their values within the application - Figure 3-12. It is worth mentioning that *Desired_x_value* and *Desired_y_value* output variables purposely remained in the EtherCAT frame and were not removed for future developers who want to quickly add two more 32bit variables (or even four 16bit variables with minor modifications). Yet, in order to reduce the execution time of APPL_OutputMapping() function and consequently EtherCAT's cycle time, these two variables are not being updated within the stack. The only necessary alteration to enable these variables is to uncomment and comment the indicated parts of the function.

```
294 ////////////////////////////////////////////////////////////////////////////////////////
295 /**
296 \param      pData  pointer to output process data
297
298 \brief    This function will copies the outputs from the ESC memory to the local memory
299             to the hardware
300 *////////////////////////////////////////////////////////////////////////////////////////
301 void APPL_OutputMapping(UINT16* pData)
302 {
303     uint16_t *pTmpData = (uint16_t *)pData;// allow byte processing
304     uint16_t data = 0;
305
306     /* RxPDO */
307     data = (*(volatile uint16_t *)pTmpData);
308     (Buttons0x7000.State_Machine) = data & 0x1;
309     data = data >> 1;
310     (Buttons0x7000.Initialize_clock) = data  & 0x1;
311     data = data >> 1;
312     (Buttons0x7000.Initialize_angles) = data  & 0x1;
313     data = data >> 1;
314     (Buttons0x7000.Inverse_Kinematics) = data  & 0x1;
315     data = data >> 1;
316     (Buttons0x7000.Blue_LED) = data  & 0x1;
317     data = data >> 1;
318     (Buttons0x7000.Red_LED) = data  & 0x1;
319     data = data >> 1;
320     (Buttons0x7000.Button1) = data  & 0x1;
321     data = data >> 1;
322     (Buttons0x7000.Button2) = data  & 0x1;
323     data = data >> 1;
324     (Buttons0x7000.Transition_time) = data  & 0xFF;
325
326     /* Uncomment the following lines if you want to read Desired_x_value and Desired_y_value
327     //pTmpData++;
328     //memcpy(&Desired_x_value0x7010.Desired_x_value,pTmpData,SIZEOF(Desired_x_value0x7010.Desired_x_value));
329     //pTmpData += 2;
330     //memcpy(&TargetMode0x7012.FilterBandwidth,pTmpData,SIZEOF(TargetMode0x7012.FilterBandwidth));
331     //pTmpData ++;
332     //memcpy(&Desired_y_value0x7014.Desired_y_value,pTmpData,SIZEOF(Desired_y_value0x7014.Desired_y_value));
333     //pTmpData += 2;*/
334
335     /* Comment the 3 lines bellow if you want to read Desired_x_value and Desired_y_value */
336     pTmpData += 3; //Don't read Desired_x_value to reduce execution time
337     memcpy(&TargetMode0x7012.FilterBandwidth,pTmpData,SIZEOF(TargetMode0x7012.FilterBandwidth));
338     pTmpData += 3; //Don't read Desired_y_value to reduce execution time
339
340     memcpy(&ControlGains0x7020.Kp100_knee,pTmpData,SIZEOF(ControlGains0x7020.Kp100_knee));
341     pTmpData ++;
342     memcpy(&ControlGains0x7020.Kd1000_knee,pTmpData,SIZEOF(ControlGains0x7020.Kd1000_knee));
343     pTmpData ++;
344     memcpy(&ControlGains0x7020.Ki100_knee,pTmpData,SIZEOF(ControlGains0x7020.Ki100_knee));
345     pTmpData ++;
346     memcpy(&ControlGains0x7020.Kp100_hip,pTmpData,SIZEOF(ControlGains0x7020.Kp100_hip));
347     pTmpData ++;
348     memcpy(&ControlGains0x7020.Kd1000_hip,pTmpData,SIZEOF(ControlGains0x7020.Kd1000_hip));
349     pTmpData ++;
350     memcpy(&ControlGains0x7020.Ki100_hip,pTmpData,SIZEOF(ControlGains0x7020.Ki100_hip));
351     pTmpData ++;
352
353     memcpy(&TrajectoryParameters0x7030.x_cntr_traj1000,pTmpData,SIZEOF(TrajectoryParameters0x7030.x_cntr_traj1000));
354     pTmpData ++;
355     memcpy(&TrajectoryParameters0x7030.y_cntr_traj1000,pTmpData,SIZEOF(TrajectoryParameters0x7030.y_cntr_traj1000));
356     pTmpData ++;
357     memcpy(&TrajectoryParameters0x7030.a_ellipse100,pTmpData,SIZEOF(TrajectoryParameters0x7030.a_ellipse100));
358     pTmpData ++;
359     memcpy(&TrajectoryParameters0x7030.b_ellipse100,pTmpData,SIZEOF(TrajectoryParameters0x7030.b_ellipse100));
360     pTmpData ++;
361     memcpy(&TrajectoryParameters0x7030.traj_freq100,pTmpData,SIZEOF(TrajectoryParameters0x7030.traj_freq100));
362     pTmpData ++;
363     memcpy(&TrajectoryParameters0x7030.phase_deg,pTmpData,SIZEOF(TrajectoryParameters0x7030.phase_deg));
364     pTmpData ++;
365     memcpy(&TrajectoryParameters0x7030.FlatnessParam100,pTmpData,SIZEOF(TrajectoryParameters0x7030.FlatnessParam100));
366 }
367
368 ////////////////////////////////////////////////////////////////////////////////////////
```

Figure 3-12. APPL_OutputMapping() function.

- *APPL_Application():* contains the User - Control Application. This function materializes the trajectory planning control which is initiated every time the State_Machine EtherCAT output variable is set to 1 and is terminated when it is set to 0 (the time variable starts and resets accordingly), the inverse kinematics algorithm if the desired end effector position is within the workspace of the leg and most importantly, updates all EtherCAT variables that are exploited within the stack (Figure 3-13).

```
368 ///////////////////////////////////////////////////////////////////////////////////////
369 /**
370 \brief    This function will called from the synchronisation ISR
371             or from the mainloop if no synchronisation is supported executing the control application
372 *///////////////////////////////////////////////////////////////////////////////////////
373 void APPL_Application(void)
374 {
375 
376     // State indicates Configuration State (0) or Operational State (1)
377     State_Machine = Buttons0x7000.State_Machine;
378     if (Buttons0x7000.Transition_time)
379         Transition_time = (float)Buttons0x7000.Transition_time; //Update the transition time of the smoothing variables
380 
381     //Retrieve the cycle time for the time variable
382     unsigned long cycleTime = sSyncManOutPar.u32Sync0CycleTime;
383     float dt = cycleTime/1000000000.0f; //Calculate the dt [s] depending on EtherCAT cycle time. Used in time variable t [s]
384 
385     GPIO_WritePin(31, !Buttons0x7000.Blue_LED); // Turn on/off blue LED (GPIO31) depending on output of Blue_LED for debug purpose
386     GPIO_WritePin(34, !Buttons0x7000.Red_LED); // Turn on/off red LED (GPIO31) depending on output of Red_LED for debug purpose
387 
388     //Rotational angles of the leg to the Master
389     hip_angle0x6010.hip_angle = (int)hip_angle100;   //Send hip angle [deg] to Master (max value that can be measured +- 327 degrees)
390     knee_angle0x6014.knee_angle = (int)knee_angle100; //Send knee angle [deg] to Master (max value that can be measured +- 327 degrees)
391 
392     //Rotational speed of the leg to the Master
393     Velocity0x6030.velocity_knee1000 = velocity_knee1000; //Send rotational speed of knee angle [rad/s] to Master
394     Velocity0x6030.velocity_hip1000 = velocity_hip1000;   //Send rotational speed of hip angle [rad/s] to Master
395 
396     //Outputs of PID controllers to motors sent to the Master
397     Commands0x6020.PWM10000_knee = command1; //Send output of PID controller for knee motor to Master
398     Commands0x6020.PWM10000_hip = command2; //Send output of PID controller for hip motor to Master
399 
400     if (TargetMode0x7012.FilterBandwidth)
401         FilterBandwidth = TargetMode0x7012.FilterBandwidth;
402 
403     /* State Machine dependent */
404     if (State_Machine){ //Operational State
405         //Start clock if State_Machine is 1
406         t = t + dt; //time variable in seconds
407         float angle = traj_vel * t + phase; //calculate angle of the ellipsis
408         if (fmodf(angle,2.0f*pi) < pi)
409             b_ellipse_flat = flatParam * b_ellipse; //simulate the ground
410         else
411             b_ellipse_flat = b_ellipse;
412         x_value = x_traj_cntr + a_ellipse * cosf(angle);      //End Effector's x axis value
413         y_value = y_traj_cntr + b_ellipse_flat * sinf(angle); //End Effector's x axis value
414 
415         if (sqrt(x_value*x_value+y_value*y_value) > rmin && sqrt(x_value*x_value+y_value*y_value) < rmax){ //if (x,y) within the workspace
416             // Inverse Kinematics calculations (included math.h for the mathematical functions)
417             float c_invk = (y_value*y_value + x_value*x_value - l1*l1 - l2*l2)/(2*l1*l2);
418             float s_invk = -sqrtf(1-c_invk*c_invk);
419             float knee_angle = - atan2f(y_value,x_value) + atan2f(l1 * s_invk, l2 + l1 * c_invk) + pi/2;
420             float hip_angle = knee_angle - atan2f(s_invk, c_invk);
421         #ifdef RIGHT_LEG //Right leg configuration
422             Desired_hip_angle=-hip_angle/(2.0f*pi);  //normalized value for PID controller (like: (hip_angle*180/pi)/360))
423             Desired_knee_angle=-knee_angle/(2.0f*pi); //normalized value for PID controller (like: (hip_angle*180/pi)/360))
424         #elif LEFT_LEG   //Left leg configuration
425             Desired_hip_angle=hip_angle/(2.0f*pi);  //normalized value for PID controller (like: (hip_angle*180/pi)/360))
426             Desired_knee_angle=knee_angle/(2.0f*pi); //normalized value for PID controller (like: (hip_angle*180/pi)/360))
427         #endif
428         }
429     }else{ // Configuration State
430         t=0.0f; //Reset clock if State_machine is 0
431     }
432 
433     //Send the time to Master
434     FeedbackTime0x6012.Time = (unsigned int)(t*100.0f); //For debug purposes to secure proper EtherCAT implementation
435 
436     //Desired rotational angle of the leg to the Master
437     hip_angle0x6010.Desired_hip_angle = (int)(Desired_hip_angle * 360.0f * 100.0f);   //Send Desired hip angle to Master
438     knee_angle0x6014.Desired_knee_angle = (int)(Desired_knee_angle * 360.0f * 100.0f); //Send Desired knee angle to Master
439 
440     // Update Control Gains using the Smooth Transition Functions (CalculateAdder and UpdateValue)
441     if (ControlGains0x7020.Kp100_knee != Kp_knee_old)
442         Kp_knee_adder = CalculateAdder(ControlGains0x7020.Kp100_knee/100.0f,Kp_knee,Transition_time,dt);
443     Kp_knee = UpdateValue(Kp_knee,ControlGains0x7020.Kp100_knee/100.0f,Kp_knee_adder);
444     if (ControlGains0x7020.Kd1000_knee != Kd_knee_old)
445             Kd_knee_adder = CalculateAdder(ControlGains0x7020.Kd1000_knee/1000.0f,Kd_knee,Transition_time,dt);
446     Kd_knee = UpdateValue(Kd_knee,ControlGains0x7020.Kd1000_knee/1000.0f,Kd_knee_adder);
447     if (ControlGains0x7020.Kp100_hip != Kp_hip_old)
448             Kp_hip_adder = CalculateAdder(ControlGains0x7020.Kp100_hip/100.0f,Kp_hip,Transition_time,dt);
449     Kp_hip = UpdateValue(Kp_hip,ControlGains0x7020.Kp100_hip/100.0f,Kp_hip_adder);
450     if (ControlGains0x7020.Kd1000_hip != Kd_hip_old)
451             Kd_hip_adder = CalculateAdder(ControlGains0x7020.Kd1000_hip/1000.0f,Kd_hip,Transition_time,dt);
452     Kd_hip = UpdateValue(Kd_hip,ControlGains0x7020.Kd1000_hip/1000.0f,Kd_hip_adder);
453 
454     // Update Parameters of Ellipsis using the Smooth Transition Functions (CalculateAdder and UpdateValue)
455     x_traj_cntr = TrajectoryParameters0x7030.x_cntr_traj1000/1000.0f;
456     y_traj_cntr = TrajectoryParameters0x7030.y_cntr_traj1000/1000.0f;
457     if (TrajectoryParameters0x7030.a_ellipse100 != a_ellipse_old)
458         a_ellipse_adder = CalculateAdder(TrajectoryParameters0x7030.a_ellipse100/100.0f,a_ellipse,Transition_time,dt);
459         a_ellipse = UpdateValue(a_ellipse,TrajectoryParameters0x7030.a_ellipse100/100.0f,a_ellipse_adder);
460     if (TrajectoryParameters0x7030.b_ellipse100 != b_ellipse_old)
461         b_ellipse_adder = CalculateAdder(TrajectoryParameters0x7030.b_ellipse100/100.0f,b_ellipse,Transition_time,dt);
462         b_ellipse = UpdateValue(b_ellipse,TrajectoryParameters0x7030.b_ellipse100/100.0f,b_ellipse_adder);
463     if (TrajectoryParameters0x7030.traj_freq100 > 0)
464          traj_freq = TrajectoryParameters0x7030.traj_freq100/100.0f; //trajectory frequency
465     traj_vel = traj_freq*2.0f*pi; // rotational speed = 2*pi*f
466     phase = ((float)TrajectoryParameters0x7030.phase_deg)*pi/180.0f;
467     flatParam = TrajectoryParameters0x7030.FlatnessParam100/100.0f; // flatness of the toe trajectory in stance phase
468 
469     //Save old parameters to compare with the new values
470     Kp_knee_old = ControlGains0x7020.Kp100_knee;
471     Kd_knee_old = ControlGains0x7020.Kd1000_knee;
472     Kp_hip_old = ControlGains0x7020.Kp100_hip;
473     Kd_hip_old = ControlGains0x7020.Kd1000_hip;
474     a_ellipse_old = TrajectoryParameters0x7030.a_ellipse100;
475     b_ellipse_old = TrajectoryParameters0x7030.b_ellipse100;
476 }
477 
478 ///////////////////////////////////////////////////////////////////////////////////////
```

Figure 3-13. APPL_Application() function.

All EtherCAT variables that affect the control output of the PIV controller are updated using a function that makes this transition smoother (linear with time) and minimizes the possibility of sharp leg movements. Note that the transition time of these functions is also handled in real time by EtherCAT (Transition_time output variable) and that there is a predefined symbol (RIGHT_LEG – LEFT_LEG) to determine whether the project is built for a right or a left Laelaps leg shown in Figure 3-13.

PIV Motor Controller

In our application, C2000 Digital Control Library [11] is exploited to implement the desired controller. Extending the initial attempt described in Chapter 5 of [5] we have configured two linear PIV controllers (Proportional - Velocity – Integral), one for each joint of every leg. The function which realizes this controller is called DCL_runPID_C1(), it is coded in assembly and it's block diagram is the one depicted in Figure 3-14. It is worth mentioning that the controller function includes a digital low-pass filter to avoid amplification of unwanted high frequency noise. The filter is a simple first order lag filter with differentiator, converted into discrete form using the Tustin transform. Hence, the velocity estimation is executed within this function and it is not supplied externally. For more information, refer to page 14 of [11] .

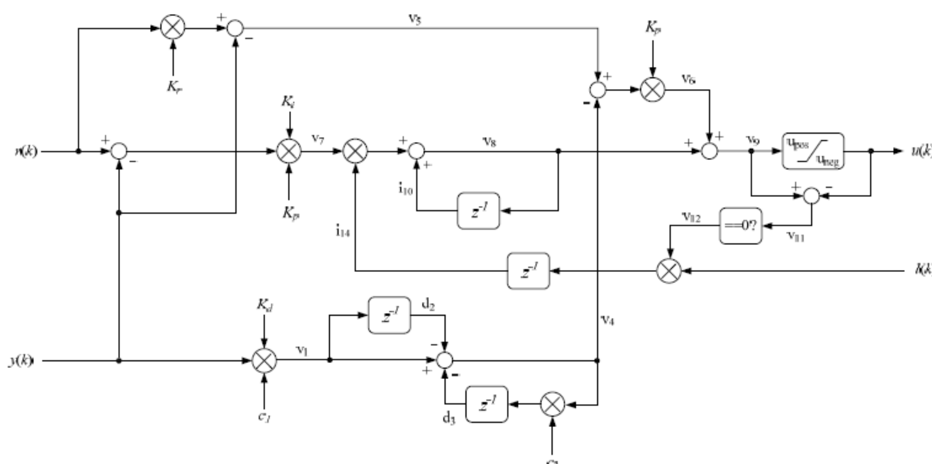


Figure 3-14. DCL_runPID_C1() block diagram.

In case users desire to realize a linear PID (Proportional - Derivative – Integral) controller (DCL_runPID_C4() function) instead of the already defined PIV, they should comment out lines 932 of Figure 3-15 and 1021 of Figure 3-16 and uncomment lines 931 and 1020 respectively. Developers who desire to realize the above PIV controller in a Matlab simulation for testing purposes, can refer to Matlab PIV controller simulation.

The control-related part of the firmware is realized in **etherCAT_slave_c28x_hal.c** (under **hal** folder). The most important functions – interrupt service routines (ISRs) configured are:

- **Epwm1_isr():** realizes the PIV controller for the knee motor (brushed) - Figure 3-15.

```
886 //-------------------------------------------------------------------------------
887 //      ISR to handle EPWM1 ISR
888 //      prdTick - EPWM1 Interrupts once every 3 QCLK counts (one period)
889 //-------------------------------------------------------------------------------
890 interrupt void epwm1_isr(void)
891 {
892     if (int1cnt == 2)   //10khz control loop frequency
893     {
894         EALLOW;
895         // Control Motor 1 Knee (Brushed)
896         // 1st Encoder Read (2000 counts per revolution)
897         // Read raw values of eQEPs
898         qep_posspeed.calc(&qep_posspeed);
899 
900         //Read raw position eQEP1
901         if ((unsigned long int) qep_posspeed.raw_pos1 > pos_init)
902             raw_pos1 = (unsigned long int) qep_posspeed.raw_pos1 - pos_init;
903         else
904             raw_pos1 = -(pos_init - (unsigned long int) qep_posspeed.raw_pos1);
905 
906         //Calculate rotational Speed of knee
907         velocity_knee1000 = (long)(qep_posspeed.Speed_pr1*1000.0f);
908 
909         //Translate raw value to normalized angle (gear ratio for knee added)
910         float ngwnia1 = (raw_pos1 * 8.0f * 26.0f)/(2000.0f * 343.0f * 48.0f); //normalized knee angle
911         knee_angle100 = ngwnia1 * 360.0f*100.0f; //knee angle for the EtherCAT Master
912 
913         float ftc = 1.0f/(FilterBandwidth*2.0f*pi); // ftc = filter time constant
914 
915         // Update gains for PID Control of knee
916         pid1.Kp=Kp_knee;
917         pid1.Kd=Kd_knee;
918         pid1.Ki=Ki_knee;
919         if (!pid1.Ki)
920             pid1.i10=0.0f;    //if I want to remove Ki term, the control output must clear the integral path
921 
922         //First order lag filter with differentiator coefficients
923         pid1.c1=2.0f/(T+2.0f*ftc);
924         pid1.c2=(T-2.0f*ftc)/(T+2.0f*ftc);
925 
926         //Max and min values of knee's PWM command
927         pid1.Umax=Umax_knee; //max
928         pid1.Umin=-Umax_knee;
929 
930         //Run PID controller
931         //uk1 = DCL_runPID_C4(&pid1, Desired_knee_angle, ngwnia1, lk1); //PID controller
932         uk1 = DCL_runPID_C1(&pid1, Desired_knee_angle, ngwnia1, lk1); //PIV controller
933         command1=(int)(uk1*10000.0f);
934 
935         // Set direction of knee motor
936         if (uk1 >= 0.0f)
937             GPIO_WritePin(DIR1_GPIO, 0);
938         else
939         {
940             GPIO_WritePin(DIR1_GPIO, 1);
941             uk1 = -uk1;
942         }
943 
944         // Update PWM duty cycle only when Operational mode(State_Machine=1)
945         if (State_Machine) //Operational mode
946             EPwm1Regs.CMPA.bit.CMPA = (1.0f - uk1) * SP;
947         else //Configuration mode
948             EPwm1Regs.CMPA.bit.CMPA = SP;
949 
950         // Reset interrupt counter
951         int1cnt = 0;
952 
953         //Uncomment the following lines to enable reading the values of the 3rd encoder
954         //---------------------------------------------------------------------------
955         //          // 3rd Encoder Read
956         //          // Read raw values of eQEPs
957         //          qep_posspeed.calc(&qep_posspeed);
958         //
959         //          //Read raw position eQEP3
960         //          if ((unsigned long int) qep_posspeed.raw_pos3 > pos_init)
961         //              raw_pos3 = (unsigned long int) qep_posspeed.raw_pos3 - pos_init;
962         //          else
963         //              raw_pos3 = -(pos_init - (unsigned long int) qep_posspeed.raw_pos3);
964         //---------------------------------------------------------------------------
965     }
966     int1cnt++;
967 
968     EDIS;
969     // Clear INT flag for this timer
970     EPwm1Regs.ETCLR.bit.INT = 1;
971     //
972     // Acknowledge this __interrupt to receive more __interrupts from group 3
973     //
974     PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
975 }
```

Figure 3-15. Epwm1_isr() function.

▪ **Epwm2_isr():** realizes the PIV controller for the hip motor (brushless) - Figure 3-16.

```
976 //-------------------------------------------------------------------------------
977 //      ISR to handle EPWM2 ISR
978 //      prdTick - EPWM2 Interrupts once every 3 QCLK counts (one period)
979 //-------------------------------------------------------------------------------
980 interrupt void epwm2_isr(void)
981 {
982     if (int2cnt == 2)   //10khz control loop frequency
983     {
984         EALLOW;
985         // Control Motor 2 Hip (Brushless)
986         // 2nd Encoder Read (2000 counts per revolution)
987         // Read raw values of eQEPs
988         qep_posspeed.calc(&qep_posspeed);
989         if ((unsigned long int) qep_posspeed.raw_pos2 > pos_init)
990             raw_pos2 = (unsigned long int) qep_posspeed.raw_pos2 - pos_init;
991         else
992             raw_pos2 = -(pos_init - (unsigned long int) qep_posspeed.raw_pos2);
993
994         //Calculate rotational Speed of hip (gear ration for hip added)
995         velocity_hip1000 = (long)(qep_posspeed.Speed_pr2*1000.0f);
996
997         //Translate raw value to normalized angle
998         float ngwnia2 = (raw_pos2 * 12.0f * 26.0f)/(2000.0f * 637.0f * 48.0f);//normalized knee angle
999         hip_angle100 = ngwnia2 * 360.0f*100.0f;  //hip angle for the EtherCAT Master
1000
1001        float ftc=1.0f/(FilterBandwidth*2.0f*pi); // ftc = filter time constant
1002
1003        // Update gains for PID Control of hip
1004        pid2.Kp=Kp_hip;
1005        pid2.Kd=Kd_hip;
1006        pid2.Ki=Ki_hip;
1007
1008        if (!pid2.Ki)
1009            pid2.i10=0.0f; //if I want to remove Ki term, the control output must clear the integral path
1010
1011        //First order lag filter with differentiator coefficients
1012        pid2.c1=2.0f/(T+2.0f*ftc);
1013        pid2.c2=(T-2.0f*ftc)/(T+2.0f*ftc);
1014
1015        //Max and min values of hip's PWM command
1016        pid2.Umax=Umax_hip;
1017        pid2.Umin=-Umax_hip;
1018
1019        // Run PID controller
1020        //uk2 = DCL_runPID_C4(&pid2, rk2, ngwnia2, lk2); //PID Controller
1021        uk2 = DCL_runPID_C1(&pid2, Desired_hip_angle, ngwnia2, lk2); //PIV Controller
1022        command2=(int)(uk2*10000.0f);
1023
1024        // Set direction of hip motor
1025        if (uk2 >= 0.0f)
1026            GPIO_WritePin(DIR2_GPIO, 0);
1027        else
1028        {
1029            GPIO_WritePin(DIR2_GPIO, 1);
1030            uk2 = -uk2;
1031        }
1032
1033        // Update PWM duty cycle only when Operational mode(State_Machine=1)
1034        if (State_Machine) //Operational mode
1035            EPwm2Regs.CMPA.bit.CMPA = (1.0f - uk2) * SP;
1036        else  //Configuration mode
1037            EPwm2Regs.CMPA.bit.CMPA = SP;
1038
1039        // Reset interrupt counter
1040        int2cnt = 0;
1041
1042    }
1043    int2cnt++;
1044
1045    EDIS;
1046    // Clear INT flag for this timer
1047    EPwm2Regs.ETCLR.bit.INT = 1;
1048    //
1049    // Acknowledge this __interrupt to receive more __interrupts from group 3
1050    //
1051    PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
1052 }
```

Figure 3-16. Epwm2_isr() function.

All variables responsible for the PIV controller are handled via EtherCAT, therefore developers can alter all parameters in real time. The only control parameters that are predefined and users must download the firmware again to all slaves in order to modify them are the maximum allowed values of PWM signals to both

motors, namely Umax_knee (38,25%) and Umax_hip (41,17%). Their definition is located at the global variable declaration section of etherCAT_slave_c28x_hal.c based, on each motor's maximum allowed continuous current. Refer to Laelaps II motors and gearheads for more information regarding the selected motors and gearheads.

Rotational Speed Calculation

Estimating the rotational speed of each joint is crucial in this application to ensure that a motor does not exceed the maximum allowable speed limit. For this purpose, a custom function was created to calculate the velocity of both joints using the eQEP Edge Capture Unit – Low Speed Calculation feature (refer to *TMS320x2833x, 2823x Enhanced Quadrature Encoder Pulse (eQEP) Module*) because the encountered rotational velocities are relatively low. This approximation is based on (3-7) where on every unit position event (X reaches the predefined number of quadrature edges [UPPS]) the capture timer [QCTMR] value is latched into the capture period register [QCPRD] and then [QCTMR} is reset. Then, the velocity is converted from [counts/time_register] to [rad/s] using the SpeedScaler as shown in Figure 3-17.

$$v(k) = \frac{X}{t(k) - t(k-1)} \tag{3-7}$$

In Table 3-3 you can observe a characteristic example of parameters used in a leg of Laelaps II experiment to gain a general idea of the values that were exploited. In later chapter, a more specific description of the parameters used in every leg will be presented.

Table 3-3. Benchmark parameters in Laelaps II experiment.

<table>
<tr><th colspan="2">Parameters</th><th>Leg Value</th></tr>
<tr><td rowspan="7">Trajectory Paramaters</td><td>x centre</td><td>0 cm</td></tr>
<tr><td>y centre</td><td>59.5 cm</td></tr>
<tr><td>a ellipse</td><td>3 cm</td></tr>
<tr><td>b ellipse</td><td>4 cm</td></tr>
<tr><td>Frequency</td><td>0.8 Hz</td></tr>
<tr><td>Phase [deg]</td><td>180</td></tr>
<tr><td>Flatness</td><td>0</td></tr>
<tr><td rowspan="3">Control Gains of Knee</td><td>Kp_knee</td><td>40</td></tr>
<tr><td>Kd_knee</td><td>0.03</td></tr>
<tr><td>Ki_knee</td><td>0</td></tr>
<tr><td rowspan="3">Control Gains of Hip</td><td>Kp_hip</td><td>40</td></tr>
<tr><td>Kd_hip</td><td>0.03</td></tr>
<tr><td>Ki_hip</td><td>0</td></tr>
<tr><td rowspan="2">PWM max values</td><td>Knee</td><td>38.25%</td></tr>
<tr><td>Hip</td><td>41.17%</td></tr>
<tr><td colspan="2">Filter Bandwidth Frequency</td><td>20 Hz</td></tr>
<tr><td colspan="2">Loop Frequency of EtherCAT</td><td>2.5 kHz</td></tr>
</table>

```
124 //
125 // POSSPEED_Calc - Perform the position calculations
126 //
127 void POSSPEED_Calc(POSSPEED *p)
128 {
129     unsigned int temp1,temp2;
130 
131     //Read raw value of position and save in gwnia variable
132     p->raw_pos1 = EQep1Regs.QPOSCNT;
133     p->raw_pos2 = EQep2Regs.QPOSCNT;
134     p->raw_pos3 = EQep3Regs.QPOSCNT;
135 
136     p->DirectionQep1 = EQep1Regs.QEPSTS.bit.QDF;  // Motor direction: 0=CCW/reverse, 1=CW/forward
137     p->DirectionQep2 = EQep2Regs.QEPSTS.bit.QDF;  // Motor direction: 0=CCW/reverse, 1=CW/forward
138 
139     //
140     // Low-speed computation using QEP capture counter for eQEP1
141     //
142     if(EQep1Regs.QEPSTS.bit.UPEVNT == 1)               // Unit position event
143     {
144         if(EQep1Regs.QEPSTS.bit.COEF == 1)             // Capture overflow, saturate the result
145         {
146             temp1 = 0xFFFF;
147             EQep1Regs.QEPSTS.bit.COEF = 1;             // Clear overflow error flag
148         }
149         else                                           // No Capture overflow
150         {
151             temp1 = EQep1Regs.QCPRD; // temp1 = t2-t1
152         }
153         if(EQep1Regs.QEPSTS.bit.CDEF == 1)             // Direction change, make velocity 0
154         {
155             p->Speed_pr1 = 0.0f;
156             EQep1Regs.QEPSTS.bit.CDEF = 1;             // Clear direction error flag
157         }
158         else
159         {
160             //
161             // p->Speed_pr = p->SpeedScaler/temp1
162             //
163             p->Speed_pr1 = p->SpeedScaler/((float)temp1); //speed in rad/sec
164         }
165         //
166         // Convert p->Speed_pr1 depending on direction
167         //
168         if(p->DirectionQep1 == 0)  // Reverse direction = negative
169         {
170             p->Speed_pr1 = - p->Speed_pr1;
171         }
172         EQep1Regs.QEPSTS.bit.UPEVNT = 1; // Clear Unit position event flag
173     }
174 
175     //
176     // Low-speed computation using QEP capture counter for eQEP2
177     //
178     if(EQep2Regs.QEPSTS.bit.UPEVNT == 1)               // Unit position event
179     {
180         if(EQep2Regs.QEPSTS.bit.COEF == 1)             // Capture overflow, saturate the result
181         {
182             temp2 = 0xFFFF;
183             EQep2Regs.QEPSTS.bit.COEF = 1;             // Clear overflow error flag
184         }
185         else                                           // No Capture overflow
186         {
187             temp2 = EQep2Regs.QCPRD; // temp2 = t2-t1
188         }
189         if(EQep2Regs.QEPSTS.bit.CDEF == 1)             // Direction change, make velocity 0
190         {
191             p->Speed_pr2 = 0.0f;
192             EQep2Regs.QEPSTS.bit.CDEF = 1;             // Clear direction error flag
193         }
194         else
195         {
196             //
197             // p->Speed_pr = p->SpeedScaler/temp1
198             //
199             p->Speed_pr2 = p->SpeedScaler/((float)temp2); //speed in rad/sec
200         }
201         //
202         // Convert p->Speed_pr2 depending on direction
203         //
204         if(p->DirectionQep2 == 0)  // Reverse direction = negative
205         {
206             p->Speed_pr2 = - p->Speed_pr2;
207         }
208         EQep2Regs.QEPSTS.bit.UPEVNT = 1; // Clear Unit position event flag
209     }
210 }
```

Figure 3-17. Position & Rotational Speed calculation.

The above configuration allows measuring the range of rotational velocities described in Table 3-4, which means that velocities outside the illustrated range will either be regarded as 0 (if they are below the minimum value) or as the maximum value (if they are above the limit). In both hip and knee columns, the speed calculations are after the gearhead and pulley transmition of the leg, illustrating the actual values of the joints.

Table 3-4. Velocity calculation range.

	Calculated Velocity [rad/s]	Hip joint [rad/s]	Knee joint [rad/s]
Max value	19634.95	248.06	200.36
Min value	0.2996	0.0038	0.0031

Configuring the Project

1. Select the desired *Build Configuration* of the imported project. EtherCAT_Laelaps_Motion_Control project contains four separate Build Configurations, two for each leg side (RAM and FLASH) namely:

- Left _Leg_FLASH_LAUNCHXLF2837D_SPIA
- Left_Leg_RAM_LAUNCHXLF2837D_SPIA
- Right_Leg_FLASH_LAUNCHXLF2837D_SPIA
- Right_Leg_RAM_LAUNCHXLF2837D_SPIA

Note that in order to be able to use the DCL library in Flash configuration, we had to altered the relative linker command file which allocated the memory of the MCU (2837x_FLASH_lnk_cpu1.cmd under cmd folder) and add the highlighted snippet at the end of the file as shown in Figure 3-18.

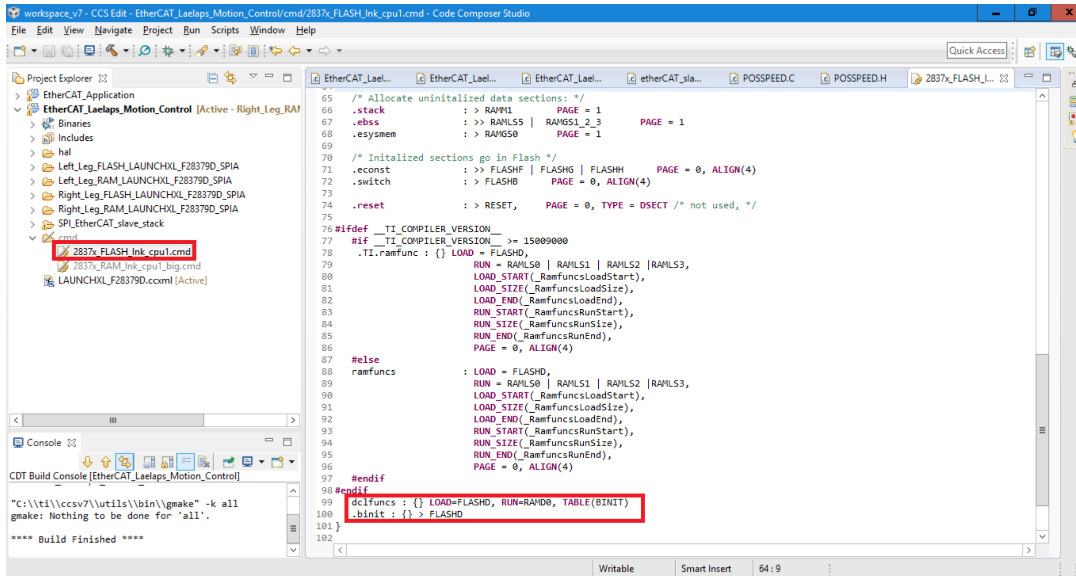


Figure 3-18. Modified linker command file to enable DCL functions.

2. Make sure that JP1, JP2 and JP3 jumpers are removed (as shown in Figure 3-8) because the microUSB port cannot supply enough current for the EtherCAT Control Tower Assembly which requires around 0.6 Ampers and external power supply is needed.

3. Turn on the Logic Power supply (>5 V, for example 10 V) connected to all EtherCAT Control Tower Assemblies of Laelaps and turn on the desired ones by pressing the *Enable* button of the voltage regulators indicated in Figure 3-8. If you have disassembled an EtherCAT Control Tower Assembly from Laelaps II for

testing reasons, because all available assemblies are mounted on the robot, you should connect the Voltage Regulator to the Logic Power Supply and press the Enable button as shown in Figure 3-19.

4. Connect the desired launchpad to the PC using the microUSB cable and *Build* the project. If no error occurs, then download the project into the launchpad by clicking the *Debug* button as shown in Figure 2-40.

5. The launchpad consists of two cores, thus you need to select CPU1 and click OK from the pop up window as illustrated in Figure 2-41.

6. The Code Composer Studio will then automatically turn into CCS Debug mode. When the project is downloaded (obviously, downloading the Flash project lasts a lot longer than RAM) click the Resume button as adumbrated in Figure 2-42 and the slave side is now up and running.

7. Repeat steps 7 → 9 for every slave in the network only if the first slave of the robot (Hind Right Leg's) is tested and properly working. Moreover, do not forget that RAM build configurations are only available until power off, so if you want to download a project to all slaves, FLASH configuration is necessary.

8. Connect the *IN* port of the first slave's FB1111-0141 ESC, using a suitable Ethernet cable (see EtherCAT Master Requirements) with the PC as illustrated in Figure 3-19 and start TwinCAT XAE (VS 2013). Since we are only configuring the EEPROM of one slave at a time, make sure that the OUT port of the slave is not connected to any other slave.

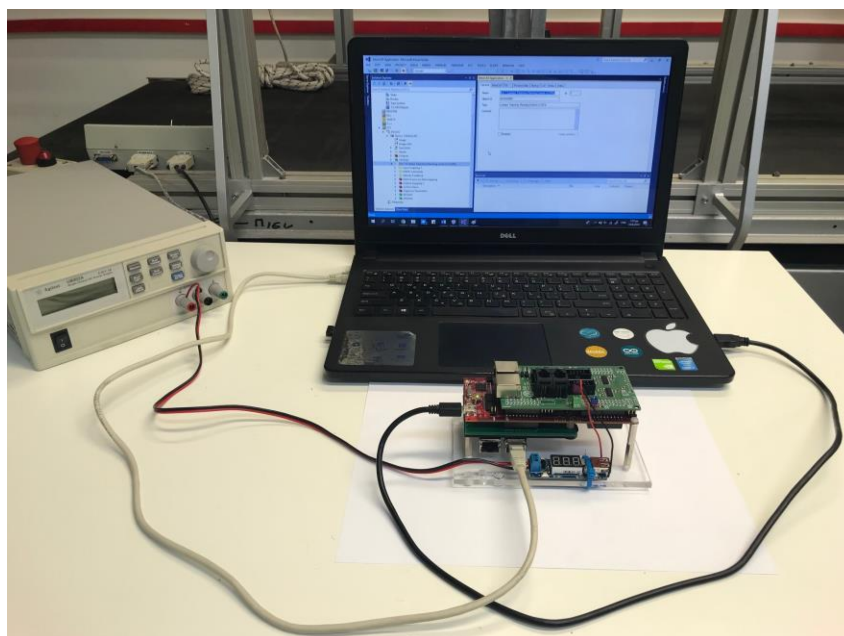


Figure 3-19. EtherCAT Control Tower Assembly wired.

9. Select *File* > *New* > *Project* as shown in Figure 2-44.

10. From the *TwinCAT Projects tab* select *TwinCAT XAE Project (XML format)*, name the project (ex EtherCAT Initialization) and click *OK* as illustrated in Figure 2-45.

11. From the *Solution Explorer* window expand the *I/O* element, right click on *Devices* and select *Add New Item* (Figure 2-46).

12. Select *EtherCAT Master* and click *OK* (Figure 2-47). This way, your PC is now configured as an EtherCAT Master device.

13. Navigate to the location of the downloaded folder from Bitbucket, copy the *EtherCAT Laelaps Motion Control v4 (SPI).xml* file and paste it at the following location *C:\TwinCAT\3.1\Config\Io\EtherCAT*

14. In Visual Studio, select *TWINCAT* > *EtherCAT Devices* > *Reload Device Descriptions* (Figure 2-48)

15. Click once on the *Device 1 (EtherCAT)* item of the *Solution Explorer* tree and the *Scan* button (Figure 2-49) will instantly become clickable. Select the *Scan* feature (automatic scan for slave devices) and select *No* in the pop up window asking whether to *Activate Free Run*. A slave (TwinCAT defines slave devices as *Boxes*) must be now visible on the *Solution Explorer* tree within the *Device 1 (EtherCAT)* master device with the last configuration that was written on the ESC's EEPROM as shown in Figure 2-50.

16. In order to write the EEPROM of the ESC with our project's description file (xml) so that it matches with the configuration of the Delfino MCU, double click on *Device 1 (EtherCAT)*, select the *Online* tab of the emerged window, right click on *Box 1* item and choose *EEPROM Update* (Figure 2-51).

17. Select *EtherCAT Laelaps Motion Control v4 (SPI).xml (11110141 /4)* and click *OK* (Figure 3-20).

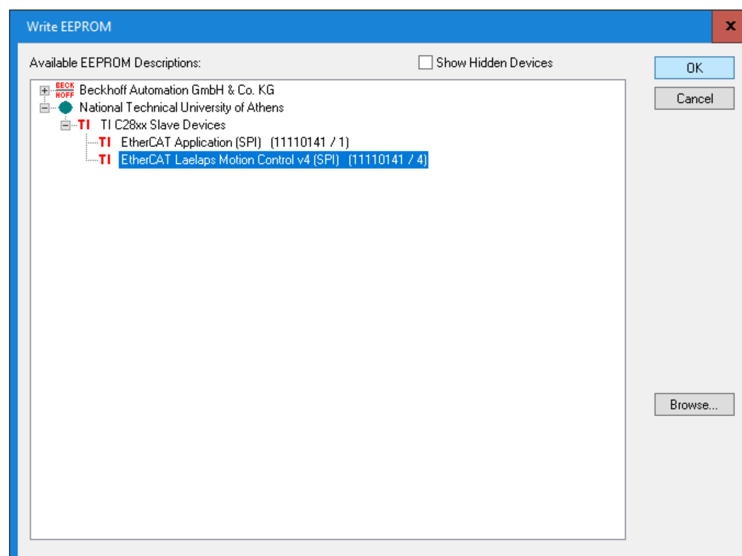


Figure 3-20. XML selection.

18. Now our slave's EEPROM memory is correctly configured, yet TwinCAT cannot automatically recognize this transition. In order to do so, on the *Solution Explorer* window, right click on *Box 1*, select *Remove* and then *OK* (Figure 2-53).

19. *Scan* the network again as indicated above and monitor *Box 1* with the desired configuration and EtherCAT variables (Table 3-1,Table 3-2).

20. Activate *Free Run Mode* by clicking *Toggle Free Run State* button (Figure 2-49) and ensure that the slave device has swiched to *Operation State* by checking the *RUN* LED of the FB1111-0141 ESC (must be ON) and the *Current State* of the *Online tab* (Figure 2-55) which must be *OP* (Operational).

21. Check that the application is properly running by executing an Online Write of Blue_LED and Red_LED and inspect the build in LEDs of the launchpad.

22. In order to force the slave to operate in DC – Sync mode and achieve the ultimate synchronization, follow the instructions of TwinCAT in Run Mode.

23. Follow the instructions of Add Watch Expression in CCS Debug to add the variables of Figure 2-59. If all integer type variables are 1, then the slave is properly functioning in DC mode and the values of the unsigned long type variables (timer_) indicate the required execution time in micro seconds.

24. Now that you have checked the whole project, repeat steps 18 → 26 for every other EtherCAT Tower Assembly of Laelaps, by connecting the IN port, one at a time, ensuring that all slaves solely turn into Operational State (in DC mode). Once you have completed this procedure, all slaves of the network are properly configured and we are now at the final phase of the solution guide.

25. Make sure that all Laelaps' slaves are physically connected to the network, create a new project (*Laelaps Control*) adding an EtherCAT Master device and *Scan* the network. All *Boxes* must me visible on the Solution Explorer window.

26. Force all slaves to operate in DC – Sync one by one as described in TwinCAT in Run Mode. Specify the EtherCAT cycle time at 400 micro seconds.

TwinCAT Scope View Configuration

1. Add TwinCAT Scope View in order to be able to save all EtherCAT variables during the experiments. Right click on *Solution 'Laelaps Control'* > *Add* > *New Project* and select *Scope YT Project* from the *TwinCAT Measurement tab* (Laelaps Control Measurement) as shown in Figure 3-21.

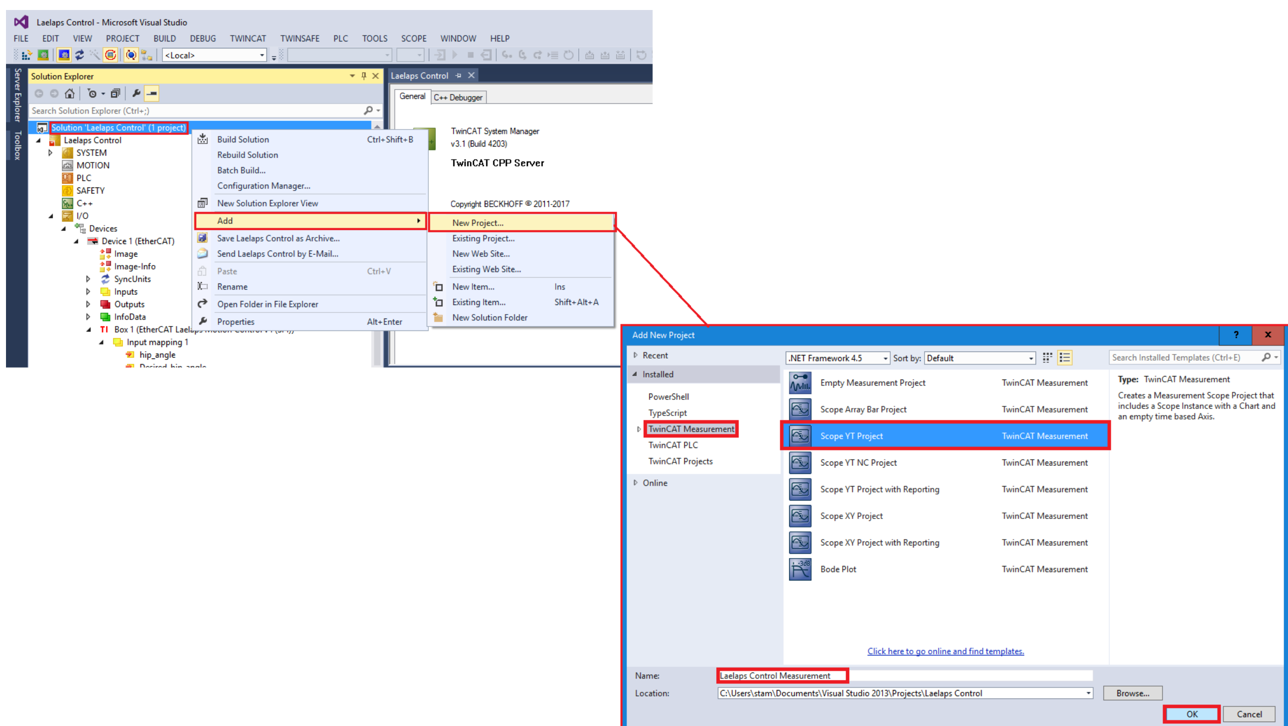


Figure 3-21. Add Scope Measurement.

2. Add a PLC task to be able to use the Scope View. Right click on PLC > Add New Item > Standard PLC Project (LaelapsControl) as indicated in Figure 3-22. If no PLC task is created, TwinCAT will not be able to plot and save the values of the desired EtherCAT variables.

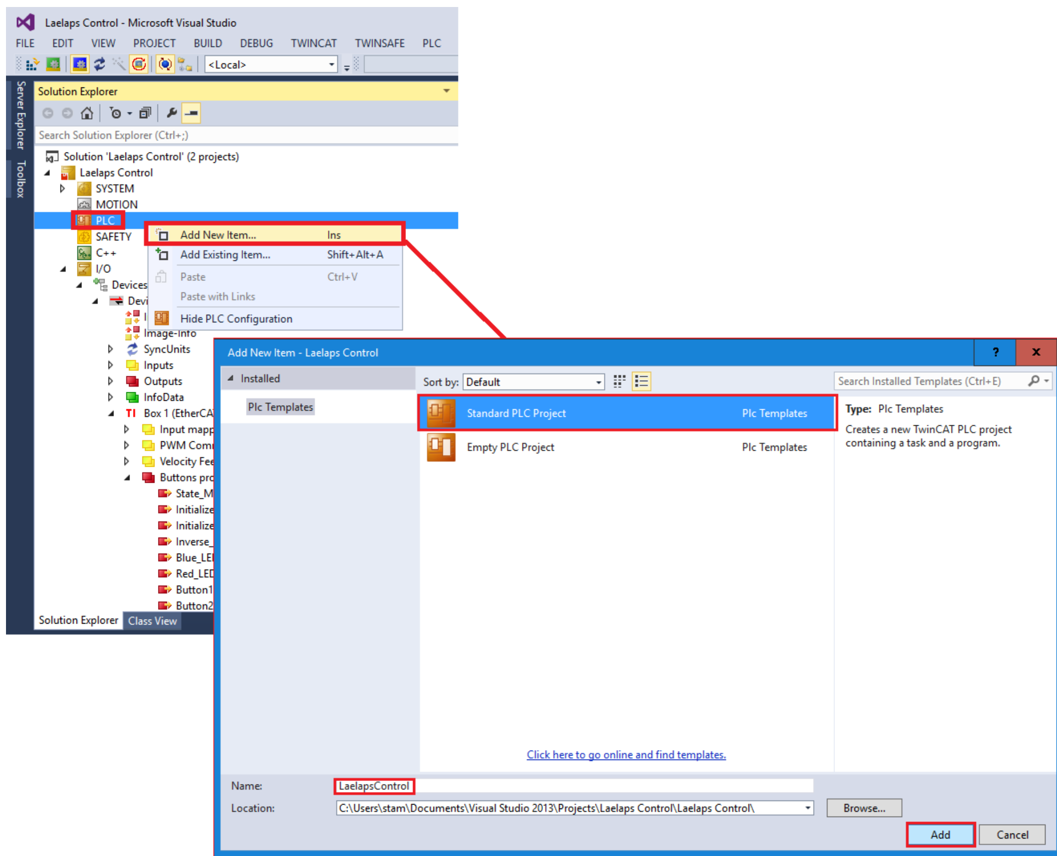


Figure 3-22. Add PLC Task.

3. Create a Global Variable list of all input variables that you want to save during the experiment. Right Click on EtherCAT Laelaps Project > Add > Global Variable List, name it (Inputs) and click Open as shown in Figure 3-23.

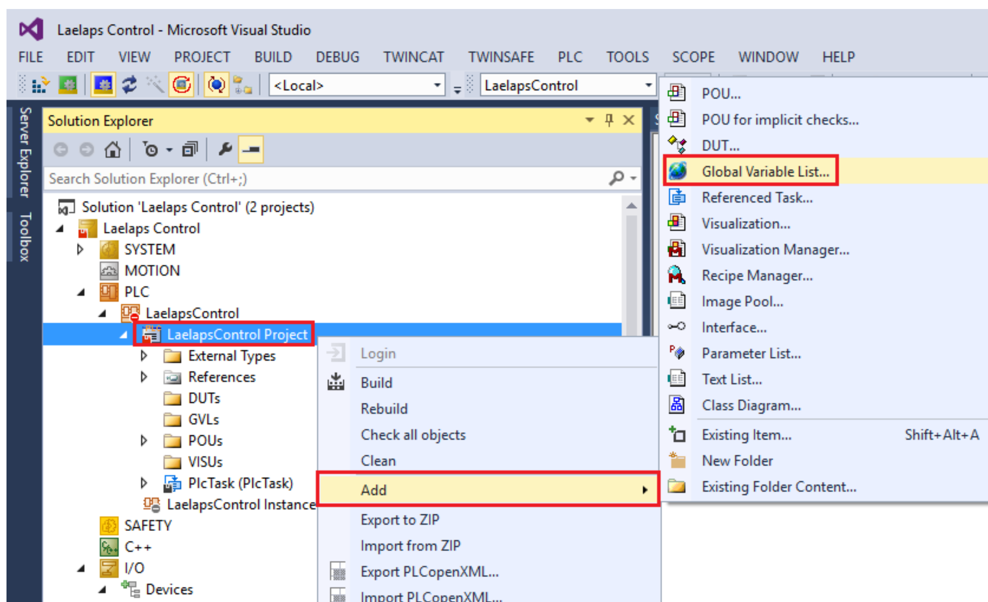


Figure 3-23. Add Global Variable List.

4. Make sure that you add all necessary variables by following the format shown in Figure 3-24 where all required inputs to the master are being scoped and saved making sure that they have the right variable type.

```
{attribute 'qualified_only'}
VAR_GLOBAL
    HR_hip_angle AT%I*: INT;
    HR_knee_angle AT%I*: INT;
    HR_Desired_hip_angle AT%I*: INT;
    HR_Desired_knee_angle AT%I*: INT;
    HR_PWM_hip AT%I*: INT;
    HR_PWM_knee AT%I*: INT;
    HR_velocity_knee AT%I*: DINT;
    HR_velocity_hip AT%I*: DINT;
    HR_time AT%I*: UINT;
    HL_hip_angle AT%I*: INT;
    HL_knee_angle AT%I*: INT;
    HL_Desired_hip_angle AT%I*: INT;
    HL_Desired_knee_angle AT%I*: INT;
    HL_PWM_hip AT%I*: INT;
    HL_PWM_knee AT%I*: INT;
    HL_velocity_knee AT%I*: DINT;
    HL_velocity_hip AT%I*: DINT;
    HL_time AT%I*: UINT;
    FL_hip_angle AT%I*: INT;
    FL_knee_angle AT%I*: INT;
    FL_Desired_hip_angle AT%I*: INT;
    FL_Desired_knee_angle AT%I*: INT;
    FL_PWM_hip AT%I*: INT;
    FL_PWM_knee AT%I*: INT;
    FL_velocity_knee AT%I*: DINT;
    FL_velocity_hip AT%I*: DINT;
    FL_time AT%I*: UINT;
    FR_hip_angle AT%I*: INT;
    FR_knee_angle AT%I*: INT;
    FR_Desired_hip_angle AT%I*: INT;
    FR_Desired_knee_angle AT%I*: INT;
    FR_PWM_hip AT%I*: INT;
    FR_PWM_knee AT%I*: INT;
    FR_velocity_knee AT%I*: DINT;
    FR_velocity_hip AT%I*: DINT;
    FR_time AT%I*: UINT;
END_VAR
```

Figure 3-24. Global Variable List.

5. In the POUS > MAIN (PRG), create a list of all variables that you want to handle simultaneously in all slaves. This process MUST be done at least for the State_Machine variable, so that the clocks in all four slaves are initiated at the exact same time. Moreover, due to the fact that all four slaves have identical configuration, the output list of Figure 3-25 was created containing all shown variables.

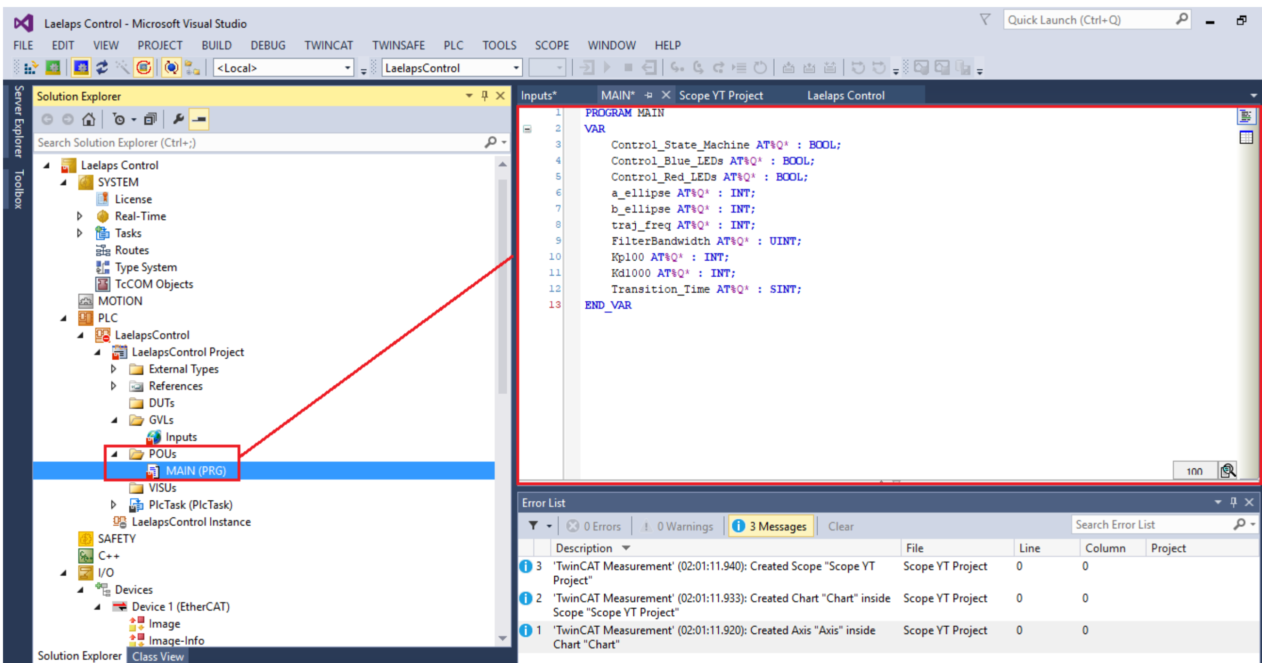


Figure 3-25. MAIN (PRG) list.

6. *Build* the solution and expand the LaelapsControl Instance to inspect all PLC variables.

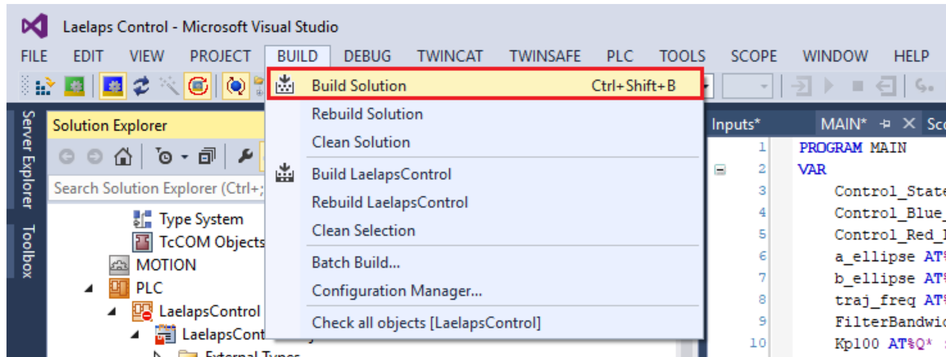


Figure 3-26. Build Solution.

7. Link all Input and Output variables of the list to the analogous EtherCAT variables by clicking twice on each PLC variable, selecting Linked to and choosing the desired from the list of all compatible variables (as far as type is concerned) as shown in Figure 3-27. Note that in order to link one output PLC variable to multiple EtherCAT output variables, select all desired EtherCAT variables from the pop up window holding Ctrl button. Now, all linked output variables of the project are handled by the *PlcTask Outputs* and *Online Writes* can only be executed through this list. Do not forget to link the State Machine Plc output variable with ALL State_Machine EtherCAT variables of all slaves to accomplish a synchronous initiation of the trajectories' time variables.

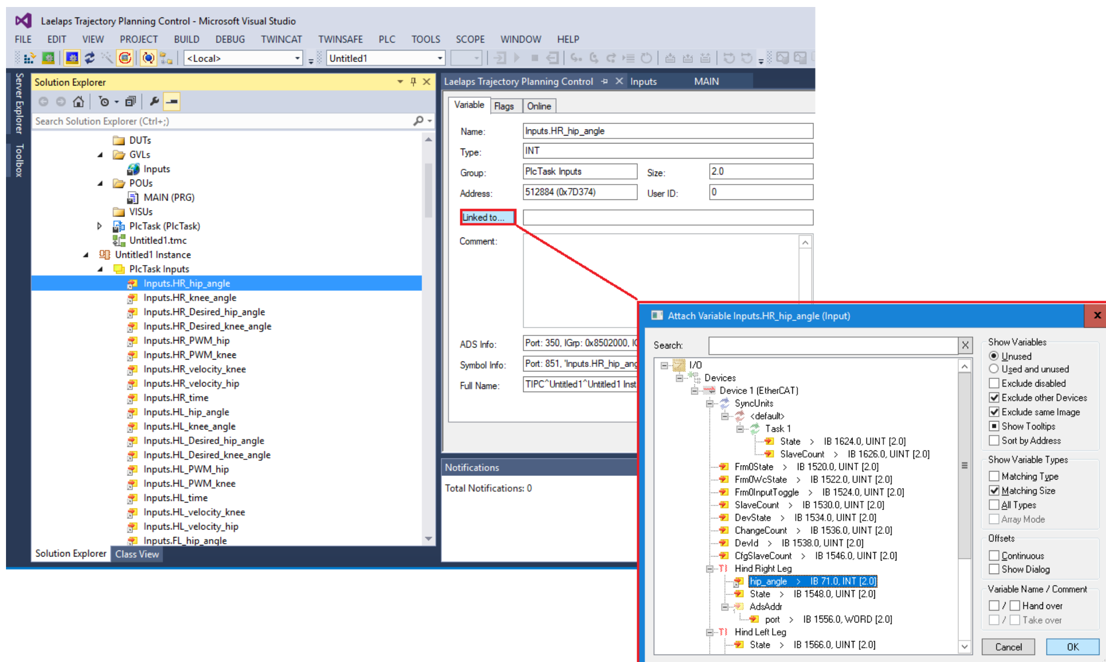


Figure 3-27. Linking PLC variables.

8. *Activate Configuration* and *Restart TwinCAT System* to update the project with the linked variables.

9. Login and Start the PLC task as indicated in Figure 3-28

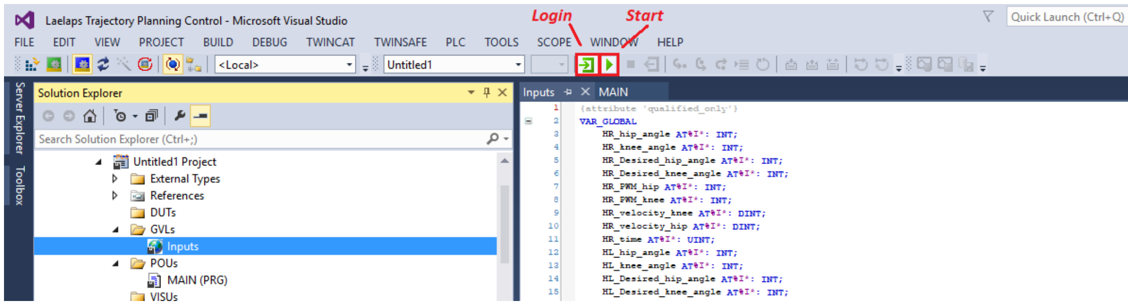


Figure 3-28. Start PLC task.

10. Navigate to Laelaps Control Measurement of the Solution Explorer and right click on *Axis* > *Target Browser* and select all desired variables from the *Global Variable List* (*Inputs*) that you want to monitor and save during the experiment (Figure 3-29).

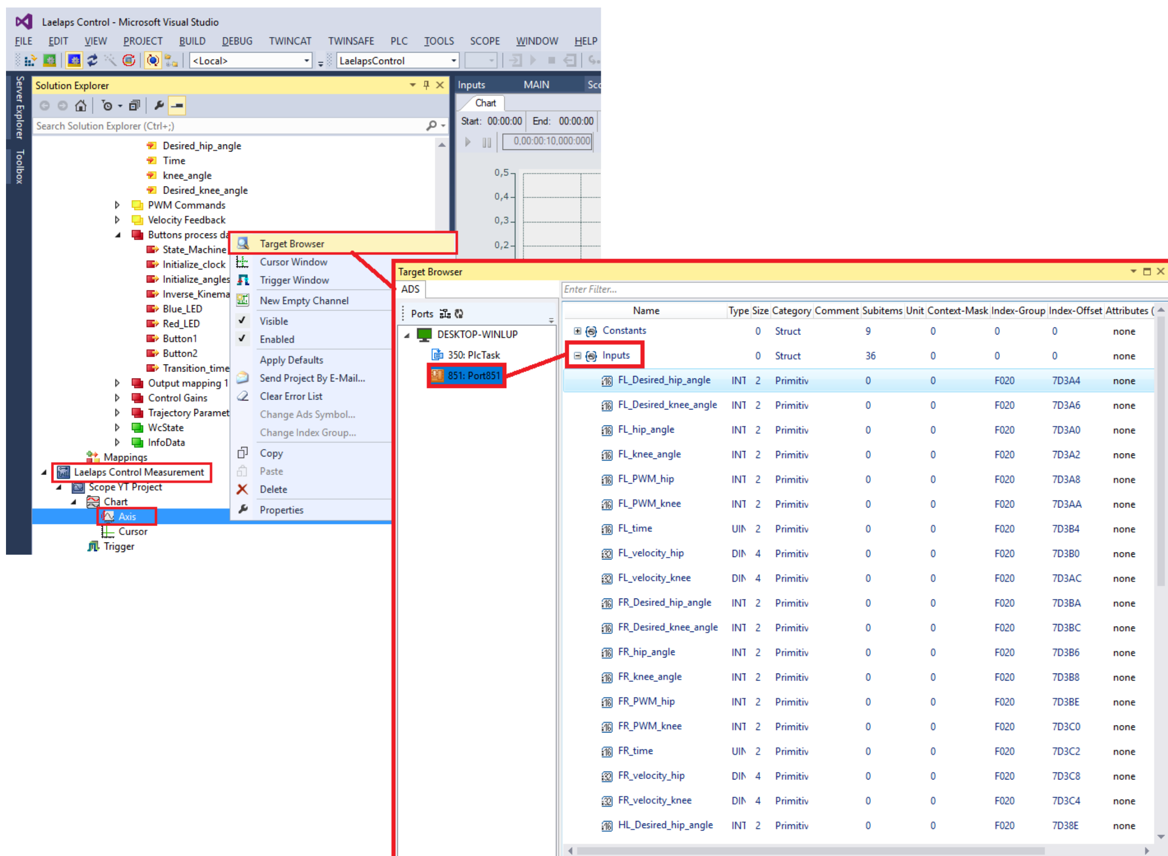


Figure 3-29. Adding variables to the Scope View.

11. To start and stop recording, use the *Record* and *Stop Record* buttons shown in Figure 3-30.

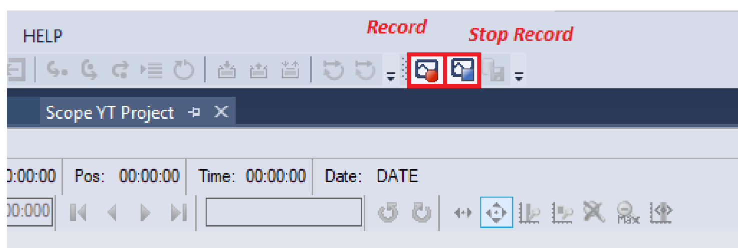


Figure 3-30. TwinCAT Scope Record.

12. After the completion of a recording, in order to save it in csv format and post process it in Matlab, click *Scope* > *Export to CSV* (Figure 3-31).

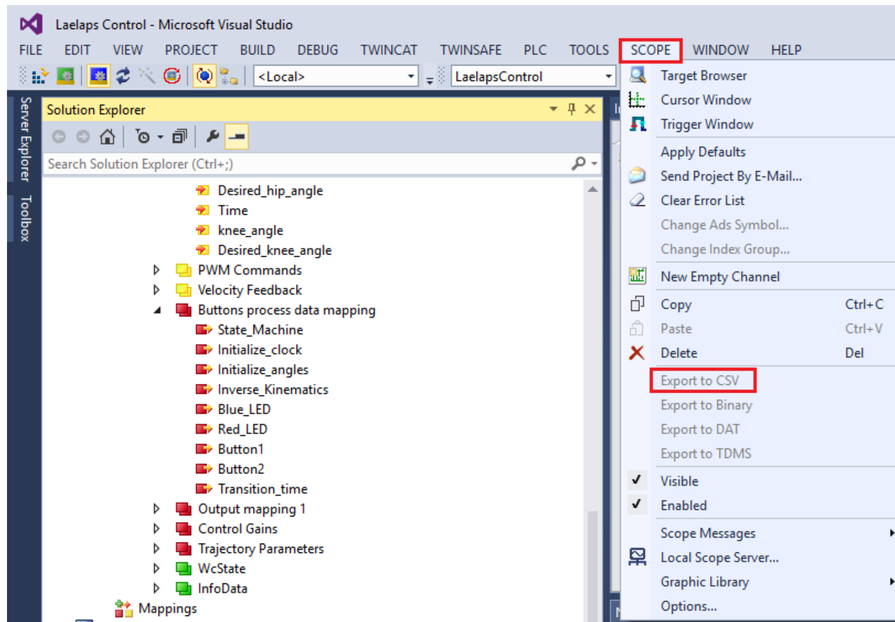


Figure 3-31. Save and Export Recording.

Leg Initialization and Execution

13. Initialize all legs by manually placing them in the position depicted in Figure 3-3 and press the Reset button (shown in Figure 3-32) of every Delfino Launchpad.

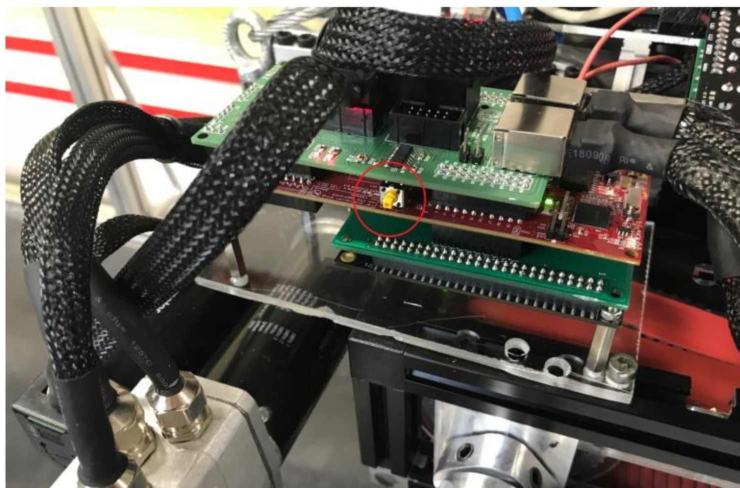


Figure 3-32. Reset button to initialize legs poise.

Laelaps is now completely ready to perform any kind of experiment and test all its features. The final step would be to check that all wires, drivers and extension boards are properly mounted on the quadruped robot and that the State Machine PLC variable (connected to all slaves' State_Machine EtherCAT variables) is set to Configurational State (0) before enabling the High Voltage Power Supply. Figure 3-33 illustrates the experimental setup of Laelaps II on the treadmill, ready to perform the desired task.

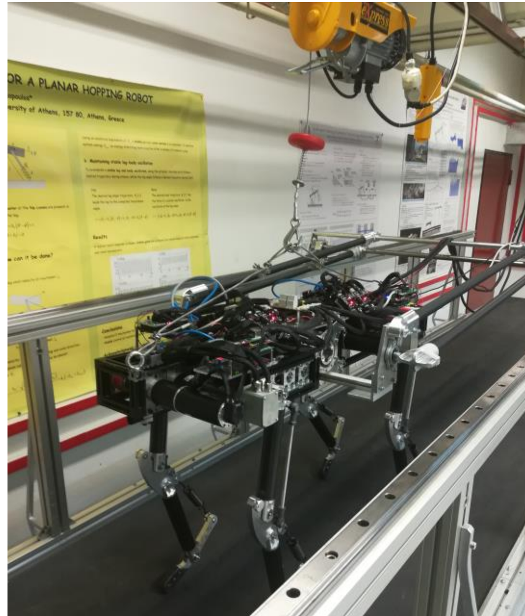


Figure 3-33. Laelaps II on treadmill ready to perform experiments.

The State Machine diagram of Laelaps is illustrated in Figure 3-34. Entering suitable parameters to all EtherCAT output variables and switching into Operational State (1) will force Laelaps to execute the desired movement.

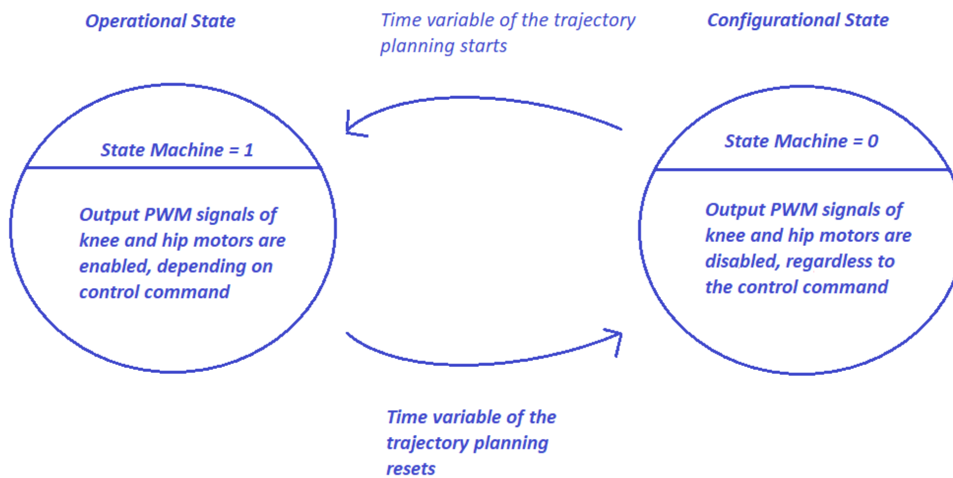


Figure 3-34. Laelaps' State Machine.

The following chapter includes several tables of suitable Control and Trajectory parameters along with all the necessary electrical and communication data. Users are encouraged to use parameter values close to the values given therein.

4 Laelaps II Locomotion Experiments

This chapter includes results from trotting experiments with Laelaps II in a low frequency with the developed control scheme. Numerous experiments were carried out successfully but this chapter only presents an indicative ones. Employing the firmware described in Motion Control of Laelaps via EtherCAT Solution Guide and running TwinCAT in a PC as the EtherCAT master, successfully led to Laelaps II first steps. Although there are several kinds of quadruped movements, the first series of experiments focused on *trotting*, one of the simplest symmetrical gaits.

In this first series of experiments, a desired elliptical trajectory is defined for the toe of each leg through EtherCAT (Twincat Runtime) along with the control gains and parameters of the system. Data are logged using TwinCAT Scope View and plotted using a custom made Matlab script [35] (also refer to Matlab Post Process Code). As indicated above, the PIV (Proportional – Integral – Velocity) controller is implemented in each slave, thus the master does not affect the control process but simply supplies each slave with the necessary parameters via EtherCAT.

For each experiment a table thoroughly describing the parameters used is given and six, in total, figures are presented, illustrating:

- The 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.
- The desired response of both knee and hip angles (red) of evry leg with their respective actual response of each knee and hip joint (black).
- The PWM commands of each leg's knee and hip motor (black) which is the output of the PIV controllers with their respective predefined PWM limits (red). As mentioned above, these values represent the continuous current limits of both motors. Refer to PIV Motor Controller for more information regarding the selected limits.
- The velocity estimation of each leg's knee and hip joint (black) and the respective predefined motor speed limits (red).

4.1 Trotting Experiment 1

In this experiment [51] Laelaps is initially in a standing position with all four legs configured with the parameters shown in Table 4-1, except for the trajectory parameters, *a ellipse* and *b ellipse*; their values are set to 0 at the beginning, therefore the elliptical trajectory is just a point. After the recording begins, *b ellipse* parameter – which corresponds to the clearance from the ground – is increased to 4 cm linearly with time (depending on the *Transition Time* variable which was set to three seconds throughout the experiment) to all slaves simultaneously, and similarly *a ellipse* 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 ellipse* and then *b ellipse*), Laelaps decelerates and eventually stops walking and remains still. The recording is terminated and all data are saved and post processed in Matlab.

Table 4-1. Trotting Experiment 1

Parameters		FL Leg	FR Leg	HL Leg	HR Leg
Trajectory Paramaters	x centre	0 cm	0 cm	0 cm	0 cm
	y centre	59.9 cm	59.9 cm	59.8 cm	59.8 cm
	a ellipse	5 cm	5 cm	5 cm	5 cm
	b ellipse	4 cm	4 cm	4 cm	4 cm
	Frequency	1 Hz	1 Hz	1 Hz	1 Hz
	Phase [deg]	180	0	0	180
	Flatness	0	0	0	0
Control Gains of Knee	Kp_knee	80	80	80	80
	Kd_knee	0.05	0.05	0.05	0.05
	Ki_knee	0	0	0	0
Control Gains of Hip	Kp_hip	80	80	80	80
	Kd_hip	0.05	0.05	0.05	0.05
	Ki_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%
Control Loop Frequency		10 kHz	10 kHz	10 kHz	10 kHz
Filter Bandwidth Frequency		20 Hz	20 Hz	20 Hz	20 Hz
Loop Frequency of EtherCAT		2.5 kHz			
Voltage Supply (System)		40.34 V			
Max Value of Current (System)		50.11 A			

During the steady state phase of the experiment, where both the *a ellipse* and *b ellipse* 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 path –trajectory of each leg's 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 (0 -Figure 3-4), are shown in Figure 4-1. 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 orbits are not closely tracked in the permanent state and a better regulation of these gains is required, especially for the hind legs.

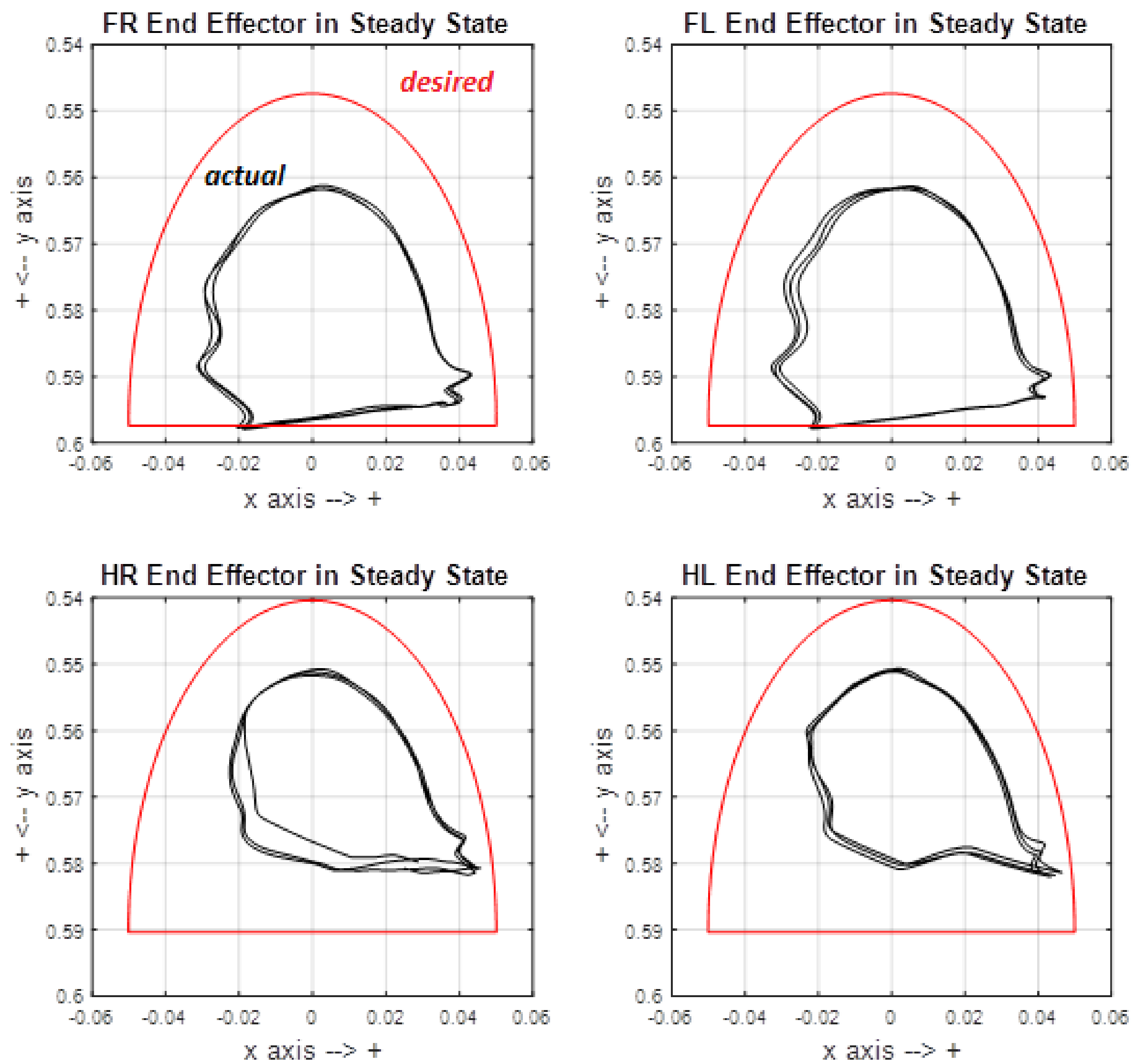


Figure 4-1. Desired elliptical trajectory of all legs toe (red) along with their actual response (black) w.r.t coordinate systems located in the hip joints of the legs.

Figure 4-2 displays the desired value of each leg's *knee* joint angle (red) and the actual – real response of every respective knee joint (black) throughout the experiment. Both the transition and the steady state phase are illustrated. The unit measurement of all values is degrees and as one may observe in these figures, 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.

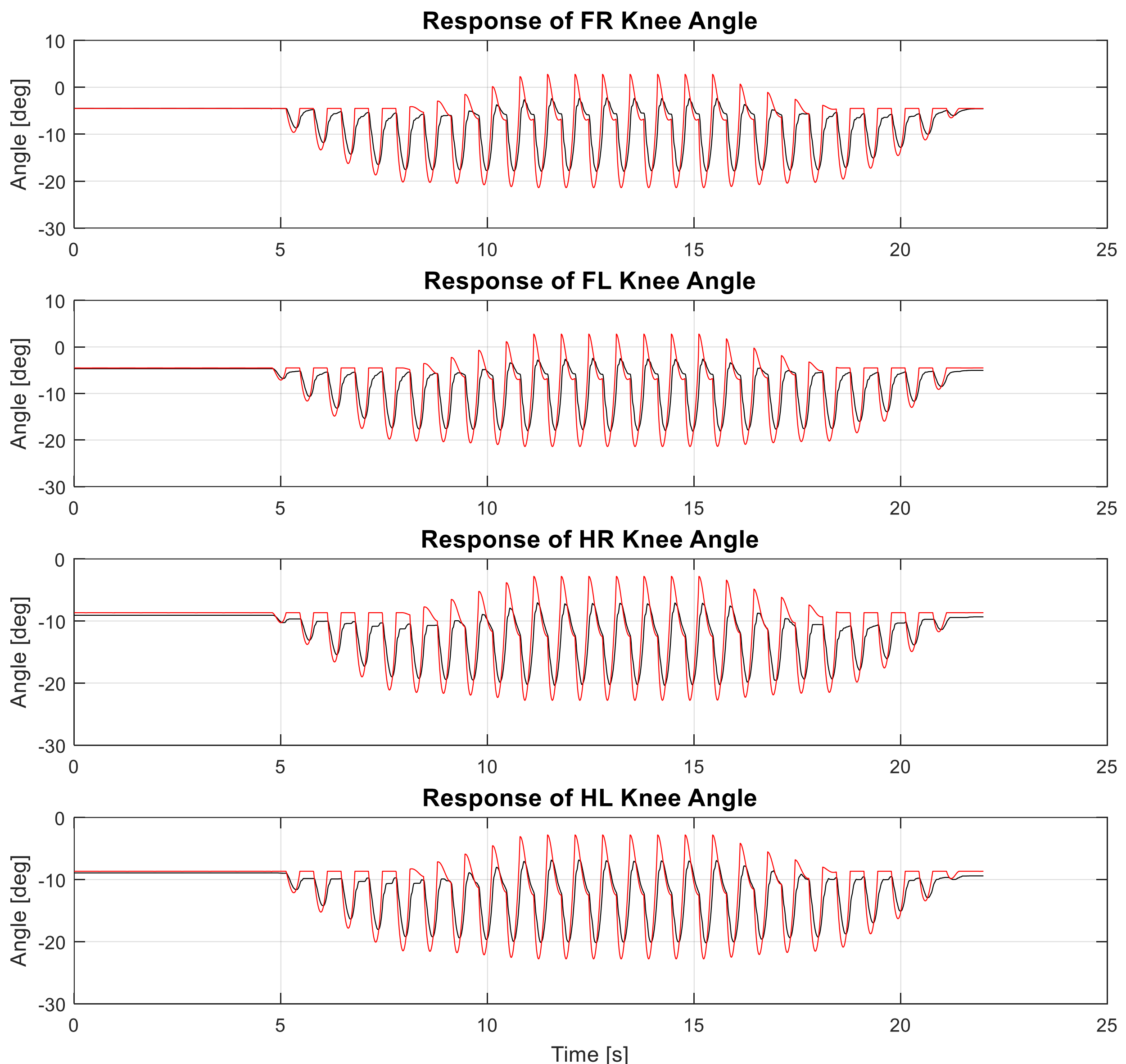


Figure 4-2. Desired response of knee angles (red) and actual response of knee joint (black).

On the other hand, Figure 4-3 describes the desired value of each leg's *hip* joint angle (red) and the actual – real response of every respective hip joint (black) throughout the experiment. Both the transition and the steady state phase are illustrated. The unit measurement of all values is degrees and as one may observe in these figures, 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 a 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, we should take into consideration that the hip joint performs a wider movement which is another reason why the resulting erros are larger compared to the knee joints.

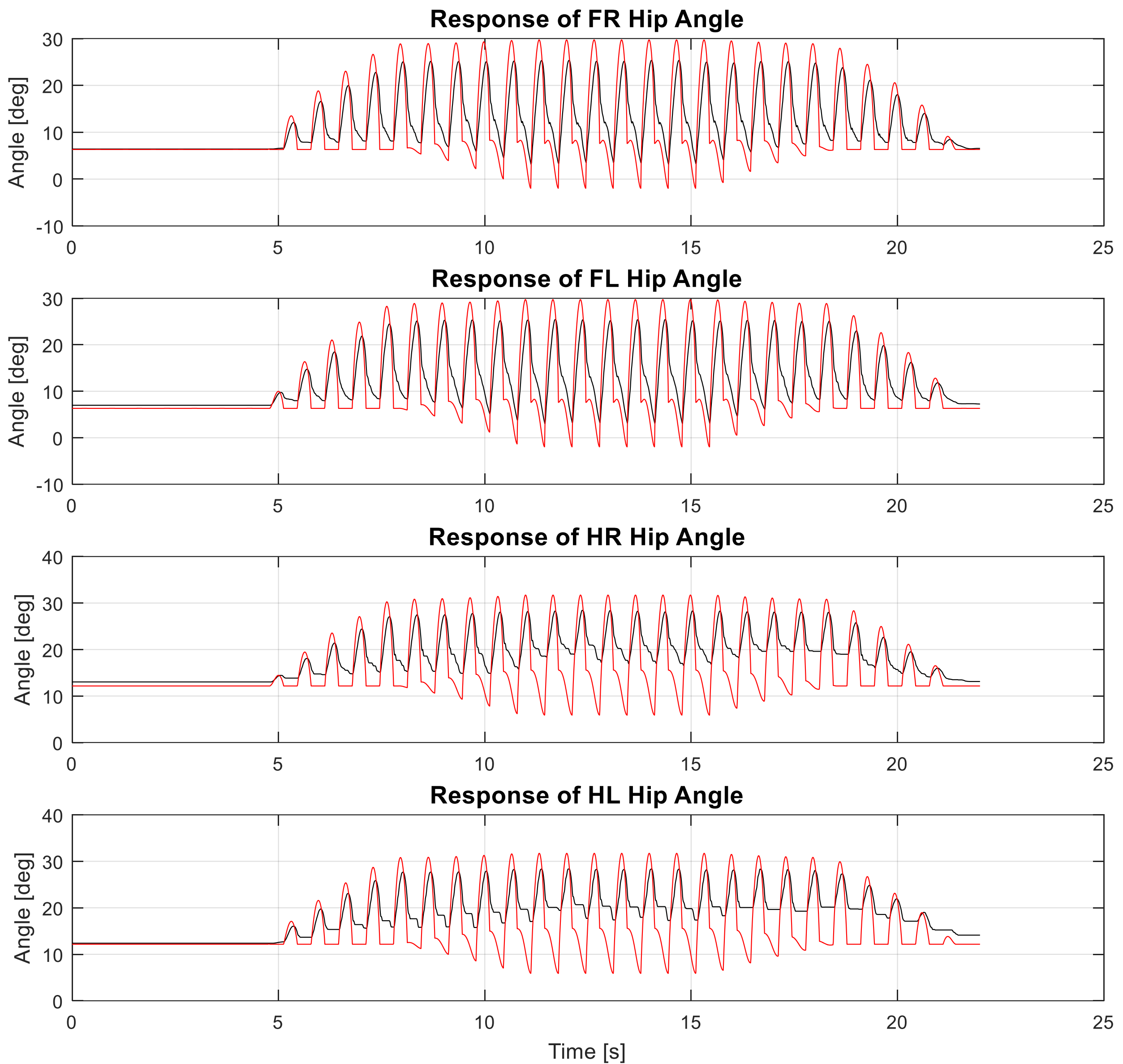


Figure 4-3. Desired response of hip angles (red) and actual response of hip joint (black).

Figure 4-4 depicts the PWM commands [%] of each leg's *knee* motor (black) with its respective predefined limit (red). These commands are the output of the knee's PIV controller exploited in our application (PV actually because the Integral Proportional gain is 0) and are directly translated in torque commands since a current control architecture is implemented. As one may observe, the commands in both hind legs are always within the limit range, hence there is no reason in modifying them. Accordingly, in the two fore legs, although the limits are reached several times, due to the fact that it hapened only for short intervals, no extra action is needed.

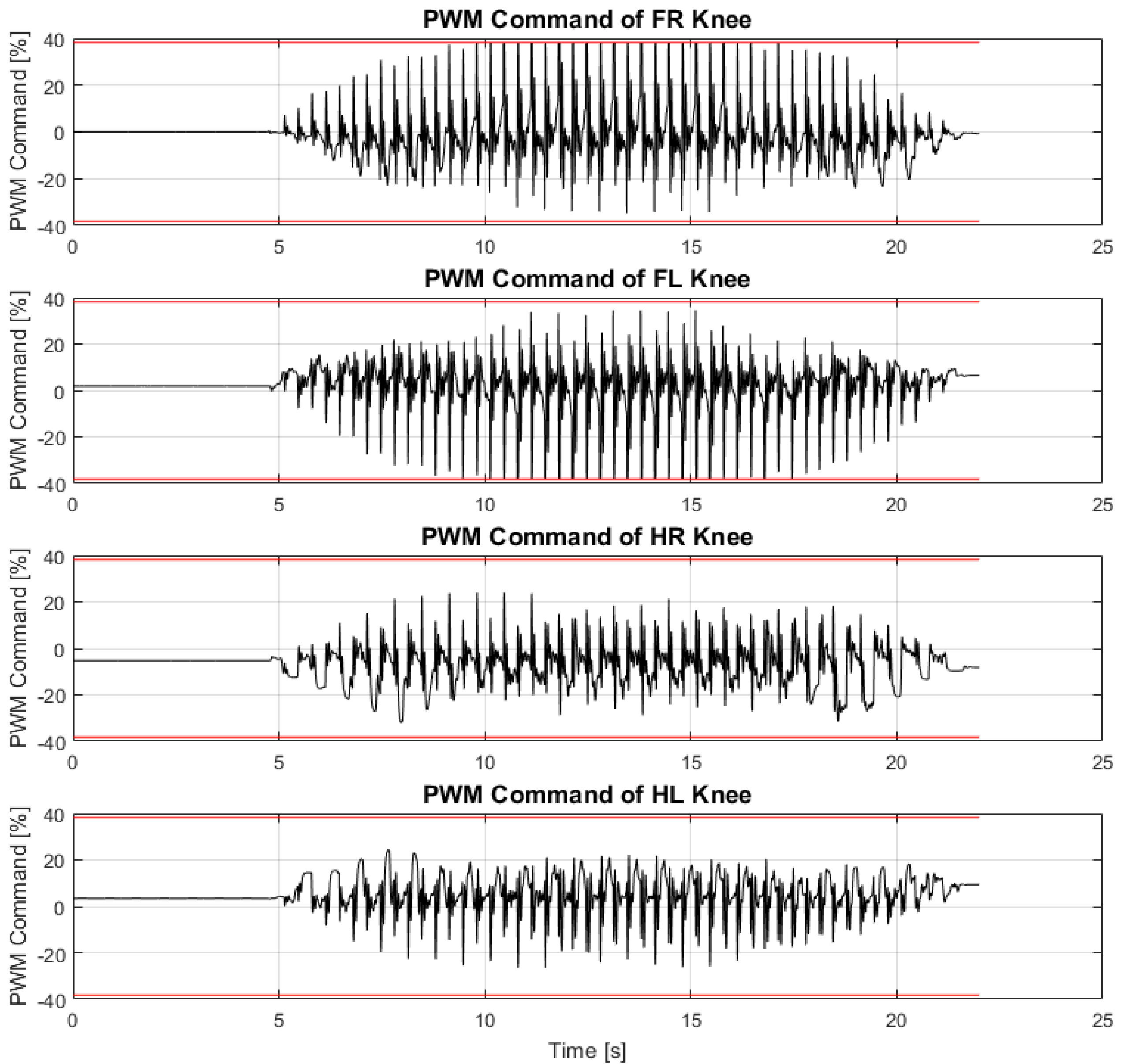


Figure 4-4. PWM commands of each leg's knee motor (black) and the respective predefined PWM limits (red).

Similarly, Figure 4-5 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, this time, exploited in our application (PV actually because the Integral Proportional gain is 0) and are directly translated in torque commands since a current control architecture is implemented. As one may observe, hip PWM limits are recurrently reached, especially in the hind legs, thus an increase of the allowed range should be considered.

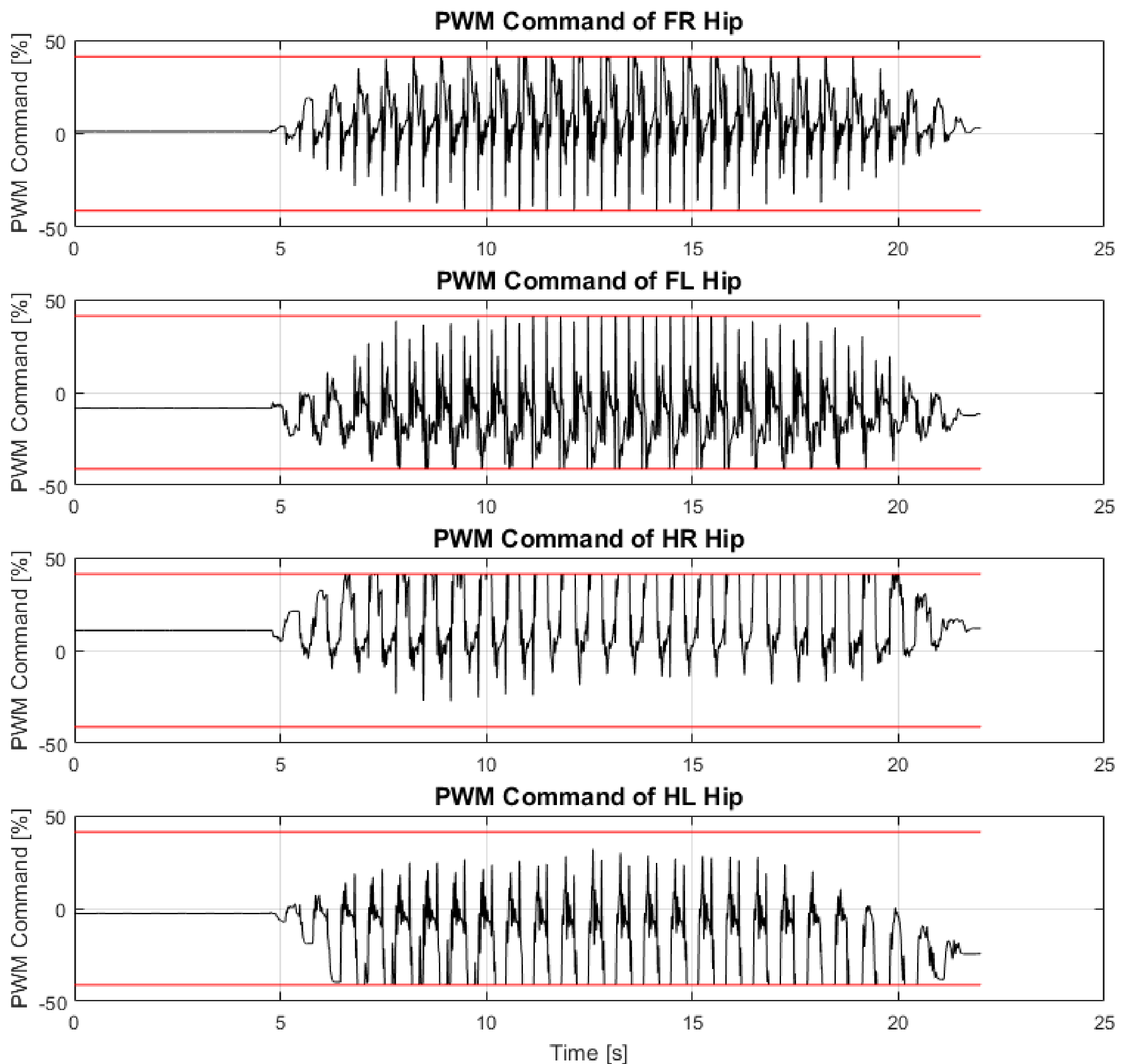


Figure 4-5. PWM commands of each leg's hip motor (black) and the respective predefined PWM limits (red).

Figure 4-6 adumbrates the velocity estimation of each legs knee joint using the modified eQEP peripheral as described in Rotational Speed (black) and the respective motor speed limits (red) as specified by the manufacturer (refer to Laelaps II motors and gearheads). As anyone can observe from the following figure, the velocities of every knee motor are always within the allowed range. Therefore, there is no concern regarding the velocities scheme that would stimulate a reduce in knee PWM limits.

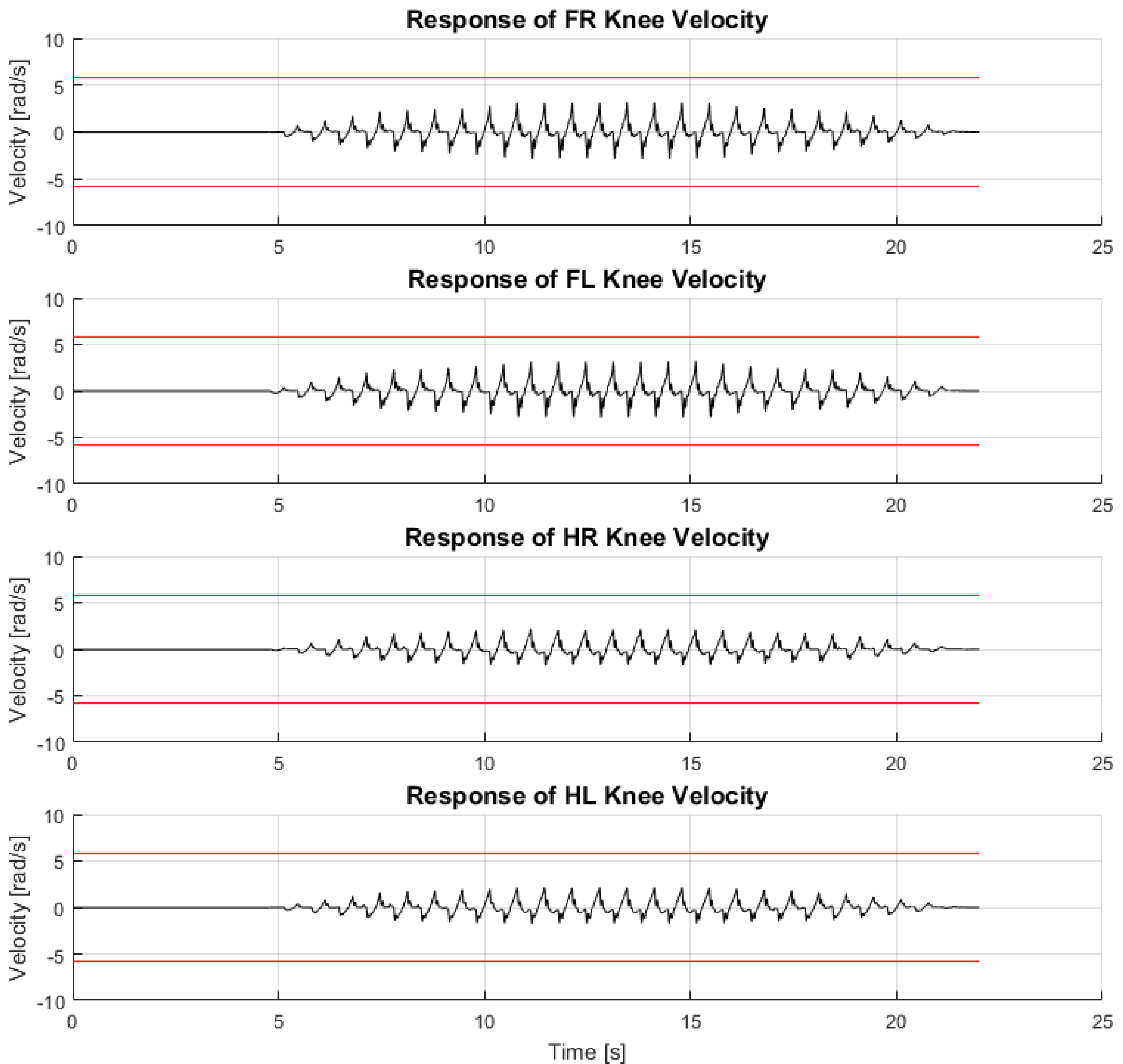


Figure 4-6. Velocity estimation of each leg's knee joint (black) and the respective predefined motor speed limits (red).

Analogously, Figure 4-7 illustrates the velocity estimation of each legs hip joint using the modified eQEP peripheral as described in Rotational Speed (black) and the respective motor speed limits (red) as specified by the manufacturer (refer to Laelaps II motors and gearheads). Once again, as anyone can understand, the velocities of every hip motor are always within the allowed range, thus there is no need to consider reducing hip PWM limits.

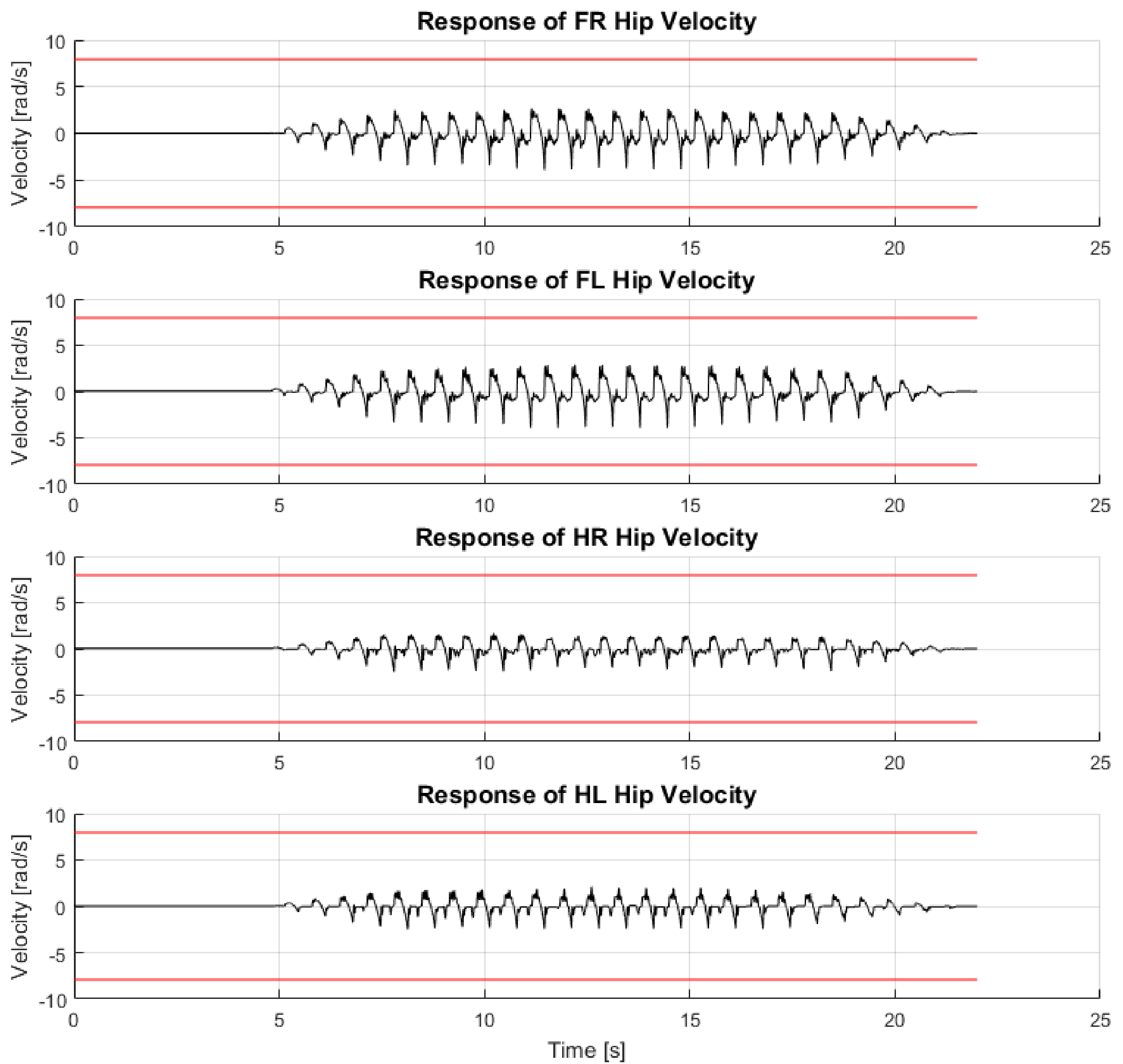


Figure 4-7. Velocity estimation of each leg's hip joint (black) and the respective predefined motor speed limits (red).

5 Conclusion and Future Work

5.1 Conclusion

EtherCAT communication protocol proved highly efficient and useful throughout the experimental validation, even without a dedicated real time EtherCAT Master node but with the TwinCAT XAE software in a Windows Operating System instead. Depending on the data payload which is intimately connected with 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). The total purchasing cost of all the required components for the new control architecture is almost 10% of the previous version enabling the procurance of several spare parts. As the promising results showed, the new decentralized architecture will certainly enable Laelaps II to perfom higher frequency motions and reach its maximum velocity using the current firmware, with only minor upgrades on its mechanical system.

Moreover, using the reference guide of EtherCAT Application Guide, developers may implement EtherCAT communication for other platforms and applications at CSL–EP laboratory. The initial and ultimate purpose of switching towards EtherCAT was to build a low-cost, powerful module that could be mounted to several projects at CSL–EP lab for motion control. The experimental validation of Laelaps II illustrated that the developed EtherCAT Control Tower Assembly (Figure 3-8) is totally functional and can play the aforementioned role for several robots and autonomous devices to come. Its portable design is both easy to assemble and mount, while the firmware structure is easy to comprehend and modify to cater to any needs that emerge.

In a nutshell, I personally believe that switching towards EtherCAT technology was undoubtedly a judicious and wise choice due to its several alleviating functionalities, especially in the motion control area. Its synchronization capabilities along with its portable and extensible architecture are ultimately soothing and flexible to cater to any application. Although it is a bit tricky to gain an overall grasp of the communication protocol and requires considerable time to confidently make custom alterations without causing erros, I sincerely reckon that it is unquestionably the most suitable layer to implement a decentralized motion control theory on robotic applications. EtherCAT's outstanding performance and efficiency have proven highly beneficial to our experimental validation and I am sanguine that it will remain this way for future projects to come at the CSL-EP laboratory of NTUA.

5.2 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 in both software and hardware level, several aspects can be improved in the future to achieve greater robustness.

First of all, a key reason for selecting dual core MCUs (Figure 2-25) was to exploit each core in a different task to efficiently distribute the overall computational effort. In the current application, the firmware (Control & EtherCAT) runs on the first CPU (Central Programming Unit) of the platform, while the second one is completely idle. In the future, this second CPU can be used either to execute one of the basic applications of

the project (EtherCAT communication or Motor Control), or just the ISR (Interrupt Service Routine) of the second motor to reduce the payload on the first core (or vice versa).

Another important feature, which is already in progess, is the replacement of the TwinCAT XAE software (running on a Windows Operating System through Visual Studio) as the EtherCAT Master, with a dedicated embedded real time computer running Linux-ROS.

Moreover, a useful feature would be to automatically initialize the knee and hip angles of the legs and skip the current manual process. This functionality could be implemented by enabling the already assembled ADC code (initialization and execution of the peripheral) within the firmware, and after mounting the already selected and purchased RLS absolute encoders to the mechanism.

Finally, an important task would be to design a case for the EtherCAT Control Tower Assembly not only for protecting it, but also for greater stability, support, robustness and portability of the module so that it can be safely and easily exploited in other robots too.

References

[1] N. Liu, Z. Liu, T. Zhang, L. Cui and H. Li, "EtherCAT Based Robot Modular Joint Controller", *Proceeding of the 2015 International Conference on Information and Automation (IEEE '15)*, Lijiang, China, August 2015.

[2] E. Papadopoulos and J. Poulakakis, "Planning and Model-Based Control for Mobile Manipulators," *Proceedings of the 2000 International Conference on Intelligent Robots and Systems (IROS '00)*, Takamatsu, Japan, October 30 - November 5 2000, pp. 245-250.

[3] B. Siciliano, L. Sciavicco, L. Villani and G. Oriolo, "Robotics: Modelling, Planning and Control", Springer, 2009.

[4] R. Zurawski, "Industrial Communication Technology Handbook, Second Edition", *CRC Press,* September 19, 2017.

[5] G. Bolanakis, "Design and Implementation of a Quadruped Robot Electronic System", Athens, Greece, 2018.

[6] Beckhoff, New Automation Technology, "Application Note ET9300 (EtherCAT Slave Stack Code)".

[7] Beckhoff, New Automation Technology, "EtherCAT Synchronization in TwinCAT".

[8] EtherCAT Technology Group (ETG), "How to set up a Network Configuration".

[9] Texas Instruments, "TMDSECATCNCD379D EtherCAT Solution Reference Guide", September 2017.

[10] Texas Instruments, "EtherCAT® Interface for High-Performance C2000™ MCU", August 2017.

[11] Texas Instruments, "C2000 Digital Control Library", November 2017.

[12] Texas Instruments, "TMS320x2833x, 2823x Enhanced Quadrature Encoder Pulse (eQEP) Module", December 2008.

[13] http://www.iebmedia.com/index.php?id=5794&parentid=63&themeid=255&showdetail=true

[14] http://www.electronicdesign.com/embedded/industrial-automation-relies-ethernet

[15] https://www.icpdas-usa.com/ecat

[16] http://www.processindustryforum.com/article/fieldbus-vs-ethernet

[17] https://www.bostondynamics.com/

[18] http://www.rsl.ethz.ch/robots-media/anymal.html

[19] http://biomimetics.mit.edu/

[20] https://kodlab.seas.upenn.edu/

[21] https://www.kuka.com/

[22] http://www.nexcom.com/

[23] https://www.shadowrobot.com/

[24] https://pal-robotics.com

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

[26] https://www.ethercat.org

[27] https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxrZW1hY2hhaXJhc3xneDo2NzA5YTRiMDMzN2Q1MTQw

[28] https://infosys.beckhoff.com/english.php?content=../content/1033/tcsystemmanager/reference/ethercat/html/ethercat_supnetworkcontroller.htm&id

[29] https://bitbucket.org/csl_legged/delfino-projects-ethercat/src/master/EtherCAT%20Application/
[30] http://nereus.mech.ntua.gr/legged/?page_id=161
[31] https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxrZW1hY2hhaXJhc3xneDo1MDE5NjVkMDg0MTdiNjRm
[32] http://nereus.mech.ntua.gr/laelaps/
[33] http://dspace.lib.ntua.gr/handle/123456789/44986
[34] http://nereus.mech.ntua.gr/laelaps-wiki/index.php/Legged-topics
[35] https://bitbucket.org/csl_legged/delfino-projects-ethercat/src/master/Matlab/
[36] https://grobotronics.com/dc-dc-step-down-5v-2a.html
[37] https://bitbucket.org/csl_legged/delfino-projects-ethercat/src/master/EtherCAT%20Laelaps%20Motion%20Control/
[38] https://www.maxonmotor.com
[39] https://www.maxonmotor.com/medias/sys_master/root/8825424609310/17-EN-221.pdf
[40] https://www.maxonmotor.com/medias/sys_master/root/8825548144670/17-EN-350-351.pdf
[41] https://www.maxonmotor.com/medias/sys_master/root/8825409470494/17-EN-133.pdf
[42] http://processors.wiki.ti.com/index.php/Download_CCS
[43] http://www.ti.com/tool/CONTROLSUITE
[44] http://www.ti.com/tool/C2000WARE
[45] https://www.ethercat.org/en/products/54FA3235E29643BC805BDD807DF199DE.htm
[46] http://www.beckhoff.com
[47] https://www.kunbus.com/fieldbus-basics.html
[48] https://www.pc-control.net/pdf/012014/interview/pcc_0114_industrial-ethernet_e.pdf
[49] https://www.automationworld.com/article/technologies/networking-connectivity/ethernet-tcp-ip/fieldbus-industrial-ethernet
[50] https://bitbucket.org/csl_legged/delfino-projects-ethercat/src/master/EtherCat%20SLave%20PCB/
[51] https://www.youtube.com/watch?v=If9bs2z-UYw

6 Appendix A

6.1 Download and Install Code Composer Studio, C2000ware & ControlSuite

Visit the following website [42] and download the latest version of Code Composer Studio for your OS (Figure 6-1). During the installation process, when you are asked about which device descriptions you want to install, make sure that you add the C2000 series device descriptions because you will not be able to do so after the installation is completed and you would have to uninstall and reinstall CCS again.

Download the latest CCS

Download 7.3.0.00019	Installers (Offline installer is recommended for slow and unreliable connections)
Windows	Offline Installer Online Installer
Mac OS	Offline Installer Online Installer
Linux 64bit	Offline Installer Online Installer

Figure 6-1. Code Composer Studio Installation.

In addition, visit TI's website using the following link [43] and download the latest version (Figure 6-2) of Control Suite (which includes several useful applications for different microcontrollers, enabling a variety of features and controlling of peripherals).

Part Number	Buy from Texas Instruments or Third Party	Alert Me	Status	Current Version	Version Date
CONTROLSUITE-ZIP: Offline (ZIP) Installer	Free Download	Alert Me	ACTIVE	v3.4.7	29-SEP-2017
CONTROLSUITE: Web (EXE) Installer	Free Download	Alert Me	ACTIVE	v3.4.7	29-SEP-2017

Figure 6-2. Control Suite Installation.

Finally, visit TI's website using the following link [44] and download the latest version of C2000ware (Figure 6-3) which includes several useful applications for C2000 architecture microcontrollers, enabling a variety of features and controlling of peripherals for different development launchpads.

TI Home > Semiconductors > Microcontrollers (MCU) > C2000Ware for C2000 MCUs

Worldwide (In English)

C2000Ware for C2000 MCUs

(ACTIVE) C2000WARE

Description & Features | Technical Documents | Support & Training | Order Now

Order Now

Part Number	Buy from Texas Instruments or Third Party	Alert Me	Status	Current Version	Version Date
C2000WARE: C2000Ware for C2000 Microcontrollers	Get Software	Alert Me	ACTIVE	V1.00.04.00	28-MAR-2018

Figure 6-3. C2000ware Installation.

6.2 Download and Install Slave Stack Code Tool

In order to download SSC Tool:

1. Navigate to EtherCAT Technology Group's website and download SSC Tool from [45] as shown in Figure 6-4.

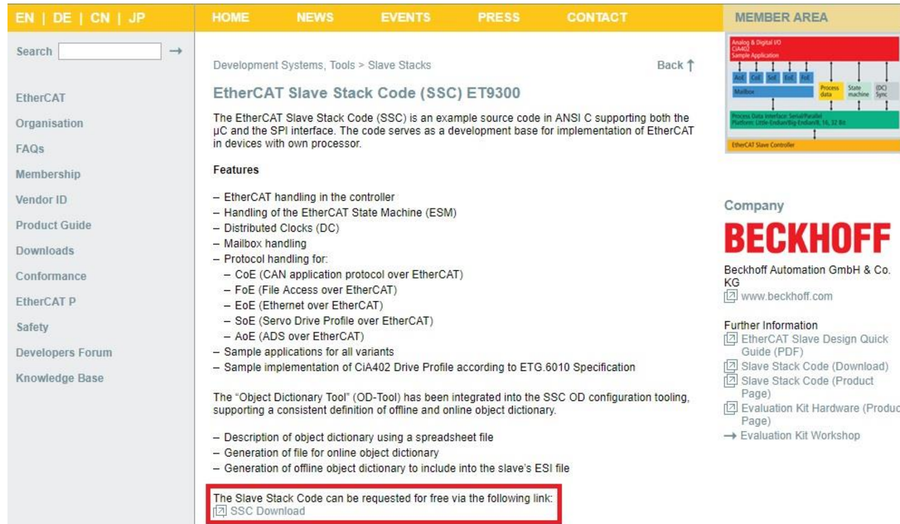


Figure 6-4. SSC Tool Download.

2. Login to the Member Area and insert the EtherCAT Vendor ID using the credentials of CSL-EP to initiate the downloading of the application.

3. Add your personal information to request the Slave Stack Code download link provided by email afterwards and Register.

4. Run *EtherCAT Slave Stack Code Tool.exe* file as administrator as shown in Figure 6-5.

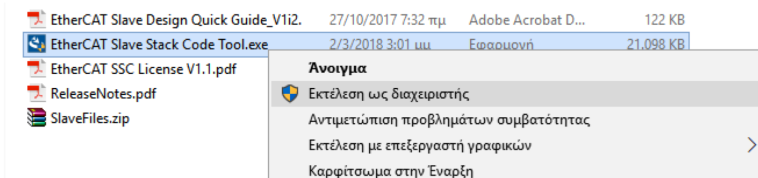


Figure 6-5. SSC Tool Installation.

6.3 Generate Slave Stack Code for C28x architecture microcontrollers

This section adumbrates the procedure of generating the necessary stack that must be downloaded to a C28x architecture microcontroller to implement an EtherCAT network. The four prerequisites of this process is to have downloaded and installed SSC Tool, Code Composer Studio, Control Suite and C2000ware to your PC as described in Download and Install Code Composer Studio, C2000ware & ControlSuite and Download and Install Slave Stack Code Tool. In order to generate the EtherCAT stack:

1. Navigate to the folder where you downloaded *ControlSuite*, point to controlSUITE\development_kits\TMDSECATCND379D_Vx folder and execute (as administrator) *EtherCAT_Slave_Demo_Code_v01_00_00_00_setup.exe* file as shown in Figure 6-6.

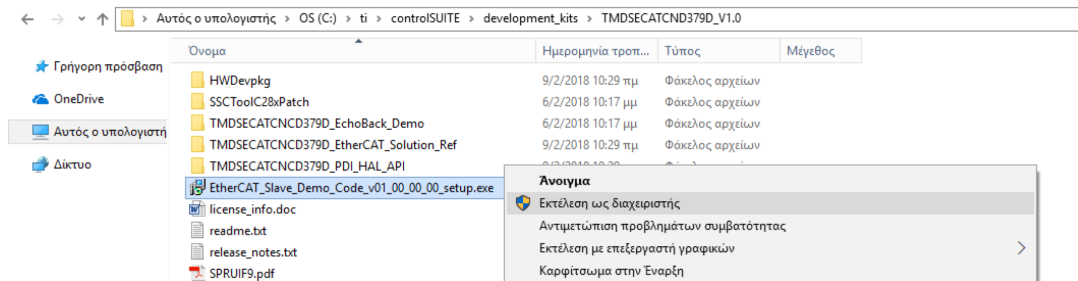


Figure 6-6. ControlSuite EtherCAT Demo Tool.

2. Open SSC tool and create a new project. The dialog box of Figure 6-7 appears.

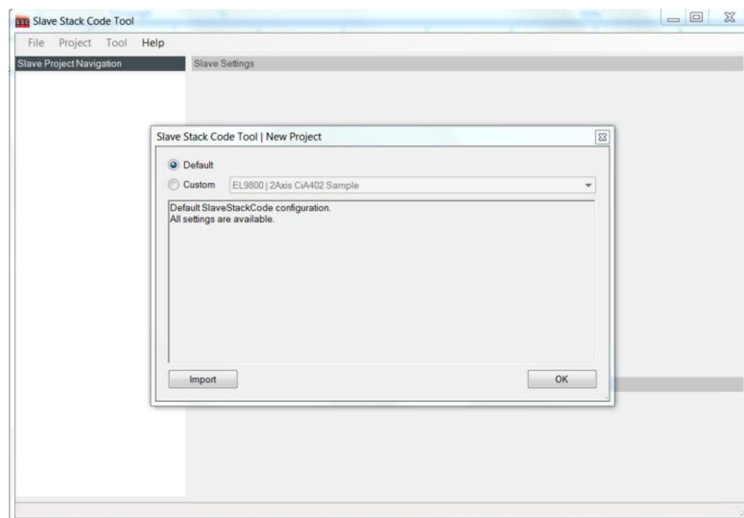


Figure 6-7. SSC Tool Create new project.

3. Click Import, point to the C28xx_Config.xml, located in the **SSCToolC28xPatch** folder created after completion of **1** as shown in Figure 6-8.

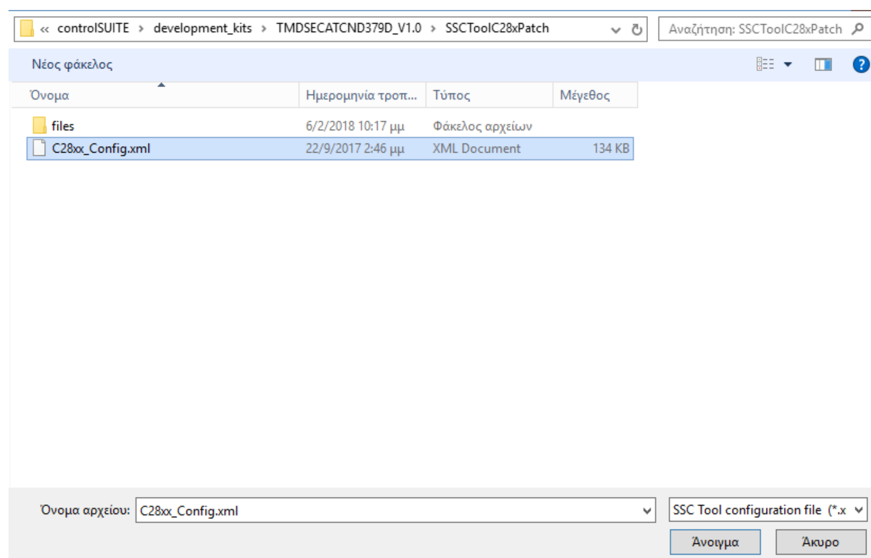


Figure 6-8. Importing ESI description file.

Figure 6-9 shows the pop up window when the C28xx_Config.xml is imported for the first time by the SSC Tool.

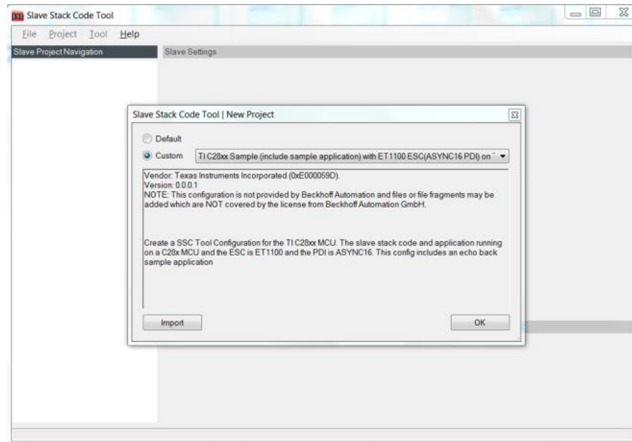


Figure 6-9. Slave Stack Code - New Project.

4. When the user selects the drop-down menu, the options shown in Figure 6-10 are provided.

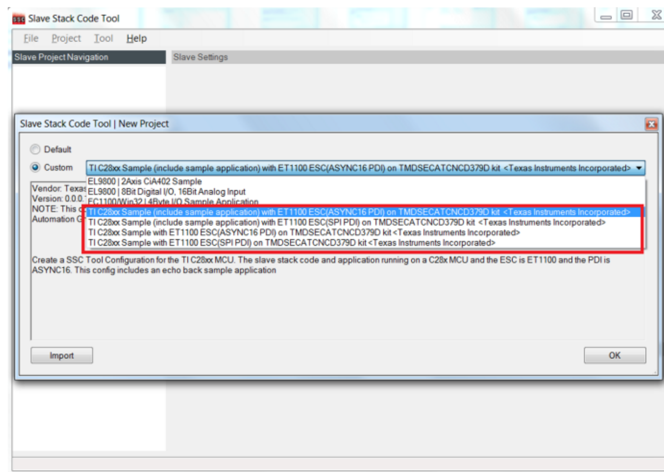


Figure 6-10. SSC Tool Configuration Options.

Four different options are available for C28x architecture microcontrollers created by Texas Instruments as follows:

- Option 1 generates EtherCAT slave stack code and EtherCAT EchoBack (sends and receives the same variables through EtherCAT network to test the communication) sample application code for ASYNC16 Process Data Interface (PDI) which depicts the communication protocol that is being used between the EtherCAT Slave Controller and the C2000 Delfino MCU to exchange data.
- Option 2 generates EtherCAT slave stack code and EtherCAT EchoBack sample application code for SPI PDI.
- Options 3 and 4 generate EtherCAT slave stack code for ASYNC16 and SPI PDI, without any default EchoBack sample application.

Note: Among SPI and ASYNC16 PDIs, there is no difference between the EtherCAT slave stack code and application code. Only the device name and product code differ, so both SPI and ASYNC16 slave nodes can be differentiated when they are both in the same network. For the EchoBack slave node profiles, the ESI files generated for SPI and ASYNC16 PDIs are also the same except for the device name and product code.

5. Choose an option (preferably one with a sample application), then click OK and click Yes, as shown in Figure 6-11.

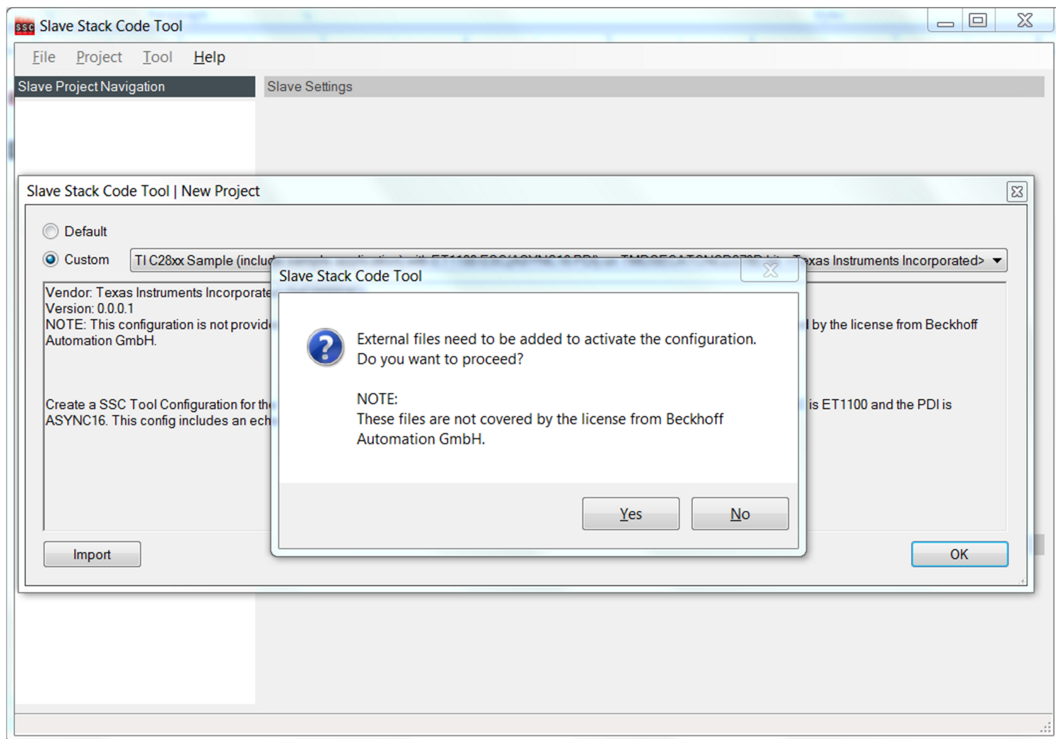


Figure 6-11. Importing project confirmation.

6. Now the C28xx configuration should be imported. Inspect the slave information as shown in Figure 6-12.

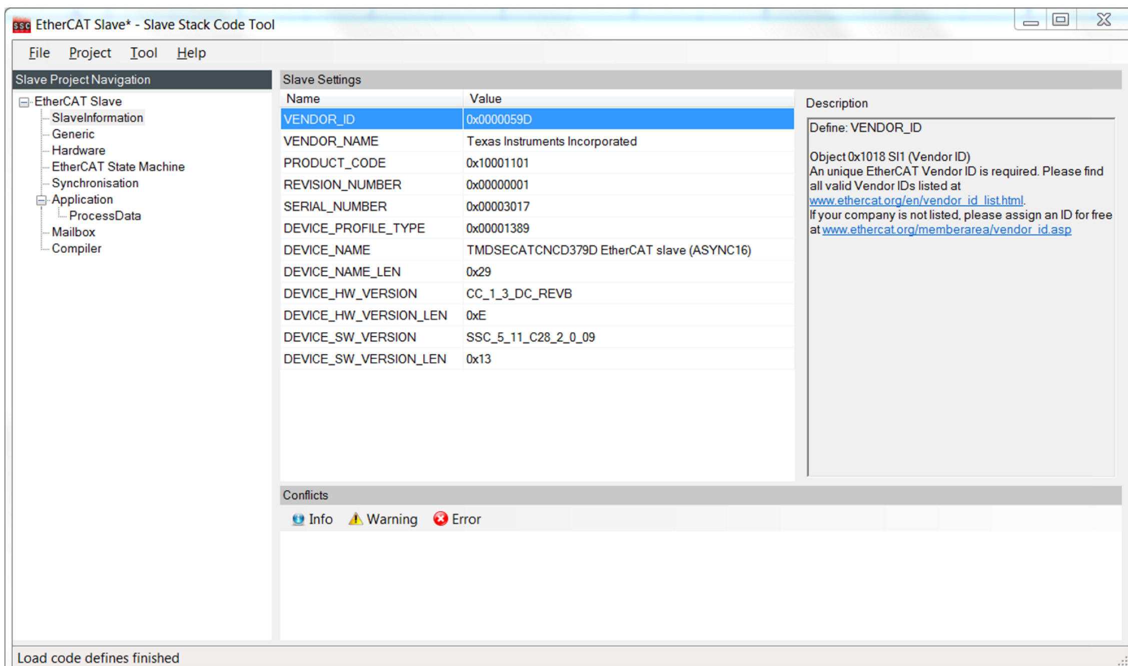


Figure 6-12. SSC Tool slave information.

7. Save the SSC Project in the following folder C:\working

8. Select Project → Create new Slave Files as shown Figure 6-13.

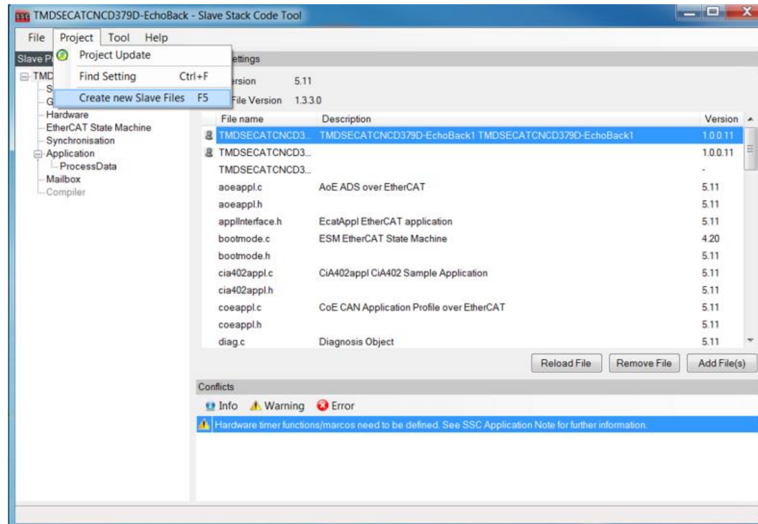


Figure 6-13. Create new Slave Files.

9. Input the Source Folder (C:\working\Src) and ESI File path (C:\working\TMDSECATCNCD379D EtherCAT slave (SPI).xml), or check where it is already defined and click Start as shown in Figure 6-14.

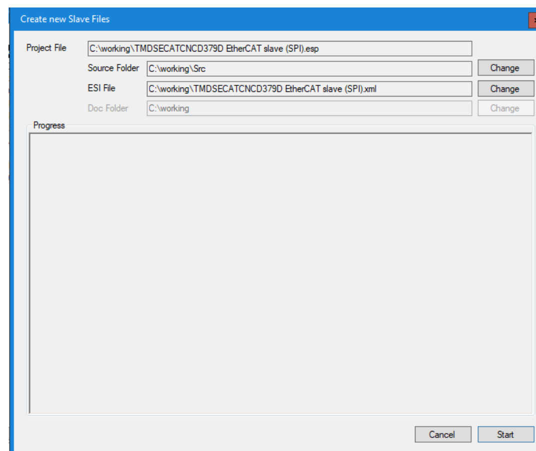


Figure 6-14. Create EtherCAT stack and xml file.

10. The slave node source files must be created. Click OK and then close the pop up window.

11. Inspect the directory in which the files were created. It must be identical with Figure 6-15.

Όνομα	Ημερομηνία τροπ...	Τύπος	Μέγεθος
Src	13/10/2017 11:13 μμ	Φάκελος αρχείων	
TMDSECATCNCD379D EtherCAT slave (SPI).esp	13/10/2017 11:08 μμ	Αρχείο ESP	431 KB
TMDSECATCNCD379D EtherCAT slave (SPI).xml	13/10/2017 11:13 μμ	Έγγραφο XML	86 KB

Figure 6-15. EtherCAT project files.

The Src folder must contain all the slave stack files and the default sample Echoback application that were generated by the tool. The esp is the slave stack project file for the slave stack tool. Users can open this file in the SSC tool and edit the project as needed and regenerate the files. The xml is the generated ESI file which must be updated with the EtherCAT master in the network to which this slave node will be connected.

12. Users must copy all the generated stack files under the Src folder to the TMDSECATCNCD379D_EtherCAT_Solution_Ref CCS project located in \controlSUITE\development_kits\TMDSECATCND379D_Vx\TMDSECATCNCD379D_EtherCAT_Solution_Ref folder as shown in Figure 6-16.

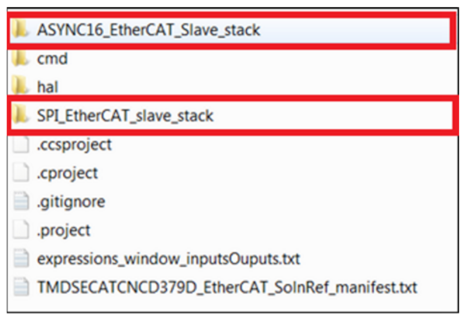


Figure 6-16. CCS project tree.

a. If the slave stack sources were generated from the SSC tool for ASYNC16 PDI, then they must be copied to the ASYNC16_EtherCAT_Slave_stack folder.

b. If they were generated from the SSC tool for SPI PDI, then they must be copied to the SPI_EtherCAT_slave_stack folder.

13. The project is now ready to be imported into Code Composer Studio. For more information regarding the aforementioned procedure see SPRUIG9.pdf (TMDSECATCNCD379D EtherCAT Solution Reference Guide) file located in \controlSUITE\development_kits\TMDSECATCND379D_Vx.

6.4 Download and Install TwinCAT 3 Software

In order to download and install TwinCAT, visit Beckhoff's official website at [46] and navigate to Download→Software→TwinCAT 3→TE1xxx | Engineering.

1. Select *TwinCAT 3.1 – eXtended Automation Engineering (XAE)* as shown in Figure 6-17 and press *Start Download* either as a guest or create an account in Beckhoff.

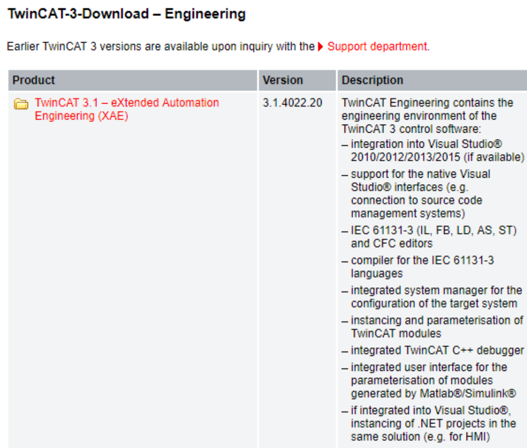


Figure 6-17. TwinCAT 3 Download scheme.

Note: Always check whether the latest version of Windows Operating Systems downloaded in your personal computer is compatible with the latest version of TwinCAT which you are about to download

because I have encountered several errors and disfunctinalities if that was the case. In the majority of cases, a warning message will emerge by Beckhoff before the downloas is initiated indicating that there is an incompatibility between Windows and TwinCAT.

2. Run the downloaded executable file as administrator as shown in Figure 6-18.

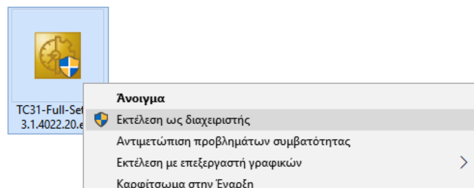


Figure 6-18. TwinCAT 3 Installation

Note: Although the TwinCAT 3.1 XAE is integrated in the Microsoft Visual Studio development environment, a previous installation of the latter software is not necessary since in case no Visual Studio installation is available on your PC, the TwinCAT 3.1 setup will install the Visual Studio shell as well.

3. Navigate to the installation folder of TwinCAT and run the *TcSwitchRuntime.exe* file as administrator located in *C:\TwinCAT\TcSwitchRuntime* folder as shown in Figure 6-19

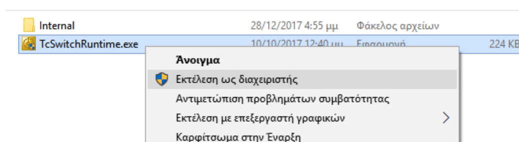


Figure 6-19. TcSwitchRuntime Installation.

4. Verify that the TcSwitchRuntime is active. The "Deactivate" button should be showing as illustrated in Figure 6-20. If this button indicates "Activate", click that button to start the TcSwitchRuntime.

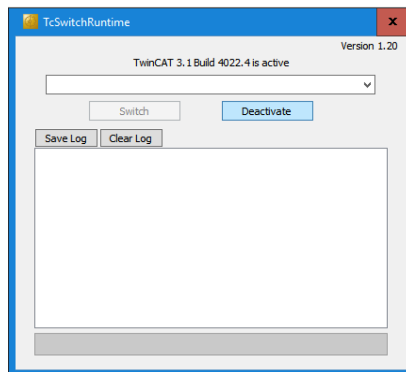


Figure 6-20. TcSwitchRuntime Activation.

5. Locate and start *TwinCAT XAE (VS 2013)* application and verify that TwinCAT is running under Visual Studio. "TwinCAT" and "PLC" options should both appear in the main toolbar as shown in Figure 6-21. If the aforementioned menu items are not shown, then the TcSwitchRuntime is not running properly. Go back to step 4 and restart (Deactivate→Activate) the TcSwitchRuntime.

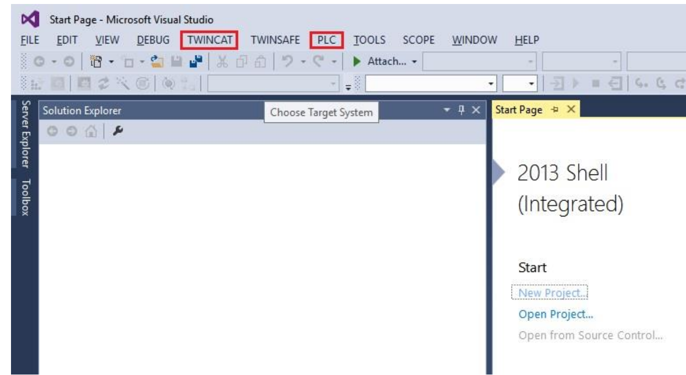


Figure 6-21. TwinCAT Verification.

6. Verify that a Realtime Ethernet Adapter is installed. Select TwinCAT→Show Realtime Ethernet Compatible Devices as shown in Figure 6-22. If no Real Time adapter is installed, select one from the list of **Compatible** devices and click "Install", then exit this popup window.

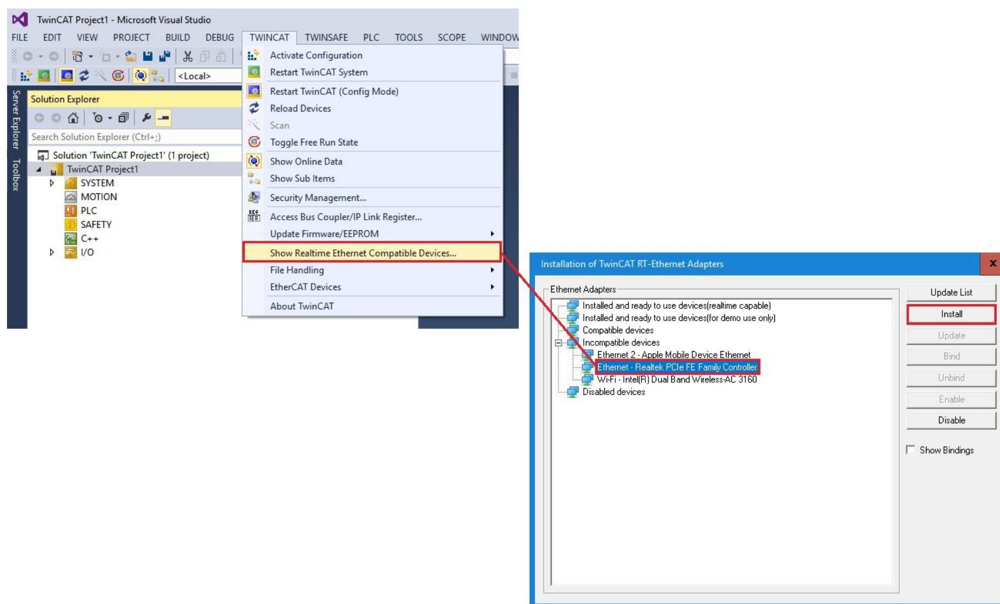


Figure 6-22. Real Time Ethernet Adapter Installation.

If a network card is listed under "Incompatible devices", this does not mean that it cannot be used to test the EtherCAT communication. It only means that this card will provide only weak real-time capabilities and will never switch to RUN mode (proper EtherCAT communication). On the contrary, it will only enable the FREE RUN mode which is regarded as an intermediate state that does not allow the renowned EtherCAT synchronization. For most of the testing purposes this is sufficient, therefore the driver can be installed. However, in our application, we will be using the DC Synchronous mode (we will elaborate in the following chapters), hence it is imperative to download adapters for compatible devices. Refer to EtherCAT Master Requirements in order to find out which networks cards are compatible with real time EtherCAT communication.

If the installation was successfully completed, the network card will be moved under the "Installed and ready to use devices" list:

- Compatible devices → Installed and ready to use devices (realtime capable)

- Incompatible devices → Installed and ready to use devices (for demo use only)

6.5 Import CCS project into Code Composer Studio

In order to Import a project in CCS and be able to download it into the desired microcontroller:

1. Start Code Composer Studio. The pop up window must look like Figure 6-23.

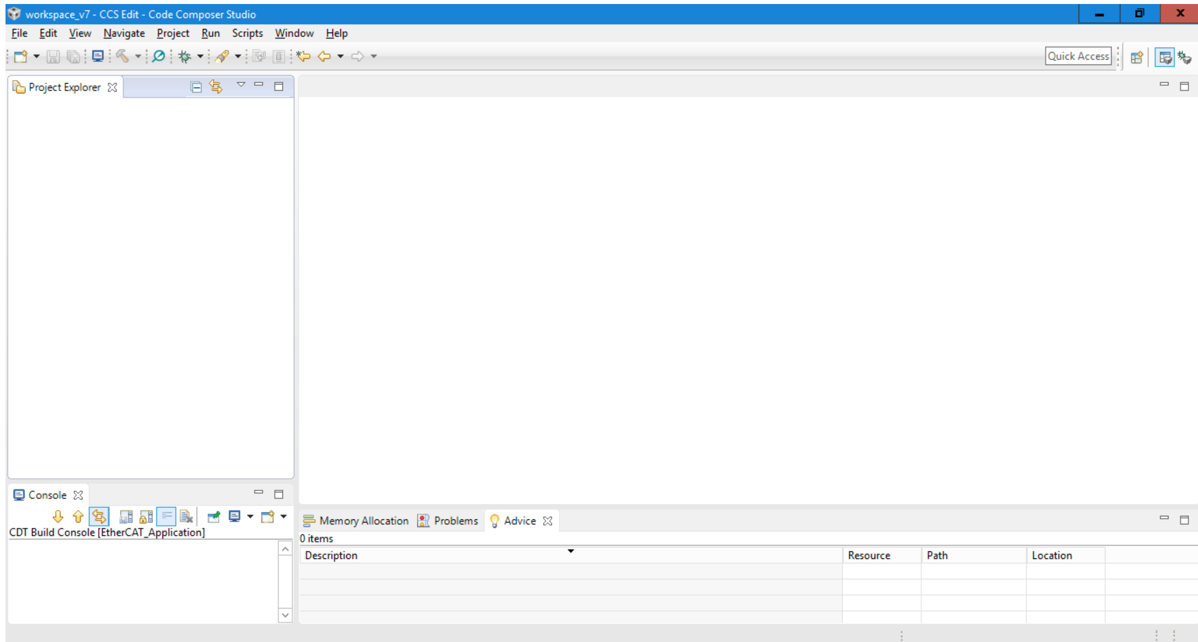


Figure 6-23. Code Composer Studio starting page.

2. Select *File* > *Import* as shown in Figure 6-24.

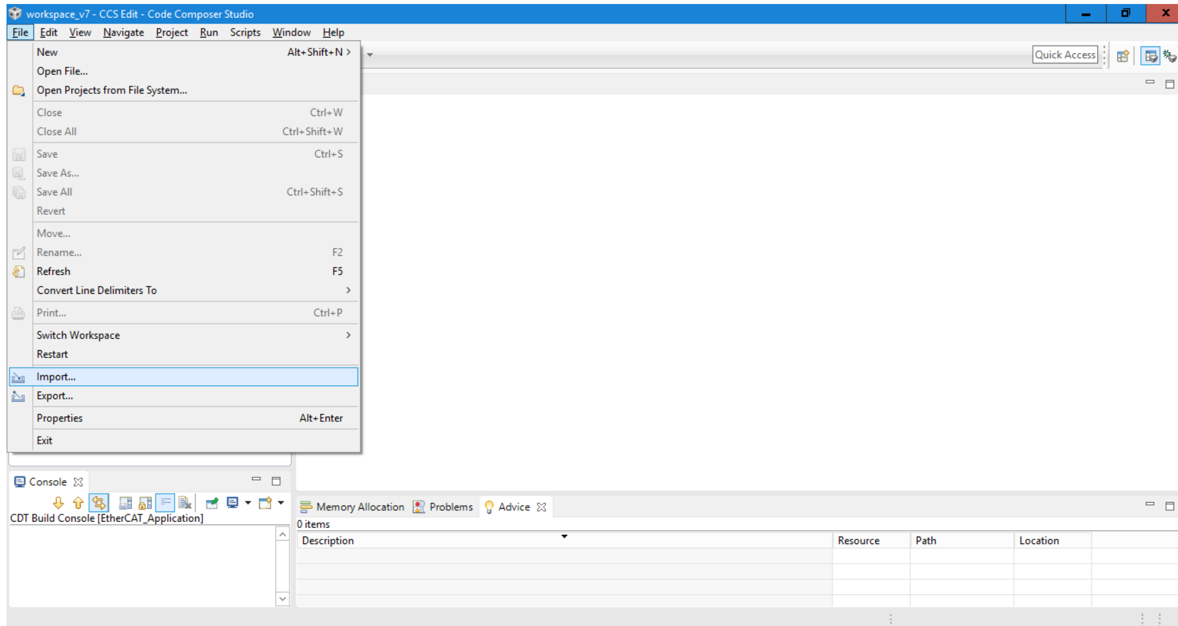


Figure 6-24. CCS Import project.

3. From the pop up window select *CCS Projects* and then click *Next* as shown in Figure 6-25.

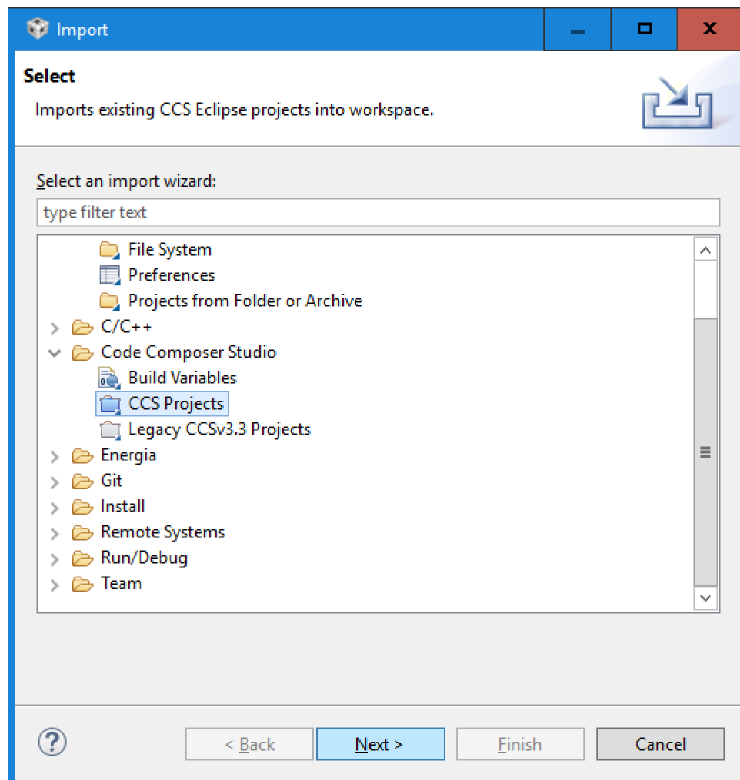


Figure 6-25. CCS project import selection.

4. *Browse* to the directory of the desired CCS project that you intend to import, click OK and then *Finish* as depicted in Figure 6-26.

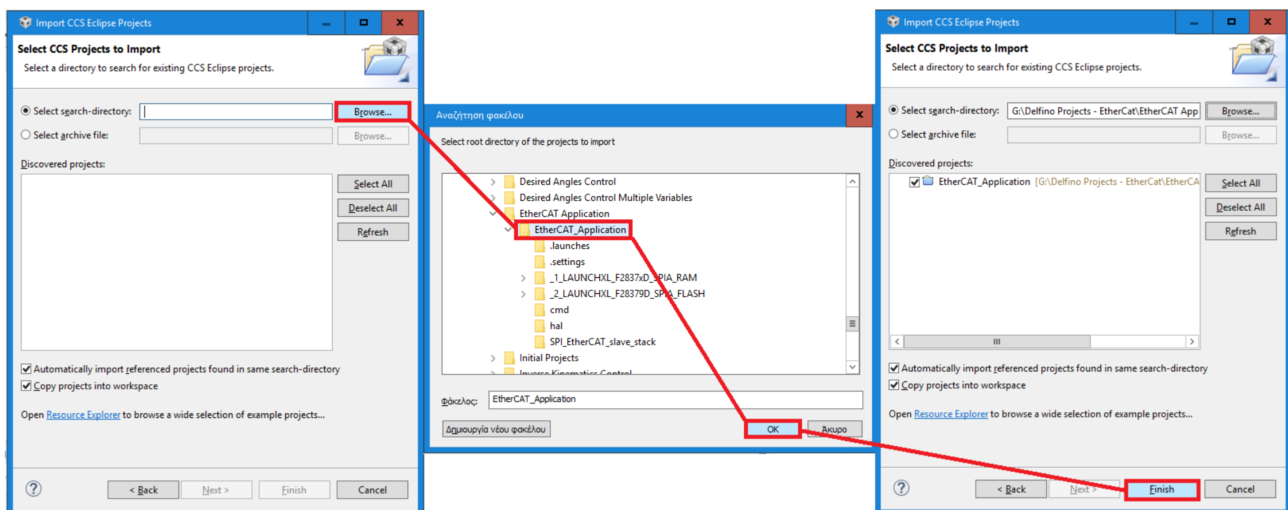


Figure 6-26. CCS browse and import.

5. If no error arises during this process, the project will show up on the *Project Explorer* (left hand side) of CCS window as illustrated in Figure 6-27 and developers may expand the tree to observe its main components.

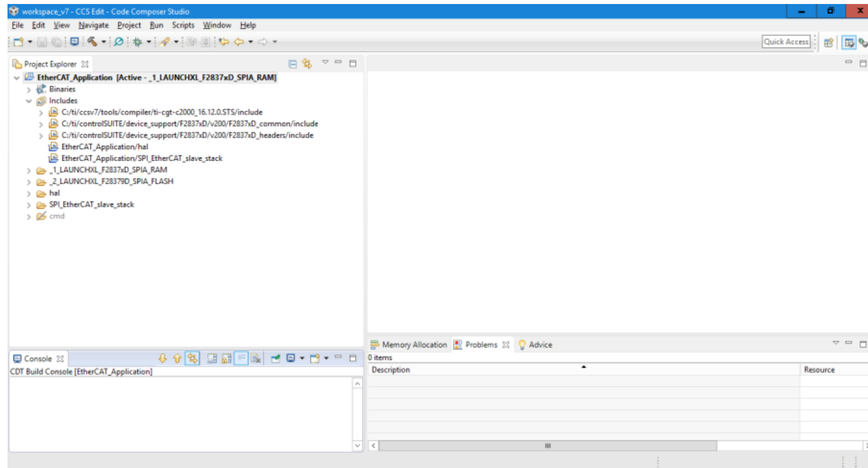


Figure 6-27. Project imported CCS window.

6.6 Define and Select Target Configuration

In order to define a target Configuration:

1. Select *View* > *Target Configuration* as depicted in Figure 6-28.

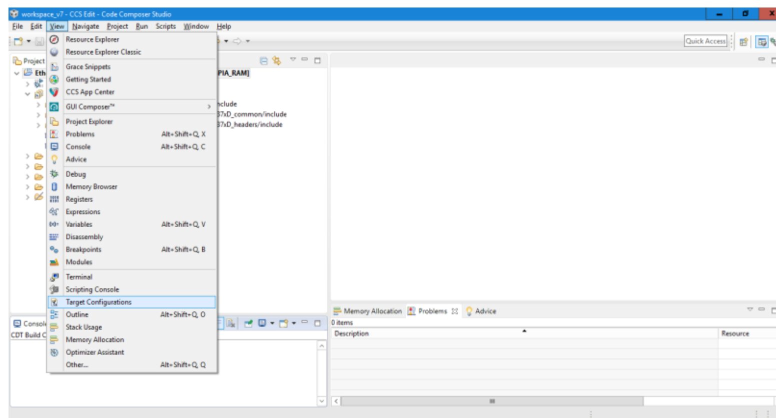


Figure 6-28. Select View Target Configuration.

2. Select *New Target Configuration* from the *Target Configurations* window.

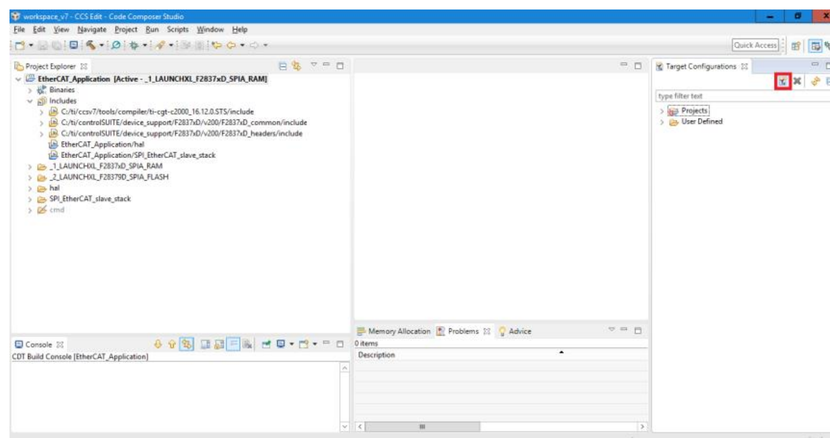


Figure 6-29. New Target Configuration.

3. Name the Target Configuration after the MCU being used and click *Finish*.

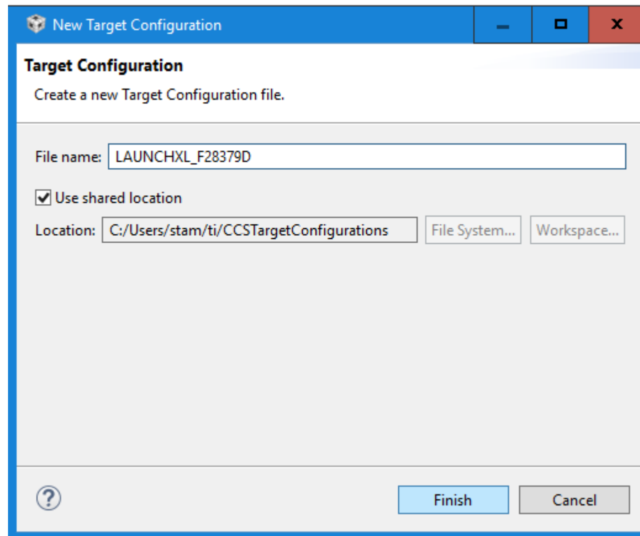


Figure 6-30. Name Target Configuration.

4. Select *Texas Instruments XDS100v2 USB Debug Probe* at the *Connection* tab, select *TMS320F28379D* at the *Board or Device* tab and click *Save* as shown in Figure 6-31.

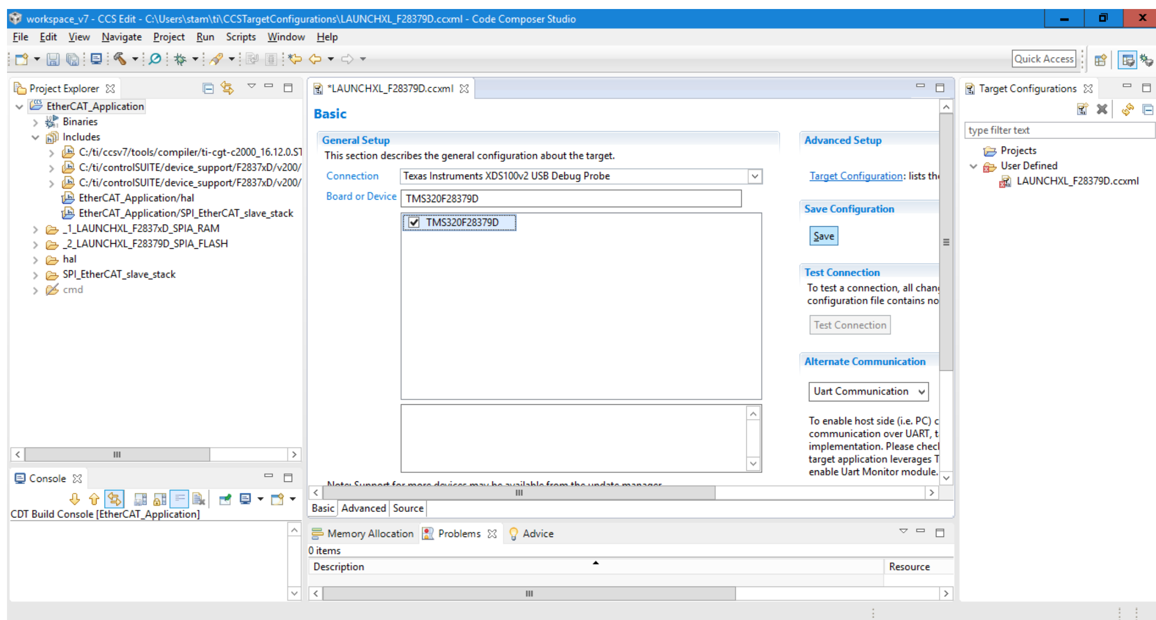


Figure 6-31. Select Connection and Device.

5. Close the *LAUNCXL_F28379D.ccxml* window after saving is completed.

In order now to link this configuration to your project:

6. At the *Target Configurations* window (right hand side) expand the *User Defined* directory, *right click* on *LAUNCHXL_F28379D.ccxml* and select *Link File To Project > [Project you want]* (ex EtherCAT_Application) as shown in Figure 6-32.

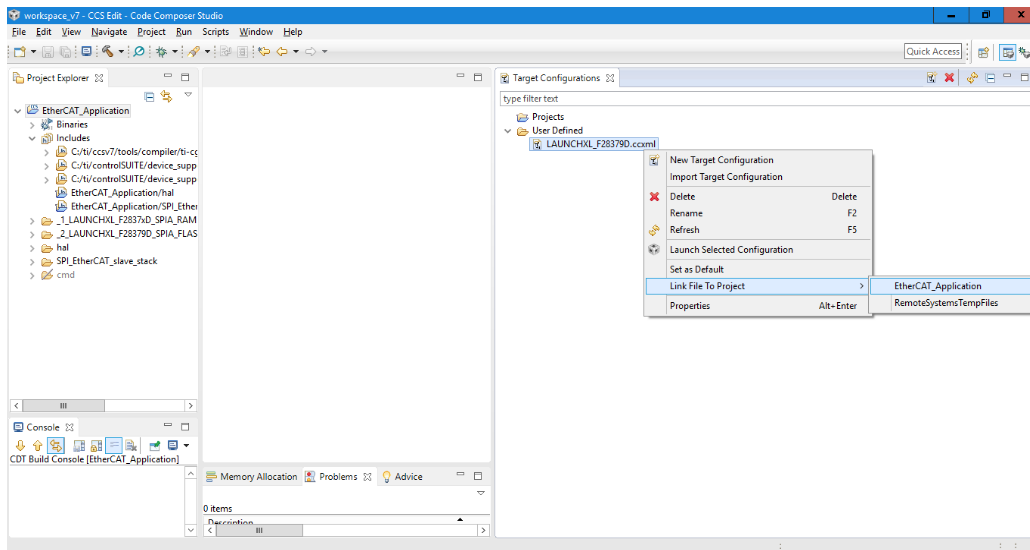


Figure 6-32. Link ccxml file to Project.

7. Close *Target Configurations* window.

6.7 Add and remove EtherCAT Input and Output variables

In this section we will describe the process of adding a Record of two output variables (16bit signed integers) in an EtherCAT slave device and how to configure these additions in both the ESC memory and the Delfino MCU [5] . The name of the Record will be *Additions* (0x7030) and the two variables will be named Out1 and Out2. In general, the address of all output variables begins with 0x70 and all input variables with 0x60. The procedure of adding input variables or removing a variable from the slave is identical and will not be separately explained. For the purpose of this section, we will use *EtherCAT Application* project which is described in EtherCAT Application Guide.

1. Download and import *EtherCAT Application* project to Code Composer Studio.

2. Open *EtherCAT Application.c* file. In the PDO_ResetOutputs() function add the indicated initialization part of Figure 6-33. If we were to add input variables, this step would be skipped.

```
55 \brief This function resets the outputs
56 *////////////////////////////////////////////////////////////////////////////////////////
57
58 void PDO_ResetOutputs(void)
59 {
60     Buttons0x7000.Button1 = 0x0;
61     Buttons0x7000.Button2 = 0x0;
62     Buttons0x7000.Button3 = 0x0;
63     Buttons0x7000.Button4 = 0x0;
64     Buttons0x7000.Blue_LED = 0x0;
65     Buttons0x7000.Red_LED = 0x0;
66     Buttons0x7000.Button7 = 0x0;
67     Buttons0x7000.Button8 = 0x0;
68     Buttons0x7000.Sync = 0x0;
69
70     Output1INT320x7010.OutINT32Var1 = 0x0;
71     OutputUINT160x7012.OutUINT16Var1 = 0x0;
72     Output2INT320x7014.OutINT32Var2 = 0x0;
73
74     OutputINT160x7020.OutINT16Var1 = 0x0;
75     OutputINT160x7020.OutINT16Var2 = 0x0;
76     OutputINT160x7020.OutINT16Var3 = 0x0;
77     OutputINT160x7020.OutINT16Var4 = 0x0;
78     OutputINT160x7020.OutINT16Var5 = 0x0;
79     OutputINT160x7020.OutINT16Var6 = 0x0;
80
81     Additions0x7030.Out1 = 0x0;
82     Additions0x7030.Out2 = 0x0;
83
84 }
```

Figure 6-33. Add initialization definition of new variables.

3. Update the **pOutputSize* pointer within the *APPL_GeneratingMapping()* function as shown in Figure 6-34. Since we added two, 16bit output varialbes which equals to four extra bites, the new value of **pOutputSize* pointer will become 28 (24 (before) + 4 (extra bytes)) whereas the *pInputSize value will remain the same.

```
UINT16 APPL_GenerateMapping(UINT16 *pInputSize,UINT16 *pOutputSize)
{
    *pInputSize = 22;
    *pOutputSize = 28;
    return ALSTATUSCODE_NOERROR;
}
```

Figure 6-34. Update pOutputSize value.

4. Update *APPL_OutputMapping()* function with the definiotions of the new variables. If we had new input variables as well, we would update *APPL_InputMapping()* function as well accordingly.

```
249 void APPL_OutputMapping(UINT16* pData)
250 {
251     uint16_t *pTmpData = (uint16_t *)pData;// allow byte processing
252     uint16_t data = 0;
253
254     /* RxPDO */
255     data = (*(volatile uint16_t *)pTmpData);
256     (Buttons0x7000.Button1) = data & 0x1;
257     data = data >> 1;
258     (Buttons0x7000.Button2) = data  & 0x1;
259     data = data >> 1;
260     (Buttons0x7000.Button3) = data  & 0x1;
261     data = data >> 1;
262     (Buttons0x7000.Button4) = data  & 0x1;
263     data = data >> 1;
264     (Buttons0x7000.Blue_LED) = data  & 0x1;
265     data = data >> 1;
266     (Buttons0x7000.Red_LED) = data  & 0x1;
267     data = data >> 1;
268     (Buttons0x7000.Button7) = data  & 0x1;
269     data = data >> 1;
270     (Buttons0x7000.Button8) = data  & 0x1;
271     data = data >> 1;
272     (Buttons0x7000.Sync) = data  & 0xFF;
273     pTmpData++;
274
275     memcpy(&Output1INT320x7010.OutINT32Var1,pTmpData,SIZEOF(Output1INT320x7010.OutINT32Var1));
276     pTmpData += 2;
277     memcpy(&OutputUINT160x7012.OutUINT16Var1,pTmpData,SIZEOF(OutputUINT160x7012.OutUINT16Var1));
278     pTmpData ++;
279     memcpy(&Output2INT320x7014.OutINT32Var2,pTmpData,SIZEOF(Output2INT320x7014.OutINT32Var2));
280     pTmpData += 2;
281
282     memcpy(&OutputINT160x7020.OutINT16Var1,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var1));
283     pTmpData ++;
284     memcpy(&OutputINT160x7020.OutINT16Var2,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var2));
285     pTmpData ++;
286     memcpy(&OutputINT160x7020.OutINT16Var3,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var3));
287     pTmpData ++;
288     memcpy(&OutputINT160x7020.OutINT16Var4,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var4));
289     pTmpData ++;
290     memcpy(&OutputINT160x7020.OutINT16Var5,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var5));
291     pTmpData ++;
292     memcpy(&OutputINT160x7020.OutINT16Var6,pTmpData,SIZEOF(OutputINT160x7020.OutINT16Var6));
293
294     pTmpData ++;
295     memcpy(&Additions0x7030.Out1,pTmpData,SIZEOF(Additions0x7030.Out1));
296     pTmpData ++;
297     memcpy(&Additions0x7030.Out2,pTmpData,SIZEOF(Additions0x7030.Out2));
298 }
```

Figure 6-35. Update APPL_OutputMapping() function with new variables.

5. Open *EtherCAT_ApplicationObjects.h* file and append the object address definition of the new variables under *Object 0x1602 : OutputINT16* shown in Figure 6-36. Note that this object is only and address reference project and does not describe the nature of the variables. This is the reason why in lines 239 and 240, the variable’s type has been declared as an unsigned 32 bit variable. Moreover, the object addressing for output variables starts with 0x16 and for input variables with 0x1A and because we are adding *0x7030* output record, the object address definition is *0x1603.*

```
221 /*******************************************************************************
222 *                        Object 0x1603 : Additions
223 ******************************************************************************/
224 /**
225 * \addtogroup 0x1603 0x1603 | Additions
226 * @{
227 * \brief Object 0x1603 (Additions) definition
228 */
229 #ifdef _OBJD_
230 /**
231 * \brief Object entry descriptions<br>
232 * <br>
233 * SubIndex 0<br>
234 * SubIndex 1 - Reference to 0x7030.1<br>
235 * SubIndex 2 - Reference to 0x7030.2<br>
236 */
237 OBJCONST TSDOINFOENTRYDESC    OBJMEM asEntryDesc0x1603[] = {
238 { DEFTYPE_UNSIGNED8 , 0x8 , ACCESS_READ },
239 { DEFTYPE_UNSIGNED32 , 0x20 , ACCESS_READ }, /* Subindex1 - Reference to 0x7030.1 */
240 { DEFTYPE_UNSIGNED32 , 0x20 , ACCESS_READ }}; /* Subindex2 - Reference to 0x7030.2 */
241
242 /**
243 * \brief Object/Entry names
244 */
245 OBJCONST UCHAR OBJMEM aName0x1603[] = "Additions\000"
246 "SubIndex 001\000"
247 "SubIndex 002\000\377";
248 #endif //#ifdef _OBJD_
249
250 #ifndef _TMDSECATCNCD379_D_ECHO_BACK_OBJECTS_H_
251 /**
252 * \brief Object structure
253 */
254 typedef struct OBJ_STRUCT_PACKED_START {
255 UINT16 u16SubIndex0;
256 UINT32 SI1; /* Subindex1 - Reference to 0x7000.1 */
257 UINT32 SI2; /* Subindex2 - Reference to 0x7000.2 */
258 } OBJ_STRUCT_PACKED_END
259 TOBJ1603;
260 #endif //#ifndef _TMDSECATCNCD379_D_ECHO_BACK_OBJECTS_H_
261
262 /**
263 * \brief Object variable
264 */
265 PROTO TOBJ1603 Additions0x1603
266 #if defined(_TMDSECATCNCD379_D_ECHO_BACK_) && (_TMDSECATCNCD379_D_ECHO_BACK_ == 1)
267 ={2,0x70300110,0x70300210}
268 #endif
269 ;
270 /** @}*/
271
```

Figure 6-36. New variables object address definition.

6. Update *Object 0x1C12 : SyncManager 2 assignment* as indicated in , with the new variable additions. Note that if we were to add input variables, then we would have to update *Object 0x1C13 : SyncManager 3* definition.

```
430 /*******************************************************************************
431 *                        Object 0x1C12 : SyncManager 2 assignment
432 ******************************************************************************/
433 /**
434 * \addtogroup 0x1C12 0x1C12 | SyncManager 2 assignment
435 * @{
436 * \brief Object 0x1C12 (SyncManager 2 assignment) definition
437 */
438 #ifdef _OBJD_
439 /**
440 * \brief Entry descriptions<br>
441 *
442 * Subindex 0<br>
443 * Subindex 1 - n (the same entry description is used)<br>
444 */
445 OBJCONST TSDOINFOENTRYDESC    OBJMEM asEntryDesc0x1C12[] = {
446 { DEFTYPE_UNSIGNED8 , 0x8 , ACCESS_READ },
447 { DEFTYPE_UNSIGNED16 , 0x10 , ACCESS_READ }};
448
449 /**
450 * \brief Object name definition<br>
451 * For Subindex 1 to n the syntax 'Subindex XXX' is used
452 */
453 OBJCONST UCHAR OBJMEM aName0x1C12[] = "SyncManager 2 assignment\000\377";
454 #endif //#ifdef _OBJD_
455
456 #ifndef _TMDSECATCNCD379_D_ECHO_BACK_OBJECTS_H_
457 /**
458 * \brief Object structure
459 */
460 typedef struct OBJ_STRUCT_PACKED_START {
461 UINT16   u16SubIndex0;  /**< \brief Subindex 0 */
462 UINT16 aEntries[4];  /**< \brief Subindex 1 - 4 */
463 } OBJ_STRUCT_PACKED_END
464 TOBJ1C12;
465 #endif //#ifndef _TMDSECATCNCD379_D_ECHO_BACK_OBJECTS_H_
466
467 /**
468 * \brief Object variable
469 */
470 PROTO TOBJ1C12 sRxPDOassign
471 #if defined(_TMDSECATCNCD379_D_ECHO_BACK_) && (_TMDSECATCNCD379_D_ECHO_BACK_ == 1)
472 = 4,{0x1600,0x1601,0x1602,0x1603}}
473 #endif
474 ;
475 /** @}*/
```

Figure 6-37. Update SyncManager assignment.

7. Insert the *Additions* object record definition of Figure 6-38 under the *Object 0x7020 : OutputINT16.*

```
1062 /******************************************************************************
1063 *                    Object 0x7030 : Additions
1064 ******************************************************************************/
1065 /**
1066 * \addtogroup 0x7030 0x7030 | Additions
1067 * @{
1068 * \brief Object 0x7020 (Additions) definition
1069 */
1070 #ifdef _OBJD_
1071 /**
1072 * \brief Object entry descriptions<br>
1073 * <br>
1074 * SubIndex 0<br>
1075 * SubIndex 1 - Out1<br>
1076 * SubIndex 2 - Out2<br>
1077 */
1078 OBJCONST TSDOINFOENTRYDESC    OBJMEM asEntryDesc0x7030[] = {
1079 { DEFTYPE_UNSIGNED8 , 0x8 , ACCESS_READ },
1080 { DEFTYPE_INTEGER16 , 0x10 , ACCESS_READWRITE | OBJACCESS_RXPDOMAPPING | OBJACCESS_SETTINGS }, /* Subindex1 - Out1 */
1081 { DEFTYPE_INTEGER16 , 0x10 , ACCESS_READWRITE | OBJACCESS_RXPDOMAPPING | OBJACCESS_SETTINGS }}; /* Subindex2 - Out2 */
1082 /**
1083 * \brief Object/Entry names
1084 */
1085 OBJCONST UCHAR OBJMEM aName0x7030[] = "Additions\000"
1086 "Out1\000"
1087 "Out2\000\377";
1088 #endif //#ifdef _OBJD_
1089
1090 #ifndef _TMDSECATCNCD379_D_ECHO_BACK_OBJECTS_H_
1091 /**
1092 * \brief Object structure
1093 */
1094 typedef struct OBJ_STRUCT_PACKED_START {
1095 UINT16 u16SubIndex0;
1096 INT16 Out1; /* Subindex1 - Out1 */
1097 INT16 Out2; /* Subindex2 - Out2 */
1098 } OBJ_STRUCT_PACKED_END
1099 TOBJ7030;
1100 #endif //#ifndef _TMDSECATCNCD379_D_ECHO_BACK_OBJECTS_H_
1101
1102 /**
1103 * \brief Object variable
1104 */
1105 PROTO TOBJ7030 Additions0x7030
1106 #if defined(_TMDSECATCNCD379_D_ECHO_BACK_) && (_TMDSECATCNCD379_D_ECHO_BACK_ == 1)
1107 ={2,0x0000,0x0000}
1108 #endif
1109 ;
1110 /** @}*/
```

Figure 6-38. Object record definition of "Additions".

8. Update ApplicationObjDic[] with the definitions of the new variables (Figure 6-39).

```
1113 #ifdef _OBJD_
1114 TOBJECT    OBJMEM ApplicationObjDic[] = {
1115 /* Object 0x1600 */
1116 {NULL , NULL ,  0x1600 , {DEFTYPE_PDOMAPPING , 9 | (OBJCODE_REC << 8)} , asEntryDesc0x1600 , aName0x1600 , &ButtonsProcessDataMapping0x1600, NULL , NULL , 0x0000 },
1117 /* Object 0x1601 */
1118 {NULL , NULL ,  0x1601 , {DEFTYPE_PDOMAPPING , 3 | (OBJCODE_REC << 8)} , asEntryDesc0x1601 , aName0x1601 , &OutputMapping10x1601, NULL , NULL , 0x0000 },
1119 /* Object 0x1602 */
1120 {NULL , NULL ,  0x1602 , {DEFTYPE_PDOMAPPING , 6 | (OBJCODE_REC << 8)} , asEntryDesc0x1602 , aName0x1602 , &OutputINT160x1602, NULL , NULL , 0x0000 },
1121 /* Object 0x1603 */
1122 {NULL , NULL ,  0x1603 , {DEFTYPE_PDOMAPPING , 2 | (OBJCODE_REC << 8)} , asEntryDesc0x1603 , aName0x1603 , &Additions0x1603, NULL , NULL , 0x0000 },
1123 /* Object 0x1A01 */
1124 {NULL , NULL ,  0x1A01 , {DEFTYPE_PDOMAPPING , 3 | (OBJCODE_REC << 8)} , asEntryDesc0x1A01 , aName0x1A01 , &InputMapping10x1A01, NULL , NULL , 0x0000 },
1125 /* Object 0x1A02 */
1126 {NULL , NULL ,  0x1A02 , {DEFTYPE_PDOMAPPING , 2 | (OBJCODE_REC << 8)} , asEntryDesc0x1A02 , aName0x1A02 , &InputINT160x1A02, NULL , NULL , 0x0000 },
1127 /* Object 0x1A03 */
1128 {NULL , NULL ,  0x1A03 , {DEFTYPE_PDOMAPPING , 2 | (OBJCODE_REC << 8)} , asEntryDesc0x1A03 , aName0x1A03 , &Input3INT320x1A03, NULL , NULL , 0x0000 },
1129 /* Object 0x1C12 */
1130 {NULL , NULL ,  0x1C12 , {DEFTYPE_UNSIGNED16 , 4 | (OBJCODE_ARR << 8)} , asEntryDesc0x1C12 , aName0x1C12 , &sRxPDOassign, NULL , NULL , 0x0000 },
1131 /* Object 0x1C13 */
1132 {NULL , NULL ,  0x1C13 , {DEFTYPE_UNSIGNED16 , 3 | (OBJCODE_ARR << 8)} , asEntryDesc0x1C13 , aName0x1C13 , &sTxPDOassign, NULL , NULL , 0x0000 },
1133 /* Object 0x6010 */
1134 {NULL , NULL ,  0x6010 , {DEFTYPE_RECORD , 1 | (OBJCODE_REC << 8)} , asEntryDesc0x6010 , aName0x6010 , &Input1INT320x6010, NULL , NULL , 0x0000 },
1135 /* Object 0x6012 */
1136 {NULL , NULL ,  0x6012 , {DEFTYPE_RECORD , 1 | (OBJCODE_REC << 8)} , asEntryDesc0x6012 , aName0x6012 , &InputUINT160x6012, NULL , NULL , 0x0000 },
1137 /* Object 0x6014 */
1138 {NULL , NULL ,  0x6014 , {DEFTYPE_RECORD , 1 | (OBJCODE_REC << 8)} , asEntryDesc0x6014 , aName0x6014 , &Input2INT320x6014, NULL , NULL , 0x0000 },
1139 /* Object 0x6020 */
1140 {NULL , NULL ,  0x6020 , {DEFTYPE_RECORD , 2 | (OBJCODE_REC << 8)} , asEntryDesc0x6020 , aName0x6020 , &InputINT160x6020, NULL , NULL , 0x0000 },
1141 /* Object 0x6030 */
1142 {NULL , NULL ,  0x6030 , {DEFTYPE_RECORD , 2 | (OBJCODE_REC << 8)} , asEntryDesc0x6030 , aName0x6030 , &Input3INT320x6030, NULL , NULL , 0x0000 },
1143 /* Object 0x7000 */
1144 {NULL , NULL ,  0x7000 , {DEFTYPE_RECORD , 9 | (OBJCODE_REC << 8)} , asEntryDesc0x7000 , aName0x7000 , &Buttons0x7000, NULL , NULL , 0x0000 },
1145 /* Object 0x7010 */
1146 {NULL , NULL ,  0x7010 , {DEFTYPE_RECORD , 1 | (OBJCODE_REC << 8)} , asEntryDesc0x7010 , aName0x7010 , &Output1INT320x7010, NULL , NULL , 0x0000 },
1147 /* Object 0x7012 */
1148 {NULL , NULL ,  0x7012 , {DEFTYPE_RECORD , 1 | (OBJCODE_REC << 8)} , asEntryDesc0x7012 , aName0x7012 , &OutputUINT160x7012, NULL , NULL , 0x0000 },
1149 /* Object 0x7014 */
1150 {NULL , NULL ,  0x7014 , {DEFTYPE_RECORD , 1 | (OBJCODE_REC << 8)} , asEntryDesc0x7014 , aName0x7014 , &Output2INT320x7014, NULL , NULL , 0x0000 },
1151 /* Object 0x7020 */
1152 {NULL , NULL ,  0x7020 , {DEFTYPE_RECORD , 6 | (OBJCODE_REC << 8)} , asEntryDesc0x7020 , aName0x7020 , &OutputINT160x7020, NULL , NULL , 0x0000 },
1153 /* Object 0x7030 */
1154 {NULL , NULL ,  0x7030 , {DEFTYPE_RECORD , 2 | (OBJCODE_REC << 8)} , asEntryDesc0x7030 , aName0x7030 , &Additions0x7030, NULL , NULL , 0x0000 },
1155 {NULL,NULL, 0xFFFF, {0, 0}, NULL, NULL, NULL, NULL}};
1156 #endif    //#ifdef _OBJD_
```

Figure 6-39. Update of ApplicationObjDic[].

9. Now that we have completed the configuration of the new variables in the Dlefino MCU, we need to update the xml file (ENI description file) which will be used to write the EEPROM memory of our ESC. In order to do so, open *EtherCAT Application (SPI).xml* and insert the definition shown in Figure 6-40 in the *DataType* section under *DT1602.*

```
<DataType>
  <Name>DT1603</Name>
  <BitSize>80</BitSize>
  <SubItem>
    <SubIdx>0</SubIdx>
    <Name>SubIndex 000</Name>
    <Type>USINT</Type>
    <BitSize>8</BitSize>
    <BitOffs>0</BitOffs>
    <Flags>
      <Access>ro</Access>
      <Category>o</Category>
    </Flags>
  </SubItem>
  <SubItem>
    <SubIdx>1</SubIdx>
    <Name>SubIndex 001</Name>
    <Type>UDINT</Type>
    <BitSize>32</BitSize>
    <BitOffs>16</BitOffs>
    <Flags>
      <Access>ro</Access>
    </Flags>
    <!--Reference to 0x7030.1-->
  </SubItem>
  <SubItem>
    <SubIdx>2</SubIdx>
    <Name>SubIndex 002</Name>
    <Type>UDINT</Type>
    <BitSize>32</BitSize>
    <BitOffs>48</BitOffs>
    <Flags>
      <Access>ro</Access>
    </Flags>
    <!--Reference to 0x7030.2-->
  </SubItem>
</DataType>
```

Figure 6-40. DT1603 DataType definition.

10. Update the definitions of DT1C12 and DT1C12ARR DataTypes (SyncManager 2) as shown in Figure 6-41.

```
<DataType>
  <Name>DT1C12ARR</Name>
  <BaseType>UINT</BaseType>
  <BitSize>64</BitSize>
  <ArrayInfo>
    <LBound>1</LBound>
    <Elements>4</Elements>
  </ArrayInfo>
</DataType>
<DataType>
  <Name>DT1C12</Name>
  <BitSize>80</BitSize>
  <SubItem>
    <SubIdx>0</SubIdx>
    <Name>SubIndex 000</Name>
    <Type>USINT</Type>
    <BitSize>8</BitSize>
    <BitOffs>0</BitOffs>
    <Flags>
      <Access>ro</Access>
      <Category>o</Category>
    </Flags>
  </SubItem>
  <SubItem>
    <Name>Elements</Name>
    <Type>DT1C12ARR</Type>
    <BitSize>64</BitSize>
    <BitOffs>16</BitOffs>
    <Flags>
      <Access>ro</Access>
    </Flags>
  </SubItem>
</DataType>
```

Figure 6-41. DT1C12 and DT1C12ARR DataType definition.

11. Insert the DataType definition of Figure 6-42 under the DataType definition of DT7020.

```
        <Flags>
          <Access>rw</Access>
          <Category>m</Category>
          <PdoMapping>r</PdoMapping>
          <Setting>1</Setting>
        </Flags>
        <!--Ki Gain of hip Motor-->
      </SubItem>
  </DataType>
  <DataType>
     <Name>DT7030</Name>
     <BitSize>48</BitSize>
     <SubItem>
       <SubIdx>0</SubIdx>
       <Name>SubIndex 000</Name>
       <Type>USINT</Type>
       <BitSize>8</BitSize>
       <BitOffs>0</BitOffs>
       <Flags>
         <Access>ro</Access>
         <Category>m</Category>
       </Flags>
     </SubItem>
     <SubItem>
       <SubIdx>1</SubIdx>
       <Name>Out1</Name>
       <Type>INT</Type>
       <BitSize>16</BitSize>
       <BitOffs>16</BitOffs>
       <Flags>
         <Access>rw</Access>
         <Category>m</Category>
         <PdoMapping>r</PdoMapping>
         <Setting>1</Setting>
       </Flags>
       <!--Output variable 1-->
     </SubItem>
     <SubItem>
       <SubIdx>2</SubIdx>
       <Name>Out2</Name>
       <Type>INT</Type>
       <BitSize>16</BitSize>
       <BitOffs>32</BitOffs>
       <Flags>
         <Access>rw</Access>
         <Category>m</Category>
         <PdoMapping>r</PdoMapping>
         <Setting>1</Setting>
       </Flags>
       <!--Output variable 2-->
     </SubItem>
   </DataType>
  </DataTypes>
  <Objects>
    <Object>
      <Index>#x1000</Index>
      <Name>Device type</Name>
      <Type>UDINT</Type>
```

Figure 6-42. DT7030 DataType definition.

12. Insert the Object definition of Figure 6-43 under the Object definition of #x1602.

```
      <DefaultData>10062070</DefaultData>
      <!--Reference to 0x7020.6-->
    </Info>
  </SubItem>
</Info>
</Object>
<Object>
  <Index>#x1603</Index>
  <Name>Additions</Name>
  <Type>DT1603</Type>
  <BitSize>48</BitSize>
  <Info>
    <SubItem>
      <Name>SubIndex 000</Name>
      <Info>
        <DefaultData>08</DefaultData>
      </Info>
    </SubItem>
    <SubItem>
      <Name>SubIndex 001</Name>
      <Info>
        <DefaultData>10013070</DefaultData>
        <!--Reference to 0x7030.1-->
      </Info>
    </SubItem>
    <SubItem>
      <Name>SubIndex 002</Name>
      <Info>
        <DefaultData>10023070</DefaultData>
        <!--Reference to 0x7030.2-->
      </Info>
    </SubItem>
  </Info>
</Object>
<Object>
  <Index>#x1A01</Index>
  <Name>Input mapping 1</Name>
```

Figure 6-43. Object definition of #x1603.

13. Update the Object definition of Sync Manager 2 as shown in Figure 6-44.

```
<Object>
  <Index>#x1C12</Index>
  <Name>SyncManager 2 assignment</Name>
  <Type>DT1C12</Type>
  <BitSize>80</BitSize>
  <Info>
    <SubItem>
      <Name>SubIndex 000</Name>
      <Info>
        <DefaultData>02</DefaultData>
      </Info>
    </SubItem>
    <SubItem>
      <Name>SubIndex 001</Name>
      <Info>
        <DefaultData>0016</DefaultData>
      </Info>
    </SubItem>
    <SubItem>
      <Name>SubIndex 002</Name>
      <Info>
        <DefaultData>0116</DefaultData>
      </Info>
    </SubItem>
    <SubItem>
      <Name>SubIndex 003</Name>
      <Info>
        <DefaultData>0216</DefaultData>
      </Info>
    </SubItem>
    <SubItem>
      <Name>SubIndex 004</Name>
      <Info>
        <DefaultData>0316</DefaultData>
      </Info>
    </SubItem>
  </Info>
</Object>
```

Figure 6-44. Updated Object definition of #x1C12.

14. Insert the Object definition of #x7030 under the Object definition of #x7020 as shown in Figure 6-45

```
            <!--OutINT16Var6-->
          </Info>
        </SubItem>
      </Info>
    </Object>
    <Object>
      <Index>#x7030</Index>
      <Name>Additions</Name>
      <Type>DT7020</Type>
      <BitSize>48</BitSize>
      <Info>
        <SubItem>
          <Name>SubIndex 000</Name>
          <Info>
            <DefaultData>08</DefaultData>
          </Info>
        </SubItem>
        <SubItem>
          <Name>Out1</Name>
          <Info>
            <DefaultData>0000</DefaultData>
            <!--Out1 variable-->
          </Info>
        </SubItem>
        <SubItem>
          <Name>Out2</Name>
          <Info>
            <DefaultData>0000</DefaultData>
            <!--Out2 variable-->
          </Info>
        </SubItem>
      </Info>
    </Object>
  </Objects>
</Dictionary>
```

Figure 6-45. Object definition of #x7030.

15. Update the Output size of the Sync Manager as indicated in Figure 6-46.

```
    </Objects>
  </Dictionary>
</Profile>
<Fmmu>Outputs</Fmmu>
<Fmmu>Inputs</Fmmu>
<Fmmu>MBoxState</Fmmu>
<Sm MinSize="34" MaxSize="128" DefaultSize="128" StartAddress="#x1000" ControlByte="#x26" Enable="1">MBoxOut</Sm>
<Sm MinSize="34" MaxSize="128" DefaultSize="128" StartAddress="#x1080" ControlByte="#x22" Enable="1">MBoxIn</Sm>
<Sm StartAddress="#x1100" ControlByte="#x64" DefaultSize="28" Enable="1">Outputs</Sm>
<Sm StartAddress="#x1400" ControlByte="#x20" DefaultSize="22" Enable="1">Inputs</Sm>
<RxPdo Mandatory="true" Fixed="true" Sm="2">
  <Index>#x1600</Index>
  <Name>Buttons</Name>
  <Entry>
    <Index>#x7000</Index>
```

Figure 6-46. Update Sync Manager Output size.

16. Finally, insert the RxPdo definition of #x1603 as shown in Figure 6-47 under RxPdo definition of #x1602.

```
        <Name>OutINT16Var6</Name>
        <DataType>INT</DataType>
      </Entry>
    </RxPdo>
    <RxPdo Mandatory="true" Fixed="true" Sm="2">
      <Index>#x1603</Index>
      <Name>Additions</Name>
      <Entry>
        <Index>#x7030</Index>
        <SubIndex>1</SubIndex>
        <BitLen>16</BitLen>
        <Name>Out1</Name>
        <DataType>INT</DataType>
      </Entry>
      <Entry>
        <Index>#x7030</Index>
        <SubIndex>2</SubIndex>
        <BitLen>16</BitLen>
        <Name>Out2</Name>
        <DataType>INT</DataType>
      </Entry>
    </RxPdo>
    <TxPdo Mandatory="true" Fixed="true" Sm="3">
      <Index>#x1A01</Index>
      <Name>Input mapping 1</Name>
      <Entry>
        <Index>#x6010</Index>
```

Figure 6-47. RxPdo definition of #x1603.

6.8 TwinCAT in Run Mode

When one or more slave devices need to work in either *SM* or *DC-Synchronous* mode, and in any case when the communication must be tested under hard real-time conditions, *Free Run* is not suitable. In such cases, the slave must be configured to operate in either SM or DC Sync in predefined cycle times mode and TwiCAT must switch into **Run Mode** in order to corroborate this scheme.To implement this functionality:

1. Configure the slave device(s) to operate in DC Sync Mode as depicted in Figure 6-48

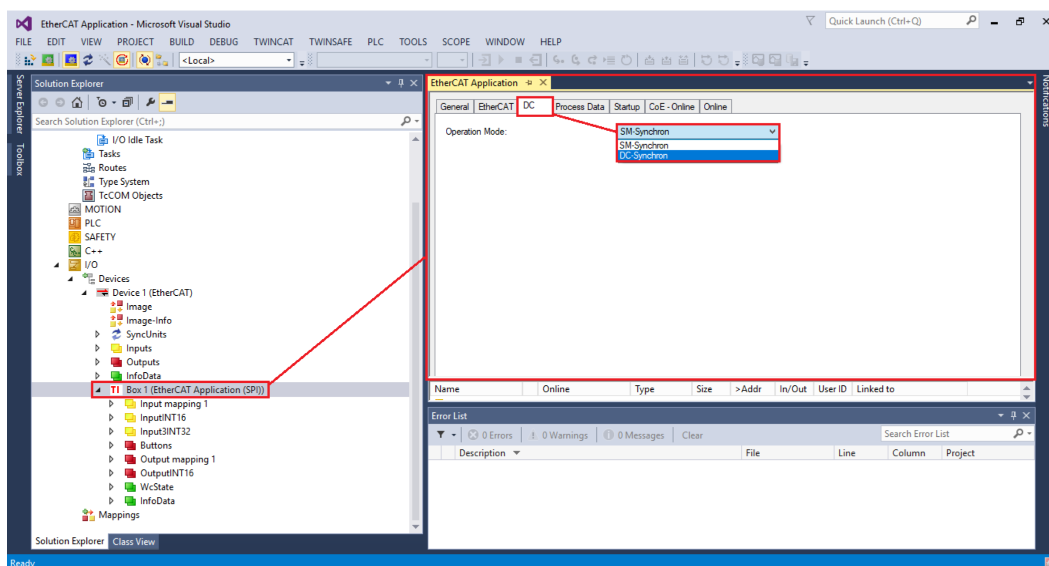


Figure 6-48. Slave device in DC Sync Mode.

2. Select the minimum allowed cycle time of EtherCAT cycle time by changing the *Base Time* of TwinCAT to the minimum allowed value of 50 microseconds as shown in Figure 6-49.

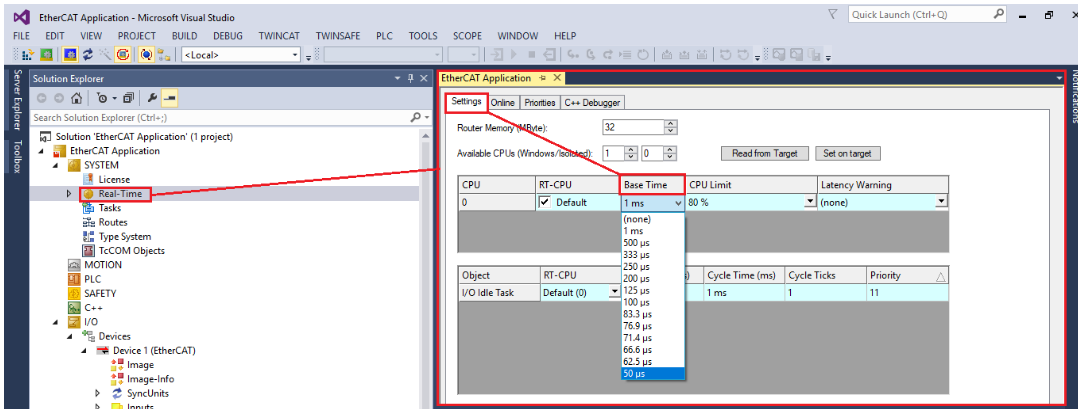


Figure 6-49. Select minimum EtherCAT cycle time.

3. Create a new *real-time I/O Task* and then define the Cycle Ticks (cycle time) of your application by specifying the integer multiple of TwinCAT's Base unit (50 micro seconds) as illustrated in Figure 6-50.

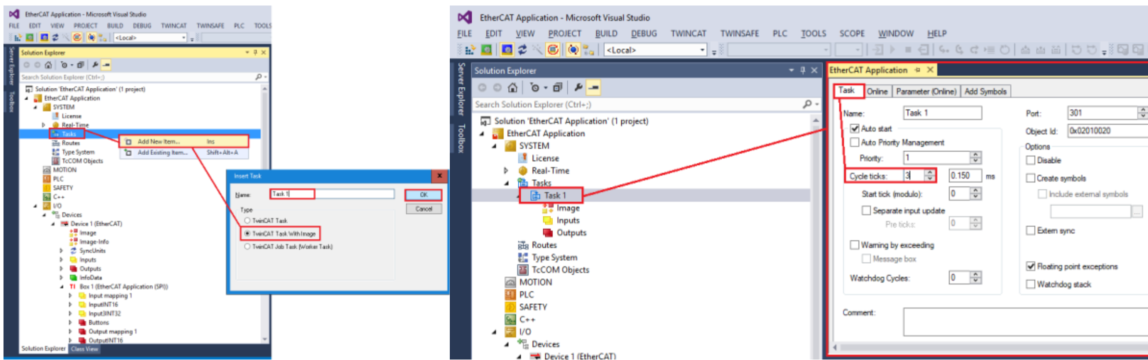


Figure 6-50. Create I/O Task with Image and Define cycle time.

4. Create one cyclic Input or Output variable (for the I/O Task) for every slave on the network that you want to operate in DC mode with a datatype matching the datatype of a process data variable in the EtherCAT network. In this case one (because there is only one slave in the network) output bollean variable was created to match one of the output boolean variables of the slave. Right click on the *Outputs* item and follow the steps described in Figure 6-51.

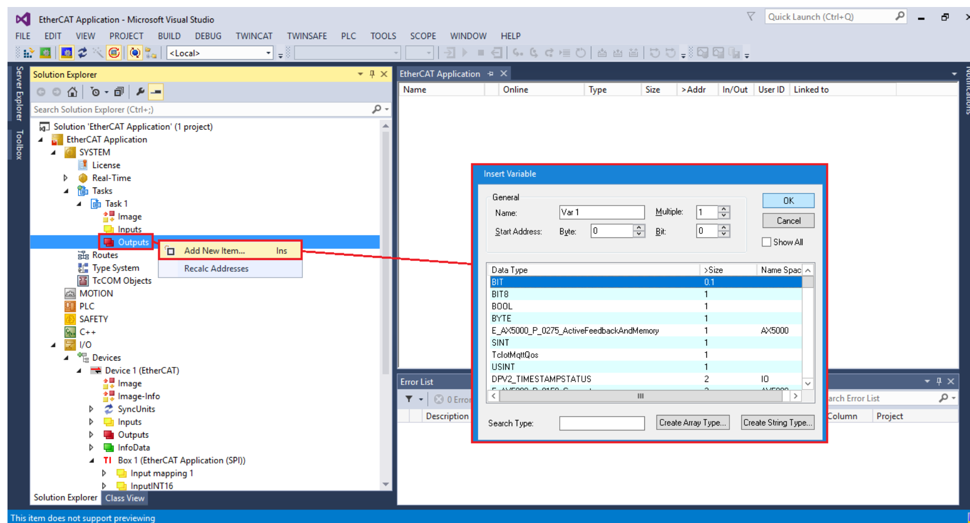


Figure 6-51. Create cyclic Output variable.

5. Create a link between the software variable in the I/O Task and the hardware variable in the EtherCAT process data by clicking on the created Output Task variable *Var 1* and navigating to the *Variable Tab*. Click *Linked to…* button and select any of the Buttons from the list (Figure 6-52).

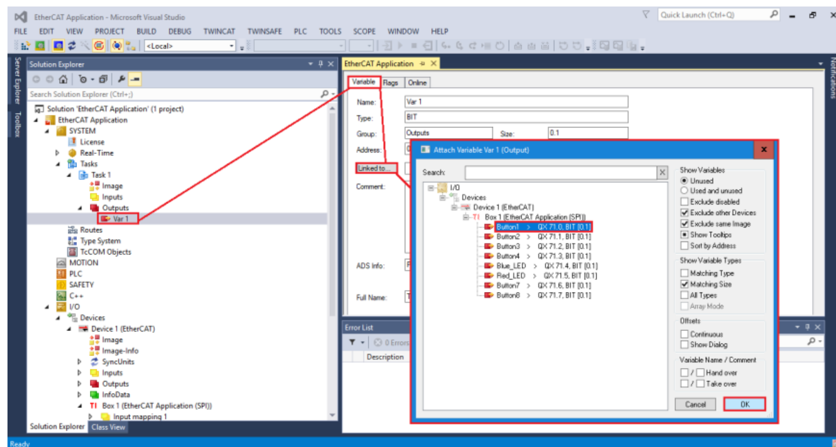


Figure 6-52. Link software to hardware variable.

6. *Activate Configuration* and start TwinCAT in Run Mode (Figure 2-49) as illustrated in Figure 6-53. The command "Activate Configuration" shall be applied every time one or more parameters are changed in the configuration and the changes need to be applied.

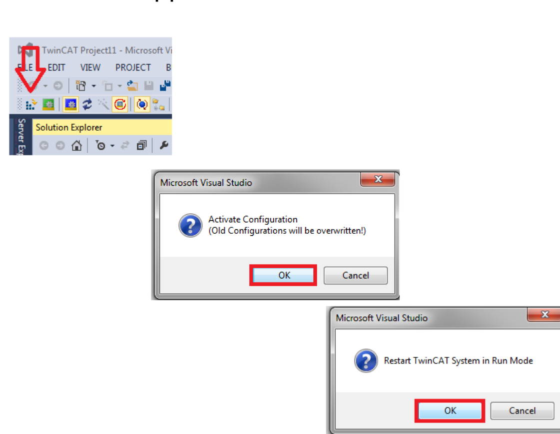


Figure 6-53. Activate Configuration and switch to Run Mode.

TwinCAT 3.1 provides a 7-day trial period for Run Mode, which can be extended for an arbitrary number of times. In order to extend the trial license for other 7-days, it will be sufficient to copy the 5-charater code which will show-up when trying to activate the configuration.

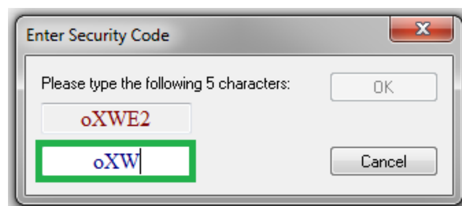


Figure 6-54. Enter Security Code.

The slave (or slaves) are now In Operational DC – Sync mode if no erros occur in the error list.

6.9 Add Watch Expression in CCS Debug

When being in Debug Mode, Code Composer Studio provides the ability to inspect only global variables in the Expresions window. In this way, developers may debug their stack without much effort. In order to add a variable in the Epressions Window and monitor its value through execution time:

1. Select *View* > *Expressions* if the *Expressions* tab window is not already visible in CCS Debug mode(Figure 6-55).

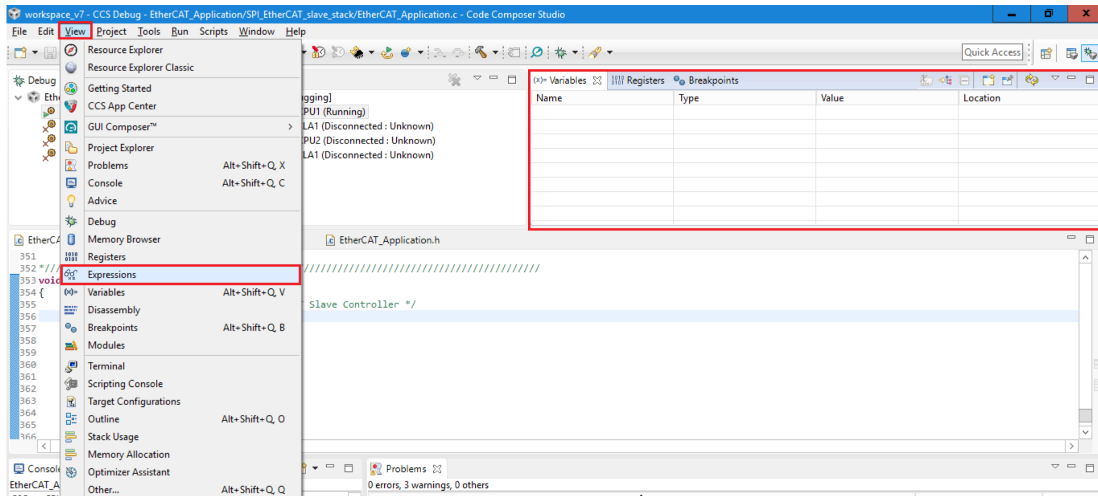


Figure 6-55. Add Expression Tab.

2. Enable the *Continuous Refresh* feature, click on *Add new expression* and type the name of the variable you desire (Figure 6-56).

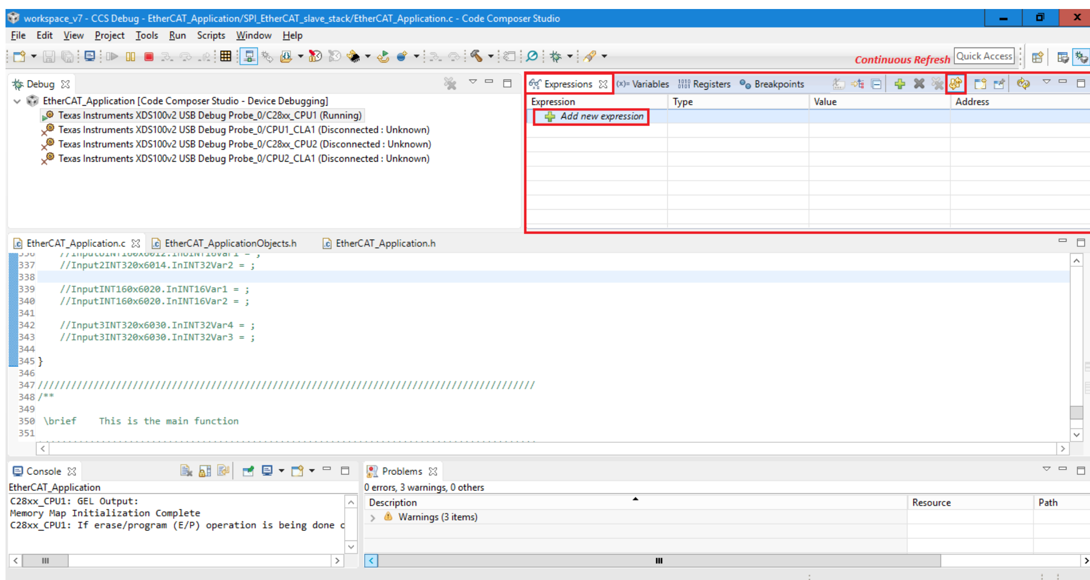


Figure 6-56. Add new expression.

3. CCS will provide a list of the variables with relevant names and you are free to choose from the list. In this example, we wanted to inspect the type and value of the global variable *timer_PDI_Isr_Output* (Figure 6-57).

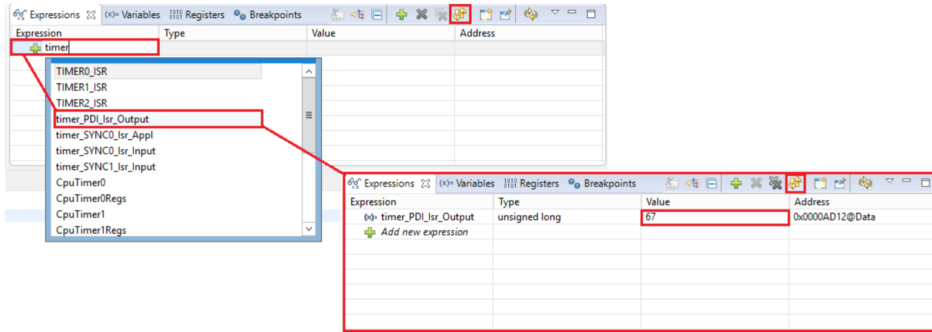


Figure 6-57. Select variable to inspect.

6.10 Laelaps II motors and gearheads

Laelaps II uses different combinations of motors and gearheads to drive its knee and hip joints, yet, in both cases, a pulley with a specific gear ration (48/26) is mounted to reduce the rotational speed of the motor even more and increase the output torque. All motors and gearheads are purchased from Maxon Motors [38] .

Hip motor – gearhead

For the hip joint of Laelaps II, *EC 45, ø45 mm, brushless motor, 250 Watt* [39] is used (Part No 136209) along with the *Planetary Gearhead GP 52 C ø52 mm, 4–30 Nm* [40] with a gear ratio of 343/8 (Part No 223089). Hence the total reduction ratio of the hip joint (gearhead and pulley combined) is 1029/13 ≈ 79,15.

Knee motor – gearhead

For the knee joint of Laelaps II, *RE 50, ø50 mm, Graphite Brushes motor, 200 Watt* [41] is used (Part No 370356) along with the *Planetary Gearhead GP 52 C ø52 mm, 4–30 Nm* [40] with a gear ratio of 637/12 (Part No 223090). Hence the total reduction ratio of the hip joint (gearhead and pulley combined) is 98.

6.11 Matlab PIV controller simulation

The block diagram for one actuated degree of freedom of Laelaps II leg (either hip or knee motor) is shown in Figure 6-58. Within our actual CCS project downloaded in all four LaunchXL-F28379D launchpads of the quadruped robot, the implemented PIV (Proportional – Integral - Velocity) controller is structured in such a way that the reference points (r_k) and measured feedback values (y_k) are normalized.

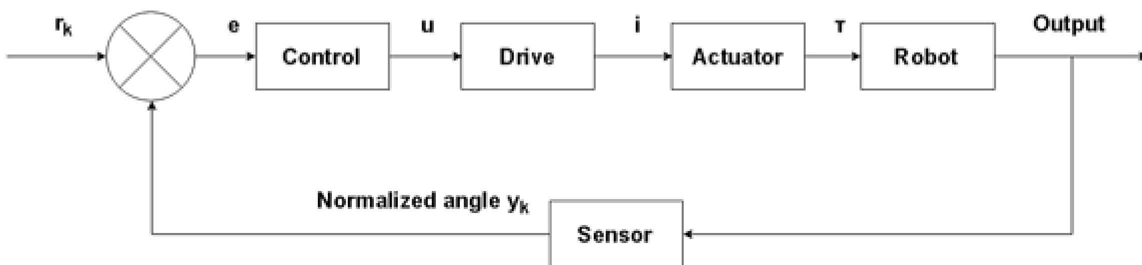


Figure 6-58. Block diagram of one actuated degree of freedom of Laelaps II leg.

A Matlab function has been assembled to match the aforementioned configuration, taking into account the mechanical limitations of the robot. The functionality of the simulating controller is identical to the one running in each leg of Laelaps II and can be located at PIV_controller.m file. A small initialization script is also

appended (ControlTesting.m file) to indicate all required global variables and specify the proper function call. The block diagram of the Matlab PIV controller is displayed in Figure 6-59.

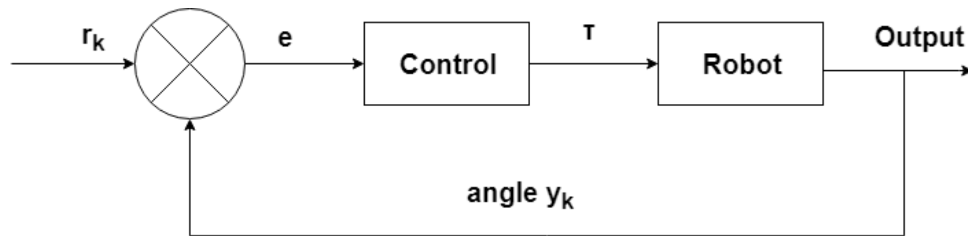


Figure 6-59. Block diagram of the Matlab PIV controller.

All motor driver boards (amplifiers) mounted on Laelaps II, have a maximum output current of 12 [A] which corresponds to 100% PWM signal. Following the data sheets of both types of motors (refer to Laelaps II motors and gearheads) and assuming negligible power losses, the knee motor (brushed) has a maximum torque output (after the gearhead) of 45 [Nm] and considering the pulley gear ratio, this torque can reach 83,077 [Nm]. On the other hand, the hip motor (brushless) has a maximum torque output (after the gearhead) of 28 [Nm] which can reach 51,692 [Nm] after the pulley transmition. This is the reason why these specific torque saturation values are exploited within the script. However, developers must also consider the fact that the couplers, mounted after the gearheads and before the pulley gear ration, have a torque limit of 30 [Nm] to avoid slipping which obviously affects the brushed motor.

In a nutshell, the PIV controller simulation function receives the reference and feedback values of both knee and hip motors and using the predefined control gains (K_p, K_v, K_i) calculates the torque outputs which will then be passed to the dynamic model of the robot. The PWM limits and torque saturation values are taken into account before calculating the final control signals each time the function is executed. All calculations are based on [11] .

7 Appendix B

7.1 Matlab Leg Modelling Code

```
clc
clear all
% ---------------------------------------------------------- %
% Forward & Inverse Kinematics
% Connecting leg angle to legs edge coordinates relative to Body Frame
% Knee Configuration (-1: forward, +1: backward)
% Left/Right leg indicator ( 1: left leg, -1: right leg)
% Ellipse semi axes: a_ellipse, b_ellipse
% ---------------------------------------------------------- %

% Trajectory Planning
% Parameters
xdes=zeros(10000,1);
ydes=zeros(10000,1);

x_traj_cntr=0.0;      % Parameter [m] (x centre of trajectory)
y_traj_cntr=0.590;    % Parameter [m] (y centre of trajectory)
a_ellipse=0.03;       % Parameter [m] (a amplitude of elliptical shape)
b_ellipse=0.05;       % Parameter [m] (b amplitude of elliptical shape)
traj_freq=1;          % Parameter [Hz] (frequency of elliptical motion)
phase=0;              % Parameter [rad] (initial phase)
param = 0;            % Parameter (flatness of the toe to model ground)

% % ---------------------------------------------------------- %
% % Move toes alogn elliptical (or semielliptical) trajectories
% % ---------------------------------------------------------- %

y_ellipse_cntr = y_traj_cntr;
x_ellipse_cntr = x_traj_cntr;
traj_vel = traj_freq*2*pi;
t=0;
iteration_number = 10000;
for l=1:iteration_number
t=t+0.01;
angle = traj_vel * t + phase*pi/180;
if mod(angle,2*pi)<pi
    b_ellipse_filtered = param * b_ellipse;
else
    b_ellipse_filtered = b_ellipse;
end
xdes(l) = x_ellipse_cntr + a_ellipse * cos(angle);
ydes(l) = y_ellipse_cntr + b_ellipse_filtered * sin(angle);
end
figure
plot(xdes,ydes)
    axis([-0.3 0.3 -0.1 0.7])
    ylabel('+ <-- y axis','fontsize',14)
    xlabel('x axis --> +','fontsize',14)
    title('Planning of Trajectory','fontsize',14)
    set(gca,'Ydir','reverse')
% Leg Parameters
l1 = 0.25;
l2 = 0.35;
```

```
knee_configuration = -1;
for l=1:iteration_number
x_value=xdes(l);
y_value=ydes(l);
% Inverse Kinematics
c_invk = (y_value^2 + x_value^2 - l1^2 - l2^2)/(2*l1*l2);
s_invk = knee_configuration*sqrt(1-c_invk^2);
k1_invk = l2 + l1 * c_invk;
k2_invk = l1 * s_invk;
knee_angle = - atan2(y_value,x_value) + atan2(k2_invk, k1_invk) + pi/2;
hip_angle = knee_angle - atan2(s_invk, c_invk);
angles_deg=[hip_angle*180/pi knee_angle*180/pi];

% Forward Kinematics
x1=l1*sin(hip_angle);
y1=l1*cos(hip_angle);
xE=l1*sin(hip_angle)+l2*sin(knee_angle);
yE=l1*cos(hip_angle)+l2*cos(knee_angle);
x(1:3)=[0 x1 xE];
y(1:3)=[0 y1 yE];

hold on
h1 = plot(x,y,'r-o')
      ylabel('+ <-- y axis','fontsize',14)
      xlabel('x axis --> +','fontsize',14)
      title('Configuration of leg','fontsize',14)
      set(gca,'Ydir','reverse')
pause(0.001)
delete(h1)
end
```

7.2 Matlab Post Process Code

```
clc
clear all
% ---------------------------------------------------------- %
% Post Processing Laelaps csv files
% Scanning the file and using Laelaps' characteristics
% we plotted the results of each experiment. This code can be used
% with minor alterations on the name of the scanned file and its format
% Note: tightfig.m file must also be copied to this folder
% ---------------------------------------------------------- %
clc
clear all
fileID = fopen('Laelaps Trajectory Planning - Trotting 11 -
Kp45.0Kd0.03Ki0.0Filter20.csv','r');
formatSpec = '%f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d
%f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f
%d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d %f %d';
sizeA = [72 Inf];
A = fscanf(fileID,formatSpec,sizeA);
fclose(fileID);

Hip_PWM_Limit = 41.17;
Knee_PWM_Limit = 38.25;

Hip_max_velocity = 75.83*2*pi/60; %rad/s in 60 V
Knee_max_velocity = 55.5*2*pi/60; %rad/s in 60 V

i_knee=(8*26)/(343*48);
```

```
i_hip=(12*26)/(637*48);

% Initializations
t=zeros(1,length(A));
% Hind Right Leg
HR_knee_angle_deg=zeros(1,length(A));
HR_hip_angle_deg=zeros(1,length(A));
HR_velocity_hip=zeros(1,length(A));
HR_velocity_knee=zeros(1,length(A));
HR_uk_hip=zeros(1,length(A));
HR_uk_knee=zeros(1,length(A));
HR_Desired_hip_angle=zeros(1,length(A));
HR_Desired_knee_angle=zeros(1,length(A));
HR_time=zeros(1,length(A));

% Hind Left Leg
HL_knee_angle_deg=zeros(1,length(A));
HL_hip_angle_deg=zeros(1,length(A));
HL_velocity_hip=zeros(1,length(A));
HL_velocity_knee=zeros(1,length(A));
HL_uk_hip=zeros(1,length(A));
HL_uk_knee=zeros(1,length(A));
HL_Desired_hip_angle=zeros(1,length(A));
HL_Desired_knee_angle=zeros(1,length(A));
HL_time=zeros(1,length(A));

% Fore Right Leg
FR_knee_angle_deg=zeros(1,length(A));
FR_hip_angle_deg=zeros(1,length(A));
FR_velocity_hip=zeros(1,length(A));
FR_velocity_knee=zeros(1,length(A));
FR_step_command_hip=zeros(1,length(A));
FR_step_command_knee=zeros(1,length(A));
FR_uk_hip=zeros(1,length(A));
FR_uk_knee=zeros(1,length(A));
FR_Desired_hip_angle=zeros(1,length(A));
FR_Desired_knee_angle=zeros(1,length(A));
FR_time=zeros(1,length(A));

% Fore Left Leg
FL_knee_angle_deg=zeros(1,length(A));
FL_hip_angle_deg=zeros(1,length(A));
FL_velocity_hip=zeros(1,length(A));
FL_velocity_knee=zeros(1,length(A));
FL_uk_hip=zeros(1,length(A));
FL_uk_knee=zeros(1,length(A));
FL_Desired_hip_angle=zeros(1,length(A));
FL_Desired_knee_angle=zeros(1,length(A));
FL_time=zeros(1,length(A));
j=1;
for i=1:length(A)
    t(i)=A(1,i)/1000;
    HR_hip_angle_deg(i)= -A(2,i)/100;
    HR_knee_angle_deg(i)= -A(4,i)/100;
    HR_Desired_hip_angle(i)=-A(6,i)/100;
    HR_Desired_knee_angle(i)=-A(8,i)/100;
    HR_uk_hip(i)=A(10,i)/100;
    HR_uk_knee(i)=A(12,i)/100;
    HR_velocity_hip(i)=-A(14,i)*i_hip/1000;
    HR_velocity_knee(i)=-A(16,i)*i_knee/1000;
```

```
    HR_time(i)=A(18,i)/100;

    HL_hip_angle_deg(i)= A(20,i)/100;
    HL_knee_angle_deg(i)= A(22,i)/100;
    HL_Desired_hip_angle(i)=A(24,i)/100;
    HL_Desired_knee_angle(i)=A(26,i)/100;
    HL_uk_hip(i)=A(28,i)/100;
    HL_uk_knee(i)=A(30,i)/100;
    HL_velocity_hip(i)=A(32,i)*i_hip/1000;
    HL_velocity_knee(i)=A(34,i)*i_knee/1000;
    HL_time(i)=A(36,i)/100;

    FR_hip_angle_deg(i)= -A(38,i)/100;
    FR_knee_angle_deg(i)= -A(40,i)/100;
    FR_Desired_hip_angle(i)=-A(42,i)/100;
    FR_Desired_knee_angle(i)=-A(44,i)/100;
    FR_uk_hip(i)=A(46,i)/100;
    FR_uk_knee(i)=A(48,i)/100;
    FR_velocity_hip(i)=-A(50,i)*i_hip/1000;
    FR_velocity_knee(i)=-A(52,i)*i_knee/1000;
    FR_time(i)=A(54,i)/100;

    FL_hip_angle_deg(i)=A(56,i)/100;
    FL_knee_angle_deg(i)=A(58,i)/100;
    FL_Desired_hip_angle(i)=A(60,i)/100;
    FL_Desired_knee_angle(i)=A(62,i)/100;
    FL_uk_hip(i)=A(64,i)/100;
    FL_uk_knee(i)=A(66,i)/100;
    FL_velocity_hip(i)=A(68,i)*i_hip/1000;
    FL_velocity_knee(i)=A(70,i)*i_knee/1000;
    FL_time(i)=A(72,i)/100;

    if (i>30000 && i<35000)
    [HR_x(j),
HR_y(j)]=ForwardKinematics(HR_hip_angle_deg(i),HR_knee_angle_deg(i));
    [HR_x_desired(j),
HR_y_desired(j)]=ForwardKinematics(HR_Desired_hip_angle(i),HR_Desired_knee_angle(
i));
    [HL_x(j),
HL_y(j)]=ForwardKinematics(HL_hip_angle_deg(i),HL_knee_angle_deg(i));
    [HL_x_desired(j),
HL_y_desired(j)]=ForwardKinematics(HL_Desired_hip_angle(i),HL_Desired_knee_angle(
i));
    [FR_x(j),
FR_y(j)]=ForwardKinematics(FR_hip_angle_deg(i),FR_knee_angle_deg(i));
    [FR_x_desired(j),
FR_y_desired(j)]=ForwardKinematics(FR_Desired_hip_angle(i),FR_Desired_knee_angle(
i));
    [FL_x(j),
FL_y(j)]=ForwardKinematics(FL_hip_angle_deg(i),FL_knee_angle_deg(i));
    [FL_x_desired(j),
FL_y_desired(j)]=ForwardKinematics(FL_Desired_hip_angle(i),FL_Desired_knee_angle(
i));
    j=j+1;
    end
end

%End Effector of Laelaps II Legs
figure
set(gcf, 'Position', [100 50 900 800],'color','w');
```

```
subplot(2,2,1)
plot(FR_x,FR_y,'k',FR_x_desired,FR_y_desired,'r')
     grid on
     ylabel('+ <-- y axis','fontsize',14)
     xlabel('x axis --> +','fontsize',14)
     title('FR End Effector in Steady State','fontsize',14)
     set(gca,'Ydir','reverse')
subplot(2,2,2)
plot(FL_x,FL_y,'k',FL_x_desired,FL_y_desired,'r')
     grid on
     ylabel('+ <-- y axis','fontsize',14)
     xlabel('x axis --> +','fontsize',14)
     title('FL End Effector in Steady State','fontsize',14)
     set(gca,'Ydir','reverse')
subplot(2,2,3)
plot(HR_x,HR_y,'k',HR_x_desired,HR_y_desired,'r')
     grid on
     ylabel('+ <-- y axis','fontsize',14)
     xlabel('x axis --> +','fontsize',14)
     title('HR End Effector in Steady State','fontsize',14)
     set(gca,'Ydir','reverse')
subplot(2,2,4)
plot(HL_x,HL_y,'k',HL_x_desired,HL_y_desired,'r')
     grid on
     ylabel('+ <-- y axis','fontsize',14)
     xlabel('x axis --> +','fontsize',14)
     title('HL End Effector in Steady State','fontsize',14)
     set(gca,'Ydir','reverse')
tightfig;

%Responce of knee angles
figure
set(gcf, 'Position', [100 50 900 800],'color','w');
subplot(4,1,1)
plot(t,FR_knee_angle_deg,'k',t,FR_Desired_knee_angle,'r')
     grid on
     ylabel('Angle [deg]')
     title('Response of FR Knee Angle','fontsize',13)
subplot(4,1,2)
plot(t,FL_knee_angle_deg,'k',t,FL_Desired_knee_angle,'r')
     grid on
     ylabel('Angle [deg]')
     title('Response of FL Knee Angle','fontsize',13)
subplot(4,1,3)
plot(t,HR_knee_angle_deg,'k',t,HR_Desired_knee_angle,'r')
     grid on
     ylabel('Angle [deg]')
     title('Response of HR Knee Angle','fontsize',13)
subplot(4,1,4)
plot(t,HL_knee_angle_deg,'k',t,HL_Desired_knee_angle,'r')
     grid on
     ylabel('Angle [deg]')
     xlabel('Time [s]')
     title('Response of HL Knee Angle','fontsize',13)
tightfig;

%Responce of hip angles
figure
set(gcf, 'Position', [100 50 900 800],'color','w');
subplot(4,1,1)
plot(t,FR_hip_angle_deg,'k',t,FR_Desired_hip_angle,'r')
```

```
    grid on
    ylabel('Angle [deg]')
    title('Response of FR Hip Angle','fontsize',13)
subplot(4,1,2)
plot(t,FL_hip_angle_deg,'k',t,FL_Desired_hip_angle,'r')
    grid on
    ylabel('Angle [deg]')
    title('Response of FL Hip Angle','fontsize',13)
subplot(4,1,3)
plot(t,HR_hip_angle_deg,'k',t,HR_Desired_hip_angle,'r')
    grid on
    ylabel('Angle [deg]')
    title('Response of HR Hip Angle','fontsize',13)
subplot(4,1,4)
plot(t,HL_hip_angle_deg,'k',t,HL_Desired_hip_angle,'r')
    grid on
    ylabel('Angle [deg]')
    xlabel('Time [s]')
    title('Response of HL Hip Angle','fontsize',13)
tightfig;

%PWM Commands of Knee motors
figure
set(gcf, 'Position', [100 50 900 800],'color','w');
subplot(4,1,1)
plot(t,FR_uk_knee,'k','LineWidth',0.1)
hold on
plot(t,Knee_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Knee_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
    grid on
    ylabel('PWM Command [%]')
    title('PWM Command of FR Knee','fontsize',13)
subplot(4,1,2)
plot(t,FL_uk_knee,'k','LineWidth',0.1)
hold on
plot(t,Knee_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Knee_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
    grid on
    ylabel('PWM Command [%]')
    title('PWM Command of FL Knee','fontsize',13)
subplot(4,1,3)
plot(t,HR_uk_knee,'k','LineWidth',0.1)
hold on
plot(t,Knee_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Knee_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
    grid on
    ylabel('PWM Command [%]')
    title('PWM Command of HR Knee','fontsize',13)
subplot(4,1,4)
plot(t,HL_uk_knee,'k','LineWidth',0.1)
hold on
plot(t,Knee_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Knee_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
    grid on
    ylabel('PWM Command [%]')
    xlabel('Time [s]')
    title('PWM Command of HL Knee','fontsize',13)
tightfig;

%PWM Commands of Hip motors
figure
```

```
set(gcf, 'Position', [100 50 900 800],'color','w');
subplot(4,1,1)
plot(t,FR_uk_hip,'k','LineWidth',0.1)
hold on
plot(t,Hip_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Hip_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
     grid on
     ylabel('PWM Command [%]')
     title('PWM Command of FR Hip','fontsize',13)
subplot(4,1,2)
plot(t,FL_uk_hip,'k','LineWidth',0.1)
hold on
plot(t,Hip_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Hip_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
     grid on
     ylabel('PWM Command [%]')
     title('PWM Command of FL Hip','fontsize',13)
subplot(4,1,3)
plot(t,HR_uk_hip,'k','LineWidth',0.1)
hold on
plot(t,Hip_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Hip_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
     grid on
     ylabel('PWM Command [%]')
     title('PWM Command of HR Hip','fontsize',13)
subplot(4,1,4)
plot(t,HL_uk_hip,'k','LineWidth',0.1)
hold on
plot(t,Hip_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Hip_PWM_Limit*ones(length(t),1),'r','LineWidth',0.1);
     grid on
     ylabel('PWM Command [%]')
     xlabel('Time [s]')
     title('PWM Command of HL Hip','fontsize',13)
tightfig;

%Velocity of Knee motors
figure
set(gcf, 'Position', [100 50 900 800],'color','w');
subplot(4,1,1)
hold on
plot(t,Knee_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Knee_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,FR_velocity_knee,'k')
     grid on
     ylabel('Velocity [rad/s]')
     title('Response of FR Knee Velocity','fontsize',13)
subplot(4,1,2)
hold on
plot(t,Knee_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Knee_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,FL_velocity_knee,'k')
     grid on
     ylabel('Velocity [rad/s]')
     title('Response of FL Knee Velocity','fontsize',13)
subplot(4,1,3)
hold on
plot(t,Knee_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Knee_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,HR_velocity_knee,'k')
     grid on
```

```
     ylabel('Velocity [rad/s]')
     title('Response of HR Knee Velocity','fontsize',13)
subplot(4,1,4)
hold on
plot(t,Knee_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Knee_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,HL_velocity_knee,'k')
     grid on
     ylabel('Velocity [rad/s]')
     xlabel('Time [s]')
     title('Response of HL Knee Velocity','fontsize',13)
tightfig;

%Velocity of Hip motors
figure
set(gcf, 'Position', [100 50 900 800],'color','w');
subplot(4,1,1)
hold on
plot(t,Hip_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Hip_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,FR_velocity_hip,'k')
     grid on
     ylabel('Velocity [rad/s]')
     title('Response of FR Hip Velocity','fontsize',13)
subplot(4,1,2)
hold on
plot(t,Hip_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Hip_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,FL_velocity_hip,'k')
     grid on
     ylabel('Velocity [rad/s]')
     title('Response of FL Hip Velocity','fontsize',13)
subplot(4,1,3)
hold on
plot(t,Hip_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Hip_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,HR_velocity_hip,'k')
     grid on
     ylabel('Velocity [rad/s]')
     title('Response of HR Hip Velocity','fontsize',13)
subplot(4,1,4)
hold on
plot(t,Hip_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,-Hip_max_velocity*ones(length(t),1),'r','LineWidth',0.1);
plot(t,HL_velocity_hip,'k')
     grid on
     ylabel('Velocity [rad/s]')
     xlabel('Time [s]')
     title('Response of HL Hip Velocity','fontsize',13)
tightfig;
```

tightfig.m file

```
function hfig = tightfig(hfig)
% tightfig: Alters a figure so that it has the minimum size necessary to
% enclose all axes in the figure without excess space around them.
% Note that tightfig will expand the figure to completely encompass all
% axes if necessary. If any 3D axes are present which have been zoomed,
% tightfig will produce an error, as these cannot easily be dealt with.
% hfig - handle to figure, if not supplied, the current figure will be used
% instead.
```

```
    if nargin == 0
        hfig = gcf;
    end
    % There can be an issue with tightfig when the user has been modifying
    % the contnts manually, the code below is an attempt to resolve this,
    % but it has not yet been satisfactorily fixed
%     origwindowstyle = get(hfig, 'WindowStyle');
    set(hfig, 'WindowStyle', 'normal');
    % 1 point is 0.3528 mm for future use
    % get all the axes handles note this will also fetch legends and
    % colorbars as well
    hax = findall(hfig, 'type', 'axes');
    % get the original axes units, so we can change and reset these again
    % later
    origaxunits = get(hax, 'Units');
    % change the axes units to cm
    set(hax, 'Units', 'centimeters');
    % get various position parameters of the axes
    if numel(hax) > 1
%         fsize = cell2mat(get(hax, 'FontSize'));
        ti = cell2mat(get(hax,'TightInset'));
        pos = cell2mat(get(hax, 'Position'));
    else
%         fsize = get(hax, 'FontSize');
        ti = get(hax,'TightInset');
        pos = get(hax, 'Position');
    end
    % ensure very tiny border so outer box always appears
    ti(ti < 0.1) = 0.15;
    % we will check if any 3d axes are zoomed, to do this we will check if
    % they are not being viewed in any of the 2d directions
    views2d = [0,90; 0,0; 90,0];
    for i = 1:numel(hax)
        set(hax(i), 'LooseInset', ti(i,:));
%         set(hax(i), 'LooseInset', [0,0,0,0]);
        % get the current viewing angle of the axes
        [az,el] = view(hax(i));
        % determine if the axes are zoomed
        iszoomed = strcmp(get(hax(i), 'CameraViewAngleMode'), 'manual');
        % test if we are viewing in 2d mode or a 3d view
        is2d = all(bsxfun(@eq, [az,el], views2d), 2);

        if iszoomed && ~any(is2d)
           error('TIGHTFIG:haszoomed3d', 'Cannot make figures containing zoomed
3D axes tight.')
        end

    end
    % we will move all the axes down and to the left by the amount
    % necessary to just show the bottom and leftmost axes and labels etc.
    moveleft = min(pos(:,1) - ti(:,1));
    movedown = min(pos(:,2) - ti(:,2));
    % we will also alter the height and width of the figure to just
    % encompass the topmost and rightmost axes and lables
    figwidth = max(pos(:,1) + pos(:,3) + ti(:,3) - moveleft);
    figheight = max(pos(:,2) + pos(:,4) + ti(:,4) - movedown);
    % move all the axes
    for i = 1:numel(hax)
        set(hax(i), 'Position', [pos(i,1:2) - [moveleft,movedown], pos(i,3:4)]);

    end
```

```
    origfigunits = get(hfig, 'Units');
    set(hfig, 'Units', 'centimeters');
    % change the size of the figure
    figpos = get(hfig, 'Position');
    set(hfig, 'Position', [figpos(1), figpos(2), figwidth, figheight]);
    % change the size of the paper
    set(hfig, 'PaperUnits','centimeters');
    set(hfig, 'PaperSize', [figwidth, figheight]);
    set(hfig, 'PaperPositionMode', 'manual');
    set(hfig, 'PaperPosition',[0 0 figwidth figheight]);
    % reset to original units for axes and figure
    if ~iscell(origaxunits)
        origaxunits = {origaxunits};
    end
    for i = 1:numel(hax)
        set(hax(i), 'Units', origaxunits{i});
    end
    set(hfig, 'Units', origfigunits);
%       set(hfig, 'WindowStyle', origwindowstyle);
end
```

ForwardKinematics.m

```
function [ x,y ] = ForwardKinematics( hip_angle_deg,knee_angle_deg )
%UNTITLED Forward Kinematics of Laelaps II
%   x (+) right, y (+) down
% Forward Kinematics
% Leg Parameters
l1 = 0.25;
l2 = 0.35;
hip_angle_rad=hip_angle_deg*pi/180;
knee_angle_rad=knee_angle_deg*pi/180;
x=l1*sin(hip_angle_rad)+l2*sin(knee_angle_rad);
y=l1*cos(hip_angle_rad)+l2*cos(knee_angle_rad);
end
```

7.3 Matlab PIV Controller Simulation

ControlTesting.m file

```
%-------------------------------------------------------------------------%
% Global variables
%-------------------------------------------------------------------------%
clear global; clear all; clc;
global d2_knee d3_knee i10_knee i14_knee
global d2_hip d3_hip i10_hip i14_hip
global c1 c2
global hip_control_torque_sat knee_control_torque_sat
global Umax_knee Umin_knee Umax_hip Umin_hip

d2_knee = 0;
d3_knee = 0;
i10_knee = 0;
i14_knee = 0;
d2_hip = 0;
d3_hip = 0;
i10_hip = 0;
i14_hip = 0;
T=1/10000;
FilterBandwidth = 20;
```

```
ftc = 1/(2*pi*FilterBandwidth);
c1 = 2/(T+2*ftc);
c2 = (T-2*ftc)/(T+2*ftc);
hip_control_torque_sat = 83.077;
knee_control_torque_sat = 51.692;
Umax_knee=0.3825;
Umin_knee=-0.3825;
Umax_hip=0.4117;
Umin_hip=-0.4117;

[ knee_cntrl_torque, hip_cntrl_torque ] = PIV_controller( 0,pi/6,1,0,-pi/8,1 );
```

PIV_controller.m file

```
function [ knee_cntrl_torque, hip_cntrl_torque ] = PIV_controller(
th_knee_rad,th_knee_des_rad,lk_knee,th_hip_rad,th_hip_des_rad,lk_hip )
%PIV_controller function for simulations
%   This functions implements the PIV controller used in
%Laelaps experiments

global d2_knee d3_knee i10_knee i14_knee
global d2_hip d3_hip i10_hip i14_hip
global c1 c2
global Umax_knee Umin_knee Umax_hip Umin_hip
global hip_control_torque_sat knee_control_torque_sat

% Knee Control Gains
Kr_knee = 1;
Kp_knee = 40;
Kd_knee = 0.01;
Ki_knee = 0;

% Hip Control Gains
Kr_hip = 1;
Kp_hip = 40;
Kd_hip = 0.01;
Ki_hip = 0;

% Normalized Values to match with real Controller
th_knee_des = th_knee_des_rad/(2*pi);
th_knee = th_knee_rad/(2*pi);
th_hip_des = th_hip_des_rad/(2*pi);
th_hip = th_hip_rad/(2*pi);

% PIV Controller for Knee
v5_knee = Kr_knee * th_knee_des - th_knee;
v8_knee = Ki_knee * Kp_knee * i14_knee * (th_knee_des - th_knee) + i10_knee;
i10_knee = v8_knee;
v1_knee = Kd_knee * c1 * th_knee;
v4_knee = v1_knee - d2_knee - d3_knee;
d2_knee = v1_knee;
d3_knee = v4_knee * c2;
v9_knee = Kp_knee * (v5_knee - v4_knee) + v8_knee;
if (v9_knee > Umax_knee)
    v10_knee = Umax_knee;
elseif (v9_knee < Umin_knee)
    v10_knee = Umin_knee;
else
    v10_knee = v9_knee;
end
```

```
if (v10_knee == v9_knee)
    v12_knee = 1;
else
    v12_knee = 0;
end
i14_knee = v12_knee * lk_knee;
knee_cntrl_torque = v10_knee * knee_control_torque_sat;

% PIV Controller for Hip
v5_hip = Kr_hip * th_hip_des - th_hip;
v8_hip = Ki_hip * Kp_hip * i14_hip * (th_hip_des - th_hip) + i10_hip;
i10_hip = v8_hip;
v1_hip = Kd_hip * c1 * th_hip;
v4_hip = v1_hip - d2_hip - d3_hip;
d2_hip = v1_hip;
d3_hip = v4_hip * c2;
v9_hip = Kp_hip * (v5_hip - v4_hip) + v8_hip;
if (v9_hip > Umax_hip)
    v10_hip = Umax_hip;
elseif (v9_hip < Umin_hip)
    v10_hip = Umin_hip;
else
    v10_hip = v9_hip;
end
if (v10_hip == v9_hip)
    v12_hip = 1;
else
    v12_hip = 0;
end
i14_hip = v12_hip * lk_hip;
hip_cntrl_torque = v10_hip * hip_control_torque_sat;
end
```