SOFTWARE FRAMEWORK FOR HIGH PRECISION MOTION CONTROL APPLICATIONS

by

Ahmet Teoman Naskali

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Sabancı University

August 2012

SOFTWARE FRAMEWORK FOR HIGH PRECISION MOTION CONTROL APPLICATIONS

APPROVED BY:

Prof. Dr. Asif Sabanovic, (Dissertation Supervisor)

[illegible] ....

Assoc. Prof. Ayhan Bozkurt

[illegible] ....

Assoc. Prof. Kemalettin Erbatur

[illegible] ....

Prof. Dr. Bernard Levrat

[illegible] ....

Assist. Prof. Gönen Eren

[illegible] ....

DATE OF APPROVAL: 07.08.2012

Abstract

Developing a motion control system requires much effort in different domains. Namely control, electronics and software engineering. In addition to these, there are the system requirements which may be completely different to these spanning from biomedical engineering to psychology. Collaboration between these fields is vital, however these fields should be involved only as much as they are needed to be in the fields of expertise of the others.

Several software frameworks exist for the creation of robotics applications. But currently there is no standard for the creation of mechatronics systems nor is there a complete software package that can deal with all aspects in the programming of such systems. Existing frameworks each have their advantages and disadvantages, however they generally have limited or no dedicated structure for the development of the motion control aspect of the problem and deal extensively with the robot-environment interactions and inter mechanism communications. Dealing with the higher levels of the problem, they are usually not well suited for hard realtime; since the interactions can run on soft realtime constraints. The software framework proposed in this study aims to achieve a level of abstraction between the different domains utilized within a system. The aim in using the framework is to achieve a sustainable software structure for the system. Sustainability is an important part of systems, as it permits a system to evolve with changing requirements and variable hardware, with the ultimate goal of having robust software that can be utilized on different platforms and with other systems using an abstraction layer between the hardware and the software. This ensures that the system can be migrated from a processing platform to any other platform and also from one set of hardware to another.

The framework was tested on several systems that have precision motion control requirements such as a 10 degree of freedom micro assembly workstation, a modular micro factory and a haptic system with time delay. Each of the systems works in different processing platforms and have different motion control requirements. The achieved results from the implementations show that the software framework is an important tool for the development of motion control software.

Özet

Bir hareket kontrol sistemi geliştirmek, denetim, elektronik ve yazılım gibi bir çok farklı alanda çalışma ve uğraşmayı gerektirir. Bunlara ek olarak, farklı sistemlerin, biyomedikal mühendisliğinden psikolojiye kadar birçok farklı sistem gereksinimleri mevcuttur. Her ne kadar, bu alanlar diğerlerinin uzmanlık alanlarının ihtiyacı doğrultusunda kullanılacak olsa da, aralarındaki işbirliği elzemdir. Robot teknolojisi uygulamaları yaratabilen birçok yazılım altyapısı bulunmaktadır. Ancak, halen mekatronik sistemler yaratmada belli standartlar ve bu tip sistemlerin programlanmasında tüm bu durumların üstesinden gelebilecek eksiksiz bir yazılım paketi yoktur. Var olan altyapılar bazı avantajlarının yanında birçok da dezavantaja sahiptir. Bu altyapılar problemin hareket kontrol sürecinin gelişiminde genellikle sınırlı kalmakta veya mevcut probleme özelleşmiş olarak çalışmamaktadır, bunun yanında, büyük ölçüde robot-çevre etkileşimleri ve mekanizmanın iç iletişimi ile uğraşmaktadır. Bu altyapılar problemin üst seviyeleriyle uğraşmaktadır ancak üst seviyeler gerçek zamanlı kısıtlara ihtiyaç duymayabilir. Bu noktadan hareketle, bu altyapılar gerçek zamanlı uygulamalar için tasarlanmamıştır.

Bu çalışmada ise, önerilen yazılım altyapısı, bir sistem içinde kullanılan farklı uzmanlık alanlarının birbirinden soyutlanarak bağımsız bir şekilde tasarlanmasını hedeflemektedir. Bu altyapının kullanılmasındaki amaç herhangi bir sistem için özelleşmiş, sürdürülebilir bir yazılım altyapısı oluşturmaktır. Sürdürülebilirlik bir sisteme, farklı platformlarda ve donanım ve yazılım arasında soyutlama katmanı kullanan diğer sistemlerle birlikte kullanılabilen güçlü bir yazılım elde etmek amacıyla değişen gereksinimler ve farklı donanımlar durumunda gelişebilme olanağı tanır. Bu nedenle, sistemler için elzem bir unsurdur. Bu sayede, sistem işlemekte olduğu platformdan başka bir platforma ve buna ek olarak bir donanım setinden diğerine taşınabilir. Altyapı, 10 serbestlik derecesine sahip mikro montaj iş istasyonu, modüler mikro fabrika ve gecikmeli dokunma duyusu aktaran sistem gibi hassas hareket kontrol gerektiren sistemler üzerinde denenmiştir. Bahsedilen her bir sistem farklı platformlarda ve farklı hareket kontrol gereksinimleriyle çalışmaktadır. Uygulamalardan elde edilen sonuçlar, önerilen yazılım altyapısının, hareket kontrol yazılımları içinde önemli bir araç olduğunu göstermektedir.

Acknowledgements

I offer my sincere gratitudes to my advisor, Asif Sabanovic, for trusting me from the beginning and for giving his support and guidance all along this thesis.

I wish to thank the members of my PhD committee; Ayhan Bozkurt, Kemalettin Erbatur , Bernard Levrat and Gönen Eren for their interest in my work.

I would like to thank Emrah Deniz Kunt for being a great colleague with whom I have bounced many ideas back and forth. You have been a great collaborator and a friend and I can never repay you. I would also like to thank Kazim Çakir who is one of the best engineers and most methodical people I know, for his constructive critiques during the different paths this work has taken.

I would like to thank Yeşim Kop for giving me full support and tolerating long sleepless nights in front of the computer.

I would like to thank all my colleagues from Microsystems Laboratory and Mechatronics department, especially; Anas Abidi, Merve Acer, Utku Seven, Ahmet Can Erdoğan, İlker Sevgen, Mehmet Guler, Zeynep Temel, Kaan Can Fidan, Edin Goluboviç and Eray Baran.

Finally I would like to thank my parents Emine and Esko and my brother Osman Naskali for supporting me during my work.

This thesis was partially supported by:

- Yousef Jameel Foundation
- SAN-TEZ Project Number 00183.STZ.2007-2
- T.R. Prime Ministry SPO Grant Number 2003k120530

TABLE OF CONTENTS

List of Tables

List of Figures

Table 0.1: Common Symbols

Symbol	**Explanation**
mm	millimeter
μm	microns
nm	nanometers
T_s	sampling time
V	volt
mV	millivolt
lbs	pounds
W	watt
Hz	hertz
kHz	kilohertz

Abbreviation	**Explanation**
RT	Realtime
DOF	Degree Of Freedom
PC	Personal Computer
IO	Input/Output
GUI	Graphical User Interface
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Arrays
MEMS	Micro-Electro-Mechanical Systems
DC	Direct Current
AC	Alternating Current
MIT	Massachussetts Institute of Technology
ERSP	Evolution Robotics Platform
MSRDS	Microsoft Robotics Developer Studio
.NET	Network
CCR	Concurrency and Coordination Runtime
OROCOS	Open Robot Control Software
TCP	Transmission Control Protocol
ROS	Robot Operating System
OS	Operating System
PID	Proportional-Integral-Derivate
CPU	Central Processing Unit
ATM	Automatic Teller Machine
VoIP	Voice over Internet Protocol
API	Application Programming Interface
MB	Megabyte
RAM	Random Access Memory
NAS	Network Attached Storage
EWF	Enhanced Write Filter
FBWF	File Based Write Filter
USB	Universal Serial Bus
CD-ROM	Compact Disk-Read Only Memory
MIPS	Microprocessor without Interlocked Pipeline Stages
ARM	Advanced RISC Machine
HDL	Hardware Description Language
ASIC	Application Specific Integrated Circuit
CLB	Configurable Logic Block

LUT	Look-Up Table
RT	Real Time
RTAI	Real Time Application Interface
UP	Uniprocessor
SMP	Symmetric Multi Processor
MUP	Multi Uniprocessor
RTOS	Real Time Operating System
GPGPU	General Purpose computing on Graphical Processing Units
FIFO	First In First Out
RMS	Rate Monotonic Scheduling
RR	Round Robin
EDF	Early Deadline First
MMI	Man-Machine Interface
RTT	Real-Time Toolkit
NSF	National Science Foundation
JTECH	Japanese Technology
PZT	Piezoelectric
MRI	Magnetic Resonance Imaging
EGR	Exhaust Gas Recirculation
VVT	Variable Valve Timing
CAM	Computer Aided Manufacturing
ID	Identification
SUMAW	Sabanci University Micro Assembly Workstation
LED	Light Emitting Diode
AFM	Atomic Force Microscope
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
RGB	Red Green Blue
SMC	System Mode Controller
PMMA	Polymethylmetacrilate
MEX	Matlab Executable

1 INTRODUCTION

As electro mechanical systems become more and more significant parts of our lives, their development also gains importance. A major part of the development of electro mechanical systems that includes moving parts is the development of motion control systems. For the fast development of motion control systems, frameworks are needed for components to be standardized and the system to be rapidly built.

Motion control is becoming increasingly complex with attempts to control system-to-environment, system-to-system and system-to-human interactions. In motion control, interactions are reflected as real or virtual forces and behavior of motion control systems must be changed due to the interaction forces. In general unstructured environments are treated in such a way that when and how interaction will appear is not known in advance, therefore motion control systems must be equipped with reflex changes due to interaction. That requires the reconfigurability of the control structure; dynamic - due to change of coordinates or static - done by designer during the design process, or on flight - by operator while system is in operation or in an emergency - due to failure of one or more components and loss of system capability because of failure.

There are several means of implementing motion control, like analog systems, however they are not very configurable and have difficulties accepting complicated references. More conveniently, motion control systems are implemented on micro controllers with external hardware attached to read positions and send analog signals. Digital signal processors (DSP's) are also widely used as their instruction set and configuration enables them to process information and do matrix multiplications needed for motion control much faster. These devices can implement interfaces for networks or have various communications protocols on them, or they may implement their own interface on them for simpler tasks. Another alternative for motion control is to use Industrial PCs equipped with IO cards capable of interfacing to the hardware that is to be controlled. These systems are easy to develop as they have an operating system that is easier to develop applications due to the availability of the software and ability to interface and design Graphical User Interfaces (GUIs) or networks. Operating systems used for motion control can achieve real-time performance. However with these systems unless there is a micro controller or a DSP for each degree of freedom, there occurs performance loss because each degree of freedom is processed in a synchronous manner inside the controller, therefore more degrees of freedom take up more time.

1.1 Motivation

Handling of hazardous materials, micro technology, space robotics, telesurgery; all necessitate precise motion control. The conception of these systems currently is a major challenge and the software to govern these systems poses an equally large problem. Currently there is no standardization for the creation of motion control software and usually projects are rewritten from scratch all the time. There is also no module structure for the creation of the software that inhibits the reuse of the software, making it cumbersome to sustain. These problems with motion control systems lead to the necessity of a motion control framework that ensures the generation of structured and reusable software.

1.2 Contribution

In this thesis we propose a new framework, that will enable the rapid development of motion control systems on multiple platforms. The proposed framework is platform and hardware independent. A modular structure forcing the creation of reusable modules brings on high sustainability as adopted by the framework. Current frameworks that were built with similar efforts focus on the robots or mechanisms interaction and communication problems and rely on somewhat off the shelf hardware to control the robots in realtime. The proposed framework is also complementary to those in the sense that it handles the actual creation of the motion control of the robot and then provides a motion control platform to the behavioral frameworks.

1.3 Thesis Structure

The thesis is organized as follows:

Chapter 2 gives a literature survey on the state of the art in motion control and delayed systems, finally motion control frameworks are discussed.

Chapter 3 first presents the theoretical background on realtime systems, then application of delayed control is discussed on bilateral systems and finally motion control system design is discussed.

Chapter 4 describes the Framework for motion control; the different components of the system are introduced and the usage of the framework is described.

Chapter 5 Validates the framework with implementations on a micro assembly workstation, a micro factory and a haptic system.

Chapter 6 Concludes the thesis, presents the contributions and describes future work.

2 LITERATURE SURVEY

This chapter presents a literature survey on methods that have been proposed to design motion control systems. In the following paragraphs, we first briefly recall some key techniques in construction of motion control systems and the control algorithms and methods that are used. Then a definition of frameworks is given and their advantages are indicated and different frameworks are examined.

2.1 Motion Control Systems

Precision engineering has been developing over the last decades in terms of research and application to meet requirements as higher performance, higher reliability, miniaturization, longer life and lower cost [1, 2]. All precision systems are built in nanometer scale by means of miniaturization and integration of electrical and mechanical components [2]. Development of motion control improved micro-electro-mechanical systems (MEMS) and provided more modern applications. New technological requirements as high speed and high-precision motion control are provided by the use of Direct Current (DC) permanent magnet linear motors (PMLMs) whose main advantages are high force density, low thermal losses, high accuracy and simplicity in mechanical structure [2, 3].

Regarding to its historical evolution, motion of mechanical systems used to be obtained from electric drives until 20th century. They have been widely used in industry up to relatively recent advances in computer technology [4]. More expert and precise motions have been performing through mechatronic applications and robotics. Ohnishi et al. refers to advances in parameter estimation, flux identification, speed estimation and design of the motion controller. They affirm that highly robust motion systems provide adjustable stiffness hence versatile applications. Alternating Current (AC) motors produced according to modern control techniques require the identification of motor parameters [4]. On the other side, in order to acquire more skillful and versatile motion, the mechanical system has to be equipped with high-performance controller thereby it should be robust against the load change and parameter variation. The structure of conventional minor control loop effective in one degree of freedom is shown in the Figure 2.1.

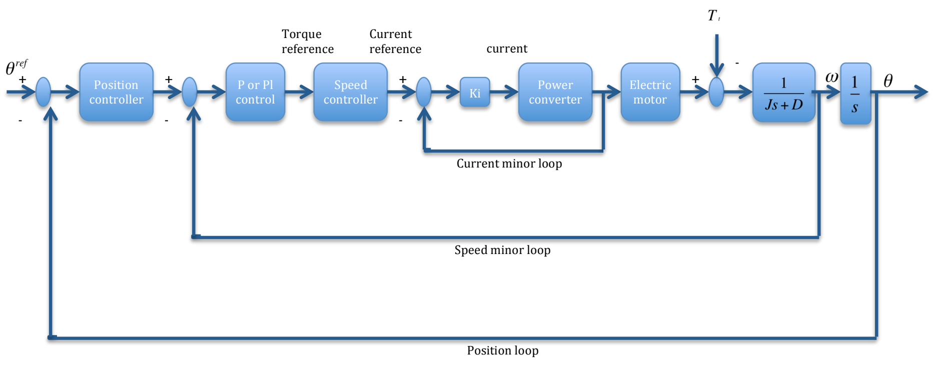


Figure 2.1: Structure of conventional minor control loop

As mentioned in [5, 6, 7, 8, 9], the identification of disturbance torque is crucial for motion control robustness. The kinematics and the dynamics or in other words the physical world is directly affected by the control.

On the other hand, Whitney affirms that the robot force control requires the combination of task goals, trajectory generation, force and position feedback and modification of the trajectories. The first robot force control appeared in 1950s and 1960s with remote manipulator and artificial arm control, so it was accomplished by natural ways. The first computer controls of robot force were realized only in late 1960s and 1970s [10].

In 1960s, when Mann invented the force feedback powered artificial elbow, the motor was driven by signals from muscle electrodes and a strain gauge in the joint [10, 11]. Soon after, it is realized that the use of force-feedback manipulators in space is inefficient and non-effective because of delays caused by coding, decoding, error checking, channel sharing, etc.

In early 1970s, in order to eliminate these delays, scientists began to work on the replacement of the human operator with a computer [10]. To structure multi-axis arm systems and to relate the requirements of a task to the motions required, Ernst [12], Barber [13], Hill [14] and Paul [15] worked on logic branching which is a scalar method consisting of strings of statements. Later, Nevins [16], Whitney [17] and Groome [18] worked on continuous force control provided by 6 axis sensors collecting multi-axis force-torque information. In pursuit of these advances, many scientists have worked on many other methods: Damping methods [15, 17], Position Methods [19, 20, 21, 22], Impedance or Energy Methods [23, 24], Explicit Force Control [16], Implicit Force Control [25], Hybrid Force-Position Control [26, 27].

In the last few decades, ultra precision manufacturing is developing. In this context, the use of PMLMs in different semiconductor processes satisfies the new technological requirements of miniature system assemblies and precision metrology. In todays technology, linear motors reduce the effects of contact-type nonlinearities and disturbances like backlash. Yet, it is also important to reduce or eliminate the model uncertainties and external disturbances for providing high speed and high precision. For instance, todays laser interferometers have measurement resolution down to one nanometer [2]. According to Heydemann, to increase the measurement resolution in sub-micrometers scales, an interpolator can be used [2, 28].

On the other hand, the control algorithms must be efficient to be executed within each time sample and they must have the sufficient capacity to provide precision motion tracking and rapid disturbance suppression. Furthermore, for cases requiring accurate positioning, mechanical system geometrical imperfections must be decently determined [2, 29]. Tan et al. tried to develop an integrated precision motion control system on an open architecture and rapid prototyping platform [2].

From et al. [30] created a new approach to motion planning and control of manipulators on ships. Robots on ships are exposed to large inertial forces due to non-inertial motions of the ship, which affects both the motion planning and control of the manipulator. Their study aims to develop an approach reducing the wear and the tear on the robot arising from these movements. They concluded that the amount of torque required for reaching the desired configuration could be reduced by including the predicted base motion in the motion planner. They also showed that in this way, the strain and tension on the robot are reduced. Hurmuzlu et al. [31] examined the problem of modeling and control of a class of non-smooth nonlinear mechanical systems also known as bipedal robots. They hereby leaded the way to clarify which stability tools one may use to characterize the stability of a bipedal robot. Indeed, a man-made walking robot is simply a robotic manipulator with a detachable and moving base. However, the complexity of the system depends on the number of degrees of freedom, the existence of feet structures, upper limbs and so on. For many years, many scientists have been studying in the field of modeling and control of bipeds [32, 33, 34, 35].

On the other hand, it is obvious that the robot walking ground characteristics constitute an important variable for the control strategy and the walking pattern. It is certainly another important research topic [31, 36, 37, 38, 39, 40, 41, 42, 43].

The control action in biped robot assures that the multi-linked kinematic chain composing the typical biped could walk suitably [31]. To completely specify the control torques, some researchers used kinematics of human gait as desired profiles [33, 44, 45]. Others specified only certain aspects of locomotion as walking speed, step length, upright torso and such [46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59]. Subsequently, a control scheme to specify the control torques will be chosen according to diverse approaches encountered as follows:

1. Linear Control: This approach assumes no deviation for the posture of biped and linearizes the equations of motion about the vertical stance [31, 44, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70].

2. Computed Torque Control: It combines the computed torque with a time scaling of the optimal trajectories, which allows the finite time convergence of the systems state towards the desired motion [31, 37, 42, 57, 59, 60, 71, 72, 73, 74, 75, 76, 77, 78, 79].

3. Variable Structure Control: This approach based on feedback law ensures tracking despite system parameters uncertainties. It ensures convergence in finite-time. The stability of the motion is provided by the controller that eliminates the errors [31, 37, 58].

4. Optimal Control: Two different approaches have been developed to regulate the smooth dynamic phase of bipedal locomotion systems. First approach deals with minimizing energy-based cost functions of selected parameters in

the objective function [31, 32, 43, 52, 54, 80, 81, 82]. Second approach is the direct application of classical optimal control methods to bipedal locomotion in order to obtain controllers that minimize cost functions [31, 83, 84, 85, 86, 87, 88, 89]. The motion of the biped is regulated over a support phase with a quadratic cost function in the study of Channon et al. [88].

5. Adaptive Control: Although this approach does not have a real advantage in controlling bipedal locomotion, Yang [59] applied it to a three link planar robot. Furthermore, some experiments on adaptive control have been made in Massachusetts Institute of Technology (MIT) Leg Lab [90].

6. Shaping Discrete Event Dynamics: It is crucial to directly control the impact effects on the system state and even an approximation of an input requires actuators with high bandwidth. Some authors derived the expression for the system state immediately before the impact instant [57, 58]. Another objective can also be to remove the effect of the impact as in the study of Dunn and Howe [91]. They minimized or eliminated the velocity jumps due to the ground impact and limb switching. Some other scientists used a feedforward input that modifies the motion at the end of each step from measurements information [92, 93, 94, 95].

7. Stability and Periodic Motions: In his study, Hurmuzlu [57] analyzed the nonlinear dynamics of planar, five-element biped and revealed a rich set of stable, periodic motions that did not conform to the classical period on locomotion. Stable gait patterns may deviate from the objective functions results because of the tracking errors in the control actions. Chang and Hurmuzlu [57, 58] overcame this difficulty by partitioning the parameter

space to choose specific values that lead to a desired gait pattern.

Lately, to handle the complexity of more sophisticated and elaborate motions, scientists have been using integrated motion control systems. Wu et al. proposed an adaptive robust motion control method to control X-Y table driven by high precision linear motors [96]. The results showed excellent tracking performance of the system.

Recently, Lin et al. proposed an interval type-2 fuzzy neural network control system to control the position of X-Y-Theta motion control stage using linear ultrasonic motors to track various contours [97]. This method is used to handle the uncertainties of the motion control system. The method is proved to be performing and robustly skillful.

2.2 Frameworks

By definition a framework is a set of common and prefabricated software building blocks that programmers can use, extend or customize for specific computing solutions. With frameworks, developers do not have to start from scratch each time they write an application. Frameworks are built from collection of objects so both the design and code of the framework may be reused [98].

A framework improves quality, consistency, and usability by forcing the creation of different modules in the framework that can be verified by themselves. There are a number of key advantages to be gained from using application frameworks. The primary technical advantage is that they provide design and code reuse. The larger and better an application framework, the more design and code reuse becomes possible. Also, systems based on frameworks are easier to maintain, because most key design and implementation decisions are localized in one place which is the framework [99].

2.2.1 Robotics Frameworks

Several frameworks for the construction of mechatronics or robotics systems have started to emerge in the market and have been adopted by many developers.

Using a framework for programming robotics and mechatronics applications can have many advantages. A unified programming environment provides easy programming and simulation, a service execution environment provides a base for the software to execute its various modules and functions. Reusable components of a framework will provide transfer of previous work to new projects. Robotics frameworks usually have drivers for the popular robotics hardwares and they provide software abilities such as computer vision, navigation and obstacle avoidance.

There are many frameworks currently on the market. From an initial point we can classify these frameworks as commercial and non commercial. Currently the development of motion control applications is growing rapidly but it is not close to the level of general computer programming or web development frameworks. One disadvantage of using a framework is that once it is in use and the time has been invested on it, it is difficult time wise and financially to migrate to another platform.

There are several commercial frameworks that deal with robotics applications such as Evolution[100], Skilligent[101], URBI[102] and Webots[103]. Microsoft has also released a robotics studio but that is free of charge.

Solutions presented by MobileRobots 2.2 and Skilligent can be considered as complete controllers for a robots interaction. After the motion control of the

Figure 2.2: MobileRobots Robotic Platform

robot is solved, these frameworks handle the interactions of the robot with its environment. Skilligent provides a complete solution for a mobile robot Figure 2.3, enabling it to be trained by the end user to perform tasks by visual learning. MobileRobots on the other hand provides indoor navigation for robots. MobileRobots software is oriented towards the needs of its own hardware platforms. Skilligent has a robust architecture that features redundant controllers, actuators and sensors making it more fault tolerant than other frameworks.

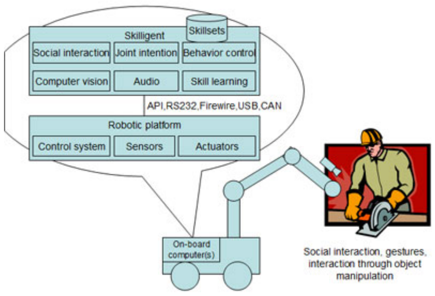


Figure 2.3: Skilligent Robotics Architecture

Gostai Urbi is a commercial robotics framework that has its own runtime environment and development tools. Urbi has tools for the creation of the user panel and some tools to record and replay the actions of the robots like dances and other movements. Urbi also provides tools for the development of the robots actions in a visual manner. Cyberbotics Webots is utilized as a simulation environment for the framework.

Evolution Robotics Platform (ERSP) is another commercial platform for robots. ERSP specializes in visual object recognition and vision based localization and mapping.

Microsoft Robotics Developer Studio (MSRDS)[104] is a robotics framework developed by Microsoft. Unlike the other commercial solutions it is free of charge which has made it somewhat popular. MSRDS has a graphical drag and drop style program creation environment. The software architecture is built on a run-time environment that is based on the .NET platform and a set of libraries called Concurrency and Coordination Runtime (CCR) to manage the communication between the different services operating the robot. MSRDS has a simulation environment but relies on third parties to supply navigation, image processing and other tasks. MSRDS supports a lot of off the shelf robots however adding a custom robot to the framework is among the framework's strong points due to the lack of lower level control algorithms.

Open Robot Control Software OROCOS[105] is a an open source set of libraries for motion control and robot control. OROCOS is free and seems to have an established community. OROCOS does not have graphical development environments and simulation environments as with some commercial solutions. However it does offer realtime abilities and kinematics and dynamics libraries. OROCOS is not intended for a distributed architecture.

Player/Stage/Gazebo[106] is a combination of robotics softwares. Player is a robotics network server that provides an interface for communication between various components of a robot such as sensors actuators and configuration scripts.

The robot structure is built upon TCP sockets where all the modules of the robot communicate with each other.

Stage is a simulation environment to simulate robots sensors and objects in a two dimensional bitmapped environment. Gazebo is a 3d simulation environment that operates a 3d environment model that simulates sensors and robots. The simulator is capable of providing sensor feedback to robot softwares.

ROS[107] or robot operating system is an open source robotics framework that provides functions and a platform for the creation of robotics applications. ROS provides device drivers, libraries, visualizers and means for different components to communicate with each other. ROS operates as a server that permits the row nodes to communicate with each other.

The open source frameworks seem to have a component based software engineering approach. Where every element of the system is modeled as an independent node which communicates with the other nodes. Some systems have adopted to make the nodes communicate directly with each other whereas on some systems a central master controller exists to orchestrate the operation of the nodes. While this node based approach permits the expansion of the systems, the passing of messages may cause the system to exit the deterministic domain and become somewhat problematic in achieving sustainable realtime systems. It is the authors belief that currently an open source and non commercial framework that is widely used is the best solution for the development of a motion control system. However if the implementation of the system requires some specific expertise for a specific application then a specified commercial solution may be more beneficial.

To sum up, several software frameworks exist for the creation of robotics applications. But currently there is no standard for the creation of mechatronics systems nor is there a complete software package that can deal with all aspects in the programming of such systems. Existing frameworks each have their advantages and disadvantages, however it is noticed that they generally have limited or no dedicated structure for the development of the motion control aspect of the problem and deal extensively with the robot-environment interactions and inter mechanism communications. Dealing with the higher levels of the problem, they are usually not well suited for hard realtime; since the interactions can run on soft realtime constraints, the relaxation of these constraints enables the base platform for deployment and development to be chosen from a broader spectrum.

2.3 Conclusion

We have presented in this section key definitions to motion control and frameworks. The complexity of the motion control problem and the lack of standardization for motion control software could be facilitated by the many advantages brought on by the use of frameworks.

In this thesis, we propose a new framework for motion control systems that will help the design of motion control systems by separating the hardware management problem from the motion control and system requirement tasks. The framework functions in a modular structure and promotes the creation of reusable modules. The proposed framework can also be intended as a complementary framework for other robotics frameworks.

3 BACKGROUND

3.1 Introduction

This section investigates different platforms on which a complete motion control system can run. In choosing a platform there are several criteria to consider. The first criterion is obviously the ability to perform realtime operations which are needed in most motion control systems. But along with suitability for a specific motion control application, there are other criteria that relate to the development using a platform. Where installation time can be comes in to play. Usually installation time is an indicator of problems to come with the platform and the general ease of use of the platform. Another equally important criterion is the availability of a support or community when difficulties do occur. Besides being realtime capable a platform must also be able to interact with devices. It is not desirable to design all the interactions of the system with the outside world and it can be advantageous to utilize off the shelf hardware. The availability of drivers is an indicator of the compatibility of a platform and some commercial platforms fail rather badly in this regard. Finally, price is always an important criterion, while some platforms may be cheap to begin with, the commercialization may require that mass production of the system has cheaper royalties for the software.

3.2 Base Platforms for Realtime Systems

Motion control requires acquiring position information, force information and various other sensor informations to be able to extract data from the surrounding. Motion control also requires outputs to be generated and sent out from the system. Motion control above all requires fast processing times which have to work under a realtime structure as any imperfections in the timing can hinder performance, stability and accuracy of the system. To be able to satisfy these functions the framework needs to be built on a fast and flexible basis that has realtime abilities so desired performance can be achieved. Motion control systems need to work in various form factors and sometimes be mobile. Mobility requires a small form factor and low power consumption, this usually means that the hardware supporting the control logic has to be efficient and lower performance not to have to deal with the excess heat issues associated with high performance. Also some motion control projects may be budget dependent and may need non-high end hardware. To satisfy both of these conditions the basis for the framework must be chosen such that it can be scaleable. Scaleability is the ability to add functions and additional features to the system as they are needed or the ability to strip the system of all unnecessary abilities so that they do not consume any processing power or memory. Also for expandability and ease of use, the framework must be easy to modify and preferably have a community that can support it. Several different platforms were examined in the quest to find a suitable platform on which to build the motion control framework.

3.2.1 Windows XP

Windows XP, is an operating system developed by Microsoft. Even though windows XP is not a realtime operating system some experiments were done with its abilities at control, and it was noted that operation of the services of the OS itself and other applications can retard the performance of the system almost at the seconds level. The operating system was tested using an IO card and a C# program using an off the shelf I/O card. A simple program was written to perform motion control for a motor using a PID controller. It was noted that sometimes the program could not operate for seconds at a time, while other applications were utilizing the CPU or while the OS was performing some services.

3.2.2 Windows XP Embedded

Windows XP Embedded is a version of Windows XP targeted towards developers of embedded devices, for use in specific consumer electronics, set-top boxes, kiosks/ATMs, medical devices, arcade video games, point-of-sale terminals, and Voice over Internet Protocol (VoIP) components.

Windows XP Embedded, commonly abbreviated "XPe", is a componentized version of the Professional edition of Windows XP. A developer is free to choose only the components needed thereby reducing operating system footprint and also reducing attack area as compared with XP Professional. As an advantage to developers, XP Embedded provides the full Windows API, and support for the full range of applications and device drivers written for Microsoft Windows. The system requirements state that XPe can run on devices with at least 32MB Compact Flash, 32MB RAM and a P-200 microprocessor. The devices targeted for XPe have included automatic teller machines, arcade games, slot machines, cash registers, industrial robotics, thin clients, set-top boxes, network attached storage (NAS), time clocks, navigation devices, railroad locomotives, etc. Custom versions of the OS can be deployed onto anything but a full-fledged PC; even though XPe supports the same hardware that XP Professional supports (x86 architecture), licensing restrictions prevent it from being deployed on to standard PCs.

Componentized OS Write Filters XPe includes feature components known as write filters, which can be used to filter out disk writes. The volumes can be marked as read-only using these filters and all writes to it can be redirected. Applications in user mode are unaware of this write filtering. XPe ships with two write filters: Enhanced Write Filter (EWF) protects a system at volume level. It redirects all disk writes to a protected drive, to RAM or a separate disk. EWF is extremely useful when used in Thin Clients that have flash memory as their primary boot source. File Based Write Filter (FBWF) allows the configuration of individual files as read/write on a protected volume. USB Boot XPe adds a USB boot option to Windows. An XPe embedded device can be configured to boot from a USB drive. This feature is use full for systems that have to be mobile, i.e. where the system is subject to high G forces or where the system vibrates where any mechanical data storage equipment such as a hard disk can become damaged. CD Boot An XPe device can be configured to boot from a CD-ROM. This allows the device to boot without the requirement of having a physical hard disk drive as well as provides a "fresh boot" every time the image is booted (a property inherited by the fact that the operating system is being booted from read-only media). One drawback to this technology is updating or servicing the image requires the complete process of setting up the runtime image to be completed once again from start to end. Network Boot An XPe device can be configured to boot from a properly configured network. Synonymous to CD Boot, Network Boot removes the requirement of having the physical hard drive as well as providing the "fresh boot" behavior. One bonus to Network Boot though is the ability to service the already setup image. Once the image is updated the image is simply posted to the RIS Server and once clients are rebooted they will receive the updated image.

While testing this platform as a basis it was noted that it is extremely easy to acquire the specifics of the deployment platform. It is easy to build the OS by means of selecting the components, and easy to launch the OS on the chosen hardware.

Windows XP Embedded in general has a lot of good features that can be used directly in motion control and robotics applications, however it does not support realtime therefore any precise motion control tasks can be preempted which will cause the motion control system to have unacceptable performance.

3.2.3 Windows XP With Intime Extension

TenAys and various other companies are developing extensions for windows that enable realtime tasks to be run on these machines. Windows XP with INtime Extension or Windows XP Embedded with INtime extension is an architecture that installs 2 virtual machines, and has standard Windows running on one virtual machine while all time critical realtime tasks run on the other virtual machine that has an INtime RTOS on it. This enables the usage of all the Windows features for creating programs.

This has proven to be easy to install and work with, however drivers for the input and output devices have to be re-written for the real-time part of the system to be able to access them. Data transfer between the Realtime side and the non real-time Windows side is done by shared objects. While this system was tested, it was discovered to be easy to transfer data between the realtime side and the Windows side, and the INtime RTOS tasks performed nicely. However this solution proved to have one major and a few minor setbacks for the work involved. Since there is a different operating system that handles the realtime tasks, for it to be able to handle the realtime IO operations needed for motion control it has to recognize the IO cards. The drivers for the available IO cards were not available from the distributor and it was discovered that drivers were very very limited for this operating system. An attempt to write a driver was semi successful, but when the trial license for the software ran out, progress on this platform stopped as the cost was not justifiable considering the extra work involved.

It was noted however that having the ability to write windows programs that can act as the man machine interface for realtime systems, increases the potential and productivity of the work dramatically. It was also noted that there is no community behind this platform therefore problems have to be solved without consulting others which is a major setback.

3.2.4 Lean Linux

We can call an operating system that has been stripped down of all its non essential components a lean operating system. Removing unnecessary components of a linux system can be considered as the linux equivalent of creating a Windows Embedded operating system. A lean linux operating system without any support for realtime was also tested. It was noted that although the operating system can go to higher frequencies and generally has a lot lower jitter that the windows XP system, randomly the jitter increases to 150ms levels while the OS is performing other tasks.

3.2.5 Windows CE

Windows CE is Microsoft's operating system, designed for devices. Unlike windows embedded it is not a stripped down version of the operating system for desktop computers but a different OS altogether. Windows CE is optimized for devices that have minimal storage, a Windows CE kernel may run in under a megabyte of memory. Devices are often configured without disk storage, and may be configured as a closed system that does not allow for end-user extension (for instance, it can be burned into ROM). Windows CE conforms to the definition of a real-time operating system, with a deterministic interrupt latency. The OS supports 256 priority levels and uses priority inheritance for dealing with priority inversion. The fundamental unit of execution is the thread. This helps to simplify the interface and improve execution time. Another feature of Windows CE is that it can operate with different processors such as Intel x86 and compatibles, MIPS, ARM, and Hitachi SuperH processors. Making it a candidate for mobile robotics or motion control on small devices.

During the trial of this platform, it was noted that the development environment was very easy to use. It was easy to create a the windows CE operating system using the platform builder software. Writing programs was also easy using the Visual Studio application. However, it was discovered that this platform too had the problem of supporting only a few IO cards. An attempt at writing a driver for the available IO card did not yield useable results, probably due to lack of experience. Also drivers for most ethernet cards do not exist. In future when this platform has wider driver support it may be adopted as the basis of a motion controller. It was also noted that unlike the INtime platform this platform has a community behind it who are willing to help answer questions and solve problems.

3.2.6 Micro Controller Based Solutions

Implementing a solution on a micro controller or a DSP can be considered a good option for motion control applications. As the micro controllers develop rapidly the have started to have more and more power and some can have operating systems on them. There are many microcontroller fast prototyping boards. Without an operating system they can be programmed to work on timer interrupts and other interrupts, and this makes them realtime if the code written in to them can finish the work in the desired time slot. However they are not suitable for fast prototyping, and are relatively time consuming to develop the other hardware components that are needed on a motion controller, also they are not expandable easily. Micro controllers can be considered as a solution after all aspects of the task have been fully considered and all the system specs have solidified, they are a solution that is more product oriented rather than development oriented.

3.2.7 FPGA Based Systems

FPGA's or Field-Programmable Gate Arrays are integrated circuits designed to be configured by the designer after it has been manufactured. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application specific integrated circuit (ASIC). FPGA's are increasingly used to implement any logical function that an ASIC could perform. The ability to update the functionality after the chip has been produced and not having to fabricate a new ASIC are among the benefits of these devices.

FPGA's contain programmable logic components called logic blocks and a hierarchy for reconfigurable interconnects that allow the blocks to be wired together. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

The most common FPGA architecture consists of an array of configurable logic blocks (CLBs), I/O pads, and routing channels. Generally, all the routing channels have the same width (number of wires). Multiple I/O pads may fit into the height of one row or the width of one column in the array. An application circuit must be mapped into an FPGA with adequate resources. While the number of CLBs and I/Os required is easily determined from the design, the number of routing tracks needed may vary considerably even among designs with the same amount of logic. (For example, a crossbar switch requires much more routing than a systolic array with the same gate count.) Since unused routing tracks increase the cost (and decrease the performance) of the part without providing any benefit, FPGA manufacturers try to provide just enough tracks so that most designs that will fit in terms of LUTs and IOs can be routed. This is determined by estimates such as those derived from Rent's rule or by experiments with existing designs.

FPGAs can have very many advantages for motion control. By their nature, the logic inside an FPGA can operate in parallel. This makes motion control systems expandable in terms of degrees of freedom until the FPGA runs out of gates. FPGA's have a major disadvantage of having slow development times. They do not support coding using higher level languages yet, and the availability of gates is not sufficient for easy development of applications. FPGA's are one of the platforms that have the ability to do very fast control and they are worth investigation in motion control applications.

3.2.8 RTAI Patch for Linux

Linux Real Time Application Interface is a patch for a Linux kernel that installs itself on top of the kernel. It is a very slight modification of the kernel, consisting of a few hundred lines of code. This change, routes all the interrupts to the RTAI code, and also makes the Linux kernel completely preemptable. What this means is that, any code running in the RTAI side, can stop the kernel and start working, also this applies to all interrupts, they are sent to the user defined realtime functions. However unlike the other realtime patches like for windows, if these interrupts are not caught by the RT code, then they are passed on to the Linux kernel where they are handled as they normally would be. RTAI has the ability to work with multiple processors, with the following schedulers.

RTAI makes available three schedulers: Uniprocessor (UP), optimized for uniprocessor machines; Symmetric Multi Processors (SMP) and Multi UniProcessor (MUP) for symmetric multiprocessors applications. The SMP scheduler affords the best compromise between flexibility and efficiency in kernel space applications using RTAI proper kernel tasks, as it can schedule any ready task on any CPU, while allowing to selectively impose selected tasks to run on a specific CPU or CPUs cluster. The MUP scheduler instead imposes that any task is assigned to a specific CPU from its very creation and can achieve better performances because it can exploit memory caching more efficiently. RTAI specific kernel tasks can in any case be moved to different CPUs dynamically at execution time. Instead inter CPU migration of Linux tasks and kernel threads cannot be done in true hard real time. There is no restriction in the use of any scheduler, real time tasks can interact without any constraint, irrespective of what CPU they are running on. SMP and MUP schedulers can be used also with uniprocessors. Another important feature of all the schedulers is the possibility of choosing between either a base periodic timing, with a fixed assigned time resolution tick, the approach mostly used in RTOSes, and an arbitrary timing, allowing scheduling a task at the resolution of the available clock by firing a one-shot timer at the time instant imposed by the highest priority task waiting on the timed list. The one-shot mode avoids any compromise on the least scheduling resolution, thus giving an almost continuous time resolution, while the periodic mode requires to have any task timed at an integer multiple of the basic timer period. However we must recall what pointed out in the introduction and avoid any illusion that on GPGPUs a task can be scheduled with a nanosecond precision. It is also important to note that the MUP scheduler uses independent per CPU timers and each of them can run independently from any other, so that timers mode of operation can be freely assigned, e.g. there can be periodic and one-shot timers and periodic timers need not to run at the same period. RTAI schedulers make available the following scheduling policies:

Fully preemptable First In First Out (FIFO), for voluntary co-operative scheduling. A task owns the CPU till it does not release it or a higher priority task preempts its execution. Under FIFO scheduling there is a support function to help meeting periodic tasks deadlines with statically assigned priorities according to the Rate Monotonic Scheduling (RMS) concept.

Round Robin (RR), like FIFO but only up to a certain allowed per task time slot, after which the CPU is tentatively handed over to any equal priority task waiting on the ready list.

Early Deadline First (EDF), to dynamically assign priorities in order to meet periodic tasks end of execution deadlines. It requires that the user assigns a relatively good estimate of the execution time required by each periodic task.

It must be noted that under symmetric multiprocessing it is also possible to handle external interrupts either in a symmetric way or to force them to a specific CPU, or CPU cluster.

3.2.9 Conclusion

All the above platforms were examined and were compared according to their ease of use, setup time, price, support, realtime abilities and driver support as illustrated in table 3.1.

Table 3.1: Comparison of Motion Control Software Platforms

Platform	**Setup Time**	**Real Time**	**Support**	**Price**	**Drivers**
Windows	0	No	10	100	Good
Windows XPe	1	No	9	20-100	Good
Windows CE	1.5	Yes	7	5-30	Limited
Windows RT Extension	1.5	Yes	1	3500	Limited
Linux	0	No	9	0	Delayed
Linux + RTAI	1.5	Yes	8	0	Delayed
Microcontroller	5	Yes	7	0	-
FPGA	5	Yes	2	0	-

Examining these platforms it is noted that it is most worthwhile to develop applications on a real time enabled Linux platform due to its flexibility and good realtime characteristics. An FPGA platform due to its potential to achieve higher speeds and parallel processing is also worth investigating. It should be noted that the windows platform is by far the easiest to develop non-realtime applications.

3.3 Framework Design

This section discusses some of the design choices made during the creation of the framework.

Design of a framework has many challenges from language selection to module design to implementation. As the complexity of a framework increases the usability of the framework decreases. More time has to be invested in learning the framework than its actual use. The coding of the framework should be as simple as possible with well defined design patterns. Creation of elegant code is a form of art using all the possible features of the programming language. However, every solution should be catered to its users, in this case, the users have a background in programming as engineers but it is not extensive.

Motion control systems are developing rapidly and the algorithms and different implementations are also increasing at a rapid rate. Utilization of a framework requires dedication to the framework. In the case that a framework ceases to provide necessary infrastructure for the task then the framework needs to be changed or extended, migration from one framework to another one is a costly process. Although this framework covers most of the aspects needed to conceive a motion control system there will inevitably be exceptions. The proposed framework provides transparency in its implementation and it is fully extensible being open source and written in a simplistic manner. During the design of the framework an object oriented approach to modeling the motion control was pursued. The most popular programming language to date is C with Java coming close and C++ coming after that [108]. These languages are mostly syntax compatible. C based languages are still widely thought in universities and most realtime systems and micro controllers are programmed using assembly or C. Therefore the framework should be programmed in C because of its simplicity, existence in curriculums and the abundance of programmers familiar with it.

Utilization of a framework in software engineering usually causes the compiled code to take up more space which is also called code bloating. In order to minimize this phenomenon the framework is provided in source code form that is compiled. The compiler's abilities for optimization are utilized to remove any unused sections which may take up unnecessary memory in the created program. This is also a necessity for the generated software to work on multiple platforms as executable files cannot be transferred from one platform to another.

The complexity of the problems in designing a motion control framework necessitates a separation of concerns approach [109] because complexity is high and needs to be reduced. Reduction of complexity and overall clarity of code is achieved by breaking up the framework up into composable pieces. Each of the pieces can then be analyzed, designed, and understood individually.

4 A FRAMEWORK FOR MOTION CONTROL SYSTEMS

Developing a motion control system requires much effort in different domains. Namely control, electronics and software engineering. In addition to these, there are the system requirements which may be completely different to these spanning from biomedical engineering to psychology. Collaboration between these fields is vital, however these fields should be involved only as much as they are needed to be in the fields of expertise of the others. The software framework proposed in this study aims to achieve a level of abstraction between the different domains utilized within a system.

The aim in using the framework is to achieve a sustainable software structure for the system. Sustainability is an important part of systems, as it permits a system to evolve with changing requirements and variable hardware, with the ultimate goal of having robust software that can be utilized on different platforms and with other systems using an abstraction layer between the hardware and the software. This ensures that the system can be migrated from a processing platform to any other platform and also from one set of hardware to another Fig 4.1.

The sustainability of the system is obtained by creating layered and modular architecture for the system which inherently brings reusability to the software that is generated. This is achieved by creating software blocks that provide an interface for them to connect to other software blocks. The standardized interfaces of these blocks enable their reuse in other projects. Modules of the system such

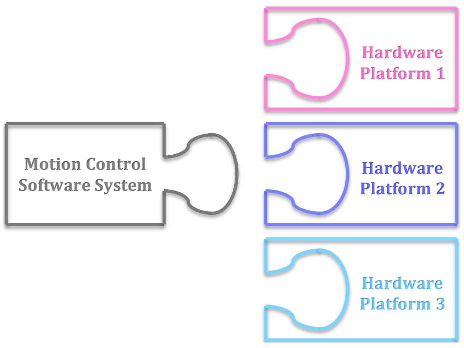


Figure 4.1: Software developed with the framework can be mapped and used on different platforms

as actuators, sensors and controllers can either be reused or generated writing structured software or by using the frameworks functions. Fig 4.2.

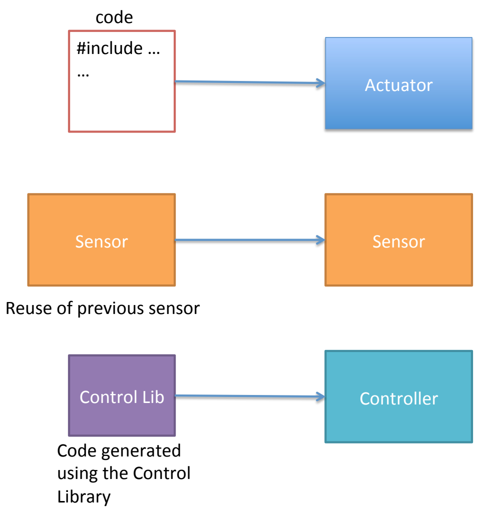


Figure 4.2: Software block examples

Layering the software has several advantages during development and deployment. Every layer of the system can be built with standard interfaces within the framework. This is achieved again by the creation of higher level software modules that utilize the lower level software modules. As a module has standard interfaces the system can be designed around even a module that does not exist. This permits both a top down and a bottom up development to be simultaneously pursued.

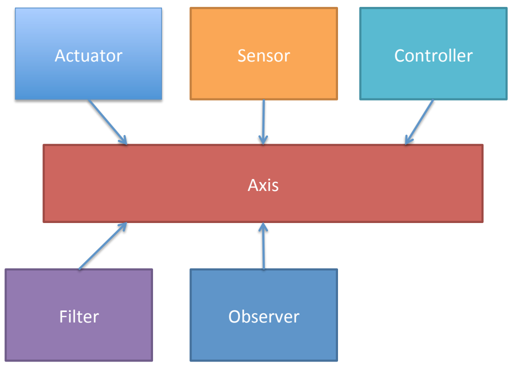


Figure 4.3: Using modules to create higher level modules

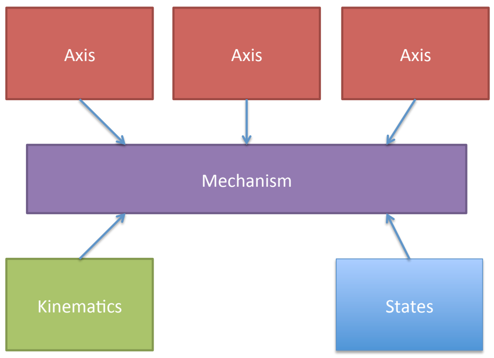


Figure 4.4: Using modules to create higher level modules

The framework models the basic components of a motion control system such as actuators, sensors, controllers and provides standard interfaces to these components. Libraries for control and other functions are provided for the connection of the modules to form higher level modules Figure 4.3. The interactions of these components are also modeled using a layered manner providing the ability to combine the modules in to higher modules like mechanisms Figure 4.4.

Initially the hardware components of the system that interact with the non software part of the system by means of IO channels are defined and integrated to the framework. This forms a complete abstraction between the hardware and software as the software interfaces with a fixed set of functions for the hardware. After this step, the workings of the physical actuators and sensors are modeled passing from an electronics domain communicating in volts and amperes to the motion control domain which deals with positions and forces. Then the framework focuses on the addition of control components such as controllers and filters. Later the interaction between the hardware components is linked using the control components, creating a higher layer of components such as degrees of freedom, i.e. modules that are capable of performing basic motion control functions such as following position or velocity references. Then these modules are further developed by grouping them together and addition of structures such as kinematics components forming mechanisms which is another layer of the system. This process continues until all the components of the system are modeled and the components are grouped together forming the motion control system. Before the task of getting the system start using its motion control abilities to perform actual tasks of use the system software is integrated with a communication module which enables the interaction with other systems or the reception of tasks by a graphical user interface Figure 4.5.

This framework is not intended to be utilized in lieu of other frameworks, rather it is intended to complement them. The current frameworks deal with the behavioral aspects, obstacle avoidance and other interactions of the systems. These frameworks do not take in to account the lower levels of the motion control creation task. And usually assume that the hardware has its motion controller attached. It can be considered that these frameworks deal with modeling the *whattodoandwhentodo* of the motion control tasks where as the proposed framework focuses on the *howtodo* aspect of the motion control problem. The motion control framework proposed in this study provides development of a system until it can be connected to an existing framework. As the framework utilizes C/C++, the systems created can either be coded into the host platform if it is realtime capable or the framework can be utilized with a realtime base platform and the resulting system can communicate over communication protocols. This makes the framework capable of complementing other frameworks or coexisting with them.

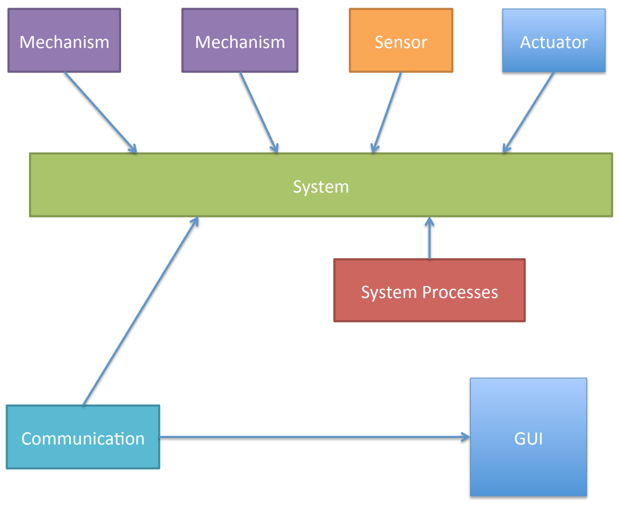


Figure 4.5: System conception

The motion control problem consists of two separate tasks, one part for the construction of the realtime control modules used in precise motion control and one non-realtime part for the construction of the offline processable tasks and the man machine interface.

4.0.1 System Design Methodology Overview

This section describes the phases necessary to identify the needs of the framework. For this aim we have conceived the following design methodology. The proposed design methodology combines initially a top down and bottom up methodology which are applied simultaneously in a methodological manner, where the top down approach is used to well define the necessities of the systems and the bottom up approach is used to model the different hardware and information components of the system in a semi object oriented manner. Then the two approaches meet in the middle in where an iteration is performed until the completion of the task. The approach relies heavily on a well structured form of layering to facilitate the separation of the different problems relating to the different engineering fields of the global task. In this methodology each layer possesses its own difficulties and necessitates its own dedicated functions. Common modules forming a framework greatly improve the development time and development ease of mechatronics systems. These functions range from ones containing control functions for the motion control layer to database connections and data-mining functions.

4.0.1.1 Hardware and Platform

Selection of items for the system is composed of the following phases. The definition of the hardware and OS prepares the system for the development of the software.

1. Definition of System Requirements

    Motion Control Requirements

    Man Machine Interface Requirements

2. Hardware Selection

    Resolution

    Frequency and Response Time

    Platform Compatibility

3. Platform/OS Selection

    Realtime Constraints

    Hardware Compatibility

An important and sometimes overlooked or rushed aspect of system conception is the well definition of the system requirements. For the system generation process to flow smoothly it is vital for this step to be completed successfully. The system definition step can be separated in to two components, initially there are the Man-Machine Interface (MMI) requirements. These define how the system will take data as inputs. MMI's can range from a single button to start and stop a system that does a very specific task to a multi computer, graphical interface and haptic device system. Defining these parameters will define the data connections and modes of operation for the system. This step also handles the addition of other devices to the system such as mice, joysticks or haptic devices. On the other side of the design aspects are the motion control needs of the system. These tasks define the lower levels of the system. The task or tasks necessitate certain tolerances and certain precision which leads directly to the choice of sensors and actuators to be utilized in the construction of the system. After the selection of the sensors and actuators these must be interfaced to the platform that runs the motion control software for the system. This requires the selection of IO cards that must meet the previously defined motion control criteria. Final stage for completing the hardware and platform design is the base platform and OS selection. The base platform and OS should be capable of supporting the selected IO electronics. The selected platform should also be capable of running in either hard or soft realtime depending on the task requirements.

4.0.1.2 Software

Sustainable software can be achieved by separation of concerns. In this framework, the conceptual aspects of:

- interfacing with hardware
- modeling of actuators and sensors
- creation of degrees of freedom
- creation of complex mechanisms
- control theory

    controller

    filtering

    estimation

- trajectory generation
- kinematics
- communication

have been separated organizationally at the implementation level.

To define lines between the separation, the concept of layers is utilized. Layering an application has particular importance in the development of large projects. In a layered project each aspect of the project can be defined, isolated and then resolved in their respective problem domains. In this text the words ”levels” and ”layers” are used in an interchangeable manner. Layers is a word that is used to segment a project into logical parts whereas the word level is used more as a segmentation from a software perspective.

Levels of an application are the degrees of complexity and detail that the application possesses. As a rule of thumb a new layer is created with every new technique or technology is introduced. A new layer is also introduced when several components of a layer are combined to form a different object in the real world like the combination of 3 linear stages forming a cartesian robot.

Typically lower levels of an application are close to the hardware and core components such as drivers and motors whereas higher levels of an application are much closer to the specifications of project tasks, but from another perspective we may also say that the lower levels are closer to software functions and mechanics calculations whereas upper layers are closer to the user level or MMI. Higher levels of the project have abstraction from the hardware and electronics of the system. In a way this hides the details of the system making it user friendly as it includes concepts from the problem domain instead of those of the mechanical and electronic domain. Separating an application into several levels eases the project development phase in that it enables the challenges in the different levels to be tackled separately. This enables a level of abstraction concerning a problem, enabling one layer not to interfere with another layer which also enables the people working on different layers of a project to work semi independently with their individual work being joined on the interface level of the different levels of the project.

Building the project up in layers is most convenient where the project will have slightly varying specifications during its lifetime and the full abilities of the machine/system are not known. Building the layers up enables the people involved in the problem domain to view the algorithms and methods, understand them and contribute to them.

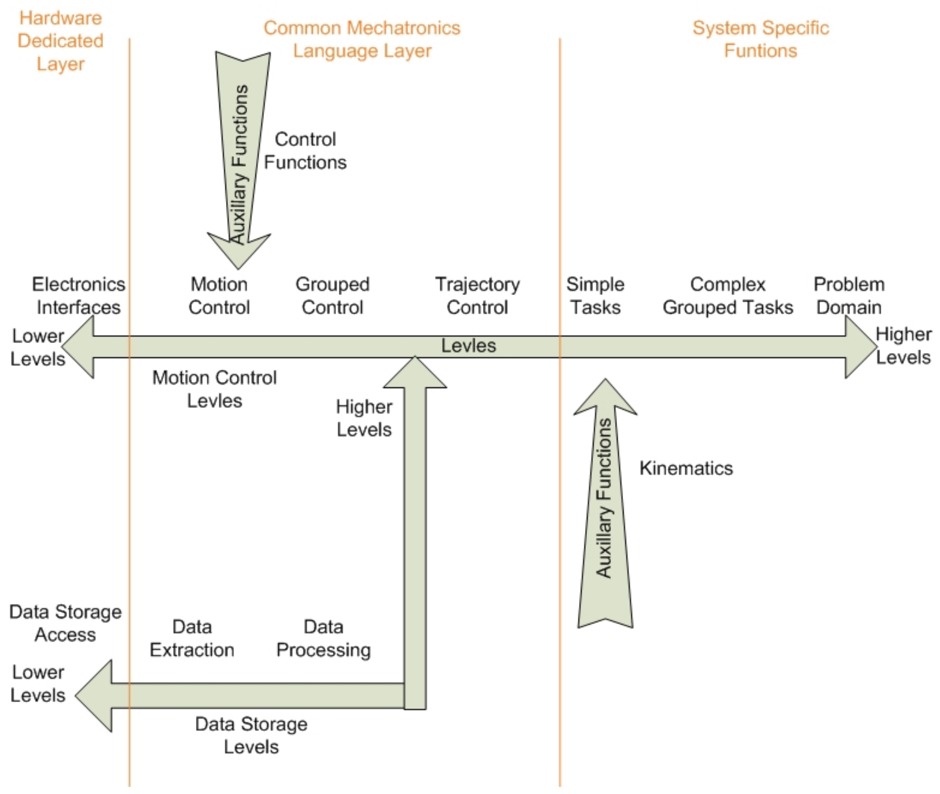


Figure 4.6: Layers of Design

The lowest level of layers in a project is the interface to various hardware elements of the system and the higher levels are closer to the man machine interface and the tasks issued by it.

Figure 4.6 depicts the different layering of a typical project. The dedicated hardware level is part of the project that interacts with the physical elements of the system which are the electronics and the databases or data storage. These functions are wrapped up in common functions and introduced to the upper layers in a common syntax. Once the functions have entered the common mechatronics language layer, then it is possible to start using these functions to model each of the components in an object oriented manner. After every simple component has been modeled and brought as an object into the mechatronics language domain, then these basic objects can be used to build the next higher layer which handles more complicated components such as the cartesian robots. For the efficient creation of the higher level objects auxiliary function libraries that permit motion control are utilized to define the interactions between the lower level components forming the upper layer. At a certain point in the development of the objects, there will be the need to access certain data about the objects. After all components have been layered, then the layers pass on to the simple task level. With the passage into this level, the components start becoming system specific. With the development of the simple tasks and the complex tasks, the system reaches the problem domain. When the system is coded up to this level, cooperating with people from the problem domain becomes easier. As the components that are in discussion are now, robots, mechanisms, conveyors and various other structures that are much easier to relate to than controllers, actuators, sensors, volts.... This level is easily discussed verbally as for every component of the system there is a software module, and focus can be put on feeding these modules with the necessary references to achieve the tasks required of the motion control system.

4.1 Components of the Framework

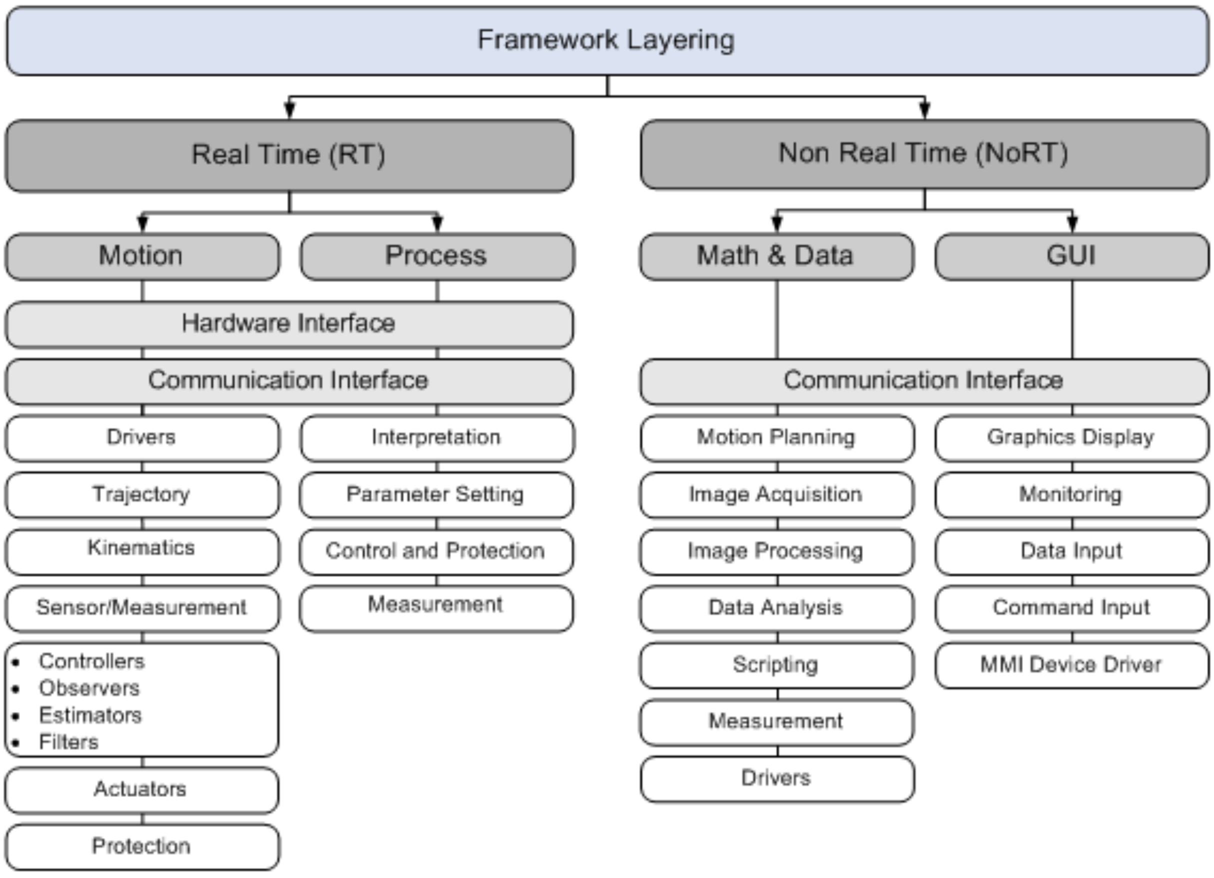


Figure 4.7: Modules of the Framework

This section describes the different modules or components of the framework.

Modern frameworks use the term nodes of independent algorithms or functional control blocks that communicate with each other to form a functioning system. Observing from a programming perspective, these are different threads or processes with different communication methods. Every independent process/thread needs at least the following sections to operate:

1. initialize

2. setup
3. loop
4. delete

Initialization phase is the activation of the hardware components such as IO cards and the creation of the components of the system. Setup phase consists of defining the connections between the components and assigning their parameters. The deletion phase consists removing the resources allocated to the components and shutting of the IO cards.

Each of the components generated within the framework have affinity to these modes of function in order for them to be summoned during the respective operation phase of the system. This approach permits created components to be injected into any already functioning system such as a system running Orocos/RTT [105] or a dSPACE[110] platform. All systems that have been observed have three primary states from a system runtime perspective. These are the initialization phase, the working phase which consists of a loop and a finalization or deletion phase. The modules constructed using the framework are developed into functions and structures that finally are finally combined in three functions *initialize*, *loop* and *delete* corresponding to the three primary states of the motion control system. This separation permits the system to be injected into any hardware platform with minor modifications to the software Figure 4.8.

A motion control application can be separated in to several layers which require their own expertise. The different layers of a typical motion control application are depicted in Figure 4.7. From a programming perspective these layers can be separated in to two categories. Real time and non real time parts. This sepa-

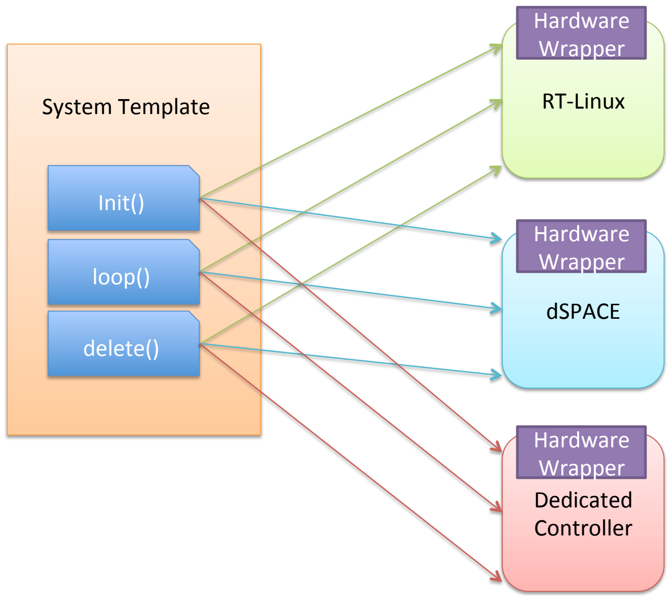


Figure 4.8: Platform independance

ration is important in the design of motion control software because of timing constraints. The layers of the system requiring realtime features are generally the ones that interact with the hardware. The computer control algorithms for motion control need to run at specific frequencies in order to perform and this can only be guaranteed if the software platform on which the software is running is realtime. Therefore interaction with hardware by the use of sensors and actuators and the associated control algorithms such as controllers observers estimators and filters are implemented as part of the realtime layers of the system. To describe more complex systems layers such as kinematics and trajectories are needed. Because these interact directly with the hardware they must also be implemented in the realtime part of the system. The non realtime part of the system are the layers that do not require such timing constraints. These layers can be considered closer user or the man machine interface. Layer such as graphics display, monitoring of variables, input of data or commands and MMI devices often require inputs from the user or display some information to the user. Image acquisition and processing have slow changing outputs relative to the faster motion control algorithms therefore they can also be considered as non realtime layers. Other non realtime layers of the system include Scripting to provide automation mimicking the inputs of a user, data analysis to evaluate the performance of the system, motion planning that is processed before the actual motions start. These layers and the components created for them are described in their respective sections.

The framework implements a modular structure from a software perspective. The requirements of the software modules of the system have been identified and the module structure in Figure 4.9 has been devised.

The module contains some data structures those are described below:

inputs structure of the module is a structure for receiving information such as position references from other modules. During the runtime of the system, interaction with the module is achieved by writing data to its inputs structure.

outputs structure of the module is a structure for sending information such as control outputs to other modules. During the runtime of the system, interaction with the module is achieved by reading the data contained in its outputs structure.

states structure of the module is used for storing the internal data of the module, for example a module requiring its previous outputs would store this data in the states structure.

parameters structure of the module is used for configuring the module. For example the control coefficients of a controller would be stored in the parameters structure.

When a module is being designed the elements of the states, inputs, outputs and parameters structures are defined. During the initialization phase of the system, the parameters of the module are configured. The inputs and outputs of the module are updated during the runtime of the system.

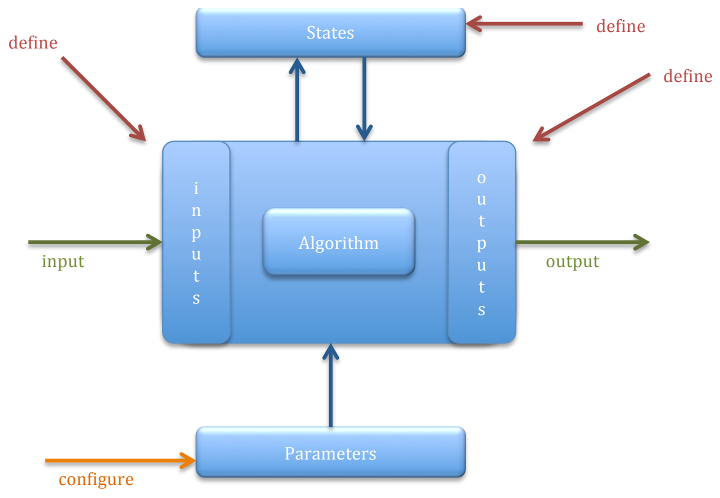


Figure 4.9: Module structure

4.1.1 Hardware Interface

Motion and process control applications have the need to control specific hardware in order to provide the desired movements. The hardware interface of the framework consists of means of providing digital and analog inputs and outputs. This module of the framework provides a standardized interface for the programmer to fill in order to obtain the functionality of the electronic cards to be used from the software that is generated by the developer. A motion control system may use many different electronic cards among virtually limitless options available in todays market. It is desired to be able to use any card that is accepted by the platform which the motion control software runs on. The standardization of this interface is achieved using a technique called wrapping. The framework provides a wrapper template for the electronic IO cards that are used by the motion control system. Once the wrapper template is filled, the remainder of the modules utilize these functions. In common mechatronics systems 6 major types of electronic interfaces between the software and the electronic components have been identified:

- Inputs

  Analog Input

  Digital Input

  Encoder Input

- Outputs

  Analog Output

  Digital Output

  Pulse Width Modulation Output

Each of these interfaces has several functions associated to them. The inputs need to be read and the outputs need to be set. These functions have been respectively prefixed with the words *get* and *set*. Furthermore on some types of IO cards the hardware needs to be initialized or the user may desire to write their own initialization for these cards. Therefore an initialize function exists for each of these interfaces.These functions have been prefixed with the word *init*. There may also be a need to delete these interfaces when the system is shut down. Therefore a delete function has been created which is respectively prefixed with the word *delete*.

4.1.2 Motion

In this section modules for the creation of the motion are presented. The motion layer of the framework presents multiple functions in different areas to enable the software to guide the degrees of freedom of the system to the desired references.

4.1.2.1 Drivers

Drivers are pieces of software that are installed on to platforms that enable the usage of devices that are attached to the platform of choice. These drivers are installed and provide utilization of the hardware. Integration of these drivers into the framework is provided by the usage of templates. Devices that expose the standard inputs and outputs are wrapped by the $HardwareInterface$ template. For stand alone motion control hardware or hardware that controls specific devices are also wrapped with a wrapper to ensure their seamless integration to the framework. It is assumed that the motion control hardware has four different functions. Similar to the input and output hardware provided, there are the initialize and delete functions. The initialize function, initializes the communication with the IO card, sends initialization commands to the device and allocates memory if necessary. The delete function closes the communication to the device and frees all the resources allocated to the device. In addition to these there are two functions one to set the reference and another to acquire the position.

4.1.2.2 Sensor/Measurement

Sensors, also called detectors, are devices that measure a physical quantity and convert it into a signal, which can be read by an observer or by an instrument [111]. These measurement instruments monitor and control processes and operations. They are also largely used in experimental engineering analysis. Certain applications of these instruments may be characterized as having essentially a monitoring function as thermometers, barometers and water, gas, and electric meters. On the other hand, a measuring instrument can serve as a component of a control system since it is first necessary to measure any variable in a feedback control system. It is essential to consider that a single control system as industrial machine and process controllers or aircraft control systems, may require information from many measuring instruments [112].

Development of sensors relies on an evaluating program carried out in 1988, by the National Science Foundation (NSF), together with The Defense Advanced Research Projects Agency, whose goal is evaluating Japanese technology with a program called JTECH. The main motivation of the project was to track the advanced japanese manufacturing processes since they invented the mechatronic concept and implemented successfully in manufacturing. Sensors played a critical role in the monitoring and control of these processes [113].

Today, we are using sensors in a wide variety of areas as indicated in the Table 4.1, to interact with the environment and to obtain information. Luo[113] affirms that sensor technologies are as important in the mechatronic system as senses are to a human being. A mechatronic product requires essentially intelligence and flexibility. Sensors are used in everyday objects as tactile sensors in elevator buttons or lamps. Innumerable application areas include cars, machines, aerospace, medicine, manufacturing and robotics. It has been revealed that 80% of the industrial measurements are of displacement nature when the scientific and technological measurements are substantially of proximity distance nature. The dynamic, unstructured and indeterminate nature of the environment increases the demand for the use of multiple diverse sensors in mechatronic products. The immediate feedback for a reliable and flexible operation in a washing machine requires for example 10 or more sensors to detect the type of materials to be washed, the degree of dirt, the concentration of detergent etc. Providing a continuous measurement of mechanical processes in diverse conditions has become crucial in the automated industrial processes to enhance the productivity and the sustainability [113].

Application Area	**Use Percentage**
Information Processing & Communications	8.0
Scientific Instrumentation	11.7
Electric Power & Energy	5.3
Manufacturing Facilities	18.1
Home Appliance	13.9
Automobiles	7.3
Transportation	1.6
Space Development	2.7
Environment, Security & Meteorology	10.0
Resources & Ocean Development	1.4
Health & Medicine	11.0
Agriculture, Forestry & Fishery	0.7
Civil Engineering & Construction	0.7
Distribution, Commerce & Finance	0.2
Others	7.3

Table 4.1: Sensor Application Areas

The transducer in the sensor senses the absolute value or a change of a physical quantity and converts it into an electrical signal that might be inconveniently

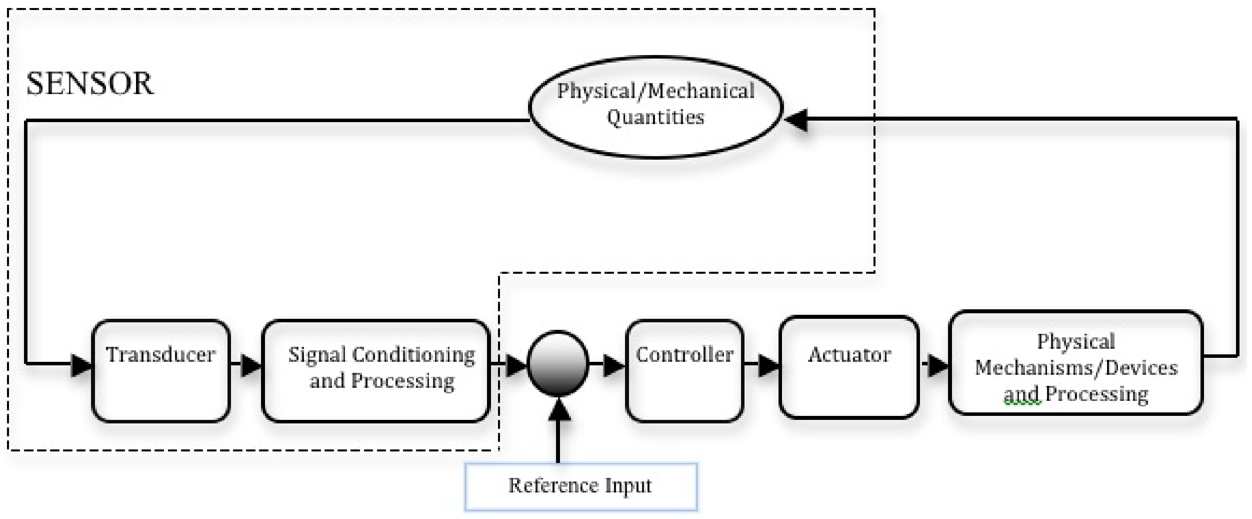

SENSOR
Physical/Mechanical Quantities
Transducer
Signal Conditioning and Processing
Controller
Actuator
Physical Mechanisms/Devices and Processing
Reference Input

Figure 4.10: Single Sensor Control

small. To handle this inconvenience, an amplifier and signal-conditioning circuit is used Figure 4.10 It is vital to consider that a mechatronic product must provide a functional and spatial interaction between mechanical, electronic, control and information technologies in a synergistic way, besides it requires intelligence and flexibility [114].

Some of the categories of sensors used in mechatronic systems are listed hereinafter:

- Inductive Proximity Sensors
- Capacitive Proximity Sensors
- Photoelectric Proximity Sensors
- Ultrasonic Proximity Sensors
- Linear Variable Differential Transformer Displacement Sensors

- Solid-State Sensors
- Fiber-Optic Sensors
- Force-Torque Sensors and Load Cells

There are two basic types of pressure/force sensing device: capacitive sensors and piezoresistive sensors. The pressure applied on the capacitive sensor, made up of a flexible diaphragm creates a change in the capacitance and induces a frequency shift in the circuit. On the other hand, a piezoresistive material is one whose electrical resistance changes with a change in pressure on the material. This is the working principle of a strain gauge [113].

A load cell converts a force into electrical signal in two stages. In the first stage, the force sensed deforms mechanically a strain gauge. The strain gauge measures the deformation as an electrical signal by changing the effective electrical resistance of the wire. A load cell may consist of four strain gauges in Wheatstone bridge or of one or two strain gauges [115, 116].

From the perspective of the framework, sensors are means of obtaining real world data to the digital domain. After this data has been digitized by the necessary means, there is a need for it to be translated into meaningful data. The objective on implementing a sensor for the mechatronics system is to create a standalone object or set of functions that will enable the sensor to be utilized by the framework Figure 4.11. For this operation a few pieces of information are needed. The function block for sensor measurement are a set of functions that translate data obtained from IO devices to the framework. Once the integration to the framework is complete, the remaining layers of the system do not need to know the details or workings of the sensor. All that is need is the data measured upon request by the sensor and the units of the sensor. The sensor interface utilizes the inputs to read the information and perform calculations to provide data to the system. Each sensor function utilizes a sensor structure where some data is kept indicating critical information such as input channel number and output data units. An objective while creating a sensor object is to enable the same sensor functions and structure to be utilized on multiple samples of the same sensor, and to be able to transfer the sensor data and parameters easily to apply filters or other operations.

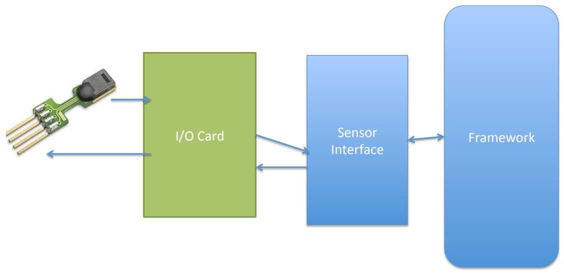


Figure 4.11: Sensor Integration to Framework

The sensors are handled by a sensor structure that contains the sensor name and two other structures. These structures are the Parameter and State structures. The parameter structure contains the units, the IO number and the conversion coefficient for the sensor. The states structure contains an array of the values of that sensor. The sensor structure can be utilized by functions. The setSensor function provides means to set the value of a sensor. The resetSensor function enables the resetting of a sensor such as an encoder. The readSensor function reads the necessary input and converts it to the necessary unit and places the value in the states structure Figure 4.12.

A SensorData structure housing the parameters and the states of the sensor

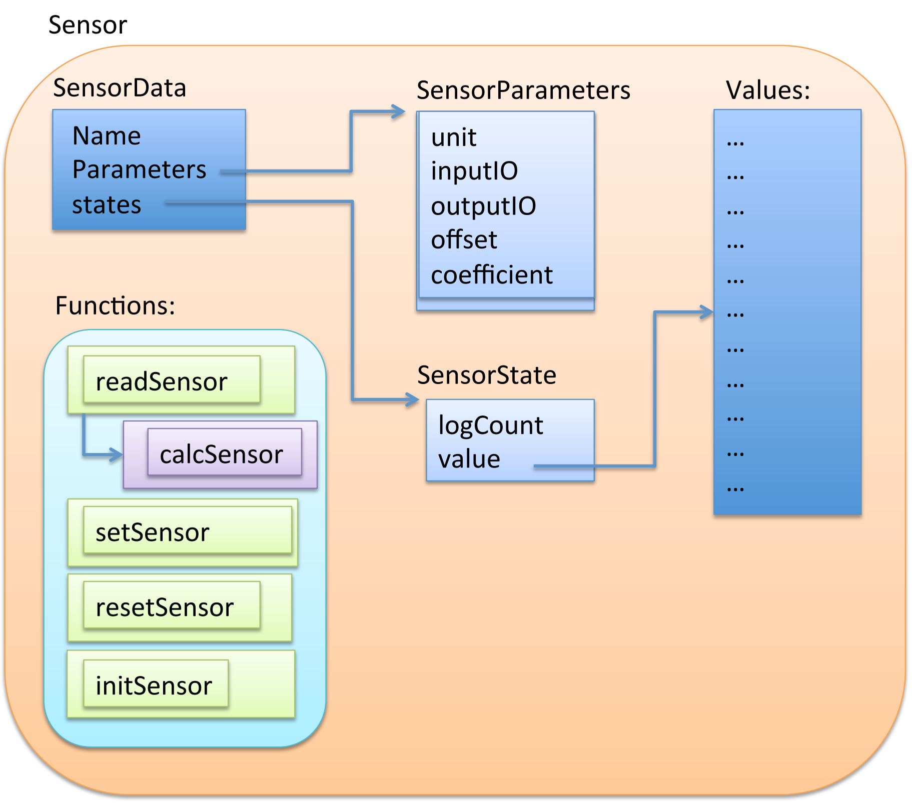


Figure 4.12: Sensor Functions and Structures

is created. This structure represents the sensor and its states. All the other functions work on this structure. The functions that initialize this structure, set its parameters according to the sensors specifications. Data contained within is separated in two parts. One part for the sensor parameters and the other part for the states of the sensor including previous values.

Storing the sensor parameters in a structure also enables the serialization of its data which renders it serializable i.e. ready for any transmission over a network or storage in a file.

The SensorState structure contains the current value and the past values of the sensor. Values are kept in an array to be able to perform filtration if required. The logCount value in this structure indicates the number of values that are stored in the sensor.

The SensorParameters structure contains information on the sensor and the necessary information to calculate the output value of the sensor using the electronic interface that is utilized. The parameters are the units of the sensor, the channel numbers for the input and output IO cards, the coefficient which the analog or digital IO card value to be transformed into the value and the offset of the sensor before this transformation.

The calcSensor function is the function responsible for converting the input from the IO card to the sensor value using the parameters in the parameters structure.

The setSensor function is utilized when the value of the sensor output is known by other means. This is utilized for calibrating the sensor or resetting the sensor to some value if the sensor is of incremental type.

The resetSensor function is utilized for resetting a sensor. This procedure unlike the setSensor function, re-initiates the communication with the sensor.

The initSensor function is utilized for initializing the sensor and initiating the communication. In this function the IO cards are initialized and the communication with the sensor is started.

4.1.2.3 Actuators

Recent developments in mechatronics provide the realization of smart adaptive systems. The operating conditions are ensured by sensors when the mechanical subfunctions are controlled by actuators [117]. A mechatronic system needs the organic combination of the following elements: Processors to control the system, sensors to intellectualize it and to detect the environmental situation, actuators to employ the motion of the system [118].

An actuator is a type of motor operated by a source of energy. The working principle of an actuator is converting that energy into some kind of motion. Actuators can be classified according to the type of control energy: electric motors and drives, hydraulic drives, pneumatic drives, internal combustion hybrids and piezo actuators [119].

Electric Actuators: They transform electrical energy to mechanical energy. Different types of electric actuators are [119]:

- DC Motors
- AC Motors
- Linear Motors
- Stepper Motors

Electric actuators are widely used because they are easily integrated into electric control systems. Besides, electricity is more available than fluid power which requires pumps and compressors.

Hydraulic Drives: They use fluids to transmit power. A hydraulic drive is composed of three parts: the pump, the inverse pump or cylinder and the valve. Pumps are power generators and inverse pumps or cylinders are power drains. Valves are used for control. Hydraulic drives are preferred traditionally in high power applications as steel press, large-scale precision motion tables, steering, brakes, propulsion, transmission in mobile systems, aerolon actuation in aircraft and fin actuation in missiles/rockets [119].

Pneumatic Drives: They have the same fundamental working principles with the hydraulic drives except that the working fluid is replaced by compressed air [119]. Pneumatic drives are further explained in this section.

Piezo actuators: Electrically controlled actuators that can be integrated into electronic control systems are of extraordinary technical importance as they compose the core module of mechatronic systems. Piezoactuators are essentially classified within this major group. On the other hand, piezoelectric ceramics (PZT) provide a higher potential than the electromagnetic actuators, hence they constitute the fourth class of actuators [117]. Piezo actuators are further explained in this section.

Some of the experiments realized using the framework include DC motors, linear actuators, pneumatic actuators and piezo actuators.

DC Motors: DC motors are rotary actuators that are powered by electric current. [119]. There are many kind of DC motors like DC Servo motors, Permanent Magnet DC motors, Brushless motors etc. DC motors are used in cases requiring accurate position and velocity control. Low noise and high efficiency are two important advantages of them [119].

Linear Actuators: The slider (rotor), the stationary part (stator) and the gap are extended in a straight line. These kind of motors are relatively expensive and large for power output. They have a miniature and simple structure.

Pneumatic Actuators: They have the same fundamental working principles with the hydraulic drives except that the working fluid is replaced by compressed air. The major disadvantage is compressibility of air, leading to low power densities and poor control properties. The advantage of these actuators is that the compressed air is widely available and environmentally friendly. It is easy to install and maintain pneumatic systems. They are used in robot grippers, assembly operations, drills/cutting tools, suction and clamping, animatronics, grippers, subsea robotics [119].

Piezo Actuators: The high specific force and displacement need is the major cause of the quick development of compact actuators with a large displacement and high force. This kind of actuators are created to be used in control surfaces of small aircraft and helicopter blades or for active vibration control of aerospace and submarine structures [120]. They are used in medical devices as surgical robots/biopsy robots, treatment tables for MRI (Magnetic Resonance Imaging), fluid management in fusion/insulin pumps, mammography; in automotive applications as camshaft adjustment, exhaust gas recirculation (EGR), adaptive spoiler, weight compensated trunk deck, seat adjustment/window lifters, variable valve timing (VVT); in automation devices as precision valves, positioning drives, pick and place automation, microdosing systems, rotary modules for robots, micro production and assembly, extreme condition remote handling; in robotics application as personal assistants for handicapped people, high precision welding robots, human machine interface with force feedback; in aviation/military applications as servo valve/electro-hydraulic actuator, flight control surface actuation, positioning/adjustment of surveillance of reconnaissance systems, antenna adjustment; in optics applications as beam steering, adaptive optics [121].

The usage of piezo actuators provides fast response, high stiffness and prevents backlash and friction [122]. These actuators consist of diverse layers of ceramic material, diverse layers of conductive material scattered between diverse layers of ceramic material and a plate attached to an end of the actuator. A piezo actuator includes an overhang portion [123].

Actuators in a motion control system are means of providing motion in the physical world. The desired outputs of the actuator such as *torque*, *velocity* or *position* are generated in the software and are transmitted to the electronics domain by IO cards. These IO cards usually have electronic drivers connected to them, to amplify these signals to create signals with enough power to drive the actuators. From a software perspective, an actuator is intended to provide the necessary physical action based on the input provided to it. From a logical perspective as far as the software is concerned the actuator function should receive a reference for the desired output and respectively, passing through the IO cards and the electronics and the motors and reaching the physical world as the output of the motion control system.

Output of the actuator, or the reference which the actuator can obey, depends also on its driver, more precisely the availability of a controller inside its driver. Such drivers are capable of providing references that usually require the control layers of the framework. However intelligent devices, or devices that have their own micro controllers can be directly controlled with the actuator interface. Actuators that only have a power stage receive their input by the actuator interface but they necessitate a sensor and the control layers to obey references such as position or velocity references.

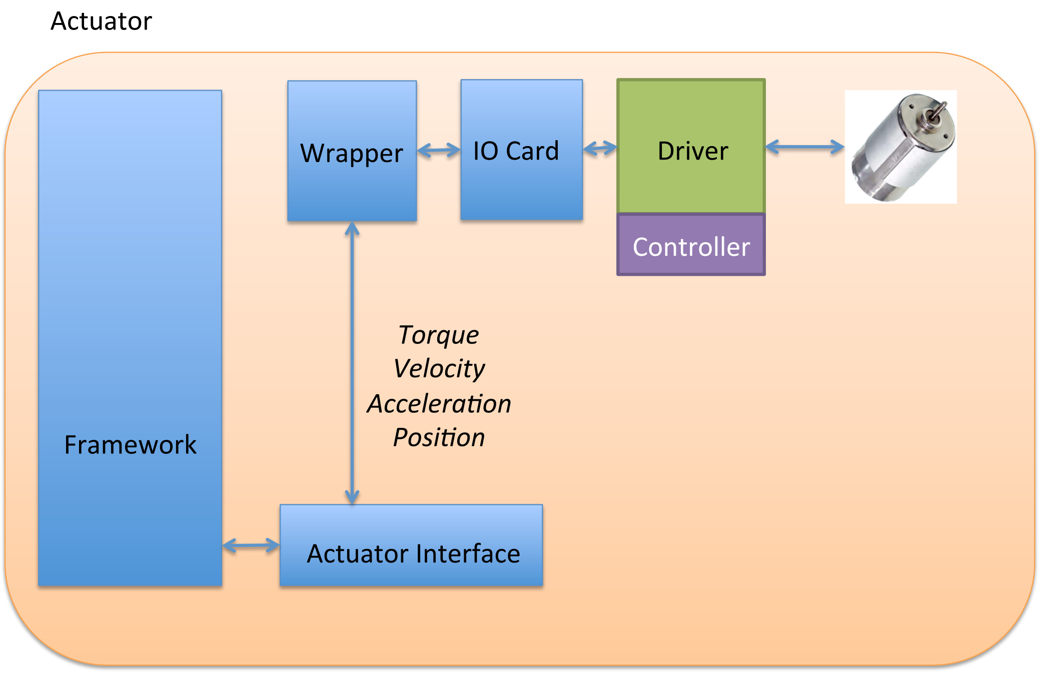


Figure 4.13: Actuator Integration to Framework

Each actuator function utilizes an actuator structure where the parameters of the actuator are stored. The actuator parameters consists of the output channel number and the input units of the desired output and the conversion factor. The conversion factor of sensor is the value that is necessary to convert the digitized data to the real world desired output. This factor includes the drivers coefficients along with the motors parameters and also the mechanical parameters associated to that particular actuator. An actuator usually has a single type of output depending on its type and availability of the driver. In general for dc motors this is torque, for step motors this is position for linear stages with drivers it can be velocity or force. The actuator architecture here works with a single type of output. When the actuator is coupled with means of measurement and control does it become an *axis* which can be commanded to obey different references such as acceleration, position or velocity.

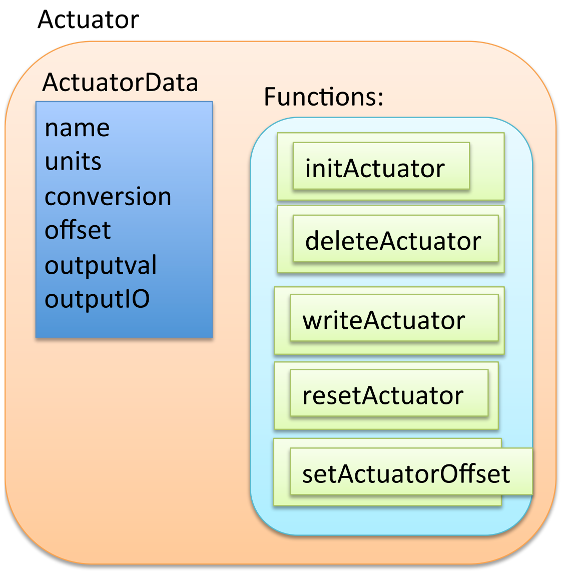


Figure 4.14: Actuator Functions and Structures

The information to operate an actuator can be grouped inside a structure that takes the following form. The ActuatorData structure, contains information relating to an actuator. This information is as follows: *name* is the name of the sensor and this data is used for generating text based report messages of the sensor. The *units* data field is also utilized for reporting of the actuators status indicating the units that the actuator outputs. *conversion* factor is the coefficient which transforms the input indicated in the *units* field to the necessary voltage value or other value sent to the actuator to achieve the desired output. The *offset* value indicates the offset that is to be sent to the driver for the actuator. This value is utilized for resetting the stage or for compensating fixed loads or imperfections in the electronics for the actuator. The *outputval* field contains the output value for the actuator; this value is to be utilized to check what value is being sent to the actuator by other functions and structures. The *inputIO* field contains the output data channel number for the actuator.

Other functions utilize the data contained within the actuator structure. The *initActuator* function takes an ActuatorData structure and performs initialization of the structure and the necessary IO channels.

There exists a *deleteActuator* function, that frees up all the resources allocated to the Actuator function, and stops communication with the IO cards if necessary.

The *writeActuator* function is responsible for providing the necessary output to an actuator. As input it takes *value* in the units defined in the ActuatorData data structure. This value is then offset by the offset value and transformed by the coefficient to make it compatible with the output units and then output from the output channel defined in the ActuatorData structure.

Actuators sometimes need to be reset in case of errors or various other needs, for this the resetActuator function resets communications with the actuator if necessary and deletes the offset value.

The output electronics or mechanics may poses an offset, this is handled by the $offset$ value in the actuator data structure. This offset value is set using the $setActuatorOffset$ function.

4.1.2.4 Filters

Series of data, also know as signals in the control domain, may not always exhibit desired behavior and can be distorted due to noise, discretization or other operations performed on the data. Therefore these data need to be filtered in order for them not to cause perturbations in the system.

Filters are necessary when certain data to be obtained has noise associated with it or in the case that derivation and discretization is involved.

When implementing a filter for software many aspects concerning the filter characteristics are important.

- Frequency Response
- Phase Shift or Group Delay
- Impulse Response
- Causality
- Stability
- Localization
- Complexity

When implementing a filter structure from a structured software perspective, other features come in to play.

- Modularity
- Reusability
- Data Structures
- Data Storage

The study of many filters shows that a filter needs the input value and previous values of the input i.e logged input, some filters also require a logged output. The algorithm of a filter and the parameters of the filter also come in to play.

Upon investigation of the filters, it was discovered that filters require tuning and often the utilization of different filters. Therefore the design requirements for the filters was the modularity of filters and the necessity to create the filters as an exchangeable component that can be exchanged easily. In addition, filters needed to be coded in a standardized method and structure so that different filters can be called by the same function. To standardize a filter function set and data structures, the following structure for filters has been devised. The filter parameters are stored in a structure and this structure is initialize by an $initFilter$ function. This function does not necessarily need to comply to strict standards, as it will be called once during the initialization of the filter and automated calling is not necessary.

The filter structure consists of a name for a filter that is useful for generating automatic status messages from the filter and an array of parameters. The number of parameters is defined as $PARAMCOUNT$. The parameters are coefficients and constants utilized in the filtration algorithm. These are fulfilled by the $initFilter$ function.

The $initFilter$ function takes as parameters the filter structure, and the parameters that are to be filled in to the filter structure. These are copied one by one in to the filter structure.

The deleteFilter function is utilized once the filter is no longer necessary. This function releases all resources allocated to the filter.

The $FilterAlg$ function is the actual algorithm of the filter. This function utilizes the filter structure to receive its parameters. In addition it receives an array of input values, the values to be filtered and the output of the filtration operation, i.e. the filter output. The algorithm is implemented within this function. Keeping the function declaration in such a standard, the filter can be automatically referenced as a function pointer.

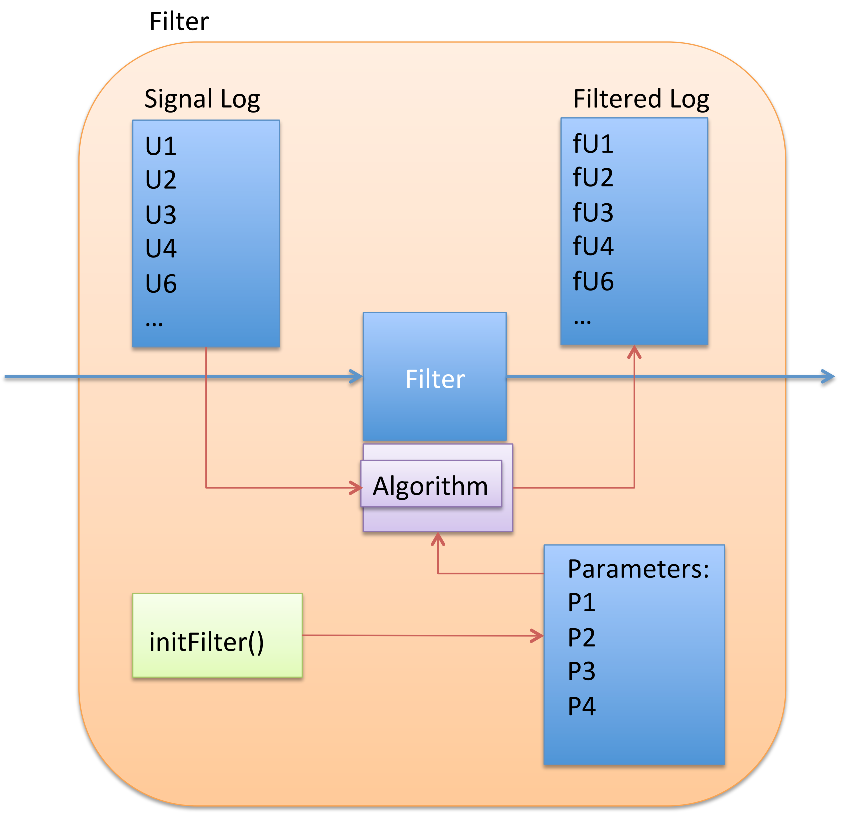


Figure 4.15: Software Implementation of Generalized Filter

4.1.2.5 Estimators and Observers

In motion control systems it is not always possible to have sensors that measure every aspect of the desired motion and the desired interaction with the world.

Signal
Filter
Filtered Signal

Figure 4.16: General Filter in Mechatronics

However using observers it is possible to estimate desired information. For example the first derivative of position is velocity. In a system where the only means of measurement is the encoder of a motor which provides position information, an estimator is utilized to obtain the velocity information. The algorithms used to obtain this information are called estimators. Several types of estimators have been included in the framework.

4.1.2.6 Observers

Observers are algorithms that combine sensor outputs with knowledge of the system to provide results superior to traditional structures, which rely wholly on sensors. Observer control based algorithms require an observer. A state observer typically combines system input/output with a mathematical model to predict the behavior of that system. Real state of the system is compared with the estimated state of the system and resulting error is utilized to compensate in the following cycles to bring the system to desired states.

4.1.2.7 Controller

Controllers are algorithms that receive a state and a reference and generate outputs to drive the state to the desired reference. In the literature there are several control algorithms that have specific applications.

The controller module of the system takes the states that are to be controlled, applies its control algorithm and outputs the control outputs. The inputs to the controller module may be states of the system that are observed, estimated or measured by the sensor module 4.17. The controller module implements the standard module structure.

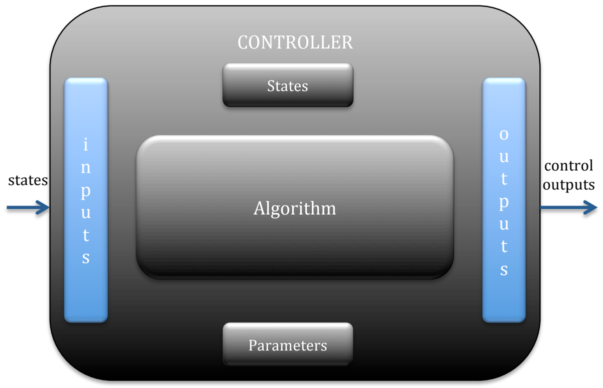


Figure 4.17: Controller Structure

4.1.2.8 Axis

In the context of the framework the control of a degree of freedom is is named an *axis*. The control of such a degree of freedom consists of acquiring the inputs from the sensors, utilizing the necessary estimators and observers to obtain unmeasurable states of the system then utilizing control functions to control the degree of freedom. And finally the control outputs are sent to the actuators utilizing the actuators functions 4.18.

The axis module abides by the module structure and interactions. And it has parameters and states structures to store its data and input and output structures to receive data during the runtime of the system. In order to control the actuator using the sensor information and the references from its inputs the axis needs

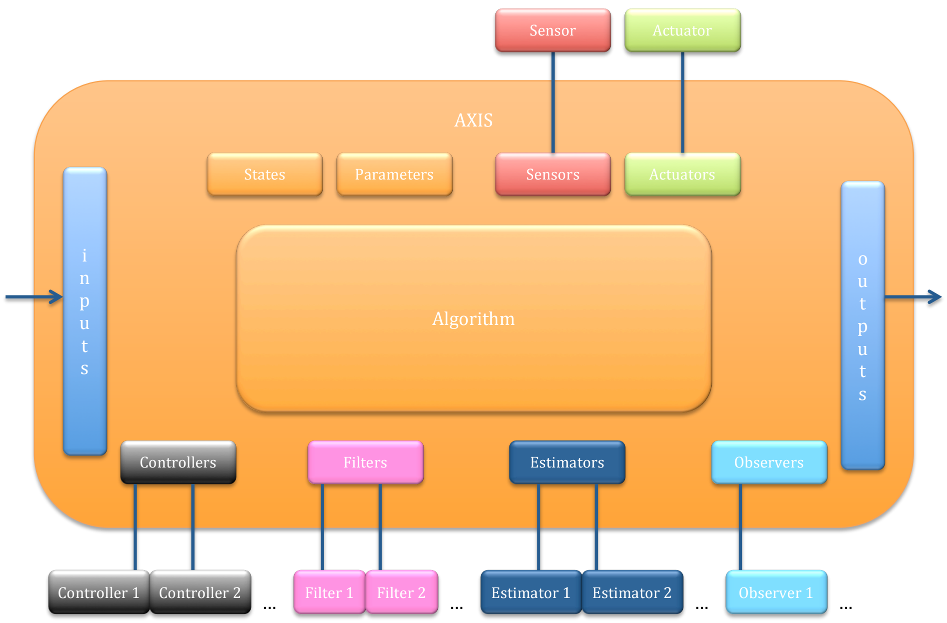


Figure 4.18: Axis Structure

access to several of the other components. The main components of the axis are the sensor and the actuator as the primary function of the axis module is to govern the relation between these two structures. To be able to govern this interaction in its *Algorithm* the axis also has access to:

Filters Used to filter inputs from the sensor or its references.

Estimators & Observers Used to obtain data that may not be provided directly from a sensor.

Controllers Used to implement the control algorithms

An axis structure has access to several of the submodules of the system as it may require may different filters of observers to obtain different informations. If the axis has a complex control structure then many controllers may be needed to be implemented to control it as described in section 5.1.3.

During the runtime of the system, another module using an axis module may also switch the controllers or other structures within the axis module. The system may require the axis to function in position controller mode during a certain period yet later may need to switch the axis in to velocity control mode as described in section 5.1.1

Initialization or creation of the modules used by the axis is done in the system initialization phase and the deletion of these components is also handled by the systems deletion phase where the resources such as IO cards are released.

4.1.2.9 Mechanism

The mechanism module is the combination of several axes, sensors and actuators to form a motion control device such as a robot. The mechanism module has the roles of insuring the coordination between the different axes and providing and interface for the mechanism. The mechanism module also abides to the standard modes of interaction by having an input and output structure and storing its parameters and states within. The objective of grouping the axes inside a mechanism is to expose an interface that only provides functions relating to a mechanism to the higher levels. For example a system using a delta robot would only be interested in giving position references to guide the robot and position information to check on the robot. The other parameters or functions of the mechanism may not be relevant to the higher levels of the system.

The multi degree of freedom mechanism has special needs expected such as forward kinematics and reverse kinematics and trajectory generation expected from it. The mechanism structure must provide access to the functionalities as well. This is achieved by providing arrays of these modules in the mechanism module

Figure 4.19.

Actuators array provides access to multiple actuators to the mechanism. Simpler actuators may be added directly to the mechanism in case they are not linked together by an axis module. Such as solenoid valves or stoppers.

Sensors array provides access to multiple sensors if they are not part of an axis structure. An example to a sensor used directly by a mechanism module would be a tray sensor that reports the presence of a tray at a certain position as described in section 5.1.2.

Axes array provides access to multiple axes. It is the mechanism primary task to perform the synchronization of these submodules.

Kinematics array provides access to kinematics modules. Kinematics modules are used by the mechanism module to convert the task space references in its inputs to joint space references in the axis inputs. Such as for the control of the parallel robots described in section 5.1.2

Trajectories array provides access to trajectory modules. Trajectory is used to move the mechanism smoothly and in a linear fashion from its actual position to its reference.

At the heart of the mechanism is a state machine that governs the operation of the mechanism, the states of the mechanism are exposed to the system level. Generally a mechanism is expected to perform in several different modes of operation such applying a certain force or going to a specified position. The state machine of the mechanism module governs the operation mode of the mechanism.

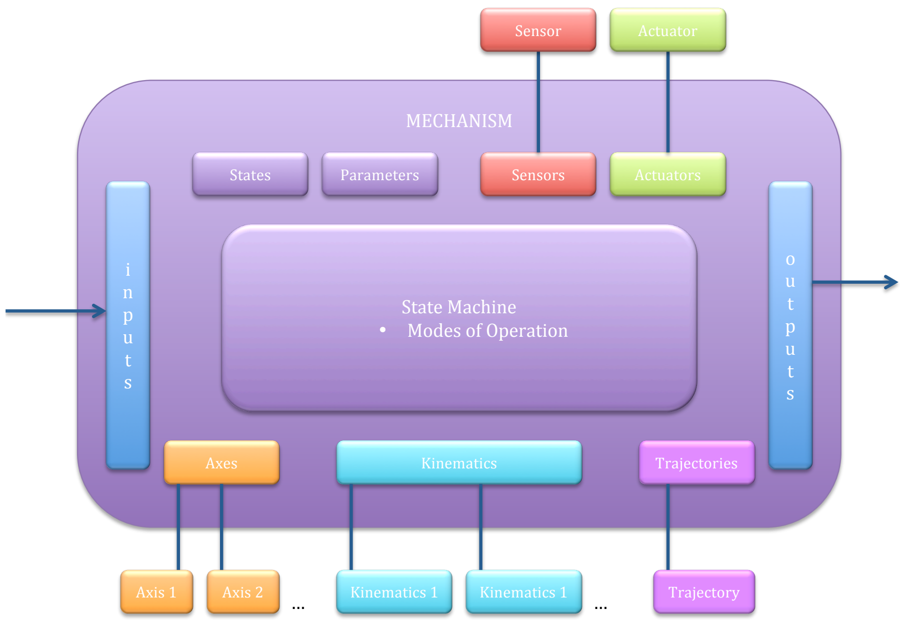


Figure 4.19: Mechanism Structure

The state machine is operated by the system to change the modes of operation of the mechanism.

From an interaction perspective, the system can call upon an mechanism to run in the following modes of operation.

- Initialization
- Homing
- Setup
- Move

More modes of operation can be added to the state machine depending on the task requirements of the system and the mechanism.

4.1.2.10 Trajectory

Trajectory module has the task of providing a smooth movement from one point to another. Generating a trajectory has two major contributions. First it can smooth out any jerks in the movement of the mechanism by providing a slow acceleration and deceleration phases, secondly it can assure that the desired position is reached in a linear manner. The trajectory module has to receive its inputs from several degrees of freedom. The data it needs are the maximum acceleration and velocities of the different degrees of freedom as well as their positions and their references Figure 4.20. Then the trajectory algorithm is performed and the algorithm outputs references for each of the degrees of freedom. The trajectory module is used by the mechanism module and it is up to the mechanism module to gather and provide the necessary inputs to the trajectory module and use its outputs. Several different modes of trajectory may be implemented in a system. A jerk free and smooth trajectory is obtained using trigonometric jerk model [124] was is implemented. A three part S-curve was formed consisting of the acceleration, constant velocity and deceleration phases where the acceleration limits determines the sinus characteristics during the acceleration and deceleration phases and the velocity limit determines the speed at the constant velocity phase in 5.1.1

4.1.2.11 Kinematics

The kinematics module has the task providing a transformation between different coordinate spaces. It receives inputs in one coordinate space and outputs in another coordinate space. To function it needs parameters of the mechanism or robot for which it will perform the transformation along with the algorithm to perform the transformation. Kinematics involves the mathematical equations that

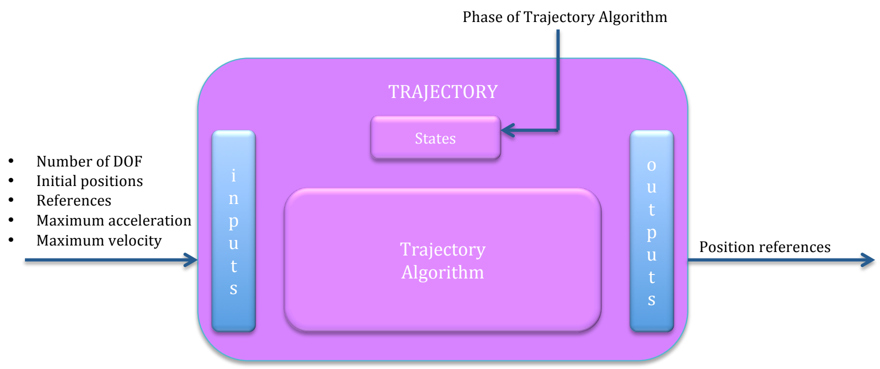


Figure 4.20: Trajectory Module

provides the transformation in between the joint coordinates and the task coordinates. In complex mechanisms, the desired motion references are generally given as task coordinates, but the software controls the actuators which are controlled in the joint space. Kinematics provides a translation between where the relevant part of a mechanism needs to go and where motors controlling that mechanism need to go .

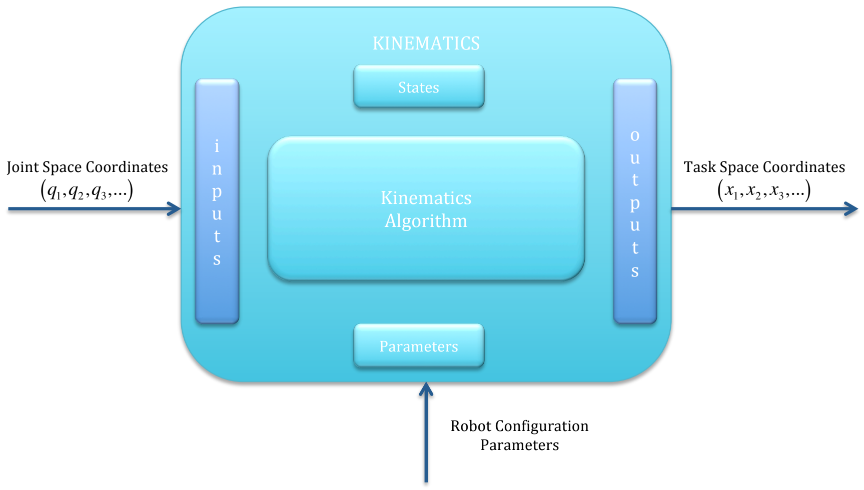


Figure 4.21: Kinematics Module

Inverse kinematics provide a translation between the joint space and the task space. Inverse kinematics are utilized to calculate the position of a mechanical structure based on the positions of its actuators.

4.1.2.12 Protection

In some cases it is desired to put certain limitations on some aspects of a system to protect it. For example an actuator may be only capable of accepting a maximum input. Protection may be added in all the modules of the application from actuators to mechanisms. In the example of

4.1.3 Process

In the context of this framework, a process is described as an operation that is to be performed by the designed system. These processes include heating, curing, cutting etc. Processes have several parameters. In the implementation a process is very much like a mechanism. In the sense that it utilizes many actuators and sensors. The process also utilizes a state machine to govern the process. The states of the process are time based or sensor based and switching between them is automatic.

4.1.3.1 Interpretation

Motion control systems may sometimes have the need to execute certain predefined motions. For milling, cutting or any type of application. For these applications a reference trajectory is needed. This reference trajectory is usually complex and cannot be defined using simple mathematics. The reference generated is derived from a certain object or a certain procedure. A language that describes a certain motion called g-code has become an industry standard. There exist many applications such as SolidCAM, Unigraphics among others that generate g-code from solid objects to other processes. This may be achieved using a g-code interpreter.

4.1.3.2 Parameter Setting

Certain devices require functions to initialize them and set their parameters. Templates for external devices exist to set their parameters and initialize them. The common parameters included are, power, speed, temperature, wavelength. Parameter setting task has a close affinity to the communication module. Parameter setting in the framework is implemented in two different manners. Either the device whose parameters are to be set is implemented as a sensor or actuator and it is initialized in the corresponding initialization procedure. In the case that the external device has communication over a serial port or such, integration in to a realtime loops of the framework is not possible. In that case it is added to the communication module. Parameters for these devices are received through the communication structures.

4.1.4 Communication

Usage of different platforms or multiple platforms in a motion control system brings forth means of communication among them. Communication requires that the communicating platforms share an area of communicative commonality.

Communicating the system with other systems or man machine interfaces is an important part of the process for the functioning of the system. There are very many communication mediums and protocols available. Therefore the communications must be separated in to two tasks, data collection and data reception and transmission. The data collection procedure for the system is setup as follows.

The data to be communicated has to be separated in two parts, incoming data and outgoing data. Outgoing data is collected at the end of every loop whereas incoming data is input at the beginning of every loop. The initial phase of designing the communication is the creation of two structures, one for the incoming data and another for the outgoing data. Outgoing data is usually the position information and the states of the actuators and sensors but any part of general system can be added to this structure whereas incoming data are the references and the states for the state machines of the system.

Creation and population of these structures enables the separation of the communication problem from its data. And the communication problem can be assigned to the person or team dealing with the communications.

The philosophy of separating the implementation of the system from its specific hardware also applies for communication and the communication procedures are also wrapped with two functions. These functions are the *send* and *receive* functions.

Communication can be diversified and the framework permits this diversification by including a message ID for the messages to be sent. This message ID has to be interpreted by the communications designer to solve what is the intent of the message. It can be sent to different users or it can be of a specific type of shorter message that contains only one part of the output data structure.

Communicating with a motion control system does require large amounts of data to be transmitted and in the setups created using the framework direct memory mirroring was utilized. Copies of the input data structure and output data structure were created on both the recipient side and the sender side i.e. the motion control side and the MMI side Figure 4.22. The entire data structures were exchanged in regular intervals between the two components of the system. This method enabled the non realtime man machine interface to access the permitted parts of the motion control software as if the two were linked by a non-realtime shared memory providing transparency.

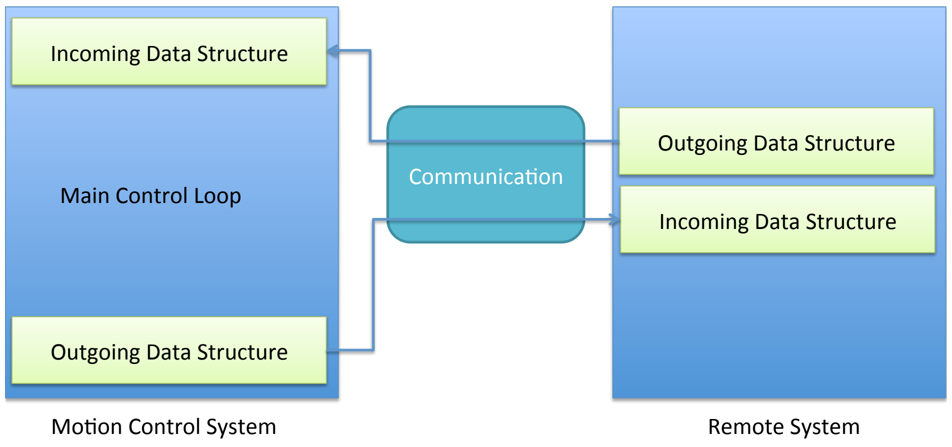


Figure 4.22: Communication Interface

The communication does not necessarily need to go over a network. The send and receive commands can synchronize the data to be exchanged to the same systems memory where no network is involved.

Many systems will require different communication needs such as synchronous or asynchronous communications. Some communications may require higher bandwidth and transfer of large structures may not be feasible. The communication method can be extended to send different commands of different types and receive different commands, each time sending or updating only a small portion of the exchange data structures.

This approach is compatible with the other frameworks such as ROS[107]. The communication needs to be initialized as a compatible structure or protocol, in this case a ROS node and the send command can be wrapped with the ROS's publish command to transfer data any recipients that might be listening for the message.

4.2 Man Machine Interface

The man machine interface of a system is the component of the system that interacts with the user of the system. The design of a man machine interface involves the definition of the interactions of the user with the system. The MMI is also a loosely defined aspect of the system that is most difficult to model as it can range from a full blown graphical user interface with a joystick to a simple start and stop button.

4.2.1 Graphics Display and GUI

The GUI is the visual software that enables the interaction between the operator and the system. It consists of main functional blocks that allow the operator to observe and intervene the features of the system using the graphical or numeric features and command input blocks that are presented on the GUI. In the framework the MMI structure is also loosely defined and mostly up to the designer to create and configure. An MMI may include mice, joysticks, haptic devices keyboards and screens. Man machine interface is a research topic all in itself. In the context of this framework, non realtime components of the motion control task are also lumped to the MMI side of the system. This is mainly due to the fact that the MMI being a non realtime system has looser constraints and therefore is simpler to program.

4.2.2 MMI Device Driver

A motion control system may necessitate the use of many external devices. These devices may be realtime motion control devices such as motion controllers or non realtime devices such as illumination systems. If these devices do not require the realtime communication, i.e. they are not in a broader realtime loop then they may also be controller by the MMI software. There already exists many tools for the creation of man machine interfaces in applications. The most widely used application, and more importantly the one with the largest user community is Microsoft's Visual Studio. Visual Studio has means of adding buttons, sliders and displays among other components to the MMI application. Another reason for this choice is the wide adaptation of the .NET framework by device manufacturers. Most devices have a device driver for Windows OS and some even have an API for the utilization of their devices. These factors greatly improve the ease of MMI creation.

4.2.3 Image Acquisition and Processing

Vision systems and image processing systems are also considered soft realtime systems and they are also handled by the MMI software. Vision systems are also a complete field of research by themselves. The vision system of the motion control systems can be considered as a vision sensor figure 4.23. In other words they are sensors that produce position, velocity, stiffness or other data. There are several frameworks that handle image processing such as OpenCV[125] or Halcon[126]. The framework can be and has been utilized with either computer vision library to integrate cameras as vision sensors to it.

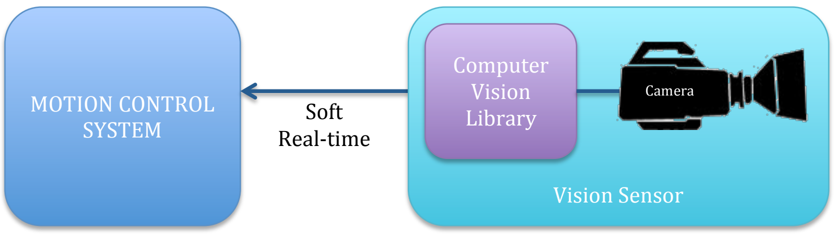


Figure 4.23: Vision Sensor

4.2.4 Communication

There are some aspects of an MMI that can be considered indispensable to most motion control systems applications that are also lacking in the MMI development tool previously mentioned. In the case of this framework the MMI must have means of communicating with the motion control system. This is achieved in a symmetrical manner to the real time motion control software. Two structures, one for receiving data and one for sending data are passed over a medium to the motion control software. This enables the MMI to partially have access to a certain portion of the realtime systems memory albeit in a non realtime manner. The positions and states of the state machines of the motion control software can be observed and manipulated by the MMI hence the user of the system.

4.2.5 Scripting

Another almost indispensable component of a motion control system is the ability for the user to create programs to automate the motion control system. This feature of software applications is called the ability to process scripts. For this purpose scripting was added to the man machine interfaces by means of reading script files. The scripting of the system was performed using regular expressions to parse each line of the script file. The script parser function would take lists of parameters which were the lists of commands in the script separated in to groups depending on their number of parameters. This implementation although tedious worked exceptionally well. However an alternative and more versatile method later became adopted. The css compiler of the .Net framework was utilized to compile C# code that would act as the script on the fly. This approach enabled the functions already available in the MMI to be scripted on the fly by the MMI application giving the users of the system ability to automate the motion control system through scripts on the MMI. The .Net Framework ships with a C# code compiler that lets you generate in-memory assemblies. This gives the ability to dynamically modify code during runtime. Most scripting languages give you a function that allows you directly to evaluate a block or raw string of code as soon as its encountered. As C# is a compiled language the C# code needs to be compiled into an assembly before it can be used. And then classes from the compiled code can be instantiated directly from the assembly. C# code can be compiled on the fly with an instance of the CSharpCodeProvider class. Additionally, C# can create an instance of the CompilerParameters class, which contains a collection of parameters that will be used when compiling the code. In the example below, it is demonstrated how to create a new C# compiler and a set of parameters that will compile the new assembly in memory. It also commands the compiler to include System.dll as a reference assembly:

```
// Create a new instance of the C# compiler
var compiler = new CSharpCodeProvider();

// Create some parameters for the compiler
var parms     = new System.CodeDom.Compiler.CompilerParameters
{
    GenerateExecutable        = false,
    GenerateInMemory          = true
};
parms.ReferencedAssemblies.Add("System.dll");
```

Once a C# compiler has been created, it can be used to compile raw source into an assembly. CSharpCodeProvider allows code compilation from a variety of sources. In the example below, using the CompileAssemblyFromSource method to compile a code directly from an array of strings is demonstrated. This string is typically CompileAssemblyFromSource will look at the code provided and return an instance of the CompilerResults class.

```
// Try to compile the string into an assembly
var results = compiler.CompileAssemblyFromSource(parms,
new string[]
{@" using System;
 class ManipulationScriptClass{
public void ASSEMBLE_PART(int x)
{
// Code to perform assembly
}
}"});
```

One thing to note is that the compilation method will complete regardless of whether or not the code has compiled successfully. To make sure the code has compiled, a check is needed for the Errors collection that is part of the CompilerResults instance returned by CompileAssemblyFromSource. If there were no errors, the code was compiled successfully and the assembly can be used.

Once the code is compiled into an assembly, it can be used as an assembly to create instances of classes from the source code written and using reflection to invoke methods and get/set properties of those classes. In the example below, creating an instance of ManipulationScriptClass and storing it as an object is shown. Reflection is then used to invoke the ASSEMBLE_PART method on the class.

```
// If there weren't any errors get an instance of
//"ManipulationScriptClass" and invoke
// the "ASSEMBLE_PART" method on it
if (results.Errors.Count == 0)
{
var script = results.CompiledAssembly.
                    CreateInstance("ManipulationScriptClass");
script.GetType().GetMethod("ASSEMBLE_PART").
                    Invoke(script, 3);
}
```

Scripting languages make it much easier to accomplish automation of systems and enhance the versatility in the generation of dynamic motion control applications.

4.2.6 Devices

Devices can be added to the general motion control system in several ways. If the devices are standard components and have soft realtime constraints then they can be added to the MMI through its OS using its drivers. However if the device does have hard realtime constraints or it is a custom made device that does not have drivers and therefore cannot be integrated to the OS of the MMI then it can be decomposed in to its sensors and actuators and integrated in to the motion control system as if it were a mechanism of the framework. The integration of a position and velocity sensor used as a man machine interface is described in 5.1.3.

4.2.7 Data Analysis

In a production unit or system, for the optimization of the process there are measures showing the efficiency of the system. The production rate, quality are some of the factors representing the efficiency. The data extraction from the system can be maintained by necessary sensors or devices and this layer provides functions that transform the raw data to meaningful measures or graphs showing the performance of the system.

4.3 Putting it all together

To create a system, initially all the modules of the system must be described. This is done by using the corresponding files of the framework and performing the necessary configurations. For the definition of a new component initially the type of the component must be chosen. This can be an actuator, a sensor, a controller, a mechanism etc... The data structure for these is fixed providing a standardization between all the same types of components. The configuration of the standard data structure to meet the specialized needs occurs by defining the contents of some of the data fields and the creation of functions that modify them. There are some functions that have to be created such as initialize, loop and delete as these are expected by other components of the system interacting with it. In addition to the required functions, functions that enable the parameter configuration of the system or other aiding functions are also defined in to the customized function list.

This can only be achieved after the system modules have been defined. After this stage, all the modules that will be accessed from the top level are added to the system. The modules of the lower levels do not need to be define as they are already defined. In other words, if the system has access to a single actuator it must be defined in the system level. If the actuator is part of a mechanism or an axis, it does not need to be created as it already exists in one of the higher levels that must in turn be defined.

Creation of the different modules and submodules is done on the system level, also the configuration of the modules is done on the system level.

5 IMPLEMENTATIONS AND EXPERIMENTAL RESULTS

5.1 Validation of the Framework

In this section test results of the framework are given on a micro assembly workstation, a micro factory and delayed haptic system. The framework is implemented on several different systems namely an industrial x86 PC running realtime linux, a dSPACE platform and an FPGA. In the case of the latter two, software is ported from one platform to another to test the platform independence of the system. Different types of modules are created for the systems and reused when the same components are utilized in other systems by reconfiguring them as necessary. The modules are linked together and the motion control systems are brought to life. Then the communication is established between the systems and their respective MMI's where several different means of communication are used such as TCP/IP, CLIB and RS232. The implementations demonstrate that the framework is capable of modeling and implementing motion control systems independent of hardware.

5.1.1 Micro Assembly Workstation (SUMAW)

In this section a Micro Assembly Workstation that was built using the framework is presented. Initial work on the motion control library began with a three year project to build a micro assembly work station. The project evolved over several different platforms for computing and the hardware and requirements were changed several times. The aim of the project is to create an open-architecture, reconfigurable micro assembly workstation for efficient and reliable assembly of micromachined parts. The software architecture and system supervision are presented. The motion control and system peripheral requirements are discussed and the software and programming of the workstation is described. The system is designed to be a versatile tool to study the problems in micro assembly and micromanipulation which are still not fully investigated. The computer configurations used for real-time and man machine interface are presented. The communication between these two parts is investigated and the methods for creating the real-time and non real-time software is explained.

5.1.1.1 Design Overview

The workstation is designed to be used as a research tool for investigation of the problems in micro assembly with reconfigurability and adaptability to perform diverse tasks. The development of the workstation includes the design of a manipulation system consisting of motion stages providing necessary travel range and precision for the realization of assembly tasks. The motion stages consist of 2 manipulators with 3 degrees of freedom (X, Y, Z) and a sampling stage with 3 degrees of freedom (X, Y and Rotation). The manipulator holders have been designed so that the manipulators angle of approach can be adjusted. The manipulation tools can also be changed with ease enabling the system to perform predefined tasks and adapt to new ones. For tasks that require very precise operation, piezo actuators can be inserted between the tool holder and the end effector. A vision system has been created to visualize the microworld and to determine the position and orientation of micro components to be assembled.The vision system consists of a microscope, with a focus and zooming system equipped with Fire-wire cameras for coarse and fine image capture and illumination systems to illuminate the parts from top and bottom. The overall control and supervision structure has therefore the task of controlling motion stages in real time and synchronizing their movement, presenting to the user their individual positions, adjusting the microscope and capturing the images from the cameras and presenting them to the user. The overall control and supervision structure also has the task of reading the commands from the user in forms of mouse clicks on the screen, or joystick movements or scripts written and then process these commands and execute the desired motions or automated tasks. This structure is implemented as a robust real-time control system in the form of an industrial PC and a graphical user interface that permits the control of all the stages in the form of a PC Fig.5.1. The system is able to perform robust motion control of its manipulators with sub micron accuracy which translates to maximum one encoder pulse control of the stages.

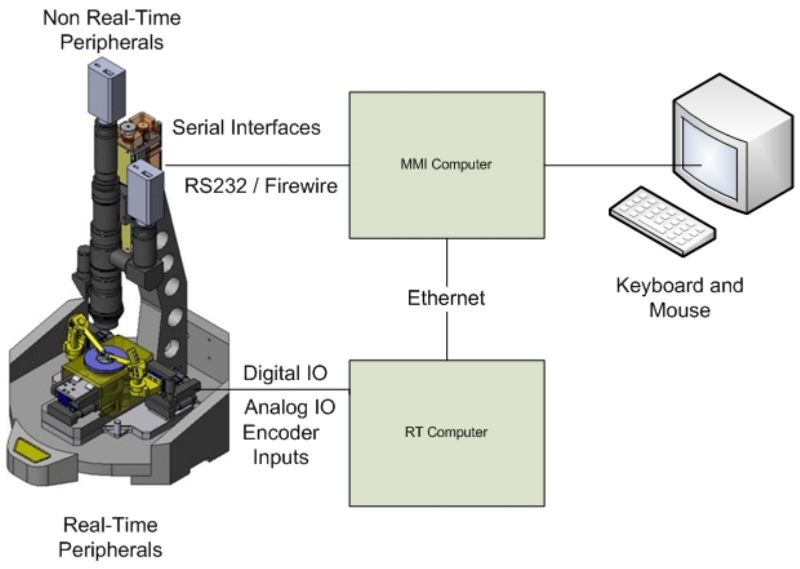


Figure 5.1: General System Layout.

The MMI computer performs data presentation, image capture, image processing whereas the RT computer operates to carry out trajectory calculations and motion control. These tasks are discussed in detail in their respective sections.

5.1.1.2 Hardware

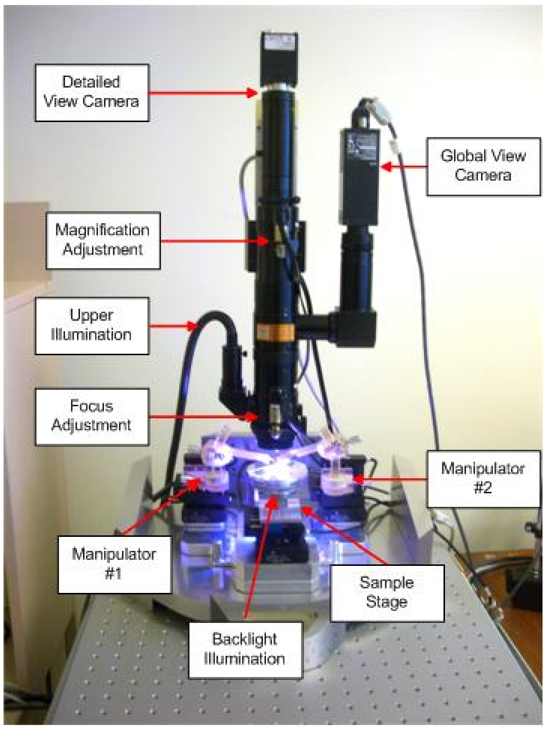


Figure 5.2: SUMAW Components.

The hardware that interacts with the assembly consists of end effectors which can be in the form of probes or grippers, tool holders, which are used to hold and do the non-actuated positioning of the probes and grippers and stages to actuate the manipulators. Each manipulator has 1 end effector 1 tool holder and 3 micro stages configured in a Cartesian (X, Y, Z) configuration to provide movement. The system has 2 manipulators and a sample stage. The sample stage has 2 micro stages that provide X and Y movements and a rotational platform that provides rotation Fig.5.2.

The supporting hardware for the system are for illumination and optics. The illumination hardware consists of LED based backlight illumination halogen based upper illumination that is transported by fiberoptic cables. These modules permit the change of intensity in case of the halogen light and color in the case of the LED light. There is also a microscope that is used to magnify the objects and cameras that capture the images from the microscope. The microscope assembly is actuated in the Z axis with a belt driven linear mechanism to adjust the safe working distance and give flexibility on the size of the sample used.

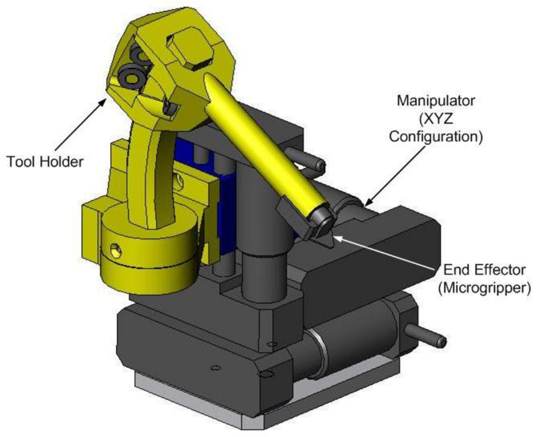


Figure 5.3: Manipulator Assembly.

The interacting elements of the system, end effectors such as micro grippers (Zyvex, FemtoTools), Atomic Force Microscope (AFM) probes (Veeco), micropipettes, tungsten probes (Zyvex) etc. are connected to the tool holder. These end effectors have amplifiers if the tool is actuated or drivers if the tool performs measurement. The drivers and amplifiers either amplify the micro volts that are generated by the end effectors, or convert the reference voltages to micro volts or micro amps for the actuators. These are interfaced to the analog and digital inputs and outputs of the IO cards. The tool holder Fig.5.3 is manufactured in a fast prototyping machine. It is designed with the ability of manually adjusting the interaction angle according to the task. These also have a base that permits the manual rotation of the tool holder.

The tool holders are coupled up with motion control stages (PI M-111.1 DG) which are configured in a Cartesian configuration of X, Y and Z axes of the manipulator. These stages contain 12V, 2W DC motors inside and are driven by a current driver that has been manufactured for this project and that is connected to the analog output of the IO card which has +- 10V output. The motion control stages have an encoder resolution of 7nm and a working distance of 20mm. The encoders of the stages are directly connected to the encoder inputs of the IO cards which generate 2048 encoder pulses per revolution. The limit switches of the stages are connected to the digital inputs of the IO cards. Piezo based precise motion control stages can be added on these (P-611.3 NanoCube) providing a travel range of 120 x 120 x 120 μm and a precision of 1 nanometers. These are driven with a piezo controller used in amplifier mode (PI E-664). The piezo stages interface to the system over the ADC and DAC of the IO cards.

The sample stage has two micro stages with the same specifications as above, and for the rotation a dc motor driven custom design rotation platform, that permits light passage from under the stage with a gap opening of 20mm. This DC motor is also controlled by an analog output of the IO card that is connected to the current driver.

5.1.1.2.1 Electronics The electronics requirements for the system has been identified as follows:

10 DOF's require 10 encoder inputs, 20 digital inputs for limit switches and 10 analog outputs. For the manipulators it is desirable to do experiments with a wide range of end effectors, to achieve this, each end effector has an analog input channel, an analog output channel, a digital input channel and a digital output channel. For piezo stages one analog input is needed for measurement and one piezo stage is needed for actuation.

To be able to control the hardware with the desired accuracy, the following electronics have been selected:

The IO cards for this project are Humusoft MF624 cards that reside on the PCI bus. Each card has 4 32 bit encoder inputs, 8 digital inputs and outputs and 8 analog inputs and outputs, making each one capable of controlling 4 degrees of freedom, a total of 3 cards have been used to control the 10 degrees of freedom.

The current driver that converts the voltage signals from the IO card is described in [127]. This driver is used for all the degrees of freedom for manipulation and for the movement of the microscope.

The optics and illumination which do not require realtime control, have been controlled by their dedicated drivers. The Thales microscope's focus and magnification is actuated with step motors that are controlled with a Thales controller which have 12000 steps in their full range. Although the device was able to be used in the desired manner, it was necessary from time to time to home the microscope and then reposition it, as small step counts sometimes did not produce any movement. For further designs, a microscope system that has closed loop control for the optics is envisaged. The upper illumination device is a DCR III and provides illumination from the top by means of a fiber optic bundle, the LED illumination device is a RGB LED illuminator and is placed under the sample stage. The driver of the LED illumination device enables the switching of the 3 color LED's inside, the LED's shine directly through a filter under the specimen or manipulated object.

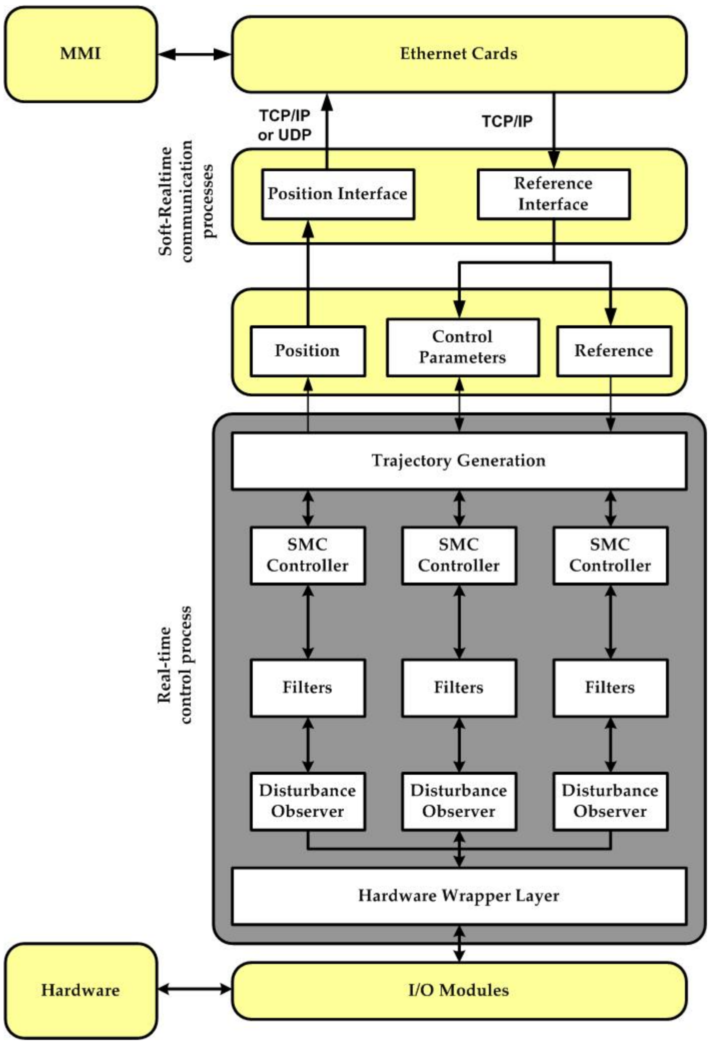


Figure 5.4: System Structure.

5.1.1.3 Software Overview

The aim of the software for the SUMAW is to enable the users to perform precise micro assembly tasks intuitively and also perform automated tasks using the system. The system has many tasks that need to be performed such as presenting data to the user, controlling actuators, acquiring images and running scripts. These tasks can be separated in two parts according to their timing constraints. For operations requiring strict timing constraints i.e. trajectory generation and motion control, a real-time platform is required. For operations such as interacting with the user, controlling non-real-time peripherals such as illumination and microscope functions a non-real-time yet easily usable and graphical user interface friendly platform is required. To ‘ this, all the interactions of the system were examined and classified according to their realtime requirements. Then the devices with dedicated controllers were examined if they required realtime communications or they were able to function as real-time systems that required non real-time references.

To achieve these requirements a computer capable of real-time operations was chosen to perform the motion control and time sensitive tasks. For the control of the MMI and non real-time tasks a windows based computer was selected because of the ease of programming and the availability of drivers for devices. Although the realtime computer has the sole task of performing the time critical operations, additional features are necessary to enable the reception of references and the transfer of status information. For these additional features the realtime systems was programmed with 3 threads, one for receiving information one for sending information and one for the realtime operations. These threads need fast communication with each other which was solved with a shared memory. These additional tasks can cause load on the system and interfere with the operations of the system. To assure that the real-time task is not interrupted, one core of the intel core 2 duo processor was dedicated to realtime operations leaving the other core for communications and the operating system. During the development phases the hardware platform was migrated several times and different IO cards were used, to facilitate this process and to harmonize the functions that access different IO cards, wrapper functions were developed.

To enable the usage of all aspects of the system the MMI has been developed. To provide intuitive operation point and click and joystick based manipulation was coded. In order to make the system more flexible and to be able to perform automation, scripting abilities have been developed Fig.5.4 . The system was calibrated and means of measuring the positions in the vertical axis was devised using the focus ability of the microscope. Tedious positioning tasks requiring manual labor were automated using image processing.

An ascii based communication protocol that is easy to debug was developed and the two computers were linked together with Ethernet to provide a fast yet flexible link that is platform independent. Ethernet can also be used to interconnect multiple systems over a wide area enabling the addition of different real-time computers for motion control or MMI computers for monitoring should expansion of the system be necessary.

5.1.1.3.1 RT Computer The purpose of the real-time system is to control the 10 degrees of freedom existing on the SUMAW and control the manipulators, individually as well as grouped motion movement. The real-time system is also capable of expanding its abilities and degrees of freedom with the framework. The realtime computer for motion control is an industrial PC with a dual core processor and IO cards as previously described. This computer has Slackware Linux on it. The kernel has been patched using the RTAI extension [128]. RTAI extension provides realtime abilities to the operating system by taking over interrupts and making RT threads non preemptible. The usefulness of this patch is demonstrated on a realtime system in [129]. Furthermore, the dual processor architecture of the processor has been utilized and one of the processors has been dedicated to the realtime operations. There are 3 main threads in the RT system, one for motion control, one for sending messages to the MMI and one for receiving messages from the MMI. These threads communicate over shared memory.

Inter-Thread Communication The communication of the 3 threads is achieved by the creation of a shared memory block that is mapped on all of the threads. This shared memory is also an effective means of transferring information from the non realtime threads to the realtime threads and vice versa. The shared memory is mapped as a C struct where every element is an array with size in accordance with the number of DOF's of the system. The shared memory is mapped on all processes. The shared memory does not distinguish between the direction of the information flow, however it is simple to separate this data in the direction of flow for ease of understanding. For example the motor position variables can be used in both directions, for informing the MMI of the positions of the motors, or resetting the position of the motor according to calibration or homing procedures. The following is the data transferred from the MMI to the RT computer:

From MMI to RT

- *Motor Position References* are position references that are generated by the user, position references are manually entered or they are interpreted from calibrated mouse clicks on the GUI of the MMI

- *Motor Velocity References* are also transferred from the MMI to the system, they are obtained from the joystick attached to the system.
- *Motor Gains* are used to adjust the output gain of the motors, these are used when testing new driver configurations.
- *Analog Outputs and Digital Outputs* are used for parts of the system that have not been fully integrated to the MMI, they are an easy way of testing new additions to the system. This information is usefull for debugging the system and also adds flexibility in the case a new tool is to be added to the system, the connection is already setup.
- *Tuning Parameters* are variables that have been added to the system purely for the ease of development of the system. They propagate from the MMI over the network to the real-time computer and then to the motion control thread. This pre-established channel enables the rapid injection of any parameter to the system without the need for re-coding and re-compilation of the whole system.
- *Start/Stop Commands* of the system are used to start the different degrees of freedom and other items attached to the system, this signal is sent once all the reference positions have been sent to the system.
- *Control Parameters* are parameters for the controllers such as the PID controller gains and SMC controller gains.
- *Enable Signal* for Stages are used to enable or disable the degrees of freedom.

- *Message Frequency* is used by the sending thread on the RT server, as the message sending frequency.

The following is the information that is sent from the real-time computer to the MMI:

From RT to MMI

- *Motor Positions* are the positions of the motors, that are read from the encoders and then converted to the appropriate values such as microns, mms, degrees with the necessary gains for the gearbox, pitch etc.
- *Analog Inputs* to the system are not always used by the MMI, but for utilizing new hardware, or hardware that does not need any RT processing they can be directly relayed to the MMI, this is also useful in debugging and fast addition of experimental hardware.
- *Digital Inputs* of the system can be read by the MMI. These are also useful when debugging the limit switches of the motors for example.
- *Scopes* are variables with a similar function to tuning parameters that can be inserted anywhere in the realtime code, and their values propagate to the MMI enabling them to be monitored.

.

5.1.1.3.2 Threads There are mainly 3 threads running on the system, besides the OS functions. All the motion control code is executed on a single dedicated processor which runs all the realtime deterministic tasks Fig.5.5. The other processor is used by the OS and the 2 other threads that are used for communication.

The realtime thread has highest priority in the system, and it is not affected by the hardware or software interrupts of the system, therefore realtime performance is assured.

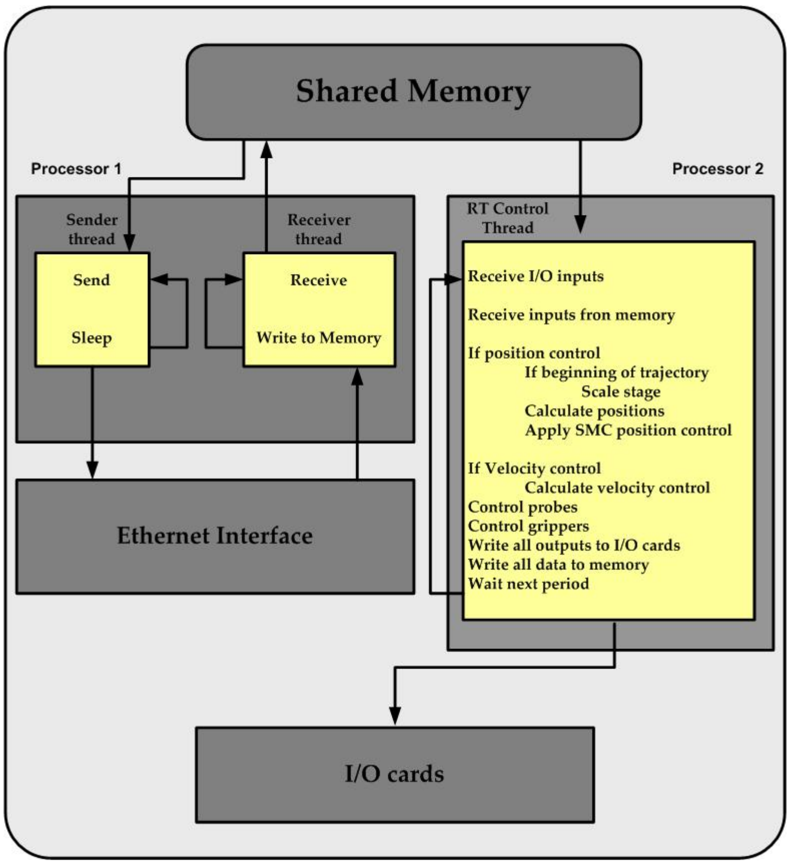


Figure 5.5: Thread Structure.

RT Receiver Thread The RT Receiver thread is in charge of listening to the network for new references from the MMI. The RT Receiver Thread resides in the non real-time, Linux OS part of the system, and therefore shares the processor with the RT Sender thread and the OS. The RT Receiver tread is started along with the RT Sender thread inside the RT Server Process. The RT process maps the shared memory and both threads can access the memory. The RT Receiver thread, which is the main thread of the RT Server process, first opens a TCP socket, then the shared memory is allocated. Then the process waits until there is a connection to the socket. After connection has been established, the RT Sender Thread is launched. Then the thread enters an infinite loop that is only interrupted by an Exit command from the MMI. In this loop the thread waits for a message over the network, when the message is received, it is parsed and the necessary data is written to the shared memory or executed in case of the Exit command.

RT Sender Thread The RT Sender Thread has the function of sending variables to the MMI. The RT Sender Thread starts up, and goes into an infinite loop. At the beginning of every loop, it generates a string containing all the elements {or just the selected ones} of the shared memory, with indicator tags in the beginning. This is later described in the communication protocol. Then the generated string is sent over the network and the thread sleeps until the next period for sending.

RT Motion Control Thread The RT Motion Control Thread implements the motion control framework. The motion control process starts up, creates the communication structures, initializes the actuators, sensors, mechanisms etc.After the initialization of the modules is complete, shared memory is created. Because of this sequence, the RT Motion Control Process has to be started before the RT Server Process. Then the IO cards are initialized. Then the RT Motion Control thread is launched and the thread works in a timed loop.

5.1.1.3.3 Hardware Interface The Hardware Wrapper template of the framework is utilized to map the functions of the Humusoft IO cards in order to provide the standardized means to the motion control actuators and sensors.

5.1.1.3.4 Motion Control System Configuration The major part of the operation of the system is performed with the formation of micro manipulation stages that are the combination of 3 liner stages. For each of the linear stages, two sensor modules were configured to receive the limit switch informations, one sensor module was configured to get the encoder input and transform it to position input and an actuator module to output to the motors of the linear stages.

For the XYZ stages two modes of operation were defined which were position control mode and velocity mode for use with a joystick. With this in regard a SMC controller module was added to the system. For velocity control a PD controller was utilized, however for this controller to function it required the velocity information which was obtained by a velocity controller. The actuators, sensors, two controllers and the estimator were grouped in the axis structure. Finally the XYZ micro manipulator was developed in to a working mechanism by integrating it in to a mechanism module where trajectory module was integrated to the mechanism to assure that a smooth trajectory was followed Figure 5.6. The mods of operation of the were set by programming the state machine of the mechanism to be able to home itself and to function in position references following mode and velocity reference following mode.

The main loop of the motion control system software is coded as follows.

In the main loop of the system, the references and modes of operation are received using the communication interface. This information is received in to

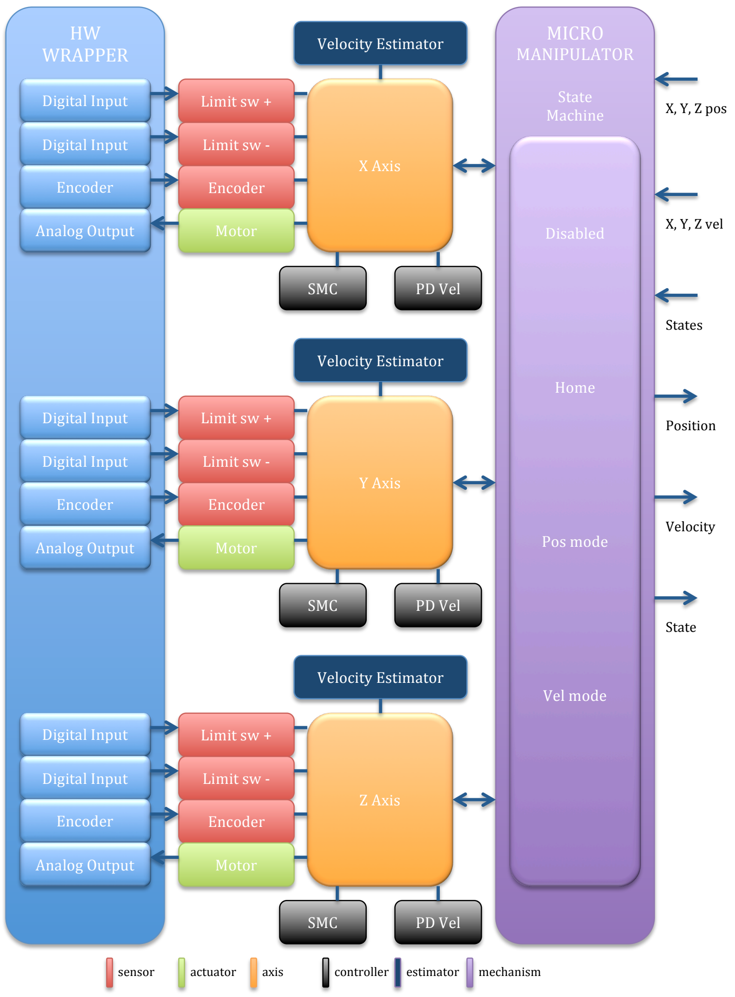


Figure 5.6: XYZ motion control stages modular implementation

a communication structure called *comInput* whenever the *InputCommunication* function is called.

The references and modes of operation are moved to the *input* array of the mechanism module. The mechanism module is made to execute its loop with the *loopMechanism* function. Finally the positions are copied from the mechanism module's *output* array to the *comOut* structure which is in turn transmitted over the network to the MMI using the *OutputCommunication* function.

Behind the scenes the *loopMechanism* function uses its state machine to determine the mode of operation calls its actuator and sensor modules and then calls the necessary control module depending on the mode of operation. This process reads the sensor data, performs trajectory generation if necessary, applies the control law and generates the control output which is transmitted to the IO card.

```
InputCommunication(); //populates the comInput structure

// Set the State machine of the Left XYZ Manipulator
//mechanism's state for mode of operation
XYZLeft.StateMachine = comInput.XYZLeft_OperationMode;

// Assign the inputs to the Left XYZ Manipulator
XYZLeft.input[X_PositionRef] = comInput.XYZLeft_X_PositionRef;
XYZLeft.input[Y_PositionRef] = comInput.XYZLeft_Y_PositionRef;
XYZLeft.input[Z_PositionRef] = comInput.XYZLeft_Z_PositionRef;

XYZLeft.input[X_VelocityRef] = comInput.XYZLeft_X_VelocityRef;
XYZLeft.input[Y_VelocityRef] = comInput.XYZLeft_Y_VelocityRef;
XYZLeft.input[Z_VelocityRef] = comInput.XYZLeft_Z_VelocityRef;

...

// Execute the loop for the mechanism
XYZLeft.loopMechanism(&XYZLeft);
```

```
...

// populate the output comOut structure
comOut.XYZLeft_X_Position = XYZLeft.output[XPosition];
comOut.XYZLeft_Y_Position = XYZLeft.output[YPosition];
comOut.XYZLeft_Z_Position = XYZLeft.output[ZPosition];


// Transmit the communication structure
OutputCommunication();
```

The above code segment describes the runtime interaction of the Left XYZ stage (a mechanism module) with the system. The module receives its inputs performs its algorithm and then the outputs of the module are used by the system and transmitted using the communication interface.

For position control, the axis modules use an SMC controller module. The SMC controller module for the position control was implemented as follows:

$$u_k = u_{k-1} + (GBT_s)^{-1}((DT_s + 1)\sigma_k - \sigma_{k-1}) \tag{5.1}$$

where u_k is discrete control input, $G = \{\lambda \; 1\}$ with λ being a positive constant, B is the input matrix, T_s is sampling time, D is a positive constant and σ_k is the sliding mode manifold. The control structure (5.1) is suitable for implementation since it requires measurement of the sliding mode function and the value of the control applied in the preceding step. Thus (5.1) is used as control structure as discrete sliding mode for each DOF.

The loop function of this controller module is implemented as follows:

```
void loopSMCController(Controller * c){

    double reference;
    double position;
    double error_old = c->state[SMC_error_old];
    double sigma_old = c->state[SMC_sigma_old];
    double u_old = c->state[SMC_u_old];

    double dt = c->parameter[SMC_DT];

    double u;

    // If the inputs are not passed on by pointers, exact
                        position and reference from the pointers
    if (c->linked){
        reference = *(c->input_p[SMC_reference]);
        position = *(c->input_p[SMC_position]);

    }
    // If the inputs are directly in the input array use them
    else{
        reference = c->input[SMC_reference];
        position = c->input[SMC_position];

    }

    // Calculate error and derivative of error
    double error =  reference - position;
    double error_d = (error - error_old)/dt;

    // Calculate sigma and derivative of sigma
    double sigma = c->parameter[SMC_C] * error + error_d;
    double sigma_d = (sigma - sigma_old)/dt;

    // Calculate control output
    u = u_old + c->parameter[SMC_Ku] *
                       (c->parameter[SMC_D] *sigma + sigma_d);

    /// Store states of the controller for the next loop
    c->state[SMC_error_old] = error;
    c->state[SMC_u_old] = u;
    c->state[SMC_sigma_old] = sigma;
```

```
        // Store the control output u in the output array
        c->output[SMC_u] = u;

}
```

5.1.1.4 MMI

The man machine interface (MMI) of the system is the graphical user interface, which is in the form of a windows program that is used with a mouse, a keyboard and a joystick Fig.5.7. The man machine interface software resides on a PC computer with Windows XP and it is written in C#. It communicates with the RT Motion Control computer over Ethernet, with the fine view and coarse view cameras connected by Fire Wire and with the illumination and microscope focus and zoom controllers by RS232 and with the Joystick by USB.

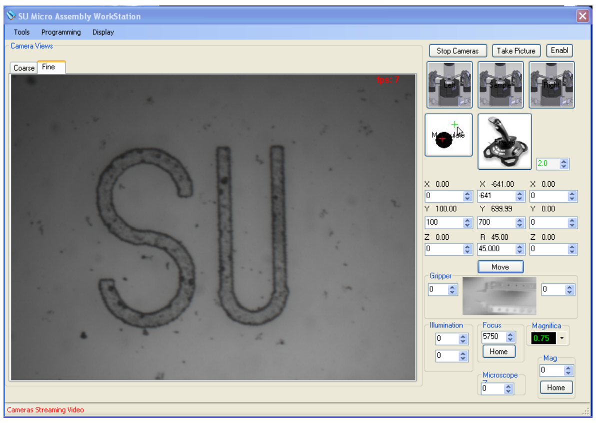


Figure 5.7: The MMI Screen.

The MMI has two basic purposes. The first is the acquisition of the images and controlling the vision system. The second is to interface with the RT computer and to send references to the real-time computer. The system also has some supporting features, such as automatic focussing abilities and calibration abilities Fig.5.8.

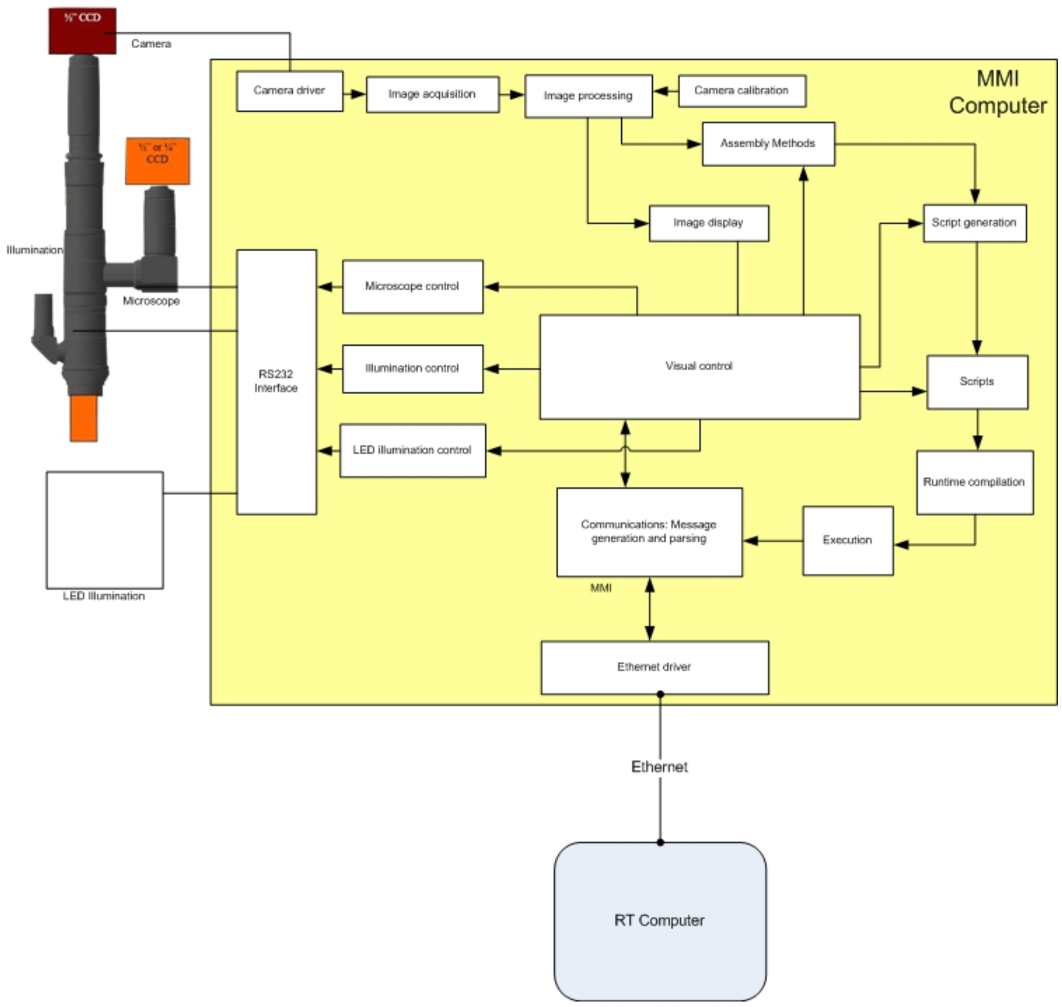


Figure 5.8: MMI Structure.

The core inputs of the MMI are the visual controls. For illumination of the samples, the lower side LED illumination module is attached by a serial port that is handled by the operating systems's serial port driver. A C# library has been generated to interface with this device, and the visual controls can send red green and blue light references to the system. Likewise the halogen light source that illuminates from the superior side is connected to a serial port and a library has been written permitting the transmission of intensity references to the illuminator. A driver for the microscope controller that communicates over a serial port has also been developed, and focusing and magnification references are sent form the visual controls to this device. The visual controls receive their inputs by means of the mouse, keyboard or joystick. All of these devices are connected over USB. They are first handled by the OS's USB drivers, then Joystick ActiveX library handles the joystick inputs. The cameras are attached to the system over the Fire Wire port. These are first handled by the camera drivers, then they pass from an image acquisition layer. The images are passed over to the image display and image processing functions for them to be displayed on the MMI. The image processing functions take the camera parameters and the results of the image processing can also be displayed on the image. Scripts that are generated on the MMI can be directly executed using the MMI. The visual controls can directly send messages over to the RT computer by using the communication message generation and message parsing interface that sends messages over the ethernet or they can use control algorithms to perform camera based automated functions.

The system is able to perform some tasks either manually by clicking and moving the objects or by mouse, alternatively, automated particle moving commands can be given to the system that perform a series of actions until the particle reaches the desired destination. The system is also capable of generating scripts and compiling them in runtime and executing them.

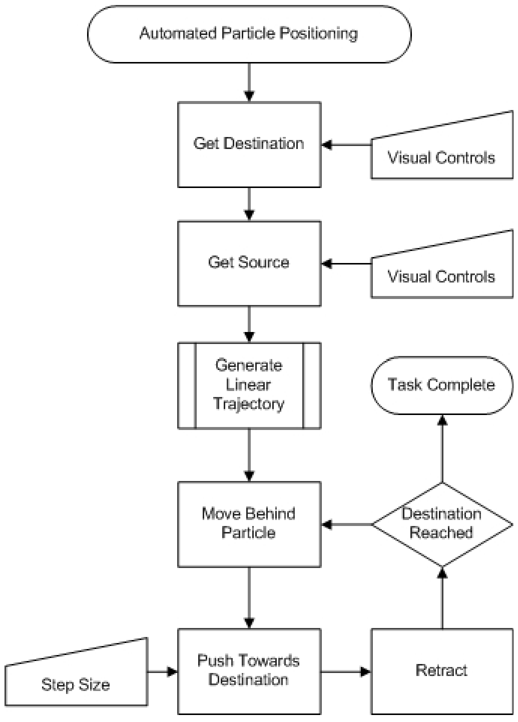
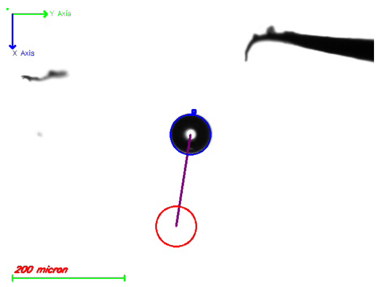


Figure 5.9: Semi-Automated Particle Pushing Algorithm.

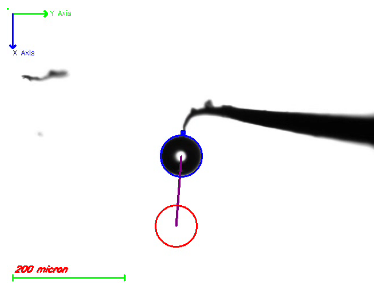
5.1.1.4.1 Modes of Operation The system is designed to manipulate 3 + 3 + 3 degrees of freedom for manipulators and the sample stage. The MMI has 2 modes of position control Fig.5.7. The first mode of position control is done by entering the position values in to numerical up down boxes in the GUI. Once all reference positions are entered to the system then the Move button is used to send the reference positions over the network to the RT Motion Control computer, followed by a Start command so that the motion control computer can move the stages to the desired positions. The second mode of position control is using the mouse and the image captured from the camera. This mode does not permit moving all degrees of freedom at the same time but rather focuses on moving a single manipulator or the sample stage at a time. Therefore it only moves 2 degrees of freedom, the ones in the X and Y axes. With this mode first the manipulator to be used is selected using the button with the icon for that stage on it. After this all the position references are transferred to the microstages of this stage. Then a position in the video image is clicked, the first click is registered as the starting position, and the second click is registered as the destination position. The MMI, that has been calibrated beforehand, calculates the difference in the X and Y coordinates and sends the position references to the RT Motion controller and at any point a right click cancels the operation. This has proven to be a very intuitive way of moving manipulators, as it is not only moving the tip of the manipulator or any other part, but rather any part of manipulator assembly that is in view can be clicked on and moved. There can be 2 grippers or other devices that can be operated with an analog voltage input by the system. The controls exist to send references to these types of end effectors. These are either indicated in percentages and the RT side handles the scaling, or a fixed scaling is applied and the references generated on the MMI are sent directly.

The manipulators can also be controlled using the joystick. The joystick information is captured from the joystick and it is sent to the RT Motion Controller as a velocity reference. Similar to position control, the manipulator to be controlled has to be selected first. One of the levers of the joystick is used to control the analog voltage reference to the selected end effector attached to the manipulator and one of the joystick's buttons has been assigned for switching between the manipulators and the sample stage. For precise operations using the joystick, the speed of the joystick can be adjusted using the adjusting numeric box in the GUI.

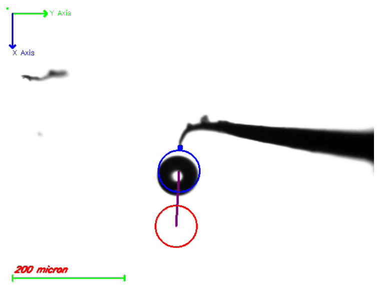
The system is also capable of performing some positioning tasks semi-automatically. The tool tip is shown to the system by means of a mouse click, then a particle and a destination position is selected. It has been noted that with the use of some probes the particles can stick to the probes. To avoid stiction an algorithm with repetitive small pushes has been developed Fig.5.9Fig.5.10.



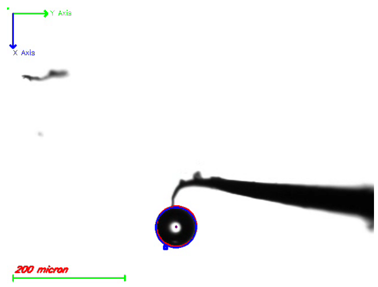
(a)



(b)



(c)



(d)

Figure 5.10: Semi Automated Manipulation

A fully automated pushing technique has also been implemented using an AFM probe. In this approach the system finds the particle and the probe automatically instead of getting any reference from the user. Details of this technique can be found in [130].

5.1.1.4.2 Scripting To create a versatile system scripting ability was added to the system. Some of the C# functions used in the system are also exposed to the scripting side of the system. The scripting language has the same syntax as C# and it has the ability to use standard Microsoft .NET framework as well as the functions developed specifically for the workstation. This enables writing scripts of different complexity levels by different levels of users. With scripting the user can write C# code using all of the functions developed for the system including all the functions for the peripherals and move commands. At design time new composite functions can be written for the system that can later be used in scripting. The system combines the script code with the available libraries for the peripherals and the system, the code is compiled and finally it is executed to run the script during runtime.

5.1.1.5 Vision

5.1.1.5.1 Calibration Visual feedback requires the images to be calibrated according to the system. To calibrate the system the startup procedure of the MMI software reads a calibration text file and parses the camera coordinate calibration parameters. These parameters exist for every magnification level of the fine view camera, and for the coarse view camera. This text file is generated using a program written in MATLAB, to which the images of a calibration grid taken using the system at a certain magnification level are supplied. This procedure enables the system to derive the relative positions in the image in every magnification.

Implementation of camera calibration for the micro assembly workstation is as follows:

- Images of a checkerboard style micro calibration grid are captured for every magnification level.
- Edge detection is applied to each image.

- The feature points are extracted from the image first using Hough Line Transform to extract the lines, and then calculate their intersections to obtain the corner points.
- Since the results are not satisfactory, edge points having a certain amount of distance from each line found by Hough transform are determined and by point-to-line fitting, new line equations are determined.
- These points with the corresponding world coordinates are then used for the determination of the reduced camera calibration matrix.

5.1.1.5.2 Depth Estimation Focusing on objects to be able to view them can be a tedious task in microsystems where depth information is not known apriori. The focus function of the microscope involves the movement of the focus lens in the vertical axis, what this means from a practical point of view is that with the change of focus, the microscope parameters stay the same, only the zone which is viewed moves in the vertical axis. Therefore, if it is know that a given object is in focus, then the focus parameter (position of focusing motor) can be used to determine its position in the Z axis. In SUMAW, as the manual focusing to find various objects and planes in the working zone can be tedious, the system can be set to detect the different layers of interest in the working zone. The algorithm for this is as follows:

- Move microscope focus to home position
- Move motor with X micron intervals downwards, and record sharpness of entire image, where X is a parameter that can be set to according to the task.

- After this process the peaks in the sharpness are assumed to be the Z positions of the objects and planes.

The different peak sharpness positions are recorded and the user can revisit these positions from a drop-down menu. The positions found using this technique are usually, probe body, planes of manipulated object, top side of lamelle, and bottom side of lamelle (which can be invisible and can cause the destruction of probes if it is tried to be reached). Acquiring the position of the probes or end effectors is not usually successful if they are not placed completely parallel to the X and Y axes and therefore there is no maximal sharpness plane. To resolve this problem a different function called microfocus is developed. The user manually defines a rectangle in the image, and the focus operation is performed on a much smaller Z scope, using the sharpness information within this rectangle. Depending on the angle of the probe, the focused region of the probe changes, and it seems as if the probe has moved. It is necessary to repeat this procedure several times until the tip or the desired part of the probe is reached. Once at the tip, the size of the rectangles can be selected small enough to get a precise focus on the tip, and then extract the Z position.

5.1.1.6 Communication

The communication between the RT Motion Control computer and the MMI computer is done over TCP/IP. For this application it was desired that the communication was performed on a more granular basis instead of transmitting the entire communication structure. A simple text based communication protocol has been developed for this purpose which is easy to debug and also enables the system to be used with a text based system such as the windows hyper terminal application.

Every piece of information to be transferred has a two or the letter long identifier.

5.1.1.6.1 RT Variable Object Each variable in the realtime system is presented as an RT Variable Object in the MMI side. An RTVariable Object is a class that keeps the information of one variable. Each instance keeps the name, identifier, DOF number , parsing regular expression, unit and update time values. In addition, functions to parse a string using the identifier and regular expression to extract the value and also generate a string to be transmitted have been developed. The update time is value is used to check that the data is not old. The RTVariables are kept in two lists, Updated Variable list for receiving variables and Outgoing variable list for sending variables to the RT computer.

When a message is received from the RT computer, it is first segmented and the parts of information are sent to the variable matching function. The information arriving from the RT computer is generated in a string that is formatted, the segmenting function starts from the beginning of the string. As the variables are kept in an orderly fashion in the Updated Variables list, they can be matched with the same order. The matching function initially starts from the first RTVariable in the list and it keeps track of the last variable received. This assures that hunting for the variable in the list is kept to a minimum Fig.5.11.

5.1.1.7 Experiments

The following paragraphs present the results obtained from the micro assembly workstation. In order to validate the efficiency and to determine the accuracy of the system we first verify the position control abilities. Then we perform higher level automated pick an place tasks to move objects. Finally to demonstrate the

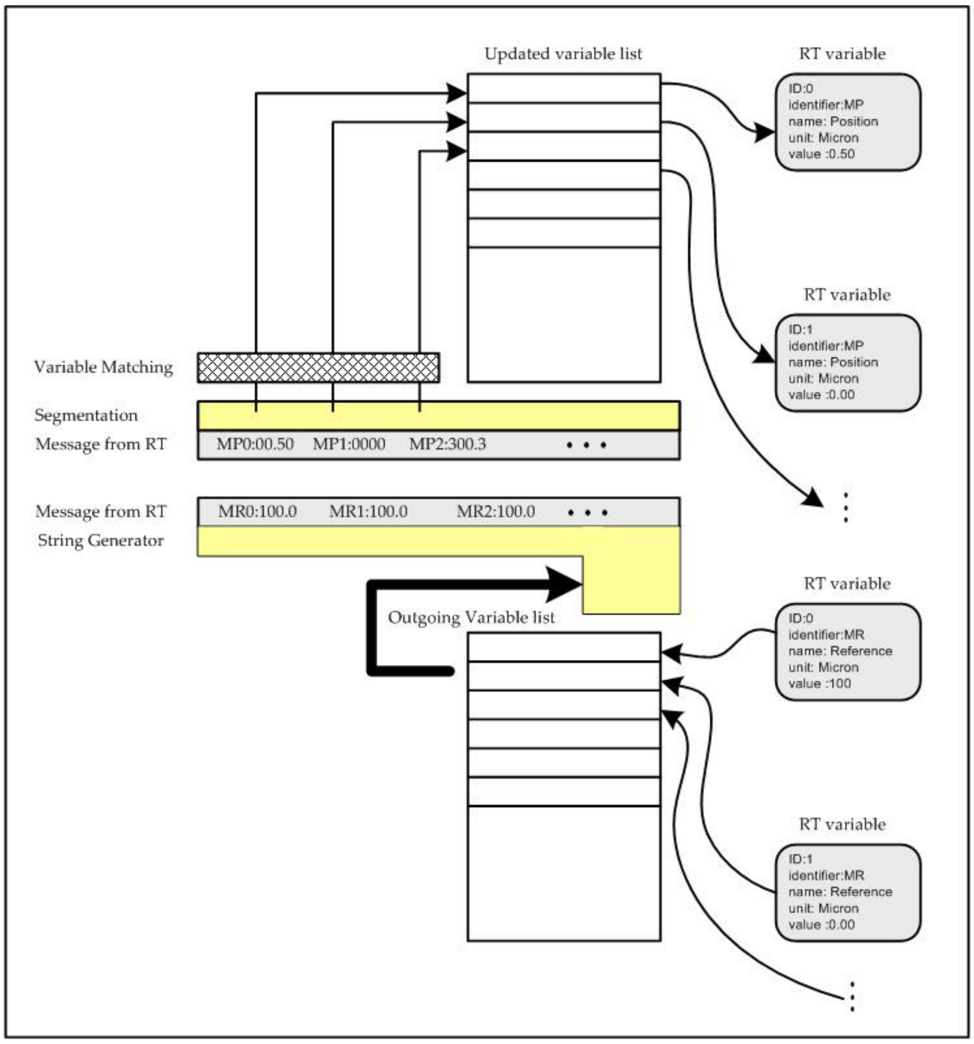


Figure 5.11: Communication With RT.

possibilities of applications, we examine a real world problem: "extraction of the mechanical properties of a zebrafish embryo".

5.1.1.7.1 Validation of the System The motion control system was configured to run with at a control frequency of 10kHz. It was observed that when all 10 degrees of freedom are following trajectories grouped in 3's, the control frequency maintained its realtime characteristics. Position control for the motion stages was achieved to the resolution of one encoder pulse (7 nm) and piezo stages were operated at a precision of 1 nanometer.

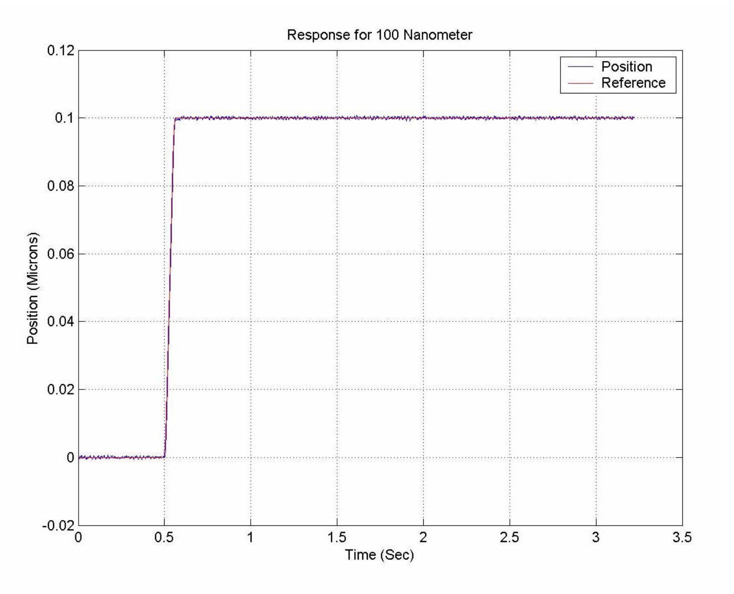


Figure 5.12: 100 Nanometer Step Response.

Manual manipulation tasks, semi-automated manipulation and automated manipulation tasks were performed using the workstation. The system was configured with tungsten probes for automatically pushing polymethylmetacrilate (PMMA) particles using the mouse interface. The system was equipped with a micro gripper to perform pick an place tasks using the joystick. Then the particles were positioned in a pattern according to predefined assembly procedure. The operators reported that the the MMI is user friendly and intuitive to use.

5.1.1.7.2 Biological Specimen Manipulation The a team at the SU Microsystems lab equipped the workstation with a force sensing probe and a micro gripper 5.13. The core realtime algorithms and components of the system were utilized in a previous version of the workstation and a task specific image processing and vision MMI was developed to extract and estimate the membrane properties of zebrafish embryos [131].

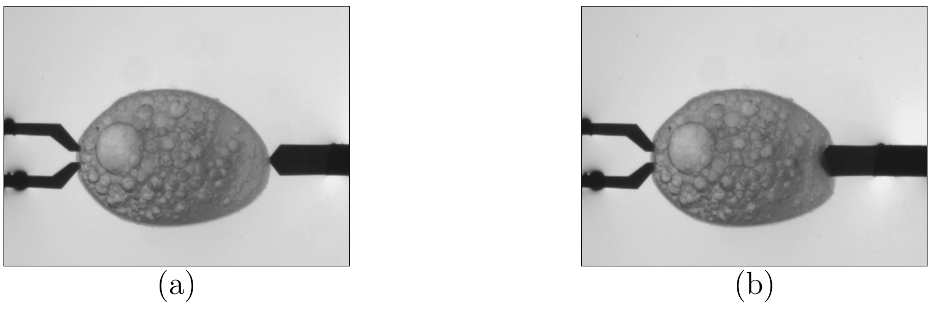


Figure 5.13: Zebra Fish Embryo in contact with microgripper from left and force sensing probe from right.

5.1.1.8 Conclusion

In this case study a novel micro assembly workstation is presented. The workstation uses standard hardware (Intel x64 based CPU and motherboard) and runs on a Linux operating system. A dedicated processor approach is used to obtain real-time performance whereas most of the previously developed systems do not support real time or depend on specific hardware. The software structure created for the workstation is designed to be modular and expandable. Wrapper interfaces are used to read and write data from/to IO cards. This enhances the portability of the system. Communication between the realtime components and the MMI is established over TCP/IP. Additionally multiple graphical user interfaces or realtime systems can be used over several computers which communicate between each other. This may achieve better performance compared to systems which are designed to run on a single machine due to the expandability of the processing power. Image processing functions were implemented to detect objects and to perform tasks on the system. The workstation includes basic image processing and blob detection functions and automated assembly functions. Experimental results show that the workstation is capable of performing precise motion control. Furthermore the system can be adopted to perform a variety of different tasks including micro parts placement operations and biological manipulations.

5.1.2 Microfactory

In this section the creation of a micro factory module[132] is presented, this module is the result work done at the Sabanci University Microsystems Laboratory as a PhD. [133]. The concept behind micro factories is the notion that the production of miniaturized parts should also be done with small machines that provide savings on space, resource utilization and energy. The minimization of the distance and the traveling masses enables high speed production. The advantages of using micro factories are numerous, space reduction, cost reduction, customization of products, flexibility, inventory cost reduction, energy savings are a few of the advantages.

The micro factory concept is implemented with a bilevel modular robotic assembly cell which provides two layers of modularity is developed for advancing the microfactory concept. The module itself is used as a brick to establish a microfactory layout acting as a process module realizing one complete process within itself. The robotic assembly module consists of all the mechanical components necessary for the assembly process, motion control hardware/software, vision system and main system supervision software. The assembly module also has parallel kinematic miniaturized robots (delta robot, pantograph), serial kinematic manipulators, carrier units, sensors, stoppers, cameras for the realization of an assembly process. The performance of the system is tested with pick place experiments realized with miniaturized delta robots (3 dof parallel kinematic robot) with the visual guidance supported by microscopic vision sensor located on the carrier unit. A graphical user interface (GUI) is designed for the operator to easily realize the desired assembly tasks and control the system. The results that we obtained through the experiments are promising in the sense of the realization of such a modular microfactory concept and the initiative to use for real life applications Figure 5.14.

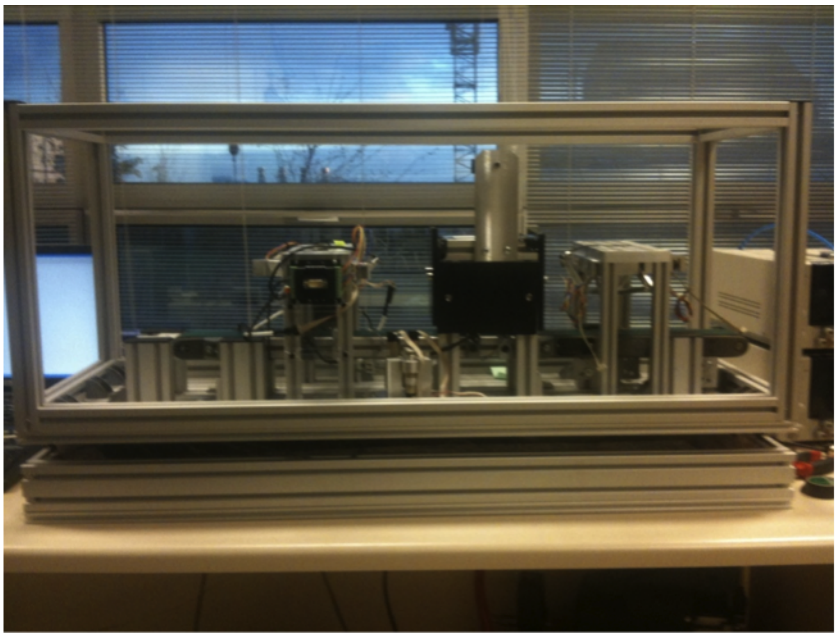


Figure 5.14: Microfactory Setup

Modularity is an important consideration in the design as one of the most important features of the microfactory concept and the units of the microfactory should be realized in order to obtain the advantages that modularity feature provides. When the modularity is achieved, flexibility appears in the production process which enables producing different products by simply reconfiguring the production units or the layouts of the system which is cheaper and faster when compared to the conventional production systems. The modularity concept can be achieved by dividing the whole system into subunits which can be called the modules. The decision of splitting up the whole system in order to configure the modules is an important step for the microfactory concept to be generated. The modules should be developed in such a way that easy configuration of a complete production system can easily be generated by cascading the modules and forming an efficient layout for the production system. For the micro factory to be truly modular the modularity also has to be reflected in the software.

For the microfactory according to the design methodology, the system requirements were specified and then the components of the system were separated according to their realtime and non realtime needs 5.15.

The components required of the micro factory module are listed as in Figure 5.16

Supervision is the main structure of the software interconnecting and controlling every module according to the flow diagram of the system. This module should have a modular structure in order to allow modularity and reconfigurability of the microfactory modules.

Graphical User Interface provides the interaction of the software and hardware units of the microfactory via human operators. The operator controls

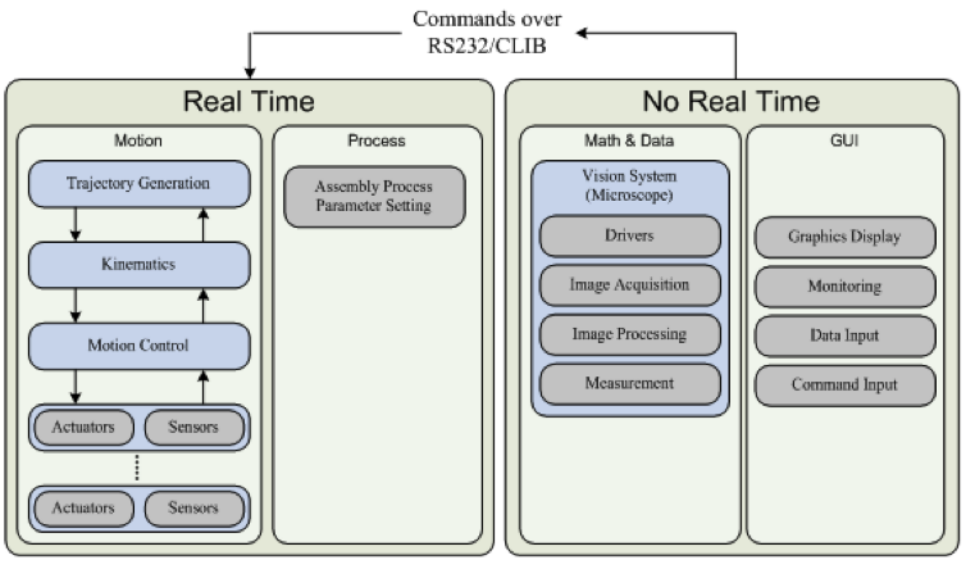


Figure 5.15: RT/NonRT Software Layout of Micro Factory

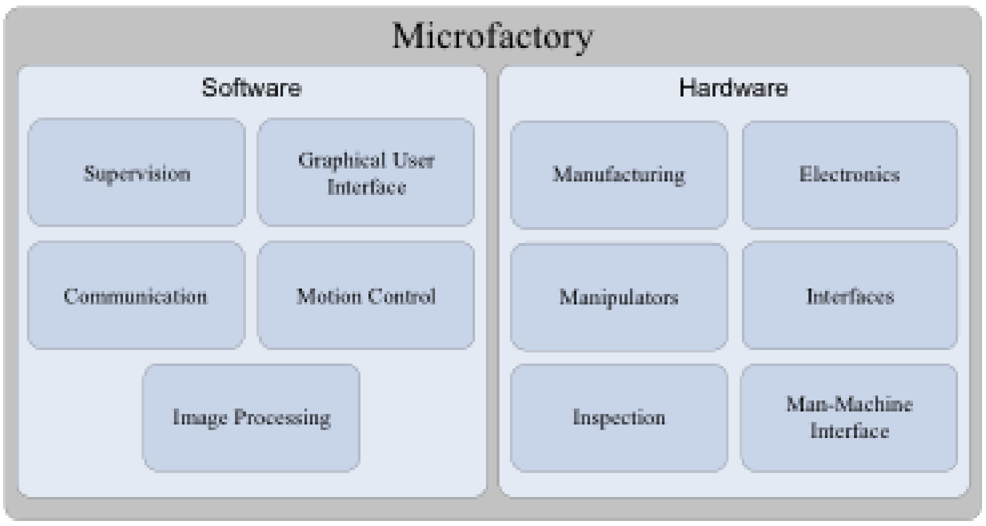


Figure 5.16: Components of Microfactory

and observes all the states of the system using the GUI which is composed of system inputs and outputs as visual indicators. The GUI may run on a computer or a handheld device according to the needs.

Communication software enables the interaction between all hardware and software units of the microfactory. The data between the units can be transferred between different units of the system using different communication protocols. Communication software is the unit that handles the data transfer according to the type of the protocol.

Motion Control is the one of the most important components of the microfactory since the precision and accuracy of the actuators mostly depend on the control performance. In order to achieve high precision and accuracy which is a must for the motion, the suitable algorithms are selected and implemented in the motion control software unit.

Image Processing is necessary for the inspection of the processes, detection of the position and orientation of the parts, object recognition and for any other purpose where visual feedback is necessary. Since there are fixturing limitations as a result of the size of the parts, for the detection of the position and orientation of the parts a vision system is inevitable in a microfactory setup. Image processing software includes the algorithms and methods that are necessary to extract the necessary features and data for the system from the visual feedback supplied by the vision sensors.

And the hardware components of the microfactory module are listed below:

Manufacturing components of a microfactory involves any type of miniaturized manufacturing system necessary for the production of the desired part. Micro lathe, micro drill, micro laser cutting, etc. can be given as examples of the manufacturing components.

Electronics components can be examined as main control unit, interfaces and drive electronics. Main control unit is the processor board on which the whole software is running. Interfaces provide the connection between the peripheral electronics equipment and the main control unit, drive electronics is the interpreter between the control unit, actuators, peripheral equipments etc.

Manipulators are robotic arms in several configurations with any number of degrees of freedom realizing the operations like transfer and assembly in the system. Serial or parallel kinematic structures can be selected according to the process necessities.

Inspection units supply the necessary feedback data for the system. Vision systems for parts detection, product quality control, etc. and different kinds of sensors providing such data can be included into this category.

Interfaces are the units providing the transaction of energy, air, vacuum and any necessary material for the flow of the production.

Man-Machine Interface is the interaction device between the operator and the system. The operator can interfere and control the defined part of the system using the man machine interface. Haptic devices and joysticks are mainly used as man-machine interface units.

The micro factory was implemented on a dSPACE platform because of its hard realtime motion control constraints with a loop frequency of 10kHz. The man machine interface was implemented on an x86 based PC running MS Windows XP Figure 5.17. Another implementation of the microfactory was done using an FPGA instead of the dSAPCE setup. The two setups are almost identical from a software perspective but the electronics implementation are different. The software of the microfactory interfaces with the hardware wrapper interface of the framework as does the FPGA. The hardware wrapper interfaces of the two implementations differ; one maps the input and output functions to dSPACE's IO functions whereas the FPGA implementations hardware wrapper maps the functions to custom memory spaces on the FPGA that in turn provide references to the various modules coded to the gates of the FPGA. For communication, the communication structures again were identical in both systems but the communication functions were setup to exchange data over CLIB in the dSPACE implementation and over RS232 in the FPGA implementation.

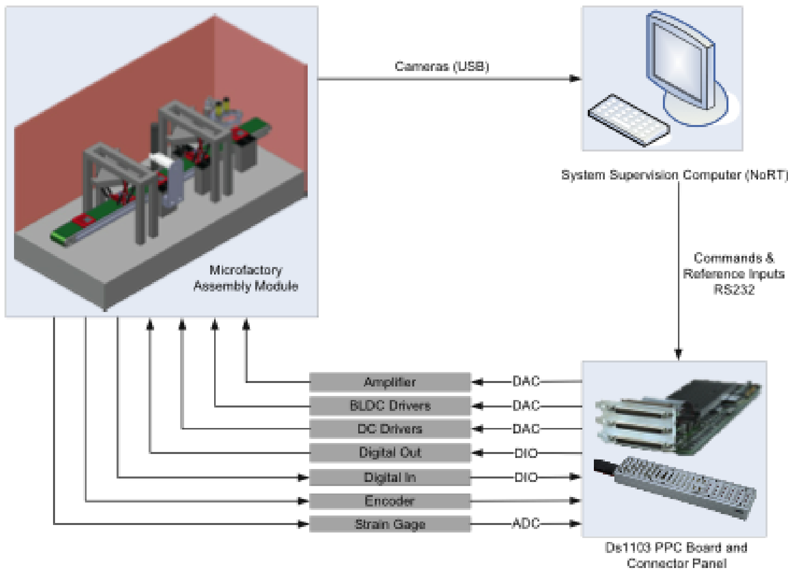

Cameras (USB)
System Supervision Computer (NoRT)
Commands & Reference Inputs RS232
Microfactory Assembly Module
Amplifier
BLDC Drivers
DC Drivers
Digital Out
Digital In
Encoder
Strain Gage
DAC
DAC
DAC
DIO
DIO
ADC
Ds1103 PPC Board and Connector Panel

Figure 5.17: System Layout of Microfactory Module

The motion control of the micro factory is composed of the following sub modules:

- 2 Delta Robots
- Pantograph Robot
- Conveyor

The sensors and actuators for these submodules were selected to meet the necessary properties. The motors, encoders and other sensors were coded as actuator and actuator modules to interface with the necessary IO's to provide the necessary functionality. After this step they were joined together in axis structures and controllers were added to them to ensure reference following. After this phase the axes of the pantograph, delta robot and conveyor were grouped in the mechanism where they are further joined with kinematics functions. At this stage the task requirements of these mechanism modules were examined and the different modes of operation were extracted. The state machines of the mechanisms were configured to provide the different modes of operation. Finally the mechanisms were assembled under the system structure and the communication module was configured to communicate with the graphical user interface.

Each of these modules were developed until they formed mechanisms in the software framework and then finally they were integrated in to the system component of the framework to form the motion control software of the microfactory.

5.1.2.1 Conveyor

The conveyor of the system consists of a moving belt, 3 stoppers to stop the moving sample trays at the stations and 3 sensors to sense the presence of trays in their stations Figure 5.18. The conveyor mechanism is a very simplistic mechanism in the sense that although it has several actuators and several sensors, these are not linked together with a controller making the conveyor a mechanism that uses only its actuators and sensors without using controllers, filters, observers or use of kinematics.

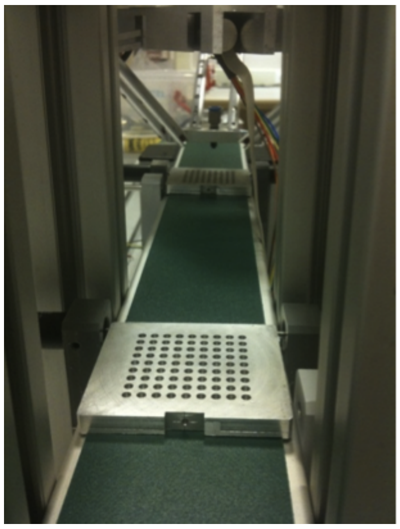


Figure 5.18: Conveyor Submodule

There are several actions required from the conveyor, these were defined as: move conveyor and move tray to station also as a parameter the speed of the conveyor could be set. The conveyor submodule has therefore 3 sensors that are hall effect sensors to sense the presence of trays, the sensors are linked to digital inputs of the IO cards. There are 4 actuators in the conveyor submodule, 3 of these are the solenoids that stop the trays and one is the motor that drives the conveyor. The stoppers are connected to the digital outputs of the IO cards and the conveyors motor is connected to an analog output of the IO card. The IO numbers are assigned to the sensor and actuator modules and they are added to the mechanism module. This permits the mechanism module to open close the stoppers, move the conveyor and sense the presence a tray on a station on the conveyor. The next step is the creation of the algorithms to move the conveyor and the trays to the desired positions. This is achieved by the configuration of the state machine in the mechanism module. The mechanism module exposes an interface for the system module to be able to control the conveyor by manipulation the states of the state machine and setting the speed of the conveyor Figure 5.19.

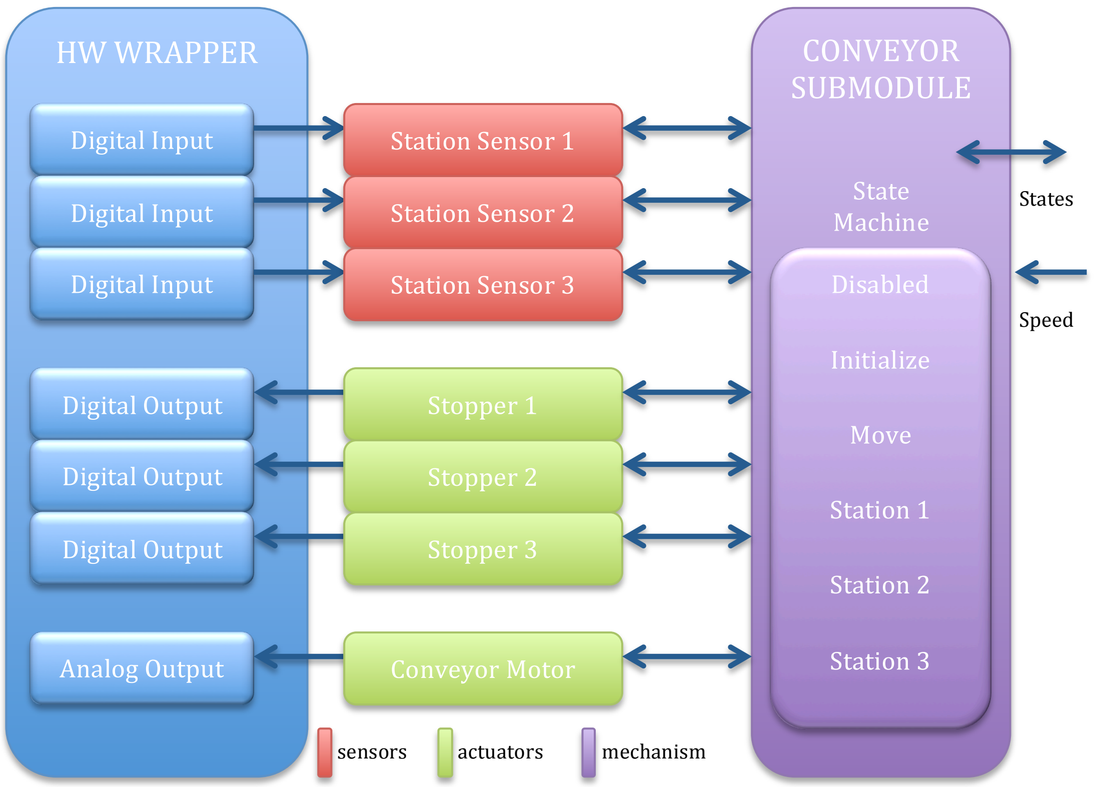


Figure 5.19: Conveyor Module Software Structure

5.1.2.2 Pantograph

A pantograph is a five link parallel mechanism with two degrees of freedom moving in x and y cartesian coordinates. The miniaturized version of the mechanism is developed to be used as a micro manipulator for the concept of the microfactory.

The mechanism is enhanced with additional degrees of freedom with a rotational axis at the tip for handling purposes and a z axis. The developed mechanism is shown in Figure 5.20. The actuators used in the mechanism are DC motors with integrated incremental encoders.

The software for the pantograph was modeled as in 5.23.

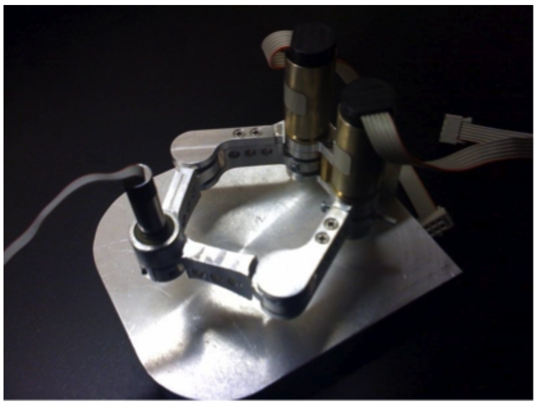


Figure 5.20: Pantograph Submodule

In order to test the performance of the pantograph an experimental setup is established where a XY position sensor is used to get task space measurement. Figure 5.21 show the result of the experiments for a circular reference trajectory with 100 μm diameter for the joint and task space measurements respectively.

5.1.2.3 Delta Robot

Delta robot is one of the most famous parallel robots which consists of a traveling plate connected to the base with three identical parallel kinematic chains each of which is actuated by a revolute motor mounted on the fixed base plate. The mechanism is shown in figure 5.22. The mechanism has three degrees of freedom moving in x, y and z coordinates. The parallelogram structure of the lower arms

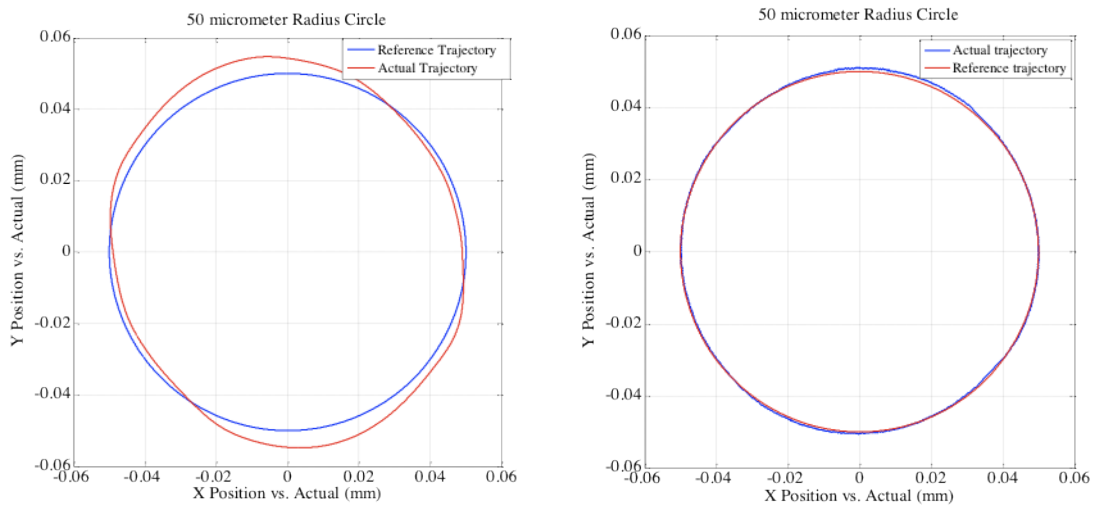


Figure 5.21: 100 micrometer circle reference and actual trajectory (configuration space (a) and task space (b) measurements)

provides parallelism of the traveling plate to the fixed base plate.

The mechanism designed is the miniaturized version of the robots that are widely used in the industry. The workspace of the robot is determined to be 40 mm cube and the kinematic parameters of the robot are determined according to that prescribed work space with an optimization algorithm.

In the final prototype of the robot, high speed brushless DC motors are used as actuators. Motors are equipped with integrated encoders and planetary gear heads achieving resolution of 0.0026° and speed of 16.4 rps.

The software for the components of the delta robot were modeled as in Figure 5.24.

Sinusoidal input references to X-Y axes of the Delta Robot are given with different amplitude and frequencies in order to achieve a circular reference. Figures 5.25 5.26 show the reference vs. encoder output and the reference vs. sensor output for a different radii circle input at different frequencies. The sensor outputs show

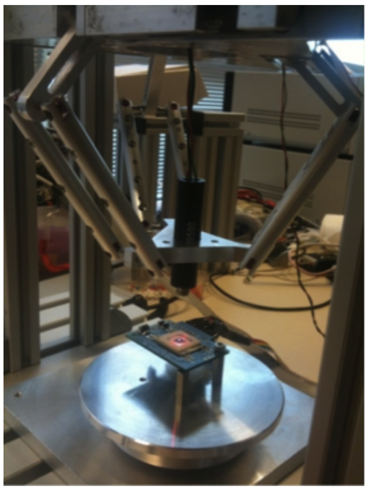


Figure 5.22: Delta Robot Submodule

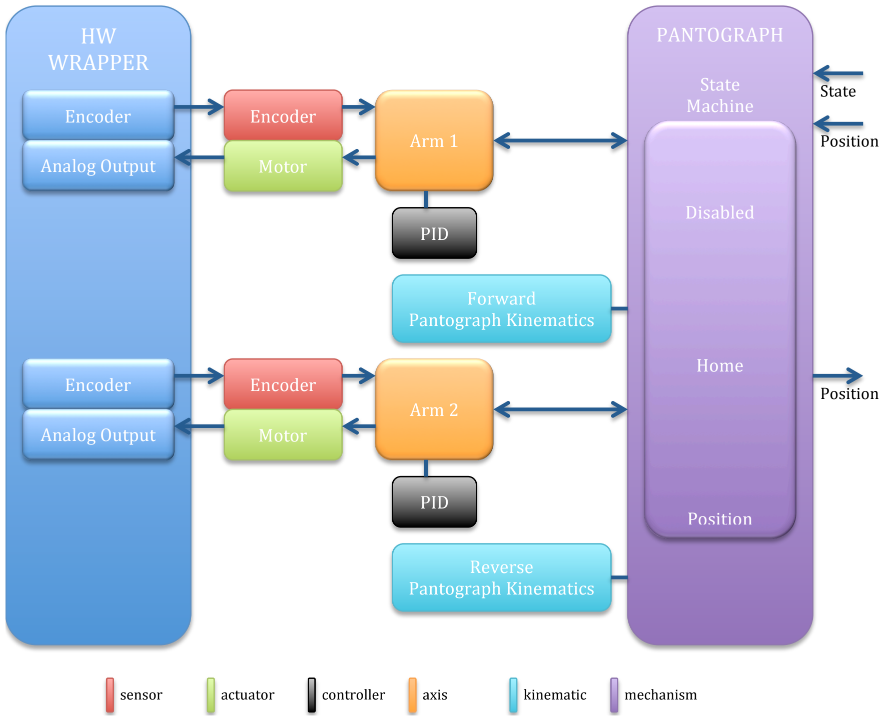


Figure 5.23: Pantograph Realtime Software

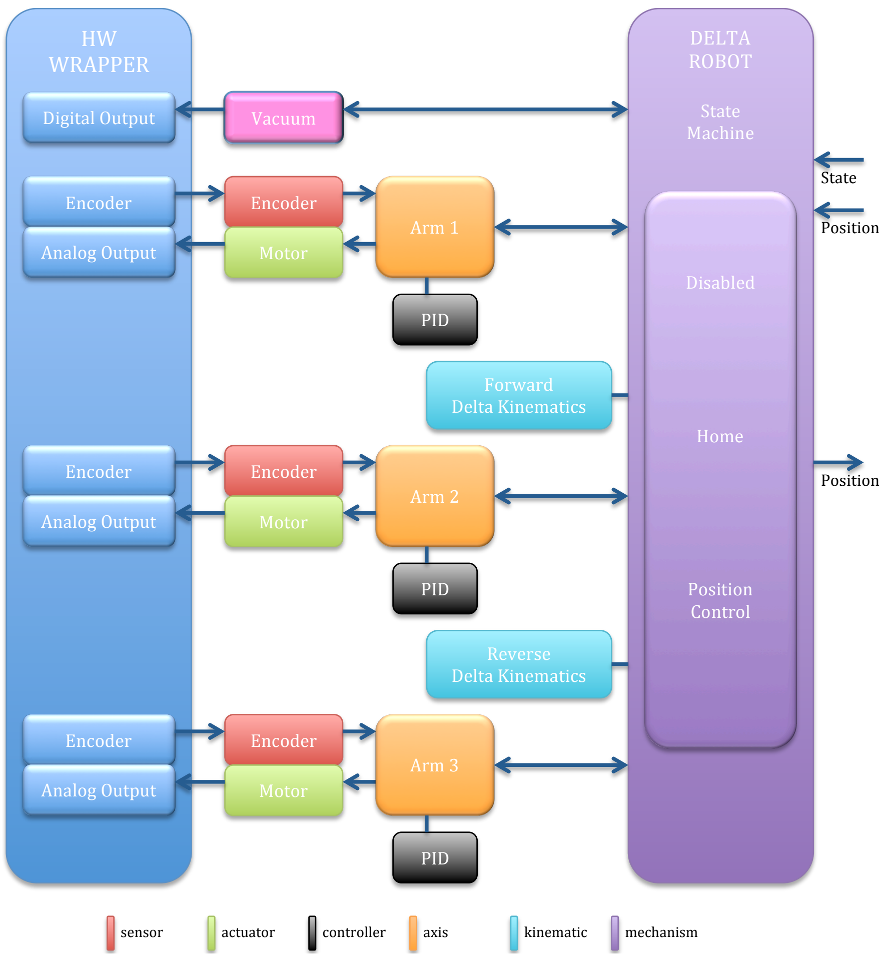


Figure 5.24: Delta Robot Realtime Software

slightly elliptic structures as a result of the horizontal alignment of the sensor and the Delta robot endeffector. This is due to the mounting of the sensor since it can not be perfectly aligned. Encoder outputs give the motor angles and using the forward kinematics equations the endeffector position is calculated and shown in the figures. However this does not representing the exact position of the end effector since the manufacturing and mounting imperfections of the robot can not be taken into account in such a calculation.

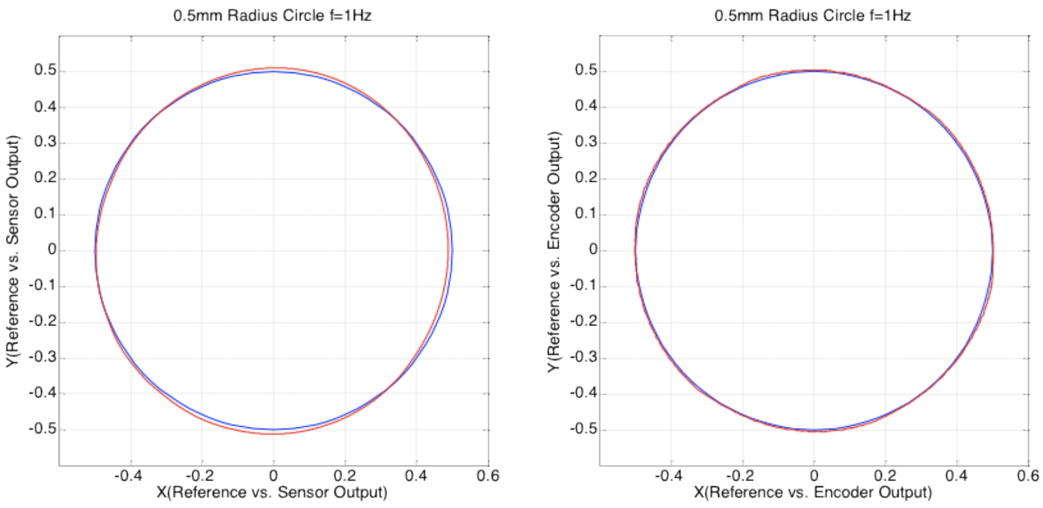


Figure 5.25: 0.5mm Radius f=1Hz Circle, Ref. vs Sensor (a) and Encoder (b)

5.1.2.4 The Micro Factory System

After the primary components of the motion control of the micro factory have been created as mechanisms. These are added to the system structure and the communication is developed. In this case data transmission over CLIB is implemented. CLIB is dSPACE's communication library. It works by gathering the addresses of the selected variables in the system. Later the host computer for the dSPACE

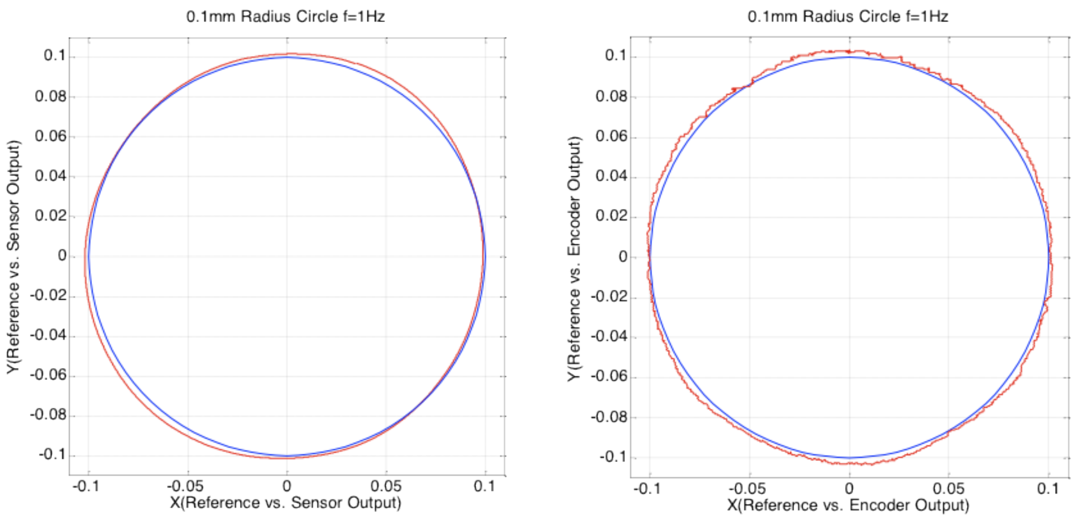


Figure 5.26: 0.1mm Radius f=1Hz Circle, Ref. vs Sensor (a) and Encoder (b)

system inquires the variables address location and retrieves its data.

The micro factory module is tested with two delta robots realizing pick place experiments simultaneously. Experiments are realized using 3 mm steel spheres located on a tray which are moving on a conveyor. The desired assembly procedure is generated by the operator using GUI according to the visual feedback gained using an optical microscope integrated to the system. After the generation of the task, the command is given to the system and the robots perform the pick place operations.

Figure 5.28 shows the experimental results for 8 steps pick place experiment using one delta robot as the manipulator. These experiments are realized on the assembly module with the FPGA control hardware.

The system is then easily migrated to the dSPACE platform. The modularity of the software framework is also tested with adding an additional delta robot to the system. 8 step pick place experiments are realized using two delta robots

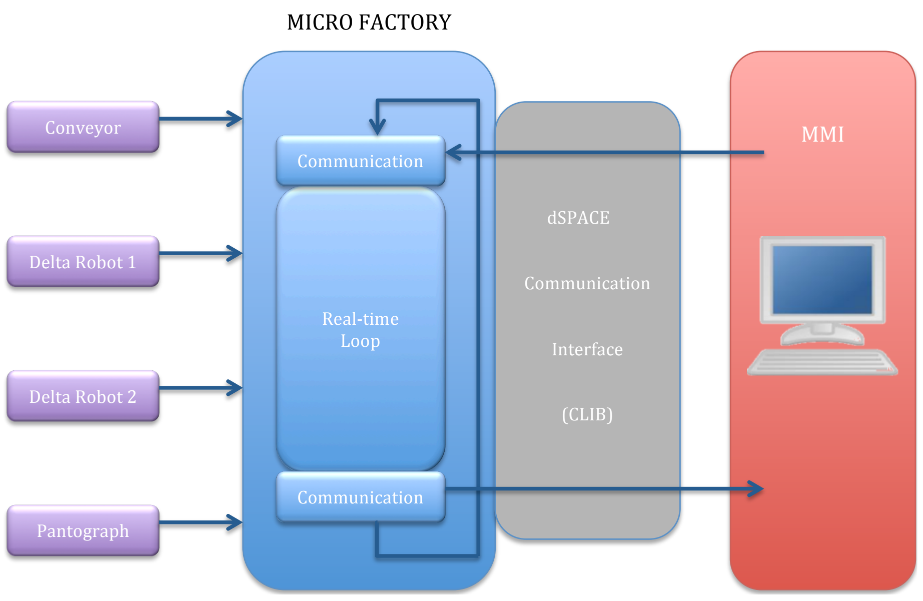


Figure 5.27: Micro Factory Motion Control Software

working simultaneously. The results of the experiments are given in Figure 5.29. The XY motion of the Delta robots are given in Figure 5.30.

Experiments done on the micro factory demonstrate that the modular structure of the framework provides ease of developing systems. Assembling the modules in to higher level modules provides easy comprehension of the system. Also the delta robot module is reused further demonstrating the effectiveness of the modular architecture. The migration from a dSPACE platform to an FPGA platform shows that the software developed is platform independent.

5.1.3 Haptic System with Time Delay

The framework was implemented on a tele-robotic system to achieve remote control for teleoperation research. The haptic system is composed of a master system to accept the inputs from a human operator and a slave system that is desired to

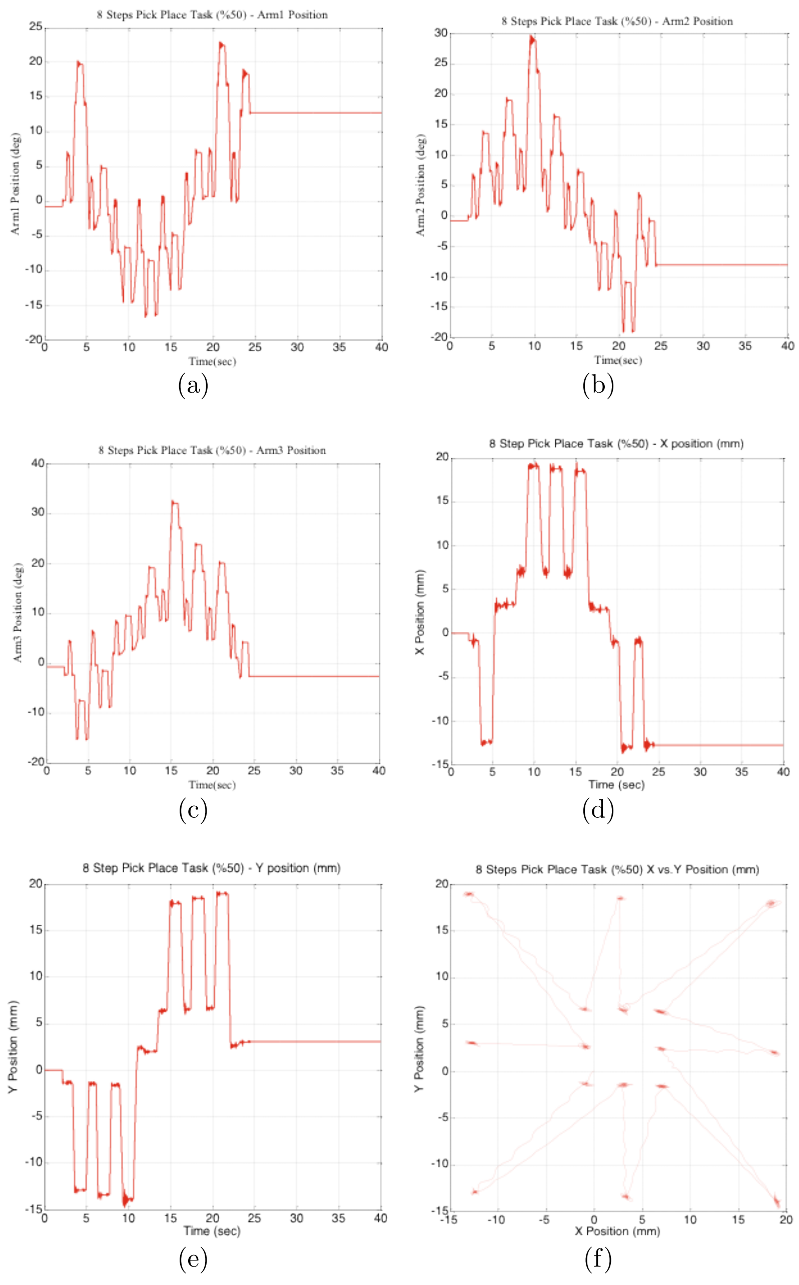


Figure 5.28: 8 Steps Pick Place Experiment (50% Speed with FPGA)

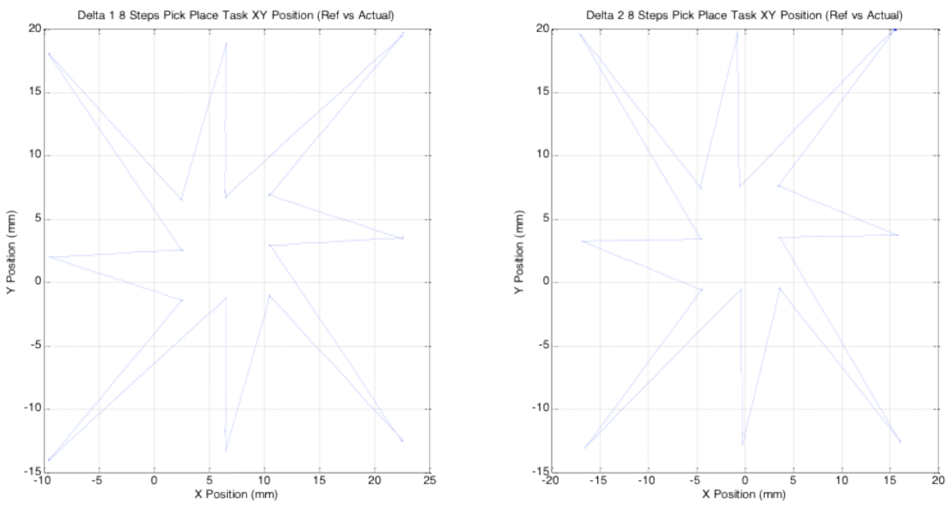


Figure 5.29: X and Y Positions of the robots during the 8 Step Pick Place Operation (Vel Max = 60mm/sec and Acc Max = 20 mm/sec^2)

mimic the master motion under the existence of time delay between data transmission of two systems. This project was constructed as a part of a masters thesis at the microsystems laboratory[134].

5.1.3.1 Experimental Setup

The system is composed of two Hitachi-ADA series linear AC motors and drivers with Renishaw RGH41incremental encoders with 1μm resolution as depicted in Figure 5.31. The setup runs on a D-Space DS1103 card and the loop time of the system is configured at 1kHz.

For conception of the system, the linear motors were modeled as actuators receiving their inputs over analog to digital converters. The torque constant was also added to the actuator structure interfacing them as force generators to the motion control software. The encoders were modeled in the system as sensors interacting over the encoder inputs of the hardware wrapper. The specifics and

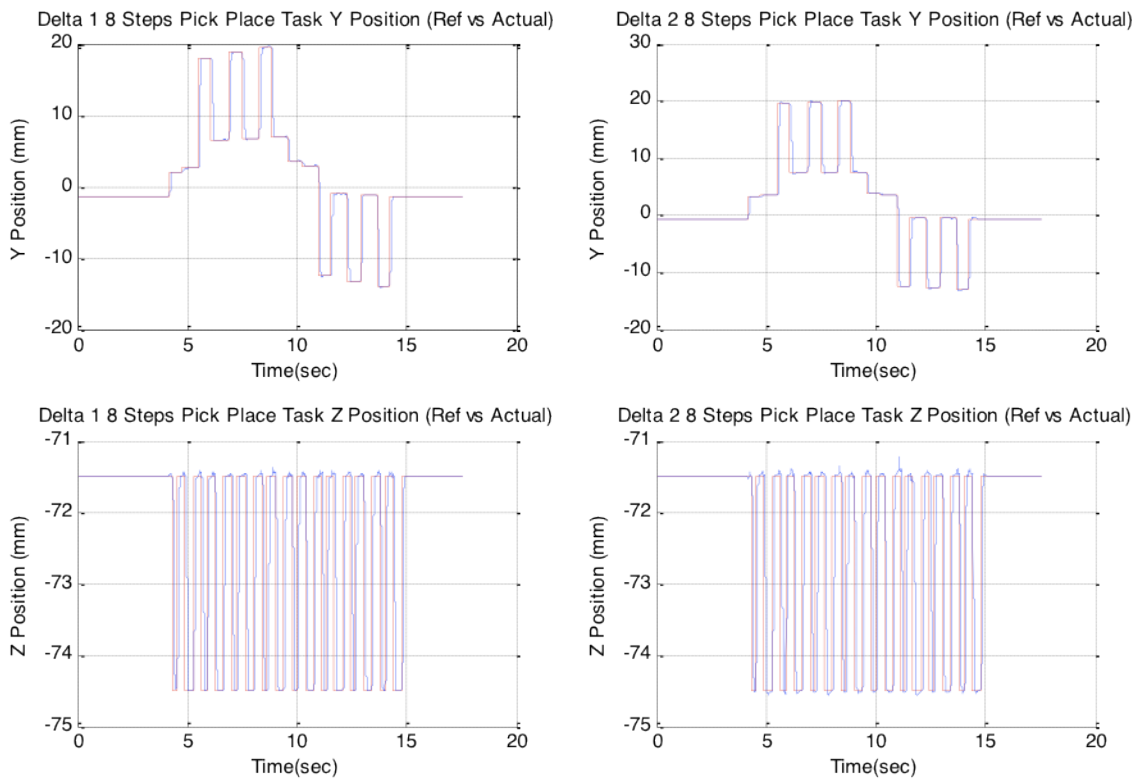


Figure 5.30: X, Y and Z positions of the Delta robots for the 8 Step Pick Place Experiment (Vel Max = 100mm/sec and Acc Max = 30 mm/sec^2

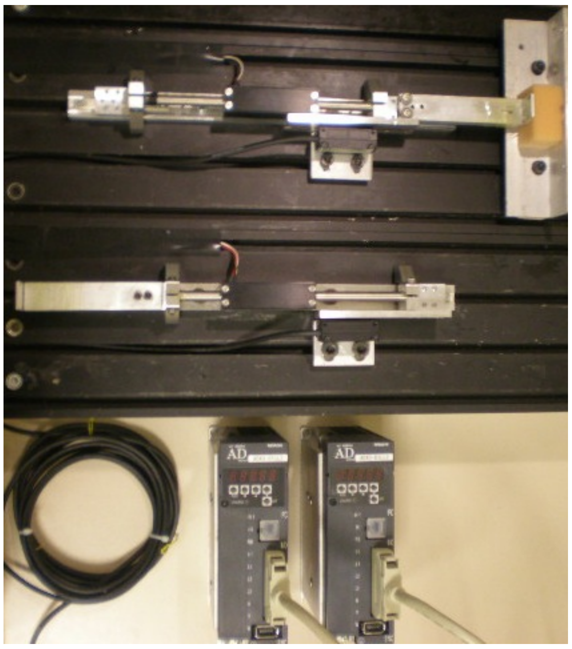


Figure 5.31: Haptic system with two linear actuators

the derivation of the control algorithm are described in detail in [135]. The master and slave systems are modeled as axes. The information the master sends a force reference to the slave and the slave sends back its velocity. Both of these are delayed. From the framework perspective the control algorithm is decided in to the following modules:

Velocity Estimator For the control of the slave, both the slaves and the masters velocities must be known. The velocity estimator module is applied to estimate the velocity from the encoder sensor information. The estimator structure used for this implementation was described in [136]

PD Controller The master system utilizes two PD controllers one to generate the references for the slave system and another to enhance convergence of the observer-controller coupled system. The slave system also utilizes a PD controller module to enhance convergence of observer-controller so that the systems are assured to carry out stable tracking.

Communication Disturbance Observer As the master system receives the velocity measurement of the slave with a delay it requires a communication disturbance observer to observe the effect of the delay on velocity measurement of the slave. Once the velocity of the slave side is estimated, the convergence terms take into action to make the two system outputs give the same results [135].

Disturbance Observer The slave side motion control may have disturbance on it and a disturbance observer is used to observe this disturbance and compensate for it. The measured disturbance is fed back to the system to compensate further the undesired effects and come up with a robust system.

The two Axes are linked to a mechanism structure which delays the signals and passes them from the master to the slave and vice versa as depicted in figure 5.32.

For this setup functions to create fixed and random delay were added to the system. These can be found in the appendix.

The experiments done by implementing the framework have proven that a methodical way of generating motion control systems is achieved. The results obtained from two different experiments indicate the difference clearly. The algorithm for the haptic system, when generated by Simulink blocks can run at a maximum of 11 Khz frequency while using the proposed structural formalism, we could achieve a maximum of 73 Khz loop frequency.

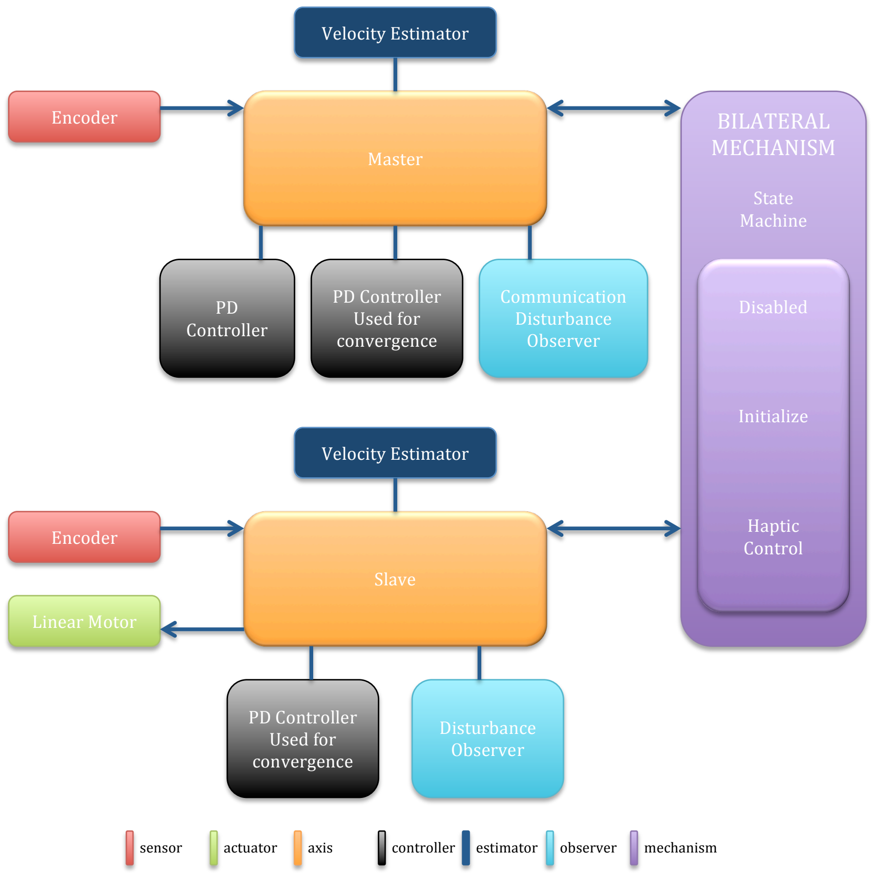


Figure 5.32: Haptic system software structure

6 CONCLUSION

A framework for precise motion control is presented in this thesis work. The primary aim of the framework is to create sustainable software that is easy to maintain. The framework models the various components of a complex multi degree of freedom motion control system. The framework was created with a modular approach where every modules interfaces are defined. A layered approach separates the different engineering challenges such as control algorithms, estimation algorithms, trajectory generation, kinematics etc.. of the motion control problem. The modules are used by other modules to create higher levels of modules leading to structures such as mechanisms and robots with different modes of control where the tasks of the problem domain can be tackled. The modular approach brings the advantage of reusability to the motion control software where simple components such as controllers can be ported to different applications or complex modules such as mechanisms can be reconfigured and used in the same project. The created motion control structure is interfaced to other systems or man machine interfaces using a communication module that can be programmed to receive and transmit data over different types of mediums.

The defined features of the software framework are tested by implementing the software on different platforms with different motion control necessities. Experiments performed on a micro assembly workstation demonstrated that the software is scaleable and can be implemented on multiple degrees of freedom. It is also shown that the framework is platform independent as migration from dSPACE environment to Linux can be performed with ease. Experiments done on the micro factory demonstrate that the modular structure of the framework provides ease of developing systems. Assembling the modules in to higher level modules provides easy comprehension of the system. Also the framework is capable of handling motion control needs such as kinematics as with the delta robot module. The components of the framework can be reused further demonstrating the effectiveness of the modular architecture. The migration from a dSPACE platform to an FPGA platform reaffirms that the software developed is platform independent. Experiments performed on the haptic system with delay show that the framework is capable of handling complex motion control logarithms. Also the software implementation is capable of achieving higher loop frequencies in real time compared to some popular automatic software generation tools such as Simulink.

The framework is demonstrated to be a useful tool for the modeling of motion control systems. To increase the usage of the framework, it will be included in the communities of the current popular open source frameworks such as ROS and OROCOS to increase its availability and also to obtain new controllers and modules.

A commercial implementation on mobile service robots is also in the negotiation phases.

Bibliography

[1] C. J. Evans, *Precision engineering: An Evolutionary View.* Cranfield, UK: Cranfield Press, 1989.

[2] K. K. Tan, K. Z. Tang, H. F. Dou, and S. N. Huang, "Development of integrated and open-architecture precision motion control system," *Control Engineering Practice*, vol. 10, pp. 757–772, 2002.

[3] A. Basak, *Permanent-magnet DC linear motors. Monographs in electrical and electronic engineering.* Oxford: Clarendon Press, 1996.

[4] K. Ohnishi, N. Matsui, and Y. Hori, "Estimation, identification, and sensorless control in motion control system," in *Proceedings of the IEEE*, vol. 82, no. 8, 1994.

[5] K. Ohishi, K. Ohnishi, and K. Miyashi, "Torque-speed regulation of dc motor based on load torque estimation," in *Int. Power Electronics Conf. IPEC*, vol. 2, Tokyo, 1983, pp. 1209–1218.

[6] T. Umeno and Y. Hori, "Two degrees of freedom controllers for robust servomechanism, their application to robot manipulators without speed sensors," in *IEEE 1st Int. Workshop on Advanced Motion Control*, Yokohama, Japan, 1990, pp. 179–188.

[7] Y. Hori, "Disturbance suppression on acceleration control type dc servo system," in *Proc ZEEE PESC'SS*, vol. 1, 1988, pp. 222–229.

[8] M. Nakao, K. Ohnishi, and K. Miyachi, "A robust decentralized joint control based on interference estimation," in *Proceedings of IEEE Int. Conf. on Robotics and Automation*, vol. 1, 1987, pp. 326–331.

[9] K. Ohnishi, M. Shibata, and T. Murakami, "Recent advances in motion control," *IEEE/ASME Transactions on mechatronics*, vol. 1, no. 1, pp. 56–67, 1996.

[10] D. E. Whitney, "Historical perspective and state of the art in robot force control," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 2, 1985, pp. 262–268.

[11] R. A. Rothchild and R. W. Mann, "An emg-controlled force sensing proportional rate elbow prosthesis," in *Proc. 1966 Symposium on Biomedical Engineering*, Milwaukee, 1966.

[12] H. A. Ernst, "Mh1-a computer operated mechanical hand," Sc. D., MIT, 1961.

[13] J. D. Barber, "Mantran: A symbolic language for supervisory control of an intelligent remote manipulator," S.M., MIT, 1967.

[14] J. W. Hill and A. J. Sword, "Manipulation based on sensor-directed control," in *17th Annual Human Factors Convention*, Washington, 1973.

[15] R. P. Paul and B. Shimano, "Compliance and control," in *Proc. JACC*, 1976, pp. 694–699.

[16] J. L. Nevins and D. E. Whitney, "The force vector assembler concept," in *Proceedings of First CISM-IFTOMM Symposium on Theory and Practice Of Robots and Manipulators*, Udine, Italy, 1973.

[17] D. E. Whitney, "Force feedback control of manipulator fine motions," *ASME Journal of Dyn. Sys. Meas. and Control*, pp. 91–97, 1977.

[18] R. C. Groome, "Force feedback steering of a teleoperator system," S.M., MIT, 1972.

[19] J. K. Salisbury, "Active stiffness control of a manipulator in cartesian coordinates," in *Proceedings of 19th IEEE Conference on Decision and Control*, 1980.

[20] S. J. K. and J. J. Craig, "Articulated hands: Force control and kinematic issues," *International Journal of Robotics Research*, vol. 1, no. 1, pp. 4–17, 1982.

[21] P. C. Watson, "A multidimensional system analysis of the assembly process as performed by a manipulator," in *1st N. Am. Robot Conference*, Chicago, 1976.

[22] S. K. Drake and S. N. Simunovic, "The use of compliance in a robot assembly system," in *IFAC symposium on Info. and Control Problems in Manufacturing Technologies*, Tokyo, 1977.

[23] N. Hogan, "Control of mechanical impedance of prosthetic arms," in *Proceedings of JACC*, San Francisco, 1980.

[24] H. Hanafusa and H. Asada, "A robot hand with elastic fingers and its application to assembly process," in *IFAC Symposium on Info. and Control Problems in Manufacturing Technologies*, Tokyo, 1977.

[25] P. Borrel, "Modele de comportements des manipulateurs, applications a l'analyse de leurs performances et a leur commande automatique," Ph.D. dissertation, Universite des Sciences et Techniques du Languedoc, Montpellier, France, 1979.

[26] M. H. Raibert and J. J. Craig, “Hybrid position/force control of manipulators,” *Trans. ASME Journal Dyn. Sys. Meas. and Control*, vol. 102, pp. 126–133, 1981.

[27] M. T. Mason, “Compliance and force control for computer controlled manipulators,” *Trans. IEEE Systems, Man and Cybernetics*, vol. 11, no. 6, pp. 418–432, 1981.

[28] P. L. M. Heydemann, “Determination and correction of quadrature fringe measurement errors in interferometers,” *Applied Optics*, vol. 20, pp. 3382–3384, 1981.

[29] K. K. Tan, S. N. Huang, and H. L. Seet, “Geometrical error compensation of precision motion systems using radial basis functions,” *IEEE Transactions on Instrumentation and Measurement*, vol. 49, no. 5, pp. 984–991, 2000.

[30] P. J. From, J. T. Gravdahl, T. Lillehagen, and P. Abbeel, “Motion planning and control of robotic manipulators on seaborne platforms,” *Control Engineering Practice*, vol. 19, pp. 809–819, 2011.

[31] Y. Hurmuzlu, F. Génot, and B. Brogliato, “Modeling, stability and control of biped robots-a general framework,” *Automatica*, vol. 40, pp. 1647–1664, 2004.

[32] M. Vucobratovic, *Walking robots and anthropomorphic mechanics.* Moskow: MIR Press, 1976.

[33] M. Vucobratovic, B. Borovac, D. Surla, and D. Stokic, *Scientific fundamentals of robotics 7: Biped locomotion.* New York: Springer, 1990.

[34] M. H. Raibert, *Legged robots that balance.* Cambridge, MA: MIT Press, 1986.

[35] D. J. Todd, *Walking machines: an introduction to legged robots.* London: Kogan Page, 1985.

[36] F. El Hafi and P. Gorce, "Behavioral approach for a bipedal robot stepping motion gait," *Robotica*, vol. 17, pp. 491–501, 1999.

[37] H. K. Lum, M. Zribi, and Y. C. Soh, "Planning and control of biped robot," *International Journal of Engineering Science*, vol. 37, pp. 1319–1349, 1999.

[38] M. Yagi and V. Lumelsky, "Local on-line planning in biped robot locomotion amongst unknown obstacles," *Robotica*, vol. 18, pp. 389–402, 2000.

[39] M. Rostami and G. Bessonnet, "Sagittal gait of a biped robot during the single support phase. part 2: Optimal motion," *Robotica*, vol. 19, pp. 241–253, 2001.

[40] C. L. Shih, "Ascending and descending stairs for a biped robot," *IEEE Transactions on Systems, Man and Cybernetics-Part A: Systems and Humans*, vol. 29, pp. 255–268, 1999.

[41] Q. Huang, K. Yokoi, S. Kajita, K. Kaneko, H. Arai, N. Koyachi, and K. Tanie, "Planning walking patterns for a biped robot," *IEEE Transactions on Robotics and Automation*, vol. 17, pp. 557–569, 2001.

[42] C. Chevallereau and Y. Aoustin, "Optimal reference trajectories for walking and running of a biped robot," *Robotica*, vol. 19, pp. 557–569, 2001.

[43] T. Saidouni and G. Bessonnet, "Generating globally optimized sagittal gait cycles of a biped robot," *Robotica*, vol. 21, pp. 199–210, 2003.

[44] H. Hemami and R. L. Farnsworth, "Postural and gait stability of a planar five link biped by simulation," *IEEE Transaction on Automatic Control*, vol. 22, pp. 452–458, 1977.

[45] B. Khosravi, S. Yurkovich, and H. Hemami, "Control of a five link biped in a back somersault maneuver," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 17, no. 2, pp. 303–325, 1987.

[46] P. S. Chudinov, "One problem of angular stabilization of bipedal locomotion," *Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 15, no. 6, pp. 49–54, 1980.

[47] P. S. Chudinov, "Problem of angular stabilization of bipedal locomotion," *Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 19, no. 1, pp. 166–169, 1984.

[48] E. K. Lavrovskii, "Impact phenomena in problems of control of bipedal locomotion," *Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 14, no. 5, pp. 41–47, 1979.

[49] E. K. Lavrovskii, "Dynamics of bipedal locomotion at high velocity," *Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 15, no. 4, pp. 50–58, 1980.

[50] V. V. Beletskii, "Dynamics of two legged walking," *II Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 10, no. 4, pp. 3–13, 1975.

[51] V. V. Beletskii and T. S. Kirsanova, “Plane linear models of biped locomotion,” *Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 11, no. 4, pp. 51–62, 1976.

[52] V. V. Beletskii and P. S. Chudinov, “The linear stabilization problem for two legged ambulation,” *Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 12, no. 6, pp. 65–74, 1977.

[53] V. V. Beletskii and P. S. Chudinov, “Control of motion of a bipedal walking robot,” *Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 15, no. 3, pp. 30–38, 1980.

[54] V. V. Beletskii, V. E. Berbyuk, and V. A. Samsonov, “Parametric optimization of motions of a bipedal walking robot,” *Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 17, no. 1, pp. 28–40, 1982.

[55] A. A. Grishin and A. M. Formal’sky, “Control of bipedal walking robot by means of impulses of finite amplitude,” *Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 25, no. 2, pp. 67–74, 1990.

[56] I. V. Novozhilov, “Control of three-dimensional motion of a bipedal walking robot,” *Izvestiya AN SSSR Mekhanika Tverdogo Tela*, vol. 19, no. 4, pp. 47–53, 1984.

[57] Y. Hurmuzlu, “Dynamics of bipedal gait; part i-objective functions and the contact event of a planar five link biped,part ii-stability analysis of a planar five link biped,” *ASME Journal of Applied Mechanics*, vol. 60, no. 2, pp. 331–344, 1993.

[58] T. H. Chang and Y. Hurmuzlu, "Sliding control without reaching phase and its application to bipedal locomotion," *ASME Journal of Dynamic Systems, Measurement and Control*, vol. 105, pp. 447–455, 1994.

[59] J. S. Yang, "A control study of a knee less biped locomotion system," *Journal of the Franklin Institute*, vol. 331b, no. 2, pp. 125–143, 1994.

[60] L. Jalics, H. Hemami, and B. Clymer, "A control strategy for terrain adaptive bipedal locomotion," *Autonomous Robots*, vol. 4, pp. 243–257, 1997.

[61] S. Kajita and K. Tani, "Experimental study of biped dynamic walking," *IEEE Control Systems*, vol. 16, pp. 13–19, 1996.

[62] S. Kajita, T. Yamaura, and A. Kobayashi, "Dynamic walking control of a biped robot along a potential energy conserving orbit," *IEEE Transactions on Robotics and Automation*, vol. 8, no. 4, pp. 431–438, 1992.

[63] A. A. Grishin, A. M. Formal'sky, A. V. Lensky, and S. V. Zhitomirsky, "Dynamic of a vehicle with two telescopic legs controlled by two drives," *The International Journal of Robotics Research*, vol. 13, no. 2, pp. 137–147, 1994.

[64] Y. F. Zheng and H. Hemami, "Impacts effects of biped contact with the environment," *IEEE Transactions on Systems, Man and Cybernetics*, vol. SMC-14, no. 3, pp. 437–443, 1984.

[65] H. Hemami, Y. F. Zheng, and M. J. Hines, "Initiation of walk and tiptoe of a planar nine link biped," *Mathematical Biosciences*, vol. 61, pp. 163–189, 1982.

[66] C. L. Golliday and H. Hemami, "An approach to analyzing biped locomotion dynamics and designing robot locomotion control," *IEEE Transactions on Automatic Control*, vol. 22, no. 6, pp. 963–972, 1977.

[67] F. Gubina, H. Hemami, and R. B. McGhee, "On the dynamic stability of biped locomotion," *IEEE Transactions on Biomedical Engineering*, vol. 21, no. 2, pp. 102–108, 1974.

[68] K. Mitobe, G. Capi, and Y. Nasu, "Control of walking robots based on manipulation of the zero moment point," *Robotica*, vol. 18, pp. 651–657, 2000.

[69] Y. J. Seo and Y. S. Yoon, "Design of a robust dynamic gait of the biped using the concept of dynamic stability margin," *Robotica*, vol. 13, pp. 461–468, 1995.

[70] E. Garcia, J. Estremera, and P. Gonzales de Santos, "A comparative study of stability margins for walking machines," *Robotica*, vol. 20, pp. 595–606, 2002.

[71] H. Hemami and A. Katbab, "Constrained inverted pendulum model for evaluating upright postural stability," *ASME Journal of Dynamic Systems, Measurement and Control*, vol. 104, pp. 343–349, 1982.

[72] T. T. Lee and J. H. Liao, "Trajectory planning and control of a 3-linked biped robot," in *Proceedings of IEEE Conference on robotics and automation*, New York, 1988, pp. 820–823.

[73] J. Song, K. H. Low, and W. Guo, "A simplified hybrid force/position controller method for the walking robots," *Robotica*, vol. 17, pp. 583–589, 1999.

[74] J. H. Park, "Impedance control for biped robot locomotion," *IEEE Transactions on Robotics and Automation*, vol. 17, pp. 870–883, 2001.

[75] G. Taga, "A model of the neuro-musculo-skeletal system for human locomotion. i. emergence of basic gait. ii. real-time adaptability under various constraints," *Biological Cybernetics*, vol. 73, pp. 97–121, 1995.

[76] C. Chevallereau, "Time-scaling control of an underactuated biped robot," *IEEE Transactions on Robotics and Automation*, vol. 19, no. 2, pp. 362–368, 2003.

[77] J. M. Bourgeot and B. Brogliato, "Tracking control of complementarity lagrangian systems," *International Journal of Bifurcations and Chaos, special issue non-smooth dynamical systems: Recent trends and perspectives*, 2005.

[78] B. Brogliato, S. I. Niculescu, and P. Orhant, "On the control of finite dimensional mechanical systems with unilateral constraints," *IEEE Transactions on Automatic Control*, vol. 42, no. 2, pp. 200–215, 1997.

[79] B. Brogliato, S. I. Niculescu, and M. Monteiro-Marques, "On the tracking control of a class of complementarity-slackness hybrid mechanical systems," *Systems and Control Letters*, vol. 39, pp. 255–266, 2000.

[80] A. Frank, "An approach to the dynamic analysis synthesis of biped locomotion machines," *Medical and Biological Engineering*, vol. 8, pp. 465–476, 1970.

[81] S. V. Rutkovskii, "Walking, skipping and running of a bipedal robot with allowance for impact," *Mechanics of Solids*, vol. 20, no. 5, pp. 44–49, 1985.

[82] P. H. Channon, S. H. Hopkins, and D. T. Pham, "Derivation of optimal walking motions for a bipedal walking robot," *Robotica*, vol. 10, pp. 165–172, 1992.

[83] V. V. Beletskii and Y. V. Bolotin, "Model estimation of the energetics of bipedal walking and running," *Mechanics of Solids*, vol. 18, no. 4, pp. 87–92, 1983.

[84] Y. V. Bolotin, "Energetically optimal gaits of a bipedal walking robot," *Mechanics of Solids*, vol. 19, no. 6, pp. 44–51, 1984.

[85] J. Furusho and A. Sano, "Sensor based control of a nine linked biped," *International Journal of Robotics Research*, vol. 9, no. 2, pp. 83–98, 1990.

[86] P. H. Channon, S. H. Hopkins, and D. T. Pham, "A gravity compensation technique for an n-legged robot," in *Proceedings of INSTN MECH ENGRS, ImechE*, vol. 210, 1996, pp. 1–14.

[87] P. H. Channon, S. H. Hopkins, and D. T. Pham, "Optimal control of an n-legged robot," *Journal of Systems and Control Engineering*, vol. 210, pp. 51–63, 1996.

[88] P. H. Channon, S. H. Hopkins, and D. T. Pham, "A variational approach to the optimization of gait for a bipedal robot," in *Proceedings of INSTN MECH ENGRS, ImechE*, vol. 210, 1996, pp. 177–186.

[89] H. van der Kooij, R. Jacobs, B. Koopman, and F. van der Halm, "An alternative approach to synthesizing bipedal walking," in *Biological Cybernetics*, vol. 88, no. 1, 2003, pp. 46–59.

[90] G. A. Pratt, “Legged robots at mit: What’s new since raibert,” *IEEE Robotics and Automation Magazine*, vol. 7, no. 3, pp. 15–19, 2000.

[91] E. Dunn and R. D. Howe, “Toward smooth bipedal walking,” in *IEEE International Conferences on Robotics and Automation*, vol. 3, San Diego, CA, 1994, pp. 2489–2494.

[92] H. Miura and I. Shimoyama, “Dynamic walking of a biped,” *International Journal of Robotics Research*, vol. 3, no. 2, pp. 60–74, 1984.

[93] J. W. Grizzle, G. Abba, and F. Plestan, “Proving asymptotic stability of a walking cycle for a five dof biped robot model,” in *Second International Conference on Climbing and Walking Robots*, Portsmouth, UK, 1999, pp. 69–81.

[94] J. W. Grizzle, G. Abba, and F. Plestan, “Asymptotically stable walking for biped robots: Analysis via systems with impulse effects,” *IEEE Transactions on Automatic Control*, vol. 46, no. 1, pp. 51–64, 2001.

[95] E. R. Werstelvelt, J. W. Grizzle, and D. E. Koditschek, “Hybrid zero dynamics of planar biped walkers,” *IEEE Transactions on Automatic Control*, vol. 48, no. 1, pp. 42–56, 2003.

[96] J. Wu, D. Pu, and H. Ding, “Adaptive robust motion control of siso nonlinear systems with implementation on linear motors,” *Mechatronics*, vol. 17, pp. 263–270, 2007.

[97] F. J. Lin, S. Y. Chen, P. H. Chou, and P. H. Shieh, “Interval type-2 fuzzy neural network control for x-y-theta motion control stage using linear ultrasonic motors,” *Neurocomputing*, vol. 72, pp. 1138–1151, 2009.

[98] [Online]. Available: www.jfwk.com/what_is.html

[99] D. Riehle, “Framework design: A role modeling approach,” Ph.D. dissertation, Swiss Federal Institute of Technology, ZURICH, 2000.

[100] [Online]. Available: http://www.evolution.com

[101] [Online]. Available: http://www.skilligent.com/

[102] [Online]. Available: http://www.gostai.com

[103] [Online]. Available: http://www.cyberbotics.com

[104] [Online]. Available: http://www.microsoft.com/robotics/

[105] [Online]. Available: http://www.orocos.org

[106] [Online]. Available: http://playerstage.sourceforge.net/

[107] [Online]. Available: http://www.ros.org

[108] Tiobe programming community index for june 2012. [Online]. Available: http://www.tiobe.com/

[109] W. L. Hürsch and C. V. Lopes, “Separation of concerns,” Tech. Rep., 1995.

[110] [Online]. Available: http://www.dspace.org/

[111] O. S. Wolfbeis, “Fiber-optic chemical sensors and biosensors,” *Analytical Chemistry*, vol. 72, no. 12, pp. 81–89, 2000.

[112] [Online]. Available: http://ac.cqupt.edu.cn/asp/course/sensor/upfile/en_ch/Intro%20to%20sensors.pdf

[113] R. Luo, "Sensor technologies and microsensor issues for mechatronics systems," *IEEE/ASME Transactions on Mechatronics*, vol. 1, pp. 39–49, 1996.

[114] *WorldDispZacement/Proximity/PositionSensors*, Mountain View, CA, 1990.

[115] (2010, december). [Online]. Available: http://www.maritimejournal.com/features101/onboard-systems/monitoring-and-control/load-cell-testing-gets-straight-to-the-point

[116] (2010, September). [Online]. Available: http://www.straightpoint.com/news/24-test-machine-launched.html

[117] P. Jänker, M. Christmann, F. Hermle, T. Lorkowski, and S. Storm, "Mechatronics using piezoelectric actuators," *Journal of the European Ceramic Society*, vol. 19, pp. 1127–1131, 1999.

[118] H. Ishihara, F. Arai, and T. Fukuda, "Micro mechatronics and micro actuators," *IEEE/ASME Transactions on Mechatronics*, vol. 1, no. 1, pp. 68–79, March 1996.

[119] J. Puranen, "Induction motor versus permanent magnet synchronous motor in motion control applications: A comparative study," Doctor of Science, Lappeenranta University of Technology, Lappeenranta, Finland, December 2006.

[120] K. Y. Kim, K. H. Park, H. C. Park, N. S. Goo, and K. J. Yoon, "Performance evaluation of lightweight piezo-composite actuators," *Sensors and Actuators*, vol. A 120, pp. 123–129, 2005.

[121] Noliac. (2012) Piezo actuator drive. [Online]. Available: www.noliac.com

[122] D. Song and J. Li, "Modeling of piezo actuator's nonlinear and frequency dependent dynamics," *Mechatronics*, vol. 9, pp. 391–410, 1999.

[123] C. A. Palanduz, I. Sauciuc, and R. Paydar, "Piezo actuator for cooling," patent no. 7,638,928 B2, December 2009.

[124] K. D. Nguyen and I. Chen, T. Ng, "On algorithms for planning s-curve motion profiles," *International Journal of Advanced Robot Systems*, vol. 5, no. 1, pp. 99–106, 2008.

[125] [Online]. Available: http://opencv.org/

[126] [Online]. Available: http://www.mvtec.com/halcon/

[127] E. D. Kunt, K. Cakir, and A. Sabanovic, "A workstation for micro assembly," *Mediterranean Conference on Control and Automation*, July 2007.

[128] L. Dozio and P. Mantegazza, "Linux real time application interface (rtai) in low cost high performance motion control," *Motion Control*, 2007.

[129] A. Neto, F. Sartori, F. Piccolo, A. Barbalace, R. Vitelli, and H. Fernandes, "Linux real-time framework for fusion devices," *Fusion Engineering and Design*, vol. 84, pp. 1408–1411, February 2009.

[130] H. Bilen and M. Unel, "Micromanipulation using a micro assembly workstation with vision and force sensing," *International Conference on Intelligent Computing*, 2008.

[131] H. Bilen, M. A. Hocaoglu, E. A. Baran, M. Unel, and D. Gozuacik, "Novel parameter estimation schemes in microsystems," *International Conference on Robotics and Automation*, pp. 2394–2399, 2009.

[132] A. Naskali, E. Kunt, and A. Sabanovic, "Bilevel modularity concept within a robotic assembly module of a microfactory setting," *International Journal of Advanced Manufacturing Technologies*, 2012.

[133] E. D. Kunt, "Microfactory concept with bilevel modularity," Ph.D. dissertation, Sabanci University, 2012.

[134] A. E. Baran, "Disturbance observer based bilateral control systems," Master's thesis, Sabanci University, 2010.

[135] A. Şabanoviç, K. Ohnishi, D. Yashiro, N. Şabanoviç, and E. A. Baran, "Motion control systems with network delay," *Automatika*, vol. 51, no. 2, pp. 119–126, 2010.

[136] E. A. Baran, E. Goluboviç, and A. Şabanoviç, "A new functional observer to estimate velocity, acceleration and disturbance for motion control systems," in *IEEE International Symposium on Industrial Electronics*, Bari Italy, 2010, pp. 384–389.

Biography

Ahmet Teoman Naskali received his B.S. degree in computer engineering from Galatasaray University in 2003 and Ms degree in Mechatronics from Sabanci University in 2005, he is currently a PhD candidate in Sabanci University Mechatronics. His interests include networked control systems and real-time systems.

A APPENDIX A

Documentation of software is almost as vital as the implementation of the software. As without the information to use the functions and modules within, the work put in to it can not be beneficial to other.

The software for the framework was documented with doxygen.

SU Framework

Generated by Doxygen 1.8.1.2

Thu Aug 2 2012 11:58:55

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 Actuator_ Struct Reference

Actuator structure contains all the parameters states IO channels and function pointers to an actuator structure.

`#include <Actuator.h>`

Public Attributes

- char ∗ **name**

  Name of the actuator.
- char ∗ **units**

  String containing units of actuator to be used for reporting.
- double **input** [**ACTMAXIN**]

  List of inputs to the actuator.
- double **output** [**ACTMAXOUT**]

  List of outputs of the actuator.
- double **state** [**ACTMAXSTATES**]

  States of the actuator, internal data that the actuator may need.
- double **parameter** [**ACTMAXPARAMETERS**]

  Parameters of the actuator, may include coefficients of offsets.
- int **outDIO** [**ACTMAXIO**]

  List of the digital output channels that the actuator may have.
- int **inDIO** [**ACTMAXIO**]

  List of the digital input channels that the actuator may have.
- int **inADC** [**ACTMAXIO**]

  List of the analog input channels that the actuator may have.
- int **outDAC** [**ACTMAXIO**]

  List of the analog output channels that the actuator may have.
- int **inENC** [**ACTMAXIO**]

  List of the encoder channels that the actuator may have.
- int(∗ **init**)(struct **Actuator_** ∗d)

  initialize the actuator
- int(∗ **write**)(struct **Actuator_** ∗d, double value)

  write to the actuator
- int(∗ **remove**)(struct **Actuator_** ∗d)

  remove actuator
- int(∗ **reset**)(struct **Actuator_** ∗d)

reset actuator

- int(∗ **setParameter**)(struct **Actuator_** ∗d, int **parameter**, double value)

  set a parameter of the actuator

3.1.1 Detailed Description

Actuator structure contains all the parameters states IO channels and function pointers to an actuator structure.

Definition at line 19 of file Actuator.h.

3.1.2 Member Data Documentation

3.1.2.1 int Actuator_::inADC[ACTMAXIO]

List of the analog input channels that the actuator may have.

Definition at line 32 of file Actuator.h.

3.1.2.2 int Actuator_::inDIO[ACTMAXIO]

List of the digital input channels that the actuator may have.

Definition at line 31 of file Actuator.h.

3.1.2.3 int Actuator_::inENC[ACTMAXIO]

List of the encoder channels that the actuator may have.

Definition at line 34 of file Actuator.h.

3.1.2.4 int(∗ Actuator_::init)(struct Actuator_ ∗d)

initialize the actuator

Function pointer to initialize the actuator Initialize the necessary inputs and outputs to be utilized for this actuators

Parameters

d	pointer to the actuator structure

Returns

0 for success, -1 for error

Definition at line 43 of file Actuator.h.

3.1.2.5 double Actuator_::input[ACTMAXIN]

List of inputs to the actuator.

Definition at line 24 of file Actuator.h.

3.1.2.6 char∗ Actuator_::name

Name of the actuator.

Definition at line 21 of file Actuator.h.

3.1.2.7 int Actuator_::outDAC[ACTMAXIO]

List of the analog output channels that the actuator may have.

Definition at line 33 of file Actuator.h.

3.1.2.8 int Actuator_::outDIO[ACTMAXIO]

List of the digital output channels that the actuator may have.

Definition at line 30 of file Actuator.h.

3.1.2.9 double Actuator_::output[ACTMAXOUT]

List of outputs of the actuator.

Definition at line 25 of file Actuator.h.

3.1.2.10 double Actuator_::parameter[ACTMAXPARAMETERS]

Parameters of the actuator, may include coefficients of offsets.

Definition at line 28 of file Actuator.h.

3.1.2.11 int(∗ Actuator_::remove)(struct Actuator_ ∗d)

remove actuator

Function pointer to remove function of the actuator This function performs the necessary cleanup operations of the actuator such as releasing any resources that have been taken up by the actuator, this function is called before the end of the application or when the actuator is removed from the application

Parameters

d	pointer to the actuator structure

Returns

0 for success, -1 for error

Definition at line 59 of file Actuator.h.

3.1.2.12 int(∗ Actuator_::reset)(struct Actuator_ ∗d)

reset actuator

Function pointer to reset the actuator This function performs the necessary operations to reset the actuator should it be needed to reset

Parameters

d	pointer to the actuator structure

Returns

0 for success, -1 for error

Definition at line 66 of file Actuator.h.

3.1.2.13 int(∗ Actuator_::setParameter)(struct Actuator_ ∗d, int parameter, double value)

set a parameter of the actuator

Function pointer to set parameters of the actuator This function is used to set the parameters of the actuator

Parameters

d	pointer to the actuator structure
parameter,number	of the parameter ID to be written to the actuator
value	to be written to the parameter, parameters may be offsets or coefficients to be utilized in the actuator

Returns

0 for success, -1 for error

Definition at line 75 of file Actuator.h.

3.1.2.14 double Actuator_::state[ACTMAXSTATES]

States of the actuator, internal data that the actuator may need.

Definition at line 27 of file Actuator.h.

3.1.2.15 char∗ Actuator_::units

String containing units of actuator to be used for reporting.

Definition at line 22 of file Actuator.h.

3.1.2.16 int(∗ Actuator_::write)(struct Actuator_ ∗d, double value)

write to the actuator

Function pointer to write to the actuator This function performs the necessary conversions and outputs to the IO cards to fulfill the actuators function

Parameters

d	pointer to the actuator structure
value	to be written to the actuator, the units of the value are stored in the unit field of the actuator structure

Returns

0 for success, -1 for error

Definition at line 52 of file Actuator.h.

The documentation for this struct was generated from the following file:

- Framework/Actuator/**Actuator.h**

3.2 Axis_ Struct Reference

```
#include <Axis.h>
```

Public Attributes

- char ∗ **name**
- **Sensor Sensors** [**AXISSENSORNUM**]
- **Actuator Actuators** [**AXISACTUATORNUM**]
- **Filter Filters** [**AXISFILTERNUM**]
- **Controller Controllers** [**AXISCONTROLLERNUM**]
- double **input** [**AXISINPUTNUM**]
- double **output** [**AXISOUTPUTNUM**]
- double **parameter** [**AXISPARAMTERNUM**]

  Parameters of the Mechanism, used to configure the Mechanism.
- double **state** [**AXISSTATENUM**]

  States of the Mechanism, used to store internal data of the Mechanism.
- int **enable**
- int(∗ **init**)(struct **Axis_** ∗a)
- int(∗ **step**)(struct **Axis_** ∗a, double **reference**)
- int(∗ **remove**)(struct **Axis_** ∗a)
- int(∗ **reset**)(struct **Axis_** ∗a)

3.2.1 Detailed Description

Definition at line 23 of file Axis.h.

3.2.2 Member Data Documentation

3.2.2.1 Actuator Axis_::Actuators[AXISACTUATORNUM]

Definition at line 28 of file Axis.h.

3.2.2.2 Controller Axis_::Controllers[AXISCONTROLLERNUM]

Definition at line 31 of file Axis.h.

3.2.2.3 int Axis_::enable

Definition at line 41 of file Axis.h.

3.2.2.4 Filter Axis_::Filters[AXISFILTERNUM]

Definition at line 29 of file Axis.h.

3.2.2.5 int(∗ Axis_::init)(struct Axis_ ∗a)

Definition at line 43 of file Axis.h.

3.2.2.6 double Axis_::input[AXISINPUTNUM]

Definition at line 34 of file Axis.h.

3.2.2.7 char∗ Axis_::name

Definition at line 25 of file Axis.h.

3.2.2.8 double Axis_::output[AXISOUTPUTNUM]

Definition at line 35 of file Axis.h.

3.2.2.9 double Axis_::parameter[AXISPARAMTERNUM]

Parameters of the Mechanism, used to configure the Mechanism.

Definition at line 37 of file Axis.h.

3.2.2.10 int(∗ Axis_::remove)(struct Axis_ ∗a)

Definition at line 45 of file Axis.h.

3.2.2.11 int(∗ Axis_::reset)(struct Axis_ ∗a)

Definition at line 46 of file Axis.h.

3.2.2.12 Sensor Axis_::Sensors[AXISSENSORNUM]

Definition at line 27 of file Axis.h.

3.2.2.13 double Axis_::state[AXISSTATENUM]

States of the Mechanism, used to store internal data of the Mechanism.

Definition at line 38 of file Axis.h.

3.2.2.14 int(∗ Axis_::step)(struct Axis_ ∗a, double reference)

Definition at line 44 of file Axis.h.

The documentation for this struct was generated from the following file:

- Framework/Axis/**Axis.h**

3.3 Controller_ Struct Reference

Controller Structure contains the necessary data to run a controller. It is composed of four main parts. Input and Output arrays are utilized to exchange data to and from the controller structure during runtime. States of a controller contain the internal varying data and parameters of the controller contain the values are used to tune the controller.

```
#include <Controller.h>
```

Public Attributes

- int **linked**

parameter to define if a controller works under linked mode or normal mode, in normal mode, the control algorithm receives its inputs directly from its input array whereas in linked mode, the input array's pointers are directed to other parameters in a system and the controller can gather its data from outside.

- double **input** [**CONTROLLERMAXIN**]

  List of inputs to the controller.
- double ∗ **input_p** [**CONTROLLERMAXIN**]

  List of inputs in pointer format to directly add references.
- double **output** [**CONTROLLERMAXOUT**]

  List of outputs of the controller.
- double **state** [**CONTROLLERMAXSTATES**]

  States of the controller, internal data that the actuator may need.
- double **parameter** [**CONTROLLERMAXPARAMETERS**]

  Parameters of the controller, may include coefficients or offsets.
- void(∗ **loopController**)(struct **Controller_** ∗c)

  This function is the algorithm of the controller, it is set by hihger level structures to run once per loop. Every different algorithm using the Controller structure must provide a loop function.

3.3.1 Detailed Description

Controller Structure contains the necessary data to run a controller. It is composed of four main parts. Input and Output arrays are utilized to exchange data to and from the controller structure during runtime. States of a controller contain the internal varying data and parameters of the controller contain the values are used to tune the controller.

Definition at line 18 of file Controller.h.

3.3.2 Member Data Documentation

3.3.2.1 double Controller_::input[CONTROLLERMAXIN]

List of inputs to the controller.

Definition at line 22 of file Controller.h.

3.3.2.2 double∗ Controller_::input_p[CONTROLLERMAXIN]

List of inputs in pointer format to directly add references.

Definition at line 24 of file Controller.h.

3.3.2.3 int Controller_::linked

parameter to define if a controller works under linked mode or normal mode, in normal mode, the control algorithm receives its inputs directly from its input array whereas in linked mode, the input array's pointers are directed to other parameters in a system and the controller can gather its data from outside.

Definition at line 20 of file Controller.h.

3.3.2.4 void(∗ Controller_::loopController)(struct Controller_ ∗c)

This function is the algorithm of the controller, it is set by hihger level structures to run once per loop. Every different algorithm using the Controller structure must provide a loop function.

this is the function pointer to the algorithm of the controller

Parameters

c	is a pointer to the Controller structure that contains the inputs, outputs, states and parameters of the controller

Returns

Function does not have a return value, the controller outputs are stored int the output array.

Definition at line 38 of file Controller.h.

3.3.2.5 double Controller_::output[CONTROLLERMAXOUT]

List of outputs of the controller.

Definition at line 26 of file Controller.h.

3.3.2.6 double Controller_::parameter[CONTROLLERMAXPARAMETERS]

Parameters of the controller, may include coefficients or offsets.

Definition at line 29 of file Controller.h.

3.3.2.7 double Controller_::state[CONTROLLERMAXSTATES]

States of the controller, internal data that the actuator may need.

Definition at line 28 of file Controller.h.

The documentation for this struct was generated from the following file:

- Framework/Controller/**Controller.h**

3.4 Filter Struct Reference

Public Attributes

- string **name**
- double **P** [**PARAMCOUNT**]

3.4.1 Detailed Description

Definition at line 13 of file FilterTemplate.c.

3.4.2 Member Data Documentation

3.4.2.1 string Filter::name

Definition at line 14 of file FilterTemplate.c.

3.4.2.2 double Filter::P[PARAMCOUNT]

Definition at line 15 of file FilterTemplate.c.

The documentation for this struct was generated from the following file:

- Framework/Filter/**FilterTemplate.c**

3.5 Filter_ Struct Reference

```
#include <Filter.h>
```

Public Attributes

- char ∗ **name**
- double **P** [**PARAMCOUNT**]

3.5.1 Detailed Description

Definition at line 16 of file Filter.h.

3.5.2 Member Data Documentation

3.5.2.1 char∗ Filter_::name

Definition at line 17 of file Filter.h.

3.5.2.2 double Filter_::P[PARAMCOUNT]

Definition at line 18 of file Filter.h.

The documentation for this struct was generated from the following file:

- Framework/Filter/**Filter.h**

3.6 inputCommunication Struct Reference

```
#include <CommunicationTemplate.h>
```

Public Attributes

- double **ConveyorSpeed**
- int **ConveyorStation**

3.6.1 Detailed Description

Definition at line 4 of file CommunicationTemplate.h.

3.6.2 Member Data Documentation

3.6.2.1 double inputCommunication::ConveyorSpeed

Definition at line 5 of file CommunicationTemplate.h.

3.6.2.2 int inputCommunication::ConveyorStation

Definition at line 6 of file CommunicationTemplate.h.

The documentation for this struct was generated from the following file:

- Framework/Communication/**CommunicationTemplate.h**

3.7 Kinematcis_ Struct Reference

Kinematics Structure contains the necessary data to run Kinematics. It is composed of four main parts. Input and Output arrays are utilized to exchange data to and from the Kinematics structure during runtime. States of Kinematics contain the internal varying data and parameters of Kinematics contain the values that are used to configure the Kinematics structure.

```
#include <Kinematics.h>
```

Public Attributes

- int **linked**

  parameter to define if kinematics works under linked mode or normal mode, in normal mode, the control algorithm receives its inputs directly from its input array whereas in linked mode, the input array's pointers are directed to other parameters in a system and the Kinematics can gather its data from outside.
- double **input** [**KINEMATICSMAXIN**]

  List of inputs to the Kinematics function.
- double ∗ **input_p** [**KINEMATICSMAXIN**]

  List of inputs in pointer format to directly add references.
- double **output** [**KINEMATICSMAXOUT**]

  List of outputs of the Kinematics function.
- double **state** [**KINEMATICSMAXSTATES**]

  States of the Kinematics funtion, internal data that the kinematics function may need.
- double **parameter** [**KINEMATICSMAXPARAMETERS**]

  Parameters of the Kinematics, may include coefficients or offsets.
- void(∗ **loopFWKinematics**)(struct Kinematics_ ∗k)

  This function is the algorithm of the forward Kinematics, it is set by hihger level structures to run once per loop. Every different algorithm using the Kinematics structure must provide a loop function.

3.7.1 Detailed Description

Kinematics Structure contains the necessary data to run Kinematics. It is composed of four main parts. Input and Output arrays are utilized to exchange data to and from the Kinematics structure during runtime. States of Kinematics contain the internal varying data and parameters of Kinematics contain the values that are used to configure the Kinematics structure.

Definition at line 18 of file Kinematics.h.

3.7.2 Member Data Documentation

3.7.2.1 double Kinematcis_::input[KINEMATICSMAXIN]

List of inputs to the Kinematics function.

Definition at line 22 of file Kinematics.h.

3.7.2.2 double∗ Kinematcis_::input_p[KINEMATICSMAXIN]

List of inputs in pointer format to directly add references.

Definition at line 24 of file Kinematics.h.

3.7.2.3 int Kinematcis_::linked

parameter to define if kinematics works under linked mode or normal mode, in normal mode, the control algorithm receives its inputs directly from its input array whereas in linked mode, the input array's pointers are directed to other parameters in a system and the Kinematics can gather its data from outside.

Definition at line 20 of file Kinematics.h.

3.7.2.4 void(∗ Kinematcis_::loopFWKinematics)(struct Kinematics_ ∗k)

This function is the algorithm of the forward Kinematics, it is set by hihger level structures to run once per loop. Every different algorithm using the Kinematics structure must provide a loop function.

this is the function pointer to the algorithm of the Kinematics

Parameters

c	is a pointer to the Kinematics structure that contains the inputs, outputs, states and parameters of the Kinematics function

Returns

Function does not have a return value, the Kinematics outputs are stored int the output array.

Definition at line 38 of file Kinematics.h.

3.7.2.5 double Kinematcis_::output[KINEMATICSMAXOUT]

List of outputs of the Kinematics function.

Definition at line 26 of file Kinematics.h.

3.7.2.6 double Kinematcis_::parameter[KINEMATICSMAXPARAMETERS]

Parameters of the Kinematics, may include coefficients or offsets.

Definition at line 29 of file Kinematics.h.

3.7.2.7 double Kinematcis_::state[KINEMATICSMAXSTATES]

States of the Kinematics funtion, internal data that the kinematics function may need.

Definition at line 28 of file Kinematics.h.

The documentation for this struct was generated from the following file:

- Framework/Kinematics/**Kinematics.h**

3.8 Mechanism_ Struct Reference

Structure containing the data for an mechanism, including inputs, outputs, state machine state, links to the actuators, sensors, axes, controllers plus, parameters and states for internal storage of data.

```
#include <Mechanism.h>
```

Public Attributes

- char ∗ **name**
  name of the actuator
- int **StateMachine**
  state machine status of the mechanism
- double **input** [**MECHINPUT**]
- double **output** [**MECHOUTPUT**]
- double **parameter** [**MECHPARAMETERS**]
  Parameters of the Mechanism, used to configure the Mechanism.
- double **state** [**MECHSTATES**]
  States of the Mechanism, used to store internal data of the Mechanism.
- **Actuator Actuators** [**MECHACTUATORS**]
- **Sensor Sensors** [**MECHSENSORS**]
- **Axis Axes** [**MECHAXES**]
- int(∗ **loopMechanism**)(struct **Mechanism_** ∗M)

3.8.1 Detailed Description

Structure containing the data for an mechanism, including inputs, outputs, state machine state, links to the actuators, sensors, axes, controllers plus, parameters and states for internal storage of data.

Definition at line 26 of file Mechanism.h.

3.8.2 Member Data Documentation

3.8.2.1 Actuator Mechanism_::Actuators[MECHACTUATORS]

Definition at line 40 of file Mechanism.h.

3.8.2.2 Axis Mechanism_::Axes[MECHAXES]

Definition at line 43 of file Mechanism.h.

3.8.2.3 double Mechanism_::input[MECHINPUT]

Definition at line 32 of file Mechanism.h.

3.8.2.4 int(∗ Mechanism_::loopMechanism)(struct Mechanism_ ∗M)

Definition at line 49 of file Mechanism.h.

3.8.2.5 char∗ Mechanism_::name

name of the actuator

Definition at line 28 of file Mechanism.h.

3.8.2.6 double Mechanism_::output[MECHOUTPUT]

Definition at line 33 of file Mechanism.h.

3.8.2.7 double Mechanism_::parameter[MECHPARAMETERS]

Parameters of the Mechanism, used to configure the Mechanism.

Definition at line 35 of file Mechanism.h.

3.8.2.8 Sensor Mechanism_::Sensors[MECHSENSORS]

Definition at line 41 of file Mechanism.h.

3.8.2.9 double Mechanism_::state[MECHSTATES]

States of the Mechanism, used to store internal data of the Mechanism.

Definition at line 36 of file Mechanism.h.

3.8.2.10 int Mechanism_::StateMachine

state machine status of the mechanism

Definition at line 30 of file Mechanism.h.

The documentation for this struct was generated from the following file:

- Framework/Mechanism/**Mechanism.h**

3.9 outputCommunication Struct Reference

```
#include <CommunicationTemplate.h>
```

Public Attributes

- int **x**

3.9.1 Detailed Description

Definition at line 12 of file CommunicationTemplate.h.

3.9.2 Member Data Documentation

3.9.2.1 int outputCommunication::x

Definition at line 13 of file CommunicationTemplate.h.

The documentation for this struct was generated from the following file:

- Framework/Communication/**CommunicationTemplate.h**

3.10 Sensor_ Struct Reference

Sensor structure contains all the parameters and states of the sesor. The sensor sturcutre also contains the input and output channel numbers for a sensor along with function pointers to functions of the sensor to initialize, read, reset and delete the sensor.

```
#include <Sensor.h>
```

Public Attributes

- char ∗ **name**

  String representing the name of the sensor.
- char ∗ **unit**

  String containing the units of the sensor.
- int **outDIO** [**SENSMAXIO**]

  List of the digital output channels that the sensor may have.
- int **inDIO** [**SENSMAXIO**]

  List of the digital input channels that the sensor may have.
- int **inADC** [**SENSMAXIO**]

  List of the analog input channels that the sensor may have.
- int **outDAC** [**SENSMAXIO**]

  List of the analog output channels that the sensor may have.
- int **inENC** [**SENSMAXIO**]

  List of the encoder channels that the sensor may have.
- double inputs double **state** [**SENSMAXSTATES**]

  States of the sensor, internal data to keep log of values or other necessary information on a sensor.
- double **parameter** [**SENSMAXPARAMETERS**]

  Parameters of the sensor, may include coefficients of offsets.
- double(∗ **read**)(struct **Sensor_** ∗s)

  Function pointer to the function that will perform reading a sensor.
- int(∗ **setParameter**)(struct **Sensor_** ∗s, int **parameter**, double value)

  Function pointer to the function that will set a parameter of the sensor. This function is used to set the different parameters of a sensor.
- int(∗ **reset**)(struct **Sensor_** ∗s)

  Function pointer to the function that will reset a sensor. May be used in case of sensor errors.
- int(∗ **remove**)(struct **Sensor_** ∗s)

  Function pointer to the function that removes a sensor. This function is called when the application is shutting down or when the sensor is no longer needed and is removed.

3.10.1 Detailed Description

Sensor structure contains all the parameters and states of the sesor. The sensor sturcutre also contains the input and output channel numbers for a sensor along with function pointers to functions of the sensor to initialize, read, reset and delete the sensor.

Definition at line 22 of file Sensor.h.

3.10.2 Member Data Documentation

3.10.2.1 int Sensor_::inADC[SENSMAXIO]

List of the analog input channels that the sensor may have.

Definition at line 29 of file Sensor.h.

3.10.2.2 int Sensor_::inDIO[SENSMAXIO]

List of the digital input channels that the sensor may have.

Definition at line 28 of file Sensor.h.

3.10.2.3 int Sensor_::inENC[SENSMAXIO]

List of the encoder channels that the sensor may have.

Definition at line 31 of file Sensor.h.

3.10.2.4 char∗ Sensor_::name

String representing the name of the sensor.

Definition at line 24 of file Sensor.h.

3.10.2.5 int Sensor_::outDAC[SENSMAXIO]

List of the analog output channels that the sensor may have.

Definition at line 30 of file Sensor.h.

3.10.2.6 int Sensor_::outDIO[SENSMAXIO]

List of the digital output channels that the sensor may have.

Definition at line 27 of file Sensor.h.

3.10.2.7 double Sensor_::parameter[SENSMAXPARAMETERS]

Parameters of the sensor, may include coefficients of offsets.

Definition at line 36 of file Sensor.h.

3.10.2.8 double(∗ Sensor_::read)(struct Sensor_ ∗s)

Function pointer to the function that will perform reading a sensor.

Definition at line 38 of file Sensor.h.

3.10.2.9 int(∗ Sensor_::remove)(struct Sensor_ ∗s)

Function pointer to the function that removes a sensor. This function is called when the application is shutting down or when the sensor is no longer needed and is removed.

Definition at line 42 of file Sensor.h.

3.10.2.10 int(∗ Sensor_::reset)(struct Sensor_ ∗s)

Function pointer to the function that will reset a sensor. May be used in case of sensor errors.

Definition at line 41 of file Sensor.h.

3.10.2.11 int(∗ Sensor_::setParameter)(struct Sensor_ ∗s, int parameter, double value)

Function pointer to the function that will set a parameter of the sensor. This function is used to set the different parameters of a sensor.

Definition at line 39 of file Sensor.h.

3.10.2.12 double inputs double Sensor_::state[SENSMAXSTATES]

States of the sensor, internal data to keep log of values or other necessary information on a sensor.

Definition at line 35 of file Sensor.h.

3.10.2.13 char∗ Sensor_::unit

String containing the units of the sensor.

Definition at line 25 of file Sensor.h.

The documentation for this struct was generated from the following file:

- Framework/Sensor/**Sensor.h**

Chapter 4

File Documentation

4.1 Framework/Actuator/Actuator.h File Reference

General definition of an actuator containing the data structures. An actuator structure Contains the staes and parameters of an actuator along with the channel numbers for the IO's. The actuator structure also has function pointers to attach initialization, writing, removal, resetting and parameter setting functions.

Classes

- struct **Actuator_**

  Actuator structure contains all the parameters states IO channels and function pointers to an actuator structure.

Macros

- #define **ACTMAXIO** 3

  Defines the number of maximum different types input or output IO channel access by a single actuator.
- #define **ACTMAXIN** 3

  Defines the maximum number of inputs all actuators may have.
- #define **ACTMAXOUT** 3

  Defines the maximum number of outputs all actuators may have.
- #define **ACTMAXSTATES** 5

  Define the maximum number of states that all actuator may have.
- #define **ACTMAXPARAMETERS** 4

  Define the maximum number of parameters that all actuators may have.

Typedefs

- typedef struct **Actuator_ Actuator**

Functions

- int **setupActuator** (**Actuator** ∗d, char ∗**name**, char ∗units, int(∗init)(**Actuator** ∗d), int(∗write)(**Actuator** ∗d, double value), int(∗remove)(**Actuator** ∗d), int(∗reset)(**Actuator** ∗d), int(∗setParameter)(**Actuator** ∗d, int parameter, double value))

  setup actuator

4.1.1 Detailed Description

General definition of an actuator containing the data structures. An actuator structure Contains the staes and parameters of an actuator along with the channel numbers for the IO's. The actuator structure also has function pointers to attach initialization, writing, removal, resetting and parameter setting functions.

Definition in file **Actuator.h**.

4.1.2 Macro Definition Documentation

4.1.2.1 #define ACTMAXIN 3

Defines the maximum number of inputs all actuators may have.

Definition at line 11 of file Actuator.h.

4.1.2.2 #define ACTMAXIO 3

Defines the number of maximum different types input or output IO channel access by a single actuator.

Definition at line 10 of file Actuator.h.

4.1.2.3 #define ACTMAXOUT 3

Defines the maximum number of outputs all actuators may have.

Definition at line 12 of file Actuator.h.

4.1.2.4 #define ACTMAXPARAMETERS 4

Define the maximum number of parameters that all actuators may have.

Definition at line 14 of file Actuator.h.

4.1.2.5 #define ACTMAXSTATES 5

Define the maximum number of states that all actuator may have.

Definition at line 13 of file Actuator.h.

4.1.3 Typedef Documentation

4.1.3.1 typedef struct Actuator_ Actuator

Definition at line 79 of file Actuator.h.

4.1.4 Function Documentation

4.1.4.1 int setupActuator (Actuator ∗ *d,* char ∗ *name,* char ∗ *units,* int(∗)(Actuator ∗d) *init,* int(∗)(Actuator ∗d, double value) *write,* int(∗)(Actuator ∗d) *remove,* int(∗)(Actuator ∗d) *reset,* int(∗)(Actuator ∗d, int parameter, double value) *setParameter*)

setup actuator

Function to setup the actuator This function is used to setup the fields of the actuator structure

Parameters

d	pointer to the actuator structure
name	contains name of the actuator as a string
units	contains the units of the actuator as a string
init	funtion pointer to the function to initialize the actuator
write	function pointer to the function to write to the actuator
remove	function pointer to the function to remove or delete the actuator
reset	function pointer to the function to reset the actuator
setParameter	function pointer to the function to set the parameters of the actuator

Returns

0 for success, -1 for error

Definition at line 94 of file Actuator.h.

4.2 Framework/Actuator/ActuatorTemplate.h File Reference

```
#include "Actuator.h"
#include "HWWrapper.h"
```

Functions

- int **initActuator** (**Actuator** ∗d)

  Add access to the IO cards of the system.
- int **writeActuator** (**Actuator** ∗d, double value)
- int **removeActuator** (**Actuator** ∗d)
- int **resetActuator** (**Actuator** ∗d)
- int **setActuatorParameter** (**Actuator** ∗d, int parameter, double value)

4.2.1 Function Documentation

4.2.1.1 int initActuator (Actuator ∗ *d*)

Add access to the IO cards of the system.

To Enable the functions of an actuator these parts should be filled Initializes the actuator.

Parameters

d	pointer to an actuator structure as defined in **Actuator.h** (p. 21)

Write initialization code for the actuator

Definition at line 18 of file ActuatorTemplate.h.

4.2.1.2 int removeActuator (Actuator ∗ *d*)

Removes resources allocated to the acutator if they exist.

Parameters

d	pointer to an actuator structure as defined in **Actuator.h** (p. 21)

Remove resources here

Definition at line 43 of file ActuatorTemplate.h.

4.2.1.3 int resetActuator (Actuator ∗ *d*)

Reset the actuator

Parameters

d	pointer to an actuator structure as defined in **Actuator.h** (p. 21)

Reset the actuator, utilized in case of an error. Offsets may be initialized here

Definition at line 52 of file ActuatorTemplate.h.

4.2.1.4 int setActuatorParameter (Actuator ∗ *d*, int *parameter*, double *value*)

Sets the offset for the actuator.

Parameters

d	pointer to an actuator structure as defined in **Actuator.h** (p. 21)

Definition at line 62 of file ActuatorTemplate.h.

4.2.1.5 int writeActuator (Actuator ∗ *d*, double *value*)

Makes the Actuator write its output to its output medium. This funciton outputs to the IO cards, the wrapper template must be included in the project beforehand

Parameters

d	pointer to an actuator structure as defined in **Actuator.h** (p. 21)
value	value to be output to the io card

Scale the input value and output it to the IO card

Perform necessary conversions

Definition at line 32 of file ActuatorTemplate.h.

4.3 Framework/Actuator/Examples/ConveyorStopper.h File Reference

```
#include "Actuator.h"
#include "HWWrapper.h"
```

Functions

- int **initConveyorStopper** (**Actuator** ∗d)
- int **writeConveyorStopper** (**Actuator** ∗d, double value)
- int **removeConveyorStopper** (**Actuator** ∗d)

- int **resetConveyorStopper** (**Actuator** ∗d)
- int **setConveyorStopperParameter** (**Actuator** ∗d, int parameter, double value)

4.3.1 Detailed Description

This file is the implementation of the solenoid actuator to stop trays at the positions of a conveyor. Any solenoids may be implemented as actuators using this file

Definition in file **ConveyorStopper.h**.

4.3.2 Function Documentation

4.3.2.1 int initConveyorStopper (**Actuator** ∗ *d*)

Addition of the actuator header file Add access to the IO cards of the system Initializes the Stopper.

Parameters

d	pointer to an actuator structure as defined in **Actuator.h** (p. 21)

Write initialization code for the actuator

Definition at line 17 of file ConveyorStopper.h.

4.3.2.2 int removeConveyorStopper (**Actuator** ∗ *d*)

Removes resources allocated to the acutator if they exist.

Parameters

d	pointer to an actuator structure as defined in **Actuator.h** (p. 21)

No resources to be remvoed

Definition at line 45 of file ConveyorStopper.h.

4.3.2.3 int resetConveyorStopper (**Actuator** ∗ *d*)

Reset the actuator

Parameters

d	pointer to an actuator structure as defined in **Actuator.h** (p. 21)

Conveyor stoppers do not have a reset so simply the output is set to low

Definition at line 54 of file ConveyorStopper.h.

4.3.2.4 int setConveyorStopperParameter (**Actuator** ∗ *d,* int *parameter,* double *value*)

Sets a parameter of the conveyor stopper.

Parameters

d	pointer to an actuator structure as defined in **Actuator.h** (p. 21)

The conveyor stopper does not have a value

Definition at line 64 of file ConveyorStopper.h.

4.3.2.5 int writeConveyorStopper (**Actuator** ∗ *d,* double *value*)

Makes the Actuator write its output to its output medium. This funciton outputs to the IO cards, the wrapper template must be included in the project beforehand

Parameters

d	pointer to an actuator structure as defined in **Actuator.h** (p. 21)
value	value to be output to the io card

Scale the input value and output it to the IO card

Perform necessary conversions

Definition at line 29 of file ConveyorStopper.h.

4.4 Framework/Axis/Axis.h File Reference

```
#include "Actuator.h"
#include "Sensor.h"
#include "Filter.h"
#include "Controller.h"
```

Classes

- struct **Axis_**

Macros

- #define **AXISSENSORNUM** 1
- #define **AXISACTUATORNUM** 1
- #define **AXISFILTERNUM** 1
- #define **AXISOBSERVERNUM** 1
- #define **AXISCONTROLLERNUM** 1
- #define **AXISPARAMTERNUM** 10
- #define **AXISSTATENUM** 5
- #define **AXISINPUTNUM** 6
- #define **AXISOUTPUTNUM** 6

Typedefs

- typedef struct **Axis_ Axis**

Functions

- int **setupAxis** (**Axis** ∗a, char ∗**name**, int(∗init)(struct **Axis_** ∗a), int(∗step)(struct **Axis_** ∗a, double **reference**), int(∗remove)(struct **Axis_** ∗a), int(∗reset)(struct **Axis_** ∗a))

4.4.1 Macro Definition Documentation

4.4.1.1 #define AXISACTUATORNUM 1

Definition at line 13 of file Axis.h.

4.4.1.2 #define AXISCONTROLLERNUM 1

Definition at line 16 of file Axis.h.

4.4.1.3 #define AXISFILTERNUM 1

Definition at line 14 of file Axis.h.

4.4.1.4 #define AXISINPUTNUM 6

Definition at line 19 of file Axis.h.

4.4.1.5 #define AXISOBSERVERNUM 1

Definition at line 15 of file Axis.h.

4.4.1.6 #define AXISOUTPUTNUM 6

Definition at line 20 of file Axis.h.

4.4.1.7 #define AXISPARAMTERNUM 10

Definition at line 17 of file Axis.h.

4.4.1.8 #define AXISSENSORNUM 1

Definition at line 12 of file Axis.h.

4.4.1.9 #define AXISSTATENUM 5

Definition at line 18 of file Axis.h.

4.4.2 Typedef Documentation

4.4.2.1 typedef struct Axis_ Axis

Definition at line 51 of file Axis.h.

4.4.3 Function Documentation

4.4.3.1 int setupAxis (Axis ∗ *a,* char ∗ *name,* int(∗)(struct Axis_ ∗a) *init,* int(∗)(struct Axis_ ∗a, double reference) *step,* int(∗)(struct Axis_ ∗a) *remove,* int(∗)(struct Axis_ ∗a) *reset*)

Definition at line 53 of file Axis.h.

4.5 Framework/Axis/AxisTemplate.c File Reference

```
#include "Axis.h"
```

Functions

- int **removeAxis** (struct **Axis_** ∗a)
- int **resetAxis** (struct **Axis_** ∗a)
- int **initAxis** (**Axis** ∗A)
- int **stepAxis** (**Axis** ∗A, double ref)

4.5.1 Function Documentation

4.5.1.1 int initAxis (Axis ∗ *A*)

Definition at line 12 of file AxisTemplate.c.

4.5.1.2 int removeAxis (struct Axis_ ∗ *a*)

Definition at line 7 of file AxisTemplate.c.

4.5.1.3 int resetAxis (struct Axis_ ∗ *a*)

Definition at line 9 of file AxisTemplate.c.

4.5.1.4 int stepAxis (Axis ∗ *A,* double *ref*)

Definition at line 18 of file AxisTemplate.c.

4.6 Framework/Axis/Examples/AxisFaulhaberLinear.c File Reference

```
#include "Axis.h"
```

Functions

- int **removeAxis** (**Axis** ∗a)
- int **resetAxis** (**Axis** ∗a)
- int **initAxis** (**Axis** ∗A)
- int **stepAxis** (**Axis** ∗A, double ref)

4.6.1 Function Documentation

4.6.1.1 int initAxis (Axis ∗ *A*)

Definition at line 12 of file AxisFaulhaberLinear.c.

4.6.1.2 int removeAxis (Axis ∗ *a*)

Definition at line 7 of file AxisFaulhaberLinear.c.

4.6.1.3 int resetAxis (Axis ∗ *a*)

Definition at line 9 of file AxisFaulhaberLinear.c.

4.6.1.4 int stepAxis (Axis ∗ *A*, double *ref*)

Definition at line 18 of file AxisFaulhaberLinear.c.

4.7 Framework/Communication/Communication.h File Reference

This file defines the communication for the system.

Functions

- int **InputCommunication** ()
- int **OutputCommunication** ()
- int **send** (int ID, void ∗data, int size)
- int **receive** (int ID, void ∗data, int size)

4.7.1 Detailed Description

This file defines the communication for the system.

Definition in file **Communication.h**.

4.7.2 Function Documentation

4.7.2.1 int InputCommunication ( )

Definition at line 20 of file CommunicationTemplate.h.

4.7.2.2 int OutputCommunication ( )

Definition at line 35 of file CommunicationTemplate.h.

4.7.2.3 int receive (int *ID*, void ∗ *data*, int *size*)

Definition at line 47 of file CommunicationTemplate.h.

4.7.2.4 int send (int *ID*, void ∗ *data*, int *size*)

Definition at line 43 of file CommunicationTemplate.h.

4.8 Framework/Communication/CommunicationTemplate.h File Reference

Classes

- struct **inputCommunication**
- struct **outputCommunication**

Functions

- int **InputCommunication** ()
- int **OutputCommunication** ()
- int **send** (int ID, void ∗data, int size)
- int **receive** (int ID, void ∗data, int size)

Variables

- struct **inputCommunication inCommunication**
- struct **outputCommunication outCommunication**

4.8.1 Function Documentation

4.8.1.1 int InputCommunication ( )

Definition at line 20 of file CommunicationTemplate.h.

4.8.1.2 int OutputCommunication ( )

Definition at line 35 of file CommunicationTemplate.h.

4.8.1.3 int receive (int *ID*, void ∗ *data*, int *size*)

Definition at line 47 of file CommunicationTemplate.h.

4.8.1.4 int send (int *ID*, void ∗ *data*, int *size*)

Definition at line 43 of file CommunicationTemplate.h.

4.8.2 Variable Documentation

4.8.2.1 struct inputCommunication inCommunication

4.8.2.2 struct outputCommunication outCommunication

4.9 Framework/Controller/Controller.h File Reference

General definition of a controller containing the data structures.

Classes

- struct **Controller_**

  Controller Structure contains the necessary data to run a controller. It is composed of four main parts. Input and Output arrays are utilized to exchange data to and from the controller structure during runtime. States of a controller contain the internal varying data and parameters of the controller contain the values are used to tune the controller.

Macros

- #define **CONTROLLERMAXPARAMETERS** 10
- #define **CONTROLLERMAXSTATES** 10
- #define **CONTROLLERMAXIN** 10
- #define **CONTROLLERMAXOUT** 10

Typedefs

- typedef struct **Controller_ Controller**

4.9.1 Detailed Description

General definition of a controller containing the data structures. A controller structure contains the states and parameters a controller. The controller structure also has a function pointer to attach an algorithm loop function that is executed in every loop.

Definition in file **Controller.h**.

4.9.2 Macro Definition Documentation

4.9.2.1 #define CONTROLLERMAXIN 10

Definition at line 14 of file Controller.h.

4.9.2.2 #define CONTROLLERMAXOUT 10

Definition at line 15 of file Controller.h.

4.9.2.3 #define CONTROLLERMAXPARAMETERS 10

Definition at line 11 of file Controller.h.

4.9.2.4 #define CONTROLLERMAXSTATES 10

Definition at line 12 of file Controller.h.

4.9.3 Typedef Documentation

4.9.3.1 typedef struct Controller_ Controller

Definition at line 41 of file Controller.h.

4.10 Framework/Controller/ControllerTemplate.h File Reference

This is a template file to generate a controller.

```
#include "Controller.h"
```

Enumerations

- enum **ControllerParameters** { **parameter1**, **parameter2** }

  Prefix Use a 3 letter prefix for the customizations.
- enum **ControllerStates** { **SMC_error_old**, **SMC_sigma_old**, **SMC_u_old** }

  Controller states, used to store data between iterations of the loop to obtain derivative values.
- enum **ControllerInput** { **reference**, **position** }

  Inputs to the control function.
- enum **ControllerOutput** { **u** }

  Outputs of the control algorithm.

Functions

- void **loopController** (**Controller** ∗c)
- void **setControllerParameters** (**Controller** ∗c)

  Set controllers parameters and time constant. Parameters may be configured with a separate function.

4.10.1 Detailed Description

This is a template file to generate a controller. A controller is implemented in this file. The Controller is based around the Controller structure, int receives its inputs from the input array and it writes its outputs to the output array. The states of the controller c∗

Definition in file **ControllerTemplate.h**.

4.10.2 Enumeration Type Documentation

4.10.2.1 enum ControllerInput

Inputs to the control function.

Enumerator:

reference

position

Definition at line 26 of file ControllerTemplate.h.

4.10.2.2 enum ControllerOutput

Outputs of the control algorithm.

Enumerator:

u Control output of control function.

Definition at line 33 of file ControllerTemplate.h.

4.10.2.3 enum ControllerParameters

Prefix Use a 3 letter prefix for the customizations.

Controller parameters, used to configure the controller

Enumerator:

parameter1

parameter2

Definition at line 13 of file ControllerTemplate.h.

4.10.2.4 enum ControllerStates

Controller states, used to store data between iterations of the loop to obtain derivative values.

Enumerator:

SMC_error_old

SMC_sigma_old

SMC_u_old

Definition at line 19 of file ControllerTemplate.h.

4.10.3 Function Documentation

4.10.3.1 void loopController (Controller * *c*)

Control algorithm

Parameters

c	pointer to a controller structure

Returns

return values are contained in the Controller structures output array

Definition at line 45 of file ControllerTemplate.h.

4.10.3.2 void setControllerParameters (Controller * *c*)

Set controllers parameters and time constant. Parameters may be configured with a separate function.

Definition at line 52 of file ControllerTemplate.h.

4.11 Framework/Controller/Examples/CengizController.h File Reference

```
#include "Controller.h"
```

Macros

- #define **PARAMCOUNT** 20

- #define **ORDER** 5

Enumerations

- enum **ProportionalControllerParameters** { **Kp** }

Functions

- void **loopProportional** (**Controller** ∗c, double ∗current, double ∗ref, double ∗output)
- void **initProportionalController** (**Controller** ∗c, double KpVal)

4.11.1 Macro Definition Documentation

4.11.1.1 #define ORDER 5

Definition at line 6 of file CengizController.h.

4.11.1.2 #define PARAMCOUNT 20

Definition at line 5 of file CengizController.h.

4.11.2 Enumeration Type Documentation

4.11.2.1 enum ProportionalControllerParameters

Enumerator:

Kp

Definition at line 8 of file CengizController.h.

4.11.3 Function Documentation

4.11.3.1 void initProportionalController (Controller ∗ *c*, double *KpVal*)

Definition at line 25 of file CengizController.h.

4.11.3.2 void loopProportional (Controller ∗ *c*, double ∗ *current*, double ∗ *ref*, double ∗ *output*)

Definition at line 10 of file CengizController.h.

4.12 Framework/Controller/Examples/PControl.c File Reference

```
#include "Controller.h"
#include <stdio.h>
```

4.13 Framework/Controller/Examples/SMCController.h File Reference

Implementation of the Sliding Mode Controller.

```
#include "Controller.h"
```

Enumerations

- enum **SMC_Parameters** { **SMC_C**, **SMC_D**, **SMC_Ku**, **SMC_DT** }

  < Included to obtain the Controller struct.
- enum **SMC_States** { **SMC_error_old**, **SMC_sigma_old**, **SMC_u_old** }

  Sliding mode States, used to store data between iterations of the loop to obtain derivative values.
- enum **SMC_Input** { **SMC_reference**, **SMC_position** }

  Inputs to the sliding mode function.
- enum **SMC_Output** { **SMC_u** }

  Outputs of the sliding mode control algorithm.

Functions

- void **loopSMCController** (**Controller** ∗c)
- void **setSMCParameters** (**Controller** ∗c, double C, double D, double Ku, double dt)

  Set sliding mode controllers parameters and time constant.

4.13.1 Detailed Description

Implementation of the Sliding Mode Controller. A sliding mode control function is implemented in this file. The sliding mode function has one algorithm attached to it and it utilizes the inputs and outputs to exchange data. The function can be utilized in two different modes, normal mode and linked mode. If normal mode is used, the inputs to the data structures must be made using the input array. If linked mode is to be used, the input_p array must be filled with pointers to the values that the SMC controller will use.

Definition in file **SMCController.h**.

4.13.2 Enumeration Type Documentation

4.13.2.1 enum SMC_Input

Inputs to the sliding mode function.

Enumerator:

SMC_reference

SMC_position

Definition at line 29 of file SMCController.h.

4.13.2.2 enum SMC_Output

Outputs of the sliding mode control algorithm.

Enumerator:

SMC_u Control output of sliding mode control function.

Definition at line 36 of file SMCController.h.

4.13.2.3 enum SMC_Parameters

< Included to obtain the Controller struct.

Prefix SMC Sliding mode control parameters, used to configure the controller

Enumerator:

SMC_C

SMC_D

SMC_Ku

SMC_DT

Definition at line 14 of file SMCController.h.

4.13.2.4 enum SMC_States

Sliding mode States, used to store data between iterations of the loop to obtain derivative values.

Enumerator:

SMC_error_old

SMC_sigma_old

SMC_u_old

Definition at line 22 of file SMCController.h.

4.13.3 Function Documentation

4.13.3.1 void loopSMCController (Controller ∗ *c*)

Sliding mode control algorithm

Parameters

c	pointer to a controller structure

Returns

return values are contained in the controller structures output array

Store states of the controller for the next loop

Definition at line 48 of file SMCController.h.

4.13.3.2 void setSMCParameters (Controller ∗ *c,* double *C,* double *D,* double *Ku,* double *dt*)

Set sliding mode controllers parameters and time constant.

Definition at line 96 of file SMCController.h.

4.14 Framework/Filter/Filter.h File Reference

Classes

- struct **Filter_**

Macros

- #define **PARAMCOUNT** 20
- #define **ORDER** 5

Typedefs

- typedef struct **Filter_ Filter**

Functions

- void **initFilter** (**Filter** ∗F, double P1, double P2, double P3)
- void **FilterAlg** (**Filter** F, double in[ ], double out[ ])
- void **deleteFilter** (**Filter** ∗F)

4.14.1 Macro Definition Documentation

4.14.1.1 #define ORDER 5

Definition at line 14 of file Filter.h.

4.14.1.2 #define PARAMCOUNT 20

Definition at line 13 of file Filter.h.

4.14.2 Typedef Documentation

4.14.2.1 typedef struct Filter_ Filter

Definition at line 22 of file Filter.h.

4.14.3 Function Documentation

4.14.3.1 void deleteFilter (Filter ∗ *F*)

Definition at line 33 of file Filter.h.

4.14.3.2 void FilterAlg (Filter *F*, double *in[ ]*, double *out[ ]*)

Definition at line 28 of file Filter.h.

4.14.3.3 void initFilter (Filter ∗ *F*, double *P1*, double *P2*, double *P3*)

Definition at line 24 of file Filter.h.

4.15 Framework/Filter/FilterTemplate.c File Reference

Classes

- struct **Filter**

Macros

- #define **PARAMCOUNT** 20
- #define **ORDER** 5

Functions

- struct **Filter initFilter** (struct **Filter** ∗F, double P1, double P2, double P3)
- void **FilterAlg** (double in[ ], double out[ ], struct **Filter** F)
- void **deleteFilter** (struct **Filter** ∗F)

Variables

- string **name**
- double **P** [**PARAMCOUNT**]

4.15.1 Macro Definition Documentation

4.15.1.1 #define ORDER 5

Definition at line 11 of file FilterTemplate.c.

4.15.1.2 #define PARAMCOUNT 20

Definition at line 10 of file FilterTemplate.c.

4.15.2 Function Documentation

4.15.2.1 void deleteFilter (struct Filter ∗ *F*)

Definition at line 28 of file FilterTemplate.c.

4.15.2.2 void FilterAlg (double *in[ ],* double *out[ ],* struct Filter *F*)

Definition at line 23 of file FilterTemplate.c.

4.15.2.3 struct Filter initFilter (struct Filter ∗ *F,* double *P1,* double *P2,* double *P3*)

Definition at line 19 of file FilterTemplate.c.

4.15.3 Variable Documentation

4.15.3.1 string name

Definition at line 20 of file FilterTemplate.c.

4.15.3.2 double P[PARAMCOUNT]

Definition at line 21 of file FilterTemplate.c.

4.16 Framework/Kinematics/Examples/DeltaFWKinematics.h File Reference

Implementation of the Delta Robot Kinematics.

```
#include "Kinematics.h"
#include <math.h>
```

Enumerations

- enum **DeltaFWKinematicsParameters** {
  DeltaLA, **DeltaLB**, **DeltaRA**, **DeltaRB**,
  DeltaTheta1, **DeltaTheta2**, **DeltaTheta3**, **DeltaZofset** }

  < Included to obtain the Kinematics struct.
- enum **DeltaFWKinematicsInput** { **DeltaFWAlpha1**, **DeltaFWAlpha2**, **DeltaFWAlpha3** }

  Inputs to the Delta robot forward kinematics array.
- enum **DeltaFWKinematicsOutput** { **DeltaFWX**, **DeltaFWY**, **DeltaFWZ** }

  Outputs of the delta robot forward kinematics.

Functions

- void **loopDeltaFWKinematics** (**Kinematics** ∗k)

Variables

- enum **DeltaFWKinematicsParameters initFilter**

4.16.1 Detailed Description

Implementation of the Delta Robot Kinematics. Forward kinematics of a delta robot are implemented in this file ∗

Definition in file **DeltaFWKinematics.h**.

4.16.2 Enumeration Type Documentation

4.16.2.1 enum DeltaFWKinematicsInput

Inputs to the Delta robot forward kinematics array.

Enumerator:

DeltaFWAlpha1	alpha1 coordinate in joint space, input
DeltaFWAlpha2	alpha2 cooridnate in joint space, input
DeltaFWAlpha3	alpha3 coordinate in joint space, input

Definition at line 33 of file DeltaFWKinematics.h.

4.16.2.2 enum DeltaFWKinematicsOutput

Outputs of the delta robot forward kinematics.

Enumerator:

DeltaFWX	X coordinate in task space.
DeltaFWY	Y coordinate in task space.
DeltaFWZ	Z coordinate in task space.

Definition at line 41 of file DeltaFWKinematics.h.

4.16.2.3 enum DeltaFWKinematicsParameters

< Included to obtain the Kinematics struct.

Prefix DeltaFWKinematics Robot paramters, used to configure the Kinematics DeltaKInematicsParameters, this enum defines the pramters array, which contains the arm lengths, base redius, ancelle raidus, arm orientations and zoffset.

Enumerator:

DeltaLA	Upper Arm Lenght.
DeltaLB	Lower Arm Length.
DeltaRA	Base Radius.
DeltaRB	Nacelle Radius.
DeltaTheta1	Arm orientation angle 1.
DeltaTheta2	Arm orientation angle 2.
DeltaTheta3	Arm orientation angle 3.
DeltaZofset	Offset for the nozzle int he vertical axis.

Definition at line 16 of file DeltaFWKinematics.h.

4.16.3 Function Documentation

4.16.3.1 void loopDeltaFWKinematics (Kinematics ∗ *k*)

Delta Robot Forward Kinematics

Parameters

c	pointer to a Kinematics structure

Returns

return values are contained in the Kinematics structures output array

Definition at line 54 of file DeltaFWKinematics.h.

4.16.4 Variable Documentation

4.16.4.1 enum DeltaFWKinematicsParameters initFilter

4.17 Framework/Kinematics/Kinematics.h File Reference

General definition of forward or reverse kinematics, containing the data structures.

Classes

- struct **Kinematcis_**

  Kinematics Structure contains the necessary data to run Kinematics. It is composed of four main parts. Input and Output arrays are utilized to exchange data to and from the Kinematics structure during runtime. States of Kinematics contain the internal varying data and parameters of Kinematics contain the values that are used to configure the Kinematics structure.

Macros

- #define **KINEMATICSMAXPARAMETERS** 10
- #define **KINEMATICSMAXSTATES** 10
- #define **KINEMATICSMAXIN** 10
- #define **KINEMATICSMAXOUT** 10

Typedefs

- typedef struct Kinematics_ **Kinematics**

4.17.1 Detailed Description

General definition of forward or reverse kinematics, containing the data structures. A Kinematics structure contains the states and parameters of funtions needed to perform Kinematics operations. The Kinematics structure also has a function pointer to attach an algorithm loop function that is executed in every loop.

Definition in file **Kinematics.h**.

4.17.2 Macro Definition Documentation

4.17.2.1 #define KINEMATICSMAXIN 10

Definition at line 14 of file Kinematics.h.

4.17.2.2 #define KINEMATICSMAXOUT 10

Definition at line 15 of file Kinematics.h.

4.17.2.3 #define KINEMATICSMAXPARAMETERS 10

Definition at line 11 of file Kinematics.h.

4.17.2.4 #define KINEMATICSMAXSTATES 10

Definition at line 12 of file Kinematics.h.

4.17.3 Typedef Documentation

4.17.3.1 typedef struct Kinematics_ Kinematics

Definition at line 42 of file Kinematics.h.

4.18 Framework/Kinematics/KinematicsTemplate.h File Reference

This is a template file to generate Kinematics structure.

```
#include "Kinematics.h"
```

Enumerations

- enum **KinematicsParameters** { **parameter1**, **parameter2** }

  Prefix Use a 3 letter prefix for the customizations.
- enum **KinematicsStates** { **state1**, **state2** }

  Kinematics states, used to store data between iterations of the loop to obtain derivative values.
- enum **KinematicsInput** { **reference**, **position** }

  Inputs to the control function.
- enum **KinematicsOutput** { **Kinematics_x**, **Kinematics_y**, **Kinematics_z** }

  Outputs of the control algorithm.

Functions

- void **loopKinematics** (**Kinematics** ∗c)
- void **setKinematicsParameters** (**Kinematics** ∗c)

  Set Kinematicss parameters. Parameters may be configured with a separate function. Optional function.

4.18.1 Detailed Description

This is a template file to generate Kinematics structure. A Kinematics function is implemented in this file. The Kinematics function is based around the Kinematics structure, int receives its inputs from the input array and it writes its outputs to the output array. The states and parameters of the Kinematics funciton are also stored

Definition in file **KinematicsTemplate.h**.

4.18.2 Enumeration Type Documentation

4.18.2.1 enum KinematicsInput

Inputs to the control function.

Enumerator:

reference

position

Definition at line 26 of file KinematicsTemplate.h.

4.18.2.2 enum KinematicsOutput

Outputs of the control algorithm.

Enumerator:

Kinematics_x	
Kinematics_y	
Kinematics_z	Control output of control function.

Definition at line 33 of file KinematicsTemplate.h.

4.18.2.3 enum KinematicsParameters

Prefix Use a 3 letter prefix for the customizations.

Kinematics parameters, used to configure the Kinematics

Enumerator:

parameter1

parameter2

Definition at line 13 of file KinematicsTemplate.h.

4.18.2.4 enum KinematicsStates

Kinematics states, used to store data between iterations of the loop to obtain derivative values.

Enumerator:

state1

state2

Definition at line 19 of file KinematicsTemplate.h.

4.18.3 Function Documentation

4.18.3.1 void loopKinematics (Kinematics ∗ *c*)

Kinematics algorithm

Parameters

c	pointer to a Kinematics structure

Returns

return values are contained in the Kinematics structures output array

Definition at line 47 of file KinematicsTemplate.h.

4.18.3.2 void setKinematicsParameters (Kinematics ∗ *c*)

Set Kinematicss parameters. Parameters may be configured with a separate function. Optional function.

Definition at line 54 of file KinematicsTemplate.h.

4.19 Framework/Mechanism/Examples/ConveyorMechanism.h File Reference

This file creates the conveyor mechanism that is utilized in a Micro Factory. The conveyor has 1 motor, 3 stoppers and 3 sensors to sense trays at the stopper locations.

```
#include "Mechanism.h"
#include "ConveyorStopper.h"
#include "ConveyorMotor.h"
#include "ConveyorStation.h"
```

Enumerations

- enum **ConveyorMechanismActuators** { **CM_Stopper1**, **CM_Stopper2**, **CM_Stopper3**, **CM_Motor** }

  Enumeration of the conveyors actuators.
- enum **ConveyorMechanismSensors** { **CM_Station1**, **CM_Station2**, **CM_Station3** }

  Enumeration of the conveyors sensors.
- enum **ConveyorMechanismInputs** { **ConveyorSpeed**, **ConveyorStation** }

  MechanismParameters define what the parameters of this mechanism.
- enum **ConveyorMechanismStates** { **Station** }

  MechanismStates define what the states of this mechanims are.
- enum **ConveyorStateMachineStates** { **CM_Inactive**, **CM_Initialize**, **CM_MoveTo**, **CM_Move** }

  MechanismStateMachineStates are the states of the State Machine within the mechanism. The state machine of the mechanims is utilized to command the mechanism from one state to the other. The default states of the state machine are inactive and initialize. Depending on the needs of the mechanism the states should also include following position references and following speed references.

Functions

- int **loopConveyorMechanism** (**Mechanism** ∗M)
- int **setupConveyor** (**Mechanism** ∗m)

  Setup Converor function is used to assign values and fill the actuators within the mechanism structure.
- int **setConveyorMoveTo** (**Mechanism** ∗M, int station, double speed)

4.19.1 Detailed Description

This file creates the conveyor mechanism that is utilized in a Micro Factory. The conveyor has 1 motor, 3 stoppers and 3 sensors to sense trays at the stopper locations.

Definition in file **ConveyorMechanism.h**.

4.19.2 Enumeration Type Documentation

4.19.2.1 enum ConveyorMechanismActuators

Enumeration of the conveyors actuators.

Enumerator:

CM_Stopper1

CM_Stopper2

CM_Stopper3

CM_Motor

Definition at line 18 of file ConveyorMechanism.h.

4.19.2.2 enum ConveyorMechanismInputs

MechanismParameters define what the parameters of this mechanism.

Enumerator:

ConveyorSpeed

ConveyorStation

Definition at line 36 of file ConveyorMechanism.h.

4.19.2.3 enum ConveyorMechanismSensors

Enumeration of the conveyors sensors.

Enumerator:

CM_Station1

CM_Station2

CM_Station3

Definition at line 26 of file ConveyorMechanism.h.

4.19.2.4 enum ConveyorMechanismStates

MechanismStates define what the states of this mechanims are.

Enumerator:

Station

Definition at line 42 of file ConveyorMechanism.h.

4.19.2.5 enum ConveyorStateMachineStates

MechanismStateMachineStates are the states of the State Machine within the mechanism. The state machine of the mechanims is utilized to command the mechanism from one state to the other. The default states of the state machine are inactive and initialize. Depending on the needs of the mechanism the states should also include following position references and following speed references.

Enumerator:

CM_Inactive

CM_Initialize

CM_MoveTo

CM_Move fill in the other states

Definition at line 48 of file ConveyorMechanism.h.

4.19.3 Function Documentation

4.19.3.1 int loopConveyorMechanism (**Mechanism** $*$ *M*)

Perform initialization of actuators sensors axes etc...

get reference for the manipulator

Transform the coordinate references to axis references

Command axees

Definition at line 56 of file ConveyorMechanism.h.

4.19.3.2 int setConveyorMoveTo (**Mechanism** $*$ *M,* int *station,* double *speed*)

Definition at line 122 of file ConveyorMechanism.h.

4.19.3.3 int setupConveyor (**Mechanism** $*$ *m*)

Setup Converor function is used to assign values and fill the actuators within the mechanism structure.

Definition at line 103 of file ConveyorMechanism.h.

4.20 Framework/Mechanism/Mechanism.h File Reference

General definition of an mechanism containing the data structures. This file contains the data structure needed to create a mechanism. The data of a mechanism includes its states, its parameters and other modules it utilizes, such as Actuators, Sensors, Axes. The mechanism structure also has access to Kinematics, Controllers, Estimators and Observers.

```
#include "Actuator.h"
#include "Sensor.h"
#include "Axis.h"
```

Classes

- struct **Mechanism_**

  Structure containing the data for an mechanism, including inputs, outputs, state machine state, links to the actuators, sensors, axes, controllers plus, parameters and states for internal storage of data.

Macros

- #define **MECHAXES** 5

  Maximum Number of Axes a mechanism may have.
- #define **MECHKINEMATICS** 2

  Maximum Number of Kinematics functions a Mechanism may have.
- #define **MECHACTUATORS** 7

  Maximum Number of Actuators a Mechansim may have.
- #define **MECHSENSORS** 3

  Maximum Number of Sensors a Mechanism may have.
- #define **MECHINPUT** 6

  Maximum Number of References a Mechanism may have.
- #define **MECHOUTPUT** 6

- #define **MECHSTATES** 10

    Maximum Number of States a Mechansim may have.
- #define **MECHPARAMETERS** 10

    Maximum Number of Parameters a Mechanism may have.

Typedefs

- typedef struct **Mechanism_ Mechanism**

4.20.1 Detailed Description

General definition of an mechanism containing the data structures. This file contains the data structure needed to create a mechanism. The data of a mechanism includes its states, its parameters and other modules it utilizes, such as Actuators, Sensors, Axes. The mechanism structure also has access to Kinematics, Controllers, Estimators and Observers.

Definition in file **Mechanism.h**.

4.20.2 Macro Definition Documentation

4.20.2.1 #define MECHACTUATORS 7

Maximum Number of Actuators a Mechansim may have.

Definition at line 15 of file Mechanism.h.

4.20.2.2 #define MECHAXES 5

Maximum Number of Axes a mechanism may have.

Definition at line 13 of file Mechanism.h.

4.20.2.3 #define MECHINPUT 6

Maximum Number of References a Mechanism may have.

Definition at line 18 of file Mechanism.h.

4.20.2.4 #define MECHKINEMATICS 2

Maximum Number of Kinematics functions a Mechanism may have.

Definition at line 14 of file Mechanism.h.

4.20.2.5 #define MECHOUTPUT 6

Definition at line 19 of file Mechanism.h.

4.20.2.6 #define MECHPARAMETERS 10

Maximum Number of Parameters a Mechanism may have.

Definition at line 22 of file Mechanism.h.

4.20.2.7 #define MECHSENSORS 3

Maximum Number of Sensors a Mechanism may have.

Definition at line 16 of file Mechanism.h.

4.20.2.8 #define MECHSTATES 10

Maximum Number of States a Mechansim may have.

Definition at line 21 of file Mechanism.h.

4.20.3 Typedef Documentation

4.20.3.1 typedef struct Mechanism_ Mechanism

Definition at line 53 of file Mechanism.h.

4.21 Framework/Mechanism/MechanismTemplate.h File Reference

This file is a template for the creation of a mechanism. The functions within must be fulfilled and the state machine configured in order to provide functionnality to the mechanism.

```
#include "Mechanism.h"
```

Enumerations

- enum **MechanismParameters**

  MechanismParameters define what the parameters of this mechanism.
- enum **MechanismStates** { **StateMachine** }

  MechanismStates define what the states of this mechanims are.
- enum **MechanismStateMachineStates**

  MechanismStateMachineStates are the states of the State Machine within the mechanism. The state machine of the mechanims is utilized to command the mechanism from one state to the other. The default states of the state machine are inactive and initialize. Depending on the needs of the mechanism the states should also include following position references and following speed references.

Functions

- enum **MechanismStateMachineStates loopMechanism** (**Mechanism** ∗M, double ref[MANIPULATORREF])

Variables

- **initialize**
- **inactive**
- **loop**

  fill in the other states

4.21.1 Detailed Description

This file is a template for the creation of a mechanism. The functions within must be fulfilled and the state machine configured in order to provide functionnality to the mechanism.

Definition in file **MechanismTemplate.h**.

4.21.2 Enumeration Type Documentation

4.21.2.1 enum **MechanismParameters**

MechanismParameters define what the parameters of this mechanism.

Definition at line 11 of file MechanismTemplate.h.

4.21.2.2 enum **MechanismStateMachineStates**

MechanismStateMachineStates are the states of the State Machine within the mechanism. The state machine of the mechanims is utilized to command the mechanism from one state to the other. The default states of the state machine are inactive and initialize. Depending on the needs of the mechanism the states should also include following position references and following speed references.

Definition at line 21 of file MechanismTemplate.h.

4.21.2.3 enum **MechanismStates**

MechanismStates define what the states of this mechanims are.

Enumerator:

StateMachine

Definition at line 16 of file MechanismTemplate.h.

4.21.3 Function Documentation

4.21.3.1 enum **MechanismStateMachineStates** loopMechanism (**Mechanism** ∗ *M,* double *ref[MANIPULATORREF]*)

Perform initialization of actuators sensors axes etc...

get reference for the manipulator

Transform the coordinate references to axis references

Command axees

Definition at line 28 of file MechanismTemplate.h.

4.21.4 Variable Documentation

4.21.4.1 inactive

Definition at line 29 of file MechanismTemplate.h.

4.21.4.2 initialize

Definition at line 29 of file MechanismTemplate.h.

4.21.4.3 loop

fill in the other states

Definition at line 29 of file MechanismTemplate.h.

4.22 Framework/Sensor/Examples/ConveyorStation.h File Reference

```
#include "Sensor.h"
```

Functions

- double **readConveyorStation** (**Sensor** ∗s)
- int **setConveyorStationParameter** (**Sensor** ∗s, int parameter, double v)
- int **resetConveyorStationSensor** (**Sensor** ∗s)
- int **removeConveyorStationSensor** (**Sensor** ∗s)

4.22.1 Function Documentation

4.22.1.1 double readConveyorStation (Sensor ∗ *s*)

Definition at line 12 of file ConveyorStation.h.

4.22.1.2 int removeConveyorStationSensor (Sensor ∗ *s*)

Definition at line 29 of file ConveyorStation.h.

4.22.1.3 int resetConveyorStationSensor (Sensor ∗ *s*)

Definition at line 24 of file ConveyorStation.h.

4.22.1.4 int setConveyorStationParameter (Sensor ∗ *s,* int *parameter,* double *v*)

Definition at line 17 of file ConveyorStation.h.

4.23 Framework/Sensor/Sensor.h File Reference

General definition of an sensor containing the data structures.

Classes

- struct **Sensor_**

  Sensor structure contains all the parameters and states of the sesor. The sensor sturcutre also contains the input and output channel numbers for a sensor along with function pointers to functions of the sensor to initialize, read, reset and delete the sensor.

Macros

- #define **SENSMAXIO** 3

    Defines the number of maximum different types input or output IO channel access by a single sensor.
- #define **SENSMAXIN** 3

    Defines the maximum number of inputs all sensors may have.
- #define **SENSMAXOUT** 3

    Defines the maximum number of outputs all sensors may have.
- #define **SENSMAXSTATES** 5

    Define the maximum number of states that all sensors may have.
- #define **SENSMAXPARAMETERS** 4

    Define the maximum number of parameters that all sensors may have.

Typedefs

- typedef struct **Sensor_ Sensor**

Functions

- int **setupSensor** (**Sensor** ∗s, char ∗**name**, char ∗unit, double(∗read)(struct **Sensor_** ∗s), int(∗set-Parameter)(struct **Sensor_** ∗s, int parameter, double v), int(∗reset)(struct **Sensor_** ∗s), int(∗remove)(struct **Sensor_** ∗s))

    setup sensor

Variables

- struct **Sensor_ loopMechanism**

4.23.1 Detailed Description

General definition of an sensor containing the data structures.

Definition in file **Sensor.h**.

4.23.2 Macro Definition Documentation

4.23.2.1 #define SENSMAXIN 3

Defines the maximum number of inputs all sensors may have.

Definition at line 12 of file Sensor.h.

4.23.2.2 #define SENSMAXIO 3

Defines the number of maximum different types input or output IO channel access by a single sensor.

Definition at line 11 of file Sensor.h.

4.23.2.3 #define SENSMAXOUT 3

Defines the maximum number of outputs all sensors may have.

Definition at line 13 of file Sensor.h.

4.23.2.4 #define SENSMAXPARAMETERS 4

Define the maximum number of parameters that all sensors may have.

Definition at line 15 of file Sensor.h.

4.23.2.5 #define SENSMAXSTATES 5

Define the maximum number of states that all sensors may have.

Definition at line 14 of file Sensor.h.

4.23.3 Typedef Documentation

4.23.3.1 typedef struct Sensor_ Sensor

Definition at line 45 of file Sensor.h.

4.23.4 Function Documentation

4.23.4.1 int setupSensor (Sensor ∗ *s*, char ∗ *name*, char ∗ *unit*, double(∗)(struct Sensor_ ∗s) *read*, int(∗)(struct Sensor_ ∗s, int parameter, double v) *setParameter*, int(∗)(struct Sensor_ ∗s) *reset*, int(∗)(struct Sensor_ ∗s) *remove*)

setup sensor

Function to setup the sensor This function is used to setup the fields of the sensor structure, such as assigning the functions to it.

Parameters

s	pointer to the sensor structure
name	contains name of the sensor as a string
units	contains the units of the sensor as a string
read	function pointer to the function to read the values of the sensor
setParameter	function pointer to the function to set the parameters of the sensor
reset	function pointer to the function to reset the sensor
remove	function pointer to the function to remove or delete the sensor

Returns

0 for success, -1 for error

Definition at line 59 of file Sensor.h.

4.23.5 Variable Documentation

4.23.5.1 struct Sensor_ loopMechanism

4.24 Framework/Sensor/SensorTemplate.h File Reference

```
#include "Sensor.h"
```

Functions

- double **readSensor** (**Sensor** ∗s)
- int **setSensorParameter** (**Sensor** ∗s, int parameter, double v)
- int **resetSensor** (**Sensor** ∗s)
- int **removeSensor** (**Sensor** ∗s)

4.24.1 Function Documentation

4.24.1.1 double readSensor (Sensor ∗ *s*)

Definition at line 12 of file SensorTemplate.h.

4.24.1.2 int removeSensor (Sensor ∗ *s*)

Definition at line 25 of file SensorTemplate.h.

4.24.1.3 int resetSensor (Sensor ∗ *s*)

Definition at line 21 of file SensorTemplate.h.

4.24.1.4 int setSensorParameter (Sensor ∗ *s*, int *parameter*, double *v*)

Definition at line 17 of file SensorTemplate.h.

4.25 Framework/Wrapper/Examples/HWWrapper.h File Reference

THis file is a dummy for test purposes and only printsout Header File For the Hardware Wrappers of motion control projects This file is to be used as a template for the development of various projects that involve hardware Objective is to create a standardized hardware functions that have the same name in all projects.

```
#include <stdio.h>
```

Functions

- int **initHardware** ()

  Initialize the hardware that will interface for the wrapper.
- int **deleteHardware** ()

  Celanup and free the resoruces allocated for the hardware.
- int **initEnocder** (int channel)

  Initializes the specified encoder channel.
- int **setEncoder** (int channel, int encvalue)

  Sets a value to the specified encoder channel. Used for homing the encoder channel.
- int **getEncoder** (int channel)

  Gets the pulse number in the encoder hardware.
- int **initEncoderMetricReader** (int channel)

  Initializes the Encoder Metric Reader. Encoder Metric Reader reads the encoder and returns a value that is converted to the desired metric, such as angles, radians, millimeters or micrometers etc.
- int **setEncoderMetricReader** (int channel, float enc2metric)

  Sets the conversion factor for the Encoder Metric Reader.

- float **getEncoderInMetric** (int channel)

  Gets the metric that has been converted from the encoder.
- int **initVelocityReader** (int channel)

  Initializes the Velocity Reading fucntion. Calculates or estimates the velocity using hardware.
- float **getVelocity** (int channel)

  Gets the Velocity that has been calculate by the hardware.
- int **initAnalogOutput** (int channel)

  Initializes the Analog output circuit.
- int **setAnalogOutput** (int channel, float Volts)

  Sets the value for analog output.
- int **initAnalogInput** (int channel)

  Initializes the Analog Input circuits.
- float **getAnalogInput** (int channel)

  Gets the Analog input.
- int **initDigitalInput** (int channel)

  Initializes the Digital Input Circuits.
- int **getDigitalInput** (int channel)

  Gets the Digitial input.
- int **initDigitalOutput** (int channel)

  Initializes the Digital Output circuit.
- int **setDigitalOutput** (int channel, int value)

  Sets the Digital Output.
- int **initPWM** (int channel)

  Initializes the PWM circuit.
- int **setPWM** (int channel, int value)

  Sets the PWM output value.

4.25.1 Detailed Description

THis file is a dummy for test purposes and only printsout Header File For the Hardware Wrappers of motion control projects This file is to be used as a template for the development of various projects that involve hardware Objective is to create a standardized hardware functions that have the same name in all projects. Created by Teoman Naskali on 14/10/10. Copyright Sabanci University 2010. All rights reserved.

Definition in file **HWWrapper.h**.

4.25.2 Function Documentation

4.25.2.1 int deleteHardware ( )

Celanup and free the resoruces allocated for the hardware.

Delete Hardware

Returns

0 for success, -1 for error.

Definition at line 28 of file HWWrapper.h.

4.25.2.2 float getAnalogInput (int *channel*)

Gets the Analog input.

Get Analog Input

Parameters

channel	channel number

Returns

Analog input in volts

Definition at line 125 of file HWWrapper.h.

4.25.2.3 int getDigitalInput (int *channel*)

Gets the Digitial input.

Get Digital Input

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 137 of file HWWrapper.h.

4.25.2.4 int getEncoder (int *channel*)

Gets the pulse number in the encoder hardware.

Get Encoder Value

Parameters

channel	is the channel number to be read

Returns

encoder pulses

Definition at line 62 of file HWWrapper.h.

4.25.2.5 float getEncoderInMetric (int *channel*)

Gets the metric that has been converted from the encoder.

Get Encoder in Metric

Parameters

channel	number

Returns

metric derived from pulses

Definition at line 85 of file HWWrapper.h.

4.25.2.6 float getVelocity (int *channel*)

Gets the Velocity that has been calculate by the hardware.

Get Velocity

Parameters

channel	channel number

Returns

velocity

Definition at line 98 of file HWWrapper.h.

4.25.2.7 int initAnalogInput (int *channel*)

Initializes the Analog Input circuits.

Initialize Analog Inputs

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 119 of file HWWrapper.h.

4.25.2.8 int initAnalogOutput (int *channel*)

Initializes the Analog output circuit.

Initialize Analog Output

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 104 of file HWWrapper.h.

4.25.2.9 int initDigitalInput (int *channel*)

Initializes the Digital Input Circuits.

Initialize Digital Input

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 131 of file HWWrapper.h.

4.25.2.10 int initDigitalOutput (int *channel*)

Initializes the Digital Output circuit.

Initialize Digital Output

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 143 of file HWWrapper.h.

4.25.2.11 int initEncoderMetricReader (int *channel*)

Initializes the Encoder Metric Reader. Encoder Metric Reader reads the encoder and returns a value that is converted to the desired metric, such as angles, radians, millimeters or micrometers etc.

Initialize Encoder Metric Reader

Parameters

channel	is the encoder channel number

Returns

encoder pulses

Definition at line 72 of file HWWrapper.h.

4.25.2.12 int initEnocder (int *channel*)

Initializes the specified encoder channel.

Initialize Encoder

Parameters

channel	the encoder channel number

Returns

0 for success, -1 for error

Definition at line 39 of file HWWrapper.h.

4.25.2.13 int initHardware ( )

Initialize the hardware that will interface for the wrapper.

Initialize Hardware

Returns

0 for success, -1 for error.

Definition at line 20 of file HWWrapper.h.

4.25.2.14 int initPWM (int *channel*)

Initializes the PWM circuit.

Initialize PWM

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 164 of file HWWrapper.h.

4.25.2.15 int initVelocityReader (int *channel*)

Initializes the Velocity Reading fucntion. Calculates or estimates the velocity using hardware.

Initialize Velocity Reader

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 92 of file HWWrapper.h.

4.25.2.16 int setAnalogOutput (int *channel,* float *Volts*)

Sets the value for analog output.

Set Analog Output

Parameters

Volts	output value in volts

Returns

0 for success, -1 for error

Definition at line 110 of file HWWrapper.h.

4.25.2.17 int setDigitalOutput (int *channel,* int *value*)

Sets the Digital Output.

Set Digital Output

Parameters

channel	channel number
value	output value in boolean

Returns

0 for success, -1 for error

Definition at line 150 of file HWWrapper.h.

4.25.2.18 int setEncoder (int *channel,* int *encvalue*)

Sets a value to the specified encoder channel. Used for homing the encoder channel.

Set the Encoder value

Parameters

channel	the encoder channel number
encvalue	the encoder value to be set to the encoder

Returns

0 for success, -1 for error

Definition at line 51 of file HWWrapper.h.

4.25.2.19 int setEncoderMetricReader (int *channel,* float *enc2metric*)

Sets the conversion factor for the Encoder Metric Reader.

Set Enocder Metric Reader

Parameters

channel	channel number
enc2metric	the conversion factor from encoder pulses to the metric

Returns

0 for success, -1 for error

Definition at line 79 of file HWWrapper.h.

4.25.2.20 int setPWM (int *channel,* int *value*)

Sets the PWM output value.

Set PWM

Parameters

value,specify	value

Returns

0 for success, -1 for error

Definition at line 170 of file HWWrapper.h.

4.26 Framework/Wrapper/HWWrapperTemplate.h File Reference

Functions

- int **initHardware** ()

  Initialize the hardware that will interface for the wrapper.
- int **deleteHardware** ()

  Celanup and free the resoruces allocated for the hardware.
- int **initEnocder** (int channel)

  Initializes the specified encoder channel.
- int **setEncoder** (int channel, int encvalue)

  Sets a value to the specified encoder channel. Used for homing the encoder channel.
- int **getEncoder** (int channel)

  Gets the pulse number in the encoder hardware.
- int **initEncoderMetricReader** (int channel)

  Initializes the Encoder Metric Reader. Encoder Metric Reader reads the encoder and returns a value that is converted to the desired metric, such as angles, radians, millimeters or micrometers etc.
- int **setEncoderMetricReader** (int channel, float enc2metric)

  Sets the conversion factor for the Encoder Metric Reader.
- float **getEncoderInMetric** (int channel)

  Gets the metric that has been converted from the encoder.
- int **initVelocityReader** (int channel)

  Initializes the Velocity Reading fucntion. Calculates or estimates the velocity using hardware.
- float **getVelocity** (int channel)

  Gets the Velocity that has been calculate by the hardware.
- int **initAnalogOutput** (int channel)

  Initializes the Analog output circuit.
- int **setAnalogOutput** (int channel, float Volts)

  Sets the value for analog output.
- int **initAnalogInput** (int channel)

  Initializes the Analog Input circuits.
- float **getAnalogInput** (int channel)

  Gets the Analog input.
- int **initDigitalInput** (int channel)

  Initializes the Digital Input Circuits.
- int **getDigitalInput** (int channel)

Gets the Digitial input.
- int **initDigitalOutput** (int channel)

  Initializes the Digital Output circuit.
- int **setDigitalOutput** (int channel, bool value)

  Sets the Digital Output.
- int **initPWM** (int channel)

  Initializes the PWM circuit.
- int **setPWM** (int channel, int value)

  Sets the PWM output value.

4.26.1 Function Documentation

4.26.1.1 int deleteHardware ( )

Celanup and free the resoruces allocated for the hardware.

Delete Hardware

Returns

0 for success, -1 for error.

Definition at line 28 of file HWWrapper.h.

4.26.1.2 float getAnalogInput (int *channel*)

Gets the Analog input.

Get Analog Input

Parameters

channel	channel number

Returns

Analog input in volts

Definition at line 125 of file HWWrapper.h.

4.26.1.3 int getDigitalInput (int *channel*)

Gets the Digitial input.

Get Digital Input

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 137 of file HWWrapper.h.

4.26.1.4 int getEncoder (int *channel*)

Gets the pulse number in the encoder hardware.

Get Encoder Value

Parameters

channel	is the channel number to be read

Returns

0 for success, -1 for error

Get Encoder Value

Parameters

channel	is the channel number to be read

Returns

encoder pulses

Definition at line 62 of file HWWrapper.h.

4.26.1.5 float getEncoderInMetric (int *channel*)

Gets the metric that has been converted from the encoder.

Get Encoder in Metric

Parameters

channel	number

Returns

metric derived from pulses

Definition at line 85 of file HWWrapper.h.

4.26.1.6 float getVelocity (int *channel*)

Gets the Velocity that has been calculate by the hardware.

Get Velocity

Parameters

channel	channel number

Returns

velocity

Definition at line 98 of file HWWrapper.h.

4.26.1.7 int initAnalogInput (int *channel*)

Initializes the Analog Input circuits.

Initialize Analog Inputs

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 119 of file HWWrapper.h.

4.26.1.8 int initAnalogOutput (int *channel*)

Initializes the Analog output circuit.

Initialize Analog Output

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 104 of file HWWrapper.h.

4.26.1.9 int initDigitalInput (int *channel*)

Initializes the Digital Input Circuits.

Initialize Digital Input

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 131 of file HWWrapper.h.

4.26.1.10 int initDigitalOutput (int *channel*)

Initializes the Digital Output circuit.

Initialize Digital Output

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 143 of file HWWrapper.h.

4.26.1.11 int initEncoderMetricReader (int *channel*)

Initializes the Encoder Metric Reader. Encoder Metric Reader reads the encoder and returns a value that is converted to the desired metric, such as angles, radians, millimeters or micrometers etc.

Initialize Encoder Metric Reader

Parameters

channel	is the encoder channel number

Returns

encoder pulses

Definition at line 72 of file HWWrapper.h.

4.26.1.12 int initEnocder (int *channel*)

Initializes the specified encoder channel.

Initialize Encoder

Parameters

channel	the encoder channel number

Returns

0 for success, -1 for error

Definition at line 39 of file HWWrapper.h.

4.26.1.13 int initHardware ( )

Initialize the hardware that will interface for the wrapper.

Initialize Hardware

Returns

0 for success, -1 for error.

Definition at line 20 of file HWWrapper.h.

4.26.1.14 int initPWM (int *channel*)

Initializes the PWM circuit.

Initialize PWM

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 164 of file HWWrapper.h.

4.26.1.15 int initVelocityReader (int *channel*)

Initializes the Velocity Reading fucntion. Calculates or estimates the velocity using hardware.

Initialize Velocity Reader

Parameters

channel	channel number

Returns

0 for success, -1 for error

Definition at line 92 of file HWWrapper.h.

4.26.1.16 int setAnalogOutput (int *channel,* float *Volts*)

Sets the value for analog output.

Set Analog Output

Parameters

Volts	output value in volts

Returns

0 for success, -1 for error

Definition at line 110 of file HWWrapper.h.

4.26.1.17 int setDigitalOutput (int *channel,* bool *value*)

Sets the Digital Output.

Set Digital Output

Parameters

channel	channel number
value	output value in boolean

Returns

0 for success, -1 for error

Definition at line 122 of file HWWrapperTemplate.h.

4.26.1.18 int setEncoder (int *channel,* int *encvalue*)

Sets a value to the specified encoder channel. Used for homing the encoder channel.

Set the Encoder value

Parameters

channel	the encoder channel number
encvalue	the encoder value to be set to the encoder

Returns

0 for success, -1 for error

Definition at line 51 of file HWWrapper.h.

4.26.1.19 int setEncoderMetricReader (int *channel,* float *enc2metric*)

Sets the conversion factor for the Encoder Metric Reader.

Set Enocder Metric Reader

Parameters

channel	channel number
enc2metric	the conversion factor from encoder pulses to the metric

Returns

0 for success, -1 for error

Definition at line 79 of file HWWrapper.h.

4.26.1.20 int setPWM (int *channel,* int *value*)

Sets the PWM output value.

Set PWM

Parameters

value,specify	value

Returns

0 for success, -1 for error

Definition at line 170 of file HWWrapper.h.