INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

ESPECIALIDAD EN SISTEMAS EMBEBIDOS

ARQUITECTURA ESCALABLE DE DESARROLLO EN LINUX PARA PLATAFORMA EMBEBIDA DE ROBÓTICA EN TIEMPO REAL

Trabajo final que para obtener el diploma de

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: Omar Jorge Avelar Suárez

Asesor: Dr. Luis Enrique González Jiménez

Tlaquepaque, Jalisco, marzo de 2017.

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Departamento de Electrónica, Sistemas e Informática

ESPECIALIDAD EN SISTEMAS EMBEBIDOS

A SCALABLE EMBEDDED ROBOTICS REAL TIME PLATFORM DEVELOPMENT ARCHITECTURE IN LINUX

Final report to earn the diploma of

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presented by: Omar Jorge Avelar Suárez

Advisor: Dr. Luis Enrique González Jiménez

Tlaquepaque, Jalisco, marzo de 2017.

ACKNOWLEDGEMENTS

I would like to thank my family members Claudia, Jorge, Elizabeth, Cinthia, Marcela, Ricardo, Sofia & André who has supported me throughout my academic journey since very young and their encouragement about engineering, science and mathematics.

I also thank my supervisor Dr. Luis Enrique González Jiménez for introducing me to the world of robotics.

And I also give my gratitude to the Consejo Nacional de Ciencia y Tecnología (CONACYT) for its financial support throughout the specialization program.

RESUMEN

La manipulación de objetos por medio de robots es elemento crucial de las herramientas avanzadas de automatización. Sin embargo, los mecanismos para controlarlos típicamente son muy específicos y requieren diseños que están profundamente atados al hardware del robot – este tipo de implementaciones resultan en código no re-utilizable y optimizaciones de algoritmos que solo funcionan en familias de robots particulares. Aquí presentaremos una propuesta de arquitectura de software para brazos robóticos que corren en el entorno ya ampliamente utilizado de GNU/Linux y hablaremos de sus beneficios y desventajas de dicha implementación.

El trabajo aquí habla de la necesidad de una arquitectura de software que sea fácil de implementar y escalable en cuanto a su utilización de recursos para prototipos de robots y sistemas completos funcionales. Aquí vamos a hablar de diferentes configuraciones y conceptos relacionados a la manipulación y el control de sistemas robóticos. Una configuración de robot ejemplo es propuesta y se utiliza como caso de estudio para mostrar las dificultades y ventajas de dicha implementación, así como sus parámetros de desempeño en cuanto a tiempos de respuesta y aplicaciones.

SUMMARY (ENGLISH)

Robotic manipulation is crucial element of advanced automation tools, however the methods for controlling it are usually crafted for specific and custom designs that are deeply tied to the hardware of the robotics. These type of implementations results in non-re-usable code and optimization algorithms that only work for specific robotic families. In here we will discuss a software architecture for robotic arms running under the freely and widely available GNU/Linux environment along with its benefits and drawbacks of such.

The work here expresses the need for a software architecture that results in an easy to implement and scalable framework for robotics prototyping and real functioning systems. In here we will be discussing different robotic configurations and the concepts associated with manipulating and controlling robotic systems. A robot configuration is used as a case of study where the challenges and benefits of the implementation are discussed along with performance data and applications developed with the framework.

CONTENTS

CHAPTER 1: INTRODUCTION 9

1.1 INTRODUCTION 9
1.2 FUNDAMENTALS OF ROBOTICS 9
1.2.1 Degrees of Freedom 10
1.2.2 Robot Kinematics 12
1.3 ROBOT HARDWARE COMPONENTS 13
1.4 TECHNOLOGIES COMING TOGETHER 14

CHAPTER 2: SOFTWARE ARCHITECTURE 15

2.1 OUR ROBOT DESCRIBED 15
2.2 SOFTWARE ARCHITECTURE 17
2.2.1 User Space vs Kernel Space 17
2.2.2 Configurability 18
2.2.3 Modules 19
2.2.3.1 Robotic Arm Class 20
2.2.3.2 Robotic Joint Class 20
2.2.3.3 Position Class 21
2.2.3.4 Movement Class 22
2.2.3.5 Custom Defined Drivers 23
2.2.3.6 User Space Drivers 23

CHAPTER 3: EXPERIMENTAL RESULTS 25

3.1 CLOSED LOOP REAL TIME CONTROL 25
3.1.1 Architecture & Implementation 26
3.1.2 Real Time Scheduler 27
3.1.3 Scheduler Configuration 28
3.1.4 Multithread Characteristics 28
3.1.5 Results & Analysis 29

CHAPTER 4: CONCLUSIONS 33

4.1 CONCLUSIONS 33
4.2 FUTURE WORK 34

REFERENCES 36

LIST OF FIGURES

Figure 1. An articulated robot consisting of two joints. 10
Figure 2. 6 DOF Model of a robot arm courtesy of TurboSquid [4]. 11
Figure 3. Schematic representation of forward and inverse kinematics. 12
Figure 4. Picture of the 2 DOF robotic arm system built.............................15
Figure 5. Component view of the controller..16
Figure 6. User space vs kernel space. 17
Figure 7. Layered software architecture. 19
Figure 8. Arm and joint container representation visualized. 21
Figure 9. Position & Movement user space custom driver class examples. .24
Figure 10. High-level sequence of events when the API changes position. .25
Figure 11. Automatic control with reference angles as inputs.....................26
Figure 12. Automatic control per joint. 27
Figure 13. Recorder tool waiting for input to stop. 31
Figure 14. Recorded trajectory being replayed..32
Figure 15. Playback tool console output..32

LIST OF TABLES

Table 1. Movement parameters of a six degrees of freedom body. 11
Table 2. Performance characteristics scaling vs. scheduler configurations in the arm. 30

CHAPTER 1: INTRODUCTION

1.1 Introduction

We have seen the rise of robots and automation since the introduction of the term in 1921 [1]. Robot manipulation is a core technology that allows robotics to make use of motion and position of robots to perform useful work in our world.

The goal of robotics has been for a while to automate and perform repetitive tasks, or complex actions automatically. Currently there is a direct application to the industry on assemblies and factories – we can see robot manipulators being used to spray paint cars, assemble complex microchip products and even do automated performance testing on smartphones [2].

1.2 Fundamentals of Robotics

Robots can be classified in various ways, depending on the components, configuration/topology, and use.

During this document we will be focusing primarily on articulated robots, or also referred to jointed arm. These robots are defined as anthropomorphic because their movement and operation resembles that of the human forearm and upper arm. A robot manipulator is a specific category of robot, they are created from a sequence of link and joint combinations. A link is a rigid member that connects two joints or axes. In analogy to the human body an example of a joint would be our shoulder, or our elbows, whereas a link would be the forearm in our arm or calf in our legs.

The axes are the movable component of a robotic manipulator that cause relative motion between adjoining links.

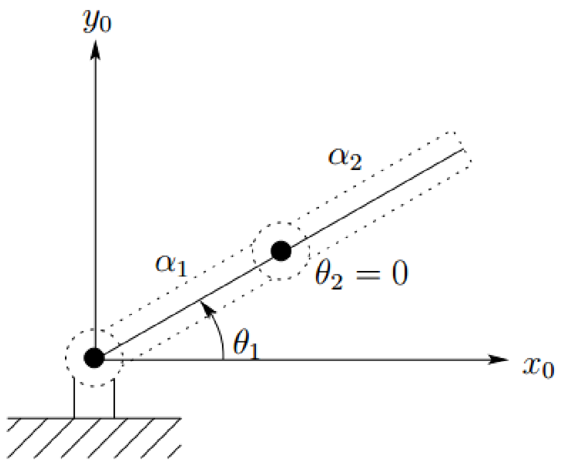


Figure 1. An articulated robot consisting of two joints.

By analyzing Figure 1 closely we can see there are important elements on robotics such as the angles that each joint is at. During next sections we will see how we can figure out the position of the robot by knowing both angles and the link length of the constituents. It is important to understand these concepts as they will later will be re-used as basic concepts of the software architecture.

1.2.1 Degrees of Freedom

This is a widely used term to describe a robot’s freedom of motion in a three dimensional space. It can be seen as the ability to move forward, backward, up, down, left and right.

For each degree of freedom a joint is required. It is proven that a robot requires six degrees of freedom to be completely versatile this means a body can move on the X, Y, and Z axes as well as change orientation between those axes, considering a dextrorotary coordinate system. The orientation is defined through three rotations: pitch, yaw, and roll as seen in Table 1.

	Translation			**Rotation**		
Movement	Along X	Along Y	Along Z	Between X and Y	Between X and Z	Between Z and Y
Also Called	Heaving	Surging	Swaying	Pitch	Yaw	Roll

Table 1. Movement parameters of a six degrees of freedom body.

However, it is still important to note that a robot with just six degrees of freedom is in comparison still clumsier than a human hand with 22 to 27 degrees of freedom [3].

Figure 2. 6 DOF Model of a robot arm courtesy of Zortrax [4].

1.2.2 Robot Kinematics

Kinematics studies the motion of bodies without consideration of the forces or moments that cause the motion. Robot kinematics refers to the analytical study of the motion of a robot manipulator.

There are two important spaces used in kinematics, the Cartesian space and the configuration space.

Robot kinematics is what will be mainly discussed here and is used for modeling the analytical solutions to our robot. **Forward kinematics** refers to the use of the kinematic equations of a robot to compute the position of the end-effector from specified values for the joint parameters. For example we could figure out the Cartesian point in space of our fist by understanding the angle at which our elbow and shoulder are at.

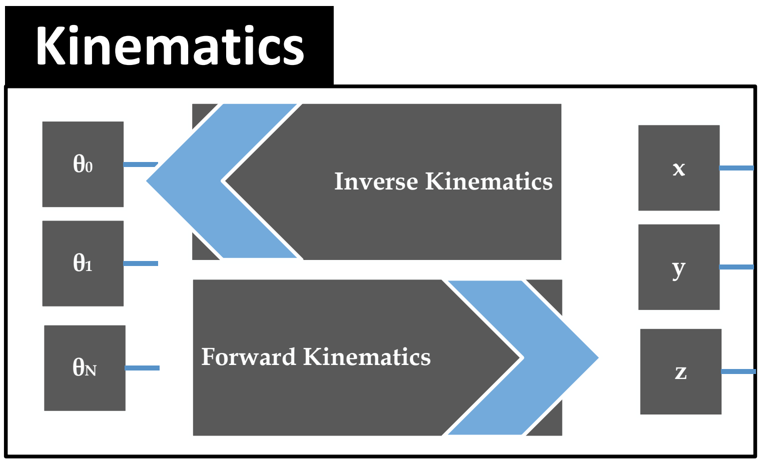


Figure 3. Schematic representation of forward and inverse kinematics.

Inverse kinematics, as the name implies, is going the opposite way, as we can see in Figure 3. This will output the angular information from the joint space based on Cartesian coordinates of the end effector as input. This is a very important principle as this will serve as the basis of positioning a robot arm end-effector in a three-dimensional Cartesian space by starting to move and change its joint configuration angles.

It is important to know that inverse kinematics is much more difficult as there can be many solutions on different DOF systems, because for complex configurations there can exist different joint configurations that achieve the same end-effector Cartesian position. There are two main solution techniques for the inverse kinematics problem, analytical and numerical methods. For the analytical technique one must know the robot configuration and have data in order to obtain such solution. Ultimately, the method to which we implement for performing the kinematic transformations can have huge impact on real time computational requirements and limitations.

1.3 Robot Hardware Components

People that are newly entering the robotics domain are presented with a myriad of possibilities of what tools they want to make use for designing a robotic prototype or test a new robotic motion planning algorithm, etc. There are various ways of categorizing the robotic system but when you are prototyping and implementing a robot manipulator it usually comes down into two big factors that impact choosing the hardware components:

1. **The controller** being usually a computing and programmable module which will be handling the actions for the robot to be performed and which corresponds to the brain and smarts of our decisions.
2. **The electro-mechanical properties of the robot**, being the manipulator and effector which in turn is formed by actuators and sensors which will define the robot configuration or topology as well as the information of the sensors that will be used as inputs and feedback to the complete system.

For the controller developers and scientists can be using vast different hardware that ranges from small microcontrollers, to FPGA’s, to commercially available Personal Computers, Programmable Logic Controllers (PLCs), etc. It is important to note that effectively changing the underlying hardware usually means re-writing programs as these different hardware systems target unique requirements and architectures.

Second, the electro-mechanical properties of the robot usually dictate the operation and constraints of it, different robot configurations usually come with different software modeling methods and different sources to obtain inputs for the feedback, one example is that someone could use a camera for obtaining the position of the robot or instead opt for cheaper alternatives such as rotary encoders.

1.4 Technologies Coming Together

The GNU/Linux Operating System has grown with deep support since its introduction in 1991. It has provided the basis of a now widely used operating system that is freely to be modified to suit various necessities.

Having a full OS for a robotic arm manipulator allows various services of the OS to be used and leveraged and hence decreasing the development time of a robotic manipulator prototyping or development [5].

There are a few things that are already provided by the GNU/Linux OS which are of tremendous benefit for robotic applications, examples such as:

- **Task scheduler** with priority, preemption, and slice configurability allowing a true multi-thread and multi-task system to exist on different target platforms be it multi-core or single-core.
- **Memory management** for dynamic memory allocation and scalability for different applications to be running as resources permit, such as different processes monitoring temperature, or gathering data metrics.
- **Tasks and threads** creation support and infrastructure in place that allow a robust operation together with the task scheduler.
- **Networking** support and driver stack for easy internet integration and security.
- **Remote administration** services such as SSH or Telnet for maintenance, analysis and remote execution.
- Broad set of **computing architectures support** such as x86, AMD64, ARM, Atmel AVR32, MIPS, OpenRISC, Power and various others.

All of these services come into play and are important when building a more robust application on top of a robotic manipulator or appliance.

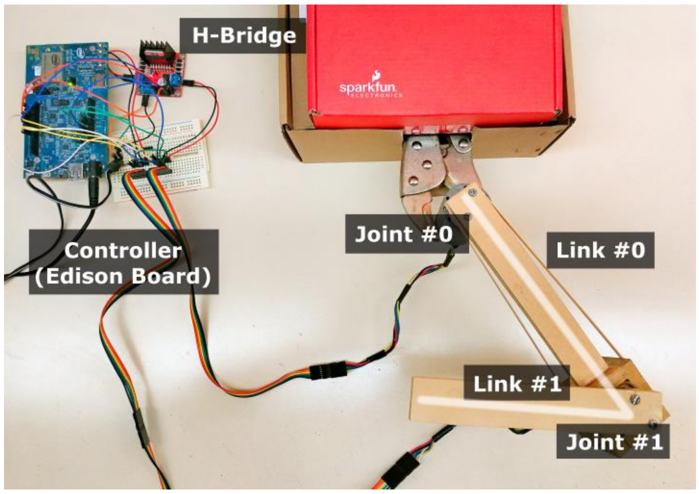
Another huge strength of using GNU/Linux is that it can be tailored and streamlined to use less or more resources depending on the target platform to be using for your robotics application, you can remove a lot of elements that are not needed for your application instead of adding them and building them from scratch.

CHAPTER 2: SOFTWARE ARCHITECTURE

2.1 Our Robot Described

We would like to begin by showing you clearly the robot that will be discussed during the testing and implementation of the framework. This will let you as the reader to understand and grasp all of the details of the implementation.

We have chosen to build an articulated robot arm, which will contain two links and two joints as seen in Figure 4. This, in turn, will give us two degrees of freedom in a two-dimensional field.



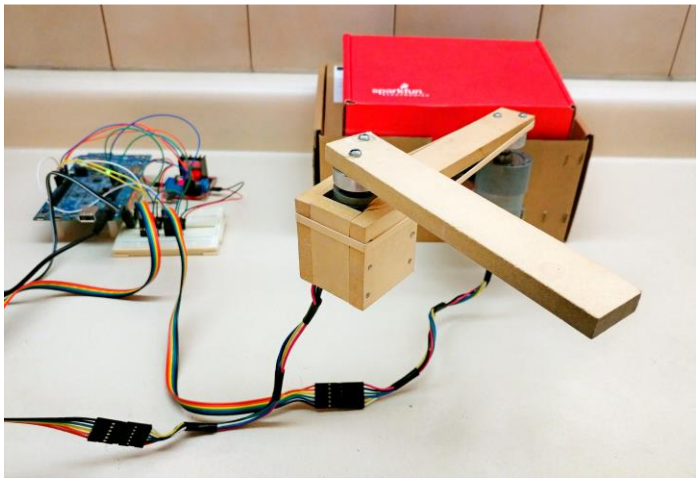


Figure 4. Picture of the 2 DOF robotic arm system built.

The controller of the robotic manipulator that will be implemented consists of the following hardware components and requirements:

- An Intel Edison Embedded System Kit running Debian Linux distribution [6].
- Two quadrature encoders hooked up to a DC motor.
- Four GPIO inputs that support level interrupt generation.
- Four PWM channels.
- An H-Bridge IC kit for the power and drive of the DC motors.

Figure 5 shows how the mentioned list is used and interconnected.

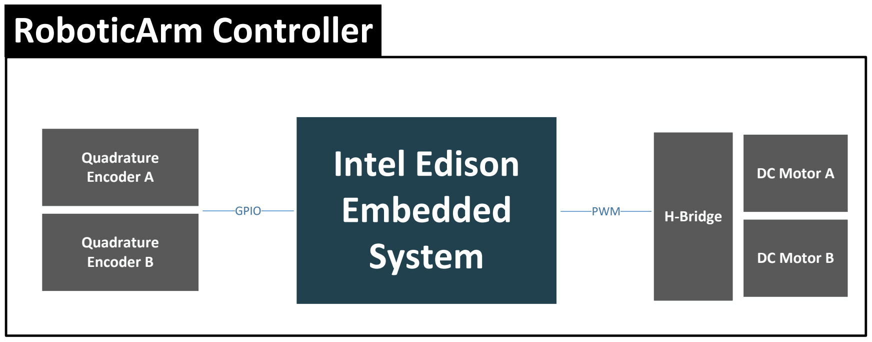


Figure 5. Component view of the controller.

Let’s take a look as to why the components are put there. There are two quadrature encoders & two DC motors, named A and B respectively.

First the DC motors will provide a way for us to rotate the links of our robot. By hooking up our pulse-width modulation outputs to a four-input H-Bridge we can specifically rotate each of the DC motors independently to the right, or left.

And second the robot sensors. We have chosen to use quadrature encoders as a cheap way to tell us the position of a DC motor relative to where we calibrate them. They provide pulses as they rotate which in turn are captured to our Intel Edison Embedded System via GPIO. Since each quadrature encoder requires two inputs, usually referred as Channel A and Channel B, a total of 4 GPIO pins are needed to hook our two quadrature encoders. Quadrature encoders permit us to know if the rotation is going clock-wise (CW) or counter-clockwise (CCW) as well as how much it has moved depending on its pulses per revolution (PPR).

With both the motors and the sensors in place we can now exert movement and know our relative angle of the motor as exemplified next.

2.2 Software Architecture

There are different elements that are required in order to have a fully functioning robot. These different ingredients will play a role in our complete software architecture intended to be running on GNU/Linux Operating System.

2.2.1 User Space vs Kernel Space

The first thing we can think when talking about the software architecture of the framework is which portions will be user space and where are our kernel space dependencies.

While all applications rely on the underlying kernel, the kernel provides an API to user space applications via system calls as seen in Figure 6. While the kernel is the only layer of abstraction between programs and the resources they require access to [7].

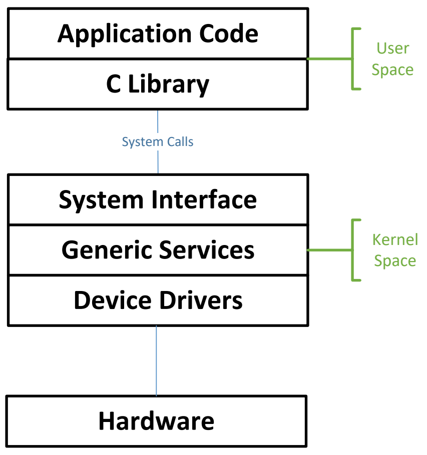


Figure 6. User space vs kernel space.

Primarily, user space was chosen since, by definition, the role of a module is to extend the Kernel functionality. Some examples of Kernel modules are USB drivers, Ethernet modules, PWM drivers, Thermal Sensors, etc.

One may falsely think that, in order to make use of a device such as a PWM module in a hardware board to control DC motors that will in turn end up performing the translation movement of our robot, we would require to be doing calls within the kernel to the drivers directly. However, this is where we will be making use of device drivers exposed to user space [8]. So as a requirement one will need to have a way or method of accessing a driver. In the actual implementation, you will find out quickly that we make use of the PWM API and GPIO API exposed as a sysfs interface [9] [10].

Another immediate benefit of using user space Linux is that this allows us to separate a low-level driver and permits robotic applications to be run in the system without the need of having to recompile the Linux kernel.

2.2.2 Configurability

The implementation of the framework provides configuration that can be modified in the **RoboticArm_Config.h** [11] file. This file contains information of the underlying hardware where you will be implementing the robot in.

This file needs to be tailored and adjusted when creating your own robot executables. It contains information such as the GPIO pins to be used, the link length in meters, and the PWM channels and can be easily extended to include USB ports to be used, or camera Linux devices.

Here is an example of how it looks like to give an idea of what things might need to be adjusted for different applications:

```
/* The physical length of each of the links in meters */

static constexpr double link_lengths[] = { 0.012, 0.012 };


/* Pair of pins used for these elements */

static constexpr int quad_encoder_pins[][2]  = {{ 49,  48}, { 41,  43}};

static constexpr int dc_motor_pins[][2]      = {{  0,   1}, {  2,   3}};


/* Calculate number of joints based of motors */

static constexpr int joints_nr =
sizeof(dc_motor_pins)/sizeof(dc_motor_pins[0]);
```

As mentioned in page 15, this corresponds to a robot that is going to have two joints. Using 2 channels of PWM per joint to control the DC motors and 2 channels for the quadrature encoder pulses.

2.2.3 Modules

Separating between different modules layers and showcasing how they connect between them is usually the best way of understanding the full system. Figure 7 depicts the representation of the full software architecture components in a hierarchical manner.

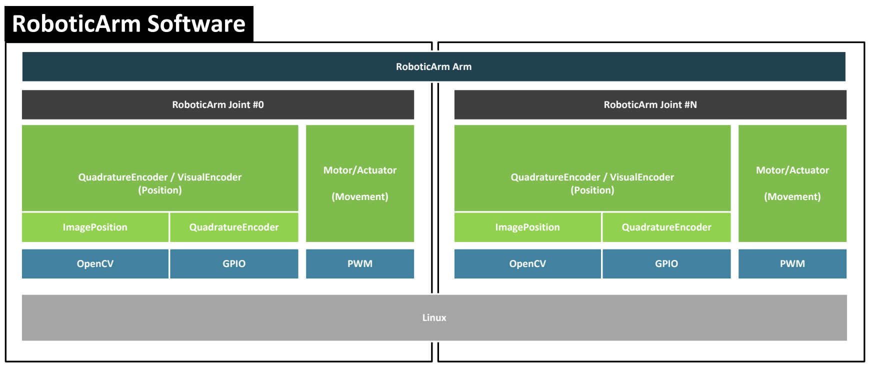


Figure 7. Layered software architecture.

The layers will be discussed into more detail later, but it is important to state with a few words why each of the components exists in the first place by looking at the following table.

Layer Type	Components	Description
Application	Arm	Consists of a collection of joints. The unison of joints and links create a working articulated robot.
	Joint	An independent joint of an articulated robot, it is able to know its position and control their own movement.
User Space Custom Defined Drivers	Position Movement QuadratureEncoder Motor VisualEncoder	Joints from the top layer are able to figure out position and movement by gluing these two layers together.
User Space Drivers	SysFS GPIO SysFS PWM	By allowing to create these custom defined drivers we can

	OpenCV	create simpler objects such as motors, servos, or rotary encoders. Objects that simply do not exist in the Linux realm at user space.
Operating System Services	Linux	Provides a safe and robust environment to be running POSIX applications as well as resource accesses.

2.2.3.1 Robotic Arm Class

A robotic arm is comprised of joints and links as described in chapter 1. A robotic arm is a C++ class that contains various joints, it can be seen as a container that makes use of the various joints.

The robotic arm is responsible of being able to set and get a three-dimensional coordinate of the actuator. A robotic arm object is what will be instated and used when creating a robot, and as such is the object that has robot functions.

The public API exposed to the programmer is as simple as:

Function	**Description**
void **Init**(void);	Executes the initialization routines for each of the N-joints that conform the robot.
void **GetPosition**(Point &pos);	Returns by reference a Point (x,z,y) Cartesian coordinate system of the end-effector.
void **SetPosition**(const Point &pos);	Sets the end-effector of the robot to the desired (x,y,z) Cartesian coordinate system.

An example of the above API is seen in the file **Robot_Keyboard.cpp** [11] where a robot's coordinates are moved by using a console Linux application with the up, down, left and right keys in a keyboard.

2.2.3.2 Robotic Joint Class

The next level down comes as joints, joints in the implementation

Function	**Description**
void **Init**(void);	Starts its movement component of it, be it servos, DC motors, etc. This initiates the joint to be in the home position and in turn start the control based feedback-loops that run on separate threads so that the joint is aware of its angular position every time and correct itself.

	Home position for the joint means that pos.x = 0, pos.y = 0 & pos.z = 0.
double **GetAngle**(void);	Returns the actual theta degrees as a floating point number that the joint is positioned at.
void **SetAngle**(const double &theta);	Sets the joint to be at the specified theta degrees as a floating point number.
void **SetZero**(void);	Used to define our new reference angle of 0 for the control based feedback-loop and the sensors underneath to be thinking that they are positioned at angle of 0 for calibration purposes and proper initialization.

One can clearly see that the term arm and joint is closely connected and Figure 8 is a representation from of the classes which have a particular resemblance to containers.

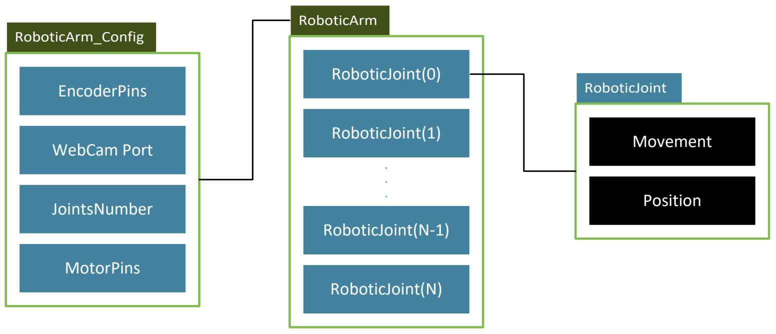


Figure 8. Arm and joint container representation visualized.

And hence the **RoboticArm** object exists because of a collection of **RoboticJoint** objects that each uniquely have movement control and position information from its sensors.

2.2.3.3 Position Class

This class must be generic so that different sensors can be connected underneath to understand where the joint is being positioned. Examples of sensors can be quadrature encoders, or a visual camera, magnetometers, etc.

Function	**Description**

double **GetAngle**(void);	Returns the actual theta degrees as a floating-point number that the underlying sensor is reading.
void **SetAngle**(const double &theta);	Sets the joint to be at the specified theta degrees as a floating-point number.
void **SetZero**(void);	Used to tell our underlying sensors that its internal states need to be reset and start measuring from relative to here.

2.2.3.4 *Movement Class*

The following implementation proved to be simple and yet generic such that we could glue different actuators underneath such as servos, motors, coils, etc. Its main functionality is to provide a joint with a method of getting itself to move.

Function	**Description**
void **Stop**(void);	Abruptly stops the actuator from driving. Used when shutting down the system if an error occurred or when exiting.
void **Start**(void);	To be used initially by the joint, this starts up the actuators.
double **GetSpeed**(void);	Used to control the desired speed that you want the actuator to have, it is a positive numerical floating-point value that ranges from 0.0 to 100.0.
void **SetSpeed**(const double &pcnt);	
Direction **GetDirection**(void);	Used to indicate if the actuator is going CW or CCW.
void **SetDirection**(const Direction &dir);	
State **GetState**(void);	Indicator if the actuator is in a stopped or running state in order to be used for decisions at the joint level for control.
void **ApplyRangeLimits**(const double &pcnt_low, const double &pcnt_high);	This is a calibration function to be used higher in the hierarchy in the joint level. The joint has routines that can set the minimum value that provides real movement in the joint (pcnt_low) and can set the maximum speed (pcnt_high) and sets them as limits. After this is set, the new 0.0 will behave as zero-real movement and just increasing to next values

	of 0.01 for example will produce real movement at the joint level. This in turn provides finer calculations at the controller level for the robot.

2.2.3.5 Custom Defined Drivers

As part of the user space custom defined drivers we have created the following drivers that satisfy gluing together the Position class to sensors, and respectively the Movement classes to an actuator. These drivers export an API in C++ to be able to talk to PWM and GPIO modules.

a) **HighLatencyGPIO**

This is a C++ class which abstracts the Linux sysfs interface to GPIO’s. It was developed on and intended for use on the BeagleBone Black (BBB) [12]. This provides interrupt registration and configuration aspects provided by the sysfs interface. This serves as the bridge between the quadrature encoder hardware and C++.

b) **HighLatencyPWM**

Based on the above, this abstracts the Linux sysfs interface but for the PWM sysfs interface. This will serve as part of the glue-logic for the DC-Motor control and exposes the PWM control pins to C++.

2.2.3.6 User Space Drivers

For the current implementation, no driver had to be developed as both PWM and GPIO drivers were exposed as user-space in the Linux environment. It is worth noting that these must exist in order to make proper connection to our top layer which are custom defined drivers.

Two examples were implemented in this framework, a DC Motor Class and a Quadrature Encoder Class, the intention of this modules is as follows.

a) **Linux-DC-Motor**

This user space module uses Kernel user-space PWM controls in order to have a working infrastructure for DC motors. Using this class for H-Bridge PWM controlled DC motors.

b) **Linux-Quadrature-Encoder**

This user space module uses Kernel user-space GPIO interrupts in order to have a working infrastructure for quadrature encoders.

The connection of the classes is exemplified in Figure 9.

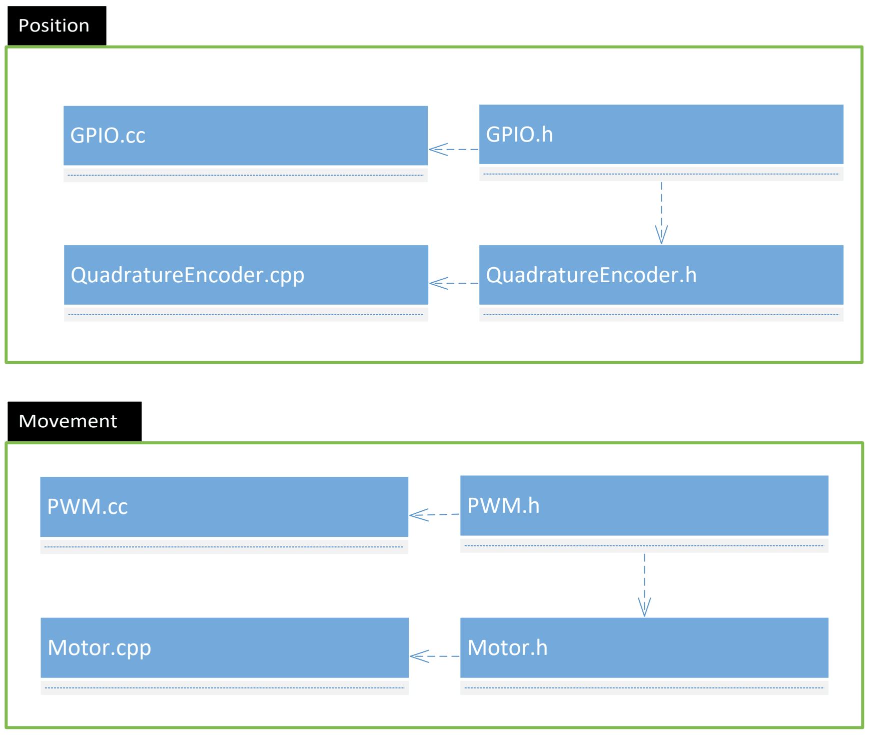


Figure 9. Position & Movement user space custom driver class examples.

These custom drivers represent tangible things that do not exist normally in Linux, for example a DC Motor that by the help of gluing PWM user space drivers that are provided by Linux can now be accessed from within C++ and have methods accessible to its upper layers.

CHAPTER 3: EXPERIMENTAL RESULTS

3.1 Closed Loop Real Time Control

It is a key factor in robots to have methods and means of controlling them. The lack of proper control the robot would make it go haywire and not fully control trajectories and target position. Figure 10 depicts the high-level architecture of where the proposed real-time control block fits.

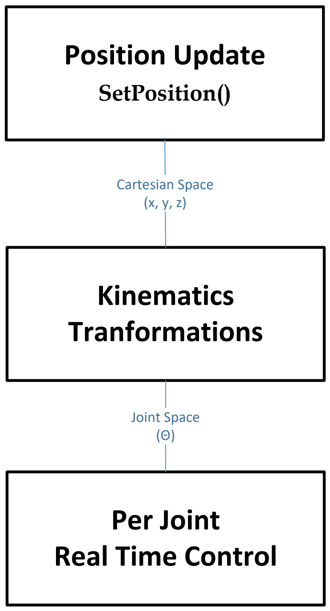


Figure 10. High-level sequence of events when the API changes position.

After the kinematic transformations occur and all of the per joint angles are calculated, those are named reference angle. A reference angle is the joint angle at which the joint should be in to target the robotic configuration and end up with the end-effector in the desired state.

It is important to note that some of the blocks in the processing chain are subject to real time constraints, controlling too late can result in erratic robot positioning, or bad jitter while taking too long on computing the kinematics control results in the robot having a huge lag or latency on changing its end-effector.

In the following section, we will be discussing the individual per joint real-time control block details.

3.1.1 Architecture & Implementation

The implementation is scalable from 1 up to N joints. There exists a reference angle theta for each of the joints in the robotic configuration.

We will be implementing an automatic control loop where by definition the controller compares a measured value of a process with a desired set value. As shown in Figure 11 the robot can be seen as controlled by uniquely independent **AutomaticControlThread** objects where the reference angles are feed directly. Once that each of these unique theta angle being input directly to the automatic control blocks, they will start moving themselves to reach the reference angle and eventually reaching the desired robot end-effector position.

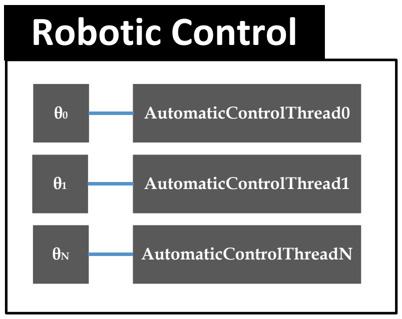


Figure 11. Automatic control with reference angles as inputs.

Where each of the individual control loops can be represented by Figure 12.The actual implementation is that of a simple proportional control.

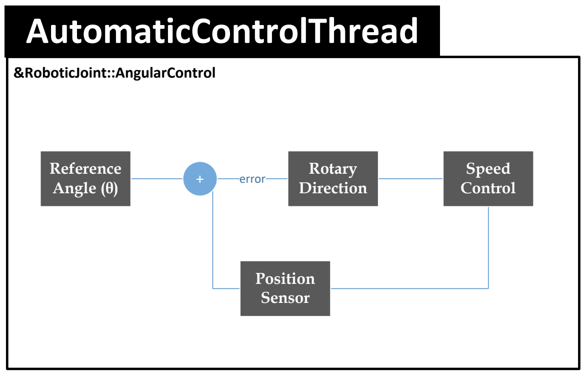


Figure 12. Automatic control per joint.

Depending on if the error is a positive number or negative that the direction of the rotation movement can change, and also the speed to which the DC motor is rotating at will depend directly on the magnitude of the error. While that is changing the position, sensor will feedback its angular data and compare it to our reference angle – as we eventually reach to a zero difference between reference angle and the actual sensor data which implies that that the control criteria has been met and thus the joint is at the desired angle.

3.1.2 Real Time Scheduler

Some robotic applications require low jitter and latency, so one may think that because we are using a general-purpose OS such as Linux that controlling the robot with it is going to be slow in terms of real time control.

The normal Linux kernel is preemptive not real time, a preemptive kernel allows the thread with higher priority to receive more CPU utilization time than a low priority thread. However, in the normal kernel no particular thread can monopolize the CPU utilization all the time.

Nonetheless, the Linux Kernel does support a RTOS mode where an important thing to note is that the thread can run if it is not pre-empted by threads which may have equal or higher priority According to Le Trung Thang the typical RTOS scheduler is just the real time Linux scheduler running with the Round Robin Policy [13].

Because each **AutomaticControlThread** object spawns as a processor thread in Linux with independent variables and address spaces the control portion of the robot can be set to a real time Round Robin priority and this is done in the C++ implementation.

3.1.3 Scheduler Configuration

There are two possible Linux RT scheduling options as mentioned in the Linux man pages or the Linux Programmers Reference [14].

a) SCHED_FIFO

When a thread becomes runnable, it will always immediately preempt any current running non-RT thread. The thread will run until either it is blocked by an I/O request, it is preempted by a higher priority thread, or it calls **sched_yield(2)**.

b) SCHED_RR

A simple enhancement of (a). Everything described above also applies, except that each thread is allowed to run only for a maximum time quantum.

For both of these scheduling real time policies we can select between ranges 1 (low) up to 99 (high). For reference, normal Linux processes will only run when there are no real-time threads running or ready to run, so the theory says that if there is work that needs to be done by the robotic application it will be performed and not starved by other processes such as hosting an HTTP server, or scanning and finding files, writing to disk and various others. We have opted on using a medium-low (19-39) priority values with SCHED_RR and have seen that it offers good performance. It is expected that different robotic configurations explore the possibility of switching between the two policies and decide on what priority works best for their application. Going on high priorities (50+) can starve the drivers and could potentially cause deadlocks on single threaded systems.

In the case of using really high real-time policies such as 80/90/99 we have seen reduced and degraded network throughput and performance such that I could not control my robot remotely through SSH properly. A clear benefit of using the Edison kit is the in-built IEEE 802.11 Wireless connectivity support that allowed me to connect my robot to the Internet and become a part of the Internet of Things (IoT).

3.1.4 Multithread Characteristics

As with multi-core scenarios exist, we must be splitting work without much of resource sharing as of to efficiently increase the performance. If we were to have a shared resource that required a semaphore or synchronization primitives this would almost immediately decrease our performance and increase jitter in most architectures [15] in most of RTOS implementations.

The current approach has been to simplify the control loop so that there are absolutely no semaphores dependencies and such that each unique spawned process has its own copy of independent variables. This allows the control loop to efficiently not because hardware stalls while fighting to achieve the lock and greatly reduce the jitter and have a more deterministic system behavior.

3.1.5 Results & Analysis

First, we will be analyzing the approach of using the two jointed robot arm on a system that does not use multi-threading for joint configurations.

A file in the repository [11] named **Robot_Diagnostics.cpp** has a diagnostics function that takes the time measurements and performs random samples of 1000 different random positions. The random positions where chosen to be in such a way that they do not reflect much of the robot mass and inertia so the criteria for the angle configurations were chosen to be of small 0 to 10 degrees.

The **Robot_Diagnostics.cpp** file was compiled with **–DDIAGNOSTICS** flag in order to execute 1000 samples on startup. This mode results in the application compiled to output the following raw data and logs as seen on the video [16].

As we also wanted to perform the experiment on the same hardware platform to have comparable results against single-core and dual-core we set to limit the running process to only use a single processor. We have chosen to us the **taskset** utility in the form of **taskset <COREMASK> <EXECUTABLE>**. This corresponds to the experiments that have the leftmost column of Table 2 set to a single processor. This methodology can also extend to bigger processor count target platforms in the future.

The CPU utilization time means that a system that has a single CPU and using 100% CPU utilization is taxing the single processor, while for a dual CPU system having a 200% utilization means that two processors are being taxed fully.

Table 2 shows that even for a simpler proportional control system we can achieve close to 43.26% of average latency reduction by going from one core to two cores.

System Configuration			Measurements			
Number of CPU Cores	RT Scheduler Policy	Number of Joints	Average CPU Utilization Time (%)	Average Latency to Target Position (us)	Median Latency to Target Position (us)	Standard Deviation (us)
1	SCHED_FIFO	2	98.81	847.32	139.00	21,457.99
	SCHED_RR	2	98.37	243.63	139.00	590.03
2	SCHED_FIFO	2	178.05	177.85	141	224.14
	SCHED_RR	2	173.75	177.07	143	189.18

Table 2. Performance characteristics scaling vs. scheduler configurations in the arm.

The median latency is a very meaningful data point as this is the 50th percentile latency of the requests, our robot performs the typical positioning action somewhere between 139 to 143 us. As you can see here there is no much difference here of using either a single core or dual core setup.

The standard deviation as seen in the rightmost column of Table 2 corresponds to the jitter the system has. Using a single-core system provides the worst jitter as the Linux scheduler will share the processor by other critical services such as networking and I/O subsystem and device drivers, by mixing the SCHED_FIFO scheduler with a single core provides the worst standard deviation of two orders of magnitude than that of the others – and there is a reason for this. The testing was performed through the network, network devices issue interrupts as their transmit and receive queues get full, and many device drivers behave quite similar, the nature of this huge jitter is that with this type of policy the task is allowed to run until work has been completed or a voluntary scheduler yield, but with all of the robotic specific interrupt handling being done with SCHED_FIFO and interrupts being asynchronous of nature this results in an extremely jitter situation that starve and block other processes for a huge amount of time.

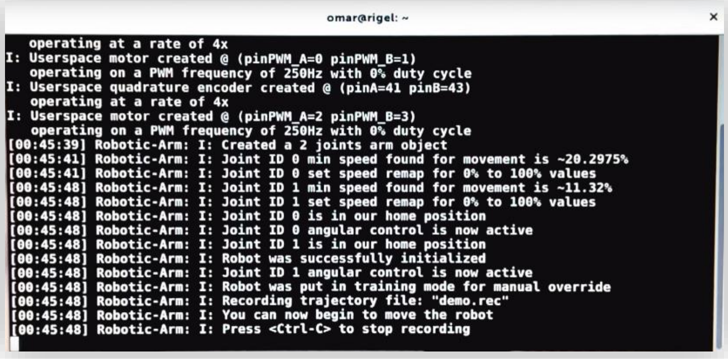
This jitter behavior is fixed by looking at the second row shows that SCHED_RR even with single core configurations provided quite a decent response for a robot with not that much jitter now. Showing that the Linux scheduler can function as a decent robotics solution for prototyping with latency responses under 1 millisecond.

But what is most important is that with the recent rise of multi-core embedded systems that the robot performance shines when utilizing all of the Edison platform capabilities of dual core. The SCHED_FIFO dual-core configuration is allowed to finish things faster by 2us due to less context switching but the ideal and most scalable solution seems to be multi-core SCHED_RR configurations, this reduced the jitter to almost nonexistent while keeping an even greater than the typical 50th percentile of the transactions under 143 us.

Multi-core SCHED_RR shines as the ideal experience on robotics where achieving reproducible and constant behavior is key while keeping other services such as network transmission or logging working. This allows making proper use of all of the Linux services and infrastructure without much interference on the robot performance.

Another demo that was developed using the same set of source code and framework is called **Robot_Record.cpp** and **Robot_Playback.cpp** [11] which consist of two more interactive applications using the software framework.

The recorder will start the robot without the automatic control loop thread running so that it can record and sense where the robot is being moved to. This lets you move the robot manually to start monitoring and record the trajectory and then dump it into a text file with coordinates and timestamps.



```
   operating at a rate of 4x
I: Userspace motor created @ (pinPWM_A=0 pinPWM_B=1)
   operating on a PWM frequency of 250Hz with 0% duty cycle
I: Userspace quadrature encoder created @ (pinA=41 pinB=43)
   operating at a rate of 4x
I: Userspace motor created @ (pinPWM_A=2 pinPWM_B=3)
   operating on a PWM frequency of 250Hz with 0% duty cycle
[00:45:39] Robotic-Arm: I: Created a 2 joints arm object
[00:45:41] Robotic-Arm: I: Joint ID 0 min speed found for movement is ~20.2975%
[00:45:41] Robotic-Arm: I: Joint ID 0 set speed remap for 0% to 100% values
[00:45:48] Robotic-Arm: I: Joint ID 1 min speed found for movement is ~11.32%
[00:45:48] Robotic-Arm: I: Joint ID 1 set speed remap for 0% to 100% values
[00:45:48] Robotic-Arm: I: Joint ID 0 is in our home position
[00:45:48] Robotic-Arm: I: Joint ID 0 angular control is now active
[00:45:48] Robotic-Arm: I: Joint ID 1 is in our home position
[00:45:48] Robotic-Arm: I: Robot was successfully initialized
[00:45:48] Robotic-Arm: I: Joint ID 1 angular control is now active
[00:45:48] Robotic-Arm: I: Robot was put in training mode for manual override
[00:45:48] Robotic-Arm: I: Recording trajectory file: "demo.rec"
[00:45:48] Robotic-Arm: I: You can now begin to move the robot
[00:45:48] Robotic-Arm: I: Press <Ctrl-C> to stop recording
```

Figure 13. Recorder tool waiting for input to stop.

This output file that was recorder can then be input of the playback application where it will replicate the recorder trajectory.

The sequence of pictures displayed in Figure 14 showcases the top row with the robot being manipulated manually and trained to perform the trajectory, while the bottom row has snapshots of the trajectory that was recorder being replayed. A full-length video of the robot utilities used for playback and recording was uploaded and can be viewed online [17].

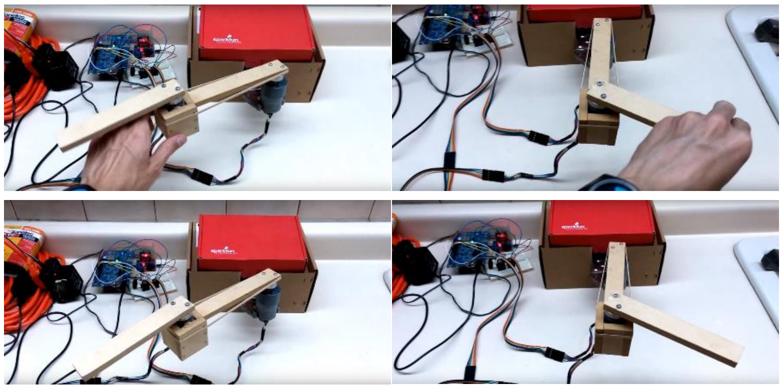
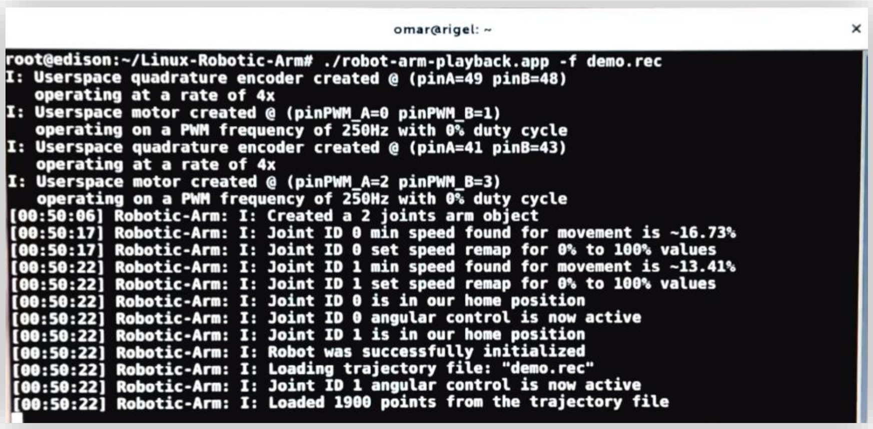


Figure 14. Recorded trajectory being replayed.



```
root@edison:~/Linux-Robotic-Arm# ./robot-arm-playback.app -f demo.rec
I: Userspace quadrature encoder created @ (pinA=49 pinB=48)
   operating at a rate of 4x
I: Userspace motor created @ (pinPWM_A=0 pinPWM_B=1)
   operating on a PWM frequency of 250Hz with 0% duty cycle
I: Userspace quadrature encoder created @ (pinA=41 pinB=43)
   operating at a rate of 4x
I: Userspace motor created @ (pinPWM_A=2 pinPWM_B=3)
   operating on a PWM frequency of 250Hz with 0% duty cycle
[00:50:06] Robotic-Arm: I: Created a 2 joints arm object
[00:50:17] Robotic-Arm: I: Joint ID 0 min speed found for movement is ~16.73%
[00:50:17] Robotic-Arm: I: Joint ID 0 set speed remap for 0% to 100% values
[00:50:22] Robotic-Arm: I: Joint ID 1 min speed found for movement is ~13.41%
[00:50:22] Robotic-Arm: I: Joint ID 1 set speed remap for 0% to 100% values
[00:50:22] Robotic-Arm: I: Joint ID 0 is in our home position
[00:50:22] Robotic-Arm: I: Joint ID 0 angular control is now active
[00:50:22] Robotic-Arm: I: Joint ID 1 is in our home position
[00:50:22] Robotic-Arm: I: Robot was successfully initialized
[00:50:22] Robotic-Arm: I: Loading trajectory file: "demo.rec"
[00:50:22] Robotic-Arm: I: Joint ID 1 angular control is now active
[00:50:22] Robotic-Arm: I: Loaded 1900 points from the trajectory file
```

Figure 15. Playback tool console output.

CHAPTER 4: CONCLUSIONS

4.1 Conclusions

Creating a robotics framework under Linux was a somewhat easy task for anyone familiar with developing UNIX based C++ applications. The use of the C++ language favored a lot at separating and creating objects that resemble real life things such as, robotic arms, joints, rotors, motors, sensors, and some others, this kept the code to be easily maintained and easy to read. There was some inspiration on Arduino's Servo Library [18] where we can have objects and various methods such as start and read states, but this was heavily enhanced because in my opinion using Arduino is good for hobby prototyping and not so good for academics and engineers as there is a lot of non-determinism and improper bug handling.

Having a robust OS such as GNU/Linux proved to be beneficial as a lot of the OS services have been well tested throughout the years and multiple hardware architectural support. The soft real-time mode of the Linux kernel also was found to be a nice addition for robotics as a lot of the hardware out there usually supports Linux. Boards such as the Raspberry Pi 1/2/3, the ODROID or the Intel's Edison and Galileo Boards are easily accessible to a lot of consumers and are found at schools throughout the world and obviously Personal Computers controlling robots are not left out since the framework supports everything that can run Linux.

While optimizations such as real time automatic control was proven to be used properly from within a GNU/Linux OS by properly compiling a RT Kernel for your board and modifying the scheduler to use one of the already mentioned real time policies. It was important to leverage a lot of the OS services already provided to get the implementation time down while keeping the return of investment on the end application high as possible.

I migrated from the Intel Galileo board which was a single core system with modest GPIO performance to use the Intel Edison in approximately 4 hours of work, so transitioning the project to a different platform proved to be almost no hassle. It was a matter of finding the documentation of the right GPIO pins to use, hook up the DC motors and sensors to it, compile the **Robot_Keyboard.cpp** and it was working.

There are a couple of crucial things to always keep in mind when planning to modify or enhance the control algorithm, avoiding any multi-core performance reducing events to be executed in the tight control loop, such things to avoid are having shared resources accesses in here, those would imply having semaphores and cause bus locks and would increase heavily the jitter. Also putting heavy I/O events such as writing to disk or the console is not recommended although somewhat useful only while debugging.

4.2 Future Work

The current framework has software support from 1 up to N joints. But there still work to be done in the kinematics transformation portion. Right now, only 1 DOF and 2 DOF manipulators have the solutions to the inverse kinematics problems. So, there is room to implement conformal geometric algebra [19] models and code them in C++ within the framework in order to test and improve the performance of higher DOF manipulators.

As we move into higher DOF robots it would be important to make use of the scalable threaded implementation and jump in to use a four-core or eight-core board so that we can utilize resources better. As higher core count systems become more popular and Moore’s law allows us to have more complex cheaper chips this would be an excellent use as a 6 DOF system will allow us to have any position within a three-dimensional space. We could have 6 real time threads assigned to 6 out of the 8 available cores controlling each joint, and still have one more for the application such as an ink drawing or printing program and another hosting web services via HTTP.

It would also be important to extend and create a visual encoder that reports the angle of a joint by using computer vision. This would allow to reflect the modularity of the software architecture by layers and objects such that different sensors can be used in the framework. This would also allow to use a single camera to report the multiple joint angles, instead of having separate quadrature encoder sensors. The framework allows using two or more sensors for feedback per joint so it would also be possible to have more accurate robot tracking by using both the visual encoder and the quadrature encoder at the same time.

And lastly, it might also be of useful experimentation to play with CPU isolation on multi-core systems, this allows keeping specific processors out of the CPU scheduler and Kernel and assign the robot application and interrupt processing to reside on a CPU and use all of the time. This sounds like a feasible approach on embedded systems that have more than two cores, on a quad-core system we could use two cores for the OS, services, Kernel and drivers while using the remaining two cores for the two joints fully.

REFERENCES

[1] Z. L. S. S. S. Richard M. Murray, "A Mathematical Introduction to Robotic Manipulation," CRC Press, California, 1994.

[2] P. A. A. L. H. M. J. L. M. S. Teemu Kanstren, "Robot-Assisted Smartphone Performance Testing," IEEE, 2015.

[3] S. W. (. F. J. M. a. R. T. Larry T. (Tim) Ross, "Fundamentals of Robotics," in *Robotics: Theory and Industrial Application*, The Goodheart-Willcox Co., 2011, pp. 30-34.

[4] Zortrax, "Zortrax Robotic Arm," [Online]. Available: http://library.zortrax.com/project/zortrax-robotic-arm/.

[5] R. G. a. R. R. Marc Brown, "Leveraing Linux to reduce software development costs," IBM Software Group, 2006.

[6] S. Hymel, "Loading Debian (Ubilinux) on the Edison," SparkFun, [Online]. Available: https://learn.sparkfun.com/tutorials/loading-debian-ubilinux-on-the-edison.

[7] S. M. (fatherlinux), "Architecting Containers Part 1: Why Understanding User Space vs. Kernel Space Matters," Red Hat, 29 July 2015. [Online]. Available: http://rhelblog.redhat.com/2015/07/29/architecting-containers-part-1-user-space-vs-kernel-space/. [Accessed 22 February 2017].

[8] H. A. a. R. Malhotra, "Embedded," Network Processing Division, Freescale, 19 November 2012. [Online]. Available: http://www.embedded.com/design/operating-systems/4401769/1/Device-drivers-in-user-space. [Accessed 20 February 2017].

[9] "Pulse Width Modulation (PWM) interface," The Linux Kernel Organization, [Online]. Available: https://www.kernel.org/doc/Documentation/pwm.txt.

[10] "GPIO Sysfs Interface for Userspace," The Linux Kernel Organization, [Online]. Available: https://www.kernel.org/doc/Documentation/gpio/sysfs.txt.

[11] O. X. Avelar, "oxavelar/Linux-Robotic-Arm," [Online]. Available: https://github.com/oxavelar/Linux-Robotic-Arm.

[12] T. Mercier, "tweej/HighLatencyGPIO," [Online]. Available: https://github.com/tweej/HighLatencyGPIO.

[13] L. T. Thang, "Comparing real-time scheduling on the Linux kernel and an RTOS," 25 April 2012. [Online]. Available: http://www.embedded.com/design/operating-systems/4371651/Comparing-the-real-time-scheduling-policies-of-the-Linux-kernel-and-an-RTOS-.

[14] R. Petersen, The Linux Programmer's Reference, 2 ed., Osborne Publishing, 1999.

[15] I. C. B. a. G. G. Z. Kashani, "On the Performance of Open-Source RTOS," IEEE, Italy, 2015.

[16] O. X. Avelar, "Linux Robotic Arm on Intel Edison," 2016. [Online]. Available: https://youtu.be/_ChBTMJYMkw.

[17] O. X. Avelar, "Intel Edison Linux Robot Arm : Recorder & Playback," 2017. [Online]. Available: https://youtu.be/boDd7_U0PQs.

[18] Arduino, "Arduino - Servo," [Online]. Available: https://www.arduino.cc/en/reference/servo.

[19] O. E. C.-E. a. E. B.-C. Luis Enrique González-Jiménez, "Geometric Techniques for the Kinematic Modeling and Control of," IEEE, 2011.