FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Implementation of Embedded Controllers for Scalable RIS Assemblies

Francisco Marques Mesquita

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Dr. Nuno Miguel Cardanha Paulino

October 20, 2025

Resumo

Esta dissertação explora o desenvolvimento de um sistema de controlo para *Reconfigurable Intelligent Surfaces (RISs)*, utilizando microcontroladores embebidos. As RISs são superfícies programáveis compostas por múltiplos elementos que podem manipular ondas eletromagnéticas de forma controlada, permitindo redirecionar, refletir ou focar sinais numa direção desejada.

Neste trabalho pretendeu-se desenvolver um sistema de controlo de uma configuração RIS composta por quatro *tiles* em duas plataformas embarcadas diferentes, o ESP32 e a Pico-Ice. O software foi implementado em ambos os controladores para permitir a configuração em tempo real de um sistema RIS. Estes controladores podem executar uma determinada ação com base numa série de comandos de alto nível que são enviados por um controlador central utilizando comunicação em série. Nomeadamente, o software permite o controlo de cada *tile* ou, alternativamente, pode controlar as quatro *tiles* em simultâneo.

Após a implementação, o desempenho global do software de controlo foi testado e optimizado. Foram implementadas melhorias, tais como tabelas de consulta trigonométricas e constantes de conversão pré-computadas, para reduzir o tempo de cálculo. As versões optimizadas foram avaliadas em vários tamanhos de *tiles* e foi efetuada uma análise comparativa entre os dois microcontroladores.

Os resultados mostram que o ESP32 alcançou tempos de execução significativamente mais baixos do que a Pico-Ice ao executar o algoritmo de *beam steering*. No entanto, uma vez que a Pico-Ice também possui um field-programmable gate array (FPGA), poderá mais tarde ser explorada para delegar alguns cálculos, diminuindo ainda mais o tempo de computação.

Ao longo deste trabalho demonstrou-se que o desenvolvimento de um sistema de controlo RIS eficaz é viável em microcontroladores de baixo consumo energético e estabelece uma base que pode ser utilizada para introduzir outras melhorias.

Abstract

This dissertation explores the development of a control system for RISs using embedded microcontrollers. RISs are programmable surfaces composed of multiple elements capable of manipulating electromagnetic waves in a controlled manner, allowing them to reflect, redirect, or focus signals in a desired direction.

This work focuses on developing a four-tile RIS setup control system on two different embedded platforms, the ESP32 and the Pico-Ice. The software was implemented on both controllers to allow real-time configuration of a RIS system. These controllers can perform a determined action based on a series of high-level commands that are sent from a central controller using serial communication. Namely, the software enables the control of each tile or, alternatively, it can control the four tiles simultaneously.

After the implementation, the overall performance of the control software was tested and optimized. Improvements, such as trigonometric lookup tables and precomputed conversion constants, were implemented to reduce computation time. The optimized versions were evaluated across several tile sizes, and a comparative analysis between the two microcontrollers was performed.

Results show that the ESP32 achieved significantly lower execution times than the Pico-Ice when running the beamforming algorithm. However, because the Pico-Ice also has a built-in FPGA, it could later be explored for offloading some calculations to further decrease computation time.

This work demonstrates that developing an effective RIS control system is feasible on low-power microcontrollers and lays a foundation that can be built upon to introduce further enhancements.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. They include 17 goals addressing critical global challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This dissertation contributes to several of these goals by proposing and implementing a low-power embedded control solution for Reconfigurable Intelligent Surfaces (RIS). These surfaces are emerging as a key enabling technology for future wireless communication systems, particularly 6G, which aim to improve energy efficiency, coverage, and reliability.

The specific Sustainable Development Goals addressed in this work are the following:

SDG 7 Ensure access to affordable, reliable, sustainable and modern energy for all.

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.

SDG	Target	Contribution	Performance Indicators and Metrics
7	7.3	The use of energy-efficient embedded platforms like the ESP32 and Pico-Ice for RIS control demonstrates ways to reduce energy consumption in communication infrastructure.	Reduced power usage per RIS configuration and control operation.
	7.a	This work provides a scalable software platform that could be used in future RIS-enabled smart grids and IoT systems, fostering clean energy research and deployment.	Software scalability and compatibility with renewable-powered communication systems.
9	9.5	Supports technological innovation by implementing and optimizing RIS beam steering algorithms on embedded systems, contributing to ongoing 6G research.	Implementation of novel control techniques, performance comparison across platforms.

Contents

List of Figures

List of Tables

List of Acronyms

FPGA	Field-Programmable Gate Array
FR4	Flame Retardant 4
GHz	Gigahertz
I2C	Inter-Integrated Circuit
IoT	Internet of Things
IWW	Intelligent Wireless Walls
LED	Light-Emitting Diode
LoS	Line of Sight
MEMS	Micro-Electro-Mechanical Systems
MHz	Megahertz
PCB	Printed Circuit Board
PIN	Positive-Intrinsic-Negative
PSRAM	Pseudostatic Random-Access Memory
qSPI	Quad Serial Peripheral Interface
RF	Radio Frequency
RIS	Reconfigurable Intelligent Surface
SoC	System on Chip
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
UART	Universal Asynchronous Receiver/Transmitter

Chapter 1

Introduction

1.1 Context

Although we have yet to fully adopt 5G, it is clear that certain applications, such as large-scale industrial Internet of Things (IoT), require greater data throughput, lower latency, and reliability. 6G technology can solve these challenges. By leveraging 6G networks, it is possible to achieve higher levels of precision and reliability in location-based systems, which can lead to an improvement in navigation systems, virtual reality systems, health-care monitoring devices and autonomous driving. Thus, recent research and standardization efforts have focused on fully realizing 6G systems. However, despite its advantages, 6G high-frequency transmission is more prone to measurement errors when the line of sight (LoS) between the emitter and receiver is lost [2]. By utilizing RISs this issue is drastically mitigated.

1.2 Motivation

RISs have proven to be extremely promising technologies for enhancing wireless communication systems beyond 5G and 6G, mainly due to the fact that they have the ability to expand the capacity and coverage of such systems.

RISs can be particularly useful to ensure reliable communication in scenarios without a clear line of sight (LoS). Cellular networks can benefit by leveraging this technology. If the LoS is obstructed by an object, for example, a high-rise building, a virtual line of sight link can be established between the signal emitter and the and the network user through a RIS, leading to a substantial improvement in spectral efficiency. Another major advantage of RISs is their power efficiency.

They can be deployed between the signal source and the user and can be controlled by embedded platforms. These controllers are responsible for adjusting the RIS configuration to maintain optimal performance. Modern microcontrollers are increasingly more capable of achieving this control efficiently.

A RIS is composed of a matrix of patch antennas, also called unit cells, or elements. The electromagnetic characteristics of each cell can be reconfigured to manipulate the propagation medium. Various control mechanisms can be employed to achieve this, such as thermal excitation, optical pump, physical stretching, and electrical control. The latter is the simplest approach, mainly because digital voltage levels are easier to control [6].

Methods to determine the required phase delay of each element, in order to generate a signal reflection in the shape of a focused beam, are well established, both for far-field and near-field conditions. However, focus on implementation of the control and firmware layers, and on optimizing the execution of RIS control algorithms for real-time at a large scale seems to be underexplored in the literature. Therefore, this work explores how this implementation could be achieved and how some optimizations could increase its performance.

1.3 Objectives

The first objective of the dissertation work was to implement a control system for a RIS using an embedded platform, such as a PicoIce and an ESP32 device. This involves developing software that can receive commands from a host device in order to control the surface. To achieve this, it was necessary to define these commands as well as decide the communication interface used between the host and the microcontroller.

After the control system was functioning properly, the second objective was to evaluate the feasibility of speeding up the beam-forming calculations by introducing optimization to the beam steering calculations.

Finally, the third objective was to conduct a performance analysis to evaluate the computational efficiency of the software running on both platforms. The effects of the applied software optimizations were assessed through benchmarking across different array sizes, allowing for a comparative evaluation of execution times.

1.3.1 Structure of the Document

This document is organized into five main chapters. In the Introduction chapter, the context, motivation, and the main objectives of the dissertation work are presented. It explains the main advantages of the utilization of RIS and what existing problems it could solve. The decision to use embedded microcontroller platforms for RIS control is also justified.

In chapter 2, the current state of the literature is reviewed. Different RIS implementation methodologies are covered, as well as practical implementations and their application in various situations.

Chapter 3 describes the proposed system architecture, including both the hardware and software components. The roles of the ESP32 and Pico-Ice platforms are explained, and the RIS emulation setup using an LED printed circuit board (PCB) is explained. The structure of the control software is also presented, along with the applied optimization strategies.

Chapter 4 presents the results of the system validation and performance evaluation. The computation time benchmarks for various RIS tile sizes are analyzed, and the performance impact of software optimizations is assessed for both platforms.

In chapter 5, the main findings are summarized and the potential contributions of the work are addressed.

Chapter 2

Related Work

2.1 Background

2.1.1 Reconfigurable Intelligent Surface Applications

Reconfigurable Intelligent Surfaces (RISs) can be deployed on various surfaces, including buildings, billboards, moving vehicles, indoor walls, and even on flexible substrates like clothing, where smaller RISs are utilized [6]. They are cost-effective since they are easily deployed, highly flexible, because they can be integrated into existing networks without needing to upgrade the existing hardware. Another two major advantages of the deployment of RISs are their energy efficiency, since a dedicated energy source is not required for radio frequency processing because they are composed of passive elements, and the absence of noise amplification because they do not contain amplifiers.

As shown in Figure 2.1, in addition to enhancing 5G or 6G networks, RISs can also improve IoT networks, assist with indoor communications (such as in virtual reality applications), and enhance control of unmanned systems.

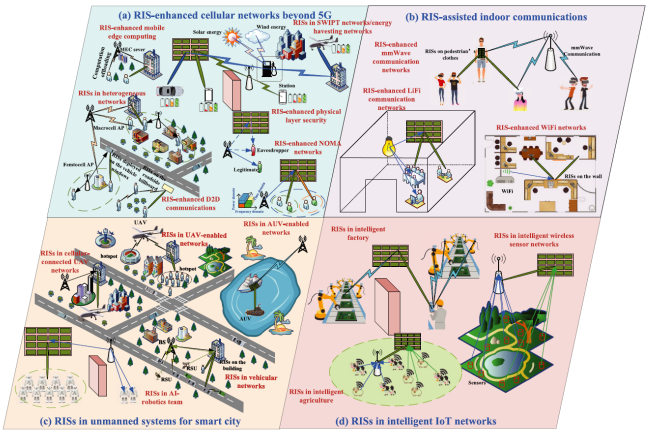


Figure 2.1: Examples of RIS enhanced networks [6]

Figure 2.2 illustrates a RIS structure, including the surface layer containing the aforementioned patch antenna elements. These elements are tunable and can interact with incident electromagnetic waves. The copper backplate, situated on the intermediate layer, serves as a way to prevent signal leakage. Finally, the last layer is responsible for the tuning of the reflective elements. This layer interfaces with a RIS controller used to configure the state of each unit cell [2].

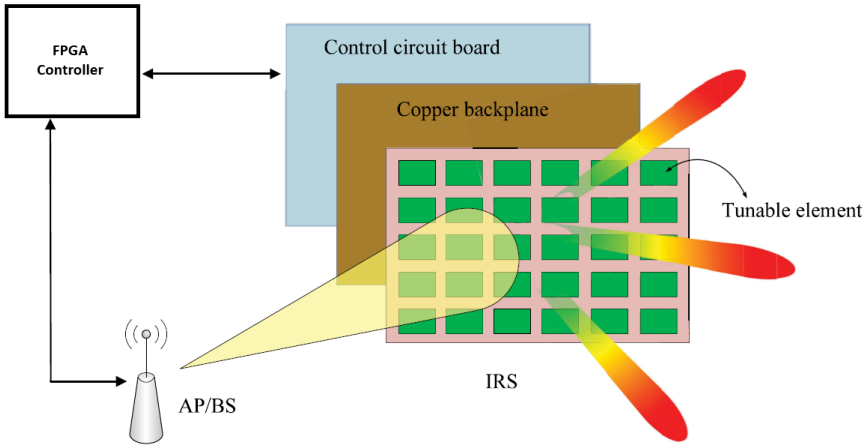


Figure 2.2: RIS operational principle and control scheme [2]

In Figure 2.3, the typical function of RISs is presented. Beamforming or beam focusing is a signal processing technique that focuses a signal in a specific direction, forming a beam. This can be accomplished by the phase shift introduced by each RIS element. One of the main advantages of beamforming is the ability to enhance wireless communication performance without requiring active amplification. By modifying the phase of each reflecting element, RISs can enhance the signal strength at the receiver.

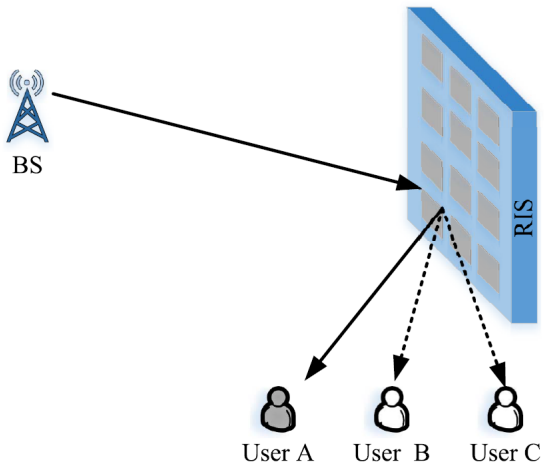


Figure 2.3: Beamforming illustration [6]

2.1.2 RIS Control Methods

A RIS is a controllable surface in the sense that it is possible to configure its electromagnetic response. Specifically, the response of each individual unit cell can be changed using switching and tuning mechanisms integrated into the surface. By controlling each element that composes the RIS we can steer a determined signal by setting a configuration that creates constructive interference in a desired direction. There are several methods to do this, namely electrical, mechanical, optical, and smart materials-based methods, with the most common being electrical control [9]. In this section, the different techniques are explored as well as their advantages and disadvantages.

Electrical Reconfiguration

To employ electrical reconfiguration, each element of the RIS is connected to a discrete component that acts as a switch, which can be individually manipulated. This changes how the current that goes through the surface is distributed, resulting in a different frequency response. These switches can be positive-intrinsic-negative (PIN) diodes, varactors, and Micro-Electro-Mechanical Systems (MEMS) [9].

PIN diodes act as binary switches to alter current paths in the RIS surface, thus changing the surface's reflection or transmission characteristics. They are inexpensive when compared to other alternatives and highly reliable.

Varactors are another type of electrical switch. They offer continuous frequency tuning by varying their voltage, which, in turn, changes their capacitance. They are easy to integrate and allow for smooth frequency control, however, they require complex bias circuitry and are more susceptible to nonlinearity issues [9].

The last type of switch usually used in RISs implementations is MEMS. Despite involving physical movement, these devices are actuated through electrical mechanisms such as electrostatic, thermal, or magnetostatic forces, distinguishing them from mechanical reconfiguration techniques.[9]. They have low power losses and have a relatively low susceptibility to noise. Despite these advantages, they also have limited durability and are relatively slow when switching states in comparison to other switches[9].

Mechanical Reconfiguration

Mechanical control of a RIS involves physically altering the current state of each RIS element with the help of a mechanical mechanism to change the surface's electromagnetic characteristics [7]. This methodology usually does not greatly impact the normal operation of the element because the mechanical mechanisms are installed beneath the surface layer [1]. Despite this advantage, the time needed to update the tunable element can be greater when compared to other control methodologies [1]. Figure 2.4 is an example of this control method, where the actuator is a motor that can be used to rotate the unit cell to achieve a determined desired configuration.

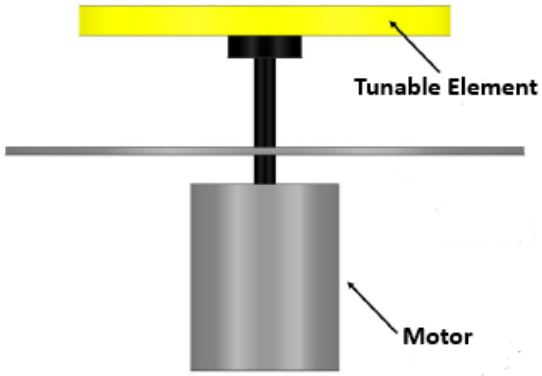


Figure 2.4: Mechanical reconfigurable unit cell [7]

Optical Reconfiguration

To optically control a RIS light can be used to control photoconductive materials present in the tunable elements [9, 13]. The switches used in the method are able to switch states faster than other methods, and because of the lack of extra wiring and bias lines, they are less prone to electromagnetic interference [9]. However, when compared to other methods, they have a complex activation mechanism and are more prone to loss.

Smart materials

Emerging research has explored smart or responsive materials, such as liquid metals and graphene to develop reconfigurable antennas. These materials can be deformed, moved, or reconfigured via electrochemical stimulation [8]. Their main advantages are the low profile and lighter weight of the resulting reconfigurable antenna. Nonetheless they have limited efficiency when compared to other methods [9].

2.1.3 Mathematical Background of Far-Field and Near-Field Configurations

The way the reflection phase profile is configured across the RIS surface directly impacts how the surface can manipulate incident waves. The mathematical framework presented in [3] models the behavior of a binary-phase RIS under two configurations: near-field and far-field.

General Principle of Phase Control

Each RIS element can introduce a phase shift to the incoming wave. This phase shift is determined by the difference between the desired reflected wave phase, $\phi_r(x_n, y_m)$, and the phase of the incident wave, $\phi_i(x_n, y_m)$, with x_n and y_m being the coordinates of each element.

$$\phi_{nm} = \phi_r(x_n, y_m) - \phi_i(x_n, y_m) \tag{2.1}$$

Since each element can only represent two different states, either on or off, in the model presented in [3], it is established that the phase shift will be set as shown below.

$$\phi_{nm} = \begin{cases} 0, & \text{if } -\frac{\pi}{2} \leq \phi_r(x_n, y_m) - \phi_i(x_n, y_m) \leq \frac{\pi}{2} \\ \pi, & \text{otherwise} \end{cases}$$

The total radiated field, taking into consideration every element, in the direction (θ, φ) is approximated as [3]

$$E_r(\theta, \varphi) = \cos(\theta) \sum_{n,m} \Gamma_{nm} E_i(x_n, y_m) \cos(\theta_{nm}) \exp\left(-jk \sin(\theta)[x_n \cos(\varphi) + y_m \sin(\varphi)]\right) \tag{2.2}$$

where Γ_{nm} is the reflection coefficient of an element, E_i is the incident field, $\cos(\theta_{nm})$ represents the angle at which the element views the transmitting antenna, θ is the zenith angle, and ϕ is the azimuth angle. Figure 2.5 illustrates the angular relations between a RIS tile and the signal emitter and receiver.

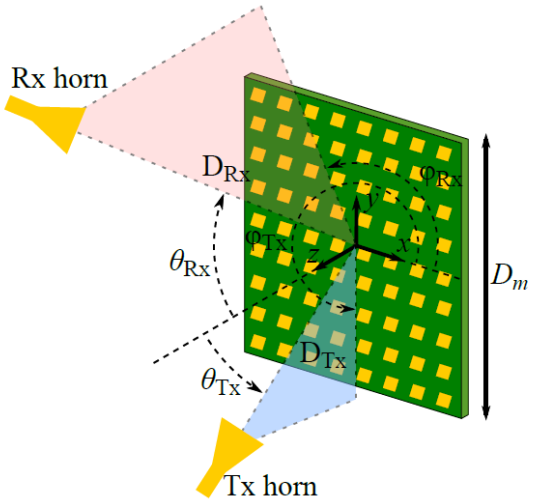


Figure 2.5: Representation of the angular relations in a RIS setup [3]

Near Field Configuration

In a near-field configuration, the signal emitter is located near the RIS tile. In this situation, the phase shift that needs to be introduced by each RIS element can be represented by the following expression [3]:

$$\phi_{nm} = -k\sin(\theta_{\mathrm{Rx}})[x_n\cos(\varphi_{\mathrm{Rx}}) + y_m\sin(\varphi_{\mathrm{Rx}})] + k\sqrt{(x_n - x_{\mathrm{Tx}})^2 + (y_m - y_{\mathrm{Tx}})^2 + z_{\mathrm{Tx}}^2} \tag{2.3}$$

The coordinates of the transmitting antenna in the spherical coordinate system are expressed as:

$$\begin{aligned} z_{\mathrm{Tx}} &= D_{\mathrm{Tx}}\cos(\theta_{\mathrm{Tx}}) \\ x_{\mathrm{Tx}} &= D_{\mathrm{Tx}}\sin(\theta_{\mathrm{Tx}})\cos(\varphi_{\mathrm{Tx}}) \\ y_{\mathrm{Tx}} &= D_{\mathrm{Tx}}\sin(\theta_{\mathrm{Tx}})\sin(\varphi_{\mathrm{Tx}}) \end{aligned}$$

Far Field Configuration

In a far-field configuration, the signal transmitter is placed far away from the RIS, in this case, the incident wave can be approximated as a planar wave and the required phase shift simplifies to [3]:

$$\phi_{nm} = -k\sin(\theta_{\mathrm{Rx}})[x_n\cos(\varphi_{\mathrm{Rx}}) + y_m\sin(\varphi_{\mathrm{Rx}})] + k\sin(\theta_{\mathrm{Tx}})[x_n\cos(\varphi_{\mathrm{Tx}}) + y_m\sin(\varphi_{\mathrm{Tx}})] \tag{2.4}$$

In this work, the developed software enables the reconfiguration of the desired positioning, meaning, both configurations are possible.

2.2 State of the Art

2.2.1 Open Source RIS for the frequency Range of 5 GHz WiFi [5]

The authors present a RIS, depicted in fig. 2.6, developed specifically for the range of 5 GHz. Although there has been increasing research interest in RISs, their recent emergence as a technology means that access to such systems for experimental purposes remains limited. The authors of this paper address this problem by making the design, fabrication data, and firmware source code available to the public.

The surface works in the 5 GHz range, mainly because this frequency range is already well established in existing technology. The authors also state that a 55 millimeter wavelength makes the realization of the RIS possible with a moderate amount of effort that allows the implementation of an array of hundreds of reflecting elements [5].

The unit cells that make up the RIS are designed so that they can be implemented using only one printed circuit board (PCB). This helps simplify the manufacturing process. Flame Retardant 4 (FR4) substrate was used in the design of the PCB in order to decrease the production cost.

This particular RIS can be connected to a computer for its control and power source. However, wireless control is also possible via Bluetooth [5].

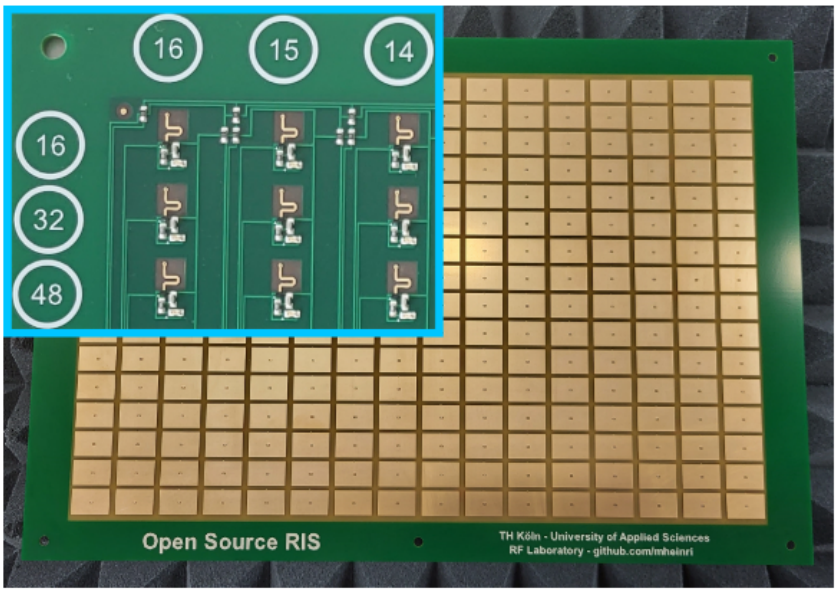


Figure 2.6: Front and backside view of the fabricated RIS composed of 256 unit cells [5]

2.2.2 Measurement System and Methodology for RIS Evaluation [4]

A methodology is proposed to test the performance of each element that composes a determined RIS [4]. The authors state that most research projects on RISs rely purely on the assumption that the manufacturing process of these surfaces was conducted without flaws. Because a RIS is usually composed by an array of hundreds of elements, if only a small amount is defective, it would be extremely difficult to identify measurement errors when only taking into consideration the overall performance of the surface. This problem can be addressed by employing rigorous testing for every element. Hence the authors propose a methodology to achieve this.

To perform these tests, a test antenna is positioned above the element to be tested in order to perform a near-field scan [4]. The test antenna is moved by a motor system that ensures that it is placed directly on top of each element with a minimal gap between the test antenna and the element. The measurement setup can be seen on Figure 2.7

The test consists in measuring the input impedance of the test antenna. Subsequently, by switching the state of the element, the recorded impedance greatly fluctuates. This change is then compared to a reference value and if these values differ considerably, then one can conclude that the element is defective.

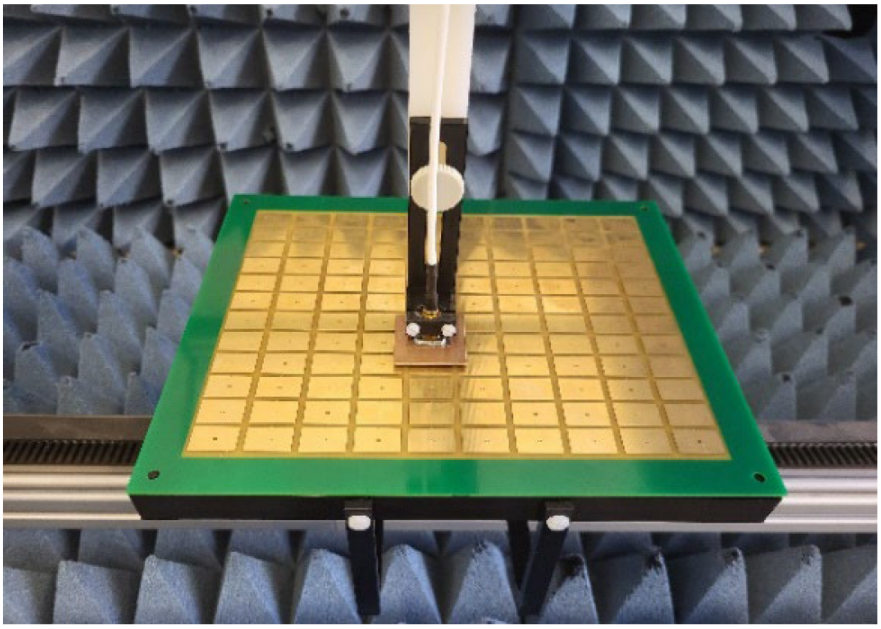


Figure 2.7: Measurement Setup [4]

2.2.3 Varactor-Based Reflecting Metasurface with Dual-Linear Polarization for Low Power RIS [10]

The authors start by addressing the fact that in the future wireless systems might compensate for high variability in the wireless channel at the transmitter, at the receiver, and even in the propagation environment. This can be accomplished by utilizing RISs.

Although varactor-based surfaces have a higher level of complexity compared to other approaches, for example, PIN diode-based RIS setups, they have a significantly better power efficiency. This is because a bias current is not required for operation. They also have a high degree of flexibility when it comes to surface tuning, because the surface reactance can be adjusted at any time [10].

For the aforementioned reasons, the authors propose a varactor-based metasurface design. Each unit cell, whose geometry was optimized using an algorithm to mitigate phase error and minimize reflection loss, has two tuning elements [10].

The proposed surface design includes a control circuit composed of high-voltage shift registers that can control the varactor diodes of 16 unit cells [10]. An example of this surface can be seen in Figure 2.8. In order to control the shift registers present in the surface, the surface can be connected to a Raspberry Pi.

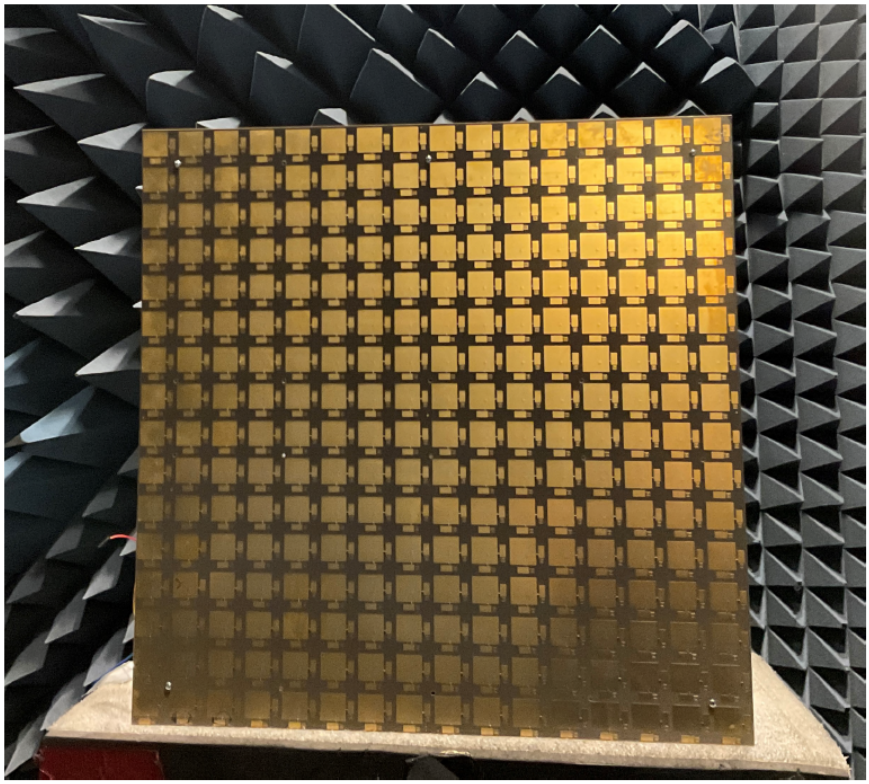


Figure 2.8: RIS prototype [10]

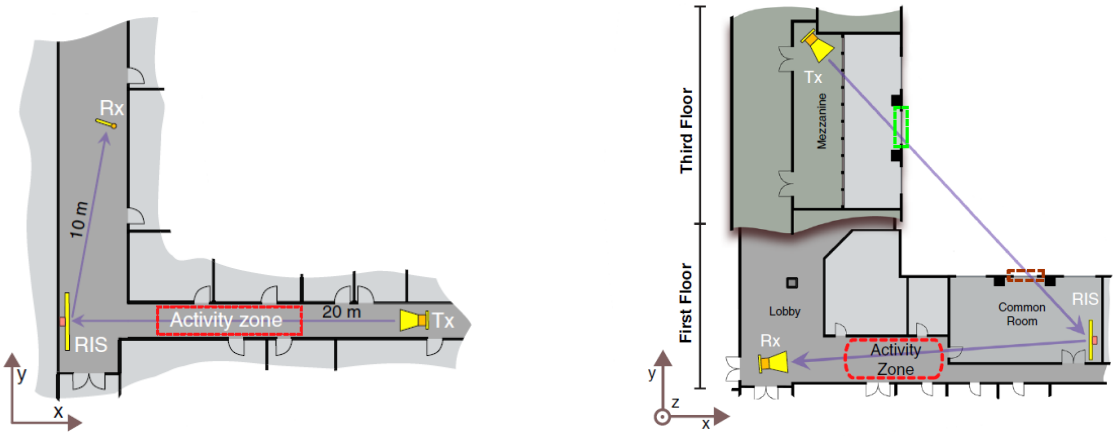
2.2.4 Intelligent walls for in-home monitoring [11]

According to the authors of the article [11] , quite a lot of research has been done on human monitoring devices. These devices can be extremely useful mainly to help older people and people with disabilities. Some examples of this technology are health-oriented wearables, which already have been widely adopted, and the use of cameras for human monitoring. However, these technologies have some drawbacks. In the case of the wearable devices they often require the user to wear them for long periods of time which means they have to be carried around wherever the user goes, this can prove to be quite inconvenient. When it comes to the use of cameras, they can be intrusive to the user's privacy, for example, when they need to be installed on someone's home.

To overcome these disadvantages, different technologies have been studied. However, the authors mainly focus on microwave-based activity monitoring. This method consists in transmitting microwave signals and analyzing variations in the reflected signals that are caused by the monitored target's motion. This has the advantage of not invading the user's privacy. Nonetheless, it also has some disadvantages. This technology has a limited range of only a few meters according to the authors [11] and when there is no direct line of sight between the emitter and receiver, the detection accuracy decreases greatly. This method is also very sensitive to interference signals, which can result in a decreased overall performance.

To solve these limitations, the utilization of intelligent wireless walls (IWW), which utilize built-in RISs, is proposed. By leveraging the RIS capabilities it is possible to steer electromagnetic waves in the desired direction, which leads to an improvement in signal strength and accuracy. RISs also have the added benefit of being relatively inexpensive to manufacture and are very energy efficient. Using machine learning algorithms with this technology can also be very effective. It can help enhance the system's ability to improve detection when there is no direct line-of-sight as well as effectively detect and distinguish different types of movement.

The tests were carried out in two different scenarios, namely a corridor junction, fig. 2.9a, where the transmitter and receiver are placed in adjacent corridors [11] and a multi-floor scenario, fig. 2.9b, where the transmitter and receiver were installed two floors apart [11]. These were chosen because in both instances there is no direct line of site between the emitter and receiver. Therefore, it is possible to evaluate if the utilization of a RIS can compensate for the lack of effectiveness of conventional microwave sensing [11].



(a) Corridor junction

(b) Multi-floor setup

Figure 2.9: Testing scenarios [11]

Two individuals, one male and one female, performed three different actions in each scenario, sitting, standing and walking. The participants started by performing every action with the RIS turned on and then proceeded to repeat them with the RIS turned off. After all experiments were conducted, the data was preprocessed and then fed to machine learning algorithms in order to classify each action. These actions can be distinguished by identifying amplitude variations in the captured signal by the receiver. However, when the RIS was not used, these variations were simply too low, making it difficult to achieve the correct distinction between actions. Nevertheless, when the RIS was turned on, the amplitude variations were easily distinguished between actions. The authors state that the maximum detection gain in the corridor junction and in the multi-floor setting, there is a maximum gain of detection of respectively 25% and 28% [11].

It is also stated that the proposed system can be adapted to include the detection of other similar actions, however, it would need to be tuned and scaled in order to also accommodate actions that record variations on a much smaller scale, therefore requiring a higher level of precision.

2.2.5 High-Accuracy Reconfigurable Intelligent Surface Using independently Controllable Methods [12]

Unlike traditional methods, the proposed system in [12] allows the control of an individual RIS element which in turn improves the accuracy of the beamforming calculations. To achieve this, the PIN diodes can be activated by a field-programmable gate array (FPGA) device which enables the control of the elementary cells, these cells are composed by 2-bit elements that can accommodate four different phase states for the eletromagnectic waves [12]. Because the PIN diodes can be individually controlled by demultiplexing, which enables the control of a large number of PIN diodes with a limited amount of FPGA pins, the proposed approach leads to an increase in measurement accuracy [12].

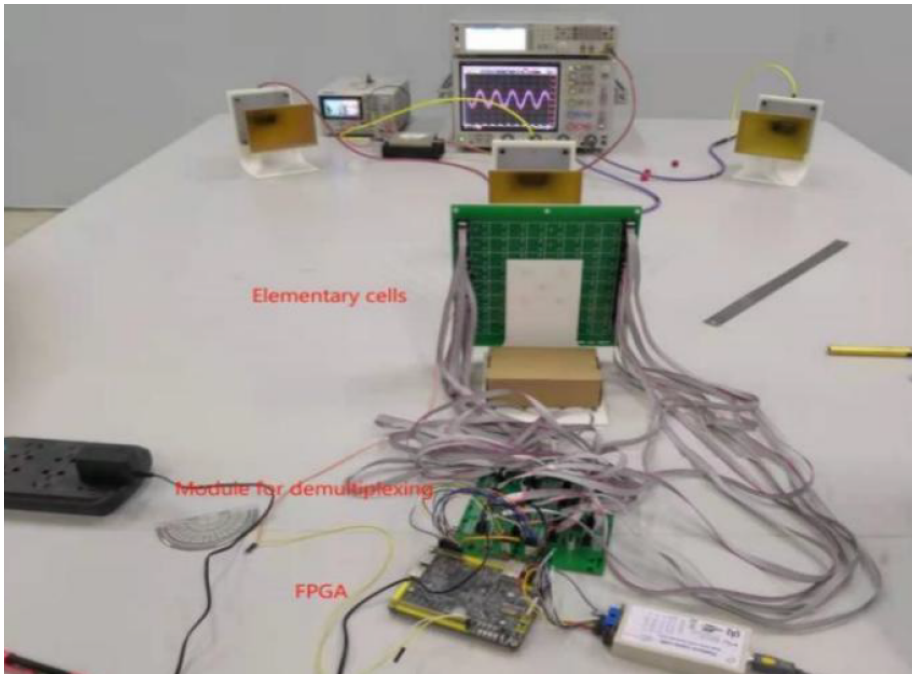


Figure 2.10: RIS prototype [12]

The testing setup of the manufactured RIS composed by 64 elements, show in fig. 2.10, was carried out in an environment that minimizes unwanted signal interference. An antenna was used to send an electromagnetic wave towards the RIS and two receiving antennas were installed to capture the signal reflected by the surface [12]. This experiment led to the conclusion that the RIS successfully adjusted the reflection angle within a 30° range with high accuracy. The error of the reflected angle was less than 1° compared to the predetermined angle.

2.3 Summary

The following table summarizes all the proposed RIS characteristics. The frequency column represents either the frequency range or the center frequency when the author does not specify the

range.

Table 2.1: Summary table of RIS designs

Reference	Number of Elements	Control	Frequency
[5]	16x16	RF switch	5.15 - 5.87 GHz
[10]	48x16	Varactor	3.3 - 3.7 GHz
[11]	48x48	PIN diode	3.75 GHz
[12]	8x8	PIN diode	-

Chapter 3

Proposed Approach

3.1 Hardware

3.1.1 System Architecture

This section presents the overall system's architecture that is responsible for controlling the RIS tiles. The system is composed of two layers: a central controller, which is implemented on a laptop, and a local controller that is implemented on a microcontroller such as an ESP32 or a Pico-Ice. The communication between these layers enables dynamic control of the RIS tiles according to predefined rules or external inputs.

The representation of the system's architecture is visible in Figure 3.1. The arrows illustrate the flow of information in order to clarify the interaction between the different system components.

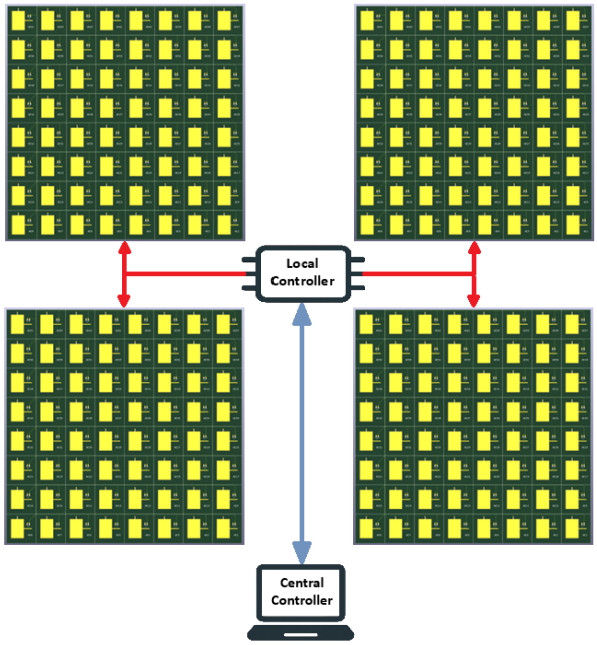


Figure 3.1: System's Architecture

Central Controller

The central controller, running on a laptop, is responsible for sending high-level commands that can be interpreted by the local controller. It is essentially a Python script that can either instruct the microcontroller to update the state of a determined tile by sending the desired steering angles, or it can provide the current state of a determined tile. In the Software section, the script as well as the interaction between it and the local controller will be explained in more detail.

Local Controller

The local controller is implemented on an embedded platform. There are two versions of the working code, one for an ESP32-S3-DevKitC-1, which will be referred to as ESP32 throughout the document, and the other for a Pico-Ice board. These boards act as an intermediary control between the central controller and the RIS tiles.

Its functionalities include:

- Receiving and parsing the commands sent by the central controller
- Performing beam steering calculations
- Sending control data to the RIS tiles
- Displaying the tile's state

RIS Tiles

Ideally, the system would be tested with an arrangement of four RIS tiles, however, due to material constraints, a PCB with LEDs that each represent a unit cell, with an eight-by-eight configuration, was used to emulate a tile. If an LED is lit, it means that the respective unit cell that would be in its place is in the ON state and vice versa.

The data stream is sent by the local controller via Serial Peripheral Interface (SPI), and the eight shift registers present on the PCB propagate the control bits, lighting up the necessary LEDs. The circuit board in question, shown in fig. 3.2, illustrates an example of a beam steering pattern.

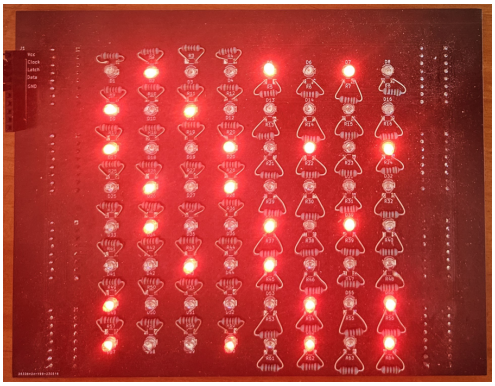


Figure 3.2: Beam steering pattern on the LED PCB

3.1.2 Microcontroller Platforms

This section presents the embedded platforms used in the development and implementation of the local control unit of the system. Two different platforms were selected in different phases of development: the ESP32-S3-DevKitC-1 and a Pico-Ice.

ESP32-S3-DevKitC-1

The ESP32-S3-DevKitC-1, shown in Figure 3.3 is a development board featuring Espressif's ESP32-S3 System on Chip (SoC). These are the board's most relevant features:

- Processor: Dual-core Xtensa LX7 - up to 240 MHz
- Memory: 512 KB SRAM and 8 MB PSRAM
- Integrated wireless communication (Wi-Fi and Bluetooth)
- Available interfaces: SPI, UART, I2C, etc.

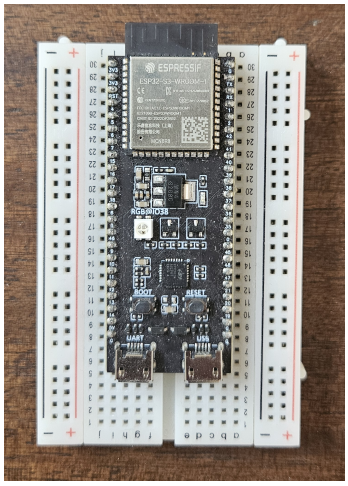


Figure 3.3: ESP32-S3-DevKitC-1 board

The ESP32 was selected as the local controller during the early development stages of the system due to its prior integration in related RIS tile designs developed within the scope of this work. In addition, existing familiarity with the platform within the research group at INESC TEC facilitated faster onboarding and technical support. It also has the benefit of having a large developer community and, because of this, there are a lot of available resources that could be used to expedite the development during this phase.

Pico-Ice

The Pico-Ice, shown in Figure 3.4, is a development board that combines the Raspberry Pi RP2040 microcontroller with a Lattice iCE40UP5K FPGA. It enables the development of applications requiring both software flexibility and hardware-level parallelism.

- Processor: Dual ARM Cortex-M0+ - up to 133 MHz
- Independent 4MB qSPI flash for the RP2040
- External FPGA memories: 4MB qSPI Flash, 8MB low-power qSPI SRAM
- Lattice UltraPlus ICE40UP5K FPGA

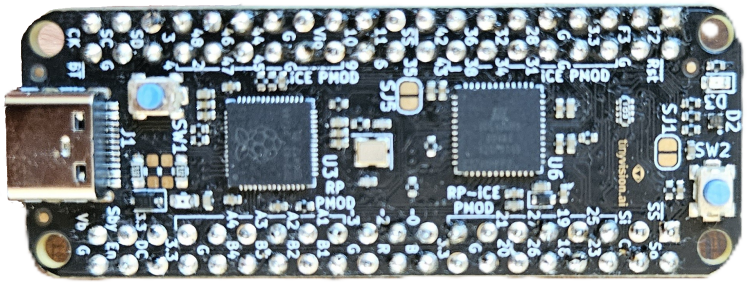


Figure 3.4: Pico-Ice board

Because this microcontroller is based on the RP2040, which is the same chip used in the Raspberry Pi pico, there is also a wide availability of resources that can be used to support development. However, it does not possess a Wi-Fi module, therefore, only serial communication is available when receiving commands from the central controller.

3.2 Software

The code running on both platforms, ESP32 and Pico-Ice, has the same structure. In this section, a diagram of this code will be presented, and the overall interaction between the different software components will be represented. Additionally, relevant functions and code structures will be delineated and explained. Lastly, software optimizations applied to the base implementation are presented, which constitute one of the main contributions of this work, significantly improving execution performance across both platforms.

3.2.1 Software Diagram

The software diagram seen in fig. 3.5 in addition of illustrating the relations between the software running on both the central and local controller, also explains the interactions between the most relevant software functions. The dashed lines illustrate dependency, meaning that the arrow points from the dependent function to the function it relies on.

The central controller is represented by the dashed red rectangle. The system's user can interact with the Python script, running on a laptop, deciding which command to send to the microcontroller. The script then sends this message by either serial communication, possible on both the ESP32 and Pico-Ice, or via W-Fi, only viable on the ESP32. After sending the desired commands,

the information sent by the microcontroller to the central controller, via serial communication, can be displayed in a command terminal.

The dashed blue rectangle represents the main functions running on the local controller. After the microcontroller receives the commands sent by the laptop, the function *ProcessSerialCommand* interprets them and calls the appropriate functions in order to reach the desired outcome. The functions represented on both sides of the *ProcessSerialCommand* function are similar. The ones on the left side are used when the user wants to treat the four tiles as one. For example, if the system is composed of four tiles with an eight-by-eight unit cell arrangement, the code will treat them as if a sixteen-by-sixteen tile is being controlled. On the other hand, the right side illustrates the functions used to control one individual tile.

In the following section, the inner workings of these functions, along with other relevant components not shown in fig. 3.5, will be explained in detail.

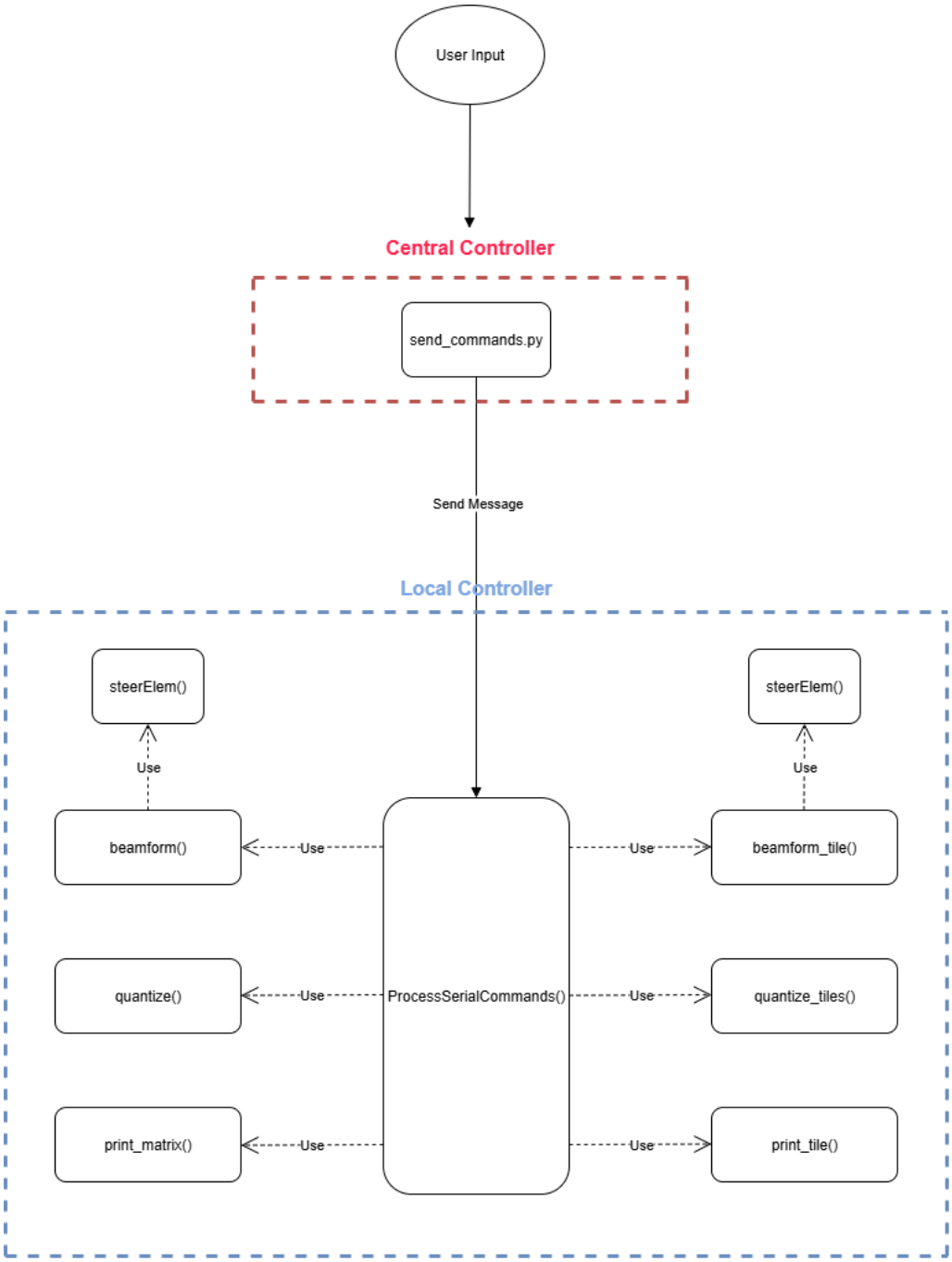


Figure 3.5: Software diagram

3.2.2 Central Controller

Python script ESP32 and Pico-Ice

As previously stated, the only component running on the central controller is a Python script, listing 3.1, that the user can interact with. This script can be used to communicate with both the ESP32 and Pico-Ice.

Firstly, a connection to the local controller is established via serial communication. If the connection is successful, a prompt asks the user which command to send to the microcontroller.

The supported commands are the following:

- beamform <full or tile number> <theta> <phi>
- print <full or tile number>

The first command requests that the microcontroller perform all the required beamforming calculations on an individual tile, numbered one through four, or on all four tiles simultaneously. As explained in the previous section, when controlling all four tiles at once, the system treats them as a single, larger tile. From this point on, this configuration will be referred to as full array mode or full tile mode. The second command instructs the local controller to display the current configuration state of a particular tile or of the full array mode.

Afterwards, the command is formatted so it can be sent to the local controller. Finally, the last loop is responsible for reading and displaying the responses from the microcontroller.

```
import serial
import time


SERIAL_PORT = '/dev/ttyACM1'
BAUDRATE = 115200

def main():
    ser = serial.Serial(SERIAL_PORT, BAUDRATE, timeout=2)
    time.sleep(2)

    try:
        while True:
            cmd = input("Enter command (e.g. 'beamform <full or tile number> <theta
                > <phi>' or 'print <tile number>): ")
            if not cmd.strip():
                continue
            else:
                ser.write((cmd.strip() + '\n').encode())
                start = time.time()
                while True:
                    line = ser.readline().decode(errors='ignore').strip()
                    if line:
                        print(line)
```

```
                        start = time.time()  # Reset timer on new data
                    elif time.time() - start > 0.3:
                        break
    except KeyboardInterrupt:
        print("Exiting.")
    finally:
        ser.close()

if __name__ == "__main__":
    main()
```

Listing 3.1: Python script running on the central controller

Earlier iteration of the python script

In an earlier iteration of the project, a different Python script, listing 3.1, was developed to communicate with the ESP32. Instead of using serial communication, the laptop running the script would connect to a network created by the microcontroller, and the relevant data would be transmitted wirelessly. At that stage, however, the software running on the ESP32 was more rudimentary, and it only supported a single tile of fixed size. As a result, the only data sent were the steering angles *theta* and *phi*.

Because it was later decided that only serial communication would be used, mainly because the Pico-Ice does not have a Wi-Fi module, Wi-Fi communication was abandoned entirely in the final version of the system. However, if necessary in the future, the script in listing 3.1 can be adapted to also support the new communication requirements.

```
import socket

esp32_ip = "192.168.4.1"
esp32_port = 1234

control = 0

def value_checker(value):
    if -90 <= value <= 90:
        return True
    else:
        return False

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

print("Type exit to quit")
print("-----------------")

while(True):
    if control == False:

```

```
22          message = input("Theta: ")
23          if message == "exit":
24              break
25
26          while(value_checker(float(message)) == False):
27              message = input("Invalid input for Phi. Please enter a value between
                    -90 and 90: ")
28
29          sock.sendto(str(message).encode(), (esp32_ip, esp32_port))
30          control = True
31
32      if control == True:
33
34          message = input("Phi: ")
35          if message == "exit":
36              break
37          while(value_checker(float(message)) == False):
38              message = input("Invalid input for Phi. Please enter a value between
                    -90 and 90: ")
39
40          sock.sendto(str(message).encode(), (esp32_ip, esp32_port))
41          print("-----------------")
42          control = False
43
44
45  sock.close()
```

Listing 3.2: Python script handling Wi-Fi communication

3.2.3 Local Controller

In this section, because the code running on both microcontrollers is very similar, only the relevant code implemented on the Pico-Ice will be presented to avoid unnecessary repetition.

Data Structures and Configuration

The structures used in the code implemented on the local controller are named *Tiles*, listing 3.3, and *Control*, listing 3.4. The first is designed to represent the physical RIS tiles. As previously explained, there are two distinct modes of operation; the tiles can be controlled either individually or in full array mode. Because these modes work independently, this code structure handles both cases separately.

The variables *size* and *tile_size* represent the number of unit cells along one side of the full tile and of an individual tile, respectively. The physical length is calculated by multiplying the number of unit cells by *CELLDIM*, which represents the physical length of one unit cell. When an instance of the *Tiles* structure is created, the constructor then allocates memory for all coordinates of each unit cell and the phase arrays for both modes. Because the beam steering calculations are

always made considering the coordinates referential is always centered on the tile that is being controlled, when attributing the coordinates to the individual tiles, the x and y axis offsets need to be considered. In this particular case, it is considered that the four tiles are arranged in a two-by-two configuration with no empty space in between them. If the configuration were to be different, the new offsets would need to be calculated.

```
struct Tiles{
    int size;
    int tile_size;
    double length;

    double *x_full;
    double *y_full;
    double **m_phase_deg_full;
    double *x_i[4];
    double *y_i[4];
    double **m_phase_deg[4];

    Tiles(int size) {
        this->size = size;
        this->tile_size = size / 2;
        length = CELLDIM * size;
        double tile_length = CELLDIM * tile_size;

        // Handle full array
        x_full = new double[size];
        y_full = new double[size];
        m_phase_deg_full = new double*[size];
        for (int i = 0; i < size; i++) {
            m_phase_deg_full[i] = new double[size];
        }
        for (int i = 0; i < size; i++) {
            double pos = (-length / 2 + CELLDIM / 2) + (i * CELLDIM);
            x_full[i] = pos;
            y_full[i] = pos;
        }

        double offsets[4][2] = {
            {-(double(tile_size)/2),  (double(tile_size)/2)},  // Top-left
            { (double(tile_size)/2),  (double(tile_size)/2)},  // Top-right
            {-(double(tile_size)/2), -(double(tile_size)/2)},  // Bottom-left
            { (double(tile_size)/2), -(double(tile_size)/2)}   // Bottom-right
        };

        // Handle each tile
        for (int t = 0; t < 4; t++){
            x_i[t] = new double[tile_size];
            y_i[t] = new double[tile_size];
            m_phase_deg[t] = new double*[tile_size];
```

```
            for (int i = 0; i < tile_size; i++) {
                m_phase_deg[t][i] = new double[tile_size];
            }

            for (int i = 0; i < tile_size; i++) {
                double pos = (-length / 2 + CELLDIM / 2) + (i * CELLDIM);
                x_i[t][i] = pos + offsets[t][0];
                y_i[t][i] = pos + offsets[t][1];
            }
        }
    }

    ~Tiles() {
        delete[] x_full;
        delete[] y_full;
        for (int i = 0; i < size; i++) {
            delete[] m_phase_deg_full[i];
        }
        delete[] m_phase_deg_full;
        for (int t = 0; t < 4; t++) {
            delete[] x_i[t];
            delete[] y_i[t];
            for (int i = 0; i < tile_size; i++) {
                delete[] m_phase_deg[t][i];
            }
            delete[] m_phase_deg[t];
        }
    }
};
```

Listing 3.3: Tiles struct

The *Control* structure is used to store all de relevant parameters needed to define the beam steering direction and reference point. The *theta_dir* and *phi_dir* represent the zenith and azimuth angle of the desired beam direction in degrees. Additionally, the variables *x_coord*, *y_coord*, and *z_coord* represent the coordinates of the feed reference point relative to the referential centered on a tile.

```
struct Control {

    double theta_dir;
    double phi_dir;
    double phase_const;
    double x_coord;
    double y_coord;
    double z_coord;

    Control() {
        theta_dir = 0;
        phi_dir = 0;
```

```
        phase_const = 0;
        x_coord = 200;
        y_coord = 0;
        z_coord = 550;
    }
};
```

Listing 3.4: Tiles struct

Relevant functions

The *loop* function, listing 3.5 is always scanning for new serial input. When a complete line is received, it is passed as input for the *processSerialCommand* function.

```
void loop() {
    static String input = "";
    while (Serial.available()){
        String input = Serial.readStringUntil('\n');
        processSerialCommand(input);
   }

}
```

Listing 3.5: Main program loop

The *processSerialCommand* function, listing 3.6, is responsible for processing the commands sent by the system's user. Firstly, it trims the input so that the whitespace and newline characters are removed. Then, the first if statement verifies if the command has any whitespaces. If there are not, then the command structure is invalid because all possible commands must start with a command word, "beamform" or "print", followed by at least one argument. After knowing for certain that the received command is valid, it is divided into two tokens. The variable *command* stores the command word, and *rest* stores the remaining arguments.

If the string that was stored in the *command* variable is "beamform" this means that the one stored in *rest* contains the mode followed by the two steering angles; therefore, this string is divided again into two new variables called *mode*, that dictates if the been steering calculations will be performed on an individual tile or on the full tile, and another called *params* which stores the *theta* or zenith angle followed by *phi*, which can also be referred as the azimuth angle.

The steering angles are extracted from the *params* string and are converted to floating-point numbers, then they are stored in the variables *theta* and *phi*. To allow the user to confirm that the correct angles were received by the local controller, the values of these variables are displayed. Afterwards, these values are verified to determine if they are valid numbers. This is important because when *toFloat()* fails, it returns zero. This would imply that a steering angle could be set to zero degrees even if the input string was invalid. Finally, after the steering angles on an instance of the *control* structure are updated with these new values, the string stored in the variable *mode* is examined. If it contains "full", the code computes the configuration of the full RIS tile. Otherwise,

it is assumed to contain the number of an individual tile, in which case, the string is converted to an integer so the code can compute the new state of the corresponding tile.

If *command* contains the string "print", the *rest* variable will contain either "full" or the number of an individual tile. In either case, the code simply displays the current state of the specified tile.

```
void processSerialCommand(const String& input) {
    String cmd = input;
    cmd.trim();
    if (cmd.length() == 0) return;

    // Split command into tokens
    int firstSpace = cmd.indexOf(' ');
    if (firstSpace == -1) {
        Serial.println("Invalid command.");
        return;
    }
    String command = cmd.substring(0, firstSpace);
    String rest = cmd.substring(firstSpace + 1);

    if (command == "beamform"){
        int nextSpace = rest.indexOf(' ');
        if (nextSpace == -1){
            Serial.println("Invalid beamform command.");
            return;
        }
        String mode = rest.substring(0, nextSpace);
        String params = rest.substring(nextSpace + 1);

        double theta, phi;

        // Manual parsing using Arduino String
        int spaceIdx = params.indexOf(' ');
        if (spaceIdx == -1){
            Serial.println("Invalid parameters. Usage: beamform <mode> <theta> <phi
                >");
            return;
        }
        String thetaStr = params.substring(0, spaceIdx);
        String phiStr = params.substring(spaceIdx + 1);

        theta = thetaStr.toFloat();
        phi = phiStr.toFloat();

        Serial.print("Parsed theta: "); Serial.println(theta, 6);
        Serial.print("Parsed phi: "); Serial.println(phi, 6);

        if ((theta == 0 && thetaStr != "0" && thetaStr != "0.0") || (phi == 0 &&
            phiStr != "0" && phiStr != "0.0")){
```

```
42              Serial.println("Invalid parameters. Usage: beamform <mode> <theta> <phi
                    >");
43              return;
44          }
45          ctrl.theta_dir = theta;
46          ctrl.phi_dir = phi;
47
48          if (mode == "full"){
49
50            beamform(tiles, ctrl);
51            quantize(tiles);
52            print_matrix(tiles);
53          }
54          else{
55              int tileNum = mode.toInt();
56              if (tileNum >= 1 && tileNum <= 4) {
57                  beamform_tile(tiles, tileNum - 1, ctrl);
58                  quantize_tile(tiles, tileNum - 1);
59                  print_tile(tiles, tileNum - 1);
60                  Serial.flush();
61              }
62              else{
63                  Serial.println("Unknown tile. Use 'full' or a tile number 1-4.");
64              }
65          }
66      }
67      else if (command == "print"){
68          rest.trim();
69          if (rest == "full"){
70              print_matrix(tiles);
71          } else {
72              int tileNum = rest.toInt();
73              if (tileNum >= 1 && tileNum <= 4) {
74                  print_tile(tiles, tileNum - 1);
75              } else {
76                  Serial.println("Unknown tile. Use 'full' or a tile number 1-4.");
77              }
78          }
79      }
80      else {
81          Serial.println("Unknown command.");
82      }
83  }
```

Listing 3.6: processSerialCommand function

The *processSerialCommand* calls several other functions in order to compute and display the RIS pattern according to the steering angles. Each of the following functions also has a similar version used for the full array mode, however, to avoid unnecessary repetition, only the functions

used in the individual tile mode will be presented.

The *beamform_tile* function is responsible for calculating the phase shift for every element in a determined tile. It passes as inputs the coordinates of every element and the control parameters to the *steerElem* function.

The next function, called *quantize_tile*, implements a two-level phase quantization scheme. Also known as binary phase quantization. This is necessary because the RIS elements can only represent two different states. They can either be on or off.

The function takes the calculated continuous phase values for a specific tile and attributes them to one of two possible discrete levels. The expression that determines both levels is shown below, where $\phi(i,j)$ represents the original phase value for the coordinates $x = i$ and $y = j$ and $\phi_q(i,j)$ is the quantized phase value.

$$\phi_q(i,j) = \begin{cases} 90^\circ, & \text{if } -90^\circ \leq \phi(i,j) < 90^\circ \\ 0^\circ, & \text{otherwise} \end{cases}$$

The last function called is *print_tile*. It has two distinct functionalities. In addition to printing the configuration of a tile, it is also responsible for sending the data that controls the RIS tile using SPI communication. The function decides if an element is going to be switched on or off based on the previous quantization levels. If the phase value of an element was set to $TresholdValue1$, the element will be switched on, otherwise, if the value was set to $TresholdValue2$, it will be switched off. A variable named *aux* registers the configuration of one row at a time and then stores it in the *data* array. When all tile rows are stored, the data is then transferred to the tile.

```
void print_tile(Tiles* tiles, int t) {
  uint8_t data[8] = {0};  // All LEDs off (64 bits set to 0)
  uint8_t aux = 0b00000000;
  resetLEDs();
    Serial.println("\nTile:");
    for (int i = 0; i < tiles->tile_size; i++) {
        for (int j = 0; j < tiles->tile_size; j++) {
            if (tiles->m_phase_deg[t][i][j] == 115.15) {
                Serial.print("1 ");
                aux = (aux >> 1) | 0b10000000;
            } else {
                Serial.print("0 ");
                aux = (aux >> 1);
            }
        }
        data[tiles->tile_size - i - 1] = aux; // Store the byte in the data array
        aux = 0b00000000;
        Serial.println();
    }
    // Send data to the PCB
    digitalWrite(LATCH_PIN, LOW);
```

```
    for (int i=0; i < tiles->tile_size; i++){
        SPI.transfer(data[i]); // Send one byte at a time
    }
    digitalWrite(LATCH_PIN, HIGH);
}
```

Listing 3.7: print_tile function

Element Phase Shift Calculation and Optimizations

As previously mentioned, the *steerElem* function, listing 3.8, is called by the *beamform_tile* function. It is essentially the core of the beamforming algorithm. It calculates the required phase shift of an element in the near-field region. After receiving an element's coordinates and control values, the phase shift of a specific element required to contribute coherently to the overall steering direction, which is dictated by both the azimuth (*phi*) and zenith (*theta*) angles, is calculated. This can be achieved by utilizing the expression below, where k is the angular wavenumber $k = \frac{2\pi}{\lambda}$, d is the Euclidean distance between the RIS element and the signal source, and pc is a :

$$\phi(i,j) = -k(d + sin\theta(xsin(\varphi) + ysin(\varphi))$$

```
double steerElem(double x, double y, Control &ctrl) {
    double R, m_phase;
    double xc = ctrl.x_coord;
    double yc = ctrl.y_coord;
    double zc = ctrl.z_coord;
    double th = ctrl.theta_dir;
    double ph = ctrl.phi_dir;
    double pc = ctrl.phase_const;


        R = sqrt((xc - x) * (xc - x) + (yc - y) * (yc - y) + zc * zc);
        m_phase = k * (R - sin(th * pi / 180) * (x * cos(ph * pi / 180) + y * sin(
            ph * pi / 180))) + pc;
        m_phase = fmod(m_phase, 2 * pi);
        m_phase = m_phase * 180 / pi - 180;

    return m_phase;
  }
```

Listing 3.8: steerElem function

This is the most resource-intensive section of the code, mainly because these calculations need to be done for all individual elements. If we are dealing with smaller tiles and if there is no need to change steering directions with a high frequency, this may not be a problem, however there is a need to scale the system, and all of these calculations can severely hinder the system's performance. Therefore, the following optimization techniques were explored:

- Trigonometry lookup tables
- Angle conversion constants

It is necessary to use trigonometric functions such as sine and cosine repeatedly in order to compute the desired RIS tile pattern. Since performing these calculations repeatedly for each individual element is time-consuming, it was decided to implement a lookup table containing the sine and cosine values for every angle between -180 and 180 degrees, with a step size of 1 degree. These tables are computed only once at program startup. Each table contains 181 entries and stores floating-point values. As a result, the total memory usage of the lookup tables is approximately 1448 bytes (2 tables × 181 entries × 4 bytes per float). In the *void_setup* function, *init_trig_tables*, listing 3.9, is called, which is responsible for creating these lookup tables.

```
void init_trig_tables(){
  for (int i=0; i < 181; i++){
    int angle = i - 90;
    cos_table[i] = cosf((angle * pi) / 180);
    sin_table[i] = sinf((angle * pi) / 180);
  }
}
```

Listing 3.9: init_trig_tables function

In the initial version of the *steerElem* function, the conversion constant from degree to radian and vice versa was computed every time a conversion was needed, which then led to unnecessary additional computation time. Therefore, these constants were declared as the global constants *deg*2*rad* and *rad*2*deg*, which saves execution time.

The results of these optimizations will be presented in the next chapter.

Chapter 4

Results

4.1 Software Functionality and Output Validation

To validate the correct functioning of the software, the base version of the steering algorithm, meaning the code with no optimizations, was tested on both microcontrollers. The result of three different combinations of steering angles will be shown on both individual tiles and on the full array. However, because the LED PCB has an eight by eight configuration, the board was only used to display the resulting configuration on an individual tile.

Firstly, the script on the central controller is initialized, fig. 4.1. The desired steering angles are sent to the controller. .

```
fmesquita@fmesquita:~/Documents/ris-control/ris-control/ris-control-fmesquita/beamforming with communication$ python3 send_commands.py
Enter command (e.g. 'beamform <full or tile number> <theta> <phi>' or 'print <tile number>): beamform full -90 90
Parsed theta: -90.000000
Parsed phi: 90.000000
Reflectarray Phase Matrix:
0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1
1 0 1 0 0 1 0 1 0 1 0 1 0 0 1 0
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0
0 1 0 0 1 0 1 0 1 0 1 0 0 1 0 1
0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1
1 0 1 0 0 1 0 1 0 1 0 0 0 0 1 0
1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0
0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0
0 1 0 0 1 0 1 0 1 0 1 0 0 1 0 1
0 1 0 0 1 0 1 0 1 0 1 0 1 0 0 1
0 0 1 0 1 0 0 0 0 0 1 0 1 0 1 0
0 0 1 0 0 0 0 1 0 1 0 0 1 0 1 0
1 0 1 0 0 1 0 1 0 1 0 1 0 0 1 0
1 0 0 1 0 1 0 1 0 1 0 1 0 0 1 0
1 0 0 1 0 1 0 1 0 1 0 1 0 0 1 0
1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0
Enter command (e.g. 'beamform <full or tile number> <theta> <phi>' or 'print <tile number>):
```

Figure 4.1: Python script sending a command to configure the full tile

In the terminal, the microcontroller confirms that the steering angles were received, and the resulting RIS matrix is displayed in the terminal.

If it is required to control an individual tile, this can be achieved by sending the tile number, one through four, counted clockwise, instead of "full", fig. 4.2, fig. 4.3. These two examples are shown to demonstrate the independent control capability of individual tiles, which is essential for modular beamforming behavior.

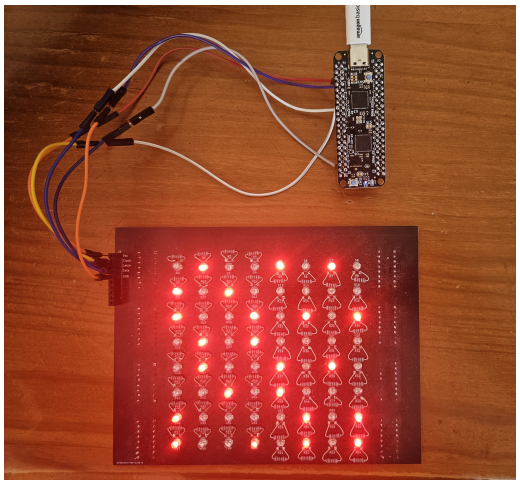
```
Enter command (e.g. 'beamform <full or tile number> <theta> <phi>' or 'print <tile number>): beamform 1 -90 90
Parsed theta: -90.000000
Parsed phi: 90.000000
Tile:
0 1 0 0 1 0 1 0
1 0 1 0 0 0 0 0
1 0 0 1 0 1 0 1
0 1 0 1 0 0 0 0
0 1 0 0 1 0 1 0
0 0 1 0 1 0 0 0
1 0 0 0 0 1 0 1
1 0 0 1 0 1 0 1
```

Figure 4.2: Python script sending a command to configure the first tile

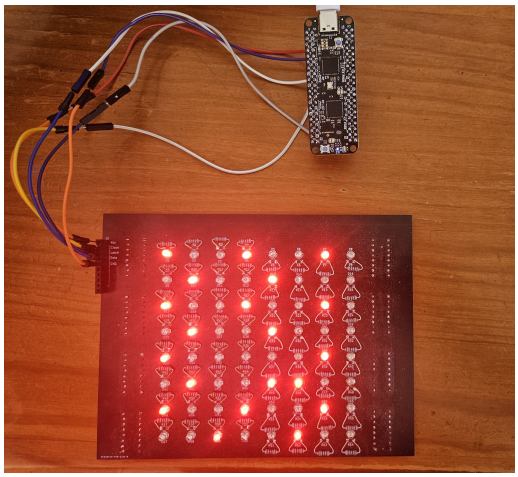
```
Enter command (e.g. 'beamform <full or tile number> <theta> <phi>' or 'print <tile number>): beamform 3 45 45
Parsed theta: 45.000000
Parsed phi: 45.000000
Tile:
1 0 0 1 0 0 1 0
0 1 0 0 1 0 0 0
1 0 0 1 0 0 1 0
0 1 0 0 1 0 0 0
1 0 0 1 0 0 1 0
0 1 0 0 1 1 0 0
1 0 0 1 0 0 1 0
0 0 1 0 0 1 0 0
```

Figure 4.3: Python script sending a command to configure the third tile

In this situation, the configuration can also be shown in the LED PCB as seen on fig. 4.4a, for the tile number one, and on fig. 4.4b for tile number three.



(a) Tile one configuration



(b) Tile three configuration

Figure 4.4: LED PCB representation of the control of individual tiles

If at any point the user requires to know the current configuration of a particular tile, this can be achieved by using the "print" command. For example, after the previous sequence of commands, the configuration of the full tile can be visualized by executing the command in fig. 4.5.

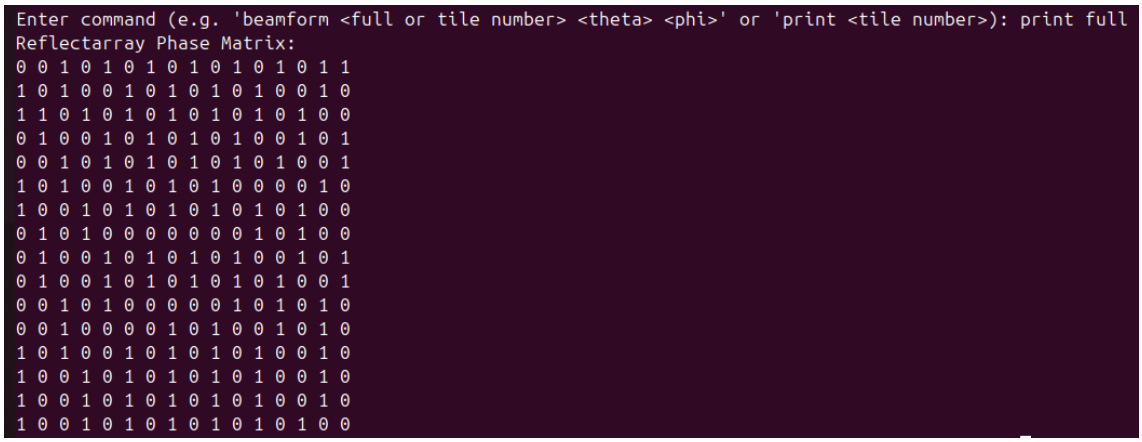


Figure 4.5: Python script sending a command to display the saved configuration of the full tile

The state of the individual tiles can also be shown, fig. 4.6. If the command "beamform" was not yet applied to a particular tile, for example, in this case, tile number two, the terminal will show all the unit cells in the OFF state, fig. 4.7.

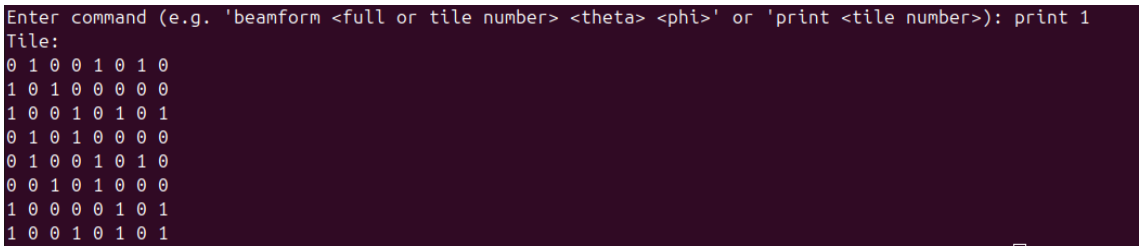


Figure 4.6: Python script sending a command to display the saved configuration of the first tile

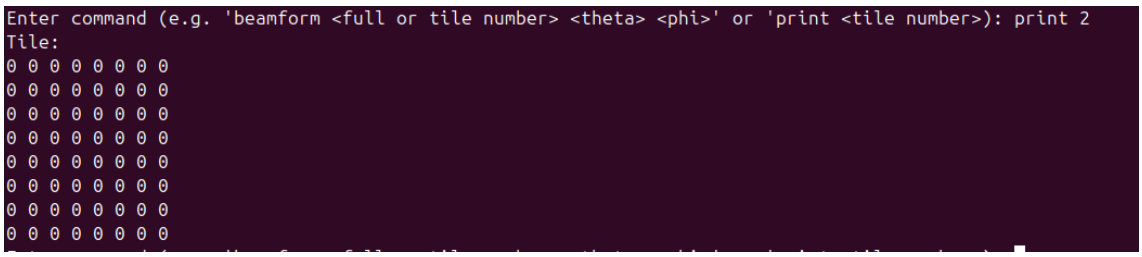


Figure 4.7: Python script sending a command to display the saved configuration of the second tile

These results confirm that the steering values are correctly computed, transmitted, and displayed across different tile configurations using the base implementation. In the next section, the performance of the optimized versions of the software and their impact on execution time will be analyzed.

4.2 Performance Evaluation and Optimized Versions

In this section, the performance of the software running on both microcontrollers is evaluated. The analysis begins by assessing the computation time of the beam steering calculations on the base version of the software, considering different array sizes both in the full array mode and on the individual tiles. Subsequently, the computation time of the optimized versions is measured under the same conditions for comparison.

4.2.1 Base Version Performance

To evaluate how scalability affects performance, three computation times were measured for each steering angle combination and tile size in both modes. The computation time in the following tables represents the average between all the computation time measurements obtained per tile size.

Table 4.1 and table 4.2 represent the computation times when running the software on the Pico-Ice.

Steering angles (°)	**8 × 8**	**16 × 16**	**32 × 32**
$\theta = -90, \phi = 90$	8.193	31.178	126.408
$\theta = -45, \phi = 45$	14.729	57.763	229.427
$\theta = 0, \phi = 0$	4.906	18.950	75.335
Average time	9.276	35.964	143.723

Table 4.1: Pico-Ice computation times (ms) for the Full Array (2 × 2 tiles)

Steering angles (°)	**4 × 4**	**8 × 8**	**16 × 16**
$\theta = -90, \phi = 90$	2.345	8.495	33.603
$\theta = -45, \phi = 45$	3.892	14.743	58.304
$\theta = 0, \phi = 0$	1.391	4.896	19.444
Average time	2.543	9.378	37.117

Table 4.2: Pico-Ice computation times (ms) for an individual tile

Table 4.3 and table 4.4 represent the computation times when running the software on the ESP32.

Steering angles (°)	**8 × 8**	**16 × 16**	**32 × 32**
$\theta = -90, \phi = 90$	1.735	6.612	26.172
$\theta = -45, \phi = 45$	2.381	9.224	36.599
$\theta = 0, \phi = 0$	0.977	3.699	14.454
Average time	1.698	6.512	25.742

Table 4.3: ESP32 computation times (ms) for the Full Array (2 × 2 tiles)

Steering angles (°)	**4 × 4**	**8 × 8**	**16 × 16**
$\theta = -90, \phi = 90$	0.506	1.737	6.623
$\theta = -45, \phi = 45$	0.67	2.375	9.23
$\theta = 0, \phi = 0$	0.324	0.988	3.676
Average time	0.5	1.7	6.510

Table 4.4: ESP32 computation times (ms) for an individual tile

By analyzing the previous tables, table 4.1 and table 4.3, it is evident that the ESP32 outperforms the pico-Ice significantly when running the base version of the beam steering algorithm. This was to be expected because the processing power of the ESP32 is higher than that of the Pico-Ice. This difference in computation time becomes more evident as the array size increases. This performance difference is also shown to follow the same trend when computing the tile configuration for an individual tile, as shown on table 4.2 and table 4.4.

The following graphs, presented in fig. 4.8 and fig. 4.9, help visualize how the computation time increases with the array size. Additionally, they also compare the speed of both microcontrollers when executing the algorithm.

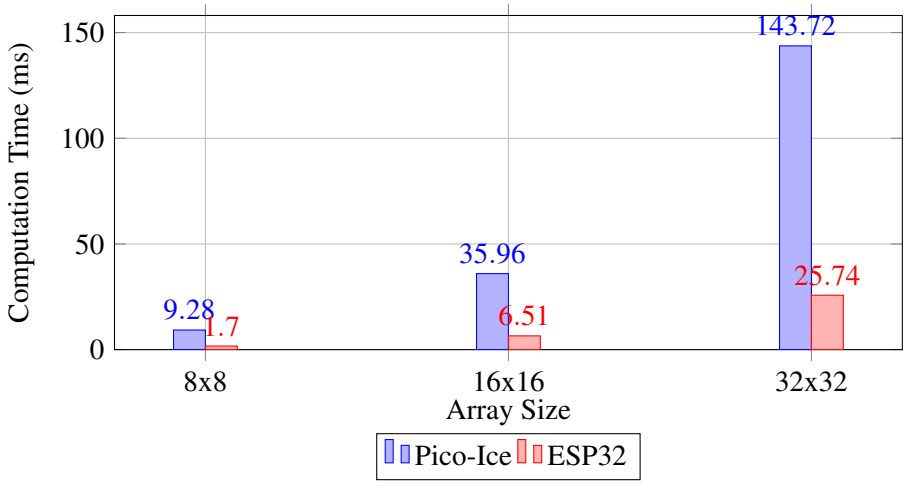


Figure 4.8: Average computation time comparison for full tile mode

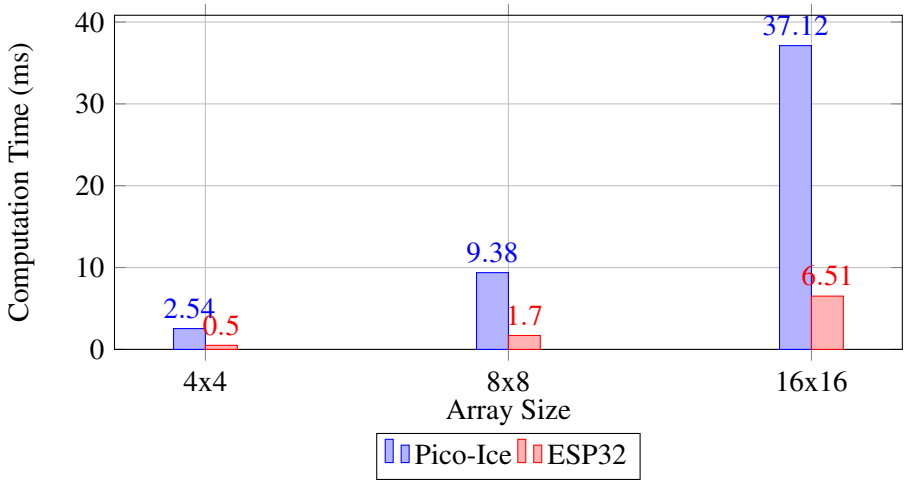


Figure 4.9: Average computation time comparison for individual tile mode

By examining the graphs, it is also possible to determine that the ESP32 performs the task in approximately 81.74% less time than the Pico-Ice, as shown by the calculations below.

The *speedup* can be calculated by dividing the computation time of the ESP32 by the computation time of the Pico-Ice:

$$\text{Speedup} = \frac{\Delta t_{\text{Pico-Ice}}}{\Delta t_{ESP32}} \tag{4.1}$$

To calculate the overall average speedup, the average of all the data points represented in the graphs represented in fig. 4.5 and fig. 4.9 is calculated.

$$\text{Average Speedup} = \frac{5.46+5.52+5.58+5.08+5.52+5.70}{6} \approx \boxed{5.477\times} \tag{4.2}$$

Now it is possible to calculate the overall improvement of using the ESP32 as opposed to the Pico-Ice:

$$\text{Improvement} = (1 - \frac{1}{AverageSpeedup}) * 100 \approx \boxed{81.74\%} \tag{4.3}$$

4.2.2 Optimized Versions

As previously mentioned in chapter 3, two optimizations were made to the *steerElem* function. Because it is the core of the beamforming algorithm, and it is called an increasing number of times as the size of a tile grows, it is important to decrease the computation time as much as possible. The implemented optimizations were the following:

- **Trigonometric Lookup Tables:**
  Instead of calculating the values of $\sin(\theta)$, $\cos(\phi)$ and $\sin(\phi)$ a lookup table was introduces. This table stores values of $\sin(x)$ and $\cos(x)$, with being an angle, in degrees, between -180 and 180 with a step size of 1°. Because it is only computed once upon the initialization of the program, it significantly reduces the computation time.

- **Angle Conversion Constants:**
  In the original implementation, θ and ϕ were converted to radians inside the *steerElem* function. This conversion is necessary because the trigonometric functions in the C++ math library expect their arguments in radians. Then, when all the values are calculated, the phase shift value is again converted into degrees. Because of this, the conversion constants $(\pi/180)$ and $(180/\pi)$ were calculated an unnecessary number of times, which increases the computation time. This is aggravated when the size of the array increases. To solve this, these constants were declared as global constants, which slightly decreases the computation time.

The computation times for the optimized version are presented in the following tables and graphs, comparing their performance against the baseline implementation on both the ESP32 and Pico-Ice microcontrollers.

Steering angles (°)	**8 × 8**	**16 × 16**	**32 × 32**
$\theta = -90, \phi = 90$	3.297	12.551	49.395
$\theta = -45, \phi = 45$	3.261	12.432	48.985
$\theta = 0, \phi = 0$	3.256	11.216	44.084
Average time	3.271	12.066	47.488

Table 4.5: Pico-Ice optimized computation times (ms) for the Full Array (2 × 2 tiles)

Steering angles (°)	**4 × 4**	**8 × 8**	**16 × 16**
$\theta = -90, \phi = 90$	0.952	3.311	12.731
$\theta = -45, \phi = 45$	0.909	3.197	12.370
$\theta = 0, \phi = 0$	0.843	2.942	11.204
Average time	0.901	3.147	12.102

Table 4.6: Pico-Ice computation times (ms) for an individual tile

Steering angles (°)	**8 × 8**	**16 × 16**	**32 × 32**
$\theta = -90, \phi = 90$	0.72	2.665	10.509
$\theta = -45, \phi = 45$	0.703	2.663	10.506
$\theta = 0, \phi = 0$	0.707	2.655	10.485
Average time	0.71	2.661	10.5

Table 4.7: ESP32 optimized computation times (ms) for the Full Array (2 × 2 tiles)

Steering angles (°)	**4 × 4**	**8 × 8**	**16 × 16**
$\theta = -90, \phi = 90$	0.226	0.708	2.677
$\theta = -45, \phi = 45$	0.224	0.703	2.681
$\theta = 0, \phi = 0$	0.226	0.697	2.658
Average time	0.225	0.703	2.672

Table 4.8: ESP32 optimized computation times (ms) for an individual tile

By analyzing these tables, it is clear that a significant improvement in performance was achieved by implementing the aforementioned optimizations. The graphs shown in fig. 4.10 and fig. 4.11 illustrate this improvement in performance when using the EPS32.

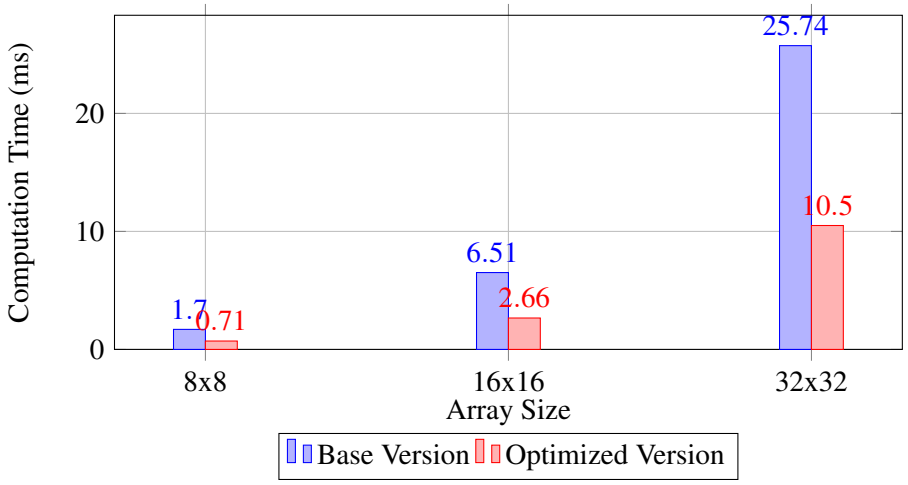


Figure 4.10: ESP32 Average computation time comparison for full tile mode

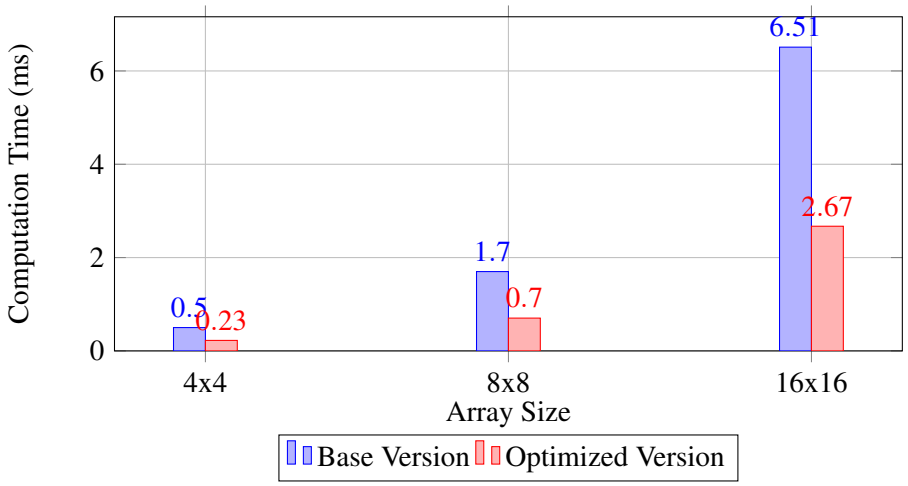


Figure 4.11: ESP32 Average computation time comparison for individual tile mode

The average speedup and the improvement in performance for the ESP32 are calculated below. These modifications resulted in a 58.12% reduction in computation time

$$\text{Speedup} = \frac{\Delta t_{\text{Base}}}{\Delta t_{optimized}} \tag{4.4}$$

$$\text{Average Speedup} = \frac{2.39+2.45+2.45+2.17+2.43+2.44}{6} \approx \boxed{2.388\times} \tag{4.5}$$

$$\text{Improvement} = (1 - \frac{1}{AverageSpeedup}) * 100 \approx \boxed{58.12\%} \tag{4.6}$$

Regarding the performance improvement on the Pico-Ice with the same optimizations, the result was similar to the ESP32. The graphs presented in fig. 4.12 and fig. 4.13 show this improvement.

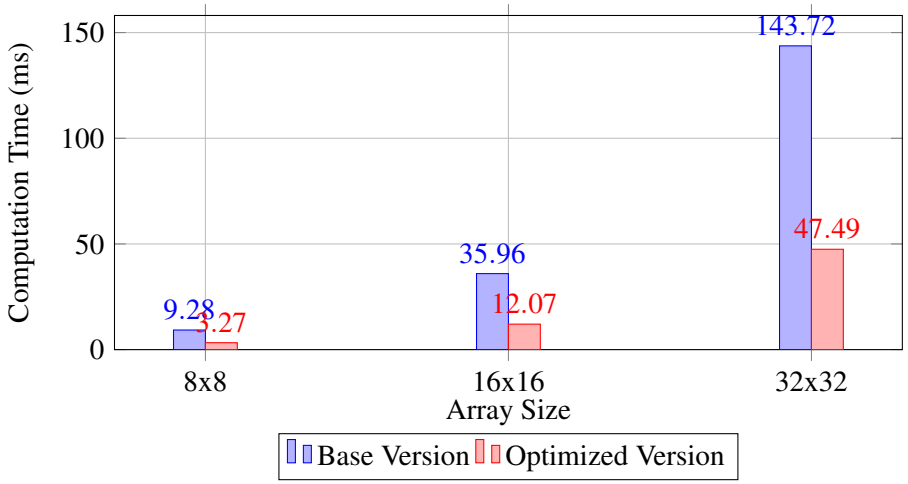


Figure 4.12: Pico-Ice Average computation time comparison for full tile mode

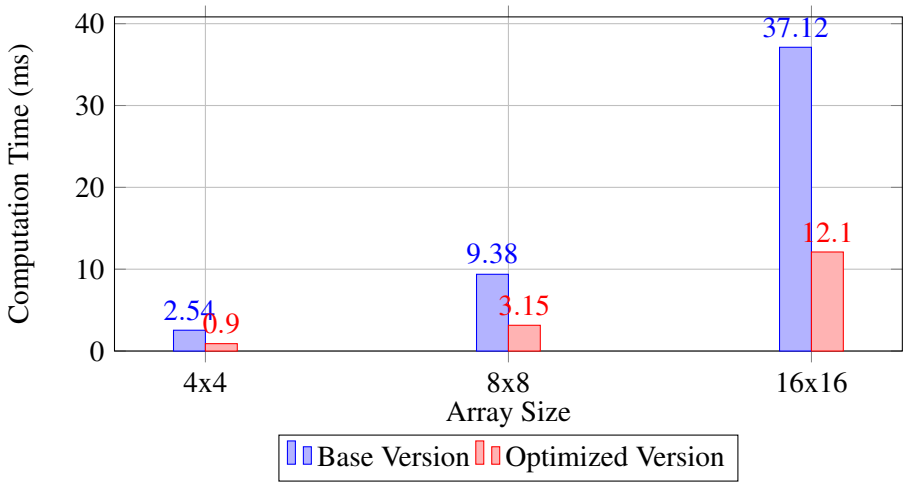


Figure 4.13: Pico-Ice Average computation time comparison for individual tile mode

Once again, the average speedup and performance improvement were calculated.

$$\text{Speedup} = \frac{\Delta t_{\text{Base}}}{\Delta t_{optimized}} \tag{4.7}$$

$$\text{Average Speedup} = \frac{2.84 + 2.98 + 3.026 + 2.82 + 2.978 + 3.07}{6} \approx \boxed{2.95\times} \tag{4.8}$$

$$\text{Improvement} = (1 - \frac{1}{AverageSpeedup}) * 100 \approx \boxed{66.56\%} \tag{4.9}$$

In this case, the performance improvement was even greater than on the ESP32, resulting in a reduction of 66.10% in computation time.

In conclusion, both optimization techniques resulted in a significant improvement in performance on both platforms. While the EPS32 remains the faster microcontroller overall after the optimizations, the Pico-Ice showed a greater relative improvement on average. These results demonstrate the importance of code optimization when developing a scalable RIS control application on embedded platforms.

Chapter 5

Conclusions

5.1 Summary and Contributions

This dissertation presented the design, implementation, and evaluation of an embedded control system for RISs. The work addressed the practical challenges of enabling real-time RIS configuration using low-power and cost-effective microcontroller platforms, specifically the ESP32 and the Pico-Ice.

A layered control system was developed, composed of a central controller implemented as a Python script, running on a laptop, and a local controller running on the microcontroller. The system supported two modes of operation: control of individual RIS tiles and full-array steering. An LED PCB was used to emulate the tile behavior and validate system functionality.

To enable beam steering capabilities, near-field phase shift calculations were implemented based on well-established mathematical models. These calculations were then optimized to improve execution speed. The applied optimizations, such as the introduction of trigonometric lookup tables and the use of precomputed angle conversion constants, significantly reduced computation times, especially for larger array sizes.

A comparative performance analysis showed that the ESP32 consistently outperformed the Pico-Ice in terms of computation speed. Nevertheless, the Pico-Ice remains a viable platform due to its onboard FPGA, which can be used for future hardware-level acceleration. Overall, the results confirm that embedded microcontroller platforms are a practical and scalable option for RIS control.

5.2 Future Work

While the implemented system demonstrates the feasibility of RIS control via microcontrollers, several improvements are still possible. In addition to the introduced optimizations, the possibility of implementing beamforming codebooks that store precomputed calculations could be explored. Additionally, the outsourcing of beam steering calculations to the FPGA of the Pico-Ice could be investigated. Implementing the main calculations of the algorithm on hardware could contribute

to a drastic reduction in computation time. Furthermore, the system could be tested with real RIS hardware, and for a better user experience, a graphical user interface could be developed. This interface would be implemented in the central controller, replacing the Python script.

References

[1] Marcos Baena-Molina et al. “1-bit RIS Unit Cell with Mechanical Reconfiguration at 28 GHz”. In: *18th European Conference on Antennas and Propagation, EuCAP 2024* (2024). DOI: 10.23919/EUCAP60739.2024.10500936.

[2] Rui Chen et al. “Reconfigurable Intelligent Surfaces for 6G IoT Wireless Positioning: A Contemporary Survey”. In: *IEEE Internet of Things Journal* 9 (23 Dec. 2022), pp. 23570–23582. ISSN: 23274662. DOI: 10.1109/JIOT.2022.3203890.

[3] Jean-Baptiste Gros et al. “A Reconfigurable Intelligent Surface at mmWave based on a binary phase tunable metasurface”. In: (Apr. 2021). DOI: 10.1109/OJCOMS.2021.3076271. URL: http://arxiv.org/abs/2104.13291%20http://dx.doi.org/10.1109/OJCOMS.2021.3076271.

[4] Markus Heinrichs, Florian Pirlet, and Rainer Kronberger. “Measurement System and Methodology for RIS-Evaluation”. In: *2023 IEEE International Symposium on Antennas and Propagation, ISAP 2023* (2023). DOI: 10.1109/ISAP57493.2023.10388696.

[5] Markus Heinrichs, Aydin Sezgin, and Rainer Kronberger. “Open Source Reconfigurable Intelligent Surface for the Frequency Range of 5 GHz WiFi”. In: *2023 IEEE International Symposium on Antennas and Propagation, ISAP 2023* (2023). DOI: 10.1109/ISAP57493.2023.10389095.

[6] Yuanwei Liu et al. “Reconfigurable Intelligent Surfaces: Principles and Opportunities”. In: *IEEE Communications Surveys and Tutorials* 23 (3 July 2021), pp. 1546–1577. ISSN: 1553877X. DOI: 10.1109/COMST.2021.3077737.

[7] Mirhamed Mirmozafari et al. “Mechanically reconfigurable, beam-scanning reflectarray and transmitarray antennas: A review”. In: *Applied Sciences (Switzerland)* 11 (15 Aug. 2021). ISSN: 20763417. DOI: 10.3390/APP11156890.

[8] Kashif Nisar Paracha et al. “Liquid Metal Antennas: Materials, Fabrication and Applications”. In: *Sensors 2020, Vol. 20, Page 177* 20 (1 Dec. 2019), p. 177. ISSN: 1424-8220. DOI: 10.3390/S20010177. URL: https://www.mdpi.com/1424-8220/20/1/177/htm%20https://www.mdpi.com/1424-8220/20/1/177.

[9] Naser Ojaroudi Parchin et al. “Reconfigurable Antennas: Switching Techniques—A Survey”. In: *Electronics 2020, Vol. 9, Page 336* 9 (2 Feb. 2020), p. 336. ISSN: 2079-9292. DOI: 10.3390/ELECTRONICS9020336. URL: https://www.mdpi.com/2079-9292/9/2/336/htm%20https://www.mdpi.com/2079-9292/9/2/336.

[10] James Rains et al. “Fully-Addressable Varactor-Based Reflecting Metasurface with Dual-Linear Polarisation for Low Power Reconfigurable Intelligent Surfaces”. In: *17th European Conference on Antennas and Propagation, EuCAP 2023* (2023). DOI: 10.23919/EUCAP57121.2023.10133408.

[11] Muhammad Usman et al. “Intelligent wireless walls for contactless in-home monitoring”. In: *Light: Science & Applications* 11.1 (2022). ISSN: 2047-7538. DOI: 10.1038/s41377-022-00906-5.

[12] Xintu Zeng et al. “High-Accuracy Reconfigurable Intelligent Surface Using Independently Controllable Methods”. In: *IEEE International Workshop on Electromagnetics: Applications and Student Innovation Competition, iWEM 2021 - Proceedings* (2021). DOI: 10.1109/IWEM53379.2021.9790555.

[13] Shou Hui Zheng, Xiong Ying Liu, and Manos M. Tentzeris. “A novel optically controlled reconfigurable antenna for cognitive radio systems”. In: *IEEE Antennas and Propagation Society, AP-S International Symposium (Digest)* (Sept. 2014), pp. 1246–1247. ISSN: 15223965. DOI: 10.1109/APS.2014.6904950. URL: https://www.researchgate.net/publication/283610297_A_novel_optically_controlled_reconfigurable_antenna_for_cognitive_radio_systems.