Theater Actor Tracking Automation System

A Design Project Report
Presented to the School of Electrical and Computer Engineering of Cornell University
in Fulfillment of the Requirements for the Degree of
Master of Engineering, Electrical and Computer Engineering

Submitted By

Names: Rachel Yan (sy625) and Anne Liu (ayl47)
M.Eng. Field Advisor: Professor V. Hunter Adams
Degree Date: December 2025

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Theater Actor Tracking Automation System

Author: Anne Liu, Rachel Yan

Abstract:

Manual actor illumination remains a fundamental challenge in small and mid-sized theater productions. Operating follow spotlights is physically demanding, prone to human error, and requires significant rehearsal and coordination between performers and lighting operators. While automated tracking and follow-spot systems exist, their high cost, complexity, and infrastructure requirements make them impractical for educational theaters and resource-constrained venues. Other relatively more accessible solutions have low automation, and could generate very obvious and constant errors when the actors shift from the planned location. This project addresses this issue, where we aim to develop a system to help the small production theaters with their spotlight system by developing a low-cost, portable, and real-time actor tracking system designed to improve spotlight accuracy while remaining accessible and easy to deploy. Our implementation utilizes modulated IR sensors for position tracking on stage, and using WiFi communication across to backstage for a computer/Raspberry Pi based control system for position indication and the actual follow light control automation.

Individual Contribution

Anne Liu mainly worked on the localization of the IR sensors. This included the research on localization of tracking a target using IR wave, and building the IR localization system. Originally the entire system, both modulated and unmodulated, were both on breadboards and tested physically with the analyzers and machines provided in the lab. Anne also worked on the modulated programming of the IR receivers and transmitters, from the hardware assembly to the software configuration of the NEC protocol. This was later integrated together with the MQTT protocol that Rachel uses for the wireless communication between the localization to the base station.

The breadboards were then upgraded to PCBs (Printed Circuit Boards), which provided flexibility and portability. Anne did the foundation of the designing the boards, while Rachel, who has more experience in designing PCBs, mentored and approved of the board before fabrication. The assembly of the boards was also done by Anne, but the testing and debugging of the system was accomplished by both teammates (Anne and Rachel).

Rachel Yan mainly worked on researching whether using computer vision tracking algorithm with an infrared camera is a feasible solution to the problem during the first semester. Several different types of algorithms are implemented and tested, which in comparison did not receive a good result with distance and darkness. After the camera plan was abandoned, Rachel mainly worked on the communication between the Raspberry Pi and RP2040 using MQTT, as well as translating the message of from the RP2040 to the coordinates of the transmitter on stage, and created the expandable GUI representation that maps to coordinates to it.

The full integration and testing of the system, as well as the design of the poster and this report, was done by both teammates.

Executive Summary

The objective of this project was to design and implement a wireless actor tracking platform capable of identifying an actor's position on stage in real time and presenting that information in an intuitive, operator-friendly format. The system uses a wearable infrared (IR) transmitter attached to the actor and a configurable grid of IR receivers placed around the stage. Received signals are processed by a Raspberry Pi backend and transmitted over WiFi using the MQTT publish–subscribe protocol. A graphical user interface (GUI) renders a stage-mapped sensor grid and continuously updates the actor's estimated position, enabling rapid feedback for spotlight operation or future automation.

Custom printed circuit boards were designed for both the IR transmitter and receiver nodes using low-cost, off-the-shelf components. Design priorities included portability, low power consumption, scalability, and ease of integration into different stage layouts. The system operates reliably using standard WiFi infrastructure and requires minimal setup, making it suitable for classroom demonstrations, student productions, and small performance spaces.

Experimental testing demonstrated consistent real-time tracking performance over indoor distances exceeding 60 ft, with low communication bandwidth requirements even under frequent position updates. The system performed robustly in typical theater lighting conditions and proved compatible with university WiFi networks. The complete hardware and software platform was implemented at a total cost of approximately $50, representing a significant cost reduction compared to commercial alternatives.

Overall, this project demonstrates that effective, real-time actor tracking can be achieved using inexpensive sensing hardware and lightweight wireless communication protocols. The system provides a practical foundation for future enhancements, including motorized follow-spot control, multi-actor tracking, and integration with additional sensing modalities such as camera-based tracking. By lowering the financial and technical barriers to automated stage illumination, this work expands access to modern theatrical technology for educational and small-scale performance environments.

Introduction/Overview and Background

In live stage productions, spotlights are traditionally operated to highlight the actors and performers as they move across the stage. However, this task can be physically demanding, prone to human error, and inefficient during complex situations involving fast or unpredictable movements of the performer. In modern theatrical environments where precision, consistency, and automation are becoming more and more important, this presents a compelling opportunity for an automation system.

As the experience of our own team member during performance, the current system of following spotlights has various shortcomings. First, if the follow spot light requires the operator to manually follow the actor, the task could be very physically and mentally demanding, as the actor or performer could have been standing in different places during different performances. Second, sometimes there is certain automation of the spotlight, but that requires the performers to be in fixed spots, which limits the performance and also, if the actors are slightly off the spot, it could become a stage accident where the actor will be in the dark. Therefore, seeking methods to solve the above shortcomings is valuable. This project proposes an automated actor tracking system that replaces manual spotlight operation with a self-adjusting mechanism capable of dynamically following a performer in real time. By equipping the actor with a lightweight infrared-emitting device and installing infrared-sensitive sensors and cameras, we aim to track position data and translate it into a motorized control system of a spotlight fixture.

The situation is that traditional manual spotlighting is labor-intensive and not always responsive enough for live performances. The complication is that existing automated solutions are often prohibitively expensive or not well suited for theater productions. Our proposed resolution leverages affordable components like the Raspberry Pi and NoIR camera modules to design a low-cost, flexible system that can be deployed in educational or community theaters.

Through this project, we aim not only to solve a specific technical problem, but also to explore how engineering can meaningfully contribute to creative fields such as theater. As MENG students in ECE, we bring together a strong background in embedded systems, signal processing, sensor integration, and real-time control—all essential for implementing an automated tracking system with reliable performance. Our background in embedded control systems and computer vision helps equip us to design and optimize systems that respond to dynamic environments, such as live performances, with the precision and responsiveness required for performance-critical applications.

In addition to technical skills, we are approaching this project with sensitivity to the practical and artistic matters of live theater. Thus, our solution must be robust, unobtrusive, and adaptable to varied lighting conditions and stage layouts. By combining engineering discipline with a user-centered design mindset, we can bridge the gap between traditional stagecraft and automation technologies.

This project represents an intersection of two distinct communities of practice: live performance and embedded control systems. As members of the latter, we are committed to contributing solutions that not only automate but also elevate the capabilities of stage production. Our work reflects the broader potential for engineers to empower artists and performers with tools that expand creative freedom, reduce technical overhead, and support seamless, immersive experiences for audiences.

Problem Statement and Issues To Be Addressed

In live stage productions, the constant movement of performers on stage poses a significant challenge for traditional spotlighting techniques. Human operators would often have to track and illuminate performers in real-time, is labor intensive, susceptible to human error, and lacks the precision needed for complex or fast-paced scenes. While some automated spotlight systems exist, they are usually expensive, bulky and not well-suited for a diverse environment of educational or community theaters. Thus, our project seeks to address these challenges by designing a low-cost, real-time actor tracking system that combines IR sensor arrays with computer vision algorithms to create an automated solution that reliably tracks a single performer's position on stage and translates that data into control signals for a motorized spotlight.

Some issues to be addressed would include numerous testing for accuracy and responsiveness as the system must track the actor's position with minimal latency and high spatial accuracy to ensure the spotlight consistently follows the performer. We would also have to consider the extent of our Raspberry Pi computation as the controller's memory and chip handling has a hard maximum in our computation speed and accuracy. By addressing these challenges, the project aims to create an accessible automation tool that enhances performance quality, safety, and creative freedom in theatrical environments.

As a summary of this section, our main design specifications are:

- Independent + portable
- Flexible
- Scalable
- Affordable
- Automated
- Accurate
- Detectable in Darkness
- Durable for long distance
- "Invisible" for the audience

Research of Solutions

To develop the actor tracking system, a wide range of solution has been proposed. For example, long distance tracking could include GPS monitoring system, we could use Camera CV object detection system. We can use thermal system of people wearing a cool/hot emitting device. For short distance tracking, there could be April Tag and overlooking camera, RFID/NFC Tag tracking on ground, we could use LiDAR + ToF sensor tracking, or IMU based tracking system.

The method that we choose is mainly the camera with CV and the IR.

Among that we did research on camera ….

The initial thought of using the camera is because camera would be able to actually figures out where the actor is on stage and tells the spotlight where to point. We researched for a few different camera and decided using the the Raspberry Pi NoIR camera is the best choice. It is sensitive to infrared light because it doesn't have the default IR filter, and it is also not expensive. This is useful since we want to track the actor in dark stage environments using a small IR-emitting marker they wear. The camera can pick up this IR light even when stage lighting is minimal, and it helps us avoid using visible markers that would distract the audience.

To process the video feed, we're using OpenCV on the Raspberry Pi. We plan to run real-time image processing by grabbing frames from the camera and doing a few steps on each one. First we convert the image to grayscale and apply Gaussian blur to remove background noise or small light artifacts that are not part of the marker. Then we'll apply thresholding to isolate the IR marker more clearly. In some cases, we might need to filter by intensity or use morphological operations like erosion and dilation to remove false positives caused by reflections or ambient interference.

After we isolate the bright areas in the frame, we're planning to implement clustering logic so the system can identify which group of pixels is the actual marker, especially if there are multiple light sources or reflections in the background. This step will help eliminate tracking errors when multiple blobs appear in the image. If more than one cluster is detected, we will choose the one closest to the last known position of the actor, which makes the tracking smoother. We are also planning to implement motion prediction using a Kalman filter, which will help us estimate where the actor is likely to move next. This is especially helpful when the marker is occluded or disappears temporarily, like when the actor turns around or walks behind a prop.

Once the actor's position is tracked in the image, we'll map that to motor commands for our pan-tilt spotlight. This will require some calibration since the camera and spotlight won't be mounted in the exact same spot, so the coordinate mapping has to account for that offset. To make the movement smooth, we'll filter the position data over time and possibly apply interpolation to reduce jitter. The tracking and motor update needs to happen fast, ideally within 100 ms or less, to make sure the spotlight is responsive and doesn't lag behind the actor.

Our approach with the vision system will incorporate noise filtering, marker clustering, and motion prediction algorithms to ensure robust tracking despite occlusion, ambient interference, or actor movement.

There are a few risks we are aware of. One is that the Raspberry Pi might not be fast enough to handle all of this processing at a decent frame rate. Another issue is ambient interference—there might be IR light from stage lights that confuse the system. Occlusion is another big problem, since in theater the actor might move behind something or get blocked from the camera's view. A few solutions of this problem might have been possible, for example using the Kalman filter for the prediction and the modulated IR to filter out and separate from the other lights. However, it turns out that the camera is still very much limited by the distance and the darkness. As though it can detect IR, it has limited ability on decoding modulated IR, which is not ideal. Also for a very small IR to be in a very large space, the camera tends to not be able to track that distance accurately as its field of view is limited at far distance. Therefore, despite the effort and research on different algorithms and solution towards the CV and camera solution, we eventually decided that this is not the most ideal solution, and thus didn't proceed working on it further.

In researching infrared (IR) communication methods, both unmodulated and modulated approaches were evaluated with respect to reliability, noise immunity, and suitability for embedded systems. Unmodulated IR, which relies on directly turning on IR LED on and off to represent data, was found to be highly susceptible to ambient infrared interference from sources such as sunlight, incandescent lighting (ceiling lights etc.), and thermal noise. This significantly degrade signal integrity and limits practical communication range and robustness. We tested this in a semi-dark room and noticed the interference and noise waveforms from the unmodulated signal. As a result, the final solution selected was modulated infrared communication, specifically using a carrier frequency of 38kHz consistent with the NEC protocol. With this approach, data is encoded by gating a high-frequency carrier on and off, rather than transmitting raw optical pulses. Modulated IR enables the use of commercially available IR receiver modules the incorporate band-pass filter, automatic gain control, and demodulation centered around the carrier frequency. These features substantially improve noise rejectio, reliability, and decoding accuracy. Consequently, modulated IR was determined to be the most effective and scalable solution for the system, aligning with the NEC industry standards (mentioned later) whilst supporting robust short-range wireless communication is noisy environment such as the theatre lighting.

To develop the actor tracking system, we will take a hybrid approach that combines analog sensor data from the IR detection system with the computer vision system, enabling accurate, real-time tracking of an actor's location on stage.

Our approach with the analog sensor data from the IR detection system is first building a working infrared detection system. We initially tested this by using an off-the-shelf IR sensor, but realized that the outputs were mainly all purely digital, and the ones with analog output could not be fully customizable into a wearable. Thus, we decided to build our own IR detection system using phototransistors and IR Leds. The circuit diagram shown below indicates our current setup. The only problem with this system is that the phototransistor detection of the IR waves emitting from the Leds is very directionally dependent, in the sense that once the Led is placed slightly off center from the transistor's direct point of view, the detection is not as powerful and the analog values produced are not as consistent. Thus, the expected result would be to have an array of these detectors or a diffraction system that would spread out the IR waves in a wider formation, such that it would be able to detect all IR waves emitted from any angle of the Leds' positions.

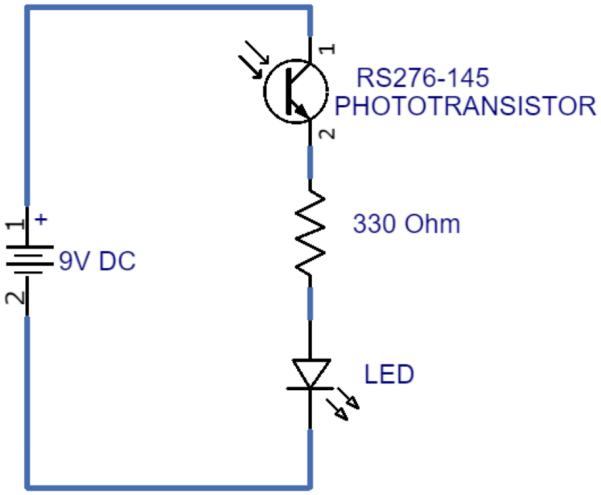

Figure 1: Example of IR detection Circuit for Unmodulated Transmission

Design implementation

System Overview

The stage tracking system is designed as a distributed, event-driven architecture that integrates embedded sensing hardware, wireless communication, a centralized backend, and a graphical user interface. The system consists of on-stage infrared (IR) receiver nodes, a wearable IR emitter, a Raspberry Pi backend controller, and a real-time GUI for position visualization.

On stage, IR receivers detect modulated infrared signals emitted by a wearable IR LED attached to the actor. Each receiver is connected to a Pico W microcontroller, which processes the

received signal and publishes detection events wirelessly using MQTT over WiFi. These telemetry messages are transmitted to a locally hosted MQTT broker running on a Raspberry Pi.

The Raspberry Pi backend subscribes to telemetry topics from all on-stage nodes, processes incoming detection events asynchronously, and serves as the coordination hub for the system. Parsed sensor events are forwarded to a Tkinter-based graphical user interface, which displays a simplified top-down representation of the stage and indicates the actor's detected position.

IR Tracking

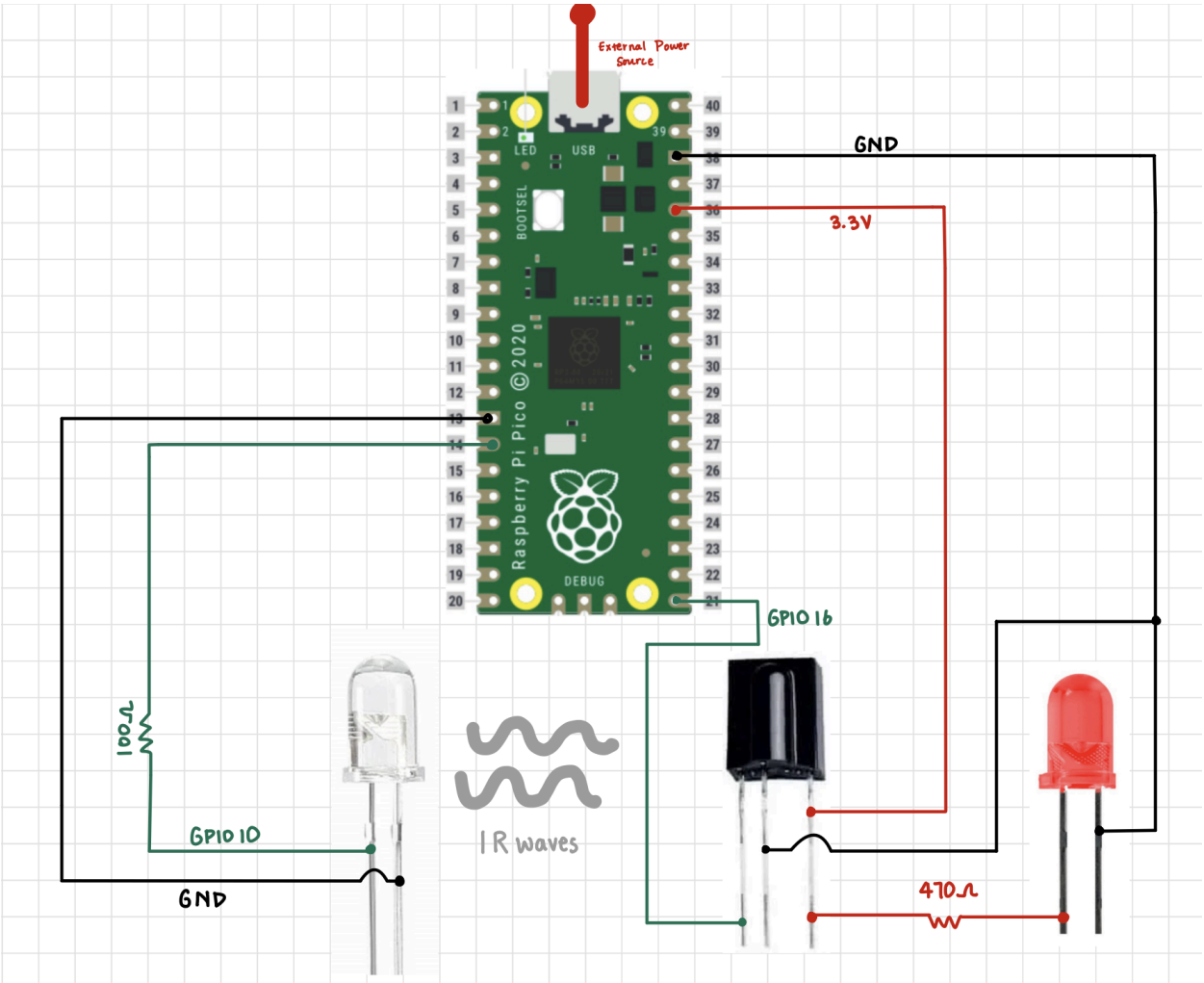

Figure 2: Breadboard Wiring Diagram of IR detection Circuit for Modulated Transmission

Our approach with the analog sensor data from the IR detection system is first building a working infrared detection system. We initially tested this by using an off-the-shelf IR sensor, but realized that the outputs were mainly all purely digital, and the ones with analog output could

not be fully customizable into a wearable. Thus, we decided to build our own IR detection system using phototransistors and IR Leds.

We had multiple iterations of our IR circuit setup, this was mainly due to multiple iterations of testing our analog sensor data. We originally had one IR LED producing a wave and tried to have the transmitter detect the signal. The only problem with this system is that the phototransistor detection of the IR waves emitting from the LEDS are very directionally dependent, in the sense that once the Led is placed slightly off center from the transistor's direct point of view, the detection is not as powerful and the analog values produced are not as consistent. In addition, we tested this in a place where there was almost no light/wave interference apart from the IR waves themselves.

We then tried to increase the range of the IR wave being emitted, so that the wave can travel in a broader distance, rather than one direction. This was done by using scotch tape and covered the IR LEDS and saw a higher target detection from the transmitter as the user was walking by. This proved our hypothesis that the range and intensity of the IR wave needed to be improved in order to have a stronger and more accurate detection of the target.

This conclusion helped formulate our final design of the IR detection system, where each IR wave is intensified by having multiple IR emitters for a single transmitter. However, there is now an issue of where to locate the person once the receiver has detected the placement of the IR transmitter.

Tracker PCB Design

To make our trackers more portable and able cover as large of an area as possible (allowing us to fulfill and locate any size theater or room). The schematic was taken directly from our working version we had laid out over on the breadboard, which we had tested throughout before. We split the transmitter and receiver between two boards, with the transmitter board having more IR LEDs to portray a stronger intensity of IR waves onto the transmitter. Both boards were purchased throughole, not only because this was much cheaper, but also giving us the freedom to add and customize the boards afterwards in the lab as well.

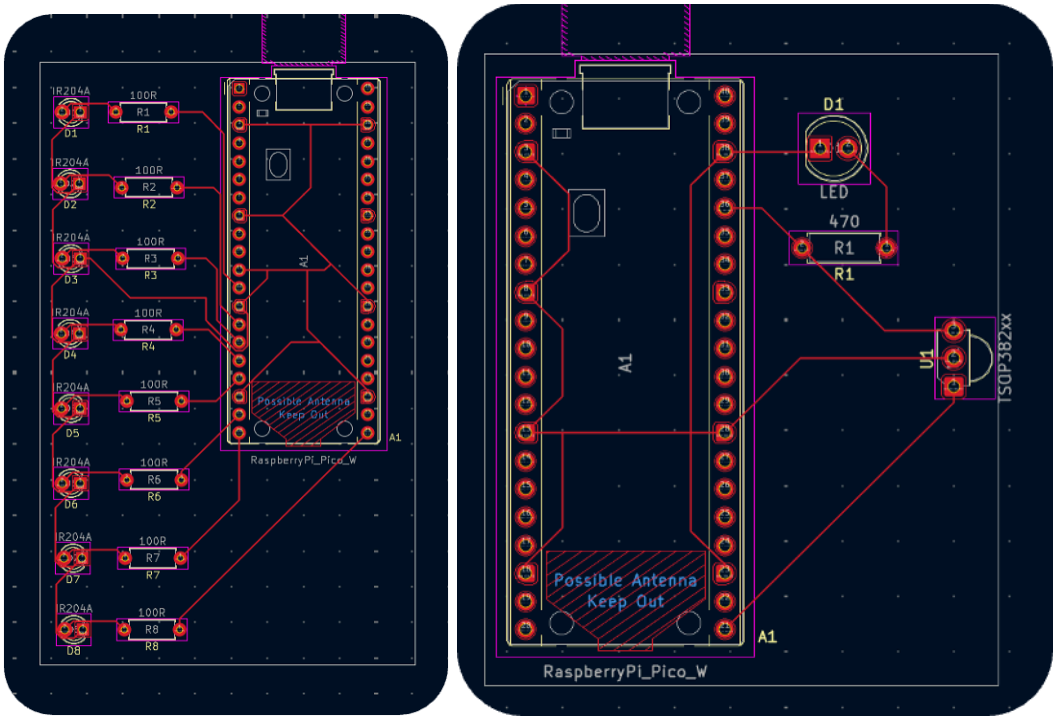

Figure 3: Printed Circuit Board Layout and Wiring of IR detection Circuit for Modulated Transmission - Receiver Board (Right) Transmitter Board (Left)

NEC Infrared Protocol

This is where the NEC protocol comes in. The NEC protocol, also known as the Nippon Electric Company Protocol is the most common Infrared standard for remote controllers. It uses pulse-distance modulation to send commands at 38KHz with high reliability. The resulting signal is logically timed encoded, each 32bits long. This protocol was chosen for many reasons. The main ones being error detection, wide industry adoption, low implementation complexity, repeat-frame mechanism, and noise-robust carrier modulation. The protocol includes bitwise inverted copies of the address and command fields, which enables simple integrity checking and automatic rejection of corrupted frames, as shown in the figure below. The NEC standard is so widely used that this system can be transferred to any other IR emitting devices, which gives it flexibility to be carried over to a newer or older system. The encoding and timing rules are relatively simple, allowing transmitters and receivers to be implemented with inexpensive components and minimal processing requirements. It also uses a standardized “repeat code” which allows efficient transmission when a user holds down a button, reducing bandwidth and simplifying receiver state machines. The biggest advantage of this protocol is that the 38KHz carrier simplifies filtering of ambient IR interference (like sunlight, or lightbulbs) which helps improve reception reliability with low-cost IR receiver modules.

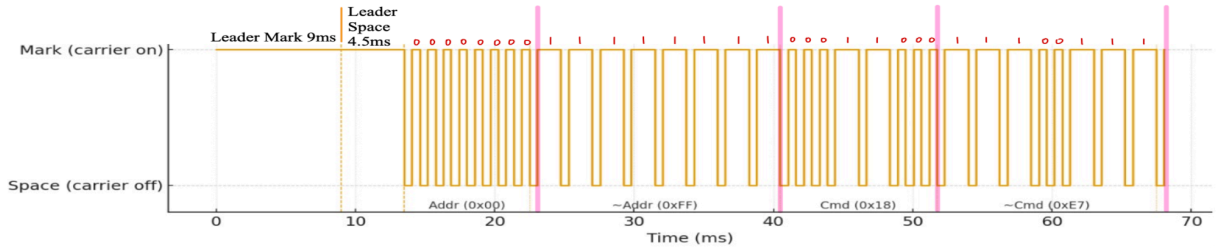

Figure 4: NEC IR Frame Timing Diagram (Addr = 0x00, Cmd = 0x18, LSB-first)

The encoded NEC bitstream is shown above. Like previously mentioned, the message starts with the leader mark and space, where the leader mark signals a new message and the leader space signals the beginning of the messaging. The logical bits 0s and 1s are produced by different distances between pulses, in our case the 0s have a shorter distance and the 1s have a larger distance. The address is broadcasted first in the bitstream, followed by its inverse for error detection. This formatting is then repeated by the command values in the bitstream. The error detection is made such that if one of the bits were lost in transit or corrupted due to external interference, the user would still be able to detect the missing/incorrect bit and logically formulate back the original bitstream with the extra inverted bits. The entire bitstream is 32 bits long, with 16 bits having important data from the signalling.

This IR communication was originally used in older television remote controllers, in the sense that each remote came with a transmitter inserted on the front end of the controller and the television had a receiver, which detected any buttons pushed wirelessly. The address of the NEC was to identify each remote controller while the command was used to identify which button was pressed on the controller. However, in our case, we do not have to identify each IR LED, just the location, so the address and command bits will be the x and y coordinates of the location instead. In the next sections below, we will go into more detail about the coordination and gui setup of our location tracker.

The program implements an embedded infrared IR communication system using the ENC protocol on a MicroPython-compatible microcontroller, Raspberry Pi Pico W. The code portion is listed in the Appendix/Reference section. The software configures two GPIO pins as PWM-driven IR emitters operating at a 38kHz carrier frequency, which is required for NEC-compliant transmission. The code generates protocol-accurate IR waveforms by explicitly controlling the marks (carrier on) and spaces (carrier off) intervals with microsecond-level timing. Using these pulses, it constructs and transmits full 32bit NEC frames consisting of the address (row), a command (column), and its inverses, which thereby enables the basic error detection at the receiver.

In addison to transmission, the program is structured to support the NEC IR reception through external IR receiver modules. Callback functions are defined to process synchronously received

IR frame, filter them by device address, decode command values into human-readable key identifiers, and store valid events for later handling in the main execution loop. This allowed for better debugging, as it always checks to see if there is anything valid to decode which made testing for interference and basic functionality simple and user-friendly. The architecture of our program was coordinated such that there is a clean separation between the interrupt-driven reception from the application-level processing. This not only helped prevent any confusion for future works on this project, but also allowed us to better organize and structure our program for the addition of WIFI communication, mentioned in the MQTT section below.

The main loop coordinates system behavior by periodically transmitting a test IR command once per second, which toggles the onboard LED as a visual status indicator, and checking for newly received IR events. When a valid IR command is detected, the code formats the decoded information into a structured payload suitable for telemetry or message-based communication, depending on the application. We originally used integer-based communication, just for simplicity and clarity, but later switched to message/string-based communication for better completion of the WIFI protocol. Hooks for WIFI and MQTT connectivity are also included, showing the integration with networked monitoring and control systems, as mentioned in the section below.

Overall, the program was structured such that it provides a modular and extensible implementation of NEC-based infrared communication, combining precise low-level signal generation with higher-level event handling and system diagnostics, and is well suited for embedded systems experimentation and future incremental expansion.

MQTT Communication Protocol

For the tracking system to be able to put in use with actual follow light control, communication from the on-stage tracking PCBs and the backstage control is essential. In order to allow distant communication, a lightweight, low-latency communication is needed. To meet this requirement, the system employs the Message Queuing Telemetry Transport (MQTT) protocol for wireless communication over WiFi. MQTT was selected due to its low overhead, publish–subscribe messaging model, and widespread support on embedded platforms.

System Architecture

The communication architecture consists of multiple Raspberry Pi Pico W nodes acting as MQTT clients and a Raspberry Pi acting as the MQTT broker and backend controller. Each Pico W establishes a WiFi connection and communicates with the broker using topic-based messaging. The property of decoupled publish–subscribe design of the MQTT enables scalable, asynchronous data exchange between distributed sensor nodes and the backstage controller.

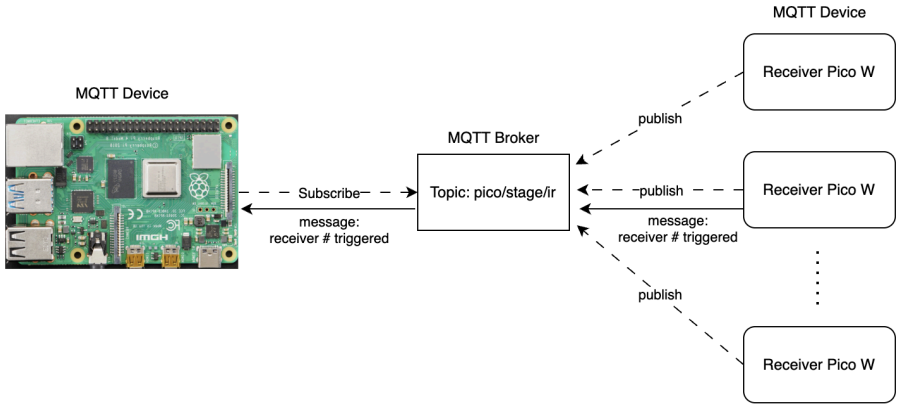

Figure 5: MQTT Signal and Relations Diagram

Each Pico W client publishes periodic telemetry data to a designated topic, while they subscribe to a command topic for incoming control messages at the same time. The Raspberry Pi backend subscribes to telemetry topics, processes incoming data, and issues control commands as needed.

On Stage PCB Message Publishing

On each on-stage Pico W node, MQTT communication is implemented using the umqtt.simple library. After connecting to WiFi, the Pico W creates an MQTTClient object using a unique CLIENT_ID and connects to the MQTT broker specified by BROKER_IP. All communication for the node is handled through this client object.
To continuously report node activity, we implemented a periodic telemetry publishing mechanism. Telemetry messages are published to the topic defined by TOPIC_TELEM "pico/track/telemetry". A software counter variable is initialized at startup and incremented once per telemetry cycle to act as a lightweight timestamp.
Inside the main execution loop, the function publish_telemetry() is called once every second. This function constructs a Python dictionary containing the current timestamp, a status field indicating that the node is active, and the decoded information of the triggered sensor. The dictionary is serialized into a JSON string using json.dumps() and sent to the broker using client.publish(TOPIC_TELEM, payload).
This periodic telemetry serves as the signal that allows the backend to confirm that each on-stage node is still connected and running, and informs the information of which node is being detected or having the actor around. The publish interval was chosen to be 1 second so that it can provide real time visibility while keeping network traffic minimal when multiple nodes are active.

Command Handling and Control

In addition to publishing telemetry, each Pico W node subscribes to a command topic defined by TOPIC_CMD "pico/track/cmd". After establishing the MQTT connection, the node registers a callback function mqtt_callback(topic, msg) using client.set_callback(), and then subscribes to the command topic with client.subscribe(TOPIC_CMD).

The function mqtt_callback() is responsible for handling all incoming command messages. When a message arrives, the callback decodes the payload from bytes to a string and parses the JSON content into a Python dictionary. The dictionary contains a cmd field that specifies the requested action.

In the current implementation, the supported commands allows us to know the status of the backend Raspberry Pi and forms as an instruction that successful communication between the two systems has been established.

To avoid blocking the main execution flow, incoming MQTT messages are handled using a non-blocking polling approach. In the main loop, client.check_msg() is called every iteration. This function only invokes mqtt_callback() when a new message is available, allowing telemetry publishing and other tasks to continue uninterrupted when no commands are received. As a result, command handling does not interfere with the fixed one-second telemetry transmission schedule.

This command-based control interface provides a flexible foundation for future system expansion. The same TOPIC_CMD channel and JSON parsing structure can be extended to support runtime configuration updates, sensor recalibration requests, or synchronization commands for coordinating multiple on-stage nodes in a distributed tracking system.

Raspberry Pi Backend Interface

On the Raspberry Pi backend, we implemented a Python-based MQTT client using the paho.mqtt.client library. The Raspberry Pi hosts the MQTT broker locally, configured to listen on BROKER_HOST = "localhost" and BROKER_PORT = 1883. By running the broker on the same machine as the backend application, all on-stage Pico W nodes can connect to a single, low-latency message hub without relying on external network infrastructure. This local broker setup simplifies system deployment, reduces communication latency, and significantly improves reliability during testing and live operation.

The backend MQTT client is configured with the broker address and port and connects at program startup. After establishing the connection, the client subscribes to the telemetry topic defined by TOPIC_TELEM "pico/track/telemetry", which is shared by all on-stage Pico W nodes. Incoming telemetry messages are handled by the callback function on_message(client, userdata, msg), which is registered using client.on_message.

When a telemetry message is received, the callback decodes the payload from bytes to a string and parses the JSON content into a Python dictionary. Each message contains fields such as a timestamp counter, node status, and triggered sensor information. The backend logs this parsed

data for monitoring purposes and can also forward the information to other subsystems, such as the graphical user interface or tracking logic.
To support bidirectional communication, the backend publishes control messages to the command topic defined by TOPIC_CMD "pico/track/cmd". Control messages are constructed as JSON dictionaries specifying a command type and any required parameters, then serialized using json.dumps() and sent using client.publish(TOPIC_CMD, payload). This mechanism allows the backend to issue commands such as LED control, state resets, or synchronization signals back to individual Pico W nodes.
The MQTT client runs in an asynchronous network loop using client.loop_start(), allowing message handling to occur in the background without blocking the main backend logic. This enables the Raspberry Pi to concurrently process telemetry from multiple Pico W nodes while issuing control commands at independent intervals, without requiring explicit polling or direct socket management.

Advantages of MQTT for Stage Tracking

Using MQTT as the communication layer provides several practical advantages for the stage tracking system. Because MQTT messages are lightweight and published over topics rather than point-to-point connections, the backend can scale naturally to support multiple on-stage tracker nodes and receiver stations. All nodes publish telemetry to the same TOPIC_TELEM, while commands can be broadcast or selectively handled depending on message content.
The asynchronous, event-driven nature of the Paho MQTT client allows the Raspberry Pi backend to react to incoming data in real time while remaining responsive to user input and GUI updates. In addition, MQTT's tolerance to intermittent WiFi connectivity makes it well-suited for theater environments, where wireless conditions may vary due to interference or audience presence.
Overall, this MQTT-based backend architecture provides a flexible and extensible interface between the embedded on-stage hardware and the Raspberry Pi control system. By clearly separating telemetry collection and command distribution through topic-based messaging, the system avoids tight coupling between components and remains easy to extend as additional tracking features or hardware nodes are introduced.

Graphical User Interface and Position Interpretation

Stage Representation and Layout Mapping

The graphical user interface is implemented on the Raspberry Pi using the Tkinter framework. The GUI renders a simplified, top-down view of the stage as a fixed grid corresponding to the physical layout of the infrared (IR) receivers deployed on stage. In the current implementation, the stage is modeled as a Row = 2 by Column = 5 grid, representing a total of ten receiver positions, but this number could be changed based on the configuration of the stage.

The layout parameters of the stage are defined explicitly using constants such as ROWS, COLS, OFFSET_X, OFFSET_Y, SPACING_X, and SPACING_Y. These parameters control the number of sensors, their spacing, and their placement on the canvas, allowing the stage geometry to be easily reconfigured by modifying these values without changing the rendering logic. The overall canvas size is computed from these parameters when the StageGUI class is initialized.
Inside the StageGUI constructor, a Tkinter Canvas object is created and used to draw the stage layout. Each IR receiver location is represented by a static outlined circle drawn using canvas.create_oval(). The center coordinates of each circle are computed based on the row and column index of the sensor and stored in a mapping table that associates a sensor ID with its corresponding (x, y) canvas coordinates. This mapping allows incoming sensor events to be translated directly into screen positions.
In addition to the static sensor outlines, a single dynamic marker is used to represent the actor's current detected position. This marker is also drawn as a filled circle on the canvas and is updated whenever a new sensor event is received. By separating static stage elements from the dynamic position marker, the GUI remains visually clean and easy to interpret during operation.

Event-Driven Position Updates

The GUI follows an event-driven update model that decouples network communication from visualization. MQTT communication is handled in a separate background thread using the Paho MQTT client. Incoming MQTT messages are processed by the on_message(client, userdata, msg) callback, which parses the received payload and extracts the sensor identifier associated with the most recent IR detection.
Rather than updating the GUI directly from the MQTT callback, each parsed sensor event is placed into a thread-safe queue, event_q, using event_q.put(sensor_id). This design avoids unsafe cross-thread GUI updates and ensures that network activity does not block or interfere with Tkinter's main event loop.
On the GUI side, the StageGUI class implements a periodic polling function, typically named poll_queue(), which is scheduled using root.after(). This function checks event_q at short intervals and processes any pending sensor events. Because the polling is non-blocking, the GUI remains responsive even during periods of high message traffic or temporary network delays.
This separation between MQTT communication and GUI rendering ensures that variations in message arrival timing do not affect the stability or responsiveness of the interface. It also allows the system to scale to higher sensor update rates without requiring continuous redrawing of the entire canvas.

Position Interpretation Logic

Each event retrieved from event_q contains the identifier of the IR receiver that most recently detected a valid signal. The GUI interprets this identifier as a discrete spatial region on the stage. Using the previously constructed sensor-to-coordinate mapping table, the GUI determines the canvas coordinates associated with the received sensor ID.

When a valid sensor ID is processed, the dynamic position marker is either created or repositioned using canvas.coords() to center it over the corresponding sensor circle. If the received message indicates that no valid sensor is currently active, the marker is removed from the canvas using canvas.delete(), visually indicating a loss of tracking.

This grid-based interpretation deliberately avoids estimating continuous spatial coordinates. Instead, it provides region-based localization that is robust to noise and sensor ambiguity. For the purposes of real-time spotlight control, this level of spatial resolution is sufficient to guide manual or automated adjustments while maintaining system simplicity and reliability.

Threading and Responsiveness Considerations

To maintain real-time responsiveness, MQTT communication and graphical rendering are executed in separate execution contexts. All MQTT networking is handled in a dedicated background thread using the Paho MQTT client, while all Tkinter rendering and user interaction occur on the main GUI thread. This separation is necessary because Tkinter is not thread-safe and must only be accessed from the main thread.

Incoming MQTT messages are processed in the on_message(client, userdata, msg) callback, which runs in the context of the MQTT network loop. Rather than modifying the GUI directly, the callback extracts the sensor identifier from the message payload and places it into a thread-safe queue, event_q, using event_q.put(). This queue serves as the sole communication channel between the networking thread and the GUI.

On the GUI side, the StageGUI class periodically polls event_q using a scheduled callback implemented with root.after(). The polling function retrieves any pending sensor events and updates the position marker accordingly. Because the queue operations are non-blocking and the polling interval is short, the GUI remains responsive even during sustained telemetry traffic or bursty message arrival patterns.

The GUI refreshes at a fixed interval determined by the Tkinter event loop rather than redrawing continuously. Visual updates are limited to repositioning or deleting the dynamic position marker using canvas.coords() or canvas.delete(), instead of re-rendering the entire stage layout. This design minimizes computational overhead and ensures smooth position updates even on limited hardware such as the Raspberry Pi.

Design Advantages and Extensibility

The GUI is structured in a modular way, with clear separation between stage layout definition, event handling, and rendering logic. Because all position updates are driven solely by sensor identifiers received over MQTT, the interface remains decoupled from the underlying sensing hardware implementation.

This design allows the GUI to be extended easily to support additional functionality. For example, multi-actor tracking could be implemented by maintaining multiple position markers, probabilistic smoothing could be added by filtering sensor events before updating the marker, and confidence visualization could be incorporated by encoding signal strength or detection reliability into marker size or color. These extensions can be implemented without modifying the MQTT communication layer or the core stage layout logic.

Overall, the graphical interface acts as a bridge between low-level embedded sensing hardware and human operators. By translating discrete IR detection events into an intuitive spatial representation, the GUI provides actionable, real-time feedback that supports both manual and automated spotlight control while remaining efficient, robust, and adaptable to future system enhancements.

Evaluation and Results

Functional Verification and Test Methodology

Initial system testing was conducted in a laboratory environment using a single on-stage PCB and one Raspberry Pi backend controller. The primary objective of this phase was to verify correct end-to-end functionality, including IR detection, MQTT telemetry publishing, backend message reception, and GUI position updates. During these tests, the on-stage PCB was manually triggered to simulate IR detection events, and the corresponding telemetry messages were observed on the Raspberry Pi console and GUI.

Once basic functionality was confirmed, the system was tested in a more realistic deployment scenario by moving the tracked subject outside the laboratory environment. In this test, the Raspberry Pi backend and GUI remained inside the lab, while a person wearing the tracking PCB walked in the hallway outside. This setup allowed us to evaluate long-distance WiFi communication, MQTT robustness across physical barriers, and GUI responsiveness when the tracked subject was no longer in direct proximity to the controller.

In both scenarios, the system behaved as expected. Telemetry messages were successfully transmitted from the Pico W node to the MQTT broker, received by the Raspberry Pi backend, and reflected in real time on the GUI. No message loss or visible degradation in responsiveness

was observed during hallway testing, demonstrating that the system meets its core functional requirements under realistic operating conditions.

Communication Reliability and MQTT Performance

A key design expectation was that MQTT would provide reliable, low-overhead communication suitable for real-time tracking without requiring centralized polling or persistent socket connections. During testing, MQTT communication remained stable over distances of up to approximately 50 feet, including through walls separating the hallway and laboratory.
Telemetry messages were published at a fixed rate of one message per second, and the backend consistently received these updates without backlog or noticeable delay. Command messages issued from the Raspberry Pi to the Pico W node were also delivered reliably, with LED control actions occurring promptly upon message receipt. This behavior aligns with the expected performance of MQTT in local-area network conditions.
The system was tested on both home WiFi networks and university-managed WiFi networks. In both cases, MQTT connectivity was established successfully, and message delivery remained reliable. This confirms that the system does not rely on specialized network configurations and can operate effectively on common WiFi infrastructure, which is essential for deployment in diverse theater environments.

Real-Time Responsiveness and GUI Behavior

Another core expectation of the system was that actor position updates would appear on the GUI in near real time, without freezing or lag due to network communication or message processing. Testing confirmed that the event-driven GUI architecture successfully isolates network activity from rendering operations.
Incoming sensor events were consistently reflected in marker position updates within a short and perceptually acceptable delay. Because the GUI only updates the dynamic position marker rather than redrawing the entire stage layout, visual updates remained smooth even during sustained message traffic. The separation of MQTT communication into a background thread and GUI updates into the main thread prevented blocking behavior commonly seen in poorly synchronized GUI applications.
In cases where no valid sensor events were received, the GUI correctly removed the position marker, indicating loss of tracking. This behavior matches the intended design and provides a clear visual cue to the operator when tracking data is unavailable.

Physical Deployment and Wearability Evaluation

Beyond software and communication testing, the system was evaluated for physical usability. The tracking PCB was designed to be attachable to clothing or fabric, allowing it to be worn by a person at variable heights and orientations. During hallway testing, the PCB was attached to clothing at different positions, and detection behavior remained consistent.

The small form factor of the custom on-stage PCB contributed positively to usability. When placed appropriately on clothing, the device was not easily noticeable, which is advantageous for live performance settings where visual distractions must be minimized. The PCB's portability also simplifies setup and teardown, making it suitable for temporary installations or rehearsals.

Cost Analysis and Design Feasibility

Cost was an important consideration in the original system design. The total cost of the tracker hardware and GUI infrastructure was calculated to be approximately $50.24, with the tracker hardware costing $11.74 and the GUI and backend components costing $38.50. This low overall cost demonstrates that the system is economically feasible for small to medium-scale theater productions.
Compared to commercial tracking or spotlighting solutions, which can be significantly more expensive, this system achieves acceptable functionality at a fraction of the cost. In addition, the 38.5 dollars is the cost of the Raspberry Pi cost, but this could be easily replaced by any existing laptop that we can incorporate WIFI and MQTT communication on, so the cost is even lower. The use of commodity hardware, open-source software libraries, and custom PCBs contributed to this affordability without sacrificing core performance.

Comparison to Original Design Expectations

Overall, the observed system behavior closely matched or exceeded original design expectations. The system successfully demonstrated reliable long-distance WiFi communication, real-time GUI updates, and robust performance in both controlled and realistic environments. The discrete grid-based localization approach proved sufficient for guiding spotlight control, validating the decision to prioritize robustness and simplicity over continuous position estimation.
No major functional failures were observed during testing. Minor sources of latency were attributable to network variability but did not significantly affect usability. These results confirm that the architectural decisions—particularly the use of MQTT, asynchronous communication, and event-driven GUI updates—were appropriate for the intended application.

In summary, system testing verified that the stage tracking platform meets its functional, performance, and usability goals. The system operates reliably across typical WiFi environments, supports real-time monitoring through an intuitive GUI, and remains portable, affordable, and unobtrusive. These results demonstrate that the system is suitable for real-world deployment and provide a solid foundation for future extensions such as multi-actor tracking, improved localization resolution, or automated spotlight control.

Conclusions and Future works

Conclusion

In this project, we designed and implemented a low-cost, modular system for automated stage tracking intended to support theatrical performances. By combining embedded sensing hardware, wireless communication, a Raspberry Pi backend, and a real-time graphical interface, we demonstrated a complete end-to-end pipeline that translates on-stage actor movement into an intuitive spatial representation. The system was built using commodity hardware, open-source software libraries, and custom PCBs, resulting in a solution that is both affordable and adaptable. The system successfully met its primary design objectives. On-stage tracker nodes reliably detected IR signals and transmitted telemetry data over WiFi using MQTT. The Raspberry Pi backend handled multiple asynchronous data streams while remaining responsive, and the GUI provided a clear, region-based visualization of actor position in real time. Testing in both laboratory and real-world hallway environments confirmed that the system is robust to network variability and suitable for deployment in theater-like spaces. Importantly, the overall system cost remained low, demonstrating that meaningful automation support for stage lighting does not require expensive commercial tracking solutions.

A key design choice was the use of discrete, grid-based localization rather than continuous position estimation. While this approach trades spatial precision for simplicity, testing showed that it is sufficient for guiding spotlight control in many theatrical contexts. By focusing on robustness, modularity, and clarity, the system provides a strong foundation for further automation while remaining understandable and maintainable.

Future works

While the current system demonstrates reliable tracking and visualization, it also opens several clear directions for future development. These extensions focus on improving accuracy, robustness, and usability, as well as completing the automation loop between sensing and physical actuation.

One of the most significant next steps is the integration of motorized follow-spot control. In this extension, the Raspberry Pi backend would directly drive a pan-tilt mechanism using servo or stepper motors to physically aim a spotlight based on the tracked actor position. The existing GUI coordinate system and sensor-to-position mapping can be repurposed to generate control signals for the motors. This would transform the system from a monitoring and visualization tool into a fully automated follow-spot solution. Closed-loop control algorithms could be introduced to ensure smooth and stable motion, reducing jitter and improving visual quality during live performances.

Another major direction for future work is the integration of a camera-based tracking system. A camera mounted near or on the spotlight could provide continuous visual feedback, allowing computer vision techniques to refine or validate the IR-based position estimates. This hybrid approach would combine the robustness of discrete IR detection with the higher spatial resolution of vision-based tracking. The camera could also be used to detect multiple actors simultaneously, enabling more complex lighting effects and transitions. Importantly, the current

MQTT-based architecture already supports the addition of new data sources without major restructuring.

Wireless synchronization is another area for improvement. While the current system performs well under typical WiFi conditions, future work could focus on tighter synchronization between multiple tracker nodes and the backend. Time-stamped telemetry messages and clock synchronization protocols could reduce ambiguity when multiple sensors are triggered in rapid succession. This would improve consistency in multi-node deployments and support higher update rates without sacrificing reliability.

Usability enhancements also present meaningful opportunities for future development. The GUI could be extended to support manual overrides, allowing a human operator to adjust or correct spotlight behavior in real time. Additional visualization features, such as confidence indicators, motion trails, or predictive position markers, could further improve operator awareness. These features would be particularly valuable in live performance settings, where quick interpretation and intervention are critical.

Finally, the modular design of the system makes it well-suited for broader experimentation. Alternative sensing modalities, such as ultra-wideband (UWB), Bluetooth beacons, or RFID, could be integrated by replacing or augmenting the IR detection layer while preserving the same backend and GUI infrastructure. This flexibility allows the system to evolve alongside advances in sensing technology without requiring a complete redesign.

Appendix

References

[1] “Implementing MQTT in python with code examples,” HiveMQ, https://www.hivemq.com/blog/implementing-mqtt-in-python/ (accessed Dec. 3, 2025). [2] “Project 31 : IR control sound and ledÁ,” Project 31 : IR Control Sound and LED - Raspberry Pi Pico Learning Kit documentation (accessed Dec. 3, 2025).
[3] H. World, “Remote Control IR with RPI,” YouTube, https://www.youtube.com/watch?v=-LxaNQjBn_s (accessed Dec. 3, 2025). [4] V. H. Adams, “Wireless Uart via infrared¶,” Infrared, https://vanhunteradams.com/Pico/Bootloader/Infrared.html (accessed Dec. 3, 2025).
[5] YoungWonks, “One can use an infrared sensor with a Raspberry Pi Pico. here’s how...,” YoungWonks, https://www.youngwonks.com/blog/How-to-use-an-infrared-sensor-with-the-Raspberry-Pi-Pico (accessed Dec. 3, 2025).
[6] “Lite-on DCC Release Lite-On Technology Corp. / Optoelectronics,” IR Emitter and Detector Product Data Sheet, https://optoelectronics.liteon.com/upload/download/DS-50-92-0005/E2871.pdf (accessed Dec. 3, 2025). [7] F. Zahid, “Infrared Detector Circuit using phototransistor,” Circuits DIY, https://www.circuits-diy.com/infrared-detector-circuit-using-phototransistor/ (accessed Dec. 3, 2025).

[7] “Top 5 object tracking methods: Enhancing precision and efficiency,” Master Computer Vision Courses Online With Augmented A.I., https://www.augmentedstartups.com/blog/top-5-object-tracking-methods-enhancing-precision-and-efficiency (accessed Dec. 15, 2025).

Parts List and Budget

Parts Name	Quantity	Total Price ($usd)
IR Receiver	1	$2.25
IR Transmitter	8	$12
RP2040 WIFI	2	$14
Raspberry Pi 4	1	$35
PCB (Receivers, Transmitters)	5 Receivers, 10 Transmitters	$39.93

Code Listing:

```
from machine import Pin, PWM

from utime import sleep_us, sleep

import time

import network, time, json

from lib.simple import MQTTClient



from lib.nec import NEC_8


WIFI_SSID="RedRover"; WIFI_PASS=""

BROKER_IP="10.49.70.17"

CLIENT_ID = "pico-alpha"

TOPIC_TELEM = b"pico/alpha/telemetry"

TOPIC_CMD   = b"pico/alpha/cmd"


# ===== IR EMITTER SETUP =====

emitter_pin = 15

emitter = Pin(emitter_pin, Pin.OUT)

pwm0 = PWM(emitter)


emitter_pin1 = 11

emitter_1 = Pin(emitter_pin1, Pin.OUT)

pwm1 = PWM(emitter_1)


# # NEC uses 38kHz modulation

# for pwm in (pwm0, pwm1):

#     pwm.freq(38000)

#     pwm.duty_u16(0) # Ensures that LED starts in OFF state


pwm0.freq(38000)

pwm0.duty_u16(0) # Ensures that LED starts in OFF state
```

```
pwm1.freq(38000)
pwm1.duty_u16(0) # Ensures that LED starts in OFF state



# ===== LED for debug =====
LEDpin = 25
led = Pin(LEDpin, Pin.OUT) # Onboard LED


# ===== IR SEND FUNCTIONS =====
def mark(pwm, usec):
   pwm.duty_u16(21845)  # ~33% duty cycle
   sleep_us(usec)
   pwm.duty_u16(0)


def space(pwm, usec):
   pwm.duty_u16(0)
   sleep_us(usec)


def send_bit(pwm, bit):
   mark(pwm, 562)  # NEC pulse length
   if bit:
       space(pwm, 1687)   # logic 1
   else:
       space(pwm, 562)    # logic 0


def send_nec(pwm, addr, cmd):
   # Build NEC 32-bit frame
   data = addr & 0xFF
   data |= ((addr ^ 0xFF) & 0xFF) << 8
   data |= (cmd & 0xFF) << 16
   data |= ((cmd ^ 0xFF) & 0xFF) << 24
```

```
    # Leader burst
    mark(pwm, 9000)
    space(pwm, 4500)

    # Send 32 bits (LSB first)
    for i in range(32):
        send_bit(pwm, (data >> i) & 1)

    # Stop bit
    mark(pwm, 562)
    space(pwm, 0)

# ===== Receiver Setup =====
pin_mod_ir = Pin(16, Pin.IN) # First Emitter pinout
pin_mod_ir1 = Pin(17, Pin.IN) # Second Emitter pinout

# Assigns different addresses for filtering
ADDR_0 = 0x00
ADDR_1 = 0x01

def decodeKeyValue(cmd):
    if cmd == 0x16:
        return "0"
    if cmd == 0x0C:
        return "1"
    if cmd == 0x18:
        return "2"
    if cmd == 0x5E:
        return "3"
    return f"Unknown ({hex(cmd)})" # Detects corruption and merging of IR waves...

latest_ir = None      # will hold (cmd, addr, ctrl)
```

```
def callback0(cmd, addr, ctrl):
    global latest_ir
    if cmd < 0:
        return
    if addr != ADDR_0:
        return

    key = decodeKeyValue(cmd)
    print(f"[Receiver {hex(addr)}] Received:", key, "Addr:", hex(addr))
    # Store event so main loop can publish via MQTT
    latest_ir = (cmd, addr, ctrl)

def callback1(cmd, addr, ctrl):
    if cmd < 0:
        # ignore errors / repeats
        return
    if addr !=ADDR_1:
        # ignores mesages not meant for this receiver
        return
    print(f"[Receiver {hex(addr)}] Received:", decodeKeyValue(cmd))

# Instantiate the NEC receiver ONCE
ir = NEC_8(pin_mod_ir, callback0) # First IR receiver
# ir1 = NEC_8(pin_mod_ir1, callback1) # Second IR receiver



def wifi_connect():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    if not wlan.isconnected():
        wlan.connect(WIFI_SSID, WIFI_PASS)
```

```
        while not wlan.isconnected():
            time.sleep(0.2)
    print("WiFi:", wlan.ifconfig())
    return wlan


def on_msg(topic, msg):
    try:
        #data = json.loads(msg)
        data = json.loads(msg.decode())
        if data.get("led") == "toggle":
            led.toggle()
        elif data.get("led") == "on":
            led.value(1)
        elif data.get("led") == "off":
            led.value(0)
    except Exception as e:
        print("Cmd parse error:", e)



# ===== MAIN LOOP =====
def main():
    global latest_ir


    wifi_connect()


    c = MQTTClient(CLIENT_ID, BROKER_IP, keepalive=30)
    c.set_callback(on_msg)
    c.connect()
    c.subscribe(TOPIC_CMD)


    print("MQTT connected & subscribed")
    print("System ready: Emitter + Receiver running...")
```

```
last_tx = time.ticks_ms()


t = 0
try:
    while True:
        c.check_msg()
        # 2) If IR receiver saw something, publish it
        if latest_ir is not None:
            cmd, addr, ctrl = latest_ir
            latest_ir = None   # consume event


            payload = {
                "t": t,
                "addr": addr,
                "led": decodeKeyValue(cmd)
            }
            print("Publishing IR telemetry:", payload)
            c.publish(TOPIC_TELEM, json.dumps(payload))


        # 3) (Optional) Still send IR periodically if you want to test transmit
        now = time.ticks_ms()
        if time.ticks_diff(now, last_tx) > 1000:   # every 1s
            last_tx = now
            led.toggle()
            send_nec(pwm0, ADDR_0, 0x18)   # "2"
            print(f"Sent IR: addr={ADDR_0} cmd=0x18")
        t += 1
        time.sleep(1)
except KeyboardInterrupt:
    print("KeyboardInterrupt, disconnecting MQTT...")
    c.disconnect()
```

```
main()
led.off()
print("Finished.")



import json
import threading
import queue
import tkinter as tk
import paho.mqtt.client as mqtt

#config
BROKER_HOST = "localhost"
BROKER_PORT = 1883
TOPIC_SIG = "pico/track/signal"
TOPIC_CMD  = "pico/track/cmd"

#GUI constants
ROWS = 2
COLS = 5
OFFSET_X = 50
OFFSET_Y = 50
SPACING_X = 80
SPACING_Y = 80
LED_RADIUS = 15

#-helper
event_q = queue.Queue() #queue for sensor events on mqtt

#for the gui
class StageGUI:
```

```
    # Map: led_id -> (cx, cy)
    # 1 through 5 on bottom row, 6 to 10 on top row
    def __init__(self, root):
        self.root = root
        self.root.title("IR Tracking Stage")

        width  = OFFSET_X * 2 + SPACING_X * (COLS - 1)
        height = OFFSET_Y * 2 + SPACING_Y * (ROWS - 1)
        self.canvas = tk.Canvas(root, width=width, height=height, bg="white")
        self.canvas.pack()
        self.led_positions = {}

        for led_id in range(1, ROWS * COLS + 1):
            logical_row = 0 if led_id <= COLS else 1 #0 bottom, 1 top
            col = (led_id - 1) % COLS
            canvas_row = (ROWS - 1) - logical_row #just need to do this because i coded
id wrong don't want to change :(
            cx = OFFSET_X + col * SPACING_X
            cy = OFFSET_Y + canvas_row * SPACING_Y
            #draw the sensor positions on stage
            self.canvas.create_oval(
                cx - LED_RADIUS, cy - LED_RADIUS,
                cx + LED_RADIUS, cy + LED_RADIUS,
                outline="gray", width=2
            )
            self.led_positions[led_id] = (cx, cy)

        #mark the tracked object
        self.marker_radius = LED_RADIUS - 5
        self.marker = None
        #preiodic check
        self.root.after(50, self.process_events)
```

```
   #periodically call in GUI thread to get the event from event queue and mark the
position
   def process_events(self):
       try: #there are events in queue
           while True:
               led_id = event_q.get_nowait()
               self.update_marker(led_id)
       except queue.Empty:
           pass
       #wait period
       self.root.after(50, self.process_events)


   #if led = 0 there is no sensor active, else mark the sensor
   def update_marker(self, led_id):
       if led_id == 0: #none detected
           if self.marker is not None:
               self.canvas.delete(self.marker)
               self.marker = None
           return
       if led_id not in self.led_positions: #out of range
           return


       cx, cy = self.led_positions[led_id]
       r = self.marker_radius
       if self.marker is None: #first signal detected
           self.marker = self.canvas.create_oval(
               cx - r, cy - r, cx + r, cy + r,
               fill="red"
           )
       else: #change existing marker if position change
           self.canvas.coords(
```

```
                self.marker,
                cx - r, cy - r, cx + r, cy + r
            )



#MQTT callback functions
#connection to topic
def on_connect(client, userdata, flags, rc):
    text = f"Connected (rc={rc})" if rc == 0 else f"Connect failed (rc={rc})"
    print("[MQTT] on_connect:", text)
    if rc == 0:
        client.subscribe(TOPIC_SIG)
        client.subscribe(TOPIC_CMD)
        print(f"[MQTT] Subscribed to {TOPIC_SIG} and {TOPIC_CMD}")

#receiving message on topic
def on_message(client, userdata, msg):
    try:
        payload_str = msg.payload.decode()
    except Exception:
        payload_str = msg.payload.decode(errors="ignore")

    print(f"[MQTT] {msg.topic} {payload_str}")

    if msg.topic == TOPIC_SIG: #we expect this topic for the sensor
        try: #parse
            payload = json.loads(payload_str)
        except Exception as e:
            print("[MQTT] JSON parse error:", e)
            return
        sensor_id = payload.get("id", 0)
        if not isinstance(sensor_id, int): #sensor id detection
```

```
            try:
                sensor_id = int(sensor_id)
            except Exception:
                sensor_id = 0

        #push to gui
        event_q.put(sensor_id)

#the mqtt thread
def mqtt_thread():
    #start mqtt
    client = mqtt.Client()
    client.on_connect = on_connect
    client.on_message = on_message

    print(f"[MQTT] Connecting to {BROKER_HOST}:{BROKER_PORT}...")
    try:
        client.connect(BROKER_HOST, BROKER_PORT, 60)
    except Exception as e:
        print("[MQTT] Initial connect failed:", e)
        return
    client.loop_forever()

#----------------MAIN-------------------------
def main():
    root = tk.Tk()
    gui = StageGUI(root)
    #mqtt in background
    t = threading.Thread(target=mqtt_thread, daemon=True)
    t.start()
    root.mainloop()
```

```
if __name__ == "__main__":

    main()
```