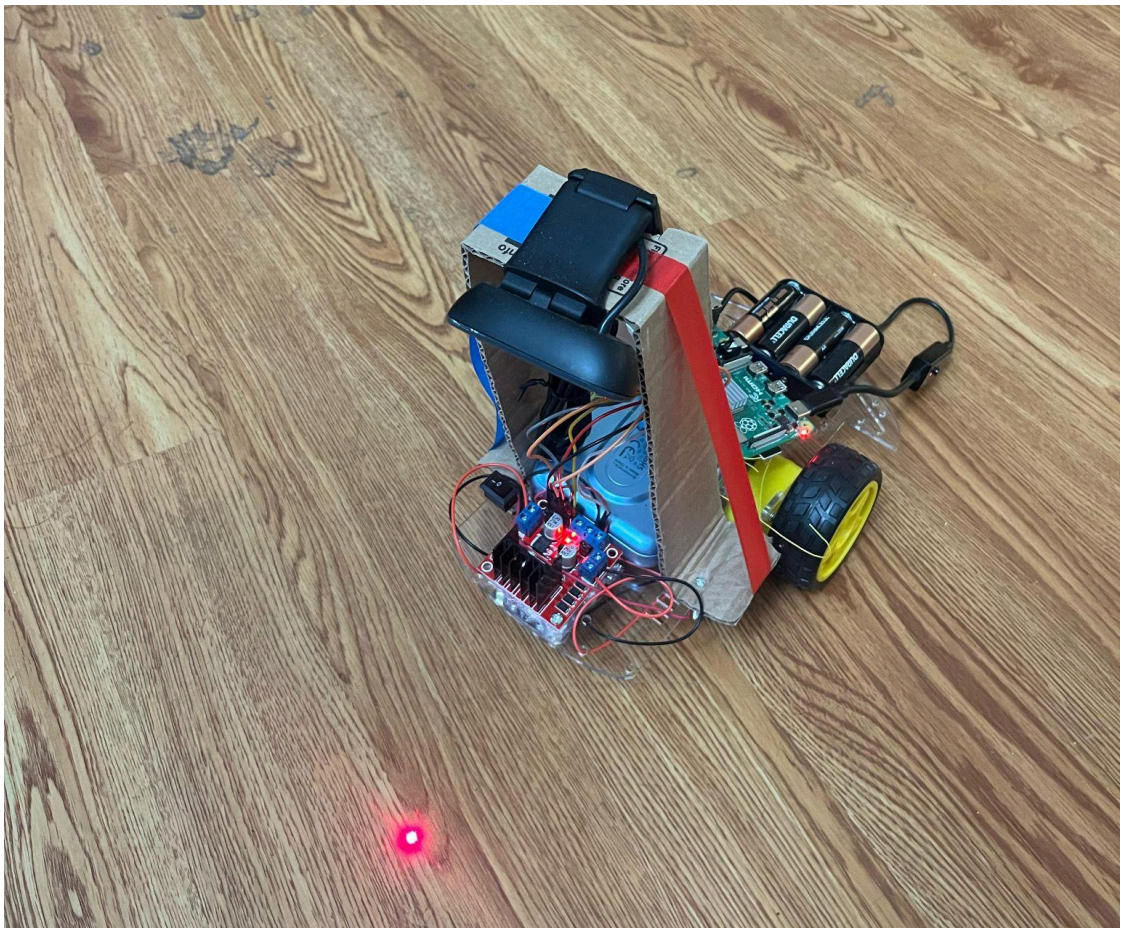
RTES FINAL PROJECT REPORT

Laser following robot



Lokesh Senthil Kumar
Bhavya Saravanan
Nalin Saxena

05/01/2025
ECEN 5623
Real Time Embedded Systems
Professor - Steve Rizor
Spring 2025

Table Of Content:

Introduction:

This project presents the design and implementation of a real-time embedded robotic system called Laser-Following Cat Bot, built using a Raspberry Pi 4B Controller with C++ code within the Linux environment. The system is inspired by the playful behavior of cats chasing laser pointers and aims to replicate that interaction through autonomous motion control. At its core, the robot uses a camera to detect a red laser dot projected within its field of view and moves in the corresponding direction to follow it in real-time.

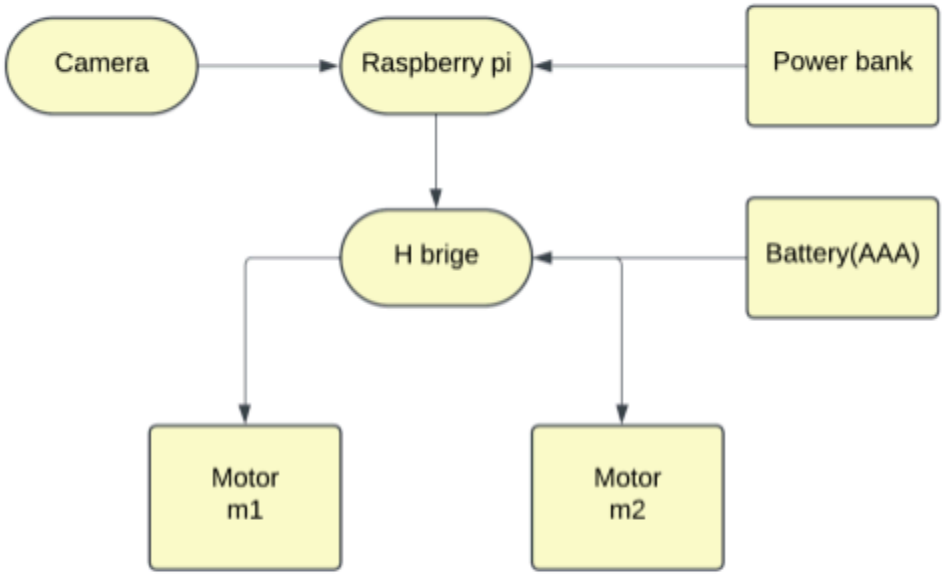
The robot's core functionality is achieved by modularizing its behavior into four primary services: camera frame capture, laser detection using image processing, direction decision-making, and motor control using GPIO PWM signals. Image acquisition is handled using Video4Linux2 (V4L2) in non-blocking mode, allowing efficient real-time frame capture. The captured frames are processed with OpenCV, where the red laser is detected through HSV masking and contour analysis. The centroid of the detected laser spot is calculated and used to decide movement direction, such as forward, left, or right.

To ensure real-time guarantees, the system employs a custom Rate Monotonic (RM) Sequencer, which coordinates the execution of these services using fixed-priority scheduling. These services run on Core 1, while Core 2 is dedicated to monitoring through a hardware watchdog timer. The watchdog service checks health flags from each service every 100 ms and allows the system to automatically reboot if any service becomes unresponsive. Additionally, a dynamic configuration service was implemented to enable real-time updates to system parameters. This allows users to adjust the robot's behavior, such as switching between forward and reverse movement, based on a configurable behavioral flag set in the configuration file.. This project demonstrates an end-to-end real-time application.

Design Overview

The following section provides various aspects of the project such as hardware, software and overall system.

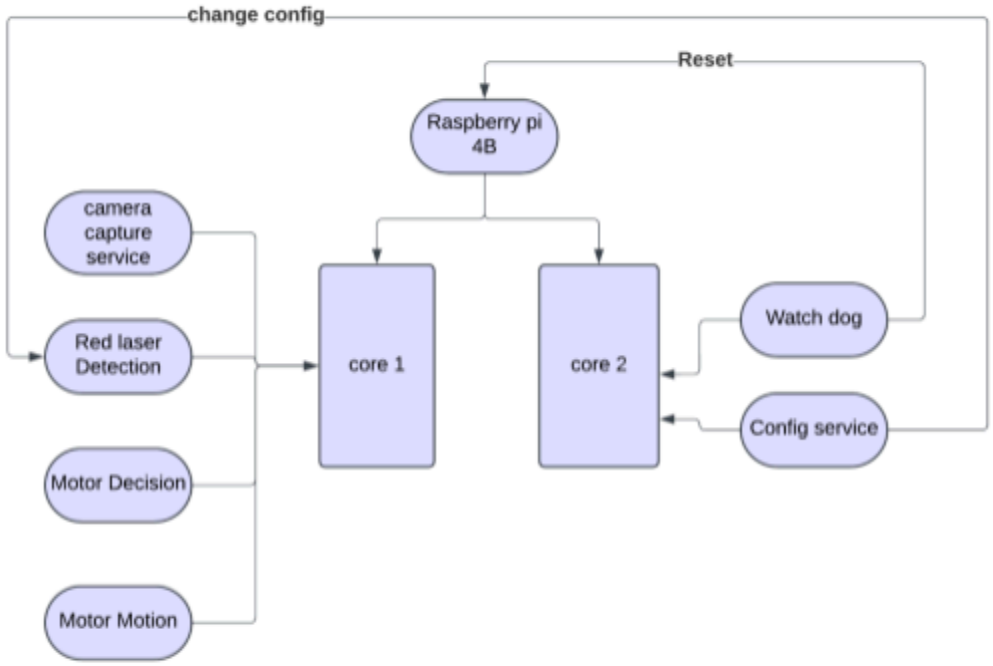
a. Hardware Design



Hardware Block Diagram

The hardware design for this project is straightforward yet effective, centered around a Raspberry Pi 4B which serves as the main processing unit. A Logitech C270 USB camera is connected to the Raspberry Pi to continuously capture video frames for image processing. The motors used to drive the robot are controlled through an L298N H-Bridge motor driver, which is responsible for translating GPIO-based PWM signals from the Raspberry Pi into appropriate control signals for motor actuation. The motors used in the setup are rated for 3V/6V with a current requirement of ≤180mA at 3V and ≤250mA at 6V, which are well within the capabilities of the L298N driver. The driver itself supports an input voltage of 5–35V and up to 2A of current per channel, and it is powered using a set of AAA Duracell batteries.To maintain the portability of the entire system, a generic USB power bank is used to supply power to the Raspberry Pi. All components in this setup, excluding the camera, Raspberry Pi were purchased as part of a hardware kit from Amazon.

b. System overview



System Block Diagram

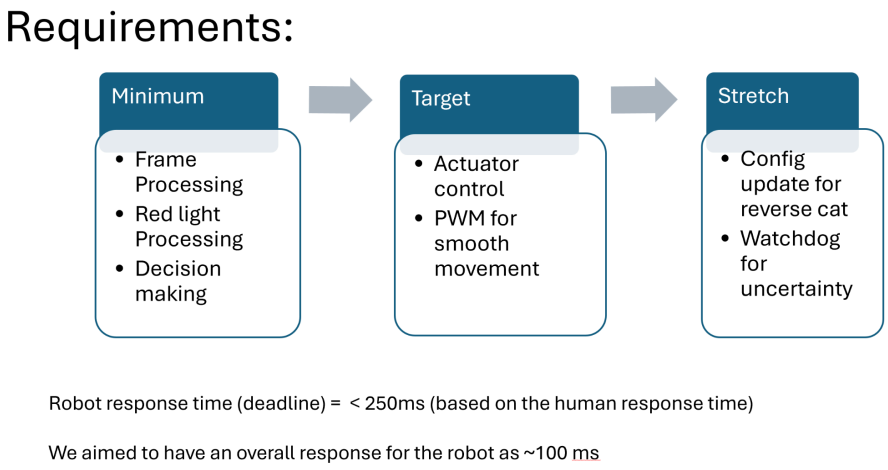
The system has 4 essential services which are camera capture, laser detection, motor direction decision and finally motor motion. All these services are pinned to core 1 and scheduled using RM (rate monotonic policy).To further enhance system flexibility and reliability, two additional services - Configuration Service and Watchdog Service are offloaded to Core 2.

System Requirements

To ensure responsive and accurate movement of the robot in real time, our system was designed with precise timing constraints and a well-defined scheduling policy. Each of the four primary services, Camera Frame Capture, Red Laser Detection, Motor Direction Decision, and Motor Control, is executed on Core 1 using a Rate-Monotonic (RM) scheduling strategy. These services were assigned specific periods, priorities, and deadlines to guarantee deterministic behavior.

To accurately determine the Worst-Case Execution Time (WCET) of each service, we executed them multiple times and recorded timestamps before and after execution. This allowed us to compute the minimum, maximum, and average execution times. For better insight, we visualized the execution profiles using histogram plots generated with Python, helping validate the real-time properties of each task.

During the design phase, our initial goal was to have the robot respond to the red laser dot with minimal latency and jitter, setting a system-wide deadline of 100ms from laser detection to robot movement. After optimization, such as making the camera capture non-blocking and fine-tuning the service code, we successfully reduced the system's response time to approximately 30ms, substantially improving performance and responsiveness.



The Camera Capture Service was assigned the highest priority with a 30ms period to maximize frame acquisition and minimize input latency. The Red Laser Detection Service followed with a 35ms period, responsible for detecting the laser dot using OpenCV-based image processing. The Direction Decision Service and Motor Control Service were scheduled with 40ms and 50ms periods, respectively, as they had lower WCETs and more predictable execution patterns.

All four services executed well within their assigned deadlines. The WCET measurements included the effects of interference from other tasks. We verified RM schedulability using the

Cheddar tool, confirming that the total processor utilization remained below the theoretical RM bound. This validated that our system was schedulable and feasible under the chosen configuration.

In addition to the real-time tasks on Core 1, we introduced two background services on Core 2:

- A Configuration Service (period: 2000ms) that allows dynamic tuning of behavioral parameters (e.g., reversing the robot's movement).

- A Hardware Watchdog Service (period: 2500ms) that monitors the health of critical services and triggers a reboot if any fail to report within the expected timeframe.

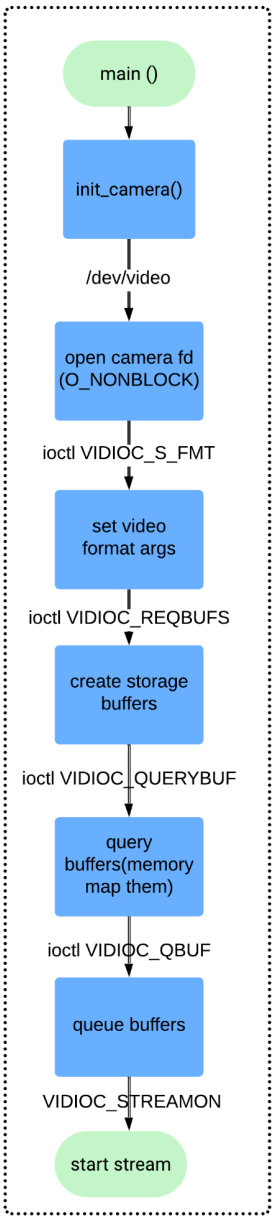
These two services do not follow the RM policy and operate at lower frequencies, ensuring they do not interfere with the real-time execution on Core 1

Detailed Design

Camera capture service:-

The camera capture service is the highest priority service in the system. The idea behind keeping this service as the highest priority is for having a smoother movement of the robot, more number of inputs are needed every second. The motor service itself has a very low execution time as compared to camera service.

By default opencv is **blocking** in nature which can jeopardize our real time system. To overcome this problem after some discussion with the instructor, **V4L2** (video for linux) apis are used to perform image capturing.



A setup process is required to get started with image capture and thus is mindfully kept in a separate function and not part of the main service. The essential steps are detailed in the block diagram on the left.

The process begins in the main() function, which initiates the camera setup by calling init_camera(). Inside this function, the camera device located at /dev/video is opened with the **O_NONBLOCK** flag, ensuring non-blocking behavior.

Once the device is successfully opened, the video format is configured by invoking the **VIDIOC_S_FMT ioctl** call, which requires specifying arguments such as resolution, pixel format, and field type.

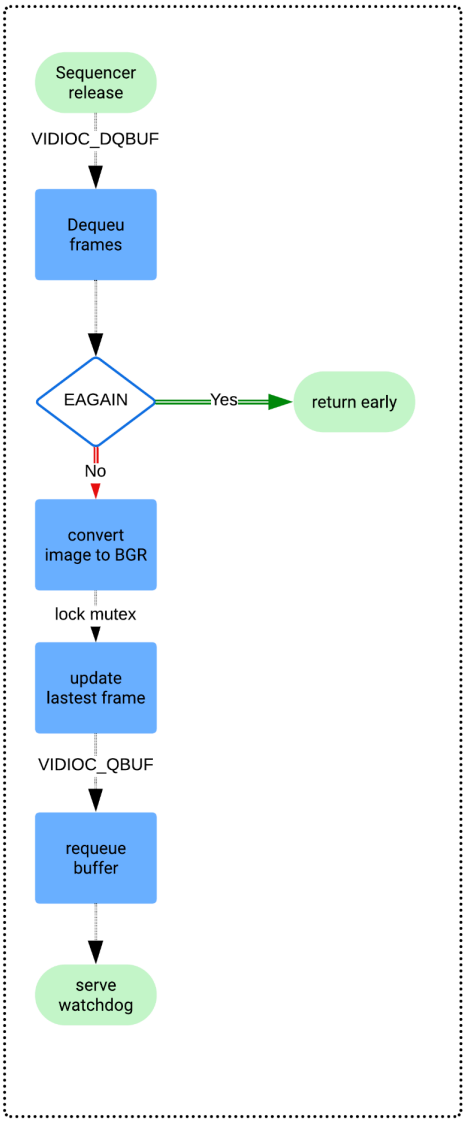
After setting the format, buffer allocation is requested using the **VIDIOC_REQBUFS** ioctl command.

Following this, the program queries the properties of each buffer using **VIDIOC_QUERYBUF**, which also provides the information needed to memory map the buffers into the user space.

Once the memory mapping is completed, the buffers are enqueued for use using the **VIDIOC_QBUF** ioctl call.

Finally, streaming is started with the **VIDIOC_STREAMON** command, signaling the camera to begin capturing and delivering video frames

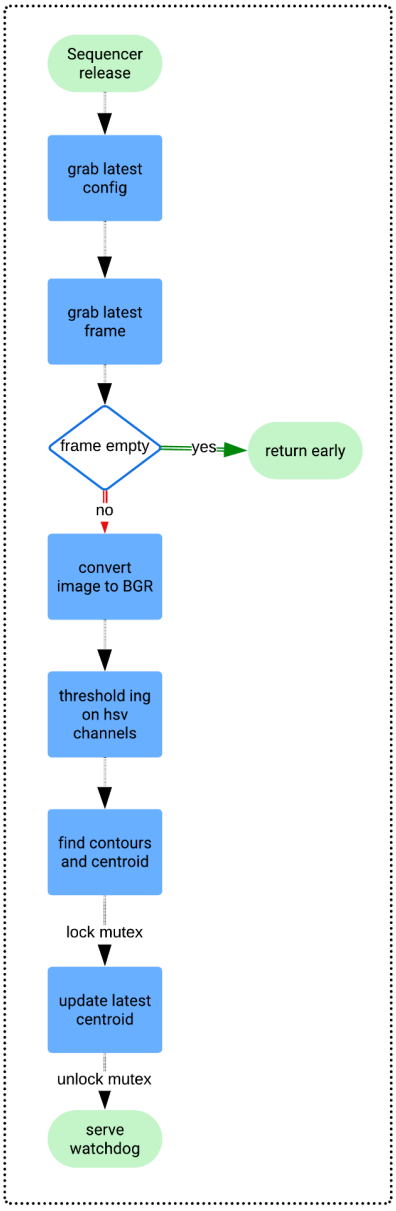
Next we will look at the core camera service.



When the sequencer releases the camera capture service, we first dequeue a frame
and check for image data. If we get the error code of EAGAIN that means there is no new data available and we return early to prevent blocking.

If an image is successfully dequeued we convert the image from v4l2 format to BGR format and then proceed to update a shared global data structure which is called latest frame (defined as a cv2::mat). To reuse the buffer slots we again requeue the buffer and finally service the watchdog to prevent a reset and indicate stable operation.

Red laser detection service:-



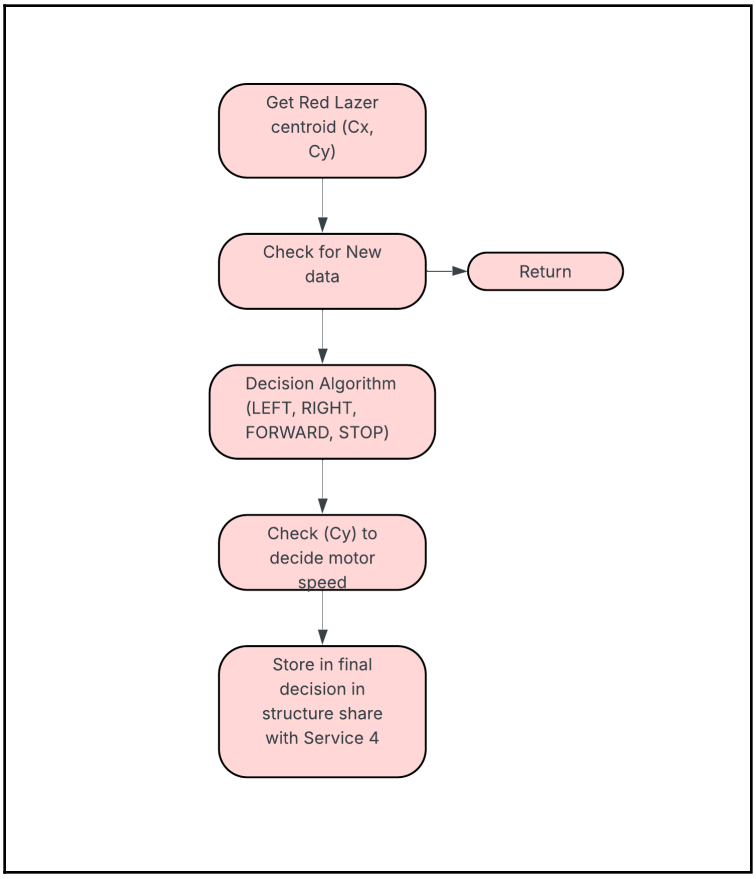
The logic of the red logic detection is derived from the following python repository . On the release of the service by the sequencer we first grab the latest config (updated via the config update service explained later). This contains necessary values for the threshold values of HSV(hue saturation and value).

Then we fetch the **latest frame**. If the frame is empty, the service exits early to save computation.Otherwise, the captured frame is converted to the **BGR color format**, then processed by applying **thresholding on HSV channels** by applying masking to isolate the red laser.

After this, **contours and centroids** are calculated from the thresholded image to detect objects or regions of interest. (source for calculating centroid). We then update the values of centroid Cx and Cy safely by locking a mutex.Finally, the system **serves a watchdog** and returns.

Motor Decision Service

Service 3: Direction Deciding is the central decision-making component of the robot's control architecture. Its primary function is to interpret the position of a detected red laser spot (provided by Service 2) and determine the appropriate movement behavior for the robot.



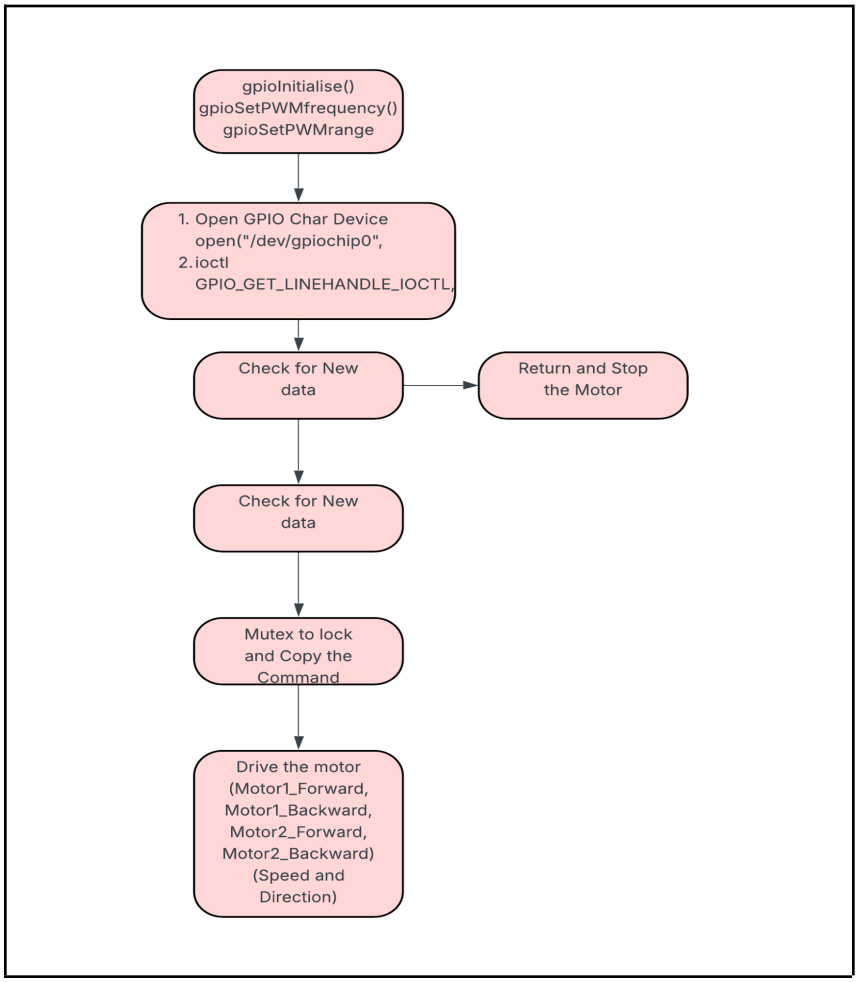
At every invocation, Service 3 checks whether a new laser point is available using the **atomic flag** point_available. If the flag is set, it safely acquires the latest detected laser coordinates (Point2D) using a **mutex-protected** shared variable (latest_laser_point). Once the data is retrieved, the flag is cleared to indicate that the point has been consumed, ensuring that the same point is not processed multiple times. The service then analyzes the (x, y) position of the laser within the camera frame to infer the robot's intended motion.

For example, if the x-coordinate lies to the left of the frame, the robot is instructed to turn left; if to the right, it turns right. If the laser lies near the center horizontally, the robot moves forward. The vertical position of the point (y-coordinate) is used to determine the **speed level**, with points near the top of the frame corresponding to higher speeds and those near the bottom corresponding to slower speeds - this effectively mimics human intent, where placing the laser farther away implies faster movement.

Once the movement decision is finalized, the service constructs a MovementCommand structure that includes the chosen direction (e.g., FORWARD, LEFT, RIGHT, STOP) and speed level (ranging from 1 to 3). Additionally, a behavior flag (Config_behave) may be set, allowing Service 4 to choose between different motor wiring or control strategies dynamically. The command is then stored into a shared variable latest_cmd using a mutex to ensure thread safety. An atomic flag cmd_available is set to indicate that a new command is ready for execution.

Motor Motion Service

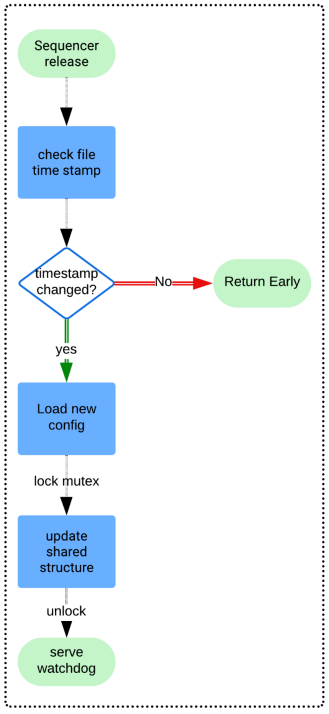
Service 4: Motor Control is responsible for translating high-level movement commands into low-level electrical signals that control the robot's motors. It receives directional and speed instructions from Service 3 (the decision-making service), and converts them into GPIO signals using a combination of digital output and PWM (Pulse Width Modulation).



Upon its first execution, Service 4 initializes the motor control subsystem. It opens a file descriptor to the Linux **GPIO character device /dev/gpiochip0** and requests control over six GPIO pins. These include four direction pins (IN1–IN4) and two enable pins (ENA and ENB) responsible for speed control. The **pigpio library** is used to configure PWM on the ENA and ENB pins, which allows precise control of motor speed through varying duty cycles. Each motor is assigned a **PWM frequency** of 800 Hz and a duty cycle range of 0–100%.

At each scheduled execution, the service checks whether a new movement command is available using an atomic flag **cmd_available**. If a command is present, it safely retrieves it using a mutex and clears the flag to prevent reprocessing. If no command is available, the robot is instructed to stop, and the motors are turned off by applying a zero duty cycle. Based on the command contents, the service determines both the direction (forward, left, right, stop) and speed level (1–3). These are then mapped to appropriate GPIO outputs and **PWM duty cycles** (70%, 80%, or 100%).

An important feature of this service is its support for dynamic driving behavior. The command structure includes a boolean field called **Config_behave**, which toggles between two different wiring logic styles for motor control to adapt the robot to different hardware setups. The selected direction and speed are then applied by calling the drive() method of the **MotorDriver class**, which sets the GPIO outputs using ioctl system calls and configures the PWM via gpioPWM()



Config Update Service

The config update service provides a nice control over the behaviour of the robot. If the config.json file is modified with a behaviour flag of 0 the robot repels the red laser light.

The config service works by checking the file stamp of the Config.json file to look for updates. If there are no updates the service returns early otherwise a new config is loaded and stored in a shared data structure which will be read by the laser detection service. This is more clearly demonstrated in the demo video linked in the appendix section.

Results and Analysis

The following section provides comments about the various results and observations related to the execution time services of the system.

Execution time observations

The service class is modified to collect logs related to max,min and average execution time and also stores these values in a ***unique .csv*** file for each service based on the ***service identifier.***

Service execution time results

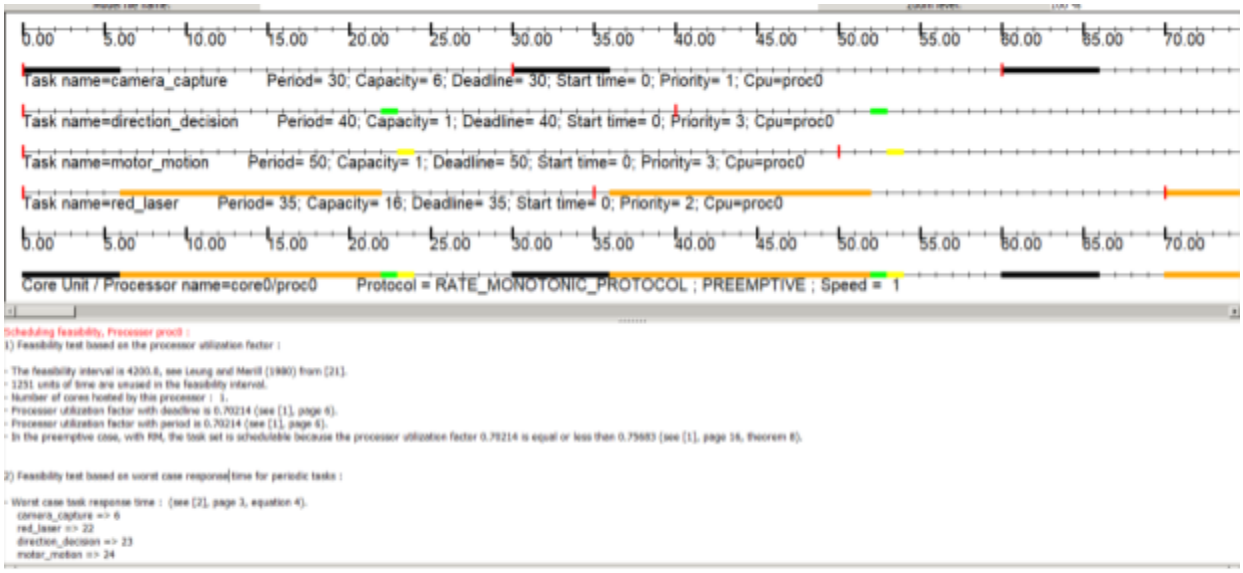
Service name	Priority	**Period**	**WCET**	**Avg exec time**
Camera Capture Service	98	30ms	5.909ms	0.826ms
Image processing Service	97	35ms	16.105ms	6.398ms
Direction Decision Service	96	40ms	0.34ms	0.012ms
Motor Motion	95	50ms	0.33ms	0.026ms

The utilization value computed is

U= 5.909/30 + 16.105/35 + 0.34/40 + 0.33/50
=0.6722095238

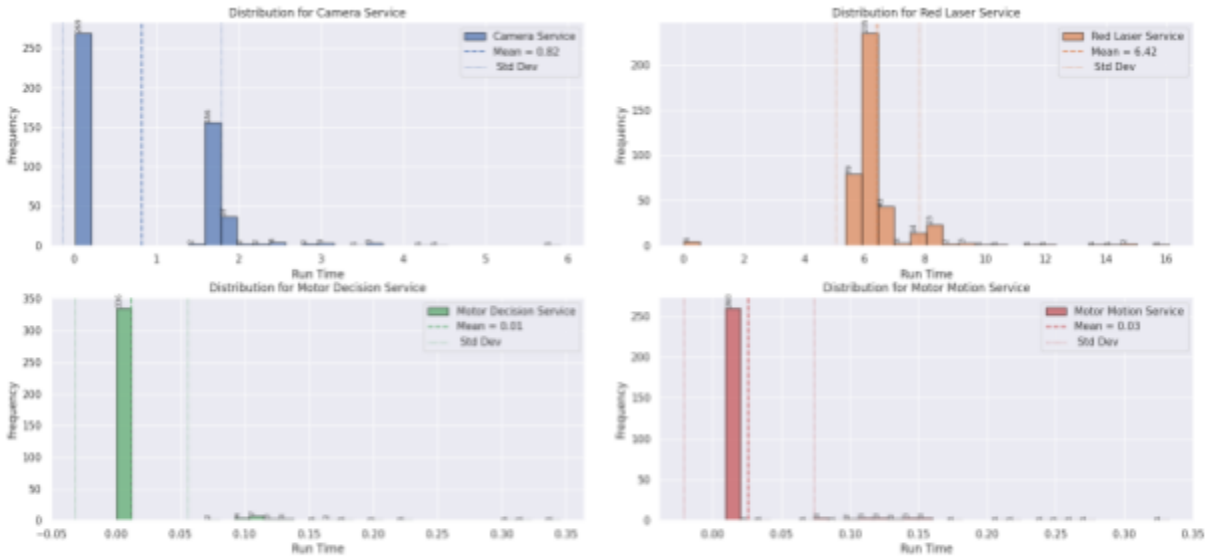
Hence the task service set is schedulable under RM policy as U<**0.75683.**

The above utilization calculation is also verified using cheddar however the utilization value given by cheddar is slightly higher than above since cheddar does not allow non-integer values for Computation time (***C***) and two of our services have C values below 1.



Cheddar Results

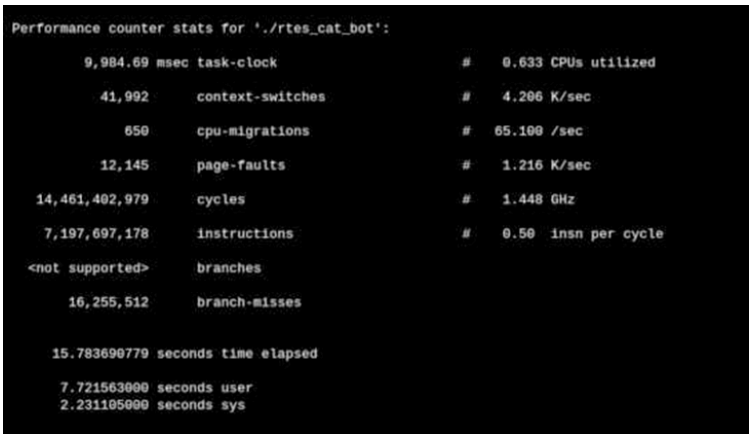
The collected csv logs are also analyzed post execution to generate histograms. The graphs help visualize the frequency of higher execution times. Mostly all the values are tightly clustered around the mean execution time and there are a very few occasions of anomalously higher execution time.

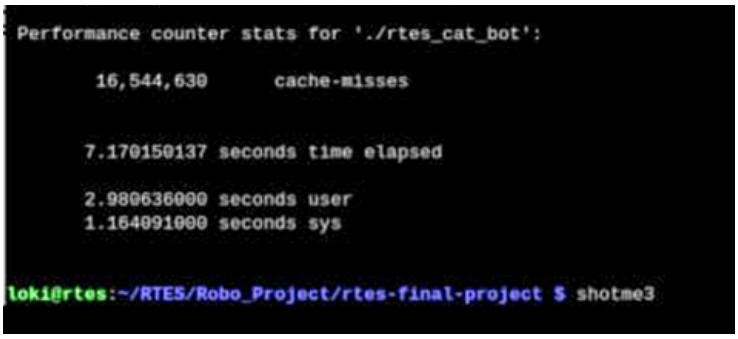


Execution time histograms

Performance Profiling with Perf:

To gain insight into how our robot system behaved at the system level, we utilized Linux's powerful profiling tool perf, specifically using the perf stat command. This tool allowed us to collect valuable performance counters such as execution time, CPU utilization, cache misses, branch mispredictions, page faults, and context switches. From our observations, while the number of cache misses and branch mispredictions appeared relatively high, we recognize that perf is best used for **comparative analysis** rather than absolute optimization metrics. The tool is especially useful in spotting **performance bottlenecks or regressions** when comparing different software versions or hardware setups.





Given the time constraints and our focus on system functionality and correctness, we did not invest significant effort into **micro-optimizations** (such as branch prediction tuning or cache alignment). However, using perf helped us understand how our multithreaded real-time services interacted with the underlying system resources. The knowledge gained from using perf has been valuable in learning how low-level system activity can impact real-time application performance.

Experimenting with Linux parameters

As an attempt to make the system more deterministic and robust we attempted to add linux tuning parameters specific to real time systems.

The following parameters were appended to ***/boot/firmware/cmdline.txt*** which is a config file to specify the Linux kernel command line parameters that are passed to the kernel during boot -

isolcpus=1 rcu_nocbs=1 nosoftlockup

Isolcpus=1- helps isolate core 1 where our essential RM services are executing. It prevents the kernel from scheduling general tasks on our designated core.

rcu_nocbs=1- Moves RCU (Read-Copy-Update) callback processing off CPU 1 and helps to keep isolated cores free of kernel interruptions.

Nosoftlockup - Disables the backtracking of tasks that run for longer than 120 seconds without yielding.

Post booting we also ensure to select the ***performance*** scaling governor which forces the cpu to run at max frequency at all times and prevents it from going into a power saving state which can cause non deterministic behaviour.

echo performance | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor

Some other attempts were also made to explore overclocking the rpi with the force_turbo=1 argument in boot parameters, however this resulted in massive overheating and poor performance and hence was reverted.

Future Work and improvements

After a discussion and feedback session we have identified the following areas which can be improved in the future-

- Instead of a contour based approach for laser detection we can explore a grid based approach to make the detection more robust and less computationally expensive.
- Experimenting with PWM frequency values according to hardware specification.
- Removing redundant mutexes in cases where there is only a single reader and writer.
- Instead of writing to a .csv file , we can utilize syslogs and grep to fetch execution times to avoid unnecessary file I/O.

Conclusion

Through this project, we successfully achieved all the goals we initially set out to accomplish. Our laser-following robot was able to respond smoothly and reliably, achieving a fast response time of approximately 30ms between laser dot detection and motor actuation, validated by analyzing the combined worst-case execution times (WCETs) of all four real-time services. This not only met but exceeded our original 100ms responsiveness target.

Throughout the development process, we explored and learned a wide range of real-time concepts. We understood how to set meaningful deadlines and priorities under Rate Monotonic Scheduling, effectively used OpenCV and V4L2 for image processing and non-blocking frame capture, implemented smooth motion via software-based PWM, and added robustness through a hardware watchdog. We also implemented a configuration service that dynamically updated parameters from JSON files during runtime. We also leveraged Linux system tuning tools such as perf and gdb, and applied kernel-level configurations like isolcpus and scaling governor

In addition, this project gave us deep exposure to various modern C++ concepts, which we plan to fully incorporate in future projects based on feedback and suggestions received. Overall, this experience was both technically enriching and challenging, significantly enhancing our understanding of real-time design concepts.

References

1. pigpio C Library Documentation - https://abyz.me.uk/rpi/pigpio/cif.html
2. Linux GPIO Interface (Character Device ABI) - https://www.kernel.org/doc/html/latest/driver-api/gpio/using-gpio.html
3. Raspberry Pi PWM Guide - https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#pulse-width-modulation-pwm
4. C++11 std::atomic and memory_order - https://en.cppreference.com/w/cpp/atomic/memory_order
5. Watchdog - https://developer.toradex.com/software/linux-resources/linux-features/watchdog-linux/
6. Laser detection - https://github.com/bradmontgomery/python-laser-tracker

Appendix

The code appendix is provided as a separate pdf file titled ***Code-appendix*** and as well .zip file is attached along with this submission.

Image processing - Camera_Service.cpp and Red_lazer_service.cpp
Decision making - Direction_decising.cpp
Motor Actuation - Motor_control.cpp
Watchdog - watchdog.cpp
Dynamic Configuration update (Moving reverse) - config_update.cpp

A link to a short demonstration of the robot is linked here.