## Project Report: Development and Deployment of a PREEMPT\_RT Kernel on Raspberry Pi 5

### 1. Objective

The primary objective was to deploy a real-time operating system (RTOS) or an OS with real-time capabilities on a Raspberry Pi 5. This system serves as the foundation for the "Real Time Monocular Depth Estimation on Edge AI" project, which requires a deterministic, low-latency environment to process camera input and trigger timely hardware alerts for a collision mitigation demonstrator. The kernel needed to be robust, support all on-board RPi 5 peripherals (especially camera, Ethernet, WiFi, and GPIO), and be capable of handling high-precision timing applications (NTP/PPS).

### 2. Final Successful Methodology (Summary)

After a rigorous process of trial, error, and refinement, the final successful methodology for creating the PREEMPT\_RT kernel was determined to be **native compilation on the Raspberry Pi 5**, based on the procedure outlined in the github.com/by/RT-Kernel guide. This approach involved:

1. Starting with a fresh, standard 64-bit Raspberry Pi OS (Debian Bookworm) installation.
2. Cloning the rpi-6.15.y kernel branch directly onto the Raspberry Pi 5.
3. Leveraging the integrated PREEMPT\_RT support within this kernel version, which eliminated the need for external patching.
4. Configuring the kernel via menuconfig to enable CONFIG\_PREEMPT\_RT and other performance and timing-specific options.
5. Building the kernel, modules, and device tree blobs natively on the device.
6. Deploying the new kernel to a separate boot directory (/boot/firmware/NTP/) and using config.txt parameters (os\_prefix, kernel) to boot from it, ensuring a safe, non-destructive update path.

### 3. Development and Troubleshooting Log

This section details the chronological journey, including all failures, attempted solutions, and the rationale for methodological shifts.

#### **Phase 1: Initial Cross-Compilation Approach (Failed)**

The initial strategy was to use a powerful Ubuntu Development PC to cross-compile the kernel for the Raspberry Pi 5, which is typically faster than native compilation.

* **Failure 1: Kernel Source and PREEMPT\_RT Patch Mismatch**
  + **Problem:** An attempt to use the latest rpi-6.6.y branch resulted in a kernel version (6.6.78) for which no matching official PREEMPT\_RT patch was available.
  + **Solution Attempted:** The strategy was pivoted to use the older but more established rpi-6.1.y branch, which yielded a source version of 6.1.93. However, we again faced a series of 404 Not Found errors when trying to download patches for the closest versions (patch-6.1.92-rt32.patch.xz, patch-6.1.91-rt31.patch.xz).
  + **What Worked (Partially):** We successfully located and downloaded a viable patch for a slightly older version (patch-6.1.90-rt30.patch.xz) from the kernel.org/older/ directory.
  + **What Didn't Work:** Relying on the head of Raspberry Pi's stable branches (.y branches) to align with available RT patches proved unreliable due to differing release cadences.
* **Failure 2: Git Tag Mismatch**
  + **Problem:** After successfully obtaining the 6.1.90-rt30 patch, an attempt to check out the corresponding 6.1.90 git tag in the Raspberry Pi kernel repository failed, as no such tag existed.
  + **Solution Attempted:** As a calculated risk, we applied the .90 patch to the .93 source code.
  + **Result (Successful):** This worked. The patch command applied the changes cleanly with only minor, acceptable offsets, demonstrating that patching closely-related versions was feasible.
* **Failure 3: The "No Boot" Issue**
  + **Problem:** Despite a successful cross-compilation and deployment of the patched 6.1.93-rt30 kernel, the Raspberry Pi 5 failed to boot (no network activity).
  + **Solution Attempted:** We attempted to revert to the backup of the original kernel (kernel8-original.img). This also failed to boot.
  + **Diagnosis:** The situation was critical. Upon inspecting the SD card, we identified the presence of multiple potential default kernel images (kernel8.img, kernel\_2712.img), realizing our revert process might have been incomplete. Suspecting potential file corruption or a subtle bootloader configuration issue, a robust recovery was needed.
  + **What Worked (Recovery):** The decision was made to **re-flash the microSD card with a fresh, standard Raspberry Pi OS image**. This was a crucial step that provided a clean, known-good baseline to move forward from.

#### **Phase 2: Pivot to Native Compilation (Successful Approach)**

Based on the challenges with cross-compilation and the desire for a tested method, we pivoted to a new guide provided by Linux.org.

* **Methodological Shift:** The new approach involved native compilation on the RPi 5 and using the rpi-6.15.y kernel branch, which claimed to have RT support integrated without needing an external patch.
* **Success:** This approach immediately resolved the versioning and patching headaches. The kernel configured and built successfully on the first attempt after all dependencies were met.

#### **Phase 3: Application Environment Troubleshooting**

With the new kernel running, a series of Python-related issues arose when setting up the demonstration application.

* **Failure 4: RPi.GPIO Library Failure**
  + **Problem:** The RPi.GPIO library failed with RuntimeError: Cannot determine SOC peripheral base address, likely due to incompatibility with the new custom kernel or RPi 5 revision.
  + **Solution Attempted:** The application script was modified to use the more modern and robust gpiozero library for controlling the RST and DC pins of the display and all other GPIOs.
  + **Result (Successful):** This immediately resolved the error, proving gpiozero to be a better choice for this platform.
* **Failure 5: ModuleNotFoundError for libcamera**
  + **Problem:** The Python script, running inside a virtual environment (tflite\_env), could not find the libcamera module, even though the python3-libcamera system package was installed via apt.
  + **Diagnosis:** We identified that the virtual environment was created with include-system-site-packages = false, isolating it from necessary system-level Python libraries.
  + **Solution Attempted:** The tflite\_env was cleanly removed and recreated with the --system-site-packages flag.
  + **Result (Successful):** This allowed the Python interpreter inside the virtual environment to see the system-installed libcamera and picamera2 packages, resolving the import errors.
* **Failure 6: GUI Display Error with sudo chrt**
  + **Problem:** Attempting to run the GUI application with real-time priority (sudo chrt ...) resulted in a qt.qpa.xcb: could not connect to display error.
  + **Diagnosis:** The sudo command switched the user context to root, which was not authorized to access the pi user's graphical desktop session.
  + **Solution Attempted:** The command was modified to sudo -E chrt ..., which preserves the user's environment, including X11 authorization variables. This was executed from a terminal running on the Raspberry Pi's own graphical desktop.
  + **Result (Successful):** The application launched correctly with its GUI while having the intended real-time priority.

### 4. Final System Specification

The project has resulted in a stable, validated, real-time capable system with the following characteristics:

**Hardware:**

* **Model:** Raspberry Pi 5 Model B Rev 1.1
* **CPU:** 2.4GHz quad-core 64-bit Arm Cortex-A76
* **RAM:** 8GB LPDDR4X-4267 SDRAM
* **Storage:** 64GB SanDisk microSD card
* **Cooling:** Official Active Cooler
* **Input:** Raspberry Pi Camera Module 3
* **Output:** HDMI Display, with alerts via a Red LED, Green LED, and an Active Buzzer on GPIO.

**Operating System & Kernel:**

* **OS Base:** Raspberry Pi OS (64-bit, Debian Bookworm base).
* **Kernel Version:** 6.15.0-rc7-v8-16k-NTP+
* **Kernel Type:** Fully Preemptible Real-Time Kernel (PREEMPT\_RT).
* **Key Configurations:** 1000 Hz timer, NO\_HZ\_FULL, performance CPU governor, PPS timing support enabled.
* **Validated Performance:** Demonstrated maximum scheduling latencies under 20 µs during heavy CPU stress and under 200 µs during heavy combined CPU and memory stress, validating its suitability for real-time tasks.

### Development & Deployment Log: PREEMPT\_RT Kernel on Raspberry Pi 5

#### 1. Objective

The primary objective was to replace the standard Raspberry Pi OS Linux kernel with a custom-compiled, 64-bit kernel featuring the PREEMPT\_RT patch. This aimed to create a deterministic, low-latency operating system foundation suitable for a real-time monocular depth estimation application requiring timely processing and immediate hardware responses for collision mitigation alerts.

#### 2. Initial Kernel Build Strategy & Challenges

The initial strategy involved a standard cross-compilation approach: selecting a recent kernel source from the official Raspberry Pi repository and applying a matching PREEMPT\_RT patch. This approach encountered several immediate failures.

* **Failure 1: Kernel & Patch Version Mismatch**
  + **Attempt:** We initially selected the rpi-6.6.y branch from the Raspberry Pi Linux repository, which provided a 6.6.78 kernel version.
  + **Problem:** Upon checking the official kernel.org archives, no PREEMPT\_RT patch was available for this specific version. The RT patch development lagged significantly behind the fast-moving stable branch of the RPi kernel.
  + **Conclusion:** This approach was unworkable. A strategy of picking the kernel source *first* and then finding a patch was unreliable.
* **Failure 2: Unavailability of "Close-Enough" Patches**
  + **Revised Strategy:** We pivoted to using the rpi-6.1.y branch (which provided a 6.1.93 kernel source) and attempted to find a patch for a very close version, as this is a common workaround.
  + **Problem:** Attempts to download patches for 6.1.92 and 6.1.91 via wget resulted in 404 Not Found errors, indicating the files were no longer in the main kernel.org directory.
  + **Solution Attempt:** We located a patch for 6.1.90 in the older/ directory on kernel.org and successfully downloaded it.
* **Failure 3: Mismatch Between Patch Version and Git Tags**
  + **Strategy:** Having secured the 6.1.90-rt30 patch, the next logical step was to check out the exact 6.1.90 kernel source from the Raspberry Pi git repository to ensure a perfect match.
  + **Problem:** The command git tag -l | grep 6.1.90 returned no results. Despite fetching all repository tags, the Raspberry Pi foundation did not have a specific git tag for this mainline version, making a perfect source/patch match difficult.

#### 3. Successful Kernel Build Methodology

A final, pragmatic approach was adopted which proved successful.

1. **Source & Patch:** We proceeded with a calculated risk: applying the **patch-6.1.90-rt30.patch** to the slightly newer **6.1.93 kernel source** from the rpi-6.1.y branch. This step succeeded without any rejected hunks.
2. **Native Compilation:** Based on a selected community guide (github.com/by/RT-Kernel), we switched from cross-compilation to **native compilation**, performing all configuration and building directly on the Raspberry Pi 5.
3. **Meticulous Kernel Configuration:** Using make menuconfig, the kernel was configured with the following key real-time and performance options:  
   * **Preemption Model:** Fully Preemptible Kernel (Real-Time) (CONFIG\_PREEMPT\_RT=y)
   * **Timer Frequency:** 1000 Hz (CONFIG\_HZ\_1000=y)
   * **Tickless System:** Full dynticks system (tickless) (CONFIG\_NO\_HZ\_FULL=y)
   * **CPU Governor:** Default set to performance.
   * **NTP/PPS Support:** Enabled for future high-precision timing applications.
   * **Debugging:** All kernel-level memory and object debugging features were disabled to reduce overhead.
   * **Drivers:** Key drivers, including brcmfmac for WiFi, were verified to be enabled as modules.
4. **Successful Build:** The kernel (6.15.0-rc7 based on the .config, with local version -v8-16k-NTP+) was successfully compiled natively on the Raspberry Pi 5.

#### 4. Deployment, Boot Failures, and System Recovery

The deployment phase proved to be the most challenging, resulting in a non-bootable system that required a full recovery.

* **Failure 4: First Boot with RT Kernel Fails**
  + **Attempt:** The newly compiled kernel image (kernel8-rt.img), modules, and device tree files were manually copied to the live SD card's boot and root partitions, and config.txt was updated.
  + **Problem:** Upon reboot, the Raspberry Pi 5 failed to boot successfully. It received power (indicated by the fan running), but never appeared on the network (Ethernet or WiFi), making SSH access impossible. Without a serial console, the system was effectively bricked.
* **Failure 5: Reverting to Backup Kernel Also Fails**
  + **Solution Attempt:** To recover the system, the SD card was mounted on the Ubuntu laptop. The backup of the original kernel (kernel8-original.img) was restored to kernel8.img, and the config.txt file was edited to remove the custom kernel entry.
  + **Problem:** Even after restoring the original kernel files, the Raspberry Pi 5 **still failed to boot**. This pointed to a more serious issue than just a bad kernel image, such as a subtle config.txt error, a problem with the overwritten device tree blobs, or filesystem corruption.
* **Root Cause & Solution: System Re-Flash**
  + **Analysis:** The boot partition contained multiple kernel images (kernel8.img, kernel\_2712.img), indicating a more complex default boot process than initially assumed. The manual file copy and revert process likely left the boot partition in an inconsistent state.
  + **Successful Solution:** The only reliable path forward was to **re-flash the microSD card with a fresh, clean image of Raspberry Pi OS ("Full" 64-bit)** using the Raspberry Pi Imager. This guaranteed a known-good starting point.

#### 5. Successful Deployment & Post-Boot Troubleshooting

With a fresh OS as the base, the *same compiled kernel artifacts* were deployed again, this time following the safer, side-by-side methodology from the chosen guide.

* **Deployment:** The RT kernel and its files were copied to a separate directory structure (/boot/firmware/NTP/) and activated via os\_prefix and kernel= directives in config.txt.
* **Success:** The Raspberry Pi 5 booted successfully with the PREEMPT\_RT kernel. This was verified with uname -a.
* **Failure 6: WiFi Not Connecting**
  + **Problem:** Although the brcmfmac driver and firmware loaded correctly (seen in dmesg), the wlan0 interface was DOWN and not connected.
  + **Solution:** Using nmcli, it was determined that the pre-configured WiFi profile was not active. An explicit sudo nmcli device wifi connect "SSID" password "PASSWORD" command successfully activated the connection.
* **Failure 7: GUI Application Fails from SSH**
  + **Problem:** Running the OpenCV-based full\_demo\_app.py via SSH with sudo chrt resulted in qt.qpa.xcb: could not connect to display and Authorization required errors.
  + **Solution:** The issue was diagnosed as sudo running as the root user, which lacked permission to access the pi user's graphical desktop session. This was solved by:
    1. Running the script from a terminal window opened directly on the Raspberry Pi's graphical desktop.
    2. Using sudo -E to preserve the pi user's environment variables (DISPLAY, XAUTHORITY) when using chrt.

#### 6. Final Validated Kernel Details

The final, working system is configured as follows:

* **Hardware:** Raspberry Pi 5 (8GB RAM), RPi Camera Module 3, Active Cooler, HDMI Display.
* **Operating System:** Raspberry Pi OS (64-bit, Debian Bookworm base).
* **Kernel Version:** 6.15.0-rc7-v8-16k-NTP+
* **Kernel Type:** PREEMPT\_RT (Fully Preemptible Kernel).
* **Architecture:** aarch64.
* **Validated Max Latency (Idle):** ~15µs
* **Validated Max Latency (Heavy CPU + Memory Load):** Maintained below 200µs.
* **Application Status:** A full demonstration application is running with real-time priority (SCHED\_FIFO, priority 75), integrating live camera input, OpenCV-based heuristic detection, and multi-threaded GPIO alerts, with visual feedback on an HDMI display.

### System Analysis: Custom PREEMPT\_RT Platform for Real-Time AI

#### 1. Hardware Platform Summary

The current system is built upon the following hardware components:

* **Compute:** Raspberry Pi 5 Model B (8GB RAM), featuring a 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU.
* **Cooling:** Official Raspberry Pi Active Cooler.
* **Storage:** 64GB SanDisk microSD card.
* **Input:** Raspberry Pi Camera Module 3.
* **Output (Visual Feedback):** Standard HDMI Display connected via a micro-HDMI port.
* **Output (Alerts):**
  + High-brightness Red and Green LEDs connected to GPIO pins.
  + Active Buzzer connected to a GPIO pin for audio alerts.

#### 2. The Baseline: Stock Raspberry Pi OS

A standard Raspberry Pi OS installation provides a user-friendly, general-purpose operating system based on Debian.

* **Kernel:** It uses a standard Linux kernel configured for general desktop and server use. Its default preemption model (CONFIG\_PREEMPT\_VOLUNTARY) allows the kernel to be interrupted at convenient points, which balances throughput and responsiveness for everyday tasks like web Browse or running a file server.
* **Strengths:** Excellent out-of-the-box hardware support, a massive software repository (apt), and a large community.
* **Weakness of our Problem Statement:** **Non-Deterministic Scheduling.** In a general-purpose OS, there are no strict guarantees about *when* a task will get to run. A critical process (like our alert logic) could be delayed by an unpredictable amount of time (from microseconds to many milliseconds) by a lower-priority but non-preemptible kernel task, a long-running interrupt handler, or other system background activity. This delay, known as **jitter**, is unacceptable for a safety-critical collision mitigation system where alerts must be timely and predictable.

#### 3. The Current Enhanced System: Custom PREEMPT\_RT Platform

we have successfully transformed the baseline system into a highly deterministic platform by replacing the standard kernel.

* **Operating System Base:** Raspberry Pi OS (64-bit, Debian Bookworm). This retains the user-friendliness and software availability of the standard OS.
* **Custom "RTOS" Kernel Details:**
  + **Version:** 6.15.0-rc7-v8-16k-NTP+
  + **Architecture:** aarch64 (64-bit)
  + **Core Feature:** **PREEMPT\_RT (Fully Preemptible Kernel)**. This is the key modification. The PREEMPT\_RT patch fundamentally changes the kernel's behavior by making almost all parts of it preemptible. This dramatically reduces the sources of unpredictable latency.
  + **Key Real-Time Configurations Enabled:**
    - **1000 Hz Timer (CONFIG\_HZ\_1000):** Provides the kernel scheduler with a finer time resolution (1ms ticks), allowing for more precise scheduling and shorter response times.
    - **Full Dynamic Ticks (CONFIG\_NO\_HZ\_FULL):** Instructs the kernel to stop the periodic scheduler "tick" on CPU cores that are dedicated to running a single application. This eliminates a major source of jitter for real-time tasks.
    - **Performance CPU Governor (CONFIG\_CPU\_FREQ\_DEFAULT\_GOV\_PERFORMANCE):** Locks the CPU at its maximum frequency (2.4GHz). This eliminates latency caused by the CPU needing to ramp up its speed, ensuring consistent processing power for our application.
    - **Kernel Debugging Disabled:** Features like memory debugging were disabled during compilation to remove unnecessary overhead and further reduce potential sources of latency.

#### 4. Comparison: Why our Current System is Better for the Problem Statement

our custom PREEMPT\_RT system is superior to the stock OS specifically because it directly addresses the "Real Time" requirement of our project in three critical areas:

**1. Predictability and Timeliness of Alerts:**

* **Stock OS:** When our depth model detects an object is too close, the command to activate the buzzer and flash the LED is just another task for the scheduler. It could be delayed by system maintenance, network activity, or other processes, making the alert's timing inconsistent. A 50ms delay might be acceptable sometimes, but a 200ms delay at a critical moment could be the difference in a collision scenario.
* **our RT System:** The PREEMPT\_RT kernel, combined with running our application at a high priority (chrt -f 75), ensures that when our script decides to trigger an alert, that task can preempt almost any other process on the system to execute immediately. Our cyclictest results proved this: even under extreme CPU and memory load, the maximum scheduling latency remained **below 200 microseconds (0.2 ms)**. This means the time from our application's software decision to the GPIO pin state change is exceptionally fast and, more importantly, **highly predictable**.

**2. Consistency of the Data Processing Pipeline:**

* **Stock OS:** our main application loop (capture -> process -> display) could experience stuttering or inconsistent frame rates (FPS). A background task could momentarily take CPU time, causing our application to miss a processing deadline, resulting in a variable "Loop Time."
* **our RT System:** By prioritizing our full\_demo\_app.py, ou ensure it gets consistent access to the CPU. This leads to more stable loop times and FPS. For a system that needs to perceive the world continuously, this consistency is crucial for providing a reliable stream of depth information to the operator.

**3. Robustness Under Load:**

* **Stock OS:** A general-purpose system under heavy load can become unresponsive, and there's no guarantee that critical tasks will be serviced before non-critical ones.
* **our RT System:** Our stress tests proved that even with all CPU cores at 100% and 4GB of RAM under active stress, the kernel's ability to schedule a high-priority task quickly was not compromised significantly. This robustness means our collision alert system is much more likely to function correctly even if other system components are unexpectedly busy.

In conclusion, while the base OS and hardware are the same, we have fundamentally re-engineered the core scheduler of the operating system. we have moved from a general-purpose system that offers good *average* performance to a specialized real-time platform that provides **guarantees on worst-case performance and predictability**, which is the essential requirement for any safety-related application like a collision mitigation system.