ECEN 5623 - Real Time Embedded Systems Final Project

Computer Vision Synchronome

Gerritt Luoma - July 29, 2025

Introduction

In embedded systems where timing precision is critical, the design and implementation of real-time computer vision applications present a unique set of challenges. This paper explores the development of a computer vision synchronome tasked with capturing images of two distinct clock types—an analog clock and a digital clock—each with strict image acquisition requirements. The analog clock demands precise capture of its unique seconds hand without duplicate or transitional frames, while the digital clock requires consistent image captures every one-tenth of a second, ensuring that no digits are missed, duplicated, or partially displayed. The system operates on a Raspberry Pi 4 running embedded Linux, requiring careful integration of real-time scheduling principles to meet deterministic timing guarantees. This paper details the engineering approach, system architecture, and real-time software techniques used to ensure that image captures are synchronized to each clock's behavior, meeting all rate-monotonic scheduling constraints and upholding the integrity of the captured data.

System Design Overview and Functional Requirements

The computer vision synchronome system is designed to operate as a real-time embedded application on a Raspberry Pi 4, executing on Raspberry Pi OS, a Debian-based distribution tailored for the hardware. This environment provides a Linux-based platform that supports the necessary low-level hardware access and real-time features required for deterministic image acquisition and processing.

High-Level Architecture

At a high level, the system consists of a camera input pipeline, image processing with real-time tick detection, and a dedicated image conversion and saving subsystem. The application is implemented using POSIX real-time threads, scheduled under the SCHED_FIFO policy to provide strict priority-based preemption. All threads are carefully assigned to CPU cores to maximize determinism and throughput. Specifically, the threads responsible for image capture and tick detection are pinned to CPU core 0, ensuring tight control over real-time timing constraints. Meanwhile, image-saving operations—which are not time-critical—are assigned to CPU core 1 to isolate file system latency and I/O delays from interfering with capture performance.

The system interfaces with a 30 frames-per-second (fps) camera, specifically the Logitech C270 HD Webcam, using the Video4Linux2 (V4L2) API, a widely supported library for handling media devices under Linux. The camera captures frames in YUYV format, which provides a compromise between image quality and bandwidth for real-time capture. Each frame is converted to either the P6 (RGB) PPM format or the P5 (grayscale) PGM format, depending on the user's configuration. Optionally, a brightening algorithm can be applied prior to format conversion to satisfy an optional requirement of performing additional processing of the frames before saving to flash.

Functional Requirements

The core functionality of the system is to reliably capture frames of a clock face—either analog or digital—at specific time intervals without any duplication, omission, or transitional blur. The system supports two operational modes:

1. 1Hz Mode (Analog Clock Mode):
    a. Captures exactly 1801 unique frames of the analog clock's seconds hand over a 30-minute period.
    b. Each frame must correspond to a discrete, non-transitional second tick.
    c. No duplicate, missed, or transitional images of the second hand are allowed.
2. 10Hz Mode (Digital Clock Mode):
    a. Captures exactly 1801 unique frames representing every 1/10th of a second over a 3-minute period.
    b. Each image must contain a fully settled and readable 1/10th-second digit.
    c. Any missed or partially transitioned digit frames are considered a failure.

To meet these stringent real-time requirements, the image capture and tick detection tasks follow a Rate Monotonic Scheduling (RMS) strategy. The tick detection service (whether detecting a change in the analog second hand or monitoring digital time updates) receives images captured by the image capture service operating at a frequency at or above the Nyquist Rate of the running requirement, ensuring synchronization with the actual clock tick rather than relying solely on wall-clock timing. This mitigates the effects of scheduling jitter and guarantees alignment with real-world events.

Runtime Configuration

The system is configured at runtime via three command-line arguments:

1. Clock Mode Selector:
    a. 0 selects analog clock mode (1Hz capture).
    b. 1 selects digital clock mode (10Hz capture).
2. Image Format Selector:
    a. 0 saves frames in RGB format as .ppm files (P6).
    b. 1 saves frames in grayscale format as .pgm files (P5).
3. Brightening Toggle:
    a. 0 disables the brightening algorithm.
    b. 1 enables brightening before conversion and storage.

This configuration flexibility allows the system to adapt to varying operational conditions and clock types, while maintaining a consistent internal structure optimized for real-time operation. The following sections describe the detailed implementation of each service and evaluate the system's real-time performance under both modes of operation.

Real-Time Requirements and Design

To meet the strict timing and determinism constraints necessary for accurate image acquisition, the system is designed as a fully real-time application using POSIX threads (pthreads) with the SCHED_FIFO scheduling policy. This fixed-priority, first-in-first-out scheduler ensures that the highest-priority ready thread always runs, allowing deterministic response times and eliminating time-sharing delays that could jeopardize timing guarantees.

Thread Scheduling and Priorities

Each core component of the system—image capture, tick detection, frame saving, and a timing sequencer—is implemented as a dedicated real-time thread. These threads are statically prioritized to reflect the rate-monotonic principle: higher frequency services are given higher priority.

Sequencer Thread (Priority Max, Core 0):
The sequencer operates as the heartbeat of the system, executing at a frequency of 60 Hz, which is the least common multiple (LCM) of all service frequencies in both operating modes (1 Hz and 10 Hz). It is triggered by a high-resolution POSIX timer and is assigned the highest real-time priority. The sequencer is responsible for waking each of the service threads by posting to their semaphores according to their defined rates, ensuring consistent periodic execution.

Image Capture Thread (Priority Max-1, Core 0):
The image capture thread is responsible for fetching frames from the camera via the V4L2 interface. In 1 Hz mode, this thread runs at 4 Hz, and in 10 Hz mode, it runs at 30 Hz. These frequencies were selected to exceed the Nyquist rate for the respective target tick frequencies, ensuring that the system can reliably acquire non-transitional, blur-free images. The capture thread holds the second-highest priority to minimize latency between sequencer wake-up and camera access.

Tick Detection Thread (Priority Max-2, Core 0):
The tick detection thread analyzes captured frames to determine if a new clock tick (second or tenth of a second) has occurred. This thread runs at 4 Hz in 1 Hz mode and 20 Hz in 10 Hz mode, again satisfying the Nyquist criterion for accurate temporal detection. It is assigned the third-highest real-time priority to ensure timely analysis of frames without interfering with capture operations.

Image Saving Thread (Unprioritized, Core 1):
The frame-saving thread is decoupled from the real-time core of the system by being pinned to CPU core 1, isolating it from contention with the timing-critical threads on core 0. This thread runs at best-effort priority and writes processed images (PPM or PGM) to the file system. Because the image queue is buffered and bounded, delayed or intermittent saving operations do not back-propagate latency into the real-time threads.

The real-time definitions for each service based on configuration can be found in Table 1 below. The period and deadlines are defined in terms of the number of timing iterations from the sequencer, not in milliseconds. To convert from periods to milliseconds, multiply the values by 100/60. More analysis of the system will be found later in the document detailing the Worst Case Execution Time (WCET), jitter, etc.

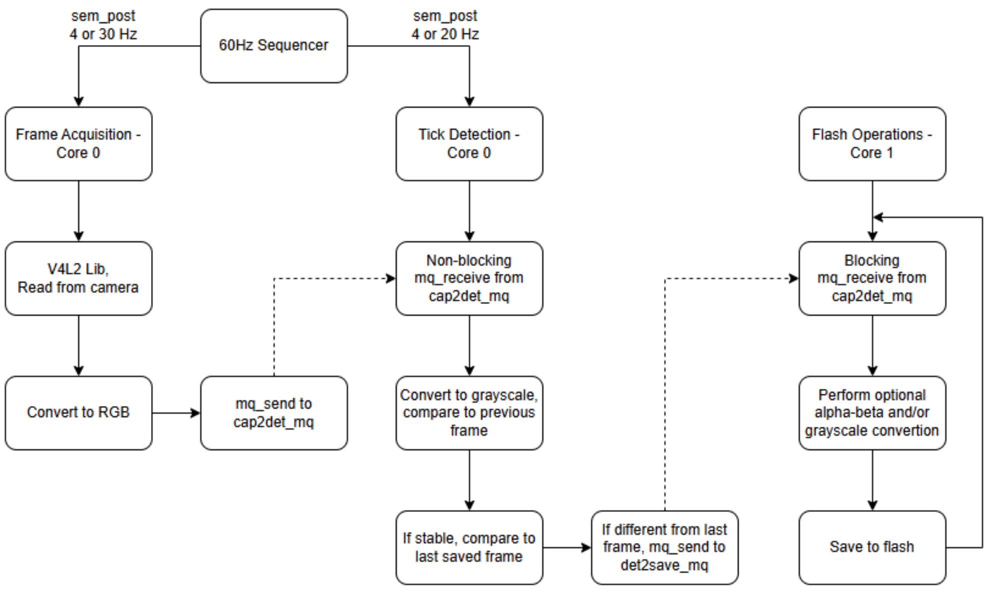
Table 1 - Real-Time Definitions

Configuration	Service	Period	Deadline	Priority	Core
1 Hz	Capture	15	15	Max - 1	0
	Detect	15	15	Max - 2	0
	Save	Undefined	Undefined	Undefined	1
10 Hz	Capture	2	2	Max - 1	0
	Detect	3	3	Max - 2	0
	Save	Undefined	Undefined	Undefined	1

Since the amount of processing being done in the capture and detection threads remains the same regardless of chosen configuration, the 1Hz configuration simply gains available laxity. This also means if the 10 Hz configuration is schedulable, the 1 Hz configuration is also guaranteed to be schedulable.

Functional Design

System Design



sem_post 4 or 30 Hz
60Hz Sequencer
sem_post 4 or 20 Hz
Frame Acquisition - Core 0
V4L2 Lib, Read from camera
Convert to RGB
mq_send to cap2det_mq
Tick Detection - Core 0
Non-blocking mq_receive from cap2det_mq
Convert to grayscale, compare to previous frame
If stable, compare to last saved frame
If different from last frame, mq_send to det2save_mq
Flash Operations - Core 1
Blocking mq_receive from cap2det_mq
Perform optional alpha-beta and/or grayscale convertion
Save to flash

Diagram 1 - System Design

The system operates as a coordinated set of real-time services driven by a high-priority sequencer thread, which is triggered by timer signals at a fixed 60 Hz rate as defined in Diagram 1. On each tick, the sequencer evaluates whether to post to each service's semaphore based on their configured frequencies. When activated, the image capture service acquires a frame from the V4L2 camera and immediately converts it to RGB format before enqueuing it to the tick detection service via a message queue. The tick detection service processes each incoming frame by first checking for frame stability against the previously received frame; if stable, it compares the current frame to the last saved frame. If a meaningful change is detected—indicating a new second or tenth-of-a-second value—it forwards the frame to the image saving service. The saving service, running on a separate CPU core, performs optional post-processing steps such as applying an alpha-beta filter for brightness correction and/or converting the image to grayscale before writing it to flash storage.

Sequencer Design

The sequencer is the central timing mechanism of the system, triggered by a 60 Hz periodic timer signal and executed with the highest real-time priority. At the start of each invocation, it increments an internal period counter to track elapsed intervals. Based on the current counter value and the configured service frequencies, it conditionally invokes sem_post() to release the semaphores for the frame capture and tick detection services at their appropriate rates. If an abort flag is set—either due to completion of frame capture or an external termination condition—the sequencer will immediately post to both service semaphores and propagate the abort state by setting their respective flags, ensuring a clean and synchronized shutdown. This behavior can be seen in Diagram 2 below:

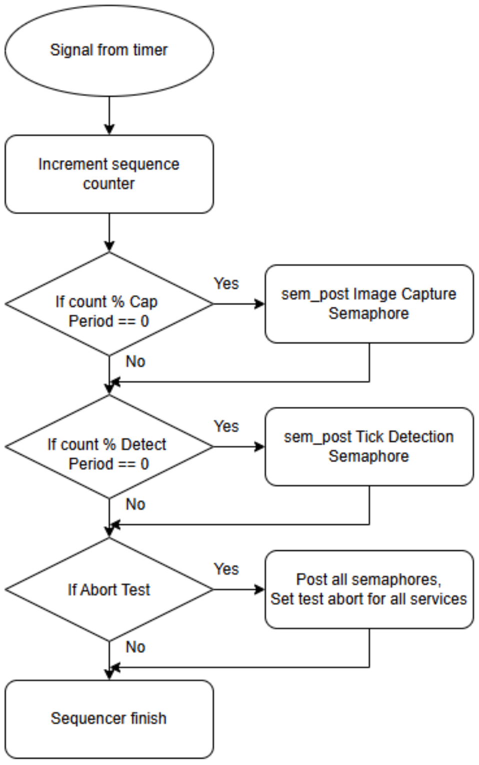

Signal from timer
Increment sequence counter
If count % Cap Period == 0
Yes
sem_post Image Capture Semaphore
No
If count % Detect Period == 0
Yes
sem_post Tick Detection Semaphore
No
If Abort Test
Yes
Post all semaphores, Set test abort for all services
No
Sequencer finish

Diagram 2 - Sequencer Flowchart

Frame Capture Design

The frame capture service is responsible for acquiring images using the V4L2 camera interface. When triggered, it dequeues the next available camera buffer and converts the YUYV-formatted frame into RGB format. The converted frame, along with a high-resolution real-time timestamp, is sent to the tick detection service via a message queue for further processing. After sending the frame, the V4L2 buffer is re-queued to allow for reuse in future captures. If the service's abort flag has been set, the frame capture thread exits gracefully and returns NULL to signal the end of its operation. This process is portrayed below in Diagram 3:

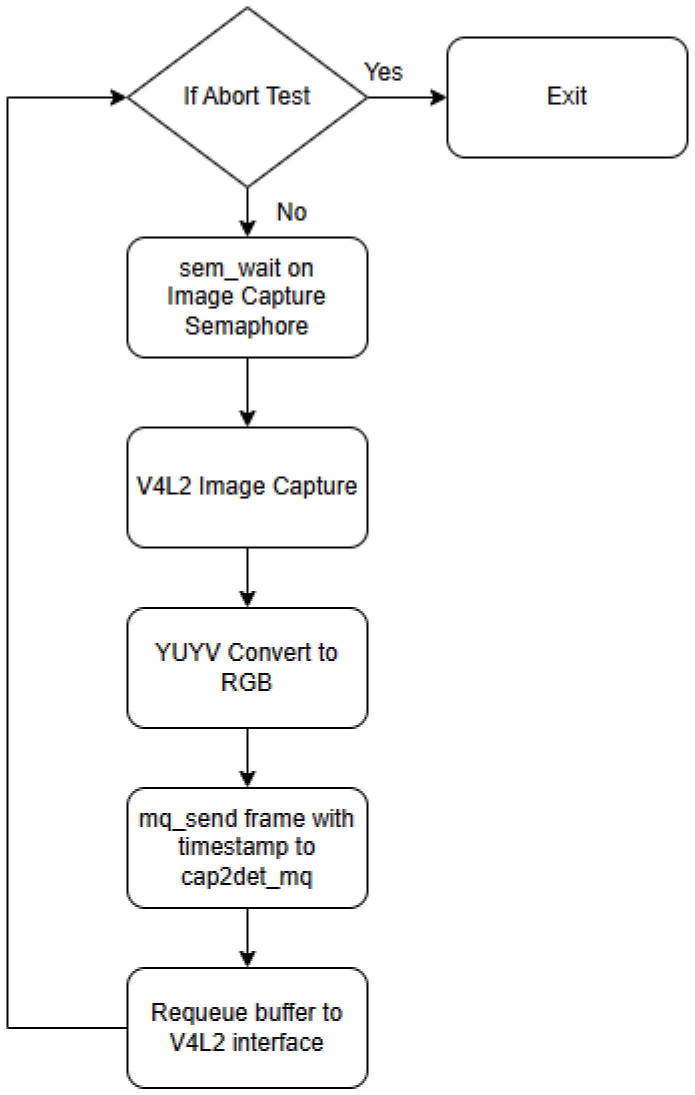
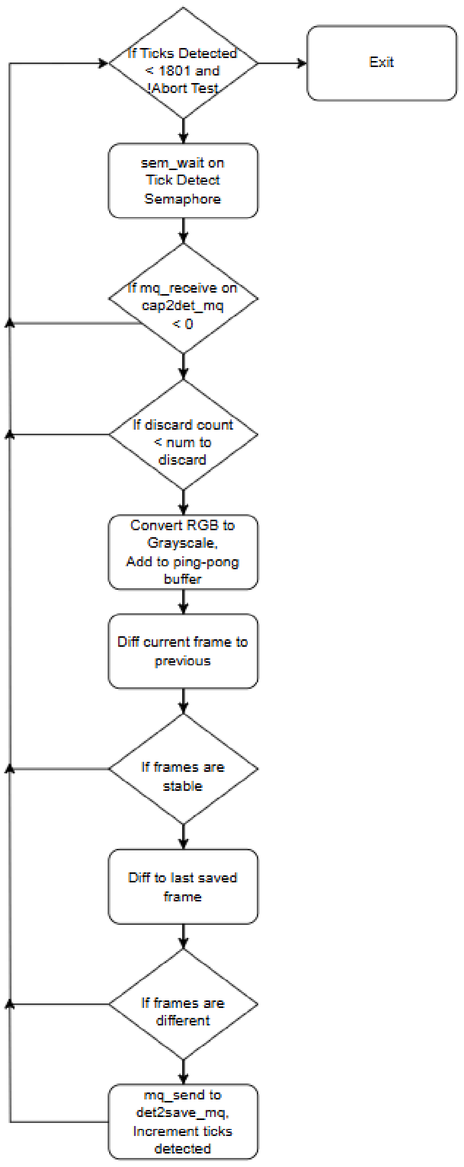

If Abort Test
Yes
Exit
No
sem_wait on Image Capture Semaphore
V4L2 Image Capture
YUYV Convert to RGB
mq_send frame with timestamp to cap2det_mq
Requeue buffer to V4L2 interface

Diagram 3 - Image Capture Flowchart

Tick Detection Design

The tick detection service begins each iteration by performing a non-blocking read from the message queue that holds frames from the capture service. If a frame is available, it is loaded into a ping-pong buffer to retain both the current and last received frames. If no frame is present, the loop breaks and the service waits on its semaphore again. The service discards a configurable number of initial frames to allow for camera stabilization. Each frame is converted to grayscale before computing a difference against the previous frame to check for tick stability. If stable, the new frame is compared against the last saved frame to detect visual changes indicating a clock tick. If a significant difference is found, the frame is marked with a time of detection timestamp and sent to the saving service via another message queue. This loop continues until no more frames are available in the message queue leading to the service once again waiting on its semaphore. Once 1801 frames have been saved, the service sets the system abort flag and exits. This process is visualized in Diagram 4:



If Ticks Detected < 1801 and !Abort Test
Exit
sem_wait on Tick Detect Semaphore
If mq_receive on cap2det_mq < 0
If discard count < num to discard
Convert RGB to Grayscale, Add to ping-pong buffer
Diff current frame to previous
If frames are stable
Diff to last saved frame
If frames are different
mq_send to det2save_mq, Increment ticks detected

Diagram 4 - Tick Detection Flowchart

Frame Saving Design

The frame saving service performs a blocking read from the message queue that holds validated frames from the tick detection service. If the optional alpha-beta correction setting is enabled, the service applies a brightness and contrast adjustment algorithm to the RGB data. If the grayscale setting is active, the frame is converted from RGB to grayscale before saving. The frame is then written to the file system in either .ppm (P6 RGB) or .pgm (P5 grayscale) format, depending on the runtime configuration. This process continues until exactly 1801 frames have been saved, at which point the service exits its main loop and terminates cleanly.

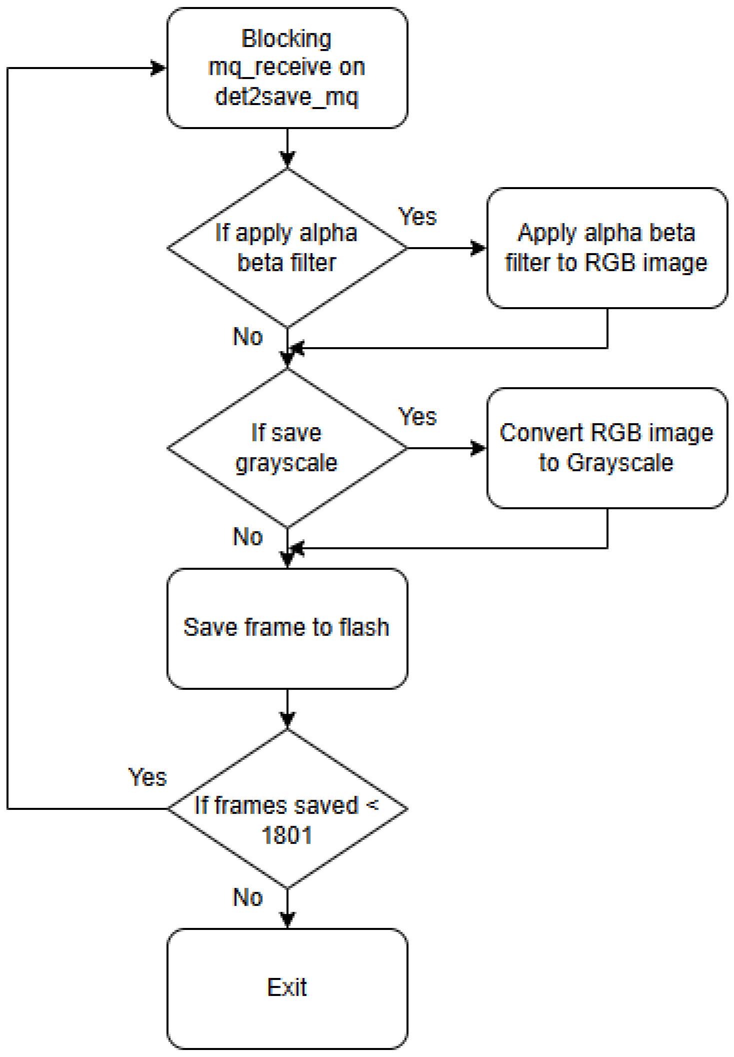

Blocking mq_receive on det2save_mq
If apply alpha beta filter
Yes
Apply alpha beta filter to RGB image
No
If save grayscale
Yes
Convert RGB image to Grayscale
No
Save frame to flash
Yes
If frames saved < 1801
No
Exit

Diagram 5 - Image Saving Service

Real-Time Analysis

To evaluate the system's real-time performance, execution tracing was performed using the Linux system log (syslog) facility. Each real-time service—Frame Capture (CAPTURE), Tick Detection (DETECT), and the best-effort Frame Saving (SAVE) service—emits structured trace logs at key points during execution. These logs follow the format SEQ <SERVICE_NAME> <EVENT> and are timestamped using the CLOCK_MONOTONIC_RAW realtime clock. The real-time services (CAPTURE and DETECT) each generate four log entries per invocation: POST, START, STOP, and DUR, which respectively denote when the sequencer signaled the service to run, when it actually started, when it completed, and how long it took to execute. The SAVE service, which operates independently of the sequencer, emits START, STOP, and DUR logs but does not produce a POST entry since it is not controlled by the sequencer.

An example of these logs can be seen below:

```
Jul 28 19:34:18 arthur capture[213273]: SEQ CAPTURE POST 1801178.858694802
Jul 28 19:34:18 arthur capture[213273]: SEQ DETECT POST 1801178.858766913
Jul 28 19:34:18 arthur capture[213273]: SEQ CAPTURE START 1801178.858794
Jul 28 19:34:18 arthur capture[213273]: SEQ CAPTURE END 1801178.861018
Jul 28 19:34:18 arthur capture[213273]: SEQ CAPTURE DUR 0.002225
Jul 28 19:34:18 arthur capture[213273]: SEQ DETECT START 1801178.861148
Jul 28 19:34:18 arthur capture[213273]: frames_rx: 6 diff_to_previous: 0.000000
Jul 28 19:34:18 arthur capture[213273]: found image current_frame_diff=0.221630
Jul 28 19:34:18 arthur capture[213273]: SEQ SAVE START 1801178.863992
Jul 28 19:34:18 arthur capture[213273]: frames_rx: 7 diff_to_previous: 0.000000
Jul 28 19:34:18 arthur capture[213273]: frame isn't different enough: current_frame_diff=0.000000
Jul 28 19:34:18 arthur capture[213273]: SEQ DETECT END 1801178.865107
Jul 28 19:34:18 arthur capture[213273]: SEQ DETECT DUR 0.003960
```

To verify the system's ability to meet its real-time deadlines, logs were collected during two full test runs for each mode (1Hz and 10Hz). These logs were then parsed and analyzed using a Python script included in the project deliverable zip file. The script computes the following timing metrics for each service:

- Worst Case Execution Time (WCET): the longest observed duration between the service's START and STOP logs.
- Response Time: the delay between the POST and START log entries, representing the time taken for the service to begin execution after being released by the sequencer.
- Completion Time: the total time from POST to STOP, capturing both queuing delay and execution time.

For each metric, the script calculates minimum, mean, and worst-case values, which are critical inputs for scheduling validation and Rate Monotonic Analysis. These results are later used in conjunction with the static analysis tool Cheddar to verify that all real-time threads operate within their defined periods and meet their deadlines without overrun. This approach ensures that both temporal predictability and data integrity are preserved across all system components in both the 1Hz analog and 10Hz digital clock capture modes.

All four of the following runs were performed with the optional grayscale and alpha beta filter features enabled. All tables are presented in terms of milliseconds to provide more relevant timing. This is different from Table 1 which was presented in multiples of the 60Hz sequencer time slice.

1Hz Requirement, Run 1

Table 2 - One Hz Requirement First Run Timing

Service	WCET	Period/Deadline	Worst Completion	Worst Response Time
Frame Capture	3.095ms	250ms	3.246ms	0.424ms
Tick Detection	2.863ms	250ms	5.806ms	1.593ms

Over the period of the first 30 minute One Hz run, the Worst Case Execution Times (WCETs) of the Capture and Detection services are 3.095ms and 2.963ms, respectively. Given that both of their periods are 250ms, the total worst-case CPU utilization of the system is 2.38% which is well below the Lowest Upper Bound (LUB) for two services, 82.84% which guarantees the system is schedulable.

1Hz Requirement, Run 2

Table 3 - One Hz Requirement Second Run Timing

Service	WCET	Period/Deadline	Worst Completion	Worst Response Time
Frame Capture	3.143ms	250ms	3.265ms	0.419ms
Tick Detection	2.769ms	250ms	5.673ms	3.322ms

Over the period of the second 30 minute One Hz run, the Worst Case Execution Times (WCETs) of the Capture and Detection services are 3.143ms and 2.769ms, respectively. Given that both of their periods are 250ms, the total worst-case CPU utilization of the system is 2.36% which is well below the Lowest Upper Bound (LUB) for two services, 82.84% which guarantees the system is schedulable.

10Hz Requirement, Run 1

Table 4 - Ten Hz Requirement First Run Timing

Service	WCET	Period/Deadline	Worst Completion	Worst Response Time
Frame Capture	2.940ms	33.33ms	3.323ms	0.383ms
Tick Detection	3.960ms	40ms	6.340ms	2.473ms

Over the period of the first 3 minute Ten Hz run, the Worst Case Execution Times (WCETs) of the Capture and Detection services are 2.940ms and 3.960ms, respectively. Given that the periods for the services are 33.33ms and 40ms respectively, the total worst-case CPU utilization of the system is 18.72% which is well below the Lowest Upper Bound (LUB) for two services, 82.84% which guarantees the system is schedulable. This result shows that the total CPU usage is greatly increased due to the higher rates of the services but it is still guaranteed to be schedulable provided the camera continues to perform.

10Hz Requirement, Run 2

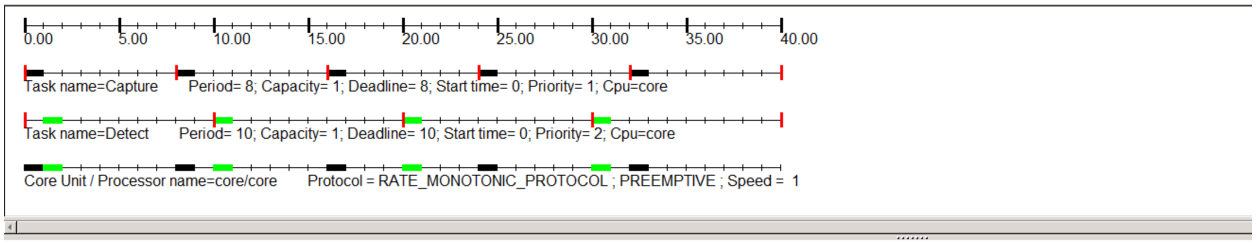
Table 5 - Ten Hz Requirement Second Run Timing

Service	WCET	Period/Deadline	Worst Completion	Worst Response Time
Frame Capture	2.976ms	33.33ms	3.385ms	0.409ms
Tick Detection	2.947ms	40ms	5.401ms	2.496ms

Over the period of the second 3 minute Ten Hz run, the Worst Case Execution Times (WCETs) of the Capture and Detection services are 2.976ms and 2.947ms, respectively. Given that the periods for the services are 33.33ms and 40ms respectively, the total worst-case CPU utilization of the system is 16.29% which is well below the Lowest Upper Bound (LUB) for two services, 82.84% which guarantees the system is schedulable.

Cheddar Analysis

Given the data procured from the four total runs it can be found that the absolute worst case execution times for the Frame Capture and Tick Detection services are 3.143 and 3.960, respectively. Given these WCETs, we can use Cheddar to perform additional validation of the system to ensure it is schedulable across the Least Common Multiple (LCM) of the system. To represent the system within cheddar, I will round both WCETs to 4ms to simulate a more loaded system or a worst-case scenario that I may have missed in my testing. I will then divide the WCETs and both periods by 4 to have capacities of 1 and periods of 8 and 10. This is so that the diagrams produced by Cheddar are more easily viewed. The result of processing with Cheddar is seen below in Diagram 6:



```
Task name=Capture    Period= 8; Capacity= 1; Deadline= 8; Start time= 0; Priority= 1; Cpu=core
Task name=Detect     Period= 10; Capacity= 1; Deadline= 10; Start time= 0; Priority= 2; Cpu=core
Core Unit / Processor name=core/core     Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE ; Speed =  1
```

```
Scheduling simulation, Processor core :
- Number of context switches :  72
- Number of preemptions :  3

- Task response time computed from simulation :
    Capture => 4/worst
    Detect => 8/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.


Scheduling simulation, Processor core :
- Number of context switches :  8
- Number of preemptions :  0

- Task response time computed from simulation :
    Capture => 1/worst
    Detect => 2/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.


Scheduling feasibility, Processor core :
1) Feasibility test based on the processor utilization factor :

- The feasibility interval is 40.0, see Leung and Merill (1980) from [21].
- 31 units of time are unused in the feasibility interval.
- Number of cores hosted by this processor :  1.
- Processor utilization factor with deadline is 0.22500 (see [1], page 6).
- Processor utilization factor with period is 0.22500 (see [1], page 6).
- In the preemptive case, with RM, the task set is schedulable because the processor utilization factor 0.22500 is equal or less than 0.82843 (see [1], page 16, theorem 8).


2) Feasibility test based on worst case response time for periodic tasks :

- Worst case task response time :  (see [2], page 3, equation 4).
    Capture => 1
    Detect => 2
- All task deadlines will be met : the task set is schedulable.
```

Diagram 6 - Cheddar Analysis

To interpret the above diagram in words, it is found that the system is fully schedulable and has no missed deadlines. This example set of services has a total CPU utilization factor 22.5% which is greater than both of the 10Hz runs examined before. Both of the scheduling point tests have also passed meaning the system is completely schedulable.

Testing

Testing for this project was an entirely manual process. This involved running to gather image difference data and empirically determine the thresholds for what constituted a stable and a different frame depending on the frequency and the clock target. To validate the system, I manually inspected each frame captured visually to ensure there were not any dropped, duplicated, or blurry frames. Along with the raw frames captured with timestamps, I have generated .mp4 videos of all four of my runs each running at 5 frames per second. This makes it so the 1 Hz project captures are running in 5x speed while the 10 Hz project captures are running at 0.5x speed allowing for easy visual confirmation of the frame.

Conclusion

In conclusion, the development and real-time validation of the computer vision synchronome demonstrate the feasibility and reliability of using embedded Linux and real-time scheduling techniques on a Raspberry Pi 4 for precise image capture tasks. Through careful architectural design—leveraging SCHED_FIFO threads, semaphore-based sequencing, and core isolation—the system successfully meets the stringent timing requirements necessary to capture both analog and digital clock transitions without missing or duplicating frames. Comprehensive logging and analysis confirm that all real-time services operate well within their deadlines, with minimal jitter and consistent performance across multiple test runs. The resulting platform not only meets its design goals but also provides a scalable foundation for future extensions in real-time computer vision and embedded automation.

References

I would like to thank Professor Sam Siewert for providing the starter code that I used to begin the project. The starter code was pulled from his repository on github, https://github.com/siewertsmooc/RTES-ECEE-5623/tree/main which was used to create my sequencer design and overall thread processing. I also used the c-brighten example from this repository to perform the alpha beta filter to brighten the image and increase the contrast.

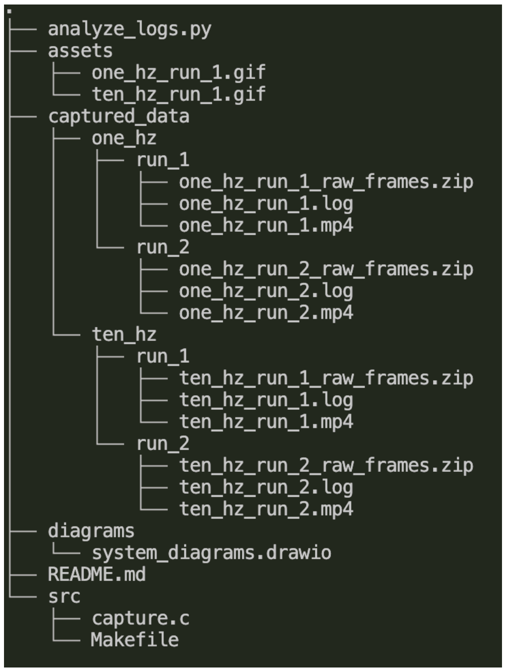
Appendix

The submitted directory structure for the program consists of my collected data from the two runs of my 1Hz program and two runs of my 10Hz program. Each run contains a zip file of all 1801 captured frames, a capture of the Linux syslogs over the course of the program, and a generated 5 FPS .mp4 file of the captured frames.

The code and Make file can be found within the src/ directory. If running on a Raspberry Pi 4, you can simply call `make` and run the program with `sudo ./capture <1 or 10 Hz mode> <save grayscale> <apply alpha beta filter`.

At the top level of the directory is a PDF of this report as well as the Python script used to analyze the syslogs produced by each run. This program can be run with `python3 <path to desired log file>`.

The overall directory structure is seed in the tree view contained in Diagram 7 below:



```
.
├── analyze_logs.py
├── assets
│   ├── one_hz_run_1.gif
│   └── ten_hz_run_1.gif
├── captured_data
│   ├── one_hz
│   │   ├── run_1
│   │   │   ├── one_hz_run_1_raw_frames.zip
│   │   │   ├── one_hz_run_1.log
│   │   │   └── one_hz_run_1.mp4
│   │   └── run_2
│   │       ├── one_hz_run_2_raw_frames.zip
│   │       ├── one_hz_run_2.log
│   │       └── one_hz_run_2.mp4
│   └── ten_hz
│       ├── run_1
│       │   ├── ten_hz_run_1_raw_frames.zip
│       │   ├── ten_hz_run_1.log
│       │   └── ten_hz_run_1.mp4
│       └── run_2
│           ├── ten_hz_run_2_raw_frames.zip
│           ├── ten_hz_run_2.log
│           └── ten_hz_run_2.mp4
├── diagrams
│   └── system_diagrams.drawio
├── README.md
└── src
    ├── capture.c
    └── Makefile
```

Diagram 7 - Artifact Structure