**Audio-Video Synchronization**

This document was created to support members of the lab working with synchronized audio and video data collected during behavioral recording sessions. The primary goal is to provide clear and practical guidance on how to process and align audio and video files using the synchronization system developed in the Provenza Lab. The system relies on a custom Arduino setup that encodes frame-level identifiers into the audio stream, allowing for precise alignment of video frames and audio segments during post-processing.

The document is divided into four main sections:

1. **How to Use the Code:** Instructions for people performing alignments, including how to organize the folder structure, prepare the raw files, and run the MATLAB code.
2. **Hardware Description:** A brief overview of the recording setup, including the cameras, audio equipment, and synchronization box.
3. **Arduino Functionality:** A summary of how the Arduino generates frame triggers and encodes serial frame IDs for synchronization.
4. **MATLAB Pipeline Overview:** An explanation of how the MATLAB code works, how it extracts and decodes the signals, and how the actual alignment is performed.

Table of Contents

[Hardware Components 1](#_Toc196991762)

[System Architecture 1](#_Toc196991763)

[Description 2](#_Toc196991764)

[Arduino Code & Frame ID Transmission 3](#_Toc196991765)

[Objective: Real-Time Frame Tagging 3](#_Toc196991766)

[Serial Communication Protocol 4](#_Toc196991767)

[Trigger Signal to Cameras 5](#_Toc196991768)

[How the Timing is Controlled 6](#_Toc196991769)

[Summary of the process 6](#_Toc196991770)

[MATLAB Post-Processing Pipeline 7](#_Toc196991771)

[Step-by-Step Pipeline Overview 7](#_Toc196991772)

[4.2 Loading Audio and Preprocessing 8](#_Toc196991773)

[Extracting Binary Signal from Channel 3 11](#_Toc196991774)

[Organize Video Files 13](#_Toc196991775)

[4.5 Matching Audio with JSON Metadata 14](#_Toc196991776)

[Aligning Audio with Video 16](#_Toc196991777)

[Appendix 18](#_Toc196991778)

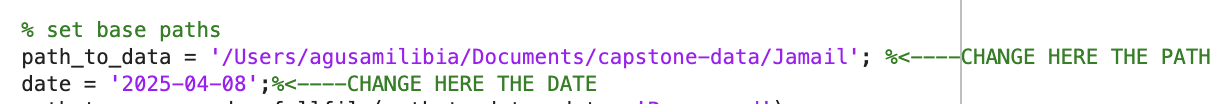
[5. Instructions for New Users 18](#_Toc196991779)

## How to use the code

This section explains how to use the synchronization code once you have already recorded the audio and video files separately. If you are unsure how to perform the actual recording of audio or video, please refer to the corresponding sections in the Notion documentation for detailed instructions.

To run the MATLAB code correctly, there are a few required steps and parameters you need to set:

* **Update the paths**:

In the script, you must specify the correct base folder path (path\_to\_data) and the name of the session folder (the date string). These should match your local folder structure. 

* **File names and folder structure**:

The expected structure is:

A black text on a white background

AI-generated content may be incorrect.

* **Audio file requirements**:

The audio folder must contain **exactly 3 audio files** — one for each channel. All three files must come from the same recording session and have **exactly the same duration**.

*Attention: DO NOT mix audio files from different sessions or devices in the same folder.*

Audio file naming recommendation:

To ensure the third audio channel (Channel 3, which contains the sync signal) is used properly by the code, it’s critical that the file for Channel 3 is the last one alphabetically.

A reliable naming scheme is:

* + audio-1.wav (Channel 1)
  + audio-2.wav (Channel 2)
  + audio-3.wav (Channel 3) ← this one will be used for decoding sync pulses

It is important to export each audio channel separately after the session has ended. This should be done by selecting one channel at a time and assigning a clear, consistent name to each exported file. For example, you can select Channel 1 and export it as “audio-1”, then do the same for Channel 2 and Channel 3. This ensures that each channel is saved as an individual .wav file, which is necessary for accurate processing and synchronization later in the pipeline.

* A screen shot of a computer

  AI-generated content may be incorrect.

A screen shot of a computer

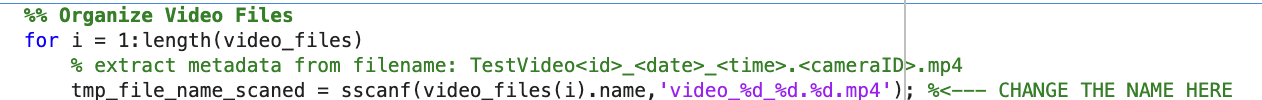
AI-generated content may be incorrect.

* **Video and audio file naming tip:**

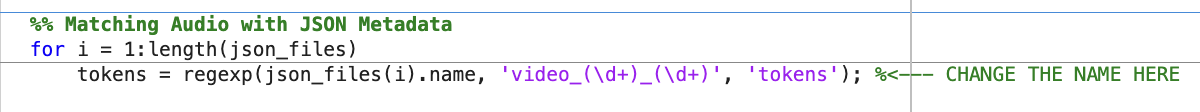
For audio files, the specific names do not matter, as long as the third file (alphabetically) corresponds to Channel 3, which contains the synchronization signal. The code will automatically use that last file without requiring you to specify its name in MATLAB.

For video files, however, the name does matter. It is highly recommended to name all video files starting with "video" so the code can detect and parse them correctly. If you choose a different naming convention, you will need to manually update the script to match the new video file pattern — both for the .mp4 files and the corresponding .json metadata files.

The specific lines that need to be changed are shown in the figure below:



And:



## Hardware Components

A diagram of a computer

AI-generated content may be incorrect.

The system integrates audio and video recordings in a synchronized way using multiple hardware components:

* **PC**: Acts as the central recording unit. It receives video frames via an Ethernet switch and records audio using Audacity.
* **Arduino MEGA**: Serves as the main timing controller. It generates trigger pulses for the cameras and sends synchronization signals to the audio interface.
* **Custom PCB**: Acts as a bridge between the Arduino and the FLIR cameras. It relays trigger signals to the cameras and serial data to the Arduino.
* **Cameras**: Capture synchronized video frames. They are connected to the PC through a NETGEAR Ethernet switch and receive triggers from the custom PCB.
* **Audio System**: Includes two microphones connected to an audio interface. Audio is recorded on the PC and contains encoded pulses from the Arduino used to align with video frames.

All components are powered and connected in a way that ensures tight temporal alignment between video and audio, which is later decoded in MATLAB for precise analysis.

### Description

**PC**

* **Functions**:
  + Records audio using Audacity.
  + Receives video from the FLIR cameras via an Ethernet Switch.
  + Runs MATLAB scripts to decode synchronization signals and align video with audio.
* **Network Adapter**: Uses a TP-Link 10Gb PCIe adapter (TX401) to handle high-bandwidth video streaming.

**Arduino MEGA**

* **Functions**:
  + Acts as the central controller for synchronization.
  + Sends trigger pulses to cameras via the custom PCB.
  + Sends synchronization pulses as 5-byte serial packets into audio channel 3, which are later decoded.
* **Key Pins Used**:
  + Digital Pins 1 (TX) and 0 (RX): For serial communication.
  + Digital Pin 11: Trigger signal output.
  + Digital Pin 10: Clock output.
  + GND and 9V: Power and reference.

**Custom PCB (Printed Circuit Board)**

* **Purpose**:
  + Interfaces between Arduino and the cameras.
  + Sends GPIO trigger signals to each camera.
  + Forwards serial data to the Arduino.
* **Cabling**:
  + Uses a 6-pin Hirose HR10 connector for GPIO connections to each camera.
  + Powered externally via a 5V 2A power supply.

**Cameras (FLIR)**

* **Purpose**:
  + Capture synchronized video frames.
  + Receive hardware trigger signals from the PCB.
* **Connection**:
  + Connected to the Ethernet Switch via Cat6 cables.
  + PoE-enabled switch powers the cameras and transfers video data to the PC.

**Ethernet Switch (NETGEAR GS305EP)**

* **Function**:
  + Manages Ethernet connections between the PC and all cameras.
  + Provides Power over Ethernet (PoE) to power each FLIR camera.
  + Handles high-speed data transmission.

**Audio System**

* **Components**:
  + Two microphones: one Sennheiser shotgun mic and one boundary mic.
  + An audio interface receives mic input and sends the signal to the PC.
* **Function**:
  + Records environmental sound plus the Arduino-generated pulses on Channel 3, which are later decoded in MATLAB to extract sync information.

## Arduino Code & Frame ID Transmission

**Objective: Real-Time Frame Tagging**

In this system, the Arduino plays a critical role: it controls the timing of video frame capture and sends a serial data stream that contains a Frame ID associated with each video frame. This Frame ID is encoded as audio, allowing precise synchronization between the recorded video and audio signals later during processing.

The primary objective of the Arduino in this setup is to tag each captured video frame with a unique identifier (Frame ID) and to synchronize the cameras and the audio system.

The Arduino performs two tasks simultaneously:

* It sends a short digital pulse to the cameras to instruct them to capture a new frame.
* It sends a corresponding Frame ID encoded as a serial data packet to the audio system.

This dual action guarantees that each frame in the video can later be matched precisely with its associated Frame ID recorded in the audio signal.

A diagram of a computer

AI-generated content may be incorrect.

**Serial Communication Protocol**

The Frame ID, which is simply a 32-bit unsigned integer (starting from 0 and incrementing by 1 with each frame), is transmitted via serial communication.

However, transmitting a 32-bit integer directly would cause problems for audio recording:

* Audio signals are limited in range, typically [-1, 1] after normalization.
* If a full 8-bit byte (0–255) were recorded, values above 127 would be interpreted as negative values due to two’s complement representation in signed 8-bit systems.
* Negative jumps or saturation would corrupt the signal and make decoding unreliable.

**Solution:**

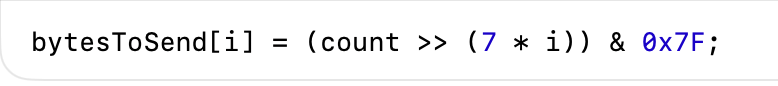
Instead of using full 8-bit bytes, each byte is limited to 7 bits (0–127).

This ensures that all transmitted values stay positive and below 128, avoiding distortion when interpreted as an audio waveform.

The 32 bits of the Frame ID are divided into five 7-bit chunks, not four 8-bit bytes:

* 5 × 7 bits = 35 bits, enough to cover the 32 bits needed.
* The extra bits are simply unused or set to 0.

The splitting operation is done in the Arduino with this line inside a loop:



* count >> (7 \* i): shifts the Frame ID right by 7 \* i bits.
* & 0x7F: applies a bitmask to keep only the lowest 7 bits.

Then, the Arduino **sends each 7-bit byte sequentially** via three serial ports:

A black text on a white background

AI-generated content may be incorrect.

* Serial1, Serial2, and Serial3 are connected to different destinations (audio recorders or backup lines).
* Each byte is transmitted individually and flushed immediately to guarantee that it’s physically sent before proceeding.

The result is that the audio channel captures a binary sequence of 0s and 1s (corresponding to low and high amplitudes) representing these bytes.

Later, in MATLAB, these sequences are decoded back into Frame IDs by reversing the process:

* Group consecutive bits into bytes.
* Recombine the bytes to reconstruct the original 32-bit Frame ID.

A whiteboard with writing on it

AI-generated content may be incorrect.

**Trigger Signal to Cameras**

In parallel to sending the serial data, the Arduino also **sends a short HIGH-LOW digital pulse** to the cameras.

This is done with:

A close-up of a computer code

AI-generated content may be incorrect.

* The pulse is sent on a designated digital output pin (pin 13).
* **Pulse HIGH**: tells the cameras to “capture a frame now.”
* **Pulse LOW**: resets the line immediately after sending the Frame ID.

This ensures that the trigger pulse and the Frame ID are tightly aligned each pulse corresponds exactly to one Frame ID sent and recorded.

This synchronization is essential because it allows MATLAB to later know exactly when each frame was taken, using only the audio recording.

**How the Timing is Controlled**

The Arduino uses a timer (from the arduino-timer library) to generate pulses at a fixed rate.

This line in the setup:



Configures the timer to call the function send\_trigger\_sync\_to\_pcb() every ~33.33 milliseconds.

33.33 ms corresponds to approximately 30 frames per second (fps): 1 second/30 frames = 33.33 ms/frame

Thus, the Arduino acts like a metronome, generating frame triggers and Frame ID transmissions exactly 30 times per second, ensuring regular spacing of frames.

**Summary of the process**

At every ~33 ms:

1. Arduino sets the trigger pin HIGH → Cameras capture a frame.
2. Arduino splits the 32-bit Frame ID into five 7-bit bytes.
3. Arduino sends the bytes over Serial1, Serial2, and Serial3.
4. Arduino sets the trigger pin LOW.
5. Arduino increments the Frame ID counter by 1.

All this happens automatically inside the timer callback function.

## MATLAB Post-Processing Pipeline

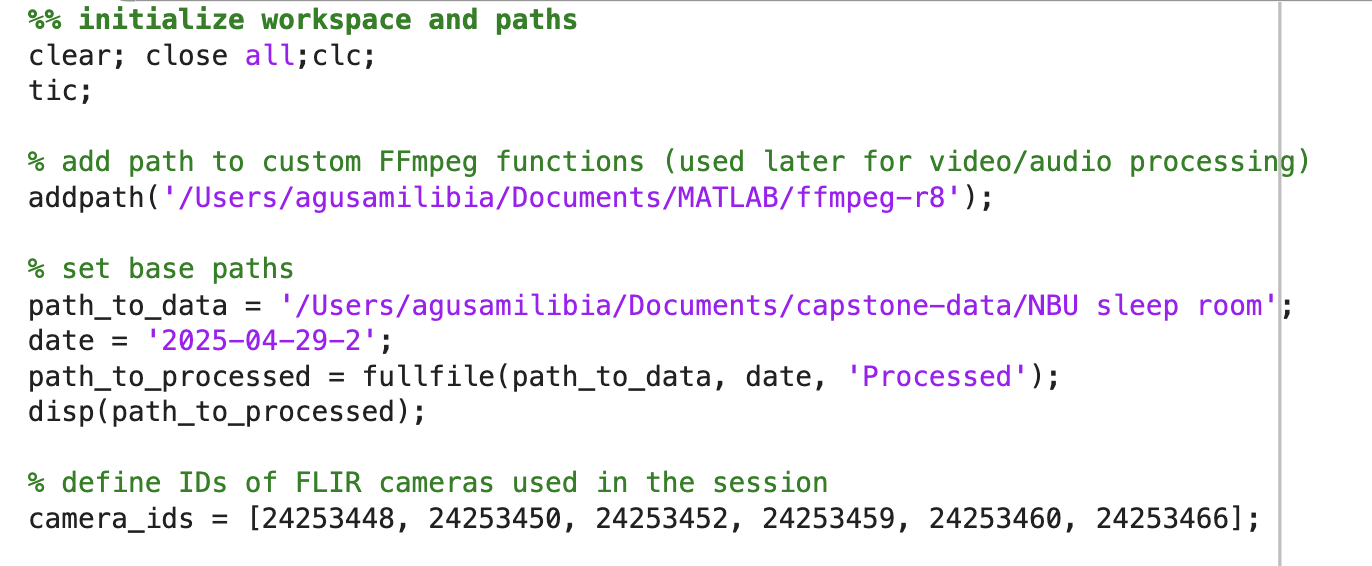
### Step-by-Step Pipeline Overview

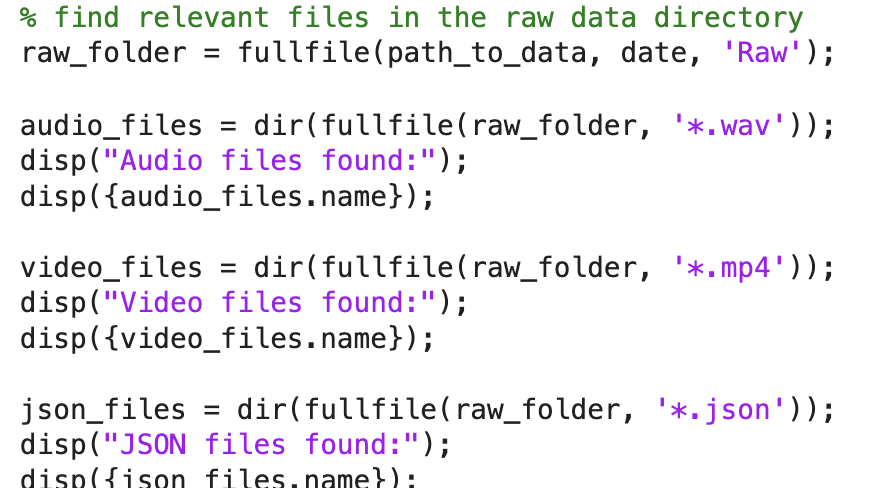
The synchronization pipeline begins by setting up the environment and loading all necessary files, including audio, video, and metadata. The focus is on Channel 3 of the audio signal, which carries the synchronization pulses encoded by the Arduino. This channel is normalized and thresholded to produce a binary signal, which is flipped in some cases (e.g., NBU sleep room) to improve decoding reliability. The system scans the binary signal to detect sync blocks and extracts frame identifiers encoded as digital pulses, storing both the decoded serial IDs and their timestamps.

Next, the system parses video filenames to extract recording session and camera ID information, organizing the data into a structured format. For each video, the corresponding JSON metadata file provides timestamp and frame ID information. Missing IDs are inferred when necessary to ensure continuity. A custom function, find\_audio\_subset, is used to match the video’s frame range to the corresponding segment in the audio signal, with silence added if needed to maintain alignment. The video is then re-encoded with a corrected frame rate based on actual frame intervals, and FFmpeg is used to merge the corrected video with the extracted audio segment. The final synchronized outputs are stored in organized folders, ready for further analysis.

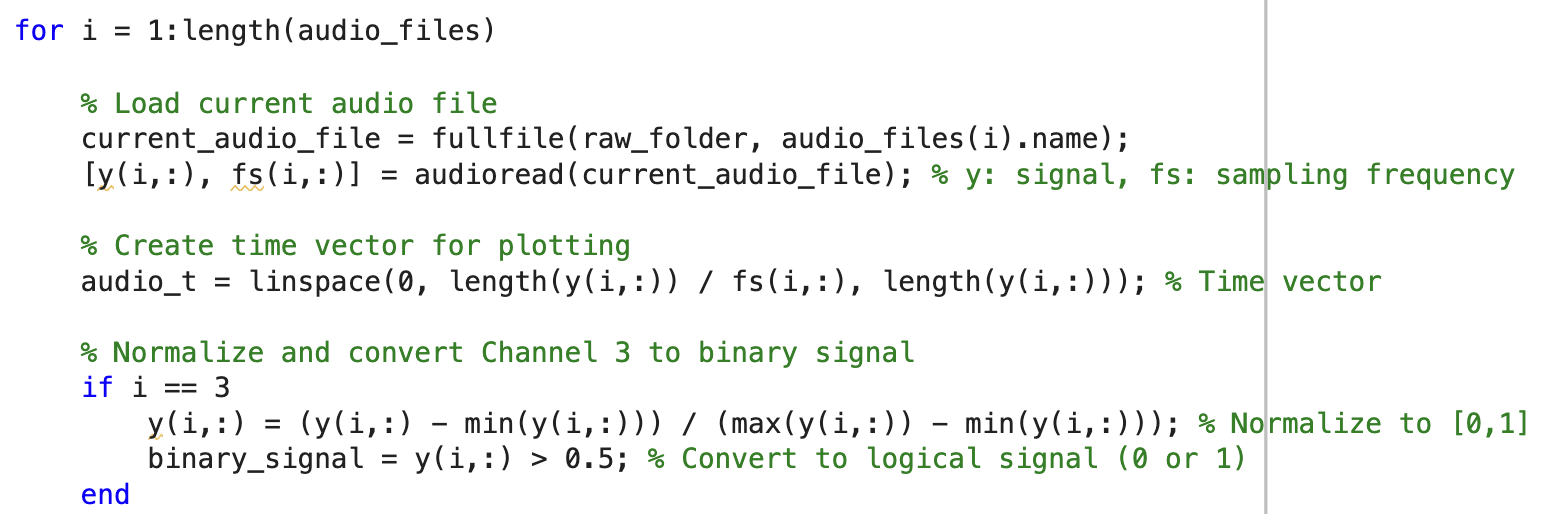
### Loading Audio and Preprocessing

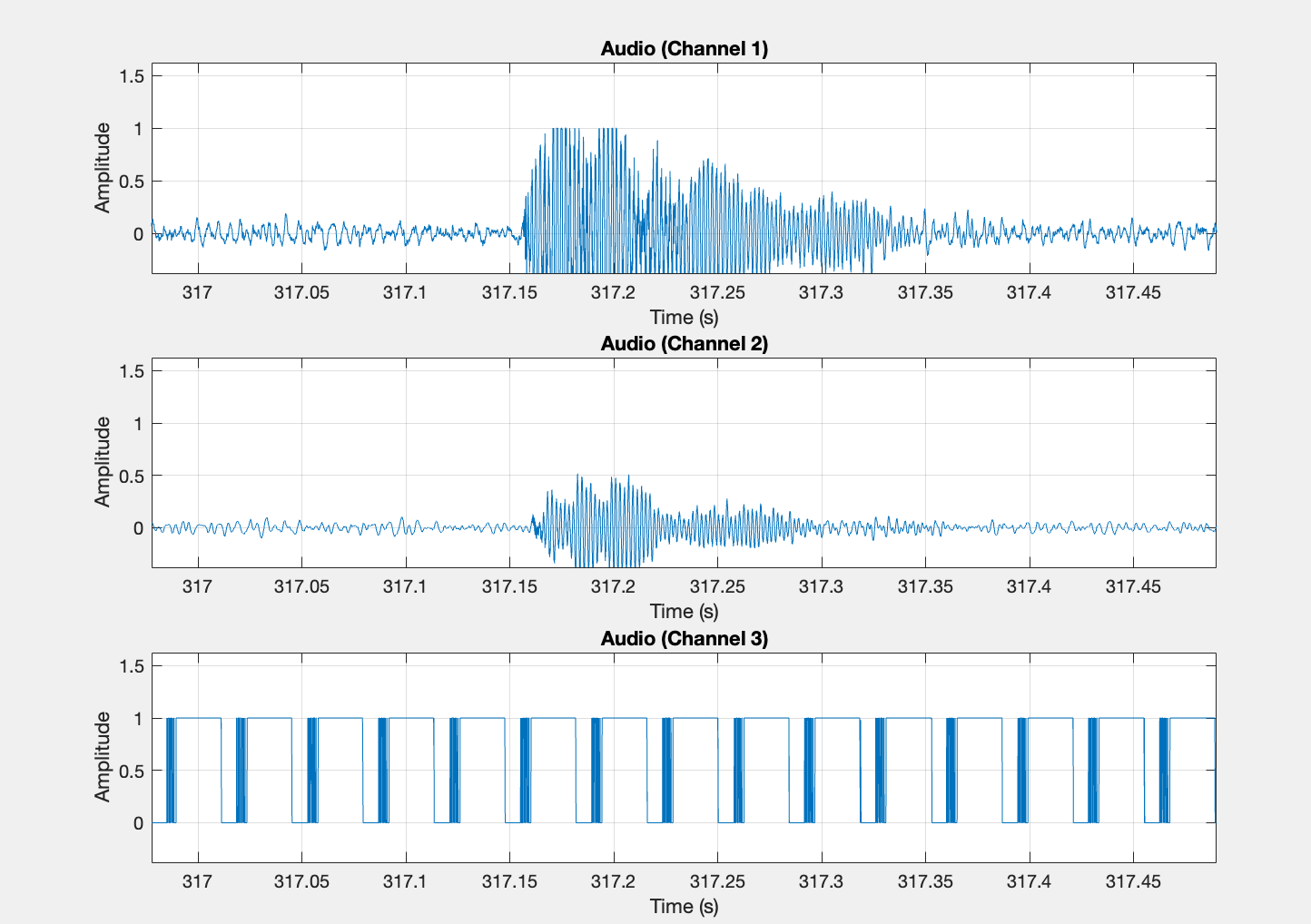
In the **Loading Audio and Preprocessing** stage, the system begins by preparing the environment and identifying the necessary files. It initializes the workspace, sets the paths where the raw data and processed data will be stored, and loads all the relevant files from the selected session folder, including .wav audio files, .mp4 video files, and .json metadata files. It also specifies the IDs of the cameras used during the session.



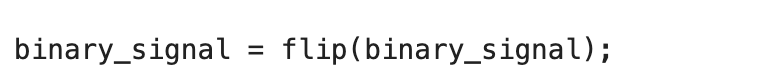


Once the audio files are located, the script loops through each of them and loads the waveform and sampling frequency using MATLAB’s audioread function. For each file, it creates a time vector that maps each audio sample to a moment in time. Special focus is given to **Channel 3**, where the synchronization signal was embedded. This channel is normalized to a 0–1 range, and then converted into a **binary signal** by thresholding: any value above 0.5 is considered a logical ‘1’, and anything below is considered a ‘0’.

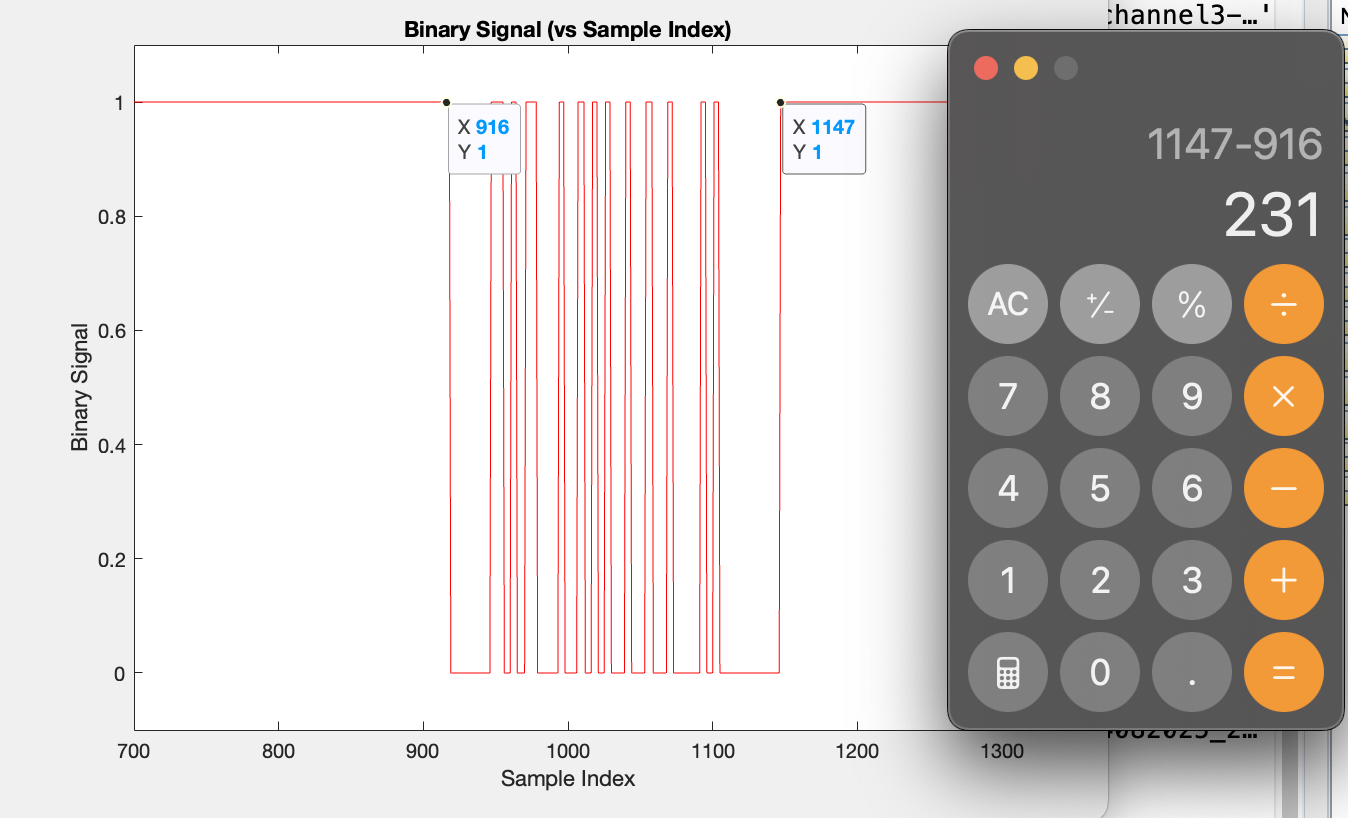




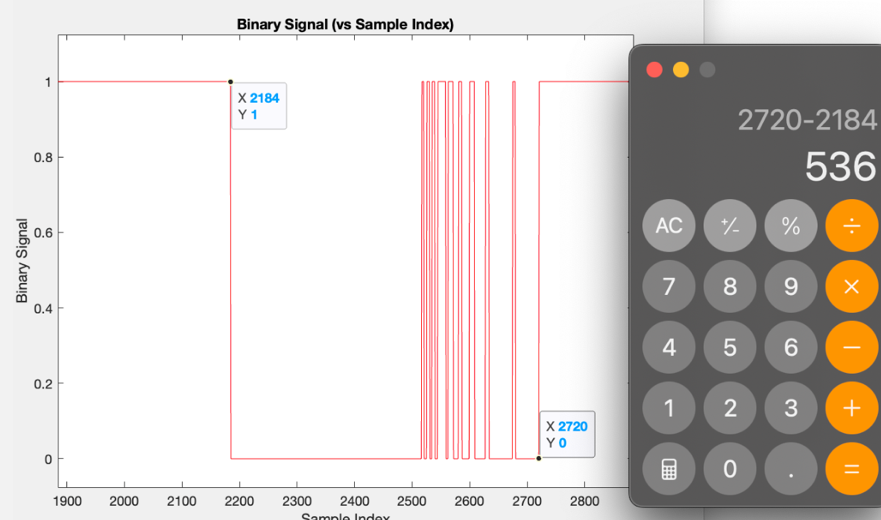
This binary signal is essential for later decoding the frame ID, but some preprocessing is needed depending on the recording environment. In the **NBU sleep room**, the beginning of the binary signal—when analyzed from start to finish—was found to contain a long and variable delay between the first falling edge (from 1 to 0) and the actual data transmission. This made it unreliable to analyze the signal forward in time. To solve this, the entire binary signal is **flipped** and analyzed **backwards**, which produces a much cleaner and more consistent detection of synchronization patterns. This flipping is **specific to NBU**, and is **not required for recordings made in Jamail**, where the signal starts transmitting data immediately after the first 1 is detected. The flipped signal allows for more robust frame ID extraction in environments where delays are unpredictable.



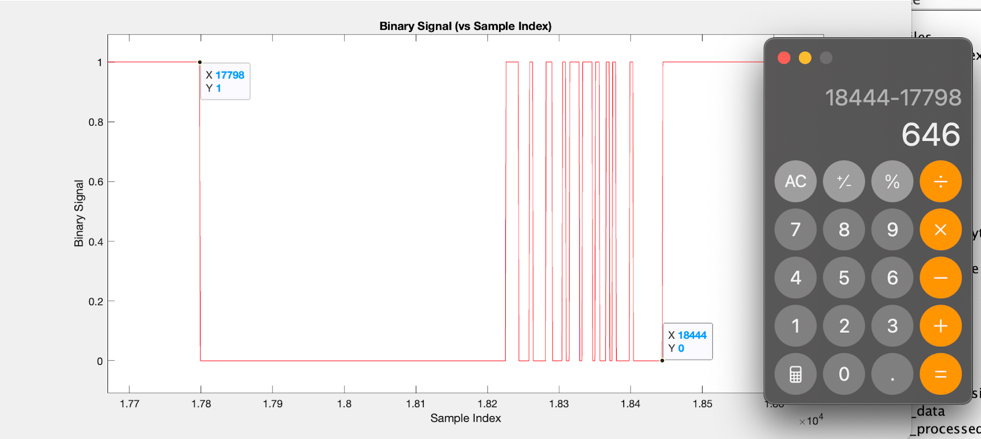
*Binary signal length in Jamail: 231 samples*



*Binary signal length in NBU sleep room: 536 samples*



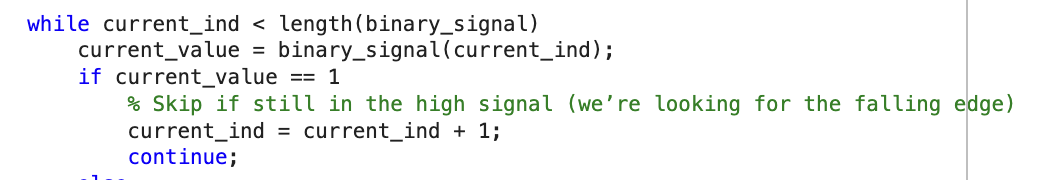
Binary signal length in NBU lounge: 646 samples



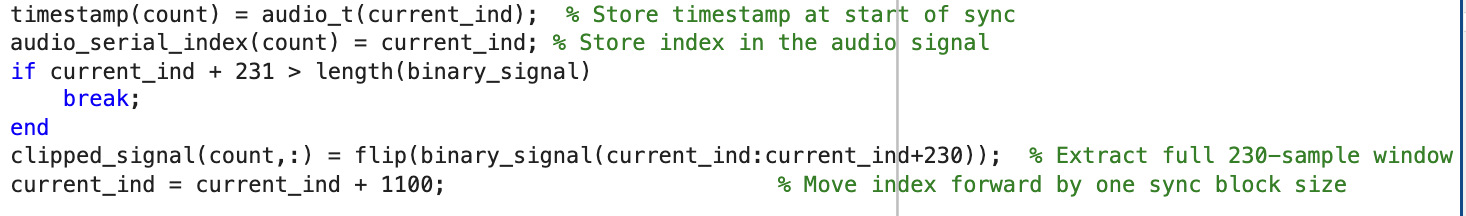
### Extracting Binary Signal from Channel 3

In the **Extracting Binary Signal from Channel 3** stage, the system reads the preprocessed binary signal and searches for specific patterns that indicate synchronization pulses embedded in the audio signal. These pulses were generated during the original recording process using Arduino and are used to encode the frame ID corresponding to each video frame.

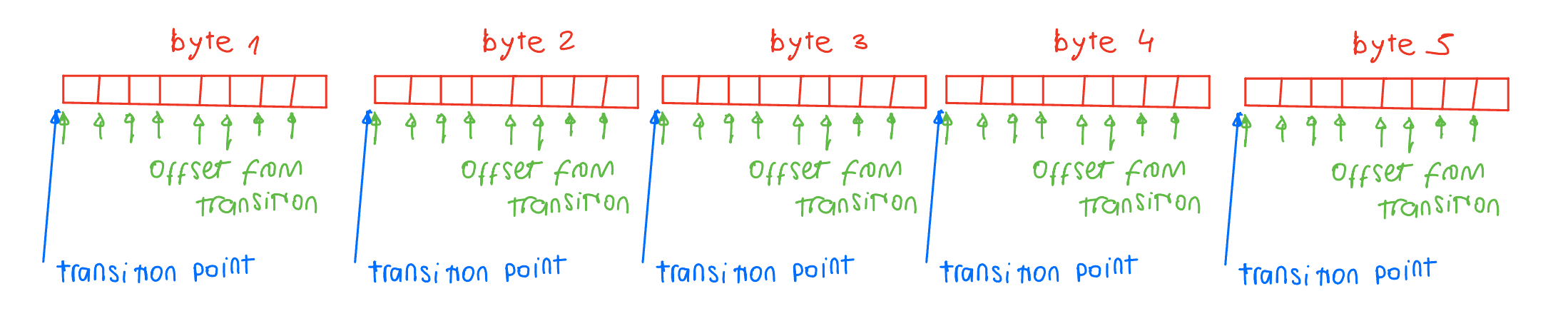
The code scans through the binary signal sample by sample. It looks for the **falling edge**—a transition from a high value (1) to a low value (0)—which marks the beginning of a sync block.

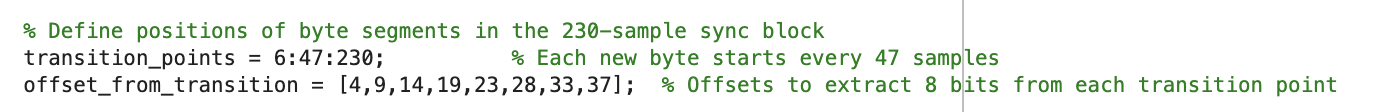


When one is detected, the code stores the timestamp of that moment and extracts a fixed-size segment of the binary signal (231 samples) beginning at that point. These windows are assumed to contain the full data transmission for a frame ID.



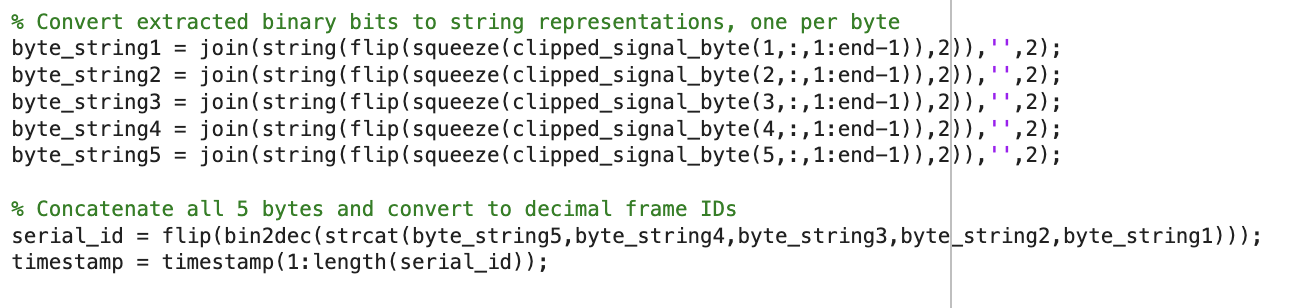
Each sync block is structured to contain 5 bytes (5 groups of 8 bits), encoded in a particular format: each byte starts at a known offset (transition\_points) and the relevant bits are located at specific sample positions relative to that offset (offset\_from\_transition). These bits are extracted and stored in a separate structure. Since the data was flipped earlier, the signal is flipped again here when extracting to restore the proper bit order.



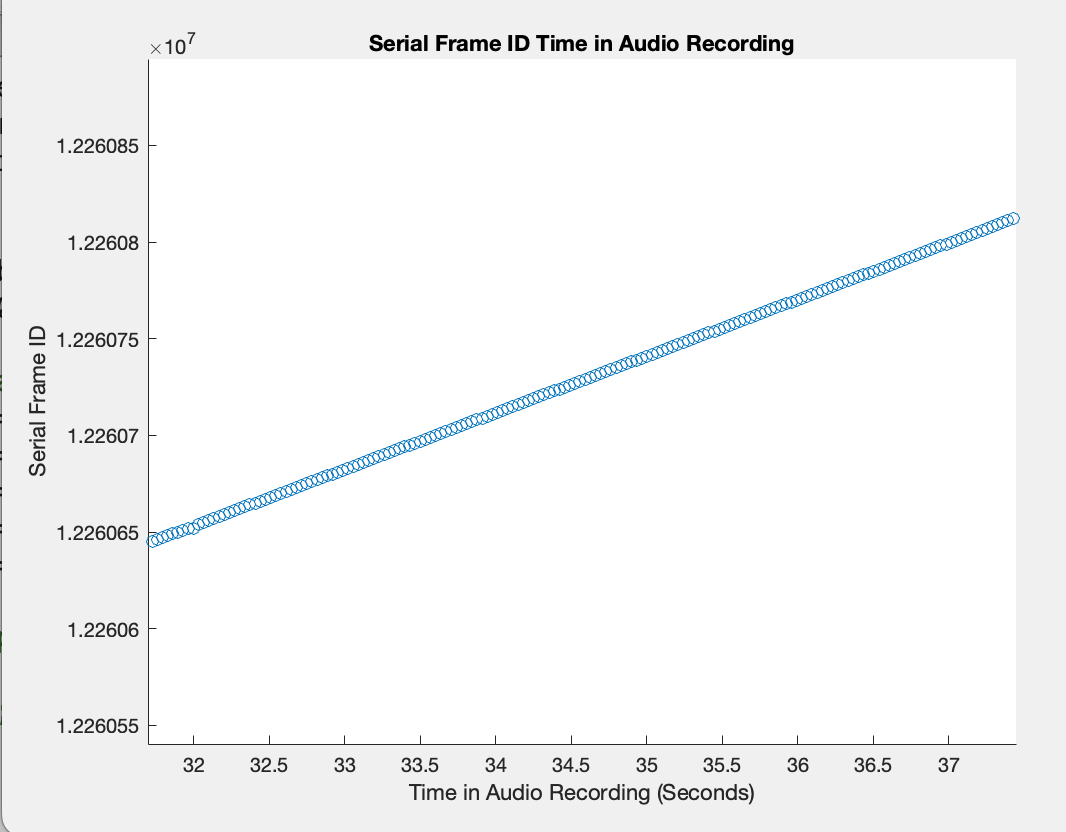




Once all bits for a given sync block are collected, they are converted from binary to strings, and then each 5-byte sequence is concatenated to form a **full 35-bit binary number** (each byte having only 7 useful bits due to the Arduino encoding scheme). These binary strings are finally converted into **decimal serial IDs**, which represent the unique identifier of the video frame associated with that sync pulse.



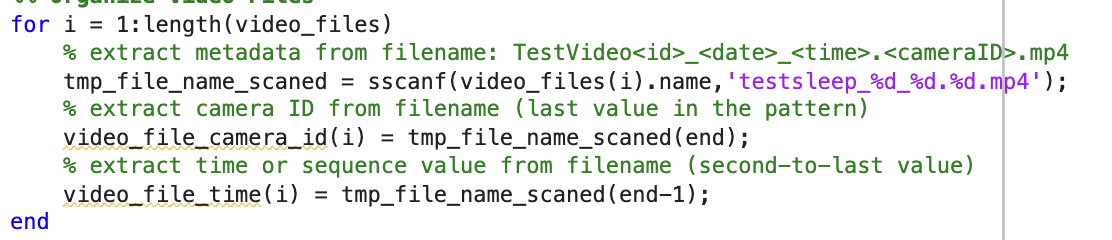
In parallel, timestamps are stored to know when each serial ID was received in the audio. This pairing of timestamp and serial\_id becomes essential for matching audio to video later. Optionally, the user can generate a **scatter plot** to visually inspect the consistency and progression of frame IDs over time. A correctly functioning system should produce a linear progression of IDs with uniform spacing.



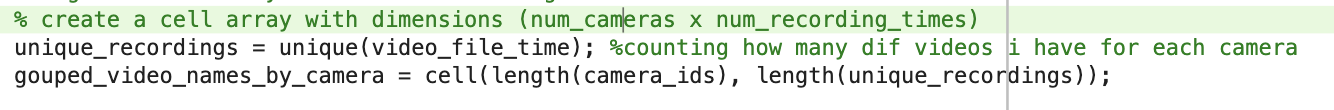
### Organize Video Files

In the **Organize Video Files** step, the system reads all video filenames in the dataset and extracts structured information from those names to correctly group the videos based on **camera ID** and **recording time**. Each filename follows a standard format (e.g., testsleep\_<date>\_<time>.<cameraID>.mp4), and this format is used to parse the embedded metadata.

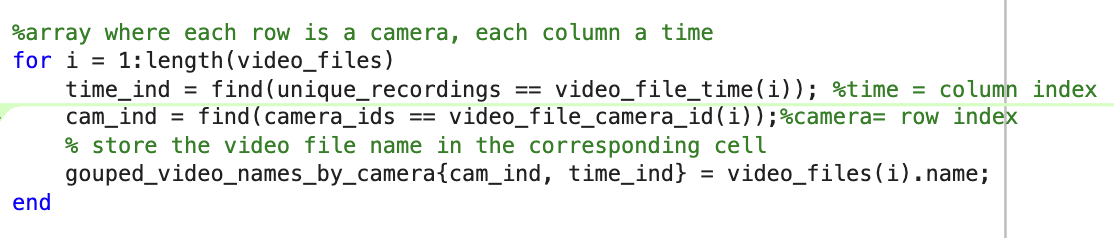
For each file, the code first scans the filename to extract three numerical components: the date, a time or sequence index (representing when in the session the video was recorded), and the camera ID (representing which of the multiple cameras recorded that file).



The script then collects all **unique time indices**, which represent the columns in the data organization structure. It also knows the set of camera IDs defined earlier, which will represent the rows.



A cell array is created to organize the filenames in a grid-like structure where each row corresponds to a specific camera, and each column corresponds to a recording time. Finally, the code assigns each video file to its corresponding position in this cell array based on its parsed camera ID and time index.



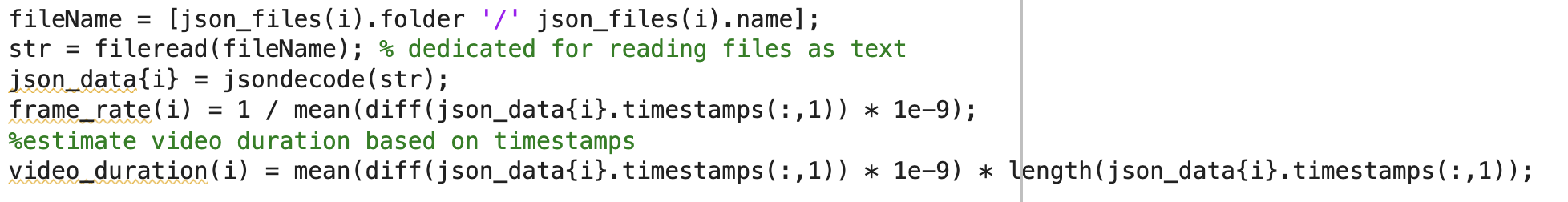
This structure is crucial for synchronizing multi-camera recordings during processing, ensuring that the correct video files are aligned across different viewpoints and time segments.

### Matching Audio with JSON Metadata

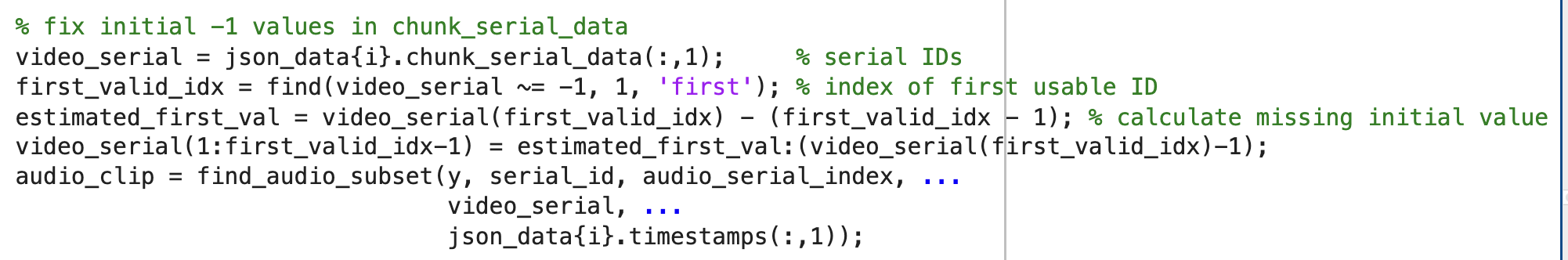
In this part of the code, we begin by going through each JSON metadata file that corresponds to a specific video recording. These files contain detailed timing and identification information about each video frame.

A screenshot of a computer

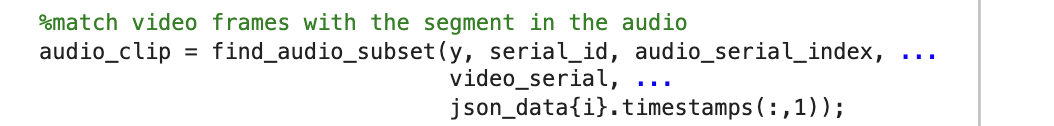
AI-generated content may be incorrect.

First, we extract the relevant numbers from each file name to determine the recording number and the order in which it should be processed. Once we read and decode the content of the JSON file, we calculate the frame rate of the video by measuring the average time difference between consecutive frame timestamps. We also estimate the video’s duration using those timestamps. 

Each video frame in the JSON file is linked to a serial number, known as chunk\_serial\_data, which was originally transmitted from the Arduino. However, in some cases, the beginning of this data may contain placeholder values of -1 instead of valid serial IDs. To solve this, we identify the first valid number (not equal to -1) and then infer what the missing initial IDs should have been, assuming the IDs increment consistently by one. We then replace the initial -1 values with the correct inferred IDs to complete the sequence.

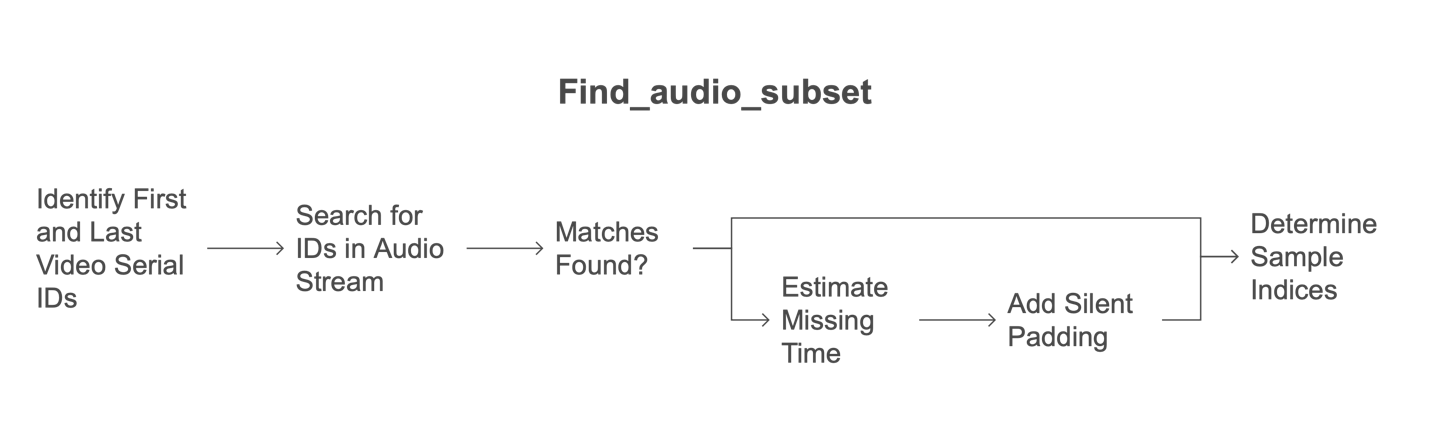


Once the video serial IDs have been corrected, the system attempts to find and extract the matching portion of the audio signal using the function find\_audio\_subset.

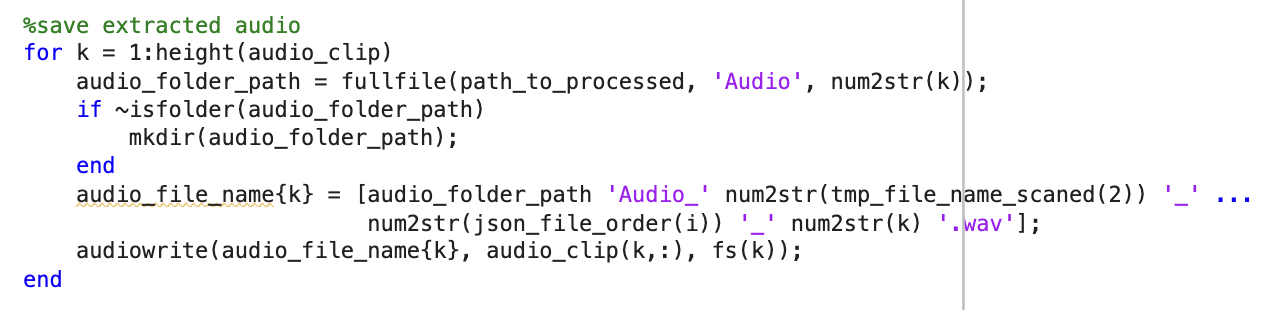
****

This function compares the list of **serial IDs from the video metadata** (video\_serial) with the list of **serial IDs extracted from the audio signal** (audio\_serial) and their corresponding **sample indices** (audio\_ind). Here’s how it works:

1. It identifies the first and last serial ID from the video.
2. It searches for where those IDs appear in the audio’s decoded serial stream.
3. Using the index positions of those matches, it locates the corresponding sample indices in the original audio waveform.
4. If a match cannot be found — for example, if the video starts before the audio recording — the function estimates how much time (and how many samples) are missing and **pads the audio signal with silence** (i.e., zeros) at the beginning or end as needed. This guarantees full alignment in length and timing.
5. The result is a clip from the audio signal that **exactly matches** the time range of the video segment, even if the audio and video started or ended at slightly different times.



Once the audio subset has been successfully extracted, it is saved as a new WAV file. The output file is named to reflect its video segment and camera ID so it can later be recombined with its respective video during the synchronization step.



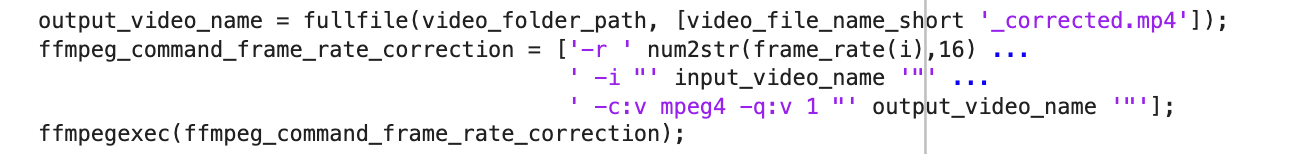
This matching process — pairing serial IDs from video and audio and slicing the corresponding time segments — is what enables precise, sample-level alignment of audio with multi-camera video, a critical feature for behavioral and neurophysiological experiments that require synchronization of signals across devices.

### Aligning Audio with Video

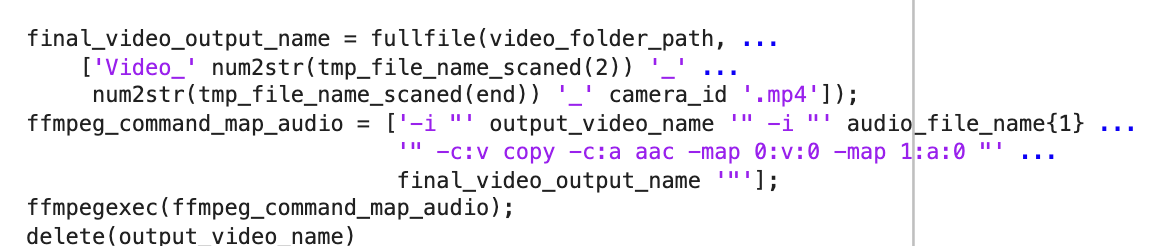
After the system extracts the relevant audio segment that corresponds to a video recording (based on matching serial frame IDs), it proceeds to align and merge the video with that audio. Each camera used in the recording generates a separate video file, and for each one, the code processes the alignment individually.

The first step is to fix the video’s frame rate. Although FLIR cameras record at an intended frame rate (e.g., 30 fps), minor fluctuations in the hardware’s internal clock can cause deviations. Therefore, instead of assuming the nominal rate, the script calculates the actual average frame rate from the .json metadata. This is done by taking the timestamps of each frame (recorded in nanoseconds), computing the differences between them, and averaging that interval. The result is inverted to get frames per second. This corrected frame rate reflects the true pacing of the frames in that specific video.

Next, FFmpeg is called to re-encode the video using this corrected frame rate. The original video is passed in, and FFmpeg applies the new rate using the -r flag, ensuring the output video has frames evenly spaced in time according to the real capture intervals. It uses MPEG-4 encoding with minimal quality loss (-q:v 1) and outputs a temporary file called \_corrected.mp4.

****

Once the corrected video is ready, the second FFmpeg command is issued. This time, the goal is to attach the audio clip — already extracted from the master audio file and matched to the video based on serial frame IDs — to the video. The command takes two inputs: the corrected video and the audio clip. It doesn’t re-encode the video (-c:v copy) but compresses the audio with AAC (-c:a aac) to ensure compatibility. Then it maps the video stream from the first input and the audio stream from the second input to the final output file, which combines them into a synchronized recording.



What’s actually being aligned is the visual frames and the audio waveform, both of which contain embedded markers (serial frame IDs) that originated from the Arduino’s broadcast. The video .json file includes these IDs in the chunk\_serial\_data, and the audio signal includes them encoded as binary pulses. Since the same IDs appear in both data streams, the system can determine the exact audio sample indices that match the video’s first and last frames. This allows it to extract a precisely aligned audio segment for each video.

This entire process is repeated for each video file recorded by the system. If six cameras were used, it executes six times — once per video — always matching the video’s true timing and its corresponding section of audio.

The final synchronized videos are stored in the Processed/Video/ folder, one subfolder per camera. These finalized videos are now ready for further behavioral or neurological analysis, with the confidence that the audio and video are tightly aligned in time using embedded serial frame IDs as the common reference.

## Appendix

#### Full Arduino Code

A screenshot of a computer program

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.