uc3m | Universidad Carlos III de Madrid

University Degree in Computer Science and Engineering
Academic Year (2023-2024)

Bachelor Thesis

Analysis, design and implementation of a sound-based keystroke monitor for Linux

Jorge Ramos Santana

Alejandro Calderón Mateos
Leganés, 2024

CC BY NC ND

This work is licensed under Creative Commons **Attribution – Non Commercial – Non Derivatives**

PREFACE

The research methodology for this thesis was designed to be resource-efficient, utilizing readily available tools and equipment. As a result, no dedicated budget was required for this project.

ABSTRACT

This thesis presents the design and implementation of a system for transmitting data over sound using Frequency Modulation (FM). The system comprises a kernel-level keylogger written in C, which captures user keystrokes, and a user-space modulator that encodes the data and transmits it acoustically using the miniaudio.h library. A Python-based demodulator receives and decodes the audio signal, reconstructing the original keystrokes. The project explores fundamental concepts of Digital Signal Processing (DSP), including Fast Fourier Transform (FFT) and modulation/demodulation techniques. Challenges encountered during the development process, such as preamble detection and error correction, are discussed. The thesis also highlights the limitations of Large Language Models (LLMs) in debugging complex code. Finally, potential applications and future improvements for the system are explored.

Keywords: Frequency Modulation, Data Transmission, Keylogger, Digital Signal Processing.

Para mis padres, por su apoyo incondicional,

y para Eliana, por ser mi pilar todos los días.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Motivation

The transmission of data through unconventional mediums is a fascinating area of research within the field of communications. This thesis explores the feasibility of utilizing near-ultrasound as the medium for data transmission, utilizing Frequency Modulation (FM) as the underlying modulation technique. While traditional data transmission relies on established infrastructure, be it wired or wireless, this project explores the potential of leveraging the readily available presence of audio devices to transmit data acoustically over very short distances.

Using near-ultrasound frequencies as a transmission medium presents a unique opportunity for covert data exfiltration. Doing so circumvents traditional, easily detectable communication channels, offering a degree of plausible deniability for individuals present in the transmission environment. The information is effectively embedded within an inaudible carrier, rendering the communication process imperceptible to the unaided ear.

One of the main goals of the project is to serve as a teaching tool for students interested in digital signal processing, data transmission, and cybersecurity. This document provides a detailed overview of the project's design, implementation, and challenges encountered during development. It explores fundamental concepts of Digital Signal Processing (DSP), data structures, Operating System (OS) programming, and modulation techniques.

1.2. Objectives

This project seeks to achieve the following objectives:

1. **Design and implement a kernel-level keylogger for Linux:** The keylogger, written in C, will operate within the Linux kernel to capture user keystrokes, providing a stream of raw input data for subsequent processing and transmission.

2. **Develop a userspace modulator for encoding and transmitting keystroke data:** The modulator, also written in C, will reside in the userspace and leverage the miniaudio.h library for audio manipulation. It will encode the captured keystroke data using Frequency Modulation (FM) and transmit it acoustically through the system's audio output.

3. **Create a demodulator to receive and decode the transmitted audio signal:** The demodulator, written in Python, will analyze the received audio signal, detect the

presence of the preamble, demodulate the FM-encoded data, and reconstruct the original keystrokes.

4. **Write an easily understandable and comprehensive thesis documenting the project:** The thesis will provide a detailed explanation of the entire system, including the design choices, implementation details, challenges faced, and potential future improvements. It will serve as an educational resource for students and researchers interested in DSP, data transmission, and cybersecurity.

1.3. Structure of the Document

This thesis is structured to provide a logical progression through the design, implementation, and evaluation of the sound-based keystroke monitor. It can be divided into two main sections. Chapters 2 through 7 focus on the DSP and data transmission aspects of the project, while Chapters 8 through 11 delve into the keylogger implementation and communication between kernel-space and user-space. The document is organized as follows:

- **Chapter 2 (Background)** provides a comprehensive background on keyloggers, data transmission over sound, and the near ultrasound spectrum, laying the foundation for the project's technical approach.
- **Chapter 3 (DSP Basics: Fundamentals of Digital Signal Processing)** delves into the fundamentals of Digital Signal Processing (DSP), introducing essential concepts like sampling, quantization, and the Nyquist-Shannon sampling theorem, which are crucial for understanding the signal manipulation techniques employed in the project.
- **Chapter 4 (Transmitting Data Over Sound)** explores the principles of data transmission over sound, covering binary data representation, ASCII encoding, modulation techniques, and the implementation of Frequency Shift Keying (FSK) for data encoding.
- **Chapter 5 (The Fourier Transform)** delves deeper into the Fourier Transform, particularly the Discrete Fourier Transform (DFT), and its application in frequency identification, which forms the basis for demodulating the transmitted signal.
- **Chapter 6 (The Preamble: Synchronizing Transmission and Reception)** introduces the concept of a preamble and its importance in synchronizing transmission and reception. The chapter discusses matched filters, convolution, and the use of Barker codes for robust preamble detection.
- **Chapter 7 (Demodulating the Message)** focuses on the demodulation process, addressing the challenges of real-time processing and the use of frames, buffering, and ring buffers to handle streaming audio data.

- **Chapter 8 (Linux)** provides an overview of the Linux operating system, specifically the Linux kernel, its architecture, and the concepts of kernel-space and user-space, which are crucial for understanding the implementation of the kernel-level keylogger.
- **Chapter 9 (Kernel Level Keylogger)** details the implementation of the kernel-level keylogger, exploring techniques for intercepting keystrokes through the keyboard notifier chain and the use of keymaps for interpreting keycodes.
- **Chapter 10 (Communication between Kernel Space and User Space)** examines the communication methods between kernel-space and user-space, focusing on the use of character devices for transferring keystroke data from the keylogger to the userspace modulator.
- **Chapter 11 (Userspace Modulator)** describes the implementation of the userspace modulator, including the use of miniaudio.h for audio playback and techniques for smoothing audio transitions to minimize audible artifacts.
- **Chapter 12 (Future Works)** discusses potential future improvements and extensions to the project, exploring areas such as error detection and correction, advanced modulation schemes, and enhanced keylogger features.
- **Chapter 13 (Uses of LLMs in the Project)** reflects on the role of Large Language Models (LLMs) in the development of the project, including their applications in code generation and text review, while also addressing their limitations.
- **Chapter 14 (Conclusions)** concludes the thesis, summarizing the project's achievements, and includes a detailed reflection on the legal and ethical implications of the system, and a socio-economic analysis of its potential impact.

The document also includes appendices containing relevant code examples and supplementary information to aid in understanding the project's implementation details.

2. BACKGROUND

2.1. Keyloggers

A keylogger is a type of surveillance software or hardware that records every keystroke made by a user on a computer system. Given that all keystrokes are recorded, the information captured can include sensitive information such as passwords, credit card numbers, and personal messages. Keyloggers are frequently employed by malicious actors for stealing information, spying on individuals, or gaining unauthorized access to systems.

2.1.1. Software Keyloggers

Software keyloggers operate within the operating system of a computer, often disguised as legitimate programs or hidden within existing software. They can capture keystrokes by hooking into keyboard input APIs, intercepting calls to system libraries that handle keyboard input and capturing keystrokes as they are processed by the operating system. Another method they use is by monitoring system events related to keyboard activity, such as key presses and releases, to record keystrokes. More sophisticated keyloggers can operate at the kernel level of the operating system, giving them access to raw keyboard input data before it is processed by higher-level system components.

One such example of a software keylogger is `spy.c`, a simple keylogger written in C that captures keystrokes at the kernel level [1]. The code is designed to run on Linux systems and uses the `input.h` header file to access the input subsystem of the Linux kernel. Once loaded, the keylogger captures keystrokes and logs them to a file, allowing an attacker to monitor the user's input without their knowledge.

2.1.2. Hardware Keyloggers

Hardware keyloggers are physical devices that are connected between the keyboard and the computer. This physical interception makes them more challenging to detect by software-based security measures, and they can capture keystrokes no matter the operating system or software in use. They are typically small and inconspicuous, making them easy to conceal. Some take the form of keyboard cable adapters that plug into the keyboard cable, intercepting keystrokes as they travel from the keyboard to the computer. Hardware keyloggers may also be integrated into the keyboards themselves, making it extremely challenging to detect them. Compared to software keyloggers, they are much more targeted and require physical access to the target system. As a result, they are often used in scenarios where the attacker has direct access to the computer, such as in shared workstations, or in corporate espionage.

2.2. Data Transmission over Sound

2.2.1. Acoustic Modems

Acoustic modems, prevalent in the early days of computing, used audible sound waves to transmit data over telephone lines. They were born out of necessity, as digital communication was not yet widespread, but most homes had analog telephone lines. Acoustic modems converted digital data into audible tones, typically using a technique called Frequency Shift Keying (FSK), where each tone represented a specific binary value (0 or 1), and the sequence of tones conveyed the data. The audible tones were transmitted over the phone line and decoded by a receiving modem at the other end. Acoustic modems were limited by the bandwidth of the telephone lines, which restricted their data transfer rates.

2.3. Near Ultrasound Spectrum

The near ultrasound spectrum refers to sound frequencies just above the upper limit of human hearing, typically ranging from 16 kHz to 20 kHz. These frequencies are inaudible to most adults, but can be easily detected by microphones. Their inaudibility to humans makes them suitable for a range of applications: Near ultrasound can be used for covert communication, allowing the transmission of data without alerting individuals nearby. Similar to how modems used audible sound waves for data transmission, near ultrasound can be employed to transmit data over short distances using acoustic signals. This idea is not new, and it has been explored in various consumer products.

2.3.1. Apple HomePod Configuration

Apple's HomePod smart speaker utilizes near ultrasound during its initial configuration process. When setting up a HomePod, it emits near ultrasonic tones that contain the configuration password. The user's iPhone, placed near the HomePod, detects these tones and automatically configures the device, simplifying the setup process. Alternatively, the user can manually enter the four digit code played by the HomePod to establish a connection.

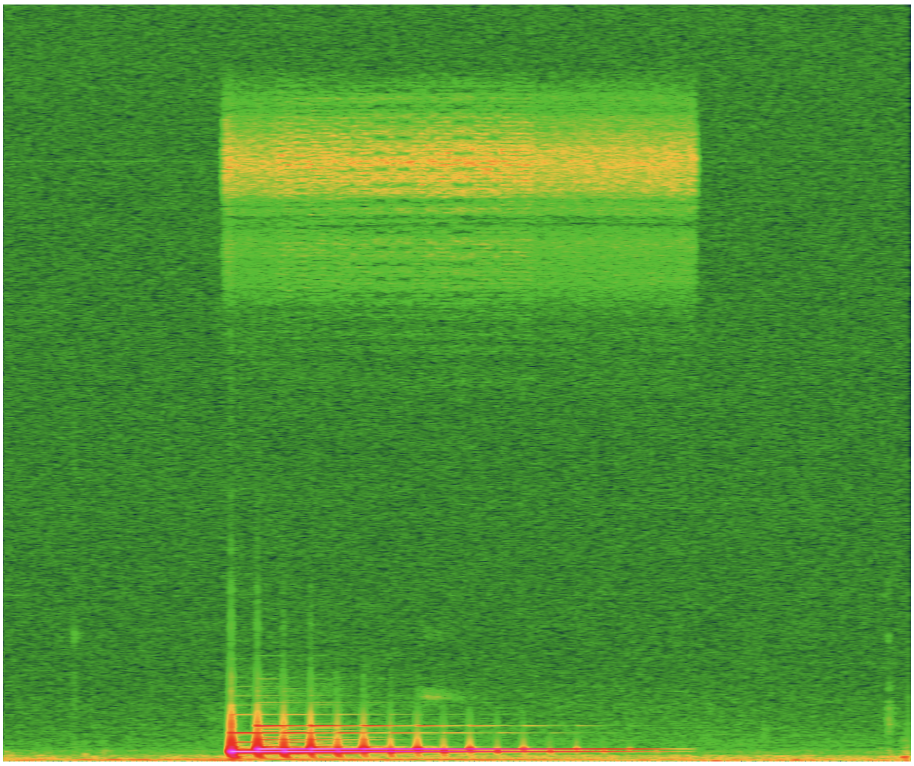


Fig. 2.1. Spectrogram of the Apple HomePod configuration tones

As shown in Figure 2.1, the HomePod configuration tones are encoded as near ultrasonic frequencies, clearly visible in the top portion of the spectrogram. The bottom portion shows the audible frequencies used for the melody played during the setup process. This spectrogram was obtained by manually recording the audio output of the HomePod during the setup process.

2.3.2. Google Chromecast Ultrasonic Pairing

Boris Smus, during his time as an engineer at Google, explored the use of near ultrasonic frequencies for data transmission in his blog [2]. This proof of concept ended up being implemented in the Google Chromecast device, allowing users to pair their devices using near ultrasonic tones, similar to the Apple HomePod configuration process.

2.3.3. Google Tone Extension

Google Tone, a Chrome browser extension, allowed for data transfer between computers using sound. In its initial implementation, Google Tone used near ultrasonic frequencies to transmit URLs between computers [3]. When a user wanted to share a URL, they would click the Google Tone extension, which emitted a series of near ultrasonic tones containing the URL. Nearby computers running the extension would get a notification prompting them to open the U.L.

2.4. Project Scope

Building upon these existing applications of near ultrasound and drawing inspiration from acoustic modems, this project explores the feasibility of transmitting keystroke data captured by a keylogger over the near ultrasound spectrum. By transmitting keystroke data over near ultrasound, the system aims to achieve a degree of stealth, as the transmission would be inaudible to human listeners. This method bypasses traditional communication channels, such as Wi-Fi or wired networks, which may be monitored or secured in sensitive environments. The use of near ultrasound offers a degree of plausible deniability, as the transmission could be attributed to other sources or dismissed as background noise.

As mentioned in the introduction, one of the main goals of the project is to serve as a teaching tool for students interested in digital signal processing, data transmission, and cybersecurity. This document provides a detailed overview of the project's design, implementation, and challenges encountered during development. The project's codebase is open-source and available on GitHub[1], allowing students and researchers to explore the implementation details and experiment with the system.

[1] See Appendix C for the project's GitHub repository.

3. DSP BASICS: FUNDAMENTALS OF DIGITAL SIGNAL PROCESSING

This chapter introduces the essential concepts of Digital Signal Processing (DSP), providing a foundation for understanding the principles that underpin modern signal manipulation and analysis techniques. We will explore the fundamental processes that enable the transformation of continuous, real-world signals into discrete, digital representations that can be processed by computers.

3.1. Introduction to Signals

In the context of DSP, a signal is defined as any quantity that varies over time or space, carrying information. Signals can be classified into two main categories: Continuous and Discrete. Continuous signals are defined at every instant of time and are represented by smooth, unbroken functions. On the other hand, discrete signals are defined only at specific, discrete points in time, often resulting from the sampling of continuous signals. When it comes to sound processing, one might be inclined to believe that continuous signals are the type of signal that musicians and researches use for their work, and while that might have been the case at one point in time (vinyls and analog recording), it simply isn't the case anymore. Ever since the rise of computers and digital recording, analog has faded into obscurity. This is mainly due to a technical limitation. Computers, with their "much-less-that-infinite" amounts of memory, are simply unable to store the potential infinity of values a continuous signal holds. The compromise is to instead "sample", this is, store the value of the signal at a specific instant, at regular intervals [4]. How often the signal is sampled results in a more accurate representation of the original signal. If it were possible to sample the signal an infinite amount of times, the stored signal would be identical to its continuous equivalent. The current standard in digital audio, known as CD quality, is to sample the signal 44,100 times per second, or at 44,100 Hertz (Hz). There are other sampling rates, such as 48,000 Hz, commonly used in professional settings, or even higher, such as 96,000 Hz and even 192,000 Hz, although these sampling rates are hardly used outside professional settings or audiophile equipment. Higher sampling rates result in higher fidelity, but at the cost of higher storage and compute requirements.

3.2. The Nyquist-Shannon Sampling Theorem

One might wonder why the sampling rate is 44,100 Hz. The current standard has a reason for being the standard. It serves as a compromise between fidelity and ease of use. The sampling rate isn't arbitrarily chosen, and depending on the requirements for the stored signal, a specific sampling rate must be picked. The Nyquist-Shannon sampling theorem is a fundamental principle in DSP that establishes the relationship between the sampling rate and the frequency content of the signal being sampled. The theorem states:

Definition 2.1: Nyquist-Shannon Sampling Theorem

To accurately represent a signal, the sampling rate must be at least twice the highest frequency component present in the signal. Mathematically, this is expressed as:

$$f_s \geq 2f_{max}$$

Where: f_s is the sampling frequency f_{max} is the maximum frequency component in the signal

This theorem is crucial for preventing aliasing, a phenomenon where high-frequency components are misrepresented as lower frequencies due to inadequate sampling. The human hearing range generally spans from 20 to 20,000Hz, although this high ceiling rapidly declines with age. If we apply the Nyquist-Shannon Sampling Theorem, we get that the sampling rate must be at least 40,000 Hz ($2 \cdot 20,000$ Hz) to accurately capture the full range of human hearing. This is why the standard sampling rate of 44,100 Hz was chosen, as it provides a slight margin above the theoretical minimum, ensuring an accurate representation of audio signals.

3.3. Quantization: Discrete Amplitude Representation

Once a signal has been sampled in time, the next step in digitization is quantization. This process involves mapping the continuous range of amplitude values to a finite set of discrete levels. Note that there is a difference between the continuous signal, which is what has been mentioned up until this point, and the continuous possible range of values that the amplitude can take, given a specific sample. Essentially, quantization chops this continuous range into a fixed number of intervals, like a ruler marking off centimeters instead of allowing for infinite fractions.

Each sampled value is then assigned to the nearest quantization level within these intervals. This replacement of the original sample value with the quantized value inevitably introduces a small amount of error, known as quantization noise. The number of these quantization intervals, and therefore the precision of our amplitude representation, is determined by the bit depth of the digital system. For example, 8-bit quantization provides 256 levels (2^8). 16-bit quantization offers a dramatic increase to 65,536 levels (2^{16}), providing significantly greater resolution. Going even further, 24-bit quantization, used in professional settings, provides more levels than the human ear can even distinguish, making it unnecessary for end use.

3.4. The Analog-to-Digital Conversion (ADC) Process

Having learned about sampling and quantization, we're ready to convert an analog signal to its digital representation. This process, known as Analog-to-Digital Conversion (ADC), can be summarized in the following steps:

1. An analog (continuous-time, continuous-value) signal is input to the system.
2. The signal is sampled at regular intervals determined by the sampling rate.
3. The sampled values are quantized to the nearest available digital representation.
4. The quantized values are encoded into binary numbers for digital storage or processing.

This process forms the basis for all digital representations of analog signals, enabling the wide range of digital signal processing techniques used in modern technology.

4. TRANSMITTING DATA OVER SOUND

This chapter delves into the fundamental concepts and techniques employed in transmitting digital data through acoustic channels. We begin by exploring the basics of binary data representation and encoding before progressing to modulation techniques, with particular emphasis on Frequency Shift Keying (FSK)—a method we'll implement in our practical work.

4.1. Binary Data and Bytes

All digital data is represented as a sequence of bits. Each bit can take one of two values: 0 or 1. This binary system, also known as base-2, forms the foundation of digital communication and computation. The number of unique symbols used in a numeral system is referred to as its "radix" or "base." While binary might seem unintuitive at first glance, this perception stems from our familiarity with the decimal system (base-10). Our preference for base-10 is largely anthropocentric, rooted in the fact that we possess ten fingers, making it a natural counting system. In fact, the word digit stems from the Latin term "digitus" which is also a synonym for finger. Computers, however, operate on a fundamentally different principle. Their architecture is built upon semiconductors, which can exist in two distinct states: on or off. This inherent limitation of computer hardware naturally lends itself to a binary system, where these two states can be easily represented. It is important to note that the chosen base doesn't limit how many numbers can be represented. All bases can represent arbitrarily high numbers. The main difference is how efficiently different bases can represent them. For instance, another popular base is hexadecimal. In this base, the decimal system is extended with 6 new symbols (A, B, C, D, E and F). All these bases can represent the same number in different ways. Note how in Table 4.1 the same number needs a different number of digits in order to be represented depending on the chosen base.

4.2. ASCII Encoding

One common way to encode text data is through the American Standard Code for Information Interchange (ASCII). ASCII is a character encoding standard that represents each text character as a 7-bit integer value. For example, the lowercase letter 'a' is represented by the decimal value 97 (binary 1100001).

When transmitting ASCII-encoded text, we would send the binary representation of each character's ASCII value. For instance, to send the word "hello", we would transmit the following sequence of bits: [**h**: 01101000, **e**: 01100101, **l**: 01101100, **l**: 01101100, **o**: 01101111]

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table 4.1. DECIMAL, BINARY, AND HEXADECIMAL EQUIVALENTS

4.3. Modems

The transmission of digital data over analog media, such as sound waves, necessitates a method of encoding information into waveforms. Fortunately, this field has been extensively researched and is widely applied, with most people having directly experienced its implementation. Modems, short for Modulator-Demodulators, are devices that perform the crucial task of modifying signals to transmit data over sound waves. These devices have been ubiquitous in various technologies, from FM radio to early internet connections in the late 1990s and early 2000s.

There are several approaches to modifying a signal for data transmission, primarily focusing on two main parameters: amplitude (signal strength) and frequency. These parameters give rise to the well-known terms AM (Amplitude Modulation) and FM (Frequency Modulation). For this project, we have chosen to employ Frequency Modulation due to its superior efficiency compared to Amplitude Modulation and its conceptual simplicity.

4.4. Frequency Shift Keying (FSK)

Building on the concept of Frequency Modulation, our project implements a simple form of modulation called Frequency Shift Keying (FSK). With this method, we represent different digital values by using different frequencies. In its most simple form, known as

Binary FSK (BFSK), we use two frequencies: one to represent a binary 0, and another to represent a binary 1. While the frequencies can be any pair, we have chosen to use frequencies in the near ultrasound spectrum. These frequencies, while below the maximum hearing ceiling of humans, are still high enough that most adults will not be able to hear them.

In our implementation, we'll use the following frequencies:

- For binary 0: 18 kHz
- For binary 1: 18.5 kHz

In this document, however, another pair of frequencies is used, in order to use easy to visualize figures:

- For binary 0: 100 Hz
- For binary 1: 1000 Hz

4.5. Symbol Duration

Knowing the frequencies, there is another parameter that must be selected: the symbol duration. This means, how long each frequency is transmitted for in order to represent one bit. This parameter is crucial, as it affects both our data rate and the reliability of our transmission. A longer duration makes it the message easier to be detected but it reduces the data rate, while a shorter duration increases the data rate but may make it more difficult to accurately detect the frequencies.

For this implementation, we'll use a symbol duration of 0.02 seconds. As with the frequencies earlier, the document will use a symbol duration of 0.01 seconds.

4.6. Encoding and Modulation

With the foundational concepts established, we can now outline the procedure for encoding and modulating our data:

1. **ASCII Conversion:** Obtain the ASCII values for each character in the text.
2. **Binary Representation:** Convert the ASCII values into their corresponding binary representations.
3. **Frequency Shift Keying (FSK):** For each bit in the binary data stream:
    - If the bit is a 0, generate a sine wave at 18 kHz for the duration of the symbol period.

- If the bit is a 1, generate a sine wave at 18.5 kHz for the duration of the symbol period.

4. **Signal Concatenation:** Concatenate the generated tones sequentially to form the final audio signal ready for transmission.

4.6.1. Example

In order to understand exactly what is done, let's walk through a step-by-step example. Let's transmit the character "a".

Before we transmit, we must set a few parameters:

- Sampling rate (F_S): 44100 Hz (standard audio sampling rate)
- Symbol duration (T_{symbol}): 0.01 seconds (as mentioned for the document examples)
- Frequency for 0 (f_0): 100 Hz
- Frequency for 1 (f_1): 1000 Hz

Now, let's go through the steps:

1. **ASCII and Binary Conversion:** The character 'a' is represented by the ASCII value 97, which has a binary representation of 01100001.

2. **FSK Modulation:** Each bit in the binary sequence is mapped to a corresponding frequency: 0 to f_0 and 1 to f_1. This results in the following frequency sequence:

$$(f_0, f_1, f_1, f_0, f_0, f_0, f_0, f_1)$$

   Each tone will last for the symbol duration of 0.01 seconds.

3. **Signal Generation:** The signal is generated using the following mathematical function. For each bit:

   - We calculate the number of samples per symbol:

$$\text{samples\_per\_symbol} = T_{\text{symbol}} \times F_S = 44,100 \times 0.1 = 441 \tag{4.1}$$

   - We select the frequency based on the bit value:

$$f = \begin{cases} f_0 & \text{if bit} = 0 \\ f_1 & \text{if bit} = 1 \end{cases} \tag{4.2}$$

- The signal for the n-th bit can be mathematically expressed as:

$$s_n(t) = \cos(2\pi f_i t), \quad 0 \leq t < T_s$$

However, since we're working with discrete-time signals, we need to work with samples.

- For each sample in the symbol, we generate a cosine wave. This requires tracking the *phase* of the wave, which indicates our current position within the cosine wave cycle:

$$\text{phase} = \text{phase} + 2\pi f / F_S \tag{4.3}$$

where:

  - $2\pi f$**:** Represents the angular frequency (how fast the wave oscillates).
  - $/F_S$**:** Dividing by the sampling rate normalizes the angular frequency to the time between samples, giving the phase increment per sample.
  - +phase: Adds the calculated increment to the current phase, ensuring a continuous progression through the cosine wave.

- The calculated phase is then used to determine the amplitude of the cosine wave for the current sample:

$$\text{sample} = \cos(\text{phase}) \tag{4.4}$$

- The phase is wrapped around 2π to prevent overflow. This is because a cosine wave repeats every 2π, and wrapping the phase ensures numerical stability without affecting the generated waveform. Mathematically, this is not necessary, but it is a common practice in signal processing to prevent numerical instability.

$$\text{if phase} > 2\pi, \text{ then phase} = \text{phase} - 2\pi \tag{4.5}$$

4. **Signal Concatenation:** The discrete-time signals for each bit are concatenated to form the final audio signal representing the character "a".

The resulting audio signal, as shown in Figure 4.1, consists of 8 distinct tones, each lasting 0.01 seconds, for a total duration of 0.08 seconds, or $441 \times 8 = 3528$ samples to transmit the single character 'a'. For visualization purposes, a separator, represented by the red line, has been added between each tone.

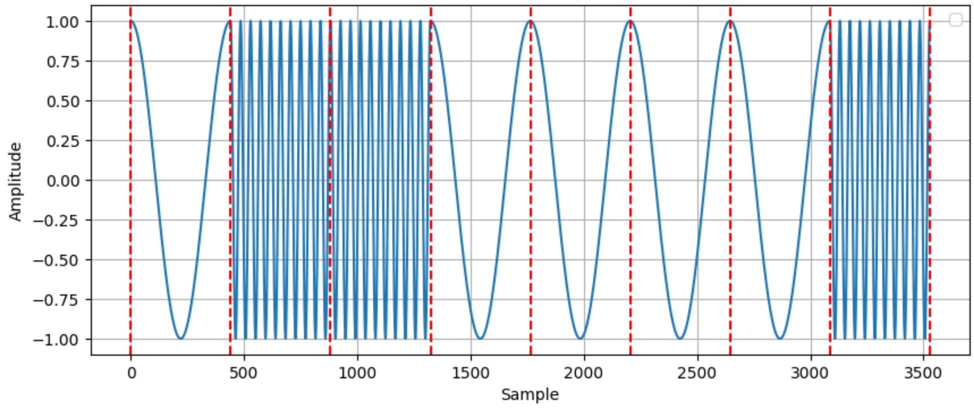


Fig. 4.1. Generated signal representing the character "a"

4.7. Demodulation and Decoding

On the receiving end, the reverse process is performed to recover the original data:

1. **Segmentation:** Divide the received audio signal into segments, each corresponding to the symbol duration.
2. **Frequency Detection:** For each segment, determine the dominant frequency.
3. **Binary Conversion:** Convert the detected frequencies back into their corresponding binary values (18 kHz to 0, 18.5 kHz to 1).
4. **Byte Grouping:** Group the binary data into 8-bit bytes.
5. **ASCII Conversion:** Convert the bytes back into their corresponding ASCII values.
6. **Text Reconstruction:** Convert the ASCII values into the original text message.

The main challenge in this process lies in accurately detecting the frequency of each segment. In an ideal world, we would receive a perfect sine wave for each symbol, making frequency identification trivial. However, real-world signals are all but perfect sines[2]. The noise present even in the quietest of environments can distort the signal, making it difficult to accurately identify the frequency.

In the next chapter, we'll explore the Fourier Transform in depth, understanding its principles and how it can be applied to our demodulation process. This fundamental concept will not only enable us to complete our FSK demodulation but will also provide insights into a wide range of signal processing applications.

[2]Noise, interference, and other imperfections in the transmission medium can distort the transmitted signal, making it more challenging to accurately identify the original frequencies.

5. THE FOURIER TRANSFORM

The Fourier Transform is a fundamental mathematical tool used in signal processing to decompose a signal into its constituent frequencies. This process is not at all trivial, as it would be similar to trying to determine the ingredients of a cake by looking at it, or unmixing paint.

The Fourier Transform provides a way to transform a signal from the *time domain*[3] to the *frequency domain*. The frequency domain represents the signal as a function of frequency, instead of time, showing the different frequencies that make up the signal.

The Fourier Transform can be applied to both continuous-time and discrete-time signals, resulting in the Continuous-Time Fourier Transform (CTFT) and the Discrete-Time Fourier Transform (DTFT), respectively. In the context of this project, since we're dealing with discrete samples, we'll focus on the Discrete-Time Fourier Transform (DTFT).

Note that the mathematical details of the Fourier Transform are beyond the scope of this document, and we'll focus on its practical implications for signal processing. We will show the definition, but we won't delve into the mathematical intricacies of the transform.

5.1. The Continuous-Time Fourier Transform (CTFT)

The Continuous-Time Fourier Transform (CTFT) is a mathematical tool used to analyze continuous-time signals in the frequency domain. It decomposes a continuous signal into its constituent frequencies, revealing the amplitude and phase of each frequency component.

Definition 1.1

The CTFT is defined as:

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt$$

where:

- $x(t)$ is the input signal in the time domain.
- $X(f)$ is the output signal in the frequency domain.
- f is the frequency variable.
- j^a is the imaginary unit ($\sqrt{-1}$).

[3]The time domain represents the signal as a function of time, showing how the signal changes over time. It's what we have been working with up until now.

[a]In this document, we use j to represent the imaginary unit, as is common in engineering. In mathematics, the imaginary unit is typically denoted by i.

In order to give a slight intuition of what the CTFT does, one can imagine it as correlating the input signal with a series of sine and cosine waves of varying frequencies. Suppose we have a signal that is a pure sine wave at 100 Hz. The CTFT would reveal a peak at 100 Hz, indicating the presence of this frequency component in the signal. The way this peak was obtained is by correlating the input signal with an infinite number of sine waves of varying frequencies, and the peak at 100 Hz indicates that the signal correlates most with the 100 Hz sine wave.

It's worth noting that the CTFT operates on continuous signals, making it unsuitable for digital signal processing. For discrete signals, we use the Discrete-Time Fourier Transform (DTFT), which we'll explore in the next section.

5.2. The Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) is the discrete counterpart of the Continuous-Time Fourier Transform (CTFT). While the CTFT operates on continuous signals, the DFT processes discrete signals, making it the only viable option for digital signal processing.

Definition 2.1: The Discrete Fourier Transform

Mathematically, the DFT is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-j2\pi kn}{N}}$$

where:

x_n is the input signal in the time domain.

X_k is the output signal in the frequency domain.

N is the number of samples in the input signal.

k is the frequency index, ranging from 0 to N-1.

j is the imaginary unit ($\sqrt{-1}$).

The resulting complex numbers $X(k)$ represent the amplitude and phase of each frequency component in the signal. The magnitude of $X(k)$ indicates the strength of the frequency component, while the phase indicates its relative timing.

The DFT takes a finite sequence of N samples of a discrete-time signal and transforms it into a sequence of N complex numbers representing the frequency components of the signal. Again, we won't delve into the mathematical details of the DFT, but it's essential to understand its purpose and utility in digital signal processing. The main difference is that the DFT has a limited number of frequency components, determined by the number of samples in the input signal.

5.3. The Fast Fourier Transform (FFT) Algorithm

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT) of a sequence of samples. The FFT reduces the computational complexity of the DFT from $O(N^2)$ to $O(N \log N)$, making it significantly faster for large input sizes. The DFT is incredibly useful, but its computational requirements can be prohibitive for real-time signal processing applications. Thanks to the FFT, we can perform complex frequency analysis in a fraction of the time it would take using the standard DFT algorithm, making it viable for real-time applications. We won't delve into the details of the FFT algorithm, but it's essential to understand its significance in digital signal processing.

5.4. Uses of the Fourier Transform in Frequency Identification

In this project, the Fourier Transform is used to identify the frequencies present in the received audio signal. By applying the DFT to the received signal, we can determine the dominant frequencies, which correspond to the encoded data.

As an example, let's apply the DFT to the signal generated in Section 4.6.1 to identify the frequencies used to encode the character 'a'.

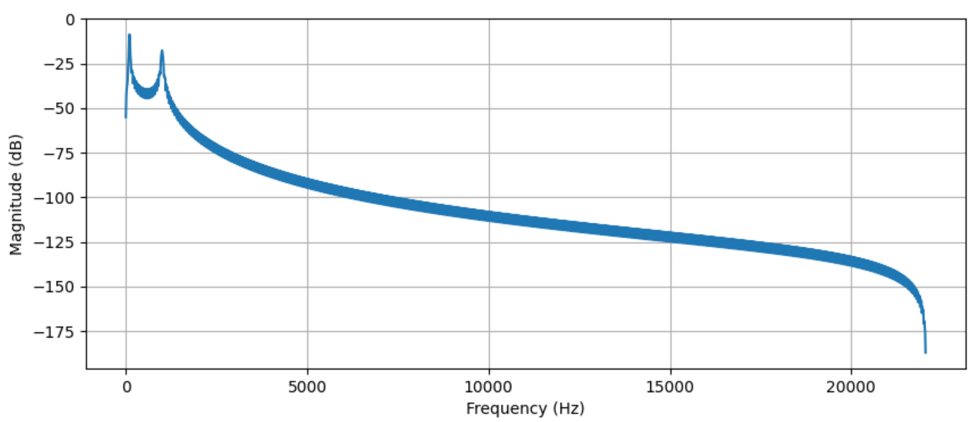


Fig. 5.1. Fourier Transform of signal with two peaks at 100Hz and 1000Hz, respectively

As shown in Figure 5.1, the Fourier Transform of the signal reveals two distinct peaks at 100 Hz and 1000 Hz, corresponding to the frequencies used to encode the binary data. The figure illustrates how the Fourier Transform can identify the frequency components present in a signal, but it is not able to determine the exact time at which these frequencies occur, as the Fourier Transform operates on the entire signal at once. If we want to determine the exact time at which these frequencies occur, we need to apply the fourier transform to smaller segments of the signal, particularly the symbol duration. This way, we can determine the frequency content of the signal at each symbol, and the peak frequency will indicate the encoded bit. This process is known as Short-Time Fourier Transform (STFT), which is a simple modification of the Fourier Transform that allows us to analyze the frequency content of a signal over time.

Segmenting the signal into smaller windows and applying the Fourier Transform to each window allows us to track the frequency content of the signal over time, enabling us to accurately identify the encoded data. Let's continue with the signal generated in Section 4.6.1 and apply the Fourier Transform to each segment to decode the original data. We will apply the Fourier Transform to the first two segments of the signal shown in Figure 4.1 to determine the encoded bits.

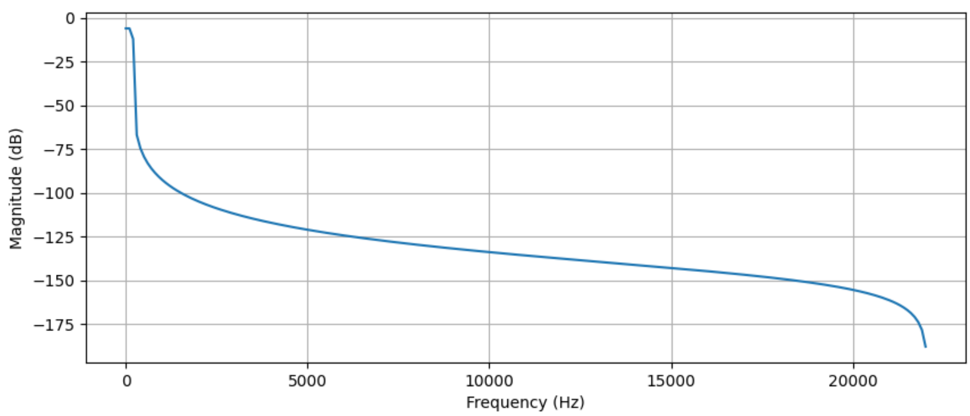


Fig. 5.2. Fourier Transform of first segment with a peak at 100Hz

With the Fourier Transform applied to the first segment, we can see a peak at 100 Hz, indicating that the first bit is a 0.

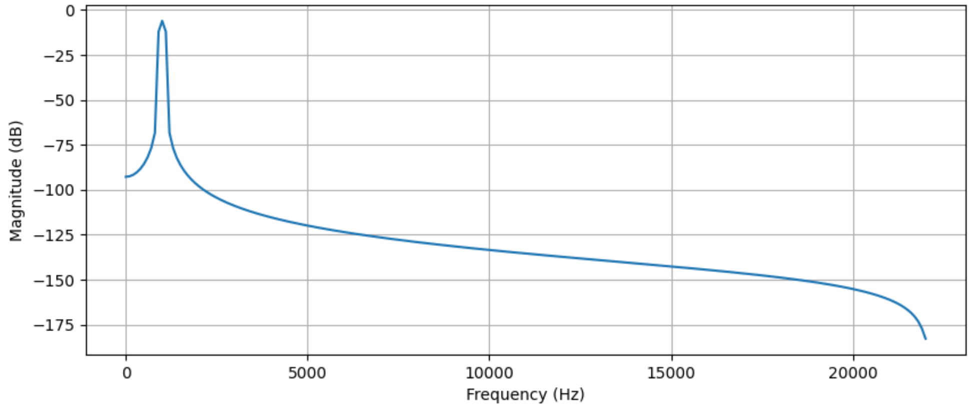


Fig. 5.3. Fourier Transform of second segment with a peak at 1000Hz

Similarly, the Fourier Transform of the second segment shows a peak at 1000 Hz, indicating that the second bit is a 1. This process is repeated for each segment, allowing us to accurately decode the original data from the received signal.

5.5. Synchronization Challenges

While the above process seems straightforward, there's a significant challenge we need to address: synchronization. When demodulating the signal, we need to know exactly where each symbol begins and ends. If our segmentation is off by even a small amount, we may misinterpret the frequencies and corrupt our data.

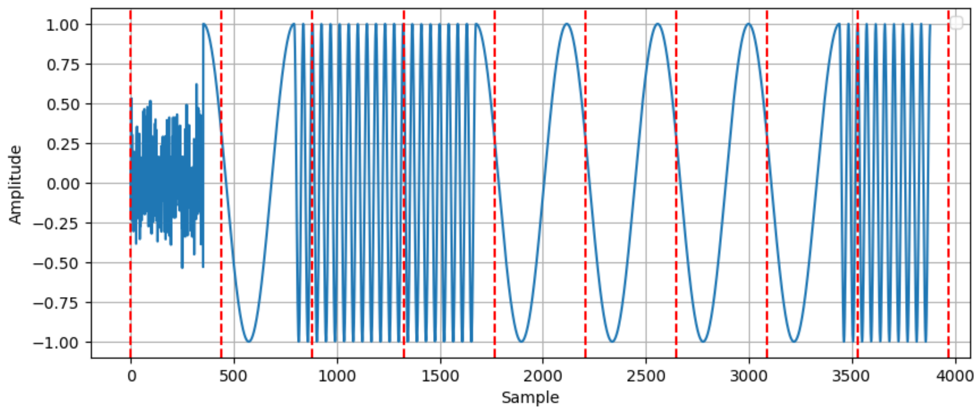


Fig. 5.4. Signal with Noise of Arbitrary Duration

To address this, we need to implement some form of synchronization mechanism. One approach is to add a "marker" or "preamble" at the beginning of our transmission that the receiver can use to align itself. This leads us to our next chapter, where we'll discuss the concept of a preamble in detail.

6. THE PREAMBLE: SYNCHRONIZING TRANSMISSION AND RECEPTION

6.1. Preamble

A preamble is a known sequence of bits transmitted before the actual message data. Its purpose is to facilitate synchronization between the modulator and the demodulator. Both parties have to agree beforehand on what the preamble will be. The known sequence is then transmitted right before the start of a message, and will aid in detection and synchronization.

Preambles are used in many transmission mechanisms. The 802.11 standard, or WiFi as we all know it, utilizes a preamble in order mark the beginning of the message [5] (frame is the technical name for an individual message in this context). Dial-up modems used a preamble to synchronize the transmitter and the receiver modem. In that case, the preamble consisted of 100 ms of V.21 marking tone. The marking tone, which refers to the frequency corresponding to binary 1 (as opposed to the space tone, which corresponds to binary 0), was either 1,270 Hz, for the calling modem, or 1,650 Hz, for the answering modem.

For our project, we will choose a slightly more sophisticated preamble. Instead of a single tone, we will send a specific sequence of 1's and 0's, and utilize a mechanism in order to detect it.

Preamble (13 bits)	Message (8 bits)

Fig. 6.1. 13-bit Preamble and Message

We now need a mechanism to find this preamble when this full signal is contaminated with noise.

6.2. Matched Filter

A matched filter is a filter designed to maximize the signal-to-noise ratio (SNR) at its output when a specific signal is present at its input. In this project, a matched filter is used to detect the presence of the preamble in the received audio signal. The operating principle behind matched filters lies in a special type of operation known as a convolution, which we'll explore in the next section.

6.3. Convolution

A convolution is an operation that can be done on two functions that produces a third one. Before we delve into the formal definition, it is important to understand the intuition behind it.

Suppose we have two functions, $f(t)$ and $g(t)$. There are two main operations we can perform on these two functions: addition and multiplication. Both operations, $f(t) + g(t)$ and $f(t) \cdot g(t)$ result in a third function, $y(t)$. Similarly, a convolution is an operation performed on two functions that results in a third one. Mathematically, the convolution of two functions is defined as:

Definition 3.1: Convolution Operation

The convolution of two functions $f(t)$ and $g(t)$ is the integral of the product of the two functions, with one of them flipped and shifted.:

$$y(t) = f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau)g(-\tau + t)d\tau$$

While the name of the operation might sound intimidating, the concept results in a very intuitive operation, and one that is particularly useful in signal processing. To better understand the convolution operation, let's walk through an example.

6.3.1. Example

Suppose we have two identical rectangular pulse functions, $f(t)$ and $g(t)$, as shown in Figure 6.2. We want to compute their convolution, $y(t) = f(t) * g(t)$.

The rectangular pulse functions $f(t)$ and $g(t)$ are defined as: $f(t)$ and $g(t)$ are defined as:

$$f(t) = g(t) = \begin{cases} 1 & \text{if } 0 \leq t \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

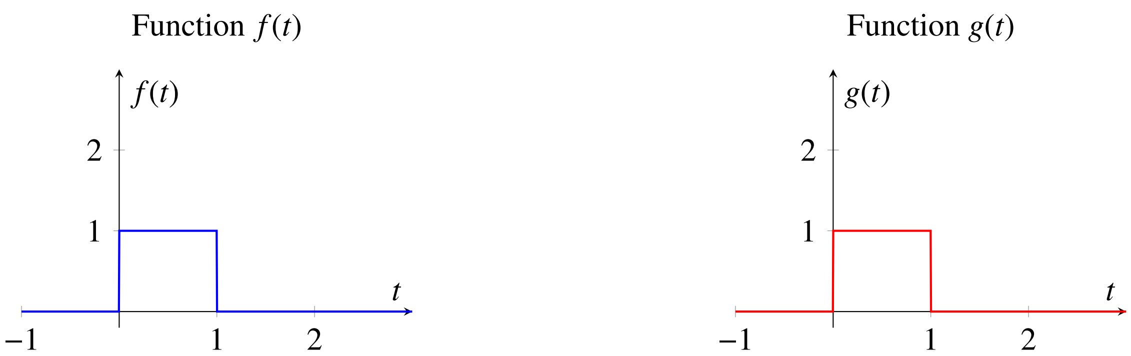


Fig. 6.2. Input functions $f(t)$ and $g(t)$

The convolution process can be visualized as follows:

First Step

The first step is to flip $g(t)$ horizontally. Doing so results in $g(-t)$, as shown in Figure 6.3.

Flipped Function $g(-t)$

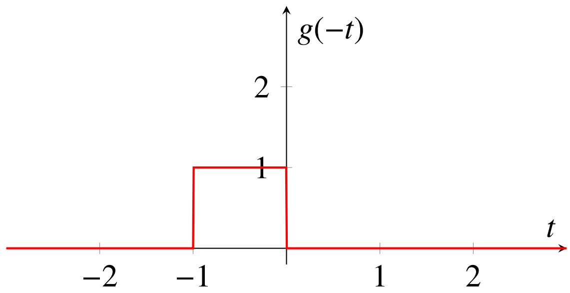


Fig. 6.3. Flipped function $g(-t)$

Second Step

The second step is to shift $g(-t)$ by τ. Shifting $g(-t)$ by τ results in $g(-\tau + t)$, as illustrated in Figure 6.4.

Shifted Function $g(-\tau + t)$

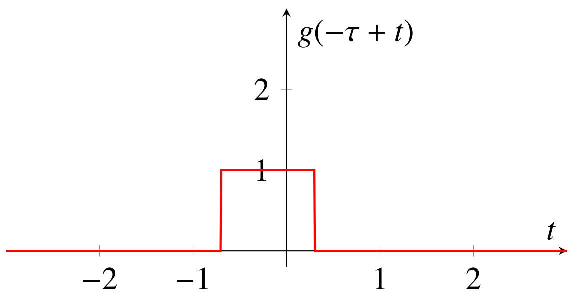


Fig. 6.4. Shifted function $g(-\tau + t)$ for $\tau = 0.3$

Third Step

We calculate the area of overlap between $f(t)$ and $g(-\tau + t)$, as shown in Figure 6.5. This area is the value of the product $f(t)g(-\tau + t)$ integrated over the overlapping region.

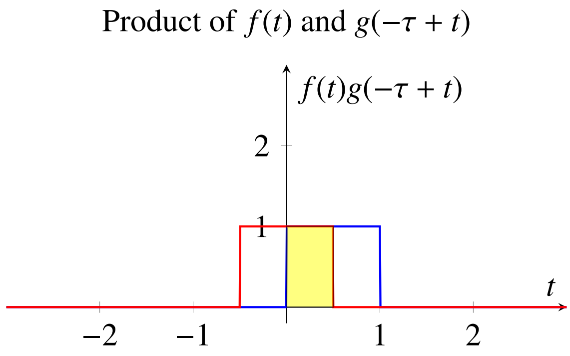


Fig. 6.5. Product of $f(t)$ and $g(-\tau + t)$ for a given τ

Fourth Step

As we vary τ, we calculate the area of overlap for each value of τ. Plotting the value of the area of overlap for all τ values gives us the convolution result $y(t) = f(t) * g(t)$, as shown in Figure 6.6.

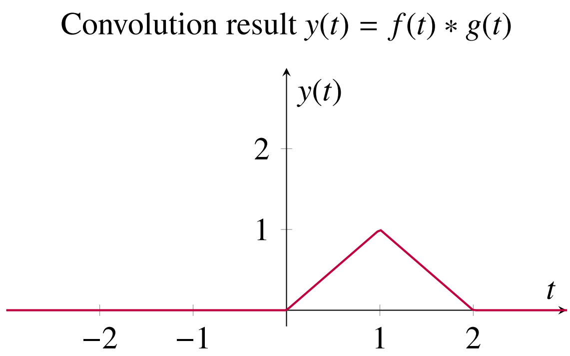


Fig. 6.6. Result of convolution $y(t) = f(t) * g(t)$

The resulting function $y(t)$ has a triangular shape, as shown in Figure 6.6. Note how the peak of the triangle occurs at the point of maximum overlap between the two functions $f(t)$ and $g(t)$.

While the continuous convolution is important for understanding the theoretical basis of matched filters, in practice, we work with discrete-time signals, hence, it's necessary to introduce the discrete version of the operation.

Definition 3.2: Discrete Convolution for infinite-length sequences

For two discrete sequences $f[n]$ and $g[n]$, their discrete convolution $y[n]$ is defined as:

$$y[n] = f[n] * g[n] = \sum_{k=-\infty}^{\infty} f[k]g[n-k]$$

Again, in practice we deal with finite-length sequences, so the final definition, and the one used for this project is the following:

Definition 3.3: Discrete Convolution for finite-lenth sequences

For sequences $x[n]$ of length N and $h[n]$ of length M, the discrete convolution is:

$$y[n] = \sum_{k=0}^{M-1} x[n-k]h[k], \quad \text{for } n = 0, 1, ..., N + M - 2$$

When the template aligns perfectly with the preamble in the received signal, it produces a peak in the output. This peak represents the point of maximum correlation. The matched filter maximizes the SNR at this peak, making it easier to detect the presence of the preamble even in noisy conditions.

Let's now explore how the matched filter is applied in the context of our project.

6.3.2. Applied Example

Recall Example 4.6.1, where we aimed to transmit the character "a". Let's add a preamble to that message, specifically the sequence "1111100110101". The complete signal will then be "1111100110101" followed by "01100001".

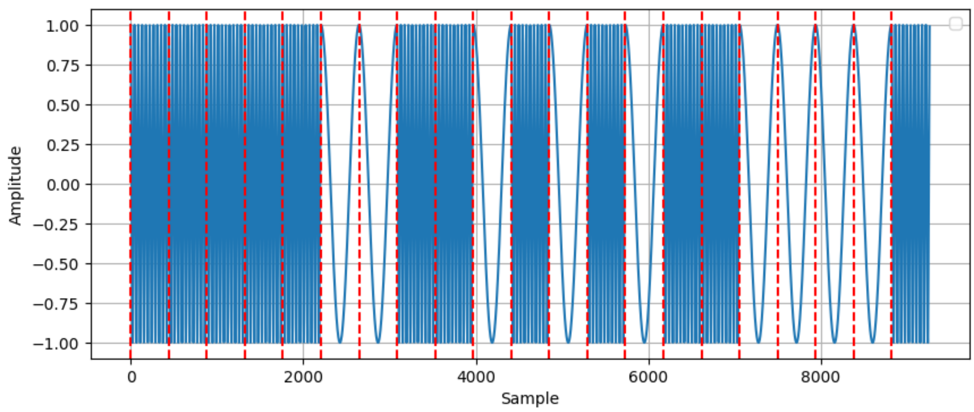


Fig. 6.7. Signal with Preamble

As we can see, the preamble is clearly visible at the beginning of the signal. The next step is to apply the matched filter to detect the preamble in the presence of noise.

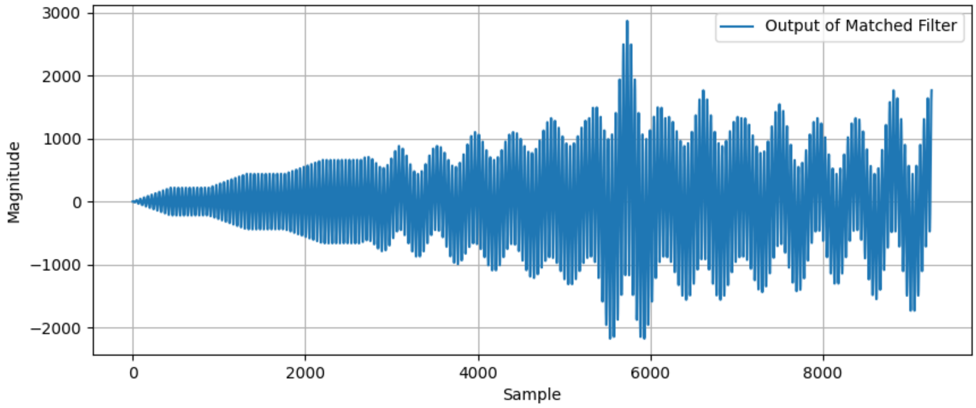


Fig. 6.8. Output of Matched Filter

The matched filter output, as shown in Figure 6.8, exhibits a clear peak at the beginning of the signal, corresponding to the preamble. As we saw earlier, this peak indicates the end of the preamble, so we can now safely begin demodulating the message data.

Let's apply some noise to the signal and see how the matched filter performs.

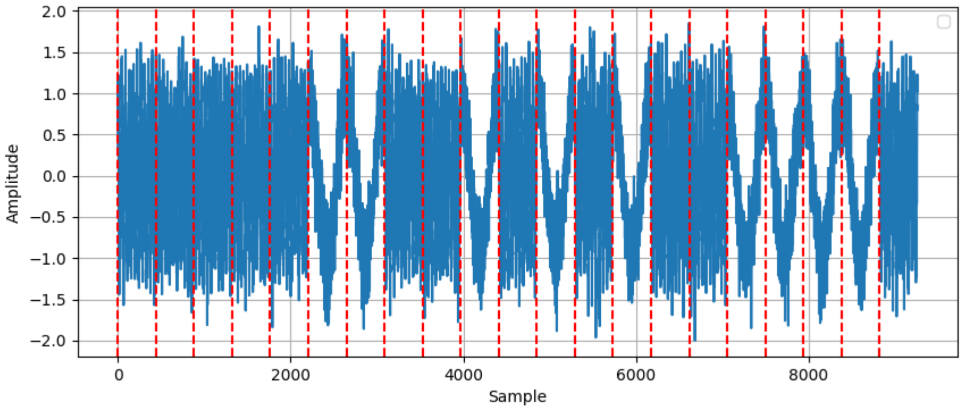


Fig. 6.9. Noisy Signal with Preamble

The signal in Figure 6.9 contains a small amount noise, so it is still possible to detect the preamble visually. This is a best-case scenario, as the noise is minimal and the preamble is still clearly visible. However, in practice, the noise level may be much higher, making it difficult to detect the preamble by visual inspection alone.

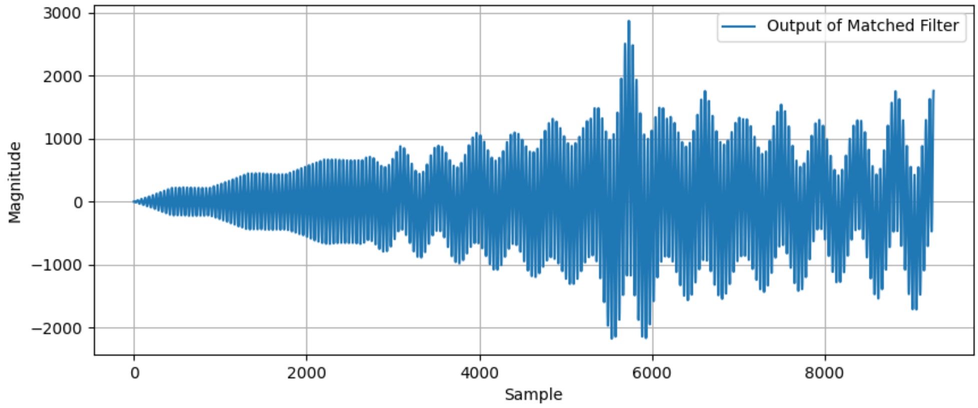


Fig. 6.10. Output of Matched Filter

Again, the matched filter output in Figure 6.10 shows a clear peak at the end of the preamble, indicating that it has been successfully detected.

As a final test, let's increase the noise level in the signal and observe the output of the matched filter.

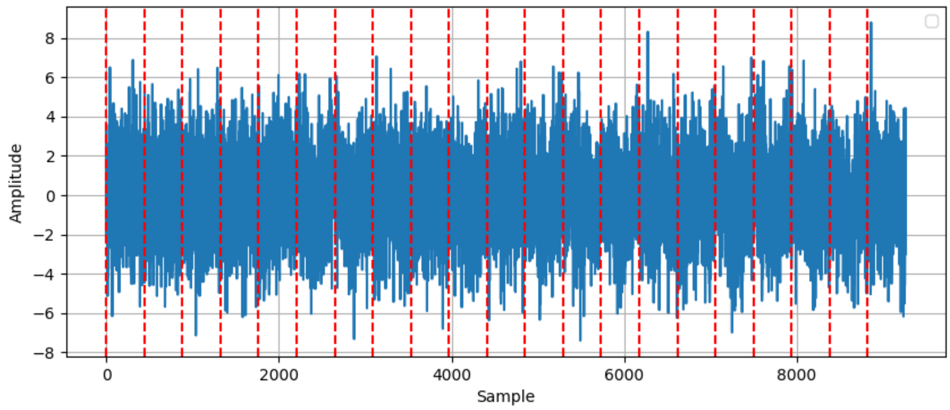


Fig. 6.11. Very Noisy Signal with Preamble

Now, the signal in Figure 6.11 contains a significant amount of noise, making it impossible to distinguish the preamble visually. This is a more realistic scenario, where the noise level is high and the preamble is fully obscured by the noise.

Applying the matched filter to this noisy signal, we obtain the following output:

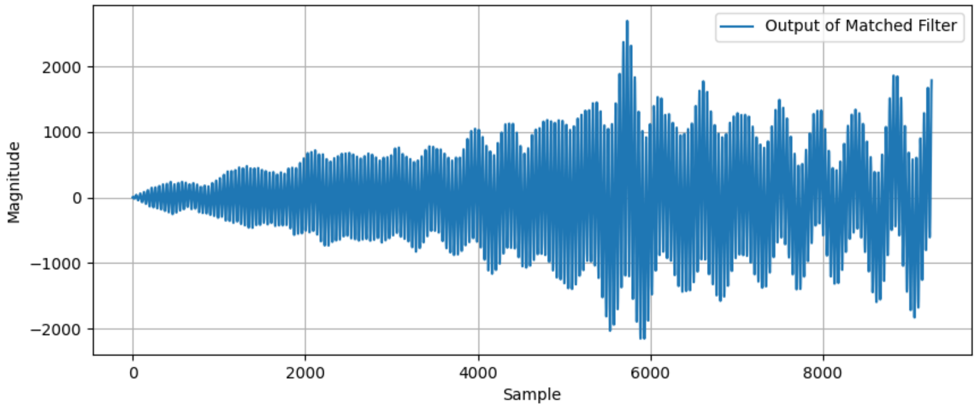


Fig. 6.12. Output of Matched Filter

The matched filter output in Figure 6.12 still exhibits a clear peak at the end of the preamble, even in the presence of high noise levels. Matched filters are robust to noise and can reliably detect the presence of the preamble, enabling accurate synchronization and demodulation of the message data.

6.4. Threshold

In practice, the matched filter output is often noisy, and it can be challenging to determine the exact point of the peak corresponding to the preamble. To address this issue, a threshold is applied to the matched filter output.

The threshold is a predefined value that determines whether a peak is present in the output. If the matched filter output exceeds the threshold, it is considered a valid peak, indicating the presence of the preamble. For our project, we use a simple thresholding mechanism based on the standard deviation of the matched filter output. By setting the threshold as a multiple of the standard deviation, we can adjust the sensitivity of the peak detection. The function `compute_statistical_threshold` in the demodulator code calculates the threshold based on the matched filter output. After testing, we found that a threshold of 3 times the standard deviation provides reliable peak detection in the presence of noise.

```python
def compute_statistical_threshold(output, factor=3):
    logging.debug("\nComputing statistical threshold...")

    mean_value = np.mean(output)
    std_dev = np.std(output)

    # Threshold is set to mean plus a certain factor times the
       standard deviation
    threshold = mean_value + factor * std_dev
    logging.debug("Statistical Threshold:", threshold)
```

```
    return threshold
```

6.5. Barker Code

While the preamble we've used so far demonstrates the effectiveness of matched filters, there are specific sequences designed to optimize synchronization performance. One such class of sequences, known as Barker codes, offers superior properties for preamble detection and synchronization. Barker codes are a set of finite sequences with exceptional autocorrelation characteristics. These sequences produce very low sidelobes in their autocorrelation function as seen in Figure 6.13, which translates to reduced false positives when used as preambles. Notice one sharp peak at the center, with minimal sidelobes.

The unique properties of Barker codes make them invaluable in various applications, including wireless communication systems and radar technology [6]. Barker codes have found widespread use in various fields. For instance, they are employed in the original IEEE 802.11 standard for wireless LANs [5], commonly known as Wi-Fi.

One of the most well-known Barker codes is the 13-bit Barker code, which has the following sequence:

Definition 5.1: Barker Code

A Barker code is a binary sequence that has a low autocorrelation sidelobe level. The 13-bit Barker code is one of the most commonly used Barker codes, with the following sequence:

$$1111100110101$$

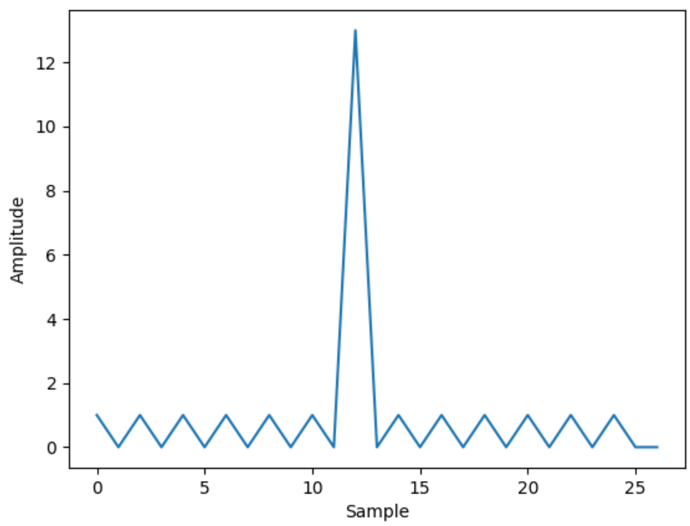


Fig. 6.13. 13-bit Barker Code Autocorrelation

When used as a preamble, the Barker code produces a distinct autocorrelation peak, making it easy to detect and synchronize the received signal.

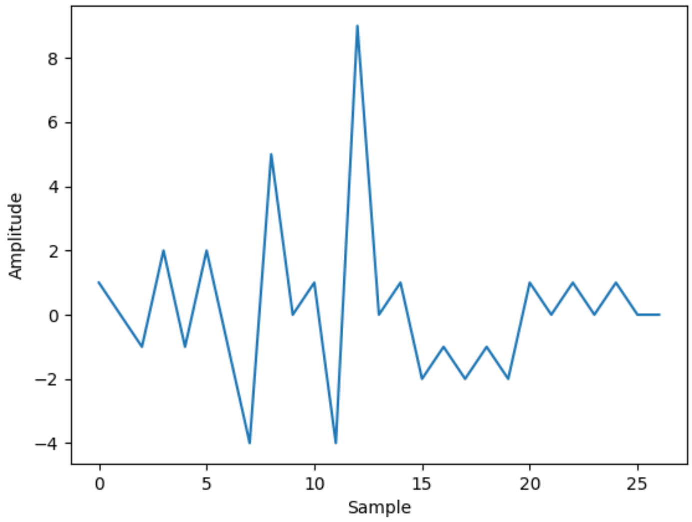


Fig. 6.14. Non-Barker Sequence Autocorrelation

In contrast, a non-Barker sequence, as shown in Figure 6.14, exhibits higher sidelobes in its autocorrelation function, making it more challenging to detect and synchronize.

The preamble used in *Example* 6.3.2, and in the project is the 13-bit Barker code.

Barker Code	**Sequence(s)**
2	10, 11
3	110
4	1101, 1110
5	11101
7	1110010
11	11100010010
13	1111100110101

Table 6.1. BARKER CODES

For comparison purposes, let's visualize the matched filter output when using non-Barker sequences as preamble. We'll use the sequence "1010101010101" as an example.

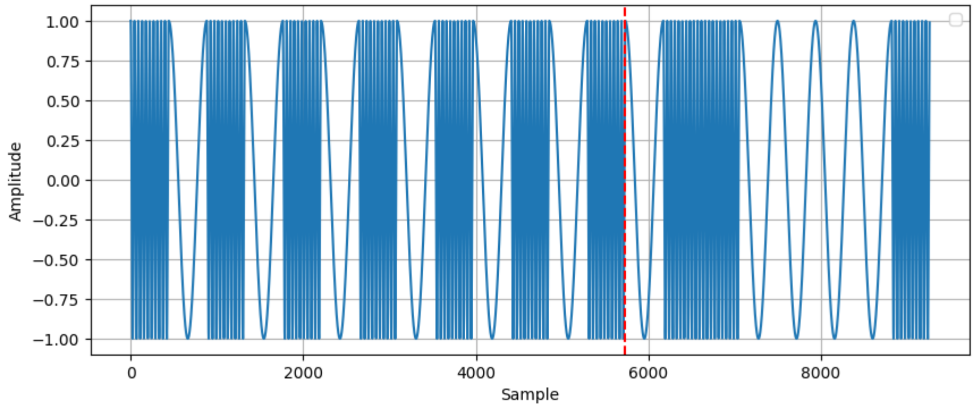


Fig. 6.15. 13-bit Non-Barker Preamble

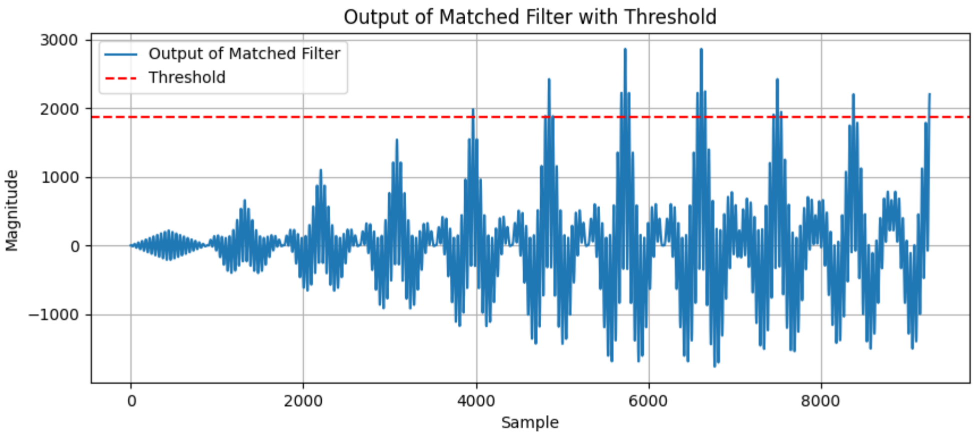


Fig. 6.16. Visualization of matched filter with 13-bit Non-Barker Preamble

As we can see in Figure 6.16, the matched filter output for the non-Barker sequence exhibits multiple peaks, making it difficult to detect the preamble accurately.

7. DEMODULATING THE MESSAGE

With the preamble successfully detected and the signal synchronized, we can now proceed to demodulate the message data. In theory, this process may seem straightforward: identify the frequency of each symbol, convert it to binary, group the bits into bytes, and decode the ASCII values, as seen in Section 4.7. However, the implementation details are not as simple, especially if the demodulator needs to handle real time processing.

7.1. Motivation

To illustrate the issue, we'll focus on a single character: the letter "a". As discussed in the modulation chapter, "a" is represented by the binary sequence "01100001". Our signal begins with the preamble "1111100110101", followed by this character data.

For this example, let's assume:

- The complete signal (preamble + character) spans 0.1 seconds
- We're analyzing a 1-second audio window
- The target signal could appear anywhere within this window, but fully contained within it

If we were dealing with an isolated 1-second sample, the process would be straightforward. As we saw in the previous chapter, we could apply the matched filter to detect the preamble, and once detected, we could proceed with demodulating the message data.

However, in a real-time scenario, we don't have the luxury of analyzing the entire signal at once, mainly because the signal is continuously streaming. This poses a significant challenge: How do we detect the preamble and demodulate the message data in real time?

7.2. Frames and Queuing

In audio processing, a common approach to handling real-time data is to divide the signal into frames. A frame is nothing more than a fixed-length segment of the signal, containing a certain number of samples. It is the fundamental unit of processing in real-time audio applications. From now on, we'll refer to the signal segments as frames. To handle real-time processing, we need to process the signal frame by frame. As time progresses, we'll receive new frames of audio data, and we need to process each frame as it arrives. We will store the frames in a queue, which as the name suggests, and as you may have guessed, follows the First-In-First-Out (FIFO) principle. This is a common data structure used in real-time processing to manage the order of incoming data.

f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	⋯

Fig. 7.1. Visualization of a Queue of Frames

As seen in Figure 7.1, the frames (f1, f2, f3, ...) are stored in a queue. As new frames arrive, they are added to the end of the queue. When we process the frames, we remove them from the front of the queue in the order they were received.

It may seem as if this fixes the problem, but there's a catch. The preamble may not be fully contained within a single frame. A first approach would be to increase the frame size to ensure the preamble fits within a single frame. However, this doesn't solve the underlying issue. What if the preamble is split between two frames? This could happen no matter how large we make the frames, as it's a matter of chance whether the preamble aligns perfectly with the frame boundaries. Taking this to the extreme would mean having frames as large as the entire signal, which defeats the purpose of processing in frames, and as we saw earlier, is not feasible in real-time processing.

We need to be able to process the preamble even if it's split between frames.

7.3. Buffering

We need to be able to process multiple frames together to detect the preamble. To do this, we will use a buffer, which will store a certain number of frames.

Definition 3.1: Buffer

A buffer is a temporary storage area that holds data while it's being processed. In this case, the buffer will store frames of the audio signal, but it can be used in multiple applications.

For example, in audio processing, a buffer can store a segment of the audio signal that is being processed, and in networking, a buffer can store packets of data that are being transmitted or received.

The number of frames we process together, known as the buffer size, is a critical parameter. We can calculate the minimum buffer size required to detect the preamble by considering the length of the preamble and the frame size. The preamble used in this project is the 13-bit Barker code, which is 13 bits long.

As we saw in Example 4.6.1, the number of samples per symbol is:

$$\text{samples\_per\_symbol} = T_{\text{symbol}} \times F_S = 44,100 \times 0.1 = 441 \tag{7.1}$$

Hence, the number of samples required to represent the preamble is:

$$\text{samples\_per\_preamble} = 13 \times \text{samples\_per\_symbol} = 13 \times 441 = 5,733 \tag{7.2}$$

The buffer size is then given by:

$$\text{buffer\_size} = \text{samples\_per\_preamble}/\text{frame\_size} = 5,733/1024 \approx 5.6 \tag{7.3}$$

The buffer size should be rounded up to the nearest integer, so we need a buffer size of at least 6 frames to detect the preamble. This ensures that the preamble is fully contained within the buffer, even if it's split between frames.

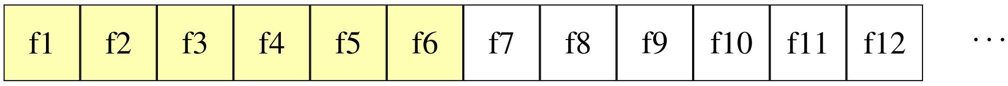


Fig. 7.2. Visualization of queue with buffer size of 6 frames

With a buffer size of 6 frames, as shown in Figure 7.2, we can ensure that the preamble is fully contained within the buffer. In order to be able to detect the preamble correctly, we need to slide the buffer along the frames, and apply the matched filter to the buffer each time. This type of buffer is known as a ring buffer, and it's a common technique used in signal processing to handle real-time data.

7.4. Ring Buffer

Definition 4.1: Ring Buffer

A ring buffer, also known as a circular buffer, is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. When the buffer is full, new data is written starting from the beginning of the buffer, overwriting the oldest data. This creates a circular effect, hence the name "ring buffer".

Let's visualize the operation of a ring buffer with a buffer size of 4 frames. Note that the buffer is only the yellow highlighted frames, and the other frames are not part of the buffer. They are shown for visualization purposes.

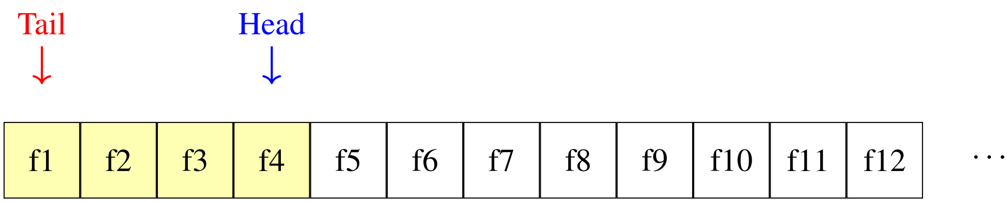


Fig. 7.3. Visualization of a Ring Buffer of Size 4

In Figure 7.3, the ring buffer contains 4 frames, f1, f2, f3, and f4. As new frames arrive, they are added to the end of the buffer. When the buffer is full, the oldest frame is overwritten by the new frame.

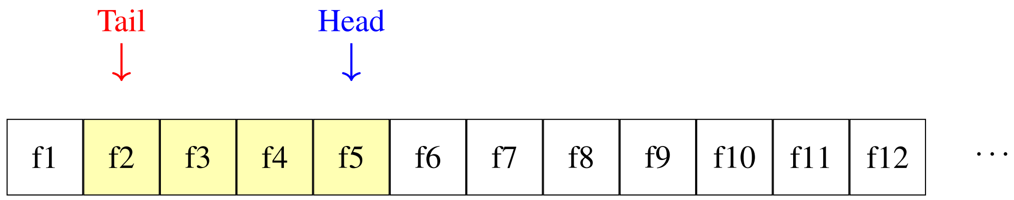


Fig. 7.4. Ring Buffer of Size 4 after inserting another frame

In Figure 7.4, a new frame, f5, is added to the buffer. This causes the oldest frame, f1, to be overwritten.

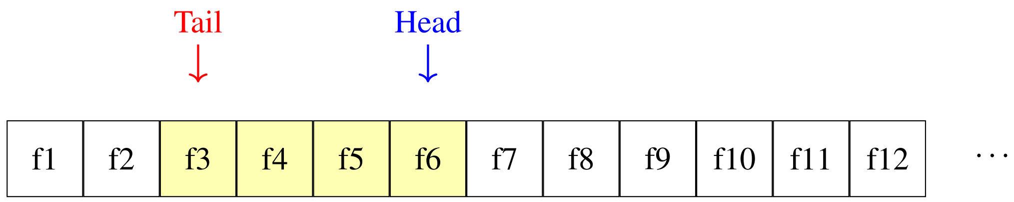


Fig. 7.5. Ring Buffer of Size 4 after inserting another frame

In Figure 7.5, another frame, f6, is added to the buffer. This time, the oldest frame, now f2, is overwritten. Note how the buffer wraps around, creating a circular effect. Elements are added to the buffer at the head and removed from the buffer at the tail. When the buffer is full, the head wraps around to the beginning of the buffer, overwriting the oldest element. A circular visualization is available in Section 10.3, which provides a more detailed explanation of the ring buffer operation.

With the ring buffer in place, we can now process the frames together to detect the preamble. We'll slide the buffer along the frames, applying the matched filter to the buffer each time.

7.5. Decoding the Message

Once we have detected the preamble, a similar problem arises when demodulating the message data. The message data may be split between frames, and we need to process the frames together to decode the message accurately. There are various ways of solving this issue, however, for this project we'll use a simple approach: since we send characters one by one, and we know the length of each character, we can add the contents of the buffer plus some extra frames to ensure we have the full character. Given that one character is 8 bits long,

$$\text{samples\_per\_char} = 8 \times \text{samples\_per\_symbol} = 8 \times 441 = 3,528 \tag{7.4}$$

We can then calculate the buffer size required to decode the message data, again, assuming a frame size of 1024 samples:

$$\text{buffer\_size} = \text{samples\_per\_char}/\text{frame\_size} = 3,528/1024 \approx 3.4 \tag{7.5}$$

Hence we need to add 4 frames to the contents of the buffer to ensure we have the full character.

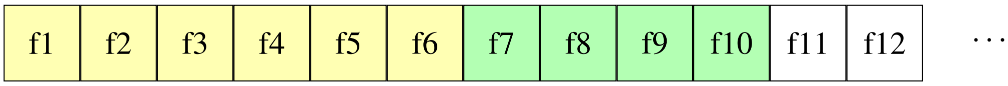


Fig. 7.6. Buffer with 6 frames and 4 extra frames

In Figure 7.6, we have the ring buffer with 6 frames, containing the preamble and part of the message data. Assuming that the preamble is present in the buffer, we can now process the frames together to decode the message data. To ensure we have the full character, we need to add 4 extra frames to the buffer.

The 6 + 4 frames are then sent to the demodulator, which decodes the message data. As each symbol is 0.1 seconds long, and each character is 8 symbols long, we can decode the message data character by character. Once the 8 symbols are decoded, the rest of the buffer is discarded, and the process continues with the next character.

In this project, we calculate the ring buffer size based on the length of the preamble. After calculating the minimum buffer size required to detect the preamble, we make it 4 times larger as it has been observed that this size works well in practice.

8. LINUX

This chapter provides an overview of the Linux operating system, focusing on the Linux kernel. We'll briefly explore the history of UNIX and Linux, the structure of the Linux kernel, and the role of the kernel in managing system resources and providing services to user-space applications.

8.1. Operating Systems

An operating system (OS) is a software that manages computer hardware and provides services for computer programs. There are many precursors to the modern operating system, however the first "modern" operating system, and the one that most modern operating systems are based on, is UNIX. UNIX started off as a project at AT&T Bell Labs in the

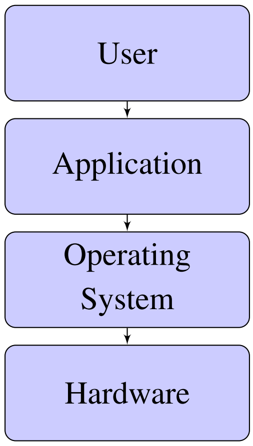


Fig. 8.1. Layers of interaction from User to Hardware

late 1960s, and was designed to be a multi-user, multitasking system. It was developed by Dennis Ritchie, Ken Thompson, and others at Bell Labs, and was written in the C programming language, which was also developed by Ritchie.

The late 1960s at Bell Laboratories were marked by the gradual withdrawal from the Multics project, a collaborative effort to build a time-sharing operating system. Time-sharing systems divide the processing time of a computer among multiple users, giving each user the illusion of having their own computer thanks to the rapid switching between tasks. As seen in Figure 8.2, a timesharing operating system allows multiple users to interact with the system simultaneously, each using their own terminal.

The operating system schedules tasks and allocates resources to ensure that each user receives a fair share of the system's processing power. From the OS's perspective, shown in Figure 8.3, it rapidly switches between users, executing tasks in small time slices. This is in contrast to the batch processing systems that were prevalent at the time, where users

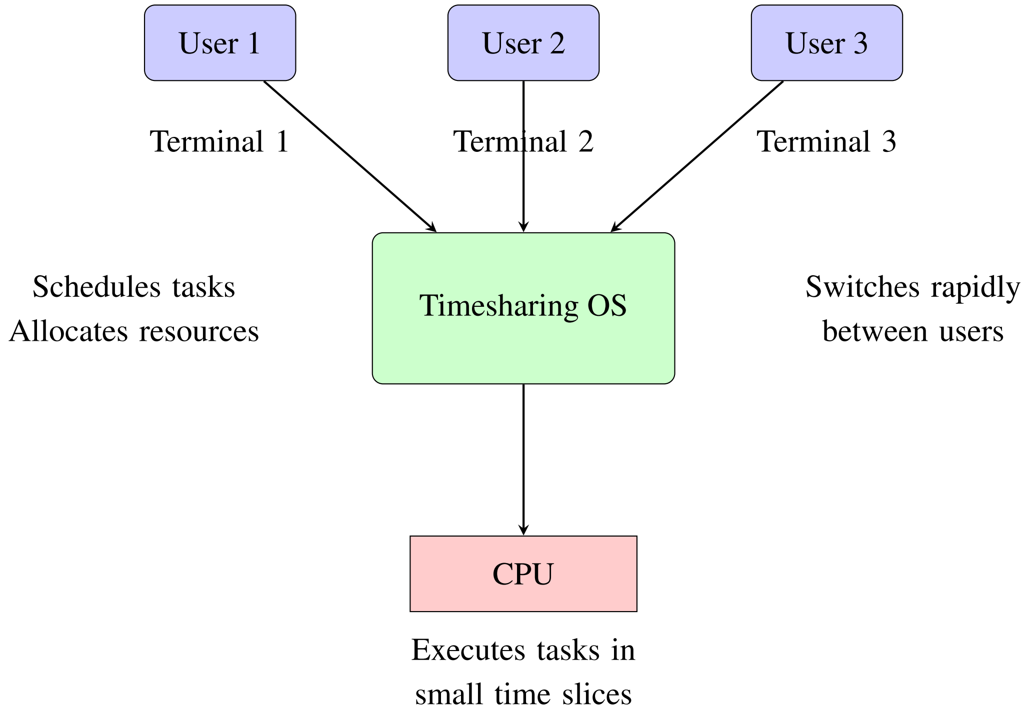


Fig. 8.2. Illustration of a Timesharing Operating System

submitted their jobs to the system, which would then process them in sequence without user interaction.

The perceived failure of Multics to deliver a usable system left a void for researchers like Ken Thompson, Dennis Ritchie, M. D. McIlroy, and J. F. Ossanna [7]. After unsuccessful attempts to secure funding for a new system, Thompson began experimenting on a little-used PDP-7 computer. He implemented a file system design he had conceived earlier with Canaday and Ritchie [7]. This rudimentary system, lacking features like pathnames and multi-programming, nevertheless laid the groundwork for what would become UNIX.

By 1970, the core components of UNIX, including the file system, process control mechanisms, and a command interpreter (shell), were in place. It was around this time that Brian Kernighan coined the name "UNIX," a playful nod to the complexities of Multics [7]. In 1970 came the PDP-11, which marked a significant step forward. The UNIX system was ported to this new architecture, bringing with it improvements such as the hierarchical file system with pathnames, a more sophisticated process control model, and features like character erase and line-kill processing for terminals [7].

8.1.1. MINIX

As the UNIX operating system grew in popularity, various organizations and institutions developed their own versions of UNIX, known as UNIX variants. These variants were often tailored to specific hardware platforms or user requirements, leading to a diverse

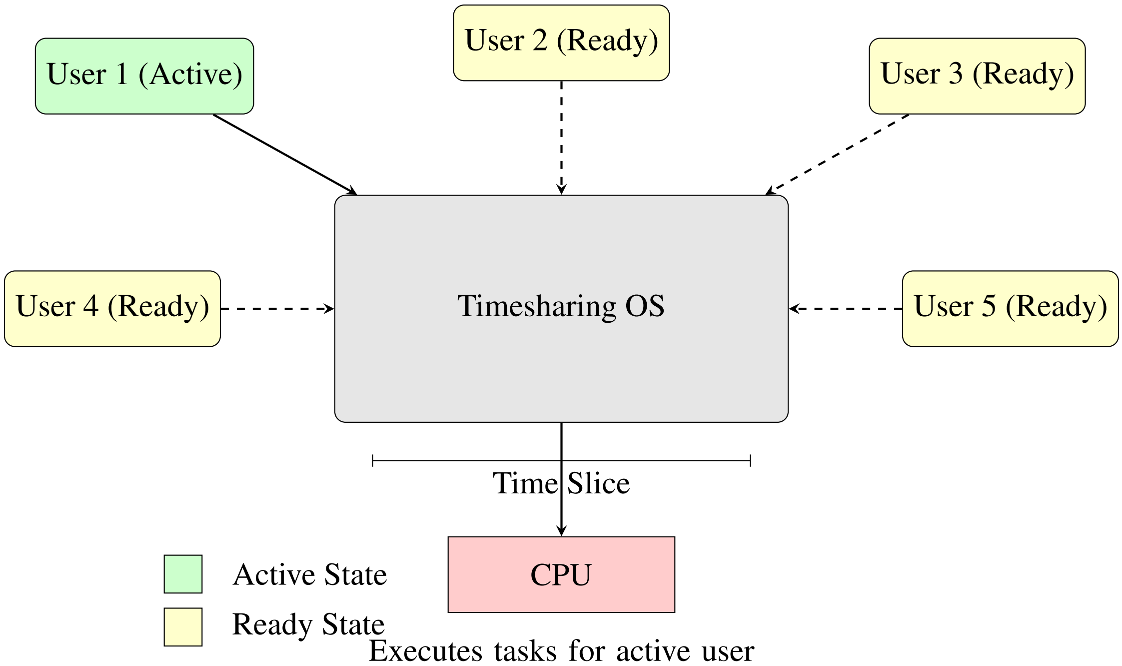


Fig. 8.3. Timesharing OS with Multiple Users in Active and Ready States

Fig. 8.4. PDP-7 at the Oslo University Museum. [8]

ecosystem of UNIX-based operating systems. One of these variants, known as MINIX, was developed by Andrew S. Tanenbaum as a teaching tool for operating system design. [9] MINIX was designed to be simple, reliable, and easy to understand, making it an ideal platform for learning about operating systems. The limitations in the licensing of MINIX, being exclusively available for educational use, led Linus Torvalds to create the Linux kernel, which along with the GNU free software tools, formed the basis of the modern Linux operating system.

8.1.2. What is a Kernel?

The kernel is the core component of an operating system that manages system resources and provides essential services to user programs. It is NOT the same as the operating system, but rather the central part of it. The kernel on MacOS is XNU, on Windows it is the Windows NT kernel, and on Linux it is the Linux kernel. Similar to the engine of a car, the kernel is responsible for managing the system's resources and ensuring that user programs can run efficiently. And just like the engine of a car, which needs the rest of the car to function properly, the kernel relies on other components of the operating system to provide a complete computing experience. In the case of Linux, the kernel interacts with the GNU utilities to form a complete operating system.

8.2. Kernel-space vs User-space

The Linux kernel is divided into two main spaces: kernel-space and user-space. Kernel-space is the privileged mode of the processor where the kernel runs, while user-space is the unprivileged mode where user programs run. In a modern operating system with virtual memory support, the kernel and user programs are isolated from each other, preventing user programs from directly accessing kernel memory or resources. The technical term for this isolation is "protection rings," with the kernel running in ring 0 (the most privileged level) and user programs running in ring 3 (the least privileged level), as shown in Figure 8.5. Rings 1 and 2 are not used in most modern operating systems.

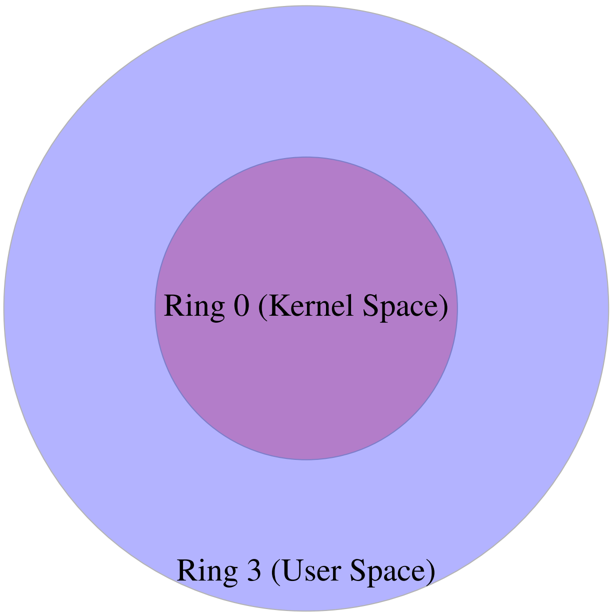


Fig. 8.5. Protection Rings in Linux

It might be helpful to think of the kernel as the manager of a company, and user programs as the employees. The manager has access to all areas of the company, including sensitive information and resources, while the employees are restricted to their designated workspaces. The manager can allocate resources, schedule tasks, and enforce rules, while employees can only perform their assigned duties. If an employee needs access to a restricted area or resource, they must request permission from the manager. Were an employee to bypass the manager and access restricted areas directly, it could lead to chaos and security risks within the company. Similarly, in an operating system, user programs must interact with the kernel to access system resources and perform privileged operations. User programs interact with the kernel through system calls, which act as a bridge between user-space and kernel-space. By using system calls, user programs can request services from the kernel, such as reading from a file, allocating memory, or creating a new process. This ensures that user programs can perform tasks that require privileged access without compromising system security.

8.2.1. Types of Kernels

There are several types of kernels, each with its own design and characteristics. The main types of kernels are monolithic kernels, microkernels, and hybrid kernels. A brief overview of each type is provided below:

- **Monolithic Kernel:** A monolithic kernel is a single large program that contains all the essential components of the operating system, such as device drivers, file systems, and system calls. Linux is an example of a monolithic kernel.
- **Microkernel:** A microkernel is a small, modular kernel that only contains essential functions, such as process management and inter-process communication. Additional services, such as device drivers and file systems, are implemented as user-space processes. MINIX is an example of a microkernel.
- **Hybrid Kernel:** A hybrid kernel combines elements of both monolithic and microkernel designs. It includes essential operating system functions in the kernel space, while other services are implemented as user-space processes. Windows NT is an example of a hybrid kernel.

Building on the analogy of the company, the type of kernel can be likened to the organizational structure of the company. A monolithic kernel is like a traditional hierarchical organization, where all functions are centralized under one management. A microkernel is akin to a decentralized organization, with separate departments handling different functions. A hybrid kernel is a mix of both, combining centralized and decentralized elements to optimize performance and flexibility.

8.3. Linux Kernel

Let's turn our attention to the Linux kernel, which is a monolithic kernel that was initially developed by Linus Torvalds in 1991. It's the core component of the Linux operating system and is responsible for managing system resources, such as memory, processes, and devices. It is mostly written in the C programming language, with some parts in assembly language. However, since version 6, the Linux kernel has started to incorporate code written in Rust [24], a systems programming language known for its memory safety features.

The Linux kernel is highly modular, allowing developers to add or remove features as needed. This modularity is achieved through the use of kernel modules, which will be explored in the next chapter.

8.3.1. Everything is a File

One of the key design principles of the Linux kernel is that "everything is a file." This principle reflects the idea that all system resources, such as devices and processes can be accessed and manipulated using file-like abstractions. This means that many system resources are represented as files in the file system, allowing users and programs to interact with them using familiar file operations, such as reading, writing, and seeking. For example, devices connected to the system, such as hard drives, keyboards, and network interfaces, are represented as files in the `/dev` directory. And processes running on the system are represented as files in the `/proc` directory, allowing users to view and manipulate process information. Another example is the `/sys` directory, which provides a file-based interface to kernel data structures and system information.

Let's look at how to retrieve information about the CPU using the /proc filesystem. The file /proc/cpuinfo contains details about the processor, such as its model name, speed, and cache size.

To read the contents of the file, we can use the `cat` command in the terminal:

```
cat /proc/cpuinfo
```

The cat command displays the contents of a file on the terminal. Here is an example output of the /proc/cpuinfo file:

```
processor : 0
BogoMIPS : 48.00
Features : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics
    fphp asimdhp cpuid asimdrdm jscvt fcma lrcpc dcpop sha3
    asimddp sha512 asimdfhm dit uscat ilrcpc flagm ssbs sb paca
    pacg dcpodp flagm2 frint
CPU implementer : 0x61
CPU architecture: 8
CPU variant : 0x0
CPU part : 0x000
CPU revision : 0

processor : 1
BogoMIPS : 48.00
Features : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics
    fphp asimdhp cpuid asimdrdm jscvt fcma lrcpc dcpop sha3
    asimddp sha512 asimdfhm dit uscat ilrcpc flagm ssbs sb paca
```

```
        pacg dcpodp flagm2 frint
    CPU implementer : 0x61
    CPU architecture: 8
    CPU variant : 0x0
    CPU part : 0x000
    CPU revision : 0
```

This command was executed on a virtual machine running on an arm64 architecture, with two processors.

9. KERNEL LEVEL KEYLOGGER

This chapter will detail the implementation of the kernel-level keylogger, discussing the techniques used to intercept keystrokes, such as hooking into the keyboard notifier chain.

9.1. Kernel Modules

As we saw in Chapter 8, the Linux kernel is the core component of the operating system that manages hardware resources and provides essential services to user programs. One of the key features of the Linux kernel is its modular design, which allows developers to extend the kernel's functionality by adding new code in the form of kernel modules. This chapter delves into the concept of kernel modules, their importance, and the mechanics of their implementation.

Kernel modules are comparable to plug-ins for the kernel; they are standalone pieces of code that can be dynamically loaded and unloaded at runtime. This adaptability offers significant flexibility, enabling developers to introduce new features, hardware support, or device drivers as needed, without the overhead of building a new kernel image from scratch. These modules are principally written in C, compiled independently from the kernel, and integrated into the kernel's address space upon demand. There are two main functions that a kernel module must implement: `module_init` and `module_exit`.

The `init` function is called when the module is loaded into the kernel. This function is responsible for initializing the module and setting up any resources it needs.

On the other hand, the `exit` function is called when the module is unloaded from the kernel. This function is responsible for cleaning up any resources allocated by the module and releasing them back to the system.

Together, these functions provide a structured and orderly mechanism for loading and unloading kernel modules. Appendix A provides a step-by-step guide on how to write a simple kernel module for Linux.

9.2. Interrupts

Whenever any device connected to the computer generates an event, it sends an interrupt signal to the CPU. One such example of this is when a key is pressed on the keyboard, or when the mouse is moved. Notice how, no matter what the CPU is doing, it must stop and handle the interrupt. Otherwise, the event would be lost, and the key press would not be registered, or the mouse would not move.

Interrupts are a fundamental concept in computer architecture, allowing devices to communicate with the CPU and request its attention. When an interrupt occurs, the CPU stops the currently executing program and transfers control to a special piece of code called an interrupt handler. The interrupt handler processes the interrupt and performs the necessary actions to handle the event. After the interrupt handler has finished executing, the CPU returns to the interrupted program to continue its execution.

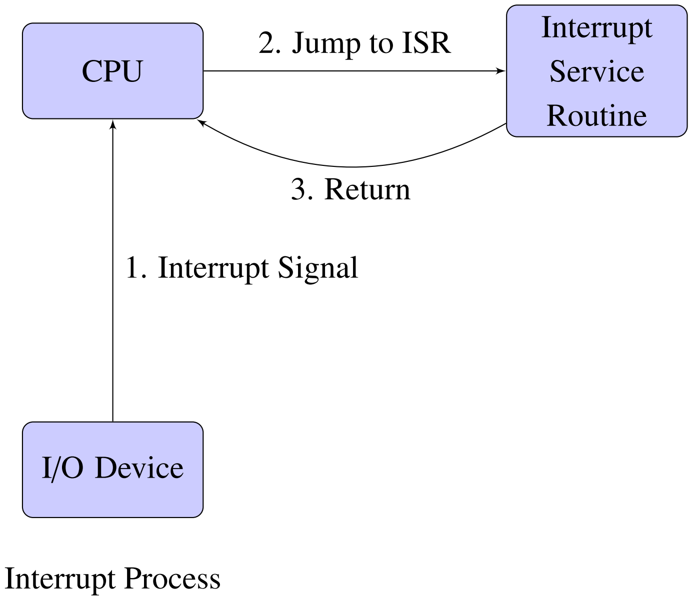


Fig. 9.1. Illustration of an Interrupt Process

9.3. Keyboard Notifier

After a key is pressed on the keyboard, the keyboard driver sends an interrupt signal to the CPU to indicate that a key event has occurred. When this interrupt is processed, a special piece of code called the keyboard notifier is invoked. The keyboard notifier is a mechanism in the Linux kernel that allows modules to subscribe to keyboard events and receive notification whenever a key is pressed or released. Many different modules can register with this keyboard notifier, creating a chain of notifiers that are called in sequence when a key event occurs. For example, the shell might register a notifier to display the key event on the screen, while another graphical application might register a notifier to update the user interface. As you can see in Figure 9.2, the notifiers are organized in a linked list, with each notifier containing a function pointer to its callback function and a priority value. For the keylogger, we will register a keyboard notifier to intercept key events and log them. To do this, we will create a notifier block with a callback function that stores the key event information. Our callback function will be called whenever a key is either pressed or released, and it will be passed the key event data as an argument, which we can then process and log.

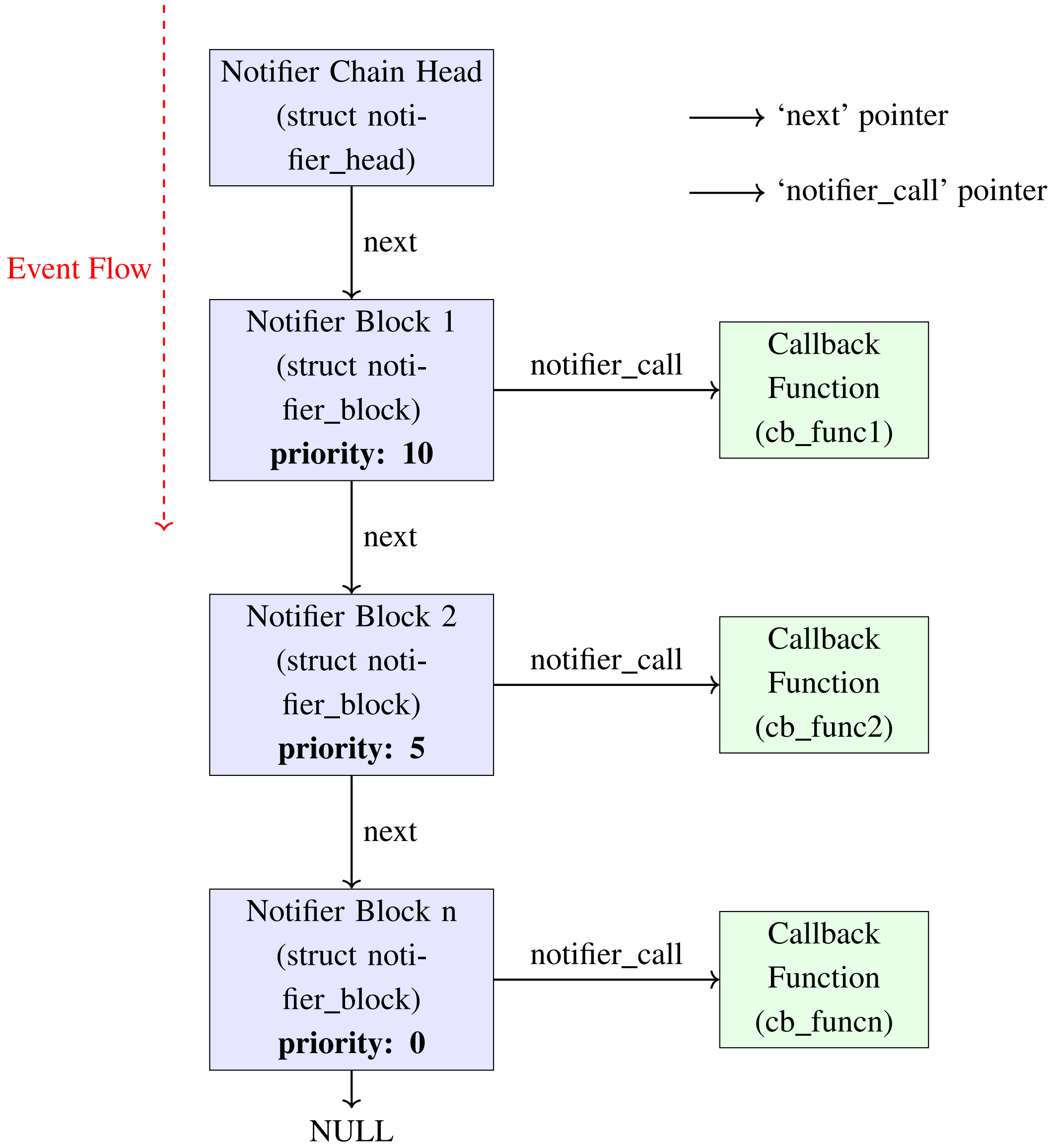


Fig. 9.2. Illustration of a Notifier Chain with Callback Functions.

9.3.1. keyboard_notifier_param

The `keyboard_notifier_param` structure contains information about the key event, such as the key code, the state of the key (pressed or released), and the time of the event. It is defined in the `include/linux/keyboard.h` header file and has the following structure:

```c
struct keyboard_notifier_param {
    struct vc_data *vc; /* VC on which the keyboard press was done */
    int down;  /* Pressure of the key? */
    int shift;  /* Current shift mask */
    int ledstate;  /* Current led state */
    unsigned int value; /* keycode, unicode value or keysym */
};
```

Fig. 9.3. Definition of the `keyboard_notifier_param` structure. [10]

The Linux Kernel Documentation provides a detailed explanation of the fields:

> 'vc' always provide the VC for which the keyboard event applies;
> 'down' is 1 for a key press event, 0 for a key release;
> 'shift' is the current modifier state, mask bit indexes are KG_*;
> 'value' depends on the type of event.
> KBD_KEYCODE events are always sent before other events, value is the keycode.
> KBD_UNBOUND_KEYCODE events are sent if the keycode is not bound to a keysym. value is the keycode.
> KBD_UNICODE events are sent if the keycode → keysym translation produced a unicode character. value is the unicode value.
> KBD_KEYSYM events are sent if the keycode → keysym translation produced a non-unicode character. value is the keysym.
> KBD_POST_KEYSYM events are sent after the treatment of non-unicode keysyms. That permits one to inspect the resulting LEDs for instance.

Fig. 9.4. Description of the `keyboard_notifier_param` structure fields [11].

There are various types of events, but for our keylogger, we are primarily interested in the `KBD_KEYCODE` event, which provides the keycode of the pressed key. Knowing this, the parameters `down`, `shift` and `value` are the most relevant for our purposes. The `down` parameter indicates whether the key was pressed or released, with a value of 1 for a key press and 0 for a key release. The `shift` parameter represents the current state of the shift keys (e.g., Shift, Ctrl, Alt), encoded as a bit mask. This means that each bit in the mask corresponds to a specific shift key.

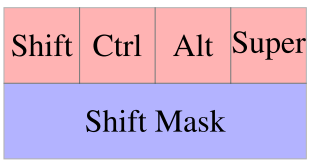


Fig. 9.5. Shift Mask Representation

Lastly, the `value` parameter contains the keycode of the pressed key, which we will use to identify the key being pressed.

9.3.2. Keycodes

Keycodes are numerical values that represent the keys on a keyboard. Each key on the keyboard is assigned a unique keycode, which is used to identify the key when it is pressed or released. They are defined in the `include/uapi/linux/input-event-codes.h` header file and are typically represented as integer values.

```
        #define KEY_RESERVED  0
        #define KEY_ESC   1
        #define KEY_1   2
        #define KEY_2   3
        #define KEY_3   4
        #define KEY_4   5
        #define KEY_5   6
        #define KEY_6   7
        #define KEY_7   8
        #define KEY_8   9
        #define KEY_9   10
        #define KEY_0   11
        #define KEY_MINUS  12
        #define KEY_EQUAL  13
        #define KEY_BACKSPACE  14
        #define KEY_TAB   15
        #define KEY_Q   16
        #define KEY_W   17
        #define KEY_E   18
        #define KEY_R   19
        #define KEY_T   20
        #define KEY_Y   21
```

Fig. 9.6. Code Snippet from `input-event-codes.h` showing Keycode Definitions [10].

While the keycodes are defined in the header file, the actual values may vary depending on the keyboard layout and configuration. For example, the keycode 21, which corresponds to the 'Y' key in a QWERTY layout, may represent a different key in a different layout. Therefore, it is essential to consider the keyboard layout when interpreting keycodes. For our keylogger, we will use the american QWERTY layout to interpret the keycodes [4]. Furthermore, we will only support a subset of keycodes to keep the implementation simple and focused. These keycodes will include the alphanumeric keys, punctuation keys, and the Shift key.

In order to take the shift key into account, we will need to consider the shift mask, which is a bit mask that represents the state of the shift keys (e.g., Shift, Ctrl, Alt). A keymap will map the keycodes to the corresponding characters based on the shift mask. For example, if the Shift key is pressed, the keymap will map the keycode for the '1' key

[4]Other keyboard layouts could be supported by using the appropriate keymap. However, for simplicity and clarity, this project will focus on the American QWERTY layout, which is widely used.

to the '!' character.

The keymap used in this project was obtained from spy.c, a simple linux keylogger example [1].

```
static const char *us_keymap[][2] = {
    {"\0", "\0"}, {"_ESC_", "_ESC_"}, {"1", "!"}, {"2", "@"},
            // 0-3
    {"3", "#"}, {"4", "$"}, {"5", "%"}, {"6", "^"},
                    // 4-7
    {"7", "&"}, {"8", "*"}, {"9", "("}, {"0", ")"},
                    // 8-11
    {"-", "_"}, {"=", "+"}, ...
       // 12-13
};
```

Fig. 9.7. Snippet of the US Keymap used in the Keylogger. [1]

10. COMMUNICATION BETWEEN KERNEL SPACE AND USER SPACE

As we have seen in the previous chapter, the keylogger intercepts key events in the kernel space. However, we need a way to communicate this information to the user space, where it can be processed and transmitted. Section 8.2 discussed the distinction between kernel space and user space, highlighting the need for a mechanism to transfer data between the two. This chapter explores various methods for communication between kernel space and user space, focusing on character devices.

10.1. Communication Methods

There are several methods for communication between kernel space and user space in Linux. Some of the most common methods include:

- **System Calls:** They provide a well-defined interface for user programs to request services from the kernel. They are the primary mechanism for user programs to interact with the kernel and access system resources.
- **Procfs:** The procfs filesystem provides a virtual filesystem that allows user programs to access information about running processes and system resources.
- **Sysfs:** Similar to procfs, it's a virtual filesystem that provides access to kernel data and system information.
- **Debugfs:** The debugfs filesystem is a special filesystem that allows user programs to access debugging information and kernel data. There are no rules or restrictions on what can be placed in debugfs, making it a flexible tool for debugging and development [11].
- **Character Devices:** Character devices provide a stream-oriented interface for communication between kernel space and user space. They are often used for devices that transfer data sequentially, such as serial ports and terminals.

For our keylogger, we will use character devices to communicate between kernel space and user space, as they provide a simple mechanism for transferring data between the two. We will create a character device in the kernel space that will store the intercepted keystroke data. User programs can then read this data from the character device and process it as needed.

10.2. Character Devices

Character devices are a type of device file in Linux that provides a stream oriented interface for communicating between the kernel space and user space. They are often used for devices that transfer data sequentially, such as serial ports and terminals. In our case, we will use a character device to transfer the keystroke data from the kernel space to the user space.

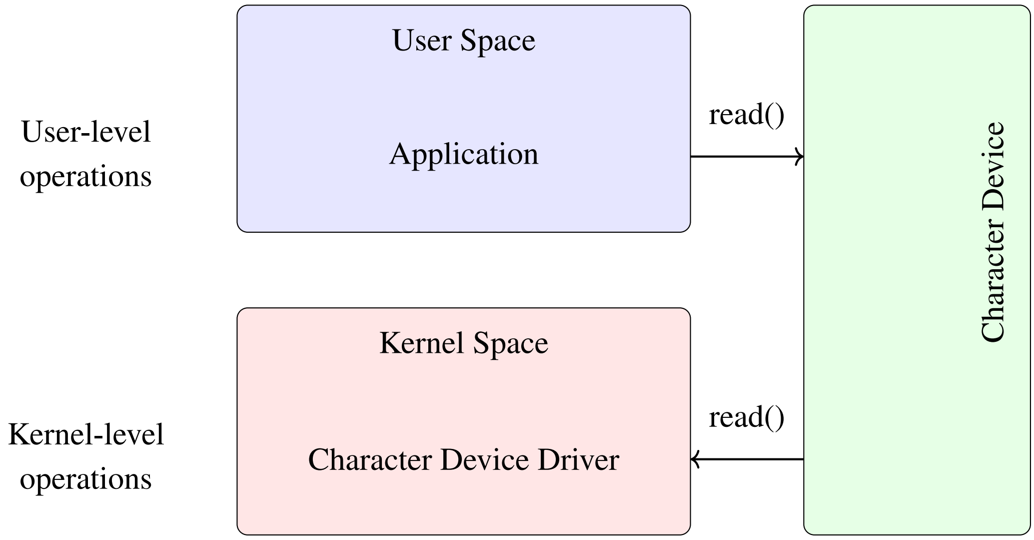


Fig. 10.1. Communication between User Space and Kernel Space via Character Device

The character device acts as a bridge between the kernel space and user space, allowing user programs to read and write data to the device. It has two parts: the character device driver in the kernel space and the character device file in the user space. The character device driver implements the operations that can be performed on the device, such as reading and writing data. On the other hand, the character device file in the user space provides an interface for user programs to interact with the device. This is analogous to a mouse or keyboard, where the device driver in the kernel space processes the input events from the device itself.

10.2.1. Registering a Character Device

To create a character device, we need to register it with the kernel using the register_chrdev function [11]. This function takes three arguments: the major number of the device, the name of the device, and a pointer to a `file_operations` structure that defines the operations that can be performed on the device.

Definition 2.1: Major and Minor Numbers

In Linux, each device is identified by a major number and a minor number. The major number identifies the type of device, while the minor number identifies a specific instance of that device.

For instance, a hard disk might have a major number of 3 and a minor number of 1, indicating the first partition on the third hard disk. The major number is used to determine which device driver should handle the device, while the minor number is used to identify a specific device instance.

The `file_operations` structure contains function pointers to the various operations that can be performed on the device, such as reading, writing, and opening. For our keylogger, we will only need to implement the `read` operation, which will be used to read the keystroke data from the kernel space.

10.2.2. Implementing the Read Operation

The `read` operation is implemented as a function that takes four arguments:

- **file:** A pointer to a `file` structure that represents the open file.
- **buffer:** A pointer to a user-space buffer where the data will be copied.
- **length:** The number of bytes to read.
- **offset:** The offset within the file to start reading from.

The `read` operation should copy the keystroke data from the kernel space to the user-space buffer and return the number of bytes copied[5]. If there is no data available, the `read` operation should return 0.

10.2.3. simple_read_from_buffer

The `simple_read_from_buffer` function is a helper function provided by the Linux kernel that simplifies the process of copying data from a buffer to a user-space buffer. It takes three arguments: the user-space buffer, the size of the buffer, and the kernel-space buffer. The function copies data from the kernel-space buffer to the user-space buffer, up to the size of the buffer, and returns the number of bytes copied. If the user-space buffer is too small to hold all the data, the function will copy as much data as possible and return the number of bytes copied.

The `simple_read_from_buffer` function is a convenient way to copy data from the kernel space to the user space without having to manually handle buffer sizes and offsets. However, the function is limited to copying data one place to another and does

[5] The `read` operation is one of several file operations defined in the `file_operations` structure. For our character device, we will only be implementing the `read` operation, as we do not need to write data to the device from userspace.

not provide any additional functionality. For our keylogger, we want each read operation to return the next keystroke in the buffer, which requires more complex logic than what `simple_read_from_buffer` provides. In technical terms, we want our read operation to NOT be idempotent.

Definition 2.2: Idempotent Operations

An idempotent operation is an operation that has the same effect when applied multiple times as it does when applied once. In the context of a read operation, an idempotent operation would return the same data each time it is called. However, in our keylogger, we want each read operation to return the next keystroke in the buffer, which requires the operation to be non-idempotent.

10.3. Ring Buffer

In order to implement this behavior, we will reuse the ring buffer data structure from Section 7.4 to store the intercepted keystroke data. Looking at figures 10.2 and 10.3, we can see how the ring buffer works. The ring buffer has a fixed size and two pointers: the head and the tail. When a new keystroke is intercepted, it is added to the buffer at the head position, and the head pointer is incremented. When the read operation is called, the data at the tail position is returned, and the tail pointer is incremented. It is important to note that in a ring buffer there is no concept of an end or beginning; the buffer is circular, and the head and tail pointers wrap around when they reach the end of the buffer.

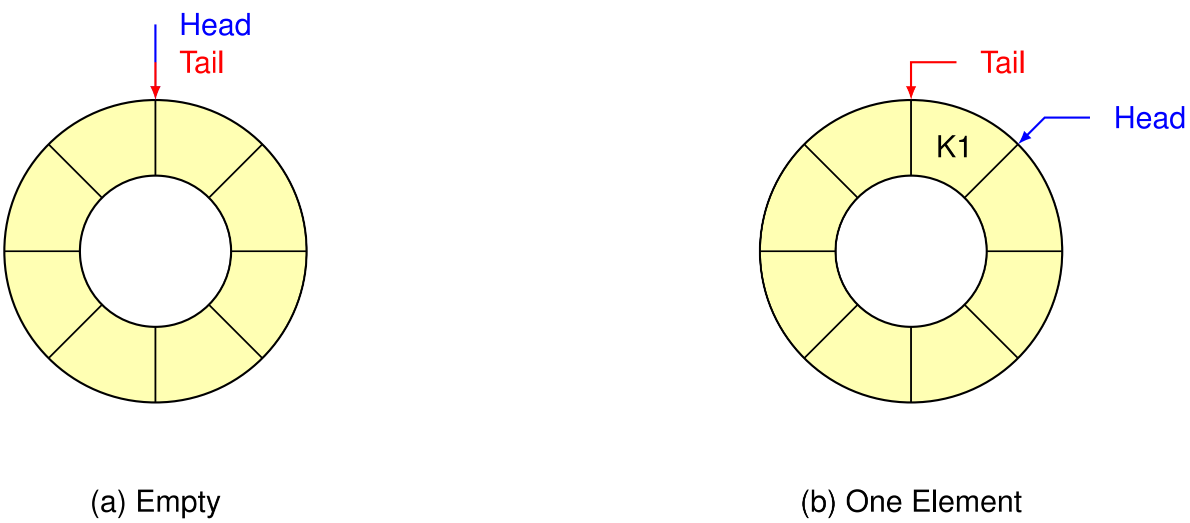
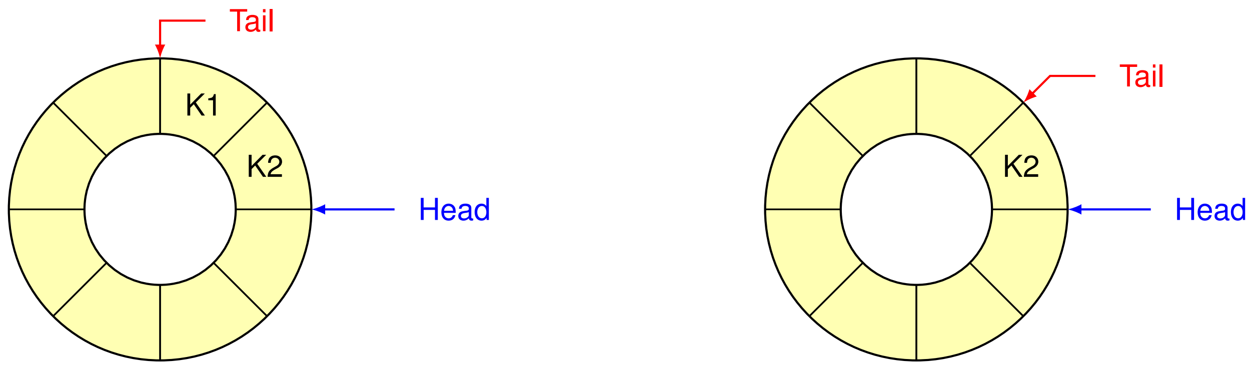


Fig. 10.2. Ring Buffer Examples



(a) Two Elements

(b) After Removing Element

Fig. 10.3. Circular Buffer: Adding and Removing Elements

With the ring buffer, we can store the keystroke data in a circular buffer and read the data sequentially, returning the next keystroke each time the read operation is called. The implementation of the read operation will involve reading the data at the tail position, incrementing the tail pointer, and returning the data to the user space.

10.3.1. User-Space Access

Once the character device is registered with the kernel, it can be accessed from user space like any other file. User programs can open the device file, read the keystroke data, and close the device file.

11. USERSPACE MODULATOR

The last component of our data transmission system is the userspace modulator. It's responsible for encoding the data into sound waves that can be transmitted over the air. We have chosen to implement the modulator in userspace to simplify the development process and avoid the complexities of kernel programming. One of the downsides of kernel programming is that a bug in the kernel module can crash the entire system, which can be inconvenient for development. Another downside is the absence of many standard libraries in the kernel, which can make development more challenging. Making the modulator a userspace application allows us to use standard libraries and tools, which will be explained in this chapter.

11.1. Reading the Keystrokes

The first step in the modulator is to read the keystrokes from the character device. To do this, we will open the character device file, read the keystroke data, and process it. Since the device was implemented as a ring buffer, we can read the keystrokes sequentially, one by one, until there are no more keystrokes available. The following function shows how to read the keystrokes from the character device file in userspace:

```
void read_keys_from_device(const char *device_path, std::
    function<void(char)> callback) {
    int fd = open(device_path, O_RDONLY | O_NONBLOCK);
    if (fd < 0) {
        perror("Failed to open the device");
        return;
    }

    char buffer[1024];
    while (true) {
        ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
        if (bytes_read > 0) {
            for (ssize_t i = 0; i < bytes_read; i++) {
                callback(buffer[i]);
            }
        } else if (bytes_read == 0 || (bytes_read == -1 &&
            errno == EAGAIN)) {
            // No data available, sleep for a short time
            std::this_thread::sleep_for(std::chrono::
                milliseconds(10));
        } else {
            perror("Error reading from device");
            break;
        }
```

```
        }

        close(fd);
    }
```

The function opens the character device file in read-only mode and reads the keystroke data from the device file. It then processes the keystrokes using a callback function, which we'll later see, and sleeps for a short time if no data is available.

The data read from the character device contains characters encoded in ASCII, which will need to be encoded into sound waves for transmission.

11.2. Audio in Linux

Audio in Linux is a complex topic, with multiple audio systems available, each with its own set of features and capabilities. For our userspace modulator, we will use a simple and lightweight audio library called miniaudio.h, which provides an easy-to-use API for audio playback and recording, and abstracts the underlying audio systems [12]. However, it's important to at least mention some of the audio systems available in Linux, as they play a crucial role in audio processing.

11.2.1. ALSA

The Advanced Linux Sound Architecture (ALSA) is a software framework and part of the Linux kernel that provides an API for sound card drivers. It's the default sound system for most Linux distributions and provides low-level access to audio devices. ALSA handles audio input and output, mixing, and routing, and supports many different devices and configurations, as documented in the project website. Since it's part of the kernel, it has direct access to the hardware and can provide low-latency audio processing. However, one of the main limitations of ALSA is that only one application can access the audio device at a time, which can be a problem for applications that require multiple audio streams. For example, if you're playing music in a media player and want to play a game that also produces sound, only one of the applications will be able to access the audio device.

This limitation has led to the development of many sound servers that sit on top of ALSA and provide a higher-level API for audio processing. One may wonder why there are multiple sound servers available in Linux, when one would suffice. Each of the sound servers has its own set of features and capabilities, and is optimized for different use cases. However, this has led to fragmentation in the Linux audio ecosystem, with different applications using different sound servers, which can make it challenging to configure and manage audio devices. [6].

[6] Relevant xkcd: https://xkcd.com/927/

11.2.2. PulseAudio

Pulseaudio is one of the most popular sound servers for linux, and has been the default for many distributions for many years. It solves the problem of multiple applications accessing the audio device by acting as a sound server that multiplexes audio streams from different applications. However, this introduces some latency, as the audio data has to be passed through the sound server before reaching the audio device, which can be a problem for applications that require low-latency audio processing, such as professional audio software.

11.2.3. JACK

This is where the JACK Audio Connection Kit comes in. It is an alternative to PulseAudio that provides low-latency audio processing and is designed for professional audio applications. Just like PulseAudio, JACK acts as a sound server that multiplexes audio streams from different applications. However, JACK is optimized for low-latency audio processing and provides tools for audio routing and mixing, making it a popular choice for audio professionals. It is commonly used in music production and audio engineering, where low-latency audio processing is essential. The downside of JACK is that since it's optimized for low-latency audio processing, doesn't provide the same level of support for consumer uses as PulseAudio does, such as global per application volume control. **jack-website** This makes it less suitable for general-purpose audio processing, such as playing music or watching videos. There are ways to use both PulseAudio and JACK together, but it requires additional configuration and can be quite cumbersome and error-prone.

11.2.4. Pipewire

Pipewire is a relatively new sound server that aims to combine the best features of PulseAudio and JACK. The newest versions of Fedora, Arch, and Ubuntu have already adopted Pipewire as the default sound server, and it's expected to become more popular in the future, as it seems to solve the fragmentation caused by PulseAudio and JACK. Figure 11.1 shows a diagram of the Linux audio stack, and how the different audio systems interact with each other.

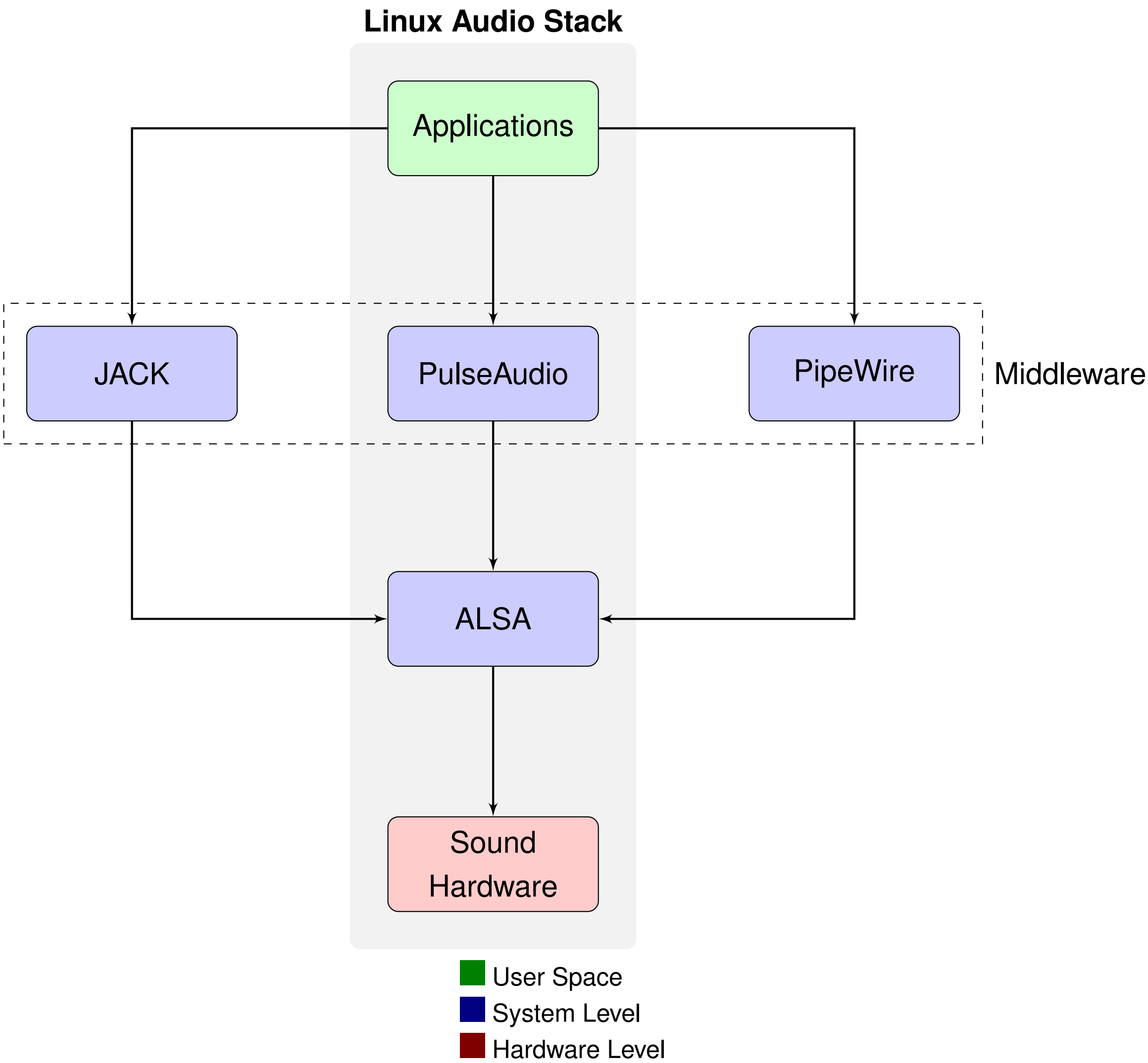


Fig. 11.1. Diagram of the Linux Audio Stack

11.3. Miniaudio.h

Having discussed the audio systems available in Linux, we can now introduce miniaudio.h, a simple and lightweight audio library that provides an easy-to-use API for audio playback and recording. It's designed to work on multiple platforms, including Linux, Windows, and macOS, and abstracts the underlying audio systems, making it easy to use and portable. This means that no matter the underlying audio systems, be it Pulseaudio, JACK, or even ALSA, miniaudio.h will work seamlessly across different platforms.

Definition 3.1: Library

A library is a collection of precompiled routines that a program can use. Libraries are particularly useful for storing frequently used routines, as they can be called upon by multiple programs without the need to recompile the code.

In simple terms, a library is a collection of functions that can be used by a program to perform specific tasks. These functions can be used by multiple programs, which helps to reduce code duplication and improve code reusability. For example, the miniaudio.h library provides functions for audio playback and recording, which can be used by our userspace modulator to generate sound waves. This way, we don't have to write the audio playback code from scratch as the library provides the necessary functions to do so. Since it is a single-file library, all the necessary code is contained in a single header file, which can be included in our project without the need for additional dependencies.

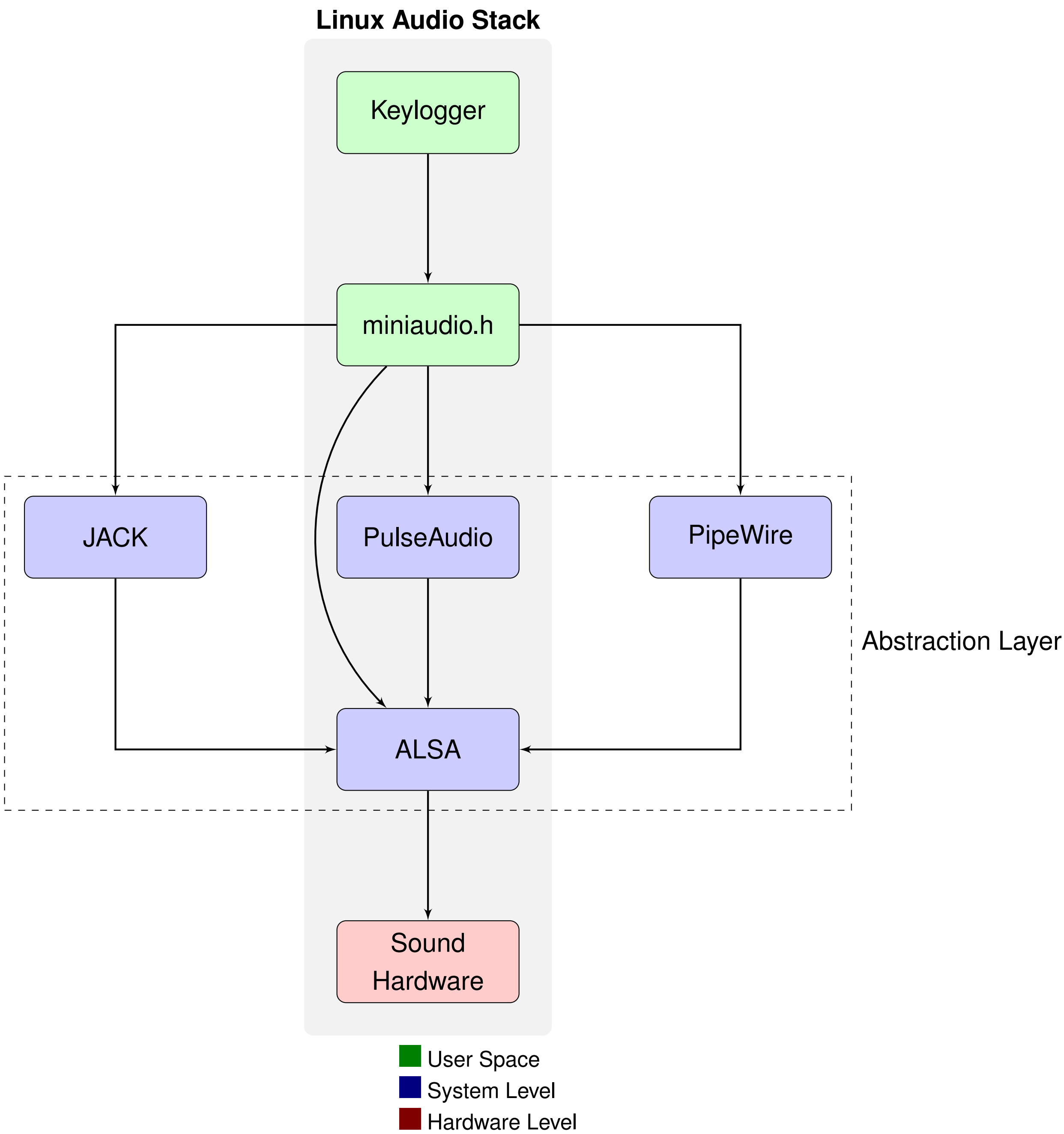


Fig. 11.2. Diagram of the Linux Audio Stack, showing how our keylogger uses miniaudio.h to abstract the underlying audio systems

11.3.1. Using miniaudio.h

To use miniaudio.h in our userspace modulator, we need to include the header file in our project and call the necessary functions to initialize the audio device and play the sound waves. Miniaudio provides a simple API, documented extensively in the project's website. For our keylogger, we will modify the simple_playback_sine example provided by miniaudio to play the sound waves generated by the modulator. We must set some parameters, such as the sample rate, the number of channels, and the buffer size, to configure the audio device. We will also need to generate the wave, as explained in Example 4.6.1. After configuring the audio device and generating the wave, we can call the init function to initialize the audio device and the play function to start playing the sound waves. We have chosen to add a delay between each keystroke to make the transmission more covert. The chosen delay is a fixed value of 1 second, but it could be randomized to make the transmission even more covert.

11.4. Clipping

Up to this point, we have discussed how to generate sound waves from the keystrokes and play them using miniaudio.h. However, after implementing the modulator, we noticed that after each keystroke, there was a noticeable click or pop sound. This is due to the abrupt change in amplitude when the sound wave transitions from silence to the keystroke sound and back to silence. Even though the actual sound wave is correct, the abrupt change in amplitude causes the click sound, which is very clearly audible and defeats the purpose of a covert channel. To address this issue, we need to smooth out the transitions between the keystroke sound and silence, preventing the abrupt changes in amplitude that cause the click sound. This is done by applying a fade-in and fade-out effect to the sound wave, gradually increasing and decreasing the amplitude at the beginning and end of the sound wave, respectively.

Let's define our original waveform as a function $f(t)$ where t is the time index of the sample. We want to apply a fade-in effect at the beginning and a fade-out effect at the end of the waveform.

For a linear fade, we can define a fade function $g(t)$ as follows:

$$g(t) = \begin{cases} \frac{t}{T}, & 0 \le t < T \\ 1, & T \le t < L - T \\ \frac{L-t}{T}, & L - T \le t < L \end{cases}$$

Where:

- T is the number of samples over which the fade occurs
- L is the total length of the waveform in samples

Plotting the fade function $g(t)$, we get the following graph:

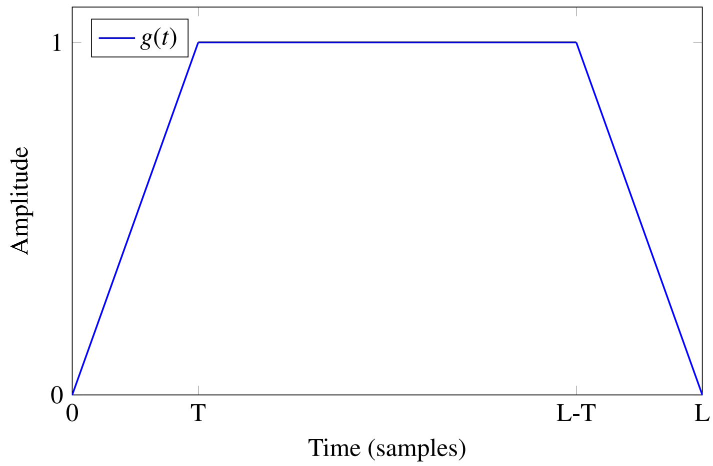


Fig. 11.3. Plot of the fade function $g(t)$

The faded waveform $f'(t)$ is then obtained by multiplying the original waveform by the fade function:

$$f'(t) = f(t) \cdot g(t)$$

This operation gradually increases the amplitude from 0 to full volume over the first T samples (fade-in), maintains the volume, and then decreases the amplitude back to 0 over the last T samples (fade-out). In the project, the implementation of the fade-in and fade-out effects is done in the apply_fade function, as shown below:

```c
void apply_fade(float *waveform, size_t length, size_t fade_samples
    ) {
    for (size_t i = 0; i < fade_samples; i++) {
        float fade = (float) i / fade_samples; // Linear fade
        waveform[i] *= fade; // Fade in
        waveform[length - 1 - i] *= fade; // Fade out
    }
}
```

In this implementation:

- `waveform` is an array representing $f(t)$
- `length` is L
- `fade_samples` is T
- The loop variable `i` represents t
- `fade` represents $g(t)$ for the fade-in and $g(L - t - 1)$ for the fade-out

With the fade-in and fade-out effects applied, the transitions between the keystroke sound and silence are smoothed out, eliminating the click sound and making the transmission inaudible to all but the youngest of ears.

12. FUTURE WORKS

Given the time constraints of the project, there are several areas that could be further explored and improved upon. There are also additional features that could be added to enhance the functionality and performance of the system. In this chapter we will discuss some of the potential future works that could be undertaken to improve the keylogger and data transmission system.

12.1. Error Detection and Correction

One of the key limitations of the current system is the lack of error detection and correction mechanisms. Since the data is transmitted over sound waves, it is very susceptible to interference and noise, which can corrupt the data and lead to errors in transmission. In our tests, we observed that noise in the environment can severely impact the quality of the transmission, leading to missing or incorrect keystrokes. To address this issue, we could implement error detection and correction mechanisms, such as checksums or error-correcting codes, to ensure the integrity of the transmitted data. Some of the techniques that could be explored include Cyclic Redundancy Checks (CRC), Hamming Codes, and Reed-Solomon Codes.

12.2. Improved Modulation Schemes

The current system uses Frequency Shift Keying (FSK) to modulate the data into sound waves. It is a simple and effective modulation scheme that works well for transmitting data over sound, but it has some limitations. For instance, it's not very efficient in terms of bandwidth usage, as it requires a separate frequency for each bit. Furthermore, we're also using binary FSK, which limits the data rate to 1 bit per symbol[7]. A simple improvement would be to use more frequencies to increase the data rate, so instead of using two frequencies for 0 and 1, we could use four frequencies to encode two bits per symbol. This would effectively double the data rate without increasing the bandwidth. More advanced modulation schemes, such as Orthogonal Frequency Division Multiplexing (OFDM), could also be explored to further increase the data rate and improve the efficiency of the transmission. However, these schemes are more complex and fall out of the scope of this project.

[7]Binary FSK is the simplest form of FSK, where only two frequencies are used. More complex variations like M-ary FSK utilize multiple frequencies to increase the data rate, at the cost of decreased robustness.

12.3. Different Transmission Mediums

While the focus of this project was on transmitting data over sound waves, there are other transmission mediums that could be explored. The data is encoded into sound waves, but it could also be transmitted over other mediums, such as light. For example, we could use an LED to transmit the data optically, encode the data into a single pixel on a screen, or even modulate the brightness of a monitor to transmit the data. This would allow us to transmit data over longer distances and in environments where sound transmission is not feasible, extending the exfiltration capabilities of the keylogger.

12.4. OpenBSD Support

The keylogger was developed for Linux, but it could be ported to other operating systems. The choice of Linux was made due to its popularity and the availability of the necessary tools and libraries, however, it is not the best choice for learning about operating systems. Linux uses GNU's coreutils, which are not the most user-friendly, and the kernel is quite complex, which can make it challenging to understand. On the other hand, OpenBSD follows a different approach, with a focus on simplicity, security, and correctness [13]. GNU, or as OpenBSD calls it, "Gigantic and Nasty" [14], contains very capable but bloated tools, which can make it difficult to understand how they work. As an example, the `echo` command, which simply prints its arguments to the standard output, has a drastically different implementation in GNU and OpenBSD. Both perform the same fundamental task, but the GNU version is 262 lines of code [15], while the OpenBSD version is only 33 lines [16]. Another example is the `ls` command, used to list files in a directory. The GNU version has a staggering 5650 lines of code [17], while the OpenBSD version has only 593 [18]. This makes OpenBSD a much better choice for learning about operating systems, as the code is simpler and easier to understand. Porting the keylogger to OpenBSD would not be without its challenges, as the operating systems have different system calls, libraries, and kernel interfaces. Fortunately, miniaudio.h does support OpenBSD, so the audio playback part of the keylogger should work without modification.

13. USES OF LLMS IN THE PROJECT

This chapter will discuss the applications of Large Language Models (LLMs) in this project, both in the development of the software and in the writing of this thesis. Some of the key uses of LLMs in the project include:

- Code generation.
- Code debugging.
- Text review and editing.

The chapter will also explore the limitations and challenges encountered when using LLMs in this project, and how they were addressed.

13.1. Models Used

During the development of the project, many LLMs were used, each with its own set of strength and weaknesses. GPT-4o by OpenAI, Claude 3.5 Sonnet by Anthropic, and Gemini 1.5 Pro by Google were the most used proprietary models. LLama 3.1 8b by Meta, Gemma 2 by Google and Codestral Mamba by Mistral AI were the most used open-source models. Each model was used for different tasks, such as code generation, debugging, and text editing, depending on its capabilities and performance.

13.2. Code Generation

One of the main uses of LLMs in the project was code generation. In this regard, the models were used to complete code snippets, debug code, and generate documentation. Copilot, which uses OpenAI's GPT-4o was extensively used as a code completion tool inside the IDE, and Claude 3.5 Sonnet was used to research and debug the codebase. These models are very powerful, and can generate working code snippets given a prompt, and enough context. To showcase the capabilities of these models, a simple test was conducted with Claude 3.5 Sonnet, where it was given a prompt to implement a ring buffer in the linux kernel to store keystrokes, as explained in Section 10.3. A ring buffer was already implemented in the project for the demodulator, albeit in Python, so reimplementing it manually in C would have been relatively easy, but time-consuming. Instead, the prompt was given to Claude 3.5, and to much astonishment, it was able to not only generate the implementation of the ring buffer, but also provided an implementation adapted to the existing codebase, and included the necessary modifications to the userspace program to read from the ring buffer.

The prompt given to Claude 3.5 Sonnet was *exactly* as follows:

> I have the following program that consists of two parts, a kernel module that logs keys and stores them in a buffer, and a userspace program that reads from that buffer. Right now the userspace program only reads once from the buffer, but I want to read continuously. Currently the reading operation does not remove anything from the buffer, so I need a way to ideally implement a ring buffer so that reading from the buffer removes the elements in order to read continuously and keep processing characters. How should I proceed? Here is the code:
>
> KERNEL MODULE:
>
> [Full code. Not shown for brevity, but included in the prompt]
>
> USERSPACE PROGRAM:
>
> [Full code. Not shown for brevity, but included in the prompt]

To which Claude 3.5 Sonnet responded with:

> *To implement continuous reading from the buffer and process characters in real-time, you'll need to make changes to both the kernel module and the userspace program. Here's how you can proceed:*
>
> *Kernel Module Changes:*
>
> *Implement a circular buffer in the kernel module:*

```
#define BUFFER_SIZE 1024

static char key_buffer[BUFFER_SIZE];
static int buf_read = 0;   // Read position
static int buf_write = 0; // Write position

static int buffer_full(void) {
    return (buf_write + 1) % BUFFER_SIZE == buf_read;
}

[rest of the implementation]
```

> *Modify the key_notify function to use the circular buffer:*

```
        [implementation]
```

> *Update the device_read function to read from the circular buffer:*

```
        [implementation]
```

> *Modify the read_keys_from_device function to read continuously:*

```
        [implementation]
```

> *Update the main function to use the new continuous reading approach:*

```
        [implementation]
```

13.3. Writing the Thesis

Gemini 1.5 Pro was used to write the first draft of the thesis. During the implementation of the keylogger, the main ideas, concepts, and issues encountered were noted, and thanks to the model's large context window, we were able to feed all the notes, along with the entire codebase into the model, and generate a coherent first draft of the thesis. This first draft contained a general overview of the project, and it was refined and expanded upon in subsequent drafts.

This document was written in LaTeX, and Claude 3.5 Sonnet was used to help with formatting and generating figures. The main use of Claude was to generate most of the figures and diagrams, and it was also used to help with the phrasing and style of the text. It is an invaluable tool for writing technical documents, as it can generate complex diagrams and figures mostly on its own, saving a lot of time and effort. However, these models are not perfect, and, as of now, they still require extensive human oversight and editing to produce a high-quality document.

13.4. Issues with LLMs in the Project

As powerful as LLMs are, they are not without their limitations and challenges. Similar to humans, LLMs can generate incorrect code and make mistakes. On the one hand, they can be very helpful in generating ideas, and help with research by pointing one in the right direction. However, they can very confidently generate incorrect information, which can be catastrophic if not caught in time. They can generate code that doesn't compile, which can be frustrating and time-consuming to debug, but they can also generate code that compiles and runs, but is incorrect, which can be even more dangerous. They can also generate code that is inefficient, or that doesn't follow best practices, which can lead to performance issues and security vulnerabilities. Therefore, it is important to use these tools just as what they are: tools. They are very helpful, but they should not be relied upon without oversight.

As an example of the limitations of LLMs, during the development of the demodulator, we encountered an issue where the model was unable to find a bug in the code.

```
    def process_audio_frames(self, preamble_signal, buffer_size):
        buffer = CircularBuffer(buffer_size, frame_size=
            SAMPLES_PER_FRAME)
        logging.info("Processing audio frames with preamble
            detection using circular buffer...")
        frames_processed = 0
        preamble_detected = False
        frames_after_preamble = []
        while not self.stop_flag or not self.frame_queue.empty():
            frame = self.get_frame()
            if frame is None:
```

```
                continue
            if not preamble_detected:
                preamble_detected, message = self.detect_preamble(buffer, frame, preamble_signal)
                if preamble_detected:
                    print("Preamble detected, message: ", message)
                    modem.save_to_wavfile("message.wav", signal, FS) # debug purposes
                    # The bug is here. There should be a continue
                    # statement to jump to the next frame.
                    # Fix:
                    continue

            if preamble_detected:
                frames_after_preamble = self.process_post_preamble(frames_after_preamble, frame, message)
                if not frames_after_preamble:  # Reset after processing
                    preamble_detected = False
                    buffer = CircularBuffer(buffer_size, frame_size=SAMPLES_PER_FRAME)

            frames_processed += 1
```

This code snippet shows the process_audio_frames method of the demodulator, which processes audio frames and detects a preamble signal. In its original implementation, the code was not correctly handling the case where the preamble was detected. When attempting to debug the issue, the relevant audio section was extracted and listened to, and it contained a slight pop sound in the middle of the transmission. This indicated that there was an issue somewhere in the code, that was causing the signal to have a cut in the middle. The issue was not immediately obvious, so after several hours of debugging, the code was fed into Claude 3.5 Sonnet, but it was unable to find the bug. Even worse, it confidently suggested other changes that did not address the issue, which led to further confusion. The code was also fed to GPT-4o, and Gemini 1.5 Pro, but they were also unable to find the bug. These models similarly suggested changes that did not address the issue, which was frustrating and time-consuming. Eventually, the bug was found by manually inspecting the code and running it step by step in a debugger, which revealed that there was a missing continue statement in the code. It was an extremely easy fix, and was quite obvious once found, but it was not caught by the models, demonstrating that they are not infallible.

13.5. A word on AI "detectors"

Since the rise in popularity of LLMs such as OpenAI's ChatGPT back in November of 2022, there has been a growing interest in developing software detectors to identify and dissuade its use in academic and professional settings. While the intentions behind these detectors may be noble, they have been met with harsh criticism from the AI community, who argue that they are ineffective and often produce false positives OpenAI created a classifier to detect AI-written text, but it was quickly found to be innefective, and was discontinued in 2023. It was announced in their blog that the classifier would be discontinued due to its low rate of accuracy.

> *"As of July 20, 2023, the AI classifier is no longer available due to its low rate of accuracy. We are working to incorporate feedback and are currently researching more effective provenance techniques for text, and have made a commitment to develop and deploy mechanisms that enable users to understand if audio or visual content is AI-generated." [19]*

This classifier, however, was not and is not the only one available today. One Google search for "AI text detector" will yield hundreds of results, with many companies offering their own solutions. These companies claim that their detectors can identify AI-written text with an extremely high degree of accuracy, but this is often not the case. As an example, parts of this document were fed into one of these detectors, and it was flagged as being AI-generated, even though it was written by a human. Even more baffling, parts of this document that were actually written with the help of an AI were not flagged as such.

Section 9.2 was fully written by hand, but running it through Originality.ai's AI detector flagged it as AI-generated.

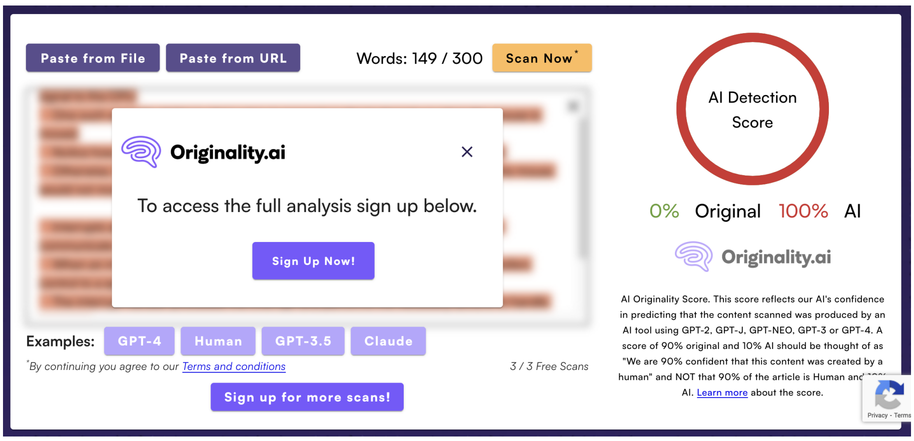


Fig. 13.1. Screenshot of the AI detector results on section 9.2

On the other hand, section 3.3 was written by hand, and then fed into Claude 3.5 Sonnet for better phrasing, yet it was flagged as only "27% AI-generated".

Regarding false positives, Originality AI has a page on their blog recommending users to download a Google Chrome extension that runs in the background while users edit documents in Google Docs, and tracks the entire edit history of the document. This way, if a user is flagged as having written AI-generated text, they can prove that they wrote the text by hand, and didn't simply copy and paste it from an AI model. Setting aside the **extreme** privacy concerns that this raises, plus the inherently flawed logic of proving a negative[8], it is also not a foolproof solution, as it relies on the user having the extension installed and running at all times, and it doesn't account for text that was written in a different editor, or even copied and pasted from another source, making it an unreliable method of proving authorship.

AI tools are here to stay, and they are becoming increasingly more powerful and useful. Just like any other new technology in the past, there is a period of adjustment where the potential of the technology is explored, and the limitations are discovered. This chapter has tried to shed some light on some of these limitations, but also on the great potential that these tools have, and how they can be used to enhance academic and professional work. As mentioned earlier, they are tools, and when used as such they can be extremely helpful.

[8] See Russell's teapot.

14. CONCLUSIONS

In this project, we have developed a covert data transmission system that uses sound waves to exfiltrate data from an air-gapped system. The system consists of a kernel-level keylogger that captures keystrokes and transmits them over sound waves using Frequency Shift Keying (FSK). The keylogger is implemented as a character device driver in the Linux kernel, which captures keystrokes and stores them in a ring buffer. The userspace modulator reads the keystrokes from the ring buffer, encodes them into sound waves, and plays them using the miniaudio.h library. The sound waves are transmitted over the air and received by a microphone connected to a receiver system, which decodes the sound waves back into keystrokes. The system was tested in a controlled environment, and the results showed that it is capable of transmitting data over sound waves with high accuracy and reliability.

14.1. Legal and Ethical Considerations

While the primary objective of this project is educational, demonstrating techniques for creating and detecting covert channels, it is imperative to address the potential for misuse and the ethical implications associated with this technology.

14.1.1. Dual-Use Dilemma

Keylogging technology inherently presents a dual-use dilemma[9]. It could be used for legitimate purposes, such as tracking computer usage in an organization, but it could also be used for malicious purposes, such as stealing sensitive information.

This project is developed solely for educational and research purposes, as a proof-of-concept to demonstrate the feasibility of transmitting data over sound waves. Any use of this technology outside of a controlled environment without the explicit consent of all parties involved would be considered unethical and potentially illegal.

14.1.2. Ethical Implications

The technology demonstrated in this project can be used to capture and exfiltrate sensitive personal information, including passwords, financial data, and private communications, making its deployment without explicit informed consent from the user a severe breach of privacy. Moreover, this technology can be very easily misused for malicious purposes.

[9] Stricly speaking, the dual-use dilemma refers to the development of technology that can be used for civilian and military purposes, but in this context, it refers to the potential for technology to be used for both legitimate and malicious purposes.

The potential for harm is significant, and the goal of this project is to raise awareness of the risks associated with ultrasonic data transmission, as it is a relatively unknown area outside of specialized fields.

14.1.3. Legal Framework: Spanish and European Law

The legal framework surrounding the technology demonstrated in this project in Spain is primarily governed by the Ley Orgánica 3/2018, de 5 de diciembre, de Protección de Datos Personales y garantía de los derechos digitales, or LOPDGDD, which implements the European General Data Protection Regulation (GDPR) at the national level. Under the LOPDGDD and GDPR, capturing keystrokes, even for educational purposes, is considered processing of personal data and therefore requires a legal basis. Keystroke data, by its nature, could reveal information about a person's ideology, beliefs, or other personal aspects, potentially falling under "special categories of data", as defined in Article 9.1 of the LOPDGDD:

> *"A los efectos del artículo 9.2.a) del Reglamento (UE) 2016/679, a fin de evitar situaciones discriminatorias, el solo consentimiento del afectado no bastará para levantar la prohibición del tratamiento de datos cuya finalidad principal sea identificar su ideología, afiliación sindical, religión, orientación sexual, creencias u origen racial o étnico.[10]"*

The article makes it clear that consent alone is not sufficient to process data that could reveal sensitive information about an individual, such as their ideology, religion, or sexual orientation. Therefore, capturing keystrokes, even for educational purposes, would require a legal basis other than consent, such as a legitimate interest or legal obligation.

Additionally, the Spanish Penal Code (Código Penal) criminalizes the unauthorized access to computer systems and the interception of communications, under Article 197 bis:

> *"1. El que por cualquier medio o procedimiento, vulnerando las medidas de seguridad establecidas para impedirlo, y sin estar debidamente autorizado, acceda o facilite a otro el acceso al conjunto o una parte de un sistema de información o se mantenga en él en contra de la voluntad de quien tenga el legítimo derecho a excluirlo, será castigado con pena de prisión de seis meses a dos años [11]."*

[10]**Unofficial translation:** For the purposes of Article 9.2.a) of Regulation (EU) 2016/679, in order to avoid discriminatory situations, the mere consent of the data subject will not be sufficient to lift the prohibition on the processing of data whose main purpose is to identify their ideology, trade union affiliation, religion, sexual orientation, beliefs or racial or ethnic origin.

[11]**Unofficial translation:** 1. Anyone who, by any means or procedure, violating the security measures established to prevent it, and without being duly authorized, accesses or facilitates access to all or part of an information system or remains in it against the will of the person who has the legitimate right to exclude it, shall be punished with imprisonment from six months to two years.

> *2. El que mediante la utilización de artificios o instrumentos técnicos, y sin estar debidamente autorizado, intercepte transmisiones no públicas de datos informáticos que se produzcan desde, hacia o dentro de un sistema de información, incluidas las emisiones electromagnéticas de los mismos, será castigado con una pena de prisión de tres meses a dos años o multa de tres a doce meses*[12].

The article criminalizes unauthorized access to computer systems, which would be necessary to install the keylogger in the first place, and the interception of communications, which is the main purpose of the keylogger. As such, the use of the technology demonstrated in this project could be considered a criminal offense under Spanish law. This overview of the legal framework is not exhaustive, and **any use of this technology must be carefully considered in light of the applicable laws and regulations.**

14.2. Socioeconomic Impact Analysis

The successful implementation of this project demonstrates the feasibility of a sound-based keystroke monitor for Linux, opening a pathway for covert data exfiltration. While the project itself is framed as an educational tool, the potential socioeconomic impact of its application warrants a critical analysis.

The potential for misuse of the technology demonstrated in this project is substantial, as one of the main uses would be exfiltrating sensitive information from air-gapped systems. This poses a significant risk to individual privacy and cybersecurity, as it could be used to steal personal data, sensitive information, or intellectual property.

The technology could be used by malicious actors to target individuals, organizations, or even governments, with the potential for significant financial, reputational, and national security consequences.

Having the keylogger implemented as a kernel module has no justification other than to make it harder to detect and to make it more difficult to remove, so it is clear that the main purpose of this project is to demonstrate the dangers behing this technology, rather than to provide a useful tool for legitimate purposes. There is no legitimate reason to use this technology in practice, as any use would require explicit informed consent from all parties involved, and with it there would be no need to hide the keylogger in the kernel.

However, just because the technology demonstrated in this project has the potential for misuse, it does not mean that it should not be explored. This project demonstrates the importance of understanding all attack vectors, even those that are less well-known, to develop effective countermeasures and protect against potential threats.

[12]**Unofficial translation:** 2. Anyone who, by using artifices or technical instruments, and without being duly authorized, intercepts non-public transmissions of computer data that occur from, to or within an information system, including their electromagnetic emissions, shall be punished with imprisonment from three months to two years or a fine from three to twelve months.

BIBLIOGRAPHY

[1] A. P. Jana, *Spy*, 2020. [Online]. Available: https://github.com/jarun/spy.

[2] B. Smus. "Ultrasonic networking on the web." (Aug. 2013), [Online]. Available: https://smus.com/ultrasonic-networking/ (visited on 08/27/2024).

[3] B. S. Alex Kauffmann. "Tone: An experimental chrome extension for instant sharing over audio," Google Research. (May 2015), [Online]. Available: https://research.google/blog/tone-an-experimental-chrome-extension-for-instant-sharing-over-audio/ (visited on 08/27/2024).

[4] A. Oppenheim and R. Schafer, *Discrete-time Signal Processing* (Prentice-Hall signal processing series). Pearson, 2010. [Online]. Available: https://books.google.es/books?id=mYsoAQAAMAAJ.

[5] "Ieee standard for wireless lan medium access control (mac) and physical layer (phy) specifications," *IEEE Std 802.11-1997*, pp. 1–445, 1997. DOI: 10.1109/IEEESTD.1997.85951.

[6] T. Darwich and C. Cavanaugh, "Amplitude shifting for sidelobes cancellation pulse compression," Center for Advanced Computer Studies, University of Louisiana at Lafayette, Lafayette, LA, Technical Report TR-2006-4-001, May 2006.

[7] D. M. Ritchie, "The evolution of the unix time-sharing system," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 6 Part 2, pp. 1577–1593, 1984.

[8] Toresbe, *PDP-7 at the Oslo University Museum*, https://upload.wikimedia.org/wikipedia/commons/5/52/Pdp7-oslo-2005.jpeg, Licensed under CC BY-SA 1.0 via Wikimedia Commons, 2005.

[9] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall, Jan. 2006.

[10] L. Torvalds and T. L. K. Organization, *Linux kernel*, version 6.1, 2023. [Online]. Available: https://www.kernel.org/.

[11] T. L. K. Organization, *Linux kernel documentation*, 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/.

[12] D. Reid, *Miniaudio*, 2020. [Online]. Available: https://github.com/mackron/miniaudio.

[13] O. Foundation. "Openbsd." Accessed: 2024-08-30. (2024), [Online]. Available: https://www.openbsd.org/.

[14] O. Foundation. "Openbsd's comments on gnu." Accessed: 2024-09-01. (2024), [Online]. Available: https://cvsweb.openbsd.org/src/gnu/README?rev=1.2&content-type=text/x-cvsweb-markup.

[15] B. Fox and C. Ramey, *Gnu coreutils echo source code*, Accessed: 2024-09-01, 2024. [Online]. Available: https://git.savannah.gnu.org/cgit/coreutils.git/tree/src/echo.c.

[16] O. Foundation, *Openbsd echo source code version 1.11*, Accessed: 2024-09-01, 2023. [Online]. Available: https://cvsweb.openbsd.org/src/bin/echo/echo.c?rev=1.11&content-type=text/x-cvsweb-markup.

[17] R. Stallman and D. MacKenzie, *Gnu coreutils ls source code*, Accessed: 2024-09-01, 2024. [Online]. Available: https://git.savannah.gnu.org/cgit/coreutils.git/tree/src/ls.c.

[18] O. Foundation, *Openbsd ls source code version 1.56*, Accessed: 2024-09-01, 2023. [Online]. Available: https://cvsweb.openbsd.org/cgi-bin/cvsweb/src/bin/ls/ls.c?rev=1.56&content-type=text/x-cvsweb-markup.

[19] J. H. Kirchner, L. Ahmad, S. Aaronson, and J. Leike. "New AI classifier for indicating AI-written text." Accessed: 2024-08-30, OpenAI. (Jan. 2023), [Online]. Available: https://openai.com/index/new-ai-classifier-for-indicating-ai-written-text/.

[20] B. Armstrong, *Quiet*, 2019. [Online]. Available: https://github.com/quiet/quiet.

[21] D. Sarwate. "Lecture 03: Matched filters – part i." Accessed: 2024-08-01. (2011), [Online]. Available: https://courses.grainger.illinois.edu/ece361/sp2011/Newlectures/Lecture03.pdf.

[22] A. Pandey, R. Asim, K. Mengal, M. Varvello, and Y. Zaki, *Sonic: Connect the unconnected via fm radio sms*, 2024. arXiv: 2407.01738 [cs.NI]. [Online]. Available: https://arxiv.org/abs/2407.01738.

[23] J. Gillham. "Ai content detector false positives – accused of using chat gpt or other ai?" Accessed: 2024-08-30, Originality.ai. (Aug. 2024), [Online]. Available: https://originality.ai/blog/ai-content-detector-false-positives.

[24] L. Torvalds. "Initial support for rust in the linux kernel." (2022), [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b.

A. HELLO, KERNEL!

This chapter will guide you through the process of writing a simple kernel module for Linux. We will cover the following topics:

- Setting up the development environment.
- Writing a basic kernel module.
- Compiling and loading the module.
- Interacting with the module.
- Unloading the module.

A.1. Setting up the Development Environment

Before we begin writing our kernel module, we need to set up the development environment. We'll be using a Linux system with the necessary tools installed.

A.1.1. Installing the Required Tools

To develop kernel modules, we need the following tools:

- **GCC:** The GNU Compiler Collection, which includes the C compiler.
- **Make:** A build automation tool that simplifies the compilation process.
- **Kernel Headers:** Header files that define the interfaces to the Linux kernel.

These tools can be installed using the package manager of your Linux distribution. For debian-based systems, you can install them using the following command:

```
sudo apt install build-essential linux-headers-$(uname -r)
```

For other distributions, you may need to adjust the package names accordingly.

A.1.2. Creating a Working Directory

Now, create a working directory for your kernel module project.

```
mkdir kernel-module
cd kernel-module
```

This directory will contain all the files related to your kernel module.

A.2. Writing a Basic Kernel Module

Now that we have set up the development environment, let's write a basic kernel module. We will create a simple module that prints a message when it is loaded and unloaded.

A.2.1. Creating the Module Source File

Create a new file named `hello.c` in your working directory and add the following code:

```c
#include <linux/init.h>
#include <linux/module.h>

static int __init hello_init(void)
{
    printk(KERN_INFO "Hello, kernel!\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "Goodbye, kernel!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple kernel module");
```

This code defines a simple kernel module that prints "Hello, kernel!" when it is loaded and "Goodbye, kernel!" when it is unloaded.

A.2.2. Compiling the Module

To compile the module, create a `Makefile` in the same directory as your `hello.c` file with the following content:

```makefile
obj-m += hello.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

This Makefile tells the build system to compile the `hello.c` file into a loadable kernel module named `hello.o`. The -C flag specifies the kernel source directory, needed to build kernel modules. The `all` target compiles the module, and the `clean` target removes the compiled files. $ (shell uname -r) is a command substitution that gets the current kernel version, which is used to locate the kernel headers.

Run the following command to compile the module:

```
make
```

If the compilation is successful, you should see a new file named `hello.ko` in your working directory.

A.2.3. Loading the Module

To load the module, run the following command:

```
sudo insmod hello.ko
```

The command will load the module into the kernel. To see the output of the `printk` function, run the following command:

```
dmesg | tail
```

Dmesg prints the kernel log buffer, and the tail command shows the last few lines of the log. The operator `|` is a pipe, which connects the output of the `dmesg` command to the input of the `tail` command.

You should see the message "Hello, kernel!" in the output.

A.2.4. Unloading the Module

Similarly, to unload the module, run the following command:

```
sudo rmmod hello
```

The command will remove the module from the kernel, and you should see the message "Goodbye, kernel!" in the log.

B. REQUIREMENTS SPECIFICATION

Below are the functional and non functional requirements for the project.

B.1. Functional Requirements

ID	Name	Description
FR001	Key Logging	The system must capture keystrokes from the keyboard.

Table B.1. KEY LOGGING

ID	Name	Description
FR002	Keystroke Encoding	Each keystroke should be encoded into a binary representation.

Table B.2. KEYSTROKE ENCODING

ID	Name	Description
FR003	Audio Modulation	The encoded binary data should be modulated onto an audio carrier signal using Frequency Shift Keying (FSK).

Table B.3. AUDIO MODULATION

ID	Name	Description
FR004	Audio Playback	The modulated audio signal should be played through the computer's speakers.

Table B.4. AUDIO PLAYBACK

ID	Name	Description
FR005	Preamble Generation	A predefined preamble should be generated and prepended to the encoded keystroke data.

Table B.5. PREAMBLE GENERATION

ID	Name	Description
FR006	Audio Recording	The system must be able to record audio from the microphone.

Table B.6. AUDIO RECORDING

ID	Name	Description
FR007	Preamble Detection	The system must be able to detect the predefined preamble in the recorded audio stream.

Table B.7. PREAMBLE DETECTION

ID	Name	Description
FR008	Audio Demodulation	After detecting the preamble, the system should demodulate the audio signal to recover the encoded binary data.

Table B.8. AUDIO DEMODULATION

ID	Name	Description
FR009	Binary Decoding	The recovered binary data should be decoded back into the original keystrokes.

Table B.9. BINARY DECODING

ID	Name	Description
FR010	Keystroke Display	The decoded keystrokes should be displayed to the user.

Table B.10. KEYSTROKE DISPLAY

B.2. Non-Functional Requirements

ID	Name	Description
NFR001	Low Latency	The keystroke capture and playback should have minimal latency.

Table B.11. LOW LATENCY

ID	Name	Description
NFR002	Real-Time Processing	Audio processing and demodulation should happen in real-time.

Table B.12. REAL-TIME PROCESSING

ID	Name	Description
NFR003	Platform Compatibility	The key logger (C code) should be compatible with the target operating system (Linux). The demodulator (Python code) should run on a system with a compatible Python environment.

Table B.13. PLATFORM COMPATIBILITY

ID	Name	Description
NFR004	Reliability	The system should reliably capture and transmit keystrokes without data loss or corruption.

Table B.14. RELIABILITY

ID	Name	Description
NFR005	Maintainability	The code should be well-structured and documented for easy maintenance and future modifications.

Table B.15. MAINTAINABILITY

ID	Name	Description
NFR006	Usability	While not directly user-facing, the output of the decoded keystrokes should be presented in a clear and understandable format.

Table B.16. USABILITY

ID	Name	Description
NFR007	Resource Usage	The system should not consume excessive CPU or memory resources.

Table B.17. RESOURCE USAGE

C. SOURCE CODE

The source code for this project can be found on GitHub at the following link:

https://github.com/mjorgers/thesis

The repository contains the following files:

- **key_logger.c:** The kernel module that captures keystrokes.
- **Makefile:** The Makefile for compiling the kernel module.
- **modulator.cpp:** The userspace program that encodes and modulates the keystrokes.
- **miniaudio.h:** The header file for the miniaudio library used for audio playback.
- **CMakeLists.txt:** The CMake configuration file for compiling the modulator.
- **demodulator.py:** The Python script that demodulates and decodes the keystrokes.
- **modem.py:** Python script containing auxiliary functions for the demodulator.

There is also a folder named **extra_scripts** that contains additional scripts used during the development and testing of the project.

The project was developed and tested on a Linux system running Linux Mint 21.3 with kernel version 5.15.0-118-generic.