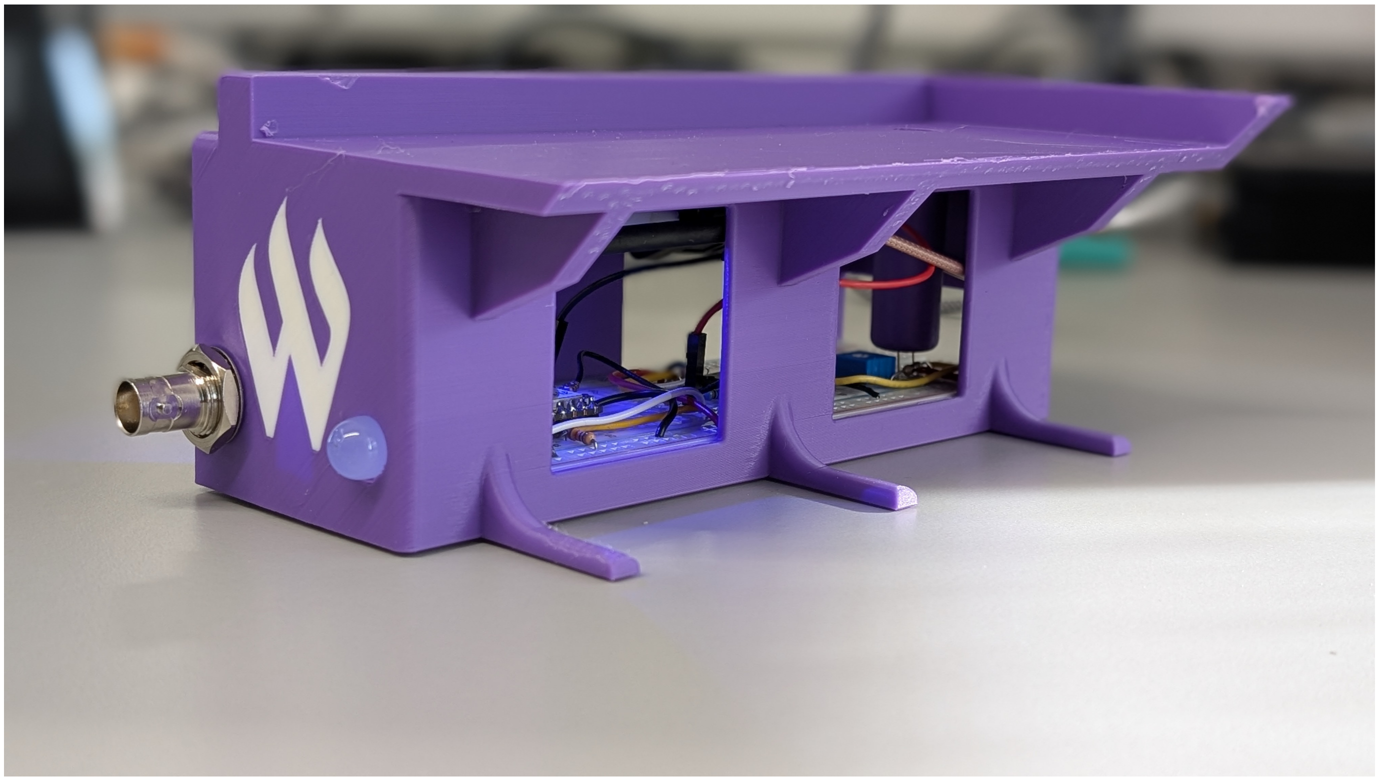
Morse Code Optical Receiver



Tanner Nelson
ECE 5440/6440 Extra Credit Project
Weber State University
Email: tannernelson3@mail.weber.edu

***Abstract*—This project report describes the design, construction, and testing of a Morse Code Optical Receiver. The system uses a photodiode transimpedance amplifier to detect light signals from a smartphone flash. The smartphone Morse code app generates pulses between 1 Hz and 100 Hz, and the circuit is designed to filter out high-frequency interference while ensuring proper operation in this range. The design includes bandwidth-limiting components and response-time calculations for optimal performance. Test results, oscilloscope outputs, and measured performance for decoding Morse code are presented.**

***Index Terms*—Morse Code, Photodiode, Transimpedance Amplifier, ESP32C3, Bandwidth, Circuit Design, Optical Receiver**

I. Introduction

This project demonstrates the design, construction, and testing of a photodiode-based optical receiver capable of decoding Morse code signals transmitted via a smartphone flash. The primary goal was to design a circuit to operate within the smartphone Morse code app's frequency range of 1 Hz to 100 Hz while rejecting high-frequency noise. Additional considerations include circuit stability, noise reduction, and real-world testing of the system's response to Morse code messages. The circuit also integrates an ESP32C3 microcontroller for decoding and displaying Morse code via a web interface.

II. Circuit Design

A. Photodiode Stage and Bandwidth Limiting

The BPX61 photodiode generates a photocurrent proportional to the incident light intensity. A parallel capacitor of $C_{ext} = 220\,pF$ was added to limit the bandwidth of the photodiode stage, ensuring proper operation in the 1–100 Hz range.

1) Photodiode Bandwidth: The photodiode has an internal capacitance of $C_d = 72\,pF$. The total capacitance is:

$$C_{total} = C_d + C_{ext} = 72\,pF + 220\,pF = 292\,pF$$

With a 1 M resistor (R_L) in parallel, the cutoff frequency is:

$$f_c = \frac{1}{2\pi R_L C_{total}} = \frac{1}{2\pi(1\times10^6)(292\times10^{-12})} \approx 545.7\,Hz$$

This bandwidth provides a generous range for the 1–100 Hz operation while filtering out high-frequency noise effectively.

B. Transimpedance Amplifier and Feedback Stability

The first stage is a transimpedance amplifier (TIA) using the LM358 op-amp. The feedback network consists of $R_f = 4.7\,k\Omega$ and $C_f = 47\,nF$. The capacitor C_f stabilizes the op-amp and attenuates high-frequency noise.

1) Feedback Network Bandwidth: The feedback network bandwidth is:

$$f_c = \frac{1}{2\pi R_{effective} C_f}$$

where $R_{effective}$ accounts for the parallel combination of R_f (4.7 k), the 1 k resistor to ground, and the 330 resistor to the second stage. The effective resistance is:

$$R_{effective} = \frac{1}{\frac{1}{R_f} + \frac{1}{330}} \approx 308$$

Substituting $C_f = 47\,nF$:

$$f_c = \frac{1}{2\pi(308)(47 \times 10^{-9})} \approx 11\,kHz$$

This high cutoff frequency ensures the feedback network does not interfere with the photodiode bandwidth.

The relationship between the output voltage, reference voltage, photocurrent, and feedback resistance is:

$$V_{out} = V_{ref} - I_d \cdot R_f$$

Where: - $V_{ref} = 3.3\,V$ - $R_f = 47\,k\Omega$

Calculation Details: 1. For $I_d = 0\,\mu A$:

$$V_{out} = V_{ref} - (0 \cdot R_f) = 3.3\,V$$

2. For $I_d = 55\,\mu A$:

$$V_{out} = V_{ref} - (55 \times 10^{-6}) \cdot 47,000$$

$$V_{out} = 3.3 - 2.585 = 0.715\,V$$

3. For $I_d = 70\,\mu A$:

$$V_{out} = V_{ref} - (70 \times 10^{-6}) \cdot 47,000$$

$$V_{out} = 3.3 - 3.29 = 0.01\,V$$

C. Second Stage Voltage Inverter

The second op-amp stage inverts and conditions the signal for digital processing. Its gain is set by $R_f = 30\,k\Omega$ and $R_{in} = 300$:

$$Gain = -\frac{R_f}{R_{in}} = -100$$

1) Calculation Details for Second Stage Output (V_{out2}): 1. For $I_d = 0\,\mu A$:

$$V_{out2} = V_{ref} - (-100 \cdot (V_{ref} - V_{out1}))$$

$$V_{out2} = 3.3 - (-100 \cdot (3.3 - 3.3))$$

$$V_{out2} = 3.3 - 0 = 0\,V(Saturated)$$

2. For $I_d = 55\,\mu A$:

$$V_{out2} = V_{ref} - (-100 \cdot (V_{ref} - V_{out1}))$$

$$V_{out2} = 3.3 - (-100 \cdot (3.3 - 0.715))$$

$$V_{out2} = 3.3 - (-100 \cdot 2.585)$$

$$V_{out2} = 3.3 + 258.5 = 3.3\,V(Saturated)$$

3. For $I_d = 70\,\mu A$:

$$V_{out2} = V_{ref} - (-100 \cdot (V_{ref} - V_{out1}))$$

$$V_{out2} = 3.3 - (-100 \cdot (3.3 - 0.01))$$

$$V_{out2} = 3.3 - (-100 \cdot 3.29)$$

$$V_{out2} = 3.3 + 329 = 3.3\,V(Saturated)$$

Output Saturation and Input Resistance Considerations:

The second stage was designed to invert and condition the signal for digital processing. This stage of the circuit was not required, but made the digitizing of the signal, in preparation for software decoding, much easier. While the op-amp saturates during operation, this behavior was deliberately maintained for the following reasons:

1) A low R_{in} reduces the loading effect on the output of the first stage, ensuring accurate signal transfer and preventing distortion in the low-voltage range.
2) The saturation of the second stage ensures binary-like output levels:
   - High output ($\approx 3.3\,V$) for low input voltages.
   - Low output ($\approx 0\,V$) for high input voltages.
3) Although saturation introduces minor delays due to recovery time, the LM358 op-amp transitions out of saturation within microseconds. To be extra cautious, the response time was measured before this stage.

TABLE I
CALCULATED OUTPUTS

Photocurrent (I_d)	$V_{out1}\,(V)$	$V_{out2}\,(V)$
0 uA	3.3	0
55 uA	0.715	3.3
70 uA	0.01	3.3

D. Calculated Response Time

The response time of the photodiode circuit was measured after the first stage, before the last stage. It is primarily determined by the RC time constant of the system, which is dominated by the photodiode's intrinsic properties and the feedback network. In this case, the photodiode's anode is connected to the inverting input of the first op-amp with a parallel load resistance $R_L = 1\,M\Omega$.

The RC time constant is calculated as:

$$\tau_{effective} = R_L \cdot C_{total}$$

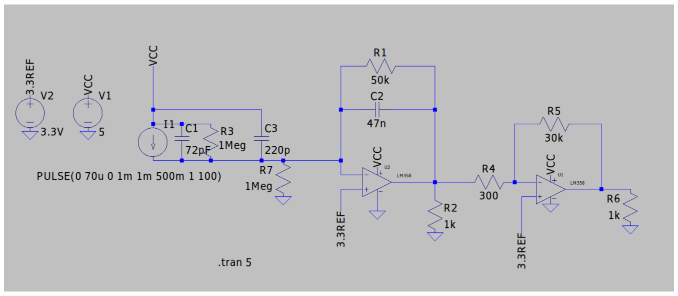
where:

$$C_{total} = C_d + C_{ext} = 72\,pF + 220\,pF = 292\,pF.$$

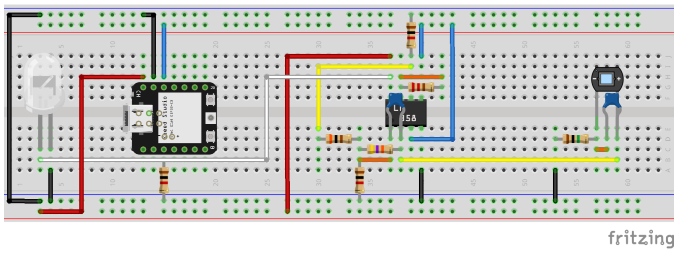
Substituting $R_L = 1\,M\Omega$:

$$\tau_{effective} = (1 \times 10^6) \cdot (292 \times 10^{-12}) = 292\,\mu s.$$

Full Schematic:



Connection Schematic:



III. 3D Model and Enclosure

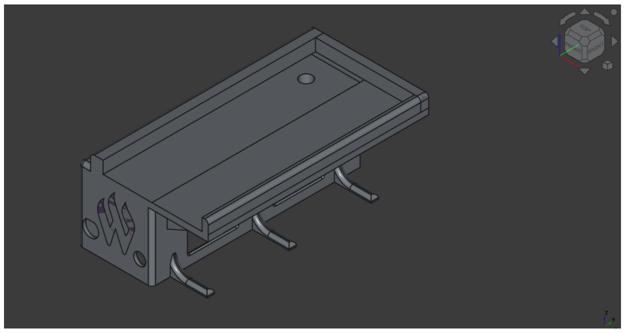
A custom 3D-printed enclosure was designed to house the optical receiver circuit, providing alignment and protection for consistent operation. The enclosure was specifically tailored to fit a breadboard containing the entire circuit and included strategically placed ports for:

- **Power Input:** A USB-C port for powering the ESP32C3 microcontroller and circuit.
- **Oscilloscope Measurements:** A BNC port for directly connecting the circuit to an oscilloscope for testing and verification.
- **LED Indicator:** A small window to allow visibility of the LED indicator on the circuit.

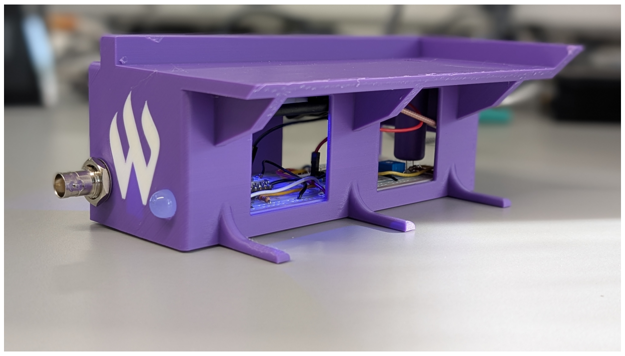
The enclosure also incorporates guides to align the Google Pixel 9 Pro's flashlight with the photodiode. This ensures that the phone's flash is centered and remains at a fixed distance of 40 mm from the photodiode. This spacing was determined to be optimal for generating a consistent voltage output from the photodiode while minimizing signal noise. The enclosure design ensures that measurements remain consistent across repeated tests by maintaining a fixed alignment and spacing.

The enclosure design was modeled in FreeCAD and printed using PLA material for rigidity and precision. A render of the 3D model and a photo of the printed enclosure are shown below:

- **3D Model Render:**



- **Printed Enclosure:**



IV. Testing and Results

TABLE II
Measured Circuit Outputs

Condition	V_{inv}	V_{out1}	V_{out2}
LED off	$3.28\,V$	$3.41\,V$	$0.01\,V$
LED on @ 40mm	$3.70\,V$	$0.61\,V$	$2.91\,V$

A. Frequency Response and Decoding Accuracy

The circuit was tested with a smartphone app that pulses the flashlight at frequencies of 20 Hz, 50 Hz, 60 Hz, and 100 Hz. The oscilloscope plots of these are shown below. Additionally, a smartphone app that generates Morse code was used to flash ”WSU ECE” with a dot pulse length of 10ms and was successfully shown on the oscilloscope. The Morse code outputs were observed at the photodiode, the first op-amp stage, and the final output

B. Measured Response Time

Oscilloscope measurements with a 50 Hz flashlight signal show a rise time of:

$$t_{rise} = 16.39\,\mu s$$

and a fall time of:

$$t_{fall} = 556.70\,\mu s$$

The average response time is:

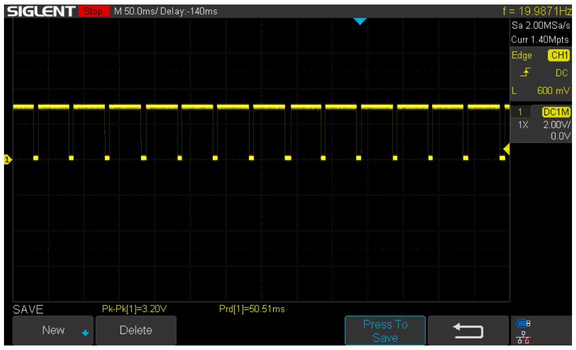
$$t_{avg} = \frac{t_{rise} + t_{fall}}{2}$$

$$t_{avg} = \frac{16.39 + 556.70}{2} \approx 286.19\,\mu s$$

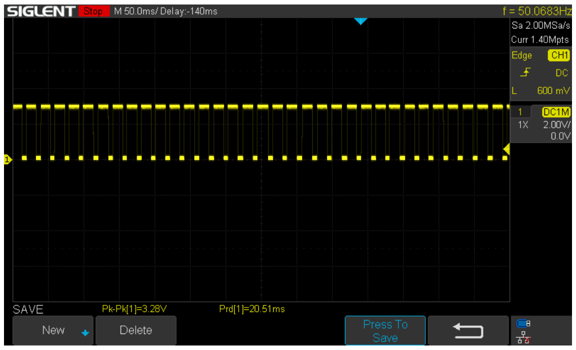
This measured response time closely matches the theoretical calculation of $292\,\mu s$.

C. Oscilloscope Outputs

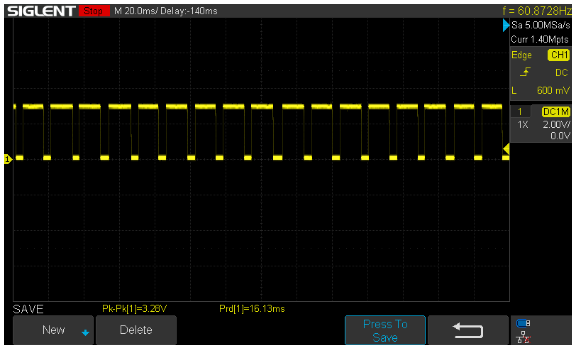
- **Frequency Test Outputs:**
  - Oscilloscope output at 20 Hz:



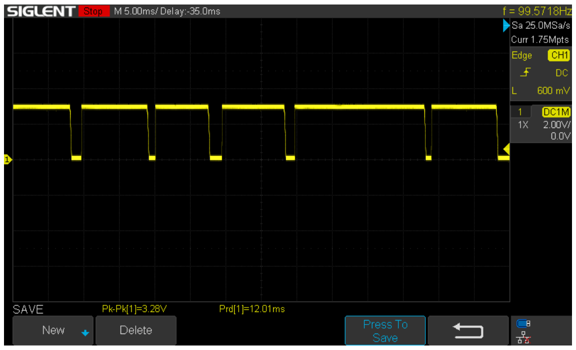
  - Oscilloscope output at 50 Hz:



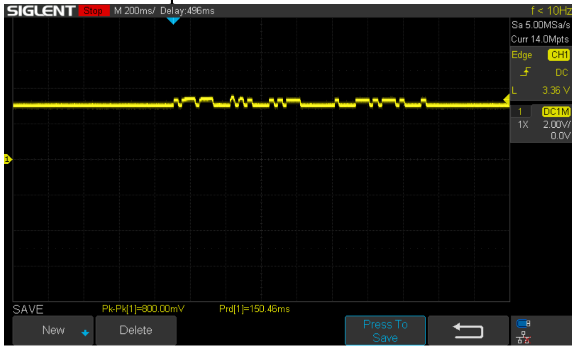
  - Oscilloscope output at 60 Hz:



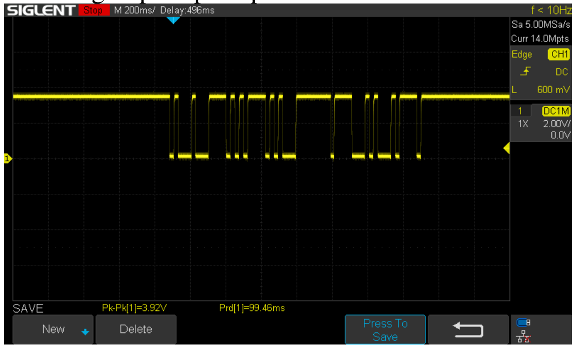
  - Oscilloscope output at 100 Hz:



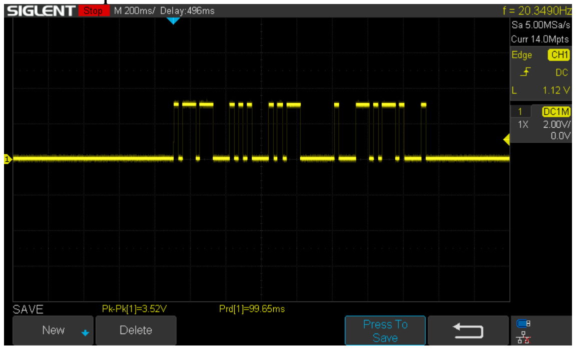
- **Morse Code Test:** ”WSU ECE” message:
  - Photodiode output:



  - First-stage op-amp output:



  - Final output:



V. Comparison with Smartphone Flash Specifications

To validate the performance of the optical receiver, we compared the measured photocurrent with theoretical calculations based on the specifications of the smartphone flash and the photodiode.

A. Assumptions and Parameters

- **Smartphone Flash Power Output** (P_{LED}): Assumed to be $80\,mW$, a typical value for smartphone LEDs.
- **Distance between Flash and Photodiode** (d): $40\,mm = 0.04\,m$.
- **Photodiode Active Area** ($A_{photodiode}$): $7.02\,mm^2 = 7.02 \times 10^{-6}\,m^2$.
- **Photodiode Responsivity** (R): $0.62\,A/W$ at $850\,nm$.

B. Calculation of Irradiance at the Photodiode

Assuming a Lambertian emission pattern, the irradiance (E) at normal incidence is given by:

$$E = \frac{P_{LED}}{\pi d^2}$$

Substituting the values:

$$E = \frac{0.08\,W}{\pi(0.04\,m)^2} = \frac{0.08}{\pi \times 0.0016} \approx 15.92\,W/m^2$$

C. Calculation of Incident Power on the Photodiode

The power incident on the photodiode is:

$$P_{incident} = E \times A_{photodiode} = 15.92\,W/m^2 \times 7.02 \times 10^{-6}\,m^2$$
$$\approx 111.8\,\mu W$$

D. Calculation of Photocurrent

Using the photodiode's responsivity:

$$I_{photo} = R \times P_{incident} = 0.62\,A/W \times 111.8\,\mu W \approx 69.3\,\mu A$$

E. Comparison with Measured Photocurrent

The calculated photocurrent of approximately $69.3\,\mu A$ closely matches the measured photocurrent of $70\,\mu A$. $80\,mW$ was chosen as an average value because this information is not known to be available to the public for this specific smartphone. However, this value could greatly impact the value. To obtain a more precise measurement, an LED with a known wattage would need to be utilized.

VI. ESP32C3 Morse Code Decoding Program

The ESP32C3 microcontroller runs a custom program written in MicroPython that decodes Morse code signals detected by the optical receiver circuit. The decoded signals are displayed through a web interface hosted on the ESP32C3, allowing users to view both the raw Morse code and the translated message.

A. Program Description

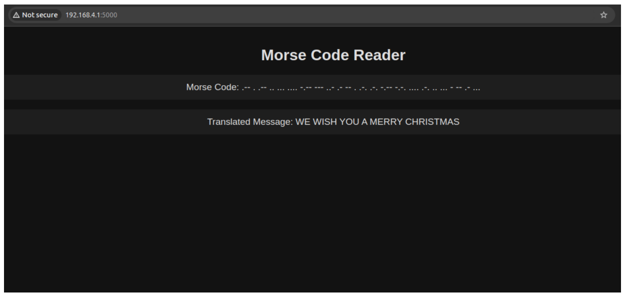
The program detects light pulses using an ADC connected to the final output of the optical receiver. The ADC values are compared against high and low thresholds to determine whether the signal is a dot, dash, or a gap. These are then processed into Morse code symbols and decoded into text using a predefined dictionary. The full source code for this program is included in **Appendix A**.

- **Timing Thresholds:** The program is configured to work optimally with dot pulses of 100 ms (100,000 s). The thresholds for dots, dashes, and gaps are adjusted based on this timing. Errors may occur with shorter pulse durations due to the limitations of MicroPython and the ESP32C3's processing speed when hosting a web server.
- **Web Interface:** The program includes a web server that displays the decoded message in real-time. Users connect to the ESP32C3's access point to access the interface.
- **Limitations:** The ESP32C3, running MicroPython, struggles to process pulses faster than approximately 100 ms due to timing inaccuracies introduced by multitasking between signal processing and the web server.

B. Performance Evaluation

Testing revealed the following:

- The program reliably decodes dot durations of 100 ms but starts to exhibit errors with shorter pulses due to processing limitations.
- The web server adds latency, impacting the detection of high-frequency signals.
- Despite these limitations, the program effectively decodes messages with dot-dash patterns and displays the results in real-time.
- This integration was tested with the input ”we wish you a merry christmas” and the following output was decoded properly:
- **Morse Decoder:**



Morse Code Reader

```
Morse Code: .-- . .-- .. ... .... -.-- --- ..- .- -- . .-. .-. -.-- -.-. .... .-. .. ... - -- .- ...
```

```
Translated Message: WE WISH YOU A MERRY CHRISTMAS
```

VII. Conclusion

This project successfully demonstrated the design, construction, and testing of a Morse Code Optical Receiver. Key design choices contributed to its success:

- **Amplifier Design:** The two-stage amplifier setup, including a transimpedance amplifier and voltage inverter, was chosen for its ability to convert small photodiode currents into usable voltages while maintaining signal integrity. This design ensured sensitivity to light signals and compatibility with digital processing.
- **Bandwidth Selection:** A bandwidth of 1–100 Hz was chosen to match the operational frequency range of the smartphone Morse code app while filtering out high-frequency noise. The addition of a parallel capacitor at the photodiode further improved signal quality by reducing noise.
- **Fixed Distance Alignment:** A distance of 40 mm between the photodiode and the phone's flash was determined to provide consistent voltage output. The 3D-printed enclosure ensured this distance was maintained, improving the reliability and repeatability of measurements.
- **Enclosure and Software Integration:** The custom 3D-printed enclosure and ESP32C3 program enhanced the system's usability and accuracy. The enclosure ensured proper alignment and spacing, while the program enabled real-time decoding and verification of Morse code signals.

Overall, this project successfully demonstrated the design and implementation of an optical receiver capable of decoding

Morse code. The careful selection of circuit components, bandwidth optimization, and integration of alignment and software tools ensured accurate and reliable performance. Future work could explore enhancing the enclosure for additional devices and optimizing the software for higher-speed signals.

APPENDIX

A. Program Code

Below is the full MicroPython code for the ESP32C3:

```
# main.py

from machine import ADC, Pin
from time import sleep_us, ticks_us,
ticks_diff
import network
import time
import uasyncio as asyncio
from microdot import Microdot
from _thread import start_new_thread,
allocate_lock
import gc
import _thread

# Timing thresholds (in microseconds)
for 200ms dot duration
DOT = 100_000  # 200 ms
DASH_DURATION = 3.15*DOT  # 600 ms
SYMBOL_GAP = DOT  # 200 ms
CHARACTER_GAP = 3*DOT  # 600 ms
WORD_GAP = 7*DOT  # 1400 ms
DOT_DURATION = 1.15 * DOT

# Configure ADC on pin 4
adc = ADC(Pin(4))
adc.atten(ADC.ATTN_0DB)
# Allows reading up to ~3.6V
adc.width(ADC.WIDTH_12BIT)
# 12-bit resolution (0-4095)

# Globals for Morse code and decoded message
morse_code = ""
decoded_message = ""
message_lock = allocate_lock()
# Lock for thread-safe access

# Morse code dictionary
MORSE_CODE_DICT = {
    ".-": "A", "-...": "B", "-.-.":
    "C", "-..": "D", ".": "E", "..-.":
    "F","--.": "G", "....": "H", "..":
    "I", ".---": "J", "-.-": "K", ".-..":
    "L","--": "M", "-.": "N", "---":
    "O", ".--.": "P", "--.-": "Q", ".-.":
    "R","...": "S", "-": "T", "..-":
    "U", "...-": "V", ".--": "W", "-..-":
    "X", "-.--": "Y", "--..": "Z", "-----":
    "0", ".----": "1", "..---": "2",
    "...--": "3", "....-": "4", ".....":
    "5", "-....": "6", "--...": "7",
    "---..": "8", "----.": "9"
}

HIGH_THRESHOLD = 2400
# 2.4V in ADC units
#(~2730 for 3.3V reference)
LOW_THRESHOLD = 500
# 0.5V in ADC units
#(~570 for 3.3V reference)

HIGH_THRESHOLD = 2400
# 2.4V in ADC units
#(~2730 for 3.3V reference)
LOW_THRESHOLD = 500
# 0.5V in ADC units
#(~570 for 3.3V reference)

# Initialize web server
app = Microdot()

def read_signal():
    """Read ADC value and return high/low
    based on thresholds."""
    adc_value = adc.read()
    if adc_value > HIGH_THRESHOLD:
        return 1  # High signal
    elif adc_value < LOW_THRESHOLD:
        return 0  # Low signal
    return -1  # Ignore noisy readings

def decode_morse(signal_sequence):
    """Decode a sequence of
    Morse code into text."""
    words = signal_sequence.split("   ")
    # Split by word gap
    decoded_text = []
    # Initialize an empty list
    # to store decoded words
    for word in words:
        symbols = word.split(" ")
        # Split by character gap
        decoded_word = ''.join
        ([MORSE_CODE_DICT.get(symbol,
        "?") for symbol in symbols])
        decoded_text.append(decoded_word)
    return ' '.join(decoded_text)
    # Join decoded words with spaces

def setup_ap():
    """Set up the ESP32 in
    Access Point mode."""
    ap = network.WLAN(network.AP_IF)
    ap.active(True)
    ap.config(essid="morse_code_reader",
    password="ece_6440",
    authmode=network.AUTH_WPA2_PSK)
    while not ap.active():
        time.sleep(0.1)
    print(f"Access Point ready, IP:
    {ap.ifconfig()[0]}")

@app.route("/")
def index(request):
    """Serve the index HTML file."""
    try:
        with open("index.html", "r") as file:
            html_content = file.read()
        print("Serving index.html")
        return html_content, 200, {"Content-
        Type": "text/html"}
    except OSError as e:
        print(f"Error serving index.html: {e}")
        return "Error: index.html not found", 404

@app.route("/message")
def message(request):
    """Return the current
    Morse code and decoded message."""
    with message_lock:
        return {
            "morse": morse_code,
            "message": decoded_message
        }
def run_web_server():
    """Run the Microdot web server."""
    print("Starting web server...")
    try:
        app.run(port=5000, debug=True)
    except Exception as e:
        print(f"Web server error: {e}")

def start_server_thread():
    """Start the web server
    in a separate thread."""
    print("Starting server thread...")
    try:
        _thread.start_new_thread
        (run_web_server, ())
    except Exception as e:
        print(f"Error starting server
        thread: {e}")

async def detect_morse():
    """Listen for and decode
    Morse code signals."""
    global morse_code, decoded_message
    print("Listening for Morse code...")
    signal_sequence = ""
    last_signal = 0
    last_change = ticks_us()
    # Microsecond precision

    while True:
        signal = read_signal()
        now = ticks_us()

        if signal != last_signal:
        # Signal changed
            duration = ticks_diff(now,
            last_change)
            last_change = now

            if last_signal == 1:
            # High signal ended
                if duration <=
                DOT_DURATION:
                    signal_sequence += "."
                elif duration <=
                DASH_DURATION:
                    signal_sequence += "-"

            elif last_signal == 0:
            # Low signal ended
                if duration >
                WORD_GAP:
                    signal_sequence += "   "
                    # Word gap
                elif duration >
                CHARACTER_GAP:
                    signal_sequence += " "
                    # Character gap

        last_signal = signal

        # Decode the message when idle
        if ticks_diff(ticks_us(), last_change)
        > 2_000_000 and signal_sequence:
            with message_lock:
                morse_code =
                signal_sequence.strip()
                decoded_message =
                decode_morse(morse_code)
            print(f"Raw Morse: {morse_code}")
            print(f"Decoded Message: {decoded_mess
            signal_sequence = ""
            # Reset for next message
        sleep_us(1)

async def main():
    """Main entry point for the program."""
    print("Starting main program")

    # Set up Access Point
    setup_ap()
    # Start the web server in a separate thread
    start_server_thread()

    # Start Morse code detection
    print("Starting detect_morse task")
    await detect_morse()

# Run the asyncio event loop
asyncio.run(main())
```

B. Web Interface

The accompanying web interface allows users to monitor the decoded Morse code and translated message in real-time. Below is the HTML code for the interface:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content=
    "width=device-width, initial-scale=1.0">
    <title>Morse Code Reader</title>
    <style>
        ...
    </style>
    <script>
        ...
    </script>
</head>
<body onload="updateOutputs()">
    ...
</body>
</html>
```