Assignment 4:
IOT Alarm System

Name: Genaro Salazar Ruiz

Course: WES 237

Date: February 21, 2026

Github: WES-237-Assignment-4

---

1. Introduction & Methodology

The Goal: Implementing a bi-directional alarm system using multiprocessing and TCP sockets to communicate between two PYNQ-Z2 boards.

Top-Down Desgin:

1. **Process Mapping:** Created two distinct processes: a Server to listen for incoming "BUZZ" triggers and a Client to monitor physical button presses
2. **Network Strategy:** Bind the server to '0.0.0.0' to ensure it listens on any available IP address, including my static IP address (**192.168.0.192**).
3. **Concurrency:** Utilized a shared Value flag (stop_signal) to allow one process to signal a full system shutdown to the other.

2. Technical Implementation

2.1 Synchronization and Reliability:

To combat socket hanging and "Address already in use" errors, we implemented several robust networking patterns:

1. **Error Handling:** Used ***try-except*** blocks around ***sock.bind*** and s***ock.connect*** to prevent the script from crashing during network instability.
2. **Non-Blocking Sockets:** Implemented ***sock.settimeout(1)*** within the server loop. This allows the server to periodically "wake up" and check if the ***stop_flag*** has been toggled, preventing zombie processes.
3. **Context Managers:** Used with ***socket.socket(...)*** to ensure that network resources are automatically cleaned up and closed when a process terminates.

2.2 Hardware Interfacing:

The primary technical breakthrough was identifying that the buzzers' center pins were non-functional, which allowed for a more efficient wiring strategy. By connecting the buzzers directly to **3.3V** and a **SIGNAL** pin, you were able to control the alarm using only a single GPIO pin per board.

Microblaze GPIO: Utilized the ***%%microblaze base.PMODA*** to handle low-level pin toggling.

Pin Configuration:

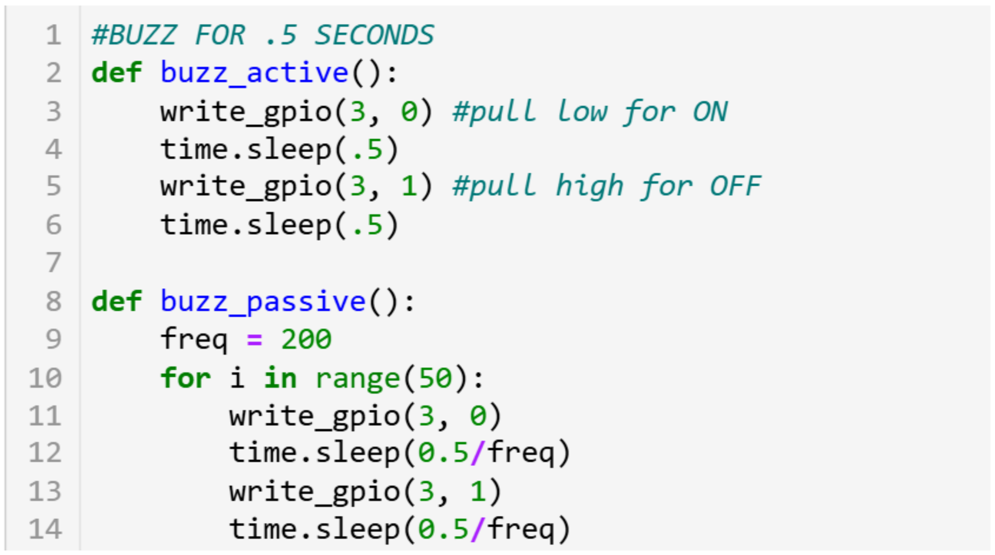
- **Pin 1:** 3.3 V
- **Pin 2:** This pin had no function
- **Pin 3:** Used as the Signal pin for the buzzer. We just flipped this between ON/OFF, depending on the buzzer type.

Tone Generation: The ***buzz()*** function pulls the signal pin low, sleeps, and then pulls it high to create the necessary oscillation for the sound.

We used both buzzers; Alex had Passive, and I had Active:

- **Active Buzzer (HW-512):** This module contains its own internal oscillator. We discovered that by pulling the signal pin LOW, the circuit completes, allowing current to flow and triggering a steady tone without needing a complex frequency loop.
- **Passive Buzzer (HW-508):** This module lacks an internal source and acts more like a speaker. Alex's implementation required a manual square wave, which rapidly toggled the signal pin between High and Low states to physically oscillate the transducer and create sound.

  Implementation Seen below:



```
#BUZZ FOR .5 SECONDS
def buzz_active():
    write_gpio(3, 0) #pull low for ON
    time.sleep(.5)
    write_gpio(3, 1) #pull high for OFF
    time.sleep(.5)

def buzz_passive():
    freq = 200
    for i in range(50):
        write_gpio(3, 0)
        time.sleep(0.5/freq)
        write_gpio(3, 1)
        time.sleep(0.5/freq)
```

3. Results and Analysis

3.1 Bi-Directional Communication

The system successfully used full-duplex communication:

- **PYNQ1 (Server):** Listened on port 12345.
- **PYNQ1 (Client):** Simultaneously monitored ***btns[1]*** to send a ***"BUZZ"*** command to PYNQ2.
- **Outcome:** Both boards could trigger each other's buzzers at the same time without interfering with their own listening capabilities.

3.2 Termination Logic:

By pressing ***btns[2]***, the client sends a ***"SHUTDOWN"*** string to the remote server. This triggers the remote ***stop_flag.value = True***, causing both the local and remote processes to exit their while loops and join back to the main thread.

4. Troubleshooting & Difficulties

- **Server Socket Timeouts:** Initially, the server would hang indefinitely while waiting for a connection, making it impossible to stop the script and adding sock.settimeout(1) solved this by allowing the loop to check the ***stop_flag*** every second.
- **Client Socket Timeouts:** If the client attempted to send a ***"BUZZ"*** or ***"SHUTDOWN"*** command to a server that had already disconnected or crashed, the ***sock.send()*** method could hang or throw an unhandled exception. By wrapping the connection and communication logic in try-except blocks, the client could catch ***ConnectionError*** or ***BrokenPipeError***.
- **Indentation and Blocks:** We encountered IndentationError when nesting the with conn: block. Correcting the block structure ensured the server could process multiple ***"BUZZ"*** commands over a single persistent connection.
- **Active vs Passive Buzzer:** While passive buzzers usually require a specific frequency, we found that simply toggling the pin provided a sufficient alarm sound for the hardware we had available (HW-512/HW-508).
- **IP Conflicts:** As documented in the logs, we had to ensure the host Mac and the PYNQ were on the same 192.168.0.x subnet. Once the Mac was switched from the 100.84.x.x campus network to the private router, the ConnectionRefusedError was resolved.
- **Buzz Sequency Iterations:** By testing out-of-order button sequences, we identified and resolved potential hang conditions using ***try-except*** blocks and socket existence checks to ensure system stability regardless of user input timing. Additionally, we implemented a 0.3-second debounce delay to prevent rapid-fire socket events from overwhelming the server listener during physical button presses.

---

5. Video Demonstration Link

Link

The video demonstrates both PYNQ’s: Connecting via BTN0, sending a remote buzz via BTN1, and a synchronized shutdown using BTN2.

---

5. Appendix: Jupyter Notebook

In [2]:
```
import socket
import time
from multiprocessing import Process, Value
from pynq.overlays.base import BaseOverlay

base = BaseOverlay("base.bit")
btns = base.btns_gpio
```

In [3]:
```
%%microblaze base.PMODA

#include "gpio.h"
#include "pyprintf.h"

//Function to turn on/off a selected pin of PMODB
unsigned int write_gpio(unsigned int pin, unsigned int val){
    if (val > 1){
        pyprintf("pin value must be 0 or 1");
    }
    gpio pin_out = gpio_open(pin);
    gpio_set_direction(pin_out, GPIO_OUT);
    gpio_write(pin_out, val);
    return 1;
}

//do not read here
//Function to read the value of a selected pin of PMODB
unsigned int read_gpio(unsigned int pin){
    gpio pin_in = gpio_open(pin);
    gpio_set_direction(pin_in, GPIO_IN);
    return gpio_read(pin_in);
}
```

In [22]:
```
#BUZZ FOR .5 SECONDS
def buzz():
    write_gpio(3, 0)
    time.sleep(.5)
    write_gpio(3, 1) #pull high

#--------------------------------------------------
# SERVER PROCESS
def run_server(stop_flag):
    #'0.0.0.0' listens on any open IP automatically
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        try:
            sock.bind(('0.0.0.0', 12345))
        except:
            sock.close
            sock.bind(('0.0.0.0', 12345))
        print ("listening...")
        sock.listen(1)
        while not stop_flag.value:
            try:
                sock.settimeout(1) #allow  to not hang on loop
                conn, addr = sock.accept()
                with conn:
                    print(f"Server: connected w/ {addr}")
                    while not stop_flag.value:
                        data = conn.recv(1024)
                        if not data:
                            break
                        if data.decode() == "BUZZ":
                            print("Server: Buzzing!")
                            buzz()
                        elif data.decode() == "SHUTDOWN":
                            print("Server: Shutting down!")
                            stop_flag.value = True

            except socket.timeout:
                continue
        sock.close()

#--------------------------------------------------
#CLIENT PROCESS
def run_client(target_ip, stop_flag):
    #BTN0 = CONNECT
    sock = None
    print("Client running. Button 0 to connect.")

    while not stop_flag.value:
        # BTN0 = connect
        if btns[0].read() == 1:
            try:
                sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                sock.connect((target_ip, 12345))
                print("client connected")
                time.sleep(0.3) # debounce
            except:
                if sock:
                    sock.close()
                print("client: connection failed.")
                time.sleep(0.3)

        # BTN1 = send buzz
        if btns[1].read() == 1 and sock:
            try:
                sock.send("BUZZ".encode())
                print("Buzz!")
            except:
                print("client: no server connected")
            time.sleep(0.3)

        # BTN2 = disconnect and stop
        if btns[2].read() == 1:
            print("shutting down...")
            try:
                if sock:
                    sock.send("SHUTDOWN".encode())
                    sock.close()
            except:
                pass
            stop_flag.value = True # end loop
            time.sleep(0.3)
```

In [ ]:

```
if __name__ == "__main__":
    write_gpio(3, 1) #start low

    OTHER_IP = '192.168.0.120' #IP INPUT HERE
    stop_signal = Value('b', False) # shared flag to kill processes

    p1 = Process(target=run_server, args=(stop_signal,))
    p2 = Process(target=run_client, args=(OTHER_IP, stop_signal))

    p1.start()
    p2.start()

    p1.join()
    print("server shutdown")


    p2.join()
    print("client shutdown")

    print("DONE")
```

```
listening...
Client running. Button 0 to connect.
Server: connected w/ ('192.168.0.120', 51214)
Server: connected w/ ('192.168.0.120', 51218)
Server: Buzzing!
Server: Buzzing!
Server: Buzzing!
Server: Buzzing!
client connected
Buzz!
Buzz!
Buzz!
Buzz!
Buzz!
Buzz!
Buzz!
```

In [ ]: