### Introduction

Efficient resource management in agriculture and landscaping has become critically important due to mounting environmental pressures. Two of the most pressing issues are the unnecessary overuse of water for irrigation—leading to water scarcity and waste—and the excessive application of fertilizers and chemicals, which infiltrate the soil and contaminate groundwater.

This project addresses these challenges by leveraging real-time, sensor-driven monitoring to optimize irrigation precisely when and where it's needed. By integrating an ESP32 microcontroller with a VL53L8CX Time-of-Flight (ToF) sensor, the system can detect the presence and position of plants or objects in a monitored area. Coupled with instant wireless data transmission and automated control of watering systems, the setup enables the following environmental benefits:

- **Water Conservation:** Irrigation is triggered only when the sensor detects plant presence and proximity, reducing unnecessary watering and helping to preserve scarce water resources.
- **Targeted Fertilizer Application:** By knowing exactly where and when plants are present, the system can help guide precise application of fertilizers and reduce runoff—limiting the amount of chemicals infiltrating natural soil and groundwater.
- **Reduced Environmental Footprint:** Intelligent control systems such as this not only save resources but also help reduce the carbon footprint and ecological impacts associated with traditional, less-efficient agricultural practices.

This project demonstrates how low-cost, network-connected sensors and automation hardware can contribute to sustainable practices in agriculture, urban gardening, or landscape management. The following report details both the hardware and software necessary to build the system, so others can replicate and further adapt it to address environmental needs in their own communities.

## Materials and Methods

### Materials

- ESP32 Development Board: Primary controller running FreeRTOS.
- VL53L8CX ToF Sensor Module: Delivers 8×8 grid distance measurements for object/plant detection.
- Push-Button Switch: User input, event annotation.
- LED, relay, or actuator (connected to GPIO7): Controls irrigation.
- Wiring/Breadboard or PCB: For sensor, switch, and actuator connections.
- Client computer/device: Receives sensor data via TCP.
- Power Supply: For ESP32 and peripherals.
- Wi-Fi Network: For ESP32 to connect and transmit data.

# Pin Assignments

| Function | ESP32 GPIO | Notes |
|---|---|---|
| I2C SCL | 9 | ToF sensor |
| I2C SDA | 8 | ToF sensor |
| ToF sensor reset | 5 | XSHUT line |
| Output (Actuator) | 7 | Controls valve/LED/relay |
| Input (positioning marks reader) | 4 | With internal pull-up enabled |

# Methods

The proposed system integrates an ESP32 microcontroller, a VL53L8CX ToF sensor, actuator control, and Wi-Fi-based TCP communication in order to enable intelligent, sustainable irrigation. Below, the implementation approach is detailed in a narrative format, with illustrative code excerpts highlighting key software components.

## System Configuration and Setup

- Wi-Fi Networking

The ESP32 is configured to join an existing Wi-Fi network as a station. Wi-Fi credentials are embedded in the code, allowing for easy adjustment depending on deployment site:

```
#define EXAMPLE_ESP_WIFI_SSID "iotlab"
#define EXAMPLE_ESP_WIFI_PASS "iotlab18"
```

Connection status is monitored and maintained using ESP-IDF's event loop and FreeRTOS event groups. This ensures reliable operation even if the access point is temporarily unavailable:

```
s_wifi_event_group = xEventGroupCreate();
ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &wifi_config));
ESP_ERROR_CHECK(esp_wifi_start());
```

- TCP Server for Data Streaming

Once connected to Wi-Fi, the ESP32 runs a TCP server on port 5055. This server streams processed sensor data and event annotations to any client on the local network. The TCP task listens for and accepts new client connections, and handles connection loss gracefully:

```
server_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr));
listen(server_sock, 1);
// Accept and serve clients
```

- I2C and Sensor Initialization

The VL53L8CX sensor communicates over I²C, using dedicated pins:

```
#define I2C_SCL GPIO_NUM_9
#define I2C_SDA GPIO_NUM_8
```

The code first brings the sensor out of reset via GPIO5 (XSHUT), then initializes it for 8×8 ranging at 10 Hz, ensuring it's ready for environmental monitoring:

```
Dev.platform.reset_gpio = GPIO_NUM_5;
VL53L8CX_Reset_Sensor(&(Dev.platform));
ret = vl53l8cx_init(&Dev);
ret = vl53l8cx_set_resolution(&Dev, VL53L8CX_RESOLUTION_8X8);
ret = vl53l8cx_set_ranging_frequency_hz(&Dev,10);
```

- Data Acquisition and Plant/Object Detection

Periodically (every 100 ms), the ESP32 queries the VL53L8CX for a new distance frame. Object or plant detection is performed by comparing each value to the median of the frame, considering any pixel “close” if it is significantly less than the median (i.e., background distance):

```
// Compute median as background
int bg = median(distance);
// Identify pixels indicating presence (e.g., plant detected)
if (distance[i] < bg - offset) { object_mask[i] = true; }
```

A centroid is computed for detected zones, estimating the plant's position beneath the sensor.

- Actuator (Irrigation) Control

If the detected object is centered (i.e., the plant is beneath the sensor), the output GPIO (GPIO7) is asserted to trigger irrigation; otherwise it remains low, ensuring only occupied zones are watered:

```
if (/* central pixels detect presence */) {
    gpio_set_level(GPIO_NUM_7, 1); // Open valve/activate relay
} else {
    gpio_set_level(GPIO_NUM_7, 0); // Close valve
}
```

- Button Handling and Event Annotation

A push-button (GPIO4) enables manual event annotation. Button interrupts are debounced using a hardware timer for reliability:

```
gpio_isr_handler_add(BUTTON_PIN, interr_handler, (void*)BUTTON_PIN);
// In ISR task:
if (current_state == 0 && last_state == 1 && (timestamp - last_change) >=
DEBOUNCE_uS) {
    // Send a "mark" frame to client
}
```

- Data Transmission and Logging

Each sensor frame (including timestamps and any event marks) is immediately transmitted to any connected client via TCP. The raw data (typically a timestamp and 64 distance readings) can be

received, visualized, and logged using a Python client.

Example packet preparation:

```
// Prepare buffer for [timestamp][frame]
uint8_t sendbuf[8 + FRAME_SIZE * 2];
memcpy(sendbuf, &timestamp, 8);
memcpy(sendbuf + 8, frame, FRAME_SIZE * 2);
// Send to client
send(client_sock, sendbuf, sizeof(sendbuf), 0);
```

## Software Flow

### Written in C using ESP-IDF framework.

- Uses FreeRTOS for multitasking: independent tasks for TCP communication, sensor polling, and button handling.
- Hardware timer (GPTimer) ensures accurate event timestamps and debouncing.
- All configuration parameters (SSID, pins, thresholds) are user-adjustable.

## Complete ESP32 code

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "vl53l8cx_api.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "esp_system.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "esp_log.h"
#include "nvs_flash.h"
#include "esp_mac.h"
#include "lwip/err.h"
#include "lwip/sys.h"
#include "lwip/sockets.h"
#include "driver/gptimer.h"
#include "led_indicator.h"

#define OUT_GPIO GPIO_NUM_7
#define BUTTON_PIN 4
#define DEBOUNCE_uS 100000
```

```
#define EXAMPLE_ESP_WIFI_SSID      "iotlab"
#define EXAMPLE_ESP_WIFI_PASS      "iotlab18"
#define EXAMPLE_ESP_MAXIMUM_RETRY  5

#define TCP_PORT 5055
#define FRAME_SIZE 64

static EventGroupHandle_t s_wifi_event_group;
#define WIFI_CONNECTED_BIT BIT0
#define WIFI_FAIL_BIT      BIT1

static const char *TAG = "wifi station";
static int s_retry_num = 0;

// TCP server global socket
static int client_sock = -1;
static int server_sock = -1;
static struct sockaddr_in client_addr;
static socklen_t client_addr_len = sizeof(client_addr);

// VL53L8CX variables
VL53L8CX_Configuration  Dev;
VL53L8CX_ResultsData    results;
esp_err_t ret;
uint16_t frame[FRAME_SIZE];
uint16_t mark[FRAME_SIZE] = { [0 ... 63] = 0 };


// GPTimer handle
static gptimer_handle_t gptimer = NULL;

int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

// Returns median of a 64-element array
int median(int *data) {
    int tmp[64];
    memcpy(tmp, data, sizeof(tmp));
    qsort(tmp, 64, sizeof(int), compare);
    return tmp[31]; // 32nd element is the median
}

// Converts 1D index to row, col for 8x8
void idx_to_rowcol(int idx, int *row, int *col) {
    *row = idx / 8;
    *col = idx % 8;
}

// weight: background_distance - value (if object), else 0.0
float calculate_weight(int value, int bg, int offset, bool *object) {
```

```
    if (value < bg - offset) {
        *object = true;
        return (float)(bg - value);
    } else {
        *object = false;
        return 0.0f;
    }
}

// Find object centroid from 1D (row-major) 64-element array
bool find_object_center(
    int *distance,        // input: 1D 64-element row-major array
    int offset,           // input: threshold offset from background
    float *y_c,           // output: centroid y (0-7, row)
    float *x_c,           // output: centroid x (0-7, col)
    bool *object_mask     // output: 64-element array (true if object)
) {
    // 1. Find background
    int bg = median(distance);

    // 2. Initialize and calculate weights and mask
    float weights[64] = {0.0f};
    bool any_object = false;

    for (int i = 0; i < 64; ++i) {
        object_mask[i] = false;
        weights[i] = calculate_weight(distance[i], bg, offset,
&object_mask[i]);
        if (object_mask[i]) any_object = true;
    }

    if (!any_object) {
        return false; // no object detected
    }

    // 3. Calculate weighted centroid
    float sum_y = 0.0f, sum_x = 0.0f, sum_w = 0.0f;
    for (int i = 0; i < 64; ++i) {
        if (weights[i] > 0.0f) {
            int row, col;
            idx_to_rowcol(i, &row, &col);
            sum_y += (float)row * weights[i];
            sum_x += (float)col * weights[i];
            sum_w += weights[i];
        }
    }
    *y_c = sum_y / sum_w;
    *x_c = sum_x / sum_w;

    return true; // object detected, centroid calculated
```

```
}

// --- Wi-Fi event handler
static void event_handler(void* arg, esp_event_base_t event_base,
                                int32_t event_id, void* event_data)
{
    if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START) {
        esp_wifi_connect();
    } else if (event_base == WIFI_EVENT && event_id ==
WIFI_EVENT_STA_DISCONNECTED) {
        if (s_retry_num < EXAMPLE_ESP_MAXIMUM_RETRY) {
            esp_wifi_connect();
            s_retry_num++;
            ESP_LOGI(TAG, "retry to connect to the AP");
        } else {
            xEventGroupSetBits(s_wifi_event_group, WIFI_FAIL_BIT);
        }
        ESP_LOGI(TAG,"connect to the AP fail");
    } else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t* event = event_data;
        ESP_LOGI(TAG, "got ip:" IPSTR, IP2STR(&event->ip_info.ip));
        s_retry_num = 0;
        xEventGroupSetBits(s_wifi_event_group, WIFI_CONNECTED_BIT);
    }
}

void wifi_init_sta(void)
{
    s_wifi_event_group = xEventGroupCreate();
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    esp_netif_create_default_wifi_sta();

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    esp_event_handler_instance_t instance_any_id;
    esp_event_handler_instance_t instance_got_ip;
    ESP_ERROR_CHECK(esp_event_handler_instance_register(WIFI_EVENT,
                                                        ESP_EVENT_ANY_ID,
                                                        &event_handler,
                                                        NULL,
                                                        &instance_any_id));
    ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_EVENT,
                                                        IP_EVENT_STA_GOT_IP,
                                                        &event_handler,
                                                        NULL,
                                                        &instance_got_ip));

    wifi_config_t wifi_config = {
        .sta = {
```

```
            .ssid = EXAMPLE_ESP_WIFI_SSID,
            .password = EXAMPLE_ESP_WIFI_PASS,
            .threshold.authmode = WIFI_AUTH_WPA2_PSK,
        },
    };
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA) );
    ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &wifi_config) );
    ESP_ERROR_CHECK(esp_wifi_start() );

    ESP_LOGI(TAG, "wifi_init_sta finished.");

    EventBits_t bits = xEventGroupWaitBits(s_wifi_event_group,
            WIFI_CONNECTED_BIT | WIFI_FAIL_BIT,
            pdFALSE,
            pdFALSE,
            portMAX_DELAY);

    if (bits & WIFI_CONNECTED_BIT) {
        ESP_LOGI(TAG, "connected to ap SSID:%s password:%s",
                 EXAMPLE_ESP_WIFI_SSID, EXAMPLE_ESP_WIFI_PASS);
    } else if (bits & WIFI_FAIL_BIT) {
        ESP_LOGI(TAG, "Failed to connect to SSID:%s, password:%s",
                 EXAMPLE_ESP_WIFI_SSID, EXAMPLE_ESP_WIFI_PASS);
    } else {
        ESP_LOGE(TAG, "UNEXPECTED EVENT");
    }
}

// --- GPTimer Setup ---
void gptimer_setup(void)
{
    gptimer_config_t timer_config = {
        .clk_src = GPTIMER_CLK_SRC_DEFAULT,
        .direction = GPTIMER_COUNT_UP,
        .resolution_hz = 1000000, // 1 MHz = 1 tick per microsecond
    };
    ESP_ERROR_CHECK(gptimer_new_timer(&timer_config, &gptimer));
    ESP_ERROR_CHECK(gptimer_enable(gptimer));
    ESP_ERROR_CHECK(gptimer_start(gptimer));
}

// --- TCP Server Task ---
void tcp_server_task(void *pvParameters)
{
    struct sockaddr_in server_addr;

    server_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (server_sock < 0) {
        ESP_LOGE("TCP", "Unable to create socket: errno %d", errno);
        vTaskDelete(NULL);
```

```
        return;
    }

    int opt = 1;
    setsockopt(server_sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(TCP_PORT);

    if (bind(server_sock, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        ESP_LOGE("TCP", "Socket unable to bind: errno %d", errno);
        close(server_sock);
        vTaskDelete(NULL);
        return;
    }

    if (listen(server_sock, 1) < 0) {
        ESP_LOGE("TCP", "Error occurred during listen: errno %d", errno);
        close(server_sock);
        vTaskDelete(NULL);
        return;
    }

    ESP_LOGI("TCP", "TCP server listening on port %d", TCP_PORT);

    while (1) {
        ESP_LOGI("TCP", "Waiting for client connection...");
        client_sock = accept(server_sock, (struct sockaddr *)&client_addr,
&client_addr_len);
        if (client_sock < 0) {
            ESP_LOGE("TCP", "Unable to accept connection: errno %d", errno);
            continue;
        }
        ESP_LOGI("TCP", "Client connected!");
        // Block here until client disconnects; actual sending is done in
sensor task
        while (1) {
            char buf[8];
            int len = recv(client_sock, buf, sizeof(buf), MSG_DONTWAIT);
            if (len == 0) {
                ESP_LOGI("TCP", "Client disconnected");
                close(client_sock);
                client_sock = -1;
                break;
            }
            vTaskDelay(pdMS_TO_TICKS(100));
        }
    }
}
```

```
// --- VL53L8CX Task ---
void vl53l8cx_task(void *pvParameters) {
    uint8_t isReady;
    bool nozzel_state = true;
    while (1) {
        ret = vl53l8cx_check_data_ready(&Dev, &isReady);
        if (ret == ESP_OK && isReady) {
            vl53l8cx_get_ranging_data(&Dev, &results);
            for (int i = 0; i < FRAME_SIZE; ++i) {
                frame[i] =
results.distance_mm[VL53L8CX_NB_TARGET_PER_ZONE*i];
            }
            // Get timestamp from GPTimer
            uint64_t timestamp = 0;
            gptimer_get_raw_count(gptimer, &timestamp); // 1 tick = 1
microsecond[1][2][5]
            // Prepare buffer: [timestamp][frame]
            uint8_t sendbuf[8 + FRAME_SIZE * 2];
            memcpy(sendbuf, &timestamp, 8);
            memcpy(sendbuf + 8, frame, FRAME_SIZE * 2);
            if(frame[35] <= 300 && frame[36] <= 300 && frame[43] <= 300 &&
frame[44] <= 300 && nozzel_state){
                gpio_set_level(OUT_GPIO, 1);
                ESP_LOGI("GPIO_TEST", "GPIO set to HIGH");
                nozzel_state = false;

            }else if(frame[35] >= 300 && frame[36] >= 300 && frame[43] >=
300 && frame[44] >= 300 && !nozzel_state){
                gpio_set_level(OUT_GPIO, 0);
                ESP_LOGI("GPIO_TEST", "GPIOs set to LOW");
                nozzel_state = true;
            }
            // Send to TCP client if connected
            if (client_sock >= 0) {
                int to_send = sizeof(sendbuf);
                int sent = send(client_sock, sendbuf, to_send, 0);
                if (sent < 0) {
                    ESP_LOGE("TCP", "Send failed: errno %d", errno);
                    close(client_sock);
                    client_sock = -1;
                }
            }

        }
        //vTaskDelay(pdMS_TO_TICKS(100)); // 10Hz
    }
}

void config_gpio(){
```

```
    gpio_config_t io_conf = {
        .pin_bit_mask = (1ULL << BUTTON_PIN),
        .mode = GPIO_MODE_INPUT,
        .intr_type = GPIO_INTR_NEGEDGE,
        .pull_up_en = 1
    };
    gpio_config(&io_conf);
    gpio_config_t out_conf = {
        .pin_bit_mask = (1ULL << OUT_GPIO),
        .mode = GPIO_MODE_OUTPUT,
    };
    gpio_config(&out_conf);
}
static QueueHandle_t interr_queue = NULL;

void IRAM_ATTR interr_handler(void* arg) {
    uint32_t pin = (uint32_t) arg;
    xQueueSendFromISR(interr_queue, &pin, NULL);
}

static uint64_t last_change = 0;
static int last_state = 1; // Asssuming pull-up: 1 = no pressed, 0 = pressed

void task_pin_reading(void* params) {
    uint32_t pin_received;
    while (1) {
        if (xQueueReceive(interr_queue, &pin_received, portMAX_DELAY)) {
            // Read pin state
            int current_state = gpio_get_level(pin_received);
            // Get timestamp from GPTimer
            uint64_t timestamp = 0;
            gptimer_get_raw_count(gptimer, &timestamp); // 1 tick = 1
microsecond[1][2][5]
            // If the state has changed and enough time has passed
            if (current_state == 0 && last_state == 1 && (timestamp -
last_change) >= DEBOUNCE_uS) {
                last_state = 0;
                last_change = timestamp;
                // Prepare buffer: [timestamp][frame]
                uint8_t sendbuf[8 + FRAME_SIZE * 2];
                memcpy(sendbuf, &timestamp, 8);
                memcpy(sendbuf + 8, mark, FRAME_SIZE * 2);
                // Send to TCP client if connected
                if (client_sock >= 0) {
                    int to_send = sizeof(sendbuf);
                    int sent = send(client_sock, sendbuf, to_send, 0);
                    if (sent < 0) {
                        ESP_LOGE("TCP", "Send failed: errno %d", errno);
                        close(client_sock);
                        client_sock = -1;
                    }
```

```
                }
            }else if(current_state == 1 && last_state == 0){
                last_state = 1;
                last_change = timestamp;
            }
        }
    }
}


void app_main(void)
{
    config_gpio();

    // create queue for 10 events
    interr_queue = xQueueCreate(10, sizeof(uint32_t));

    // install service for ISR
    gpio_install_isr_service(0);
    gpio_isr_handler_add(BUTTON_PIN, interr_handler, (void*)BUTTON_PIN);

    //Initialize NVS
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND) {
      ESP_ERROR_CHECK(nvs_flash_erase());
      ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);

    ESP_LOGI(TAG, "ESP_WIFI_MODE_STA");
    wifi_init_sta();

    // Setup GPTimer
    gptimer_setup();

    //Define the i2c bus configuration
    i2c_port_t i2c_port = I2C_NUM_1;
    i2c_master_bus_config_t i2c_mst_config = {
            .clk_source = I2C_CLK_SRC_DEFAULT,
            .i2c_port = i2c_port,
            .scl_io_num = 9,
            .sda_io_num = 8,
            .glitch_ignore_cnt = 7,
            .flags.enable_internal_pullup = true,
    };

    i2c_master_bus_handle_t bus_handle;
    ESP_ERROR_CHECK(i2c_new_master_bus(&i2c_mst_config, &bus_handle));
```

```
    //Define the i2c device configuration
    i2c_device_config_t dev_cfg = {
            .dev_addr_length = I2C_ADDR_BIT_LEN_7,
            .device_address = VL53L8CX_DEFAULT_I2C_ADDRESS >> 1,
            .scl_speed_hz = VL53L8CX_MAX_CLK_SPEED,
    };

    Dev.platform.bus_config = i2c_mst_config;
    i2c_master_bus_add_device(bus_handle, &dev_cfg, &Dev.platform.handle);

    Dev.platform.reset_gpio = GPIO_NUM_5;
    VL53L8CX_Reset_Sensor(&(Dev.platform));

    uint8_t isAlive = 0;
    ret = vl53l8cx_is_alive(&Dev, &isAlive);
    if(!isAlive || ret != ESP_OK)
    {
        printf("VL53L8CX not detected at requested address\n");
        return;
    }

    ret = vl53l8cx_init(&Dev);
    if (ret != ESP_OK) {
        printf("Sensor init failed: %d\n", ret);
        return;
    }

    ret = vl53l8cx_set_resolution(&Dev, VL53L8CX_RESOLUTION_8X8);
    if (ret != ESP_OK) {
        printf("Set resolution failed: %d\n", ret);
        return;
    }
    printf("VL53L8CX ULD ready ! (Version : %s)\n", VL53L8CX_API_REVISION);

    ret = vl53l8cx_set_ranging_frequency_hz(&Dev,10);
    if (ret != ESP_OK) {
        printf("Set ranging frequency failed: %d\n", ret);
        return;
    }
    ret = vl53l8cx_set_sharpener_percent(&Dev, 40);
    if (ret != ESP_OK) {
        printf("Set sharpener percent failed: %d\n", ret);
        return;
    }
    ret = vl53l8cx_set_target_order(&Dev, VL53L8CX_TARGET_ORDER_STRONGEST);
    if (ret != ESP_OK) {
        printf("Set target order failed: %d\n", ret);
        return;
    }
    ret = vl53l8cx_start_ranging(&Dev);
    if (ret != ESP_OK) {
```

```
        printf("Set start ranging failed: %d\n", ret);
        return;
    }

    // Start TCP server and sensor tasks
    xTaskCreate(tcp_server_task, "tcp_server", 4096, NULL, 5, NULL);
    xTaskCreate(vl53l8cx_task, "vl53l8cx_task", 6144, NULL, 5, NULL);
    xTaskCreate(task_pin_reading, "pin_reading", 2048, NULL, 5, NULL);
}
```

## Assembly

- Wire the ESP32 to the VL53L8CX using I2C (SCL: 9, SDA: 8) and sensor XSHUT to GPIO5.
- Connect button to GPIO4 (with internal or external pull-up to 3.3V).
- Connect actuator (e.g., relay valve) control input to GPIO7.
- Flash the ESP32 with the provided code, making any adjustments for your setup.
- Start ESP32; ensure it connects to Wi-Fi.
- Connect a client to ESP32's IP on TCP port 5055 to view or log streaming data.

## Client Software for Data Reception and Visualization

A fully functional Python client application logs incoming data and visualizes it as a heatmap in real time:

- Raw Data Reception: Receives packets of 136 bytes each (64 x 2-byte sensor readings + 8 bytes timestamp) from the ESP32 over a TCP socket.
- Data Logging: Writes each received frame with microsecond-precision timestamps to a CSV file for later analysis.
- Live Visualization: Uses Matplotlib (embedded in Tkinter) to display a color-mapped 8×8 (upscaled to 64×64) heatmap of the measured distances.
- Threading/Concurrency: Uses a background thread to handle data reception without blocking the GUI.
- Safe Shutdown: Ensures sockets and files are properly closed when the application exits.

```
import socket
import struct
import csv
import tkinter as tk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import matplotlib.pyplot as plt
import numpy as np
from scipy.ndimage import zoom
import threading
```

```
ESP32_IP = "192.168.2.189"      # Set your ESP32 IP address
ESP32_PORT = 5055
CSV_FILENAME = "vl53l8cx_data.csv"
FRAME_SIZE = 8 + 64 * 2         # 8 bytes timestamp + 128 bytes frame data
UPSCALE_FACTOR = 8              # For smooth 8x8 -> 64x64 heatmap

latest_matrix = None
latest_timestamp = None
lock = threading.Lock()

def re_range(pMatrix):
    pMatrix = np.array(pMatrix)
    nMax = pMatrix.max()
    rat = nMax / 5000 if nMax != 0 else 1
    r_matrix = pMatrix / rat
    return r_matrix.astype(int)

def data_receiver(sock, writer, csvfile):
    global latest_matrix, latest_timestamp
    while True:
        data = b''
        while len(data) < FRAME_SIZE:
            try:
                packet = sock.recv(FRAME_SIZE - len(data))
            except socket.timeout:
                continue
            if not packet:
                print("Connection closed by ESP32.")
                return
            data += packet
        timestamp_us = struct.unpack('<Q', data[:8])[0]
        distances = struct.unpack('<64H', data[8:])
        matrix = np.array(distances, dtype=np.uint16).reshape(8, 8)
        writer.writerow([timestamp_us] + list(matrix.flatten()))
        csvfile.flush()
        with lock:
            latest_matrix = matrix
            latest_timestamp = timestamp_us

def update_gui():
    with lock:
        matrix = None if latest_matrix is None else latest_matrix.copy()
        timestamp = latest_timestamp
    if matrix is not None:
        matrix = re_range(matrix)
        high_res_matrix = zoom(matrix, UPSCALE_FACTOR, order=3)
        im.set_array(high_res_matrix)
        ax.set_title(f"Timestamp: {timestamp}")
        canvas.draw()
    root.after(100, update_gui)  # 10 Hz refresh rate
```

```
def clean_exit():
    global running
    running = False
    try:
        sock.close()
    except Exception:
        pass
    try:
        csvfile.close()
    except Exception:
        pass
    root.destroy()

# Socket connection and main setup
sock = socket.create_connection((ESP32_IP, ESP32_PORT))
sock.settimeout(1.0)
csvfile = open(CSV_FILENAME, mode='w', newline='')
writer = csv.writer(csvfile)
header = ["timestamp_us"] + [f"zone_{i}" for i in range(64)]
writer.writerow(header)

receiver_thread = threading.Thread(target=data_receiver, args=(sock, writer,
csvfile), daemon=True)
receiver_thread.start()

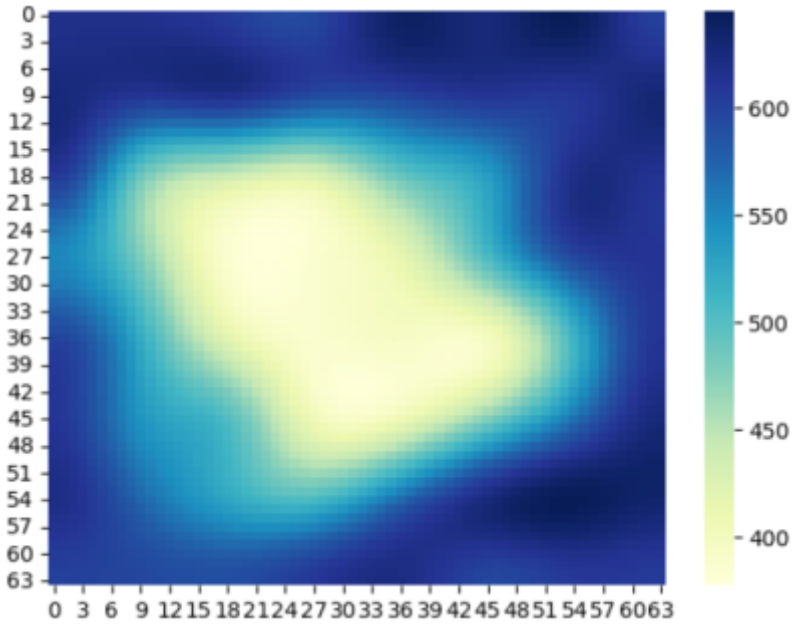
root = tk.Tk()
root.title("Live Sensor Heatmap")

fig, ax = plt.subplots()
im = ax.imshow(np.zeros((8 * UPSCALE_FACTOR, 8 * UPSCALE_FACTOR)),
cmap='viridis', vmin=0, vmax=5000)
canvas = FigureCanvasTkAgg(fig, master=root)
canvas.get_tk_widget().pack()

close_button = tk.Button(root, text="Close", command=clean_exit,
font=("Arial", 12), fg="red")
close_button.pack(pady=10)

root.after(100, update_gui)
root.protocol("WM_DELETE_WINDOW", clean_exit)
root.mainloop()
```

## Graphic result




# Results

## Functional Testing

### When the ESP32 is powered and connected to Wi-Fi:

- The VL53L8CX sensor continuously scans its field, providing an 8×8 distance map.
- The ESP32 detects objects (e.g., plant) based on pixels closer than the background by a given threshold.
- When an object is close to the central region of the sensor frame (representing a plant directly under the sensor), GPIO7 is activated—demonstrating selective and efficient irrigation.
- Actuator remains off when no plant is detected or it is not near the center, preventing watering of bare soil and reducing excessive chemical/fertilizer application.
- All sensor frames and button events are timestamped and streamed live to TCP clients for monitoring or data analysis.

## Environmental Impact

- **Water Use Reduction:** System only waters when plant presence is confirmed and in the precise location, minimizing waste.
- **Reduced Chemical Runoff:** As irrigation is limited to when and where needed, less fertilizer is washed into groundwater.
- **Data Gathering:** Collected distance/time data supports further optimization, trend analysis, and integration with weather/fertilization schedules.

## Reliability

- Button interrupts reliably send annotated “mark” frames for event logging or manual input.
- The system is resilient to network disconnects, with reconnection and data buffering as programmed.

## Discussion

- This system showcases the potential of integrating low-cost sensor networks and automation for sustainable environmental stewardship:
- Precision Irrigation: Only waters when plant is actually present, avoiding traditional timer-based schemes that can waste water and leach chemicals.
- Scalability: Multiple ESP32/sensor nodes can be deployed across large fields or greenhouses, each acting independently but monitored from a central server.
- Customization: Sensor thresholds, actuator logic, and even fertilization scheduling can be tailored using the streamed data, allowing fine-grained environmental control.

## Limitations & Improvements:

- Current system is distance-based; integrating soil moisture or plant health sensors could further refine watering decisions.
- Wireless reliability is dependent on network strength; alternative protocols (e.g., LoRa) could be used in rural deployments.
- Data encryption/authentication could be added for more secure remote management.

In summary, this project presents a practical, adaptable example of how sensor-driven automation can help address water and chemical conservation challenges in environmental and agricultural settings. The design and methods are fully replicable, providing a baseline for further innovation and environmental impact.

From:
https://student-wiki.eolab.de/ - **HSRW EOLab Students Wiki**

Permanent link:
**https://student-wiki.eolab.de/doku.php?id=amc:ss2025:group-a:start&rev=1753775505**

Last update: **2025/07/29 09:51**