**SOFT564z**

**Summary**

The purpose of this project was to create a distributed solution for the remote monitoring of hydroponic plants, or other water-level dependent systems. In order to achieve this, the system was designed to be robust and user-friendly. This report will document the Design, Implementation, and Testing, then evaluate the success of the Final product. The source code, and instructions for implementation are available in this [git repository](https://github.com/noblegasses/SOFT564z.git).

**Design**

The goal of this project is to produce a teleoperated robot for application of remote monitoring of hydroponic plants. It is equipped with an ultrasonic sensor for obstacle avoidance, and a water level sensor for water level monitoring.

This uses an Arduino Mega as a control platform. This device controls the motors of the buggy, and Servo arm for the water sensor. Additionally, the Mega measures and processes the data from the sensors. The distance sensor is used for measuring the distance from the target for water level measurement, and the water level sensor is used to detect the water level of the target, to detect if the system needs to be filled or emptied.

An ESP32 is used as the communication hub for the system, either connecting to a specific WiFi network, or if that is unavailable, hosting it’s own, short range access point. The ESP32 requests sensor data from the arduino and transmits it to the user’s PC, as well as receiving movement commands from the PC and transmitting it to the arduino. The ESP also has a set of LED’s that is used to communicate the status of the system (see fig 1).

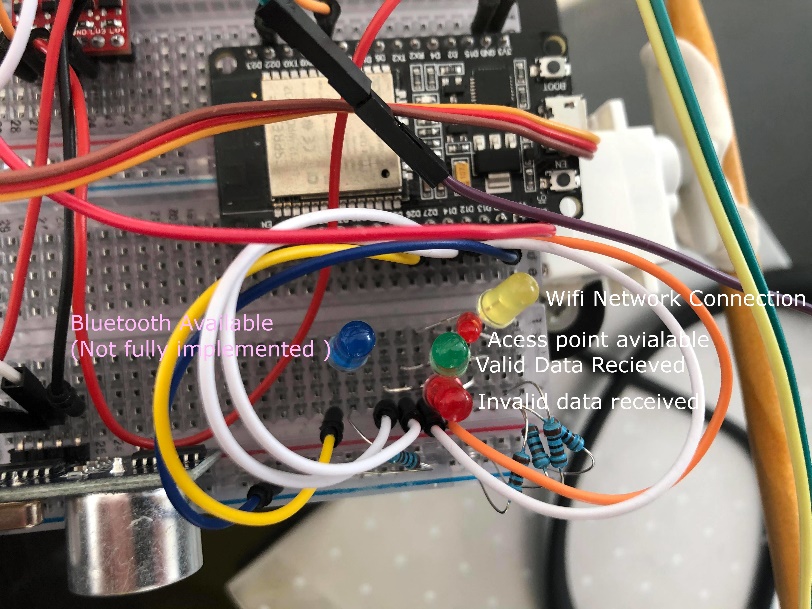


Figure 1. Image of ESP32 Status LED’s and their purposes

The PC hosts a Graphical user interface that displays the read in data, as well as the servo’s current position and the controls for the robot. By pressing the WASD key, the user can control the robot’s movement, and by pressing V and F, lower and raise the servo arm for the water level sensor.



Figure 2. High level system overview

**Implementation**

This section will explain how data is processed and handled for each part of the system. This will overview each of the systems and communication protocols for each device from a low level standpoint. Starting with the Arduino, followed by the ESP32 and I2C communication, and finally the PC client and TCP communication it uses.

The Arduino Mega board is responsible for the measurement of sensors, and the movement of the motors of the device. The Arduino is connected to the HC-SR04 Ultrasonic distance sensor via the echo pin on GPIO digital 6, trigger pin on GPIO digital 5. The trigger pin is used to start a sample on the ultrasonic sensor, and the echo pin is used to read the data from that sample into the Arduino. Once the sample is read in as a PWM signal, it has to be translated from a travel time from and to the sensor into a distance from the sensor to the object. This is achieved by multiplying the travel time by the speed of sound, scaled to us, and dividing the result by 2 to have only the distance to the device. The water level sensor gives an analog reading, and is connected via the analog 1 pin. The data read in is not very useful. The Arduino map function allows the data to be scaled from the upper and lower limits to the length of the water sensor, giving a depth in millimeters.

The motors are controlled by the Arduino as well. The Arduino moves both the motor gearboxes for the buggy, and the servo that is used to raise and lower the water level sensor. The data for motor movement is stored in a single char array called MotorMove. The first character of the array expected as one of either ‘F’ (forwards), ‘B’ (Backwards), ‘R’ (Right), ‘L’ (left), or ‘S’ (stop). If an unexpected character is read in, the system will stop moving otherwise the motors will be set to begin movement in the desired direction. The final 3 characters of the array are numbers that give a servo position. These characters are converted into an integer value and move the servo to the given position using the [Servo.h library](https://www.arduino.cc/reference/en/libraries/servo/).

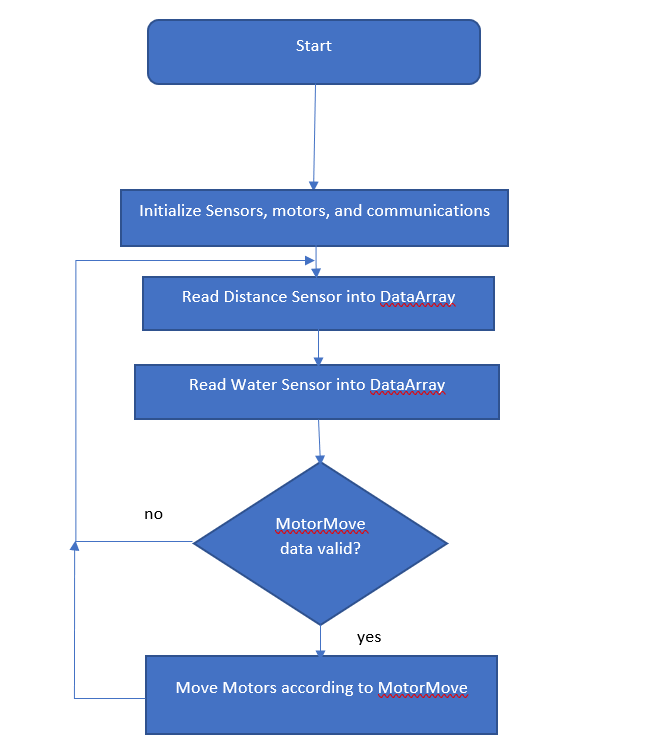


Figure 3. Arduino non-interrupt main loop

The Arduino is constantly sampling the two sensors and storing them in a volatile dataArray to be used for communication as needed. Since the I2C slave code is entirely interrupt driven, the Arduino can update data and move the motors until ready. I2C is implemented through the [Arduino Wire.h](https://www.arduino.cc/en/reference/wire) library, and although not used currently, the [LiquidCrystal\_I2C.h library](https://www.arduino.cc/reference/en/libraries/liquidcrystal-i2c/) is available to write messages and errors to the LCD display. The Arduino I2C library offers 2 interrupts for ISR’s to be attached to for an I2C slave. The onRequest interrupt fires if the master device requests a number of bits from the slave. The ISR I have implemented for the onRequest interrupt sends the contents of the dataArray array in bytes. Since the mega stores integers in 2 bytes, each integer must be split up in order to send the data over I2C. The onReceive interrupt is triggered when the master has data to send to the slave device, it passes an argument to the ISR, this argument contains the number of bytes being sent over. The ISR I have implemented reads in 4 bytes from the master device and stores them as chars in the MotoMove array.

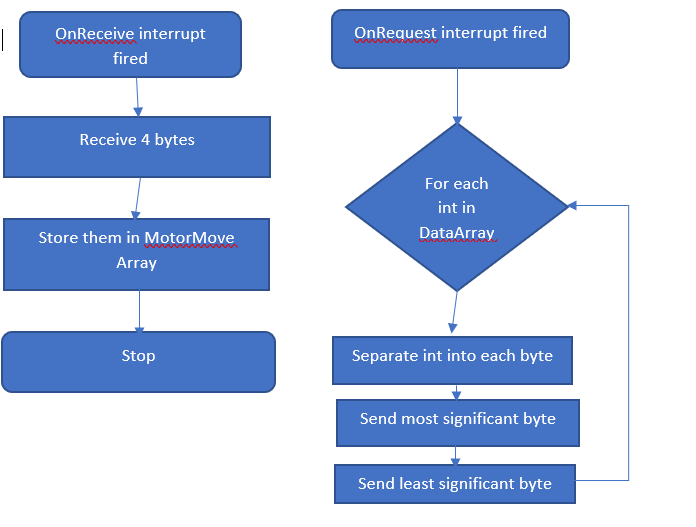


Figure 4. Flow Chart of I2C ISRs

The ESP32 is the networking hub for the system, it hosts a TCP server to communicate with a PC, and acts as an I2C master to communicate to the Arduino. On setup, the ESP32 attempt to connect to a Wifi network. it attempts this 5 times, if this fails, it hosts its own access point called espWifi, with the password “ShortRange”. This is intended as a short-range work around if the intended network is not available. The type of connection is displayed via LED. All WiFi connection and communication is done via the [Wifi.h library​​​](https://www.arduino.cc/en/Reference/). Regardless of what network the ESP32 connects to, it pulls the static internal IP 192.168.1.85, and opens a TCP socket on the port 49153. This port was selected as it is not a reserved port and will likely not interfere on a busy network.

The TCP communication is the main loop for the ESP32. Once setup is complete, the code loops, waiting for a client to connect to the open socket. Once a client has connected, the server starts polling for incoming data and lights the green LED to show a client is connected. If data starts incoming, it checks each incoming byte for one of the valid Movement start characters. Once a valid byte is read, it is stored in the movement data array moveArray each of the next incoming bytes is checked if it is a value between 0 and 9. If not the bad data LED is lit until valid data is received. If valid data is received, the new data is sent via I2C to the Arduino (configured at address 5). As with the Arduino[, the wire.h library](https://www.arduino.cc/en/reference/wire) is used for handling I2C communications for the ESP32. Because of this, the ESP32 could not be configured as a I2C slave, as the ESP32 version of the wire library does not support I2C slave configurations. Whilst the system is checking for incoming data it also requests sensor data from the Arduino. The Arduino sends integer in individual bytes, which need to be shifted back in. Once the data is updated from the Arduino, the ESP32 sends the data to the TCP client. The data is packaged as two integers, each surrounded by a set of Start and stop characters “S” and “E” to ensure that the client does not read the data in incorrectly. If the client disconnects, the Arduino is immediately sent a stop command, and the client connected light shuts off. See below for pseudocode of the ESP32 operation.

Setup{

Setup\_I2C

connect to wifi

Setup\_TCP\_server

}

Loop forever{

if client is connected{

if data from client{

Read data

validate data

send movement data to buggy

}

request sensor data

receive sensor data from buggy

send sensor data to client

}

stop buggy

}

The TCP client for the PC is written in python, and uses the following 3rd party libraries: [Pygame](https://www.pygame.org/news) to create the Graphical User Interface (GUI), [OS](https://docs.python.org/3/library/os.html) to access the filesystem of the computer, [time](https://docs.python.org/3/library/time.html) to add delays and timing, [Socket](https://docs.python.org/3/library/socket.html) to handle TCP communication, [copy](https://docs.python.org/3/library/copy.html) to compare different versions of variables, and [collections](https://docs.python.org/3/library/collections.html) to access the default dictionary structure.

The program is split into two parts, the GUI, offers an easy way to display controls and sensor data to the user, and a TCP back end that transmits and receives data from the robot. The program starts by loading in all of the needed assets, and opening the window the GUI will be hosted on. It then attempts to establish a connection to the TPC server at the static IP and port that the robot should be transmitting to. If this connection fails, the program will attempt to reconnect indefinitely.

Once a connection is achieved, the main user interface begins, displaying the water level and obstacle distance, both in millimeters in the top corners of the UI, as well as displaying the available keys to press and the current servo position and its controls. In the main loop of the program, system continuously polls, first to see if a key has been pressed, then to redraw the UI with any new data. Every 50ms, the system will check the TCP server to see if new data is available, if any new data is available, it is read in, checking for the start and stop characters, noted in the ESP32 section to delineate each entry, the UI is updated with the new data. If any new movement data has been generated, that is sent to the TCP server as the same 4-byte char array.

If a key has been pressed, the UI captures which key was pressed if the key is W, A, S, D the movement array is updated with the relevant, ‘F’, ‘B’, ‘L’, ‘R’ character, and if none of those keys are pressed, the movement array is updated with ‘S’. V and F are keys to decrement and increment the servo positions. If these keys are pressed, the internal value is updated, unless the increase would bring the value above 255 or the decrease would bring the servo position value below 0. The servo position value is translated into a character array, and the data is sent to the ESP via TCP.

If the TCP connection is dropped, the UI will immediately close any existing connections and attempt to reconnect. See pseudocode:

Setup{

connect to TCP server

Initalize UI

}

loop forever{

if keys pressed:

if WASD{

update cart movement

}

else{

Stop cart

}

if VF{

update servo position

}

update onscreen data

every 50 ms{

if data from TCP{

read in Data to data array

}

if new movement data{

send data to tcp

}

}

}

**Testing**

The primary form of testing implemented was Blackbox testing, by using programs that sent specified data over communications protocols to see if the expected output was achieved, many flaws or errors were detected. These tests can be seen in the testing folder of the git repository.

Other tests were implemented to check for functionality in different conditions. By switching off the router, the access point functionality could be tested. Each available sensor was tested, and these tests can be seen in the Sensor tests folder.

Some testing involved simply running the code with different inputs and accounting for bad data, etc. with the application of LED’s and the Serial monitor, more internal tests could be performed.

**Evaluation**

This system performs its job adequately, and reliably. The tests performed show the system to be reliable and resilient. The control is distributed over multiple systems, and the communication is reliable. However, there is still some room for improvement. By distributing the system further, and adding another board to handle specifically sensor data, can increase the efficiency of the Mega, and allow more sensors to be added without running out of physical pins or slowing down the system. The full implementation of the Bluetooth functionality of the ESP would allow for further redundancy. Adding Real Time Operating System (RTOS) elements to all aspects of the system would allow for parallelization of each system ensuring more reliable communications. The UI while functional, is somewhat basic, and with the addition of alpha-beta testing (under non-pandemic conditions) would allow for a more user-friendly user interface.

**Video Demonstration link**

**Git Repo Link**

<https://github.com/noblegasses/SOFT564z.git>