Embedded Systems Project Report

Name: Dagnino Simone
Student ID: 247194

Name: Pastorelli Patrick
Student ID: 247440

***Abstract*—In this project we developed a firmware for an Internet of Things system based on ESP-32. This new device gathers data from an ultra-wide band radio and uses them to localize nodes in the surrounding environment, using techniques of relative localization. It also gathers additional information about the environment through other additional sensors and prints everything on a small LCD display. The program provides a user-friendly graphical interface organized in different screens which are accessible using 3 main buttons. Wi-Fi is used to send on the internet all the data gathered by the sensors. The project aims at developing a fast, robust, and efficient firmware, which saves as much energy as possible to maximize the duration of the battery.**

I. Introduction (*Heading 1*)

Over the past years the concept of Internet of Things (IoT) emerged as a dominant technology in everyone's life. A lot of different sectors took advantage of the power of IoT devices, ranging from health care to electrical appliances, from automotive to weather forecasting. One of the most active research areas is about relative localization, which aims at localizing the nodes of a network with respect to each other and without a fixed global reference, and this is one of the goals of this new device. The firmware we have developed allows the system to interface with another MCU, which implements an Ultra-Wide Band (UWB) radio used to receive the position of other nodes through the UWB technology. These positions are calculated exploiting techniques of relative localization and, once they are received by the system, they are elaborated and rendered on the display to create a local map of the space around the device.
For gathering these positions, we chose to use ultra-wide band technology as it allows high-bandwidth communications with very low energy consumption. This aspect is crucial since the device is battery driven; therefore, an energy-efficient firmware is the key in obtaining a performant system with an acceptable autonomy.

The UWB radio is not the only peripheral which needs to be managed, we also interface with a bunch of other sensors:

- Magnetic sensor
- Gyroscope / accelerometer
- Brightness sensor
- Temperature / humidity sensor
- LCD display (240x240 pixel RGB)

The device is based on an ESP-32 equipped with 4MB of RAM, enough to store all the necessary structures to manage the peripherals, including the LCD display, which requires allocating a 240x240 RGB buffer to be properly managed.
We use a magnetometer to implement a compass, which is rendered on the LCD together with the UWB nodes.
The gyroscope and the accelerometer can be used to gather information about the movement of the system.
The brightness sensor is used to dynamically adjust the brightness of the display based on the environmental level of light. This functionality has two main benefits: it improves the user experience by adjusting the brightness level to a comfortable level, and it helps reduce the energy consumption of the device. Finally, all these data are sent to a server using the integrated Wi-Fi module of the ESP-32.
We used ESP-IDF framework to setup the project and FreeRTOS to simplify the creation and management of the various tasks that the device executes.
This device could be used in many different applications. Imagine companies gaining real-time visibility into their assets, which can be used to ensure security through continuous surveillance; the presence of an accelerometer is crucial in this scenario to detect unauthorized use of any asset. Hospitals can enhance patients' safety and staff efficiency by tracking the position of equipment and/or of patients themselves. Here the presence of an accelerometer is useful as well since it can detect dangerous situations like the fall of a patient. The potential of this device can go beyond surveillance scenarios, and it may reach the realm of entertainment as well: museums could transform into more interactive environments, where visitors can plan personalized journeys by discovering the points of interests thanks to the UWB localization of the entire building. Activities like escape rooms, which have been growing a lot in the past few years, also could take advantage of this device by building new location-based challenges.
The rest of the report is organized as follows: related work is presented in section II; in section III we provide a brief description of the sensors and how we configured them. In section IV we describe how sensors can communicate with the ESP32; the software architecture is explained in Section V while in section VI we propose a model of the system by using a UML state machine diagram. Section VII presents the results of a power analysis of the whole system and Section VIII concludes the paper.

II. Related Work

The relative localization technique has already been used in different scenarios. [1] proposes a system which can track objects in an indoor environment. This system could be used to track objects of interest which are not geo-referenced and that can possibly be moving during the process. However, it proposes a comparison between two different commercial products which use different technologies (UWB and BLE)

rather than the description of a novel system and its possible use cases, like we do in this paper. Other works try to use the UWB technology to develop a thrustful and robust localization systems to be used in more critical scenarios, [2] proposes an indoor localization method for pedestrians which uses UWB and PDR (pedestrian dead reckoning) and that achieves a decimeter-level positioning, which is a crucial aspect for the security of large indoor environments such as airports and mines. [3] focuses mainly on the security of company's assets by providing a real-time localization system. In addition, they integrate this process with the creation of a 3D map of the environment based on the data collected by the localization system. Finally, [4, hospital env] describes the design of a prototype which can provide seamless indoor/outdoor location tracking by using both the UWB technology for the indoor scenario and the GPS to provide accurate outdoor localization. What is common to all these proposals is the lack of any additional sensor to use during the process; that's why we integrated sensors to support the core which is in charge of doing localization. The presence of an accelerometer/gyroscope can be crucial in safety scenarios, where a sudden movement could mean that something critical is happening: think about a hospital scenario, where a patient who has mobility difficulties could fall down on the floor without the possibility of calling for help; with an adequate setup, the accelerometer could detect the fall and autonomously send a notification to a member of the staff. The same procedure could be used to secure industry's assets: if some equipment is not supposed to be used but the device attached to it detects a movement, then it could signal an unauthorized access to the equipment, in addition to the real-time localization through UWB.

III. SENSORS

The presence of sensors is crucial in electronic systems to gather information from the external environment, in this system we used 4 different sensors to gather all the needed data; here we shortly describe how they work and how we used them.

Magnetometer

The LIS2MDL is an ultra-low-power 3-channels magnetometer that includes an I2C and a SPI communication interface. It senses the magnetic field around itself and then it uses an ADC to convert these values into a digital representation. The conversion is done for each channel (x, y and z) and each result is stored in a pair of 8-bit registers: one for the MSB and one for the LSB. When reading the conversion from the sensor, it will return each register once at a time, therefore we need to merge the 8-bit values into a single 16-bit value. Once we have a 16-bit value, we can convert it in milligauss (mG) by calling the appropriate function provided by the driver.

The sensor has a lot of other important registers which allow to change its default behavior:

- Output Data Rate (ODR) specifies at what frequency the output registers are updated; the available rates are 10-20-50-100 Hz, we set this value at 10Hz both because we don't need a high refresh rate and to keep the consumption low.
- Temperature compensation enhances measurement precision but also leads to higher overall power consumption.
- Operating mode allows you to select either idle mode, continuous mode, or single mode. In continuous mode the sensor is continuously measuring the magnetic field, while in single mode the sensor returns in idle mode after one measurement. In our implementation we keep the sensor in idle until the correct page is selected by the user, then we set it to continuous mode while the user remains on the screen.

We use the magnetometer to draw a compass which dynamically changes as the device moves; to obtain the orientation of the device from the measurements of the 3 channels we use the following C-style pseudocode:

1. take the x and y components of the magnetic field and calculate $D = atan2(y/x)$
2. $D = D * 180/\pi$, convert D from radians to degree.
3. if D <= 45 or 315 < D then NORD
4. else if 45 <= D < 135 then EAST
5. else if 135 <= D < 225 then SOUTH
6. else WEST

Accelerometer/Gyroscope

The LSM6DSOX features a 3D accelerometer and a 3D gyroscope, both I2C and SPI communication interfaces and a small machine learning core which can do tilt and fall detection, as well as a step detector and step counter. This sensor offers four different operating modes, from the simplest where it behaves like a slave and the CPU acts as a master, to the more complex where an additional module (e.g camera module) can be connected to the sensor through I2C or SPI to send/receive additional data; in this project we used the first one as we don't need more advanced features. Every measurement of acceleration involves an analog low-pass filter before the conversion through an ADC, and an additional digital low-pass filter as soon as the conversion has completed. The digital value for each axis is stored in two 8-bit registers; after we retrieve these values through the I2C bus we need to convert them into milliG (mG) for the linear acceleration and into millidegree per second (mdps) for the angular acceleration using the correct function provided by the driver.

The sensor exposes a lot of registers to customize its final behavior; in the project we set up the following:

- CTRL2_G and CTRL1_XL to set the ODR and the scale for, respectively, gyroscope and accelerometer. the scale is the interval of measurable values, which can be set either to ±2, ±4, ±8, ±16 G and ±250, ±500, ±1000, ±2000 dps; we opted for the default value of ±4G for linear acceleration and ±500dps for the gyroscope.
- CTRL3_C to enable the block data update, which updates the output registers only after both MSB and LSB registers have been read.
- CTRL8_XL to configure an additional high-pass filter after the linear acceleration conversion.

We decided to set the ODR at 12.5Hz, which is the slowest one, since we don't need high speed in our working scenario; furthermore, we decided to set ODR to 0 when the user is not on the accelerometer/gyroscope screen, saving the power needed to do the measurements.

Temperature/Humidity sensor

The SHT4X is an ultra-low power temperature/humidity sensor equipped with an I2C interface. Even if the communication protocol is the same as the other sensors, in this case we cannot freely access the internal registers of the device but, instead, the hardware offers a set of commands to interact with the sensor. The main command is the one that allows to retrieve temperature and humidity values, which are sent in a packet of 3 bytes: two of them represent the actual value while the third is a checksum byte which uses CRC algorithm. Once the data has arrived the programmer should convert them to the appropriate measurement unit, using the function provided by the driver.

Brightness sensor

The TSL2591 is a very-high sensitivity brightness sensor equipped with an I2C interface. The 8-bit registers provide a variety of control function and can be read to obtain the result of ADC conversions.

The registers that we used the most in this project are:

- C0DataL and C0DataH which stores, respectively, the LSB and MSB of the ADC conversion.
- ENABLE register, which allows to disable the sensor between one read and the next one.

After the data has been read from the C0DataL and C0DataH, the programmer can convert the raw value into the corresponding value measured in LUX.

IV. Peripherals communication

All the sensors are connected to the MCU via Inter-Integrated circuit protocol (I2C), which is a serial, synchronous, half-duplex protocol, widely used for on-chip communication.

Table 17. Transfer when master is writing one byte to slave

Master	ST	SAD + W		SUB		DATA		SP
Slave			SAK		SAK		SAK	

Table 18. Transfer when master is writing multiple bytes to slave

Master	ST	SAD + W		SUB		DATA		DATA		SP
Slave			SAK		SAK		SAK		SAK	

Table 19. Transfer when master is receiving (reading) one byte of data from slave

Master	ST	SAD + W		SUB		SR	SAD + R			NMAK	SP
Slave			SAK		SAK			SAK	DATA		

Table 20. Transfer when master is receiving (reading) multiple bytes of data from slave

Master	ST	SAD+W		SUB		SR	SAD+R			MAK		MAK		NMAK	SP
Slave			SAK		SAK			SAK	DATA		DAT A		DATA		

This specific protocol simplifies the hardware a lot because all sensors share the same bus, therefore only GPIOs for serial data line (SDA) and serial clock line (SCL) are needed. Software is also a lot simpler since the protocol structure is straightforward, and its implementation across the different drivers of the different sensors are quite similar with each other. The general scheme for I2C transactions is shown here: For example, the operation to read one byte from a device can be described like this:

1. Master sends START message over the bus.
2. Master sends slave address (SAD) and specifies that it wants to write (W)
3. Slave sends back a slave-acknowledgment message (SAK)
4. Master writes the sub-address, that is, the address of the register to be read.
5. Slave sends SAK back.
6. Master sends a second start (SR)
7. Master sends the slave address again, but this times it specifies that the operation is a read (SAD+R)
8. Slave sends SAK back.
9. Slave sends data from the register over the bus.
10. Master sends a not-master-Acknowledgment that informs the slave to stop sending data (NMAK)
11. Master sends STOP message.

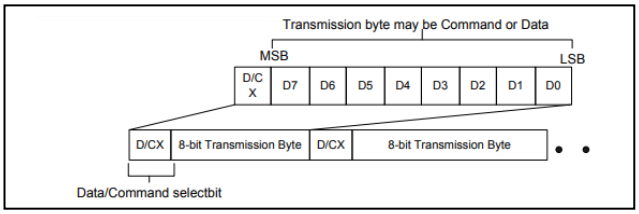
This type of communication is used for:

- Magnetic sensor
- Gyroscope / accelerometer
- Brightness sensor
- Temperature / humidity sensor

The LCD display is the only peripheral which uses Serial Peripheral Interface (SPI) to communicate with the MCU. The protocol is implemented in half-duplex mode, and it requires 5 GPIOs on the MCU:

1. Serial data in/out (SDI/SDO)
2. Clock line (SCL)
3. Data/command line, to specify the type of transaction.
4. Chip selection line, active low for this device.
5. Reset line, to trigger a reset of the display.

A general read operation for this peripheral is structured in this way:



1. Master sends data/command bit that informs the display how to interpret the following byte.
2. Master sends the actual byte containing data/command.

The cycle repeats until the master has sent all the information.

V. Software architecture

Our implementation of the firmware started with the development of the drivers for each sensor; while most of them already had a basic driver to integrate, we had to write the driver for the TSL2591 from scratch. To integrate the other drivers we had to implement platform specific write/read functions, to make them work with the ESP32 API. To handle the display, we used a component called dispcolor. The component is based on a driver for the display, and it builds on top of it a lot of handy features like a functioning printf and other different functions to draw primitive shapes.

The driver allocates a buffer in RAM of 240x240 pixels, each one encoded with a 16-bit integer since the screen works with 16-bit RGB color depth (5 red | 6 green | 5 blue).
All the operations on the screen, like drawing a new pixel, are done inside this local buffer until the specific update function is called, then the driver starts a new SPI transaction to send the entire content of the buffer to the LCD. The transmission could, in theory, send only those pixels that have actually changed since the last call of update, but the driver doesn't implement this feature.
Since the transmission is a simple and repetitive task, the driver delegates it to a DMA device so that the CPU is free to do more intensive and important operations.
Dispcolor is a library based on the driver and it comes with many useful features to draw more complex shapes without the need to think in terms of single pixels. It offers functions for drawing segments, rectangles, triangles, circles and arcs.
The library also offers a working printf function, which is extremely useful in our application to write on the screen all the data gathered by the different peripherals.
In our software architecture we decided that the main task will become the display handler once it finishes setting up all the system, therefore we decided to call the update function only inside this task, after a new screen has been drawn in the buffer. We set a 150 ms delay between one update and the other since we don't need a high refresh rate in our working scenario. Once the drivers were working, we started developing the actual firmware and we chose to exploit the abstraction provided by FreeRTOS. FreeRTOS is one of the most used real-time OS on the market; it provides a small kernel, which exposes a set of syscalls to avoid managing the hardware resources manually (e.g. the I2C interface or the interrupt interface); it also implements higher-level concepts like tasks and queues, that we used heavily in the firmware.
A task in FreeRTOS is like a thread in UNIX world, which is the schedulable entity used in the system; using different tasks allows us to exploit the concurrency offered by the kernel and, potentially, parallelism since the ESP-32 is a dual-core chip. A queue is a thread-safe buffer that we used to achieve thread-safe communication between a pair of tasks.
We created a separated task for every single sensor, whose goal is to read from the sensor and to write the result in the appropriate queue; each queue is read by the GUI task, whose goal is to render the correct screen with the up-to-date data coming from the queues and, finally, to refresh the display.
The graphical user interface (GUI) is organized in a tree-like structure: the homepage is the root of the tree, where a menu allows the user to navigate through all the other screens. We provide different pages to navigate through:

- Homepage, which contains the menu with other pages, together with some general information about temperature, humidity and brightness.
- Gyroscope/accelerometer page, which contains data gathered from the sensor.
- Compass page, which contains the 4 cardinal points and a needle which behaves like a compass. This is done by reading data from the magnetic sensor.
- Compass page with UWB points, which adds to the compass the set of points read from the UWB radio.

The system for navigating through the screens consists of 3 buttons:

- Homepage button, to go back to the menu.
- Enter button, to enter the currently selected screen on the menu.
- Scroll-down button, to move the cursor on the menu.

Every screen shows the battery level as a colored circle on the border of the display; as the battery discharges the color changes from green to red.
To save as much energy as possible we put in low-power mode a sensor when the corresponding screen is not being viewed, and we put the task which reads from it in suspended state. Then we wake the sensor up together with its task when the user chooses to enter the relative screen. This mechanism would be complex without the support of FreeRTOS, but luckily its abstraction level helps us in the process.

Finally, there is the Wi-Fi task, which uses the Wi-Fi module built inside ESP board to share all the data to a server.
The Wi-Fi module built inside the ESP32 requires a quite complex setup procedure, but the key steps are:

- Turn on the Wi-Fi module and initialize the WIFI station structure, which will be used to establish a connection.
- Set up the structure which holds the SSID and PSW of the network we want to connect to
- Start the procedure to connect to the network.

Once the connection is established, we set up a new task which reads data from all the queues which contain data from the different sensors and formats them into a JSON message which is then sent to a server using the UDP protocol.
The connection to the server works the same as in the POSIX environment:

- We create a socket to talk to the server.
- We specify IP and PORT of the server and call the getaddrinfo function, which returns the data needed to connect to the server.
- We call connnect on the socket using the output of getaddrinfo

Once the connection to the server is established, the task enters a loop:

- It reads data from the queues.
- It formats them into a JSON message where each line has the form: "data_name: data_value"
- It sends the message to the server using the POSIX write syscall.
- For debugging purposes we made a small UDP server in python which echoes back every message in uppercase, so the task blocks until the server sends its response.
- Finally, it waits for 2 seconds and the loop restarts.

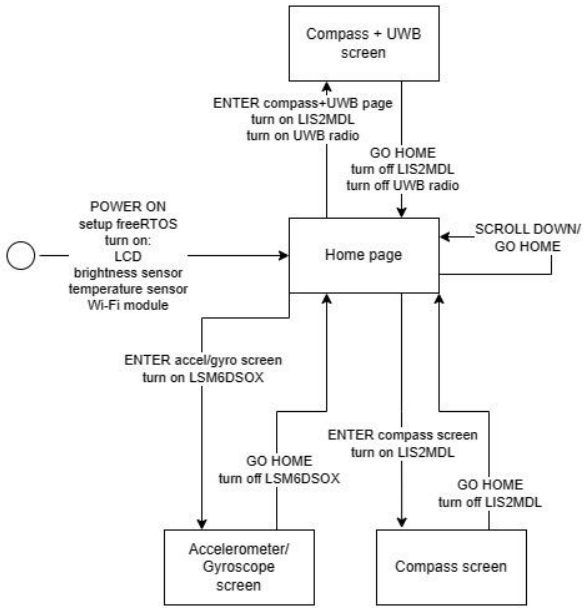
Even though the UDP protocol is not as reliable as TCP, it doesn’t have all the overhead of the three-way handshake and of the retransmission mechanism, which is quite heavy and, in this case, not worth the cost.

VI. State Machine Diagram

The behavior of our system can be easily schematized using a UML state machine diagram, one of the many diagrams

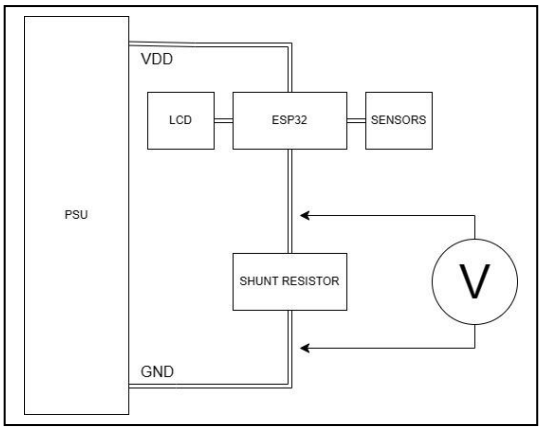
available in the UML standard. This diagram is particularly useful when we need to schematize the behavior of an event-driven systems, which is a system that it's continuously waiting for the occurrence of an event which, in our case, is the press of a button. This diagram provides nodes and edges to draw a graph-like structure; different nodes represent different state of the machine, while an edge represent an event that can trigger a change of the state of the machine (i.e. move to another node); in our implementation each different state is represented by a different screen, and each state transition is triggered by the user pressing one of the available buttons. Different sensors may be turned on/off accordingly to the current state of the machine and to the current event (e.g. the accelerometer is turned on only if the user enters the corresponding screen), while other peripherals are always on, in order to offer a pleasant user experience (e.g. the LCD, otherwise the user cannot have feedback from the system). The Wi-Fi module needs to be always on as well to send packets at regular intervals without the need to reconnect to the access point at every single transmission.

The entry point of the diagram is the transition associated with the event POWER_ON, which sets up freeRTOS, the different tasks and the always-on peripherals. To make the whole diagram more readable, we assume that these peripherals are always running in the background after the first initialization.



VII. Power analysis

The last part of this project focused on the analysis of the current and power consumption of the system. We measured the system in many different states to figure out how big is the contribution to the whole system of every single component. The measurements were done using an external power supply unit, which provided a Vdd of 5V to the system, a shunt resistance connected in series between the system and the common ground and an oscilloscope which measured the voltage across the shunt resistor. The setup is schematized in the following image.



Since the factor conversion of the shunt resistor is 1, the voltage measured by the oscilloscope can be read also as the current across the shunt resistor, which is constant across the whole circuit. All the measurements are one minute long in order to have a relevant set of data to analyze.

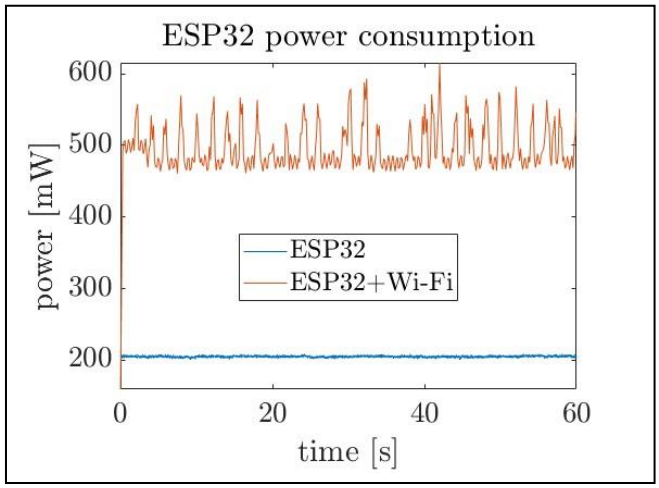
We measured the current for different configurations:

- ESP32
- ESP32 + Wi-Fi module
- ESP32 + LCD display at 100 lux (to simulate a dark environment)
- ESP32 + LCD display at 450 lux (to simulate the normal brightness inside a building)
- ESP32 + LCD display at 3000 lux (to simulate the high brightness of sunny day)
- ESP32 + brightness sensor
- ESP32 + magnetometer
- ESP32 + accelerometer/gyroscope
- ESP32 + temperature/humidity sensor

Once we have the current for every scenario, we need to isolate every contribution of each peripheral, which is simply done by subtracting the contribution of the ESP32 alone from every measurement. What is left are the currents used by each component which can be used to calculate the relative power consumption using the formula:

$$V_{DropShunt} = (20uV * measuredCurrent) + 10uV$$
$$P = \left(Vdd - V_{DropShunt}\right) * measuredCurrent$$

Then we calculated the average current and power, and we obtained the following results:



Configuration	Avg current [mA]	Avg power [mW]
ESP32	41.03	205.15
ESP32 + WiFi	99.09	495.44
LCD 100 lux	24.80	124.01
LCD 450 lux	26.23	131.18
LCD 3000 lux	31.27	156.36
Brightness sensor	0.74	5.74
Magnetometer	0.21	1.08
Gyroscope/Accelerometer	0.82	4.12
Temperature/Humidity sensor	0.54	2.71

From the measurements it clearly emerges that the sensors are negligible if compared to the ESP32 and to the LCD, which account for 97-98% of the total power consumption. In particular, the Wi-Fi module consumes more than the ESP32 itself without the module activated. The LCD power consumption varies depending on the light of the external environment, although it is always confined in the range of (120, 160) mW.

VIII. Conclusions

We have presented the description of a novel system which integrates a UWB radio to do relative localization, a set of sensors to gather data from the environment and an interface for the user consisting in a display and a set of 3 buttons. We have briefly described which sensors have been embedded in the system and the structure of the I2C protocol which allows the MCU to talk with the peripherals. The firmware we have developed is heavily based on freeRTOS, which allows us to manage the entire system from a higher level of abstraction thanks to concepts like task and thread-safe queue. We have described the behavior of the system in response to the human's behavior by building a UML state machine diagram and, finally, we have carried out a power analysis, which showed that the biggest contributions to the overall power consumption are from the ESP32, especially when enabling its Wi-Fi module, and from the LCD; together they account for 97-98% of the overall power consumption.

References

[1] A. R. Jiménez and F. Seco, "Finding objects using UWB or BLE localization technology: A museum-like use case," 2017 International Conference on Indoor Positioning and Indoor Navigation (IPIN), Sapporo, Japan, 2017, pp. 1-8, doi: 10.1109/IPIN.2017.8115865.

[2] Liu, F.; Wang, J.; Zhang, J.; Han, H. An Indoor Localization Method for Pedestrians Base on Combined UWB/PDR/Floor Map. Sensors 2019, 19, 2578. https://doi.org/10.3390/s19112578

[3] IF. M. Nur and N. Z. Janah, "3D Visualization Design of Realtime Position Tracker Based On Ultra-Wideband Device," 2019 2nd International Conference on Applied Engineering (ICAE), Batam, Indonesia, 2019, pp. 1-6, doi: 10.1109/ICAE47758.2019.9221739. keywords: {3D map;tracking;positioning;real-time},

[4] Lijun Jiang, Lim Nam Hoe and Lay Leong Loon, "Integrated UWB and GPS location sensing system in hospital environment," 2010 5th IEEE Conference on Industrial Electronics and Applications, Taichung, 2010, pp. 286-289, doi: 10.1109/ICIEA.2010.5516828