Hochschule Konstanz
Fakultät Elektrotechnik
und Informationstechnik

HTWG - Hochschule für Technik, Wirtschaft und Gestaltung

Masterprojektarbeit

UWB Software Guide

Positionsbestimmung mittels Ultrabreitband

erstellt von
Tom Herter
EIM 3
Mtr. 306971
&
Sebastian Krebs
EIM 3
Mtr. 298229

betreut durch
Prof. Dr. Knievel

Konstanz, February 27, 2024

Contents

1 Introduction

1.1 Description

This software guide is designed to explain the concept of the firmware running on the UWB tag and all the UWB anchor devices. The goal is to give a understanding what classes and files are used and how the interoperate together. Additionally to this guide a full Doxygen documentation will be provided.

1.2 Naming Conventions

Within this Software Guide detailing the Time-of-Flight (TOF) function, it's important to note that the terms "initiator" and "tag" are used interchangeably. Both refer to the UWB (Ultra-Wideband) device responsible for initiating the TOF measurement by transmitting a UWB request to the relevant devices in its vicinity.
These devices can also be alternatively referred to as "anchor", "responder" or "landmark". This nomenclature is employed because these devices play the pivotal role of responding to the UWB request and guiding the initiator in computing the distance between them. The interchangeable use of these terms is intended to provide clarity and flexibility in describing the dynamic roles these devices undertake in the TOF measurement process.

1.3 Basic functionality and setup

The primary function of this setup revolves around the estimation of tag positions within a predefined coordinate space. This estimation is achieved by leveraging the distances between the tag and various anchors strategically placed within the room. To facilitate a clearer grasp of the setup's workings, an illustrative sketch featuring example coordinates can be found in Figure 1.1.

In this setup, the anchors are situated throughout the room in a deliberate yet arbitrary manner, each possessing known coordinates. The key to this system's functionality lies in the tag device's ability to discern the positions of these anchors. With this information, the tag can deduce its own location in all the X, Y and Z directions by gauging the distances to all the anchors.

In Figure 1.1, these distances are visually represented and labeled as d_1 to d_5. The critical aspect of this process is that it enables the tag device to employ an Extended Kalman Filter (EKF) for the purpose of position estimation. This filter algorithm plays a

pivotal role in refining and enhancing the accuracy of the tag's estimated position within the coordinate space and is further discussed in chapter 4.

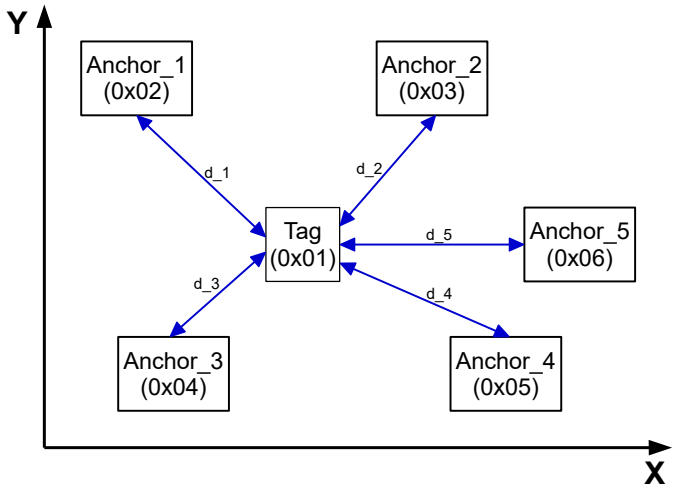

Figure 1.1: Sketch of the complete setup.

1.4 Flow of Information

In Figure 1.2 the information flow of the entire system is pictured. The most important component are the UWB ranging messages exchanged between the Tag and every Anchor that are ultimately used to estimate the Tag's position. With every sent UWB message from the Tag to an Anchor the channel parameters are extracted and transmitted via WiFi to an MQTT-Server. The UWB Tag itself also transmits its coordinates over MQTT to a different topic. The MQTT mechanics are further discussed in chapter 6.

The configuration of the Tag, meaning putting in the coordinate of each anchor into the EEPROM, is executed via Bluetooth by a Bluetooth-Capable device running the configuration software. Additionally the Tag sends its position via Bluetooth where the configuration software has the ability to display it in a 2D plot in real time. Bluetooth Handling techniques are further elaborated in 5.

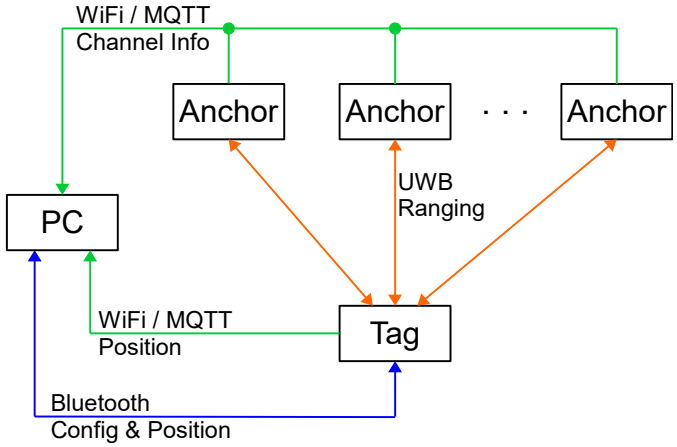

Figure 1.2: Sketch of the complete setup.

2 Main File Structure

This chapter outlines the structure and organization of the main.cpp file of the UWB Project that runs on all of the UWB devices.

2.1 Identification Parameters

All UWB devices, irrespective of their function as anchors or tags, utilize the same codebase during the flashing process. The differentiation between these devices relies on two critical parameters configured during the initial flashing procedure. These parameters serve a dual purpose: defining the device's role as a tag or anchor and assigning a unique device identifier.

In essence, these parameters not only dictate whether a device acts as a tag or an anchor but also bestow each device with a distinctive identifier, enabling them to autonomously determine their respective addresses. This configuration approach ensures versatility and individuality among the UWB devices while maintaining a unified codebase.

2.1.1 Set Identification Parameters

The configuration of each device is primarily defined by two key parameters: the device ID and its designated role. To establish these parameters, a setup process is executed through the preproduction_eeprom_settings function, which is integrated into the setup section of the main.cpp file. This function accepts the current role and device ID as uint8_t variables and records them in the EEPROM memory at an arbitrary position.

It is important to note that this setup process should only be performed once per device, typically during the initial configuration. Once the parameters are set and saved in the EEPROM, there is no need to invoke the preproduction_eeprom_settings function again during subsequent flashing processes unless a change in device configuration is desired. This approach streamlines the device setup procedure and ensures the persistence of the specified parameters across device power cycles.

Role

The role parameter is pivotal in determining whether a device functions as a tag or an anchor within the system. This parameter is stored in the EEPROM memory at address 0 and is represented by a binary value of 0 or 1. Throughout the codebase, this parameter is accessed and utilized through the IS_INITIATOR constant.

If the value stored at EEPROM address 0 is 1, the device is configured to act as a tag. In

this mode, it typically takes on the role of initiating UWB messages and serves specific functions accordingly. Conversely, if the value is 0, the device operates as an anchor. In this capacity, it primarily responds to incoming UWB messages, contributing to the overall UWB system by providing reference points and data to other devices.

This parameterization approach allows for dynamic role assignment and behavior based on a single value, making the system flexible and adaptive to the needs of each device.

Device ID

The device ID is also stored in the EEPROM, and it serves the crucial function of assigning a unique address to each device. This individualized addressing is essential for facilitating clear and unambiguous communication through UWB packages within the system. Each device must possess a distinct device ID to ensure that messages are accurately addressed to the intended recipient, and that devices can be uniquely identified in the network.

By storing the device ID in EEPROM, the ID can persist across power cycles and reboots, maintaining consistent identification for each device in the system. This approach is vital for seamless and reliable communication and data exchange between UWB devices, ensuring that messages reach their intended destinations without ambiguity.

2.1.2 Address

To ensure the secure and accurate encryption of UWB packages using the Advanced Encryption Standard (AES), each device requires a unique 32-bit address. This address is essential for enabling precise encryption and decryption processes. By encrypting a message with the destination address, the system guarantees that only the designated receiver can successfully decrypt the message. As a result, every device, both anchors and tags, must be aware of their own unique address.

For anchors, their individual addresses are directly determined by their respective device IDs. Each anchor selects its address from a predefined list based on its specific device ID. In contrast, the address for tags remains constant and is universally known across all devices as the global constant INITIATOR_ADDR. This address assignment methodology ensures that each device can securely encrypt and decrypt messages using the appropriate addresses, contributing to the overall data security and integrity of the UWB communication system.

2.2 User Input

User input to change the current operating mode is facilitated through User Button 1. By associating an interrupt with this button, the ble_kill_flag is set, and the current_mode variable is updated accordingly. For a comprehensive understanding of how these components are managed, please refer to section 2.5.3. This section will delve into

the specifics of how the interrupt, ble_kill_flag, and current_mode variable interact to facilitate seamless mode switching in the tag device's operation.

2.3 Main Variables

- **double distances[NUM_LANDMARKS]**
  This array stores the measured distances to landmarks, providing input for the Extended Kalman Filter (EKF) used for position estimation. It is constantly updated by an instance of the TofInitiator class.

- **uint8_t ble_kill_flag**
  A flag used to control the operation of the BLE-Task, allowing for a smooth transition between operational and config mode.

- **coordinate own_position**
  A coordinate structure that holds the estimated 3D position of the UWB device, which is continuously updated by the EKF-Task.

- **DynamicJsonDocument latest_rx_diagnostics**
  This JSON document stores the most recent diagnostics information received from other UWB devices. It is utilized in UWB anchor mode to share information to an MQTT broker.

- **uint8_t current_mode**
  Represents the operating mode of the UWB device, distinguishing between config and operational mode. It is toggled based on user input and controls the execution of relevant tasks.

- **uwb_addr dest_addr_list[]**
  An array containing UWB addresses of responder devices. In UWB initiator mode, this list is used for initiating TOF measurements with multiple responder devices.

- **Task handles (ekf_task_handle, tdoa_task_handle, tof_task_handle, ble_task_handle, mqtt_task_handle)**
  These handles are used to manage and control the execution of different FreeRTOS tasks responsible for specific functionalities in the UWB device.

2.4 Main Functions

2.4.1 setup

The setup function in the UWB device code serves as the initialization routine that is run first when the ESP32 is powered on. It begins by configuring serial communication for debugging and EEPROM for persistent data storage. The pin where the user button 1 is connected to is set as an input and the user_1_button function is attached to it as

an interrupt routine. Furthermore the three pins for the user LEDs are defined as output and all set to zero upon initialization. The current mode is set to "uwb_mode" and the MQTT- as well as the TOF-Task are started.

The function then checks if the device, it is running on, is defined as an initiator. If so the BLE- and EKF-Task are started as well.

If the device is defined as a responder, it reads the JSON file named "uwb_diacnostic_type.json" from the SPIFFS on the ESP32. It then creates a DynamicJsonDocument object (latest_rx_diagnostics) with three times the size of the JSON file, providing extra space for storing communication channel data. The code proceeds to deserialize the contents of the JSON file into the latest_rx_diagnostics object. If there's an error during deserialization, it prints an error message to the serial monitor; otherwise, it indicates successful insertion of the JSON template. The file is then closed. Finally, the device ID is retrieved from EEPROM, and the "DeviceId" field in the latest_rx_diagnostics JSON document is set to this device ID so the MQTT broker can tell where from what responder the data originated.

2.4.2 void user_1_button

The user_1_button function serves as an interrupt routine arrached to the user button 1. First the interrupt is detached. If the device is an initiator and in uwb_mode it will toggle the ble_kill_flag and set to current mode to ble_mode. If it is in ble_mode it will toggle the ble_kill_flag and set to current mode to uwb_mode. Lastly the interrupt gets attached back to the user button 1.

2.4.3 void animate_leds

This function sequentially illuminates and extinguishes LEDs in a predefined pattern, creating a visual animation. It provides a quick and noticeable indication of a specific mode. The LEDs involved in the animation are connected to GPIO pins on the ESP32.

2.4.4 void preproduction_eeprom_settings

This function takes two parameters - dev_id representing the unique device identifier and is_initiator indicating whether the device operates as an initiator or not. It writes these values into EEPROM, ensuring that the correct device ID and role information are stored persistently. This initialization process is crucial for the proper functioning of other tasks and functionalities in the UWB device, as they depend on the settings retrieved from EEPROM.

2.4.5 Freertos Tasks

The functions for the TOF-, MQTT-, BLE-, and EKF-Task are handeled by FreeRTOS to ensure concurrent and parallel execution of those functionalities within the device

since they are the most critical to the operation. There functions are handled separately in the sections 2.5.1 - 2.5.4.

2.5 Scheduler

To efficiently measure distances to multiple anchors simultaneously, calculate their respective positions, and transmit the data to the software running on the devices, a scheduling system is implemented. This scheduling system is powered by the FreeRTOS library, an open-source real-time operating system, which leverages the dual-kernel capabilities of the ESP32 microcontroller.

The dual-kernel architecture ensures that critical tasks, like estimating distances, remain uninterrupted, reducing the risk of producing corrupted data. All tasks are initialized, started, and halted within the 'main.cpp' file, creating a well-structured and organized framework for managing these essential operations.

2.5.1 TOF_Task

In the context of TOF-Task, it plays a pivotal role in the overall system, particularly in the domain of Time-of-Flight (TOF) distance measurement and positioning. This section outlines the primary functions of TOF-Task in a narrative form.

The initial task for TOF-Task is to identify the device's role within the system, specifically whether it should operate as a TofInitiator or a TofResponder. This role assignment is crucial as it dictates the device's behavior and responsibilities throughout the entire process.

Once the device's role is established, TOF-Task proceeds to create an instance of the appropriate object, either TofInitiator or TofResponder, and initializes it. This instantiated object assumes the critical responsibility of managing the distance measurement process.

At the core of TOF-Task's operation lies the facilitation of distance measurements. If the device takes on the role of a responder, it remains in a passive stance during the measurement process, awaiting initiation from another device. Conversely, when the device serves as an initiator, it leads the way in conducting distance measurements.

In scenarios where the device functions as the initiator, a crucial user feedback mechanism comes into play. User LED 1 is activated to provide visual feedback, signaling that this particular device is taking the initiative in distance measurement and position estimation. Such feedback proves invaluable, especially in systems involving multiple devices, ensuring users are aware of which device is actively involved in determining the position.

For the initiator role, TOF-Task manages the storage of calculated distances. These distances, representing measurements to each anchor, are stored within the "double distances[]" array. This array serves as a vital source of information for the subsequent stages of the positioning algorithm.

In summary, TOF-Task serves as the central orchestrator, dynamically allocating roles to devices as initiators or responders and configuring the corresponding objects to execute distance measurements. This dynamic allocation of tasks ensures the effective functioning of the system, and when necessary, it offers user feedback to enhance the transparency of the positioning process.

2.5.2 EKF-Task

The EKF task is a critical component of the system, running exclusively on the tag device. Its primary purpose is to leverage the information provided by the "double distances[]" array, which contains the distances to all available anchors. With this data, the EKF task performs calculations to estimate the tag's position. This estimation process is triggered every time when all landmark distances are covered, ensuring that the system continually updates its understanding of the tag's location.

The EKF algorithm, a cornerstone of this task, incorporates statistical techniques and dynamic models to refine the accuracy of the position estimate. By doing so, it can effectively handle factors like noise and uncertainties, making it a robust solution for real-time positioning.

Once the EKF-task completes its calculations, the resulting X and Y coordinate values are transmitted through the UART interface and via MQTT.

For a comprehensive understanding of the theory and principles that underpin the Extended Kalman Filter and its role in this system, you'll find in-depth coverage in Chapter 4. This chapter will delve into the mathematical and conceptual foundations of the EKF, ensuring that readers gain a clear grasp of how it operates and contributes to the overall functionality of the system.

2.5.3 BLE_Task

Similar to the EKF-Task, the BLE-Task is designed to run exclusively on the tag device, and it serves two primary objectives.

The first objective is to facilitate device configuration, a process carried out through Bluetooth connectivity with the dedicated "UWB Configuration" desktop software. This configuration process entails defining the coordinates of each anchor device and saving these values in the device's EEPROM. A comprehensive guide on how to connect the tag device with the desktop software can be found in the UWB User Guide.

The second objective of the BLE-Task is to relay the current position of the tag device via Bluetooth to the corresponding desktop software. The software then plots these positions in an X-Y coordinate system, providing a real-time visual representation of the tag's movements.

When the BLE-Task is first initialized, it instantiates the BleConfigLoader class. The workings of this class, along with the broader Bluetooth management process, will be expounded upon in the forthcoming chapter, referenced as 5.

The tag device operates in two distinct modes saved in the current_mode variable. The first is the uwb_mode, which serves as its standard operating mode. In this mode, the device performs distance measurements, calculates its own position, and transmits this information via UART, as previously described in 2.5.1.

The other operational mode is the ble_mode, which is exclusively focused on maintaining a connection with the desktop software. In this mode the device waits for the trigger from the UWB Configuration software to take in the newly inputted anchor positions and save them into the EEPROM. It also outputs the currently saved anchor positions so that they can be displayed and changed in the UWB Configuration desktop software.

Output current position

In the default startup scenario, where the uwb_mode is automatically selected without any additional user input, the BLE-Task is responsible for taking the currently estimated position from the global own_position variable and transmitting it via Bluetooth. This transmission is achieved using the send_position method of the BleConfigLoader class. Detailed insight into the functionality of this process will be provided in the forthcoming chapter, referenced as 5.

This continuous updating procedure persists indefinitely, with the current position being sent repeatedly, unless the ble_kill_flag is triggered. The flag is set when a user initiates a mode change to ble_mode by pressing user button 1.

This mode-switching functionality ensures that the device can seamlessly transition between its primary operating modes based on user input.

2.5.4 MQTT_Task

The MQTT_Task function handles all the MQTT communication. It first determines if it is run by the tag or any responder and figuring out its device ID by reading the EEPROM. Then it creates a string identifier (String clientId) containing its device ID so the MQTT messages can be told apart from where they originated from.

Subsequently, the function creates an instance of the MqttClient class, encapsulating the essential functionality for MQTT communication. This includes setting up the MQTT topic ("uwb_devices"), specifying the MQTT server details, providing WiFi credentials, and configuring a buffer size for efficient data handling.

Within the core of the function lies an infinite loop that serves as the heartbeat of the MQTT communication task. The primary objective of this loop is to ensure the persistent operation of the MQTT client. Consequently, the mqttClient.update() method is called at every iteration, allowing the client to handle incoming messages and maintain the MQTT connection.

As part of the communication strategy, the MQTT_Task function publishes messages to specific MQTT topics based on the device's role. In cases where the device is an initiator (is_initiator = 1), it publishes position information, formatted as a JSON

string, to the "tag/dev_id/position" topic. Conversely, if the device is not an initiator (is_initiator = 0), it publishes UWB diagnostics information, serialized into JSON, to the "anchor/dev_id/uwb_info" topic.

This task structure enables the integration of the device into a larger Ultra-Wideband (UWB) communication system. It showcases the capability to disseminate critical information, such as device positions or UWB diagnostics, via the MQTT protocol. The inclusion of delays between message publications controls the communication frequency, aligning with the specific requirements of the UWB communication system.

Configuration

When the user triggers a mode change to ble_mode by pressing button 1, the ble_kill_flag is activated, which effectively terminates the loop responsible for continuously transmitting the current position.

In the ble_mode, the device first retrieves its current configuration, encompassing all previously saved anchor positions, from the EEPROM. This retrieval process is achieved through the load_config_from_eeprom method of the BleConfigLoader class. Subsequently, the device compiles this configuration data into a string formatted as a JSON object and transmits it via Bluetooth, using the save_config_to_ble method. This JSON-formatted data includes the anchor positions.

Following this configuration update transmission, a new loop commences. In this loop, the device continually receives the most up-to-date anchor positions sent from the configurator device via Bluetooth. These positions are then saved locally on the tag device, and the new anchor configuration is conveyed back to the configurator device, ensuring that the positions are received and updated accurately.

In addition to this configuration exchange, the animate_leds function is invoked, causing the user LEDs to flash in a distinct pattern. This visual feedback serves as an indicator to the user that the device is in Bluetooth configuration mode.

The configuration loop persists until the "Save Config" button is pressed within the UWB Configuration desktop GUI. This action initiates the sending of a character to the tag device, triggering the save_config_to_eeprom method. This method is responsible for writing the updated anchor coordinates into the EEPROM of the device and subsequently restarting the device with the new configuration in place. This completes the process of updating and saving anchor positions for the tag device during its operation in ble_mode.

3 UWB Handling

3.1 Distance Measurement

The distance measurements in this system are conducted using a Time-of-Flight (ToF) methodology. This means that the time taken for an Ultra-Wideband (UWB) message to be transmitted from one device and received by another is a crucial factor. By subtracting the time when the UWB message is transmitted from the time it is received, we obtain the flight time. This represents the duration it took for the electromagnetic field to propagate from the sender to the receiver. This time is typically measured in seconds.

To derive the distance between the two UWB devices, this time is multiplied by the approximate velocity at which these electromagnetic fields expand. In a perfect vacuum, this velocity would equal the speed of light, which is 299,792,458 meters per second, often denoted as 'c'. However, under atmospheric conditions where this setup operates, the speed of light is slightly slower. Therefore, in this setup, an approximated speed of 299,702,547 meters per second is used.

It's worth noting that the accuracy of the time measurement is crucial since even a small error in this time measurement can result in a relatively significant error in distance estimation. To ensure precise time measurements, the DW3000 IC requires a highly accurate clock signal with minimal drift. The DWM3000 module offers an advantage in this regard, as it already integrates the DW3000 chip with an on-module oscillator suitable for UWB operations. A visual representation of the ToF distance calculation process is depicted in Figure 3.1.

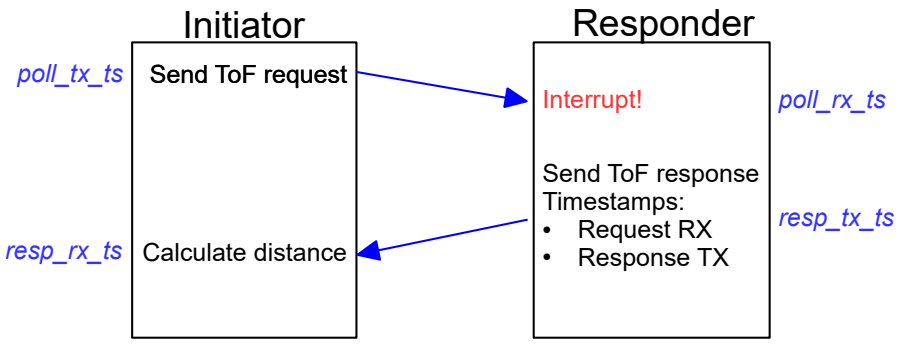

Figure 3.1: Sketch of the complete setup.

To estimate the distance, four timestamps, marked in blue in Figure 3.1, are used. These timestamps are expressed in ticks and are stored in a 32-bit variable. The timestamps are as follows:

poll_tx_ts	Time of request being sent by Initiator
poll_rx_ts	Time of request being received by Responder
resp_tx_ts	Time of response being sent by Responder
resp_rx_ts	Time of response being received by Initiator

Table 3.1: Timestamp variables.

To calculate the time it took for the poll message to travel from the initiator to the responder, the poll_tx_ts timestamp is subtracted from the poll_rx_ts timestamp, yielding the rtd_init value. Similarly, for the time the response message took to travel in the opposite direction, resp_tx_ts is subtracted from resp_rx_ts, resulting in the rtd_resp variable.

To determine the distance between the initiator and the responder, the two times (rtd_init and rtd_resp) are averaged. Before averaging, the response time (rtd_resp) is adjusted by applying an offset factor (1 - clockOffsetRatio). This compensates for potential variations that may arise if the clock of the initiator runs slightly faster or slower than the responder due to factors such as aging effects, temperature, voltage, or manufacturing discrepancies.

Finally, to compute the distance between the two devices, the average time is converted to seconds and then multiplied by the predefined speed of light, representing the speed of electromagnetic field expansion.

This process is subsequently repeated for all of the saved anchors to get the distance to every single one. The calculated distances are saved in the distances array.

3.2 TofDevice

The TofDevice class is a base class for Tof devices, serving as a superclass for both TOF initiators and responders. It encapsulates common functionality and configurations required for TOF devices using the DW3000 chip. The class provides methods for initializing and configuring the TOF device, starting a watchdog timer for handling failure events, enabling LEDs for debugging, and implementing the main loop for the TOF device. Additionally, it includes parameters and configurations specific to the DW3000 chip, such as communication settings, encryption keys, and antenna delay values.

3.2.1 TofDevice Variables

Private Variables

- **Watchdog my_watchdog**
  The my_watchdog variable is an instance of the Watchdog class, representing a software watchdog timer used to monitor the proper execution of the TOF device's main loop. It is initialized with a timeout value, and its purpose is to reset the device if the timer is not reset within the specified duration, enhancing the device's reliability and fault tolerance. The section3.5 delves into the watchdog class.

- **uwb_addr src_address**
  The src_address of type long long holds a 32-Bit UWB address.

- **String type**
  The type variable of type String holds the information whether the device is identified as tag or responder.

- **mac_frame_802_15_4_format_t mac_frame**
  The mac_frame variable is of the type mac_frame_802_15_4_format_t and used to construct and alter the UWB message corresponding to the IEEE 802.15.4 communication standard.

- **uint8_t poll_msg[4]**
  poll_msg is a four byte array representing the specific poll message payload. In this case it is labeled 'P', 'o', 'l', 'l' and is used by the tag ot initiate a UWB cycle.

- **uint8_t resp_msg[16]**
  The resp_msg is similar to the poll_msg used to construct MAC frames. In this case the message is 16 bytes ling with the first 8 bytes reserved for the timestamps of the responder. The last 8 bytes carry the characters 'R', 'e', 's', 'p', 'o', 'n', 's', 'e' indicating the nature of the message.

- **uint8_t rx_buffer[RX_BUF_LEN]**
  The rx_buffer array allocates memory to store a incoming UWB message of the maximum size.

- **uint32_t status_reg**
  The status_reg holds information about the process of a message being sent. It masks errors and timeouts that may occur when a UWB message is transmitted or received.

- **dwt_aes_job_t aes_job_tx, aes_job_rx**
  Both aes_job_tx and aes_job_rx are of the struct type dwt_aes_job_t and contain information needed for the AES encryption/decryption of each MAC frame being send or received.

- **int8_t status**
  The status variable is used to save and evaluate the status of a AES encryption. It masks different AES errors to its bits.

- **dwt_aes_config_t aes_config**
  The aes_config variable represents an instance of the dwt_aes_config_t type. It is used to configure and specify parameters for the AES operations within the TOF device. The dwt_aes_config_t type includes fields for selecting the AES core type, setting the key source, and defining other encryption-related configurations.

- **dwt_config_t config**
  The config variable is an instance of the dwt_config_t type, and it is used to store configuration parameters related to the DW3000 chip, which is used for TOF communication. The dwt_config_t type includes fields such as channel number, preamble length, data rate, and other settings that determine the behavior of the DW3000 chip during communication. This variable is utilized in the setup process to initialize and configure the TOF device for proper operation.

- **dwt_aes_key_t keys_options[NUM_OF_KEY_OPTIONS]**
  The keys_options variable in the TofDevice class is an array of dwt_aes_key_t elements. It is used to store different encryption keys for AES operations within the TOF device. Each element in the array (dwt_aes_key_t) represents a set of encryption keys, allowing for multiple key options. These keys are used to secure communication between TOF devices, providing a level of cryptographic protection for the transmitted data.

3.2.2 TofDevice Constructor

The TofDevice constructor initializes a new instance of the TofDevice class within the Device namespace. It takes two parameters: uwb_addr src for the source address of the TOF device and unsigned long wdt_timeout for the watchdog timer's timeout duration. It begins by initializing SPI comminication to the DWM3000 chip and ensuring that it is in a stable state after startup through a soft reset and a brief delay. Subsequently, it checks the device's idleness in the Idle RC state before proceeding. The method then initializes the DW3000 chip using the dwt_initialise function. If any of these steps fail, it prints error messages and enter a loop indicating failure. Lastly the config data for the operation of the DWM and the AES encryption as well as are the keys_options set in the corresponding variables config, aes_config and keys_options are set

3.2.3 virtual void setup()

The setup method in the TofDevice class serves as the initialization routine for the TOF device. It uses the dwt_configure function to configure the DWM3000 with the config options set in the constructor describes in 3.2.2. If this fails it will print an error message and go in a loop. The configuration of the transmission spectrum parameters

is handled by dwt_configuretxrf, and default antenna delay values are applied through dwt_setrxantennadelay and dwt_settxantennadelay.

3.2.4 void start_wdt()

The start_wdt method in the TofDevice class initiates the watchdog timer associated with the TOF device using the begin method of the Watchdog class targeted in 3.5.4. This method is responsible for commencing the timer, which monitors the execution of the device's main loop.

3.2.5 void enable_leds()

The enable_leds method activates the RX and TX LEDs for debugging purposes. By enabling the LEDs, each transmission triggers a visual indication of sent and received messages, aiding in the debugging process. Note that in practical low-power applications, LEDs might be disabled to conserve power, but in a debug scenario or when the device is hooked up to a stable supply like the main grid.

3.2.6 virtual void loop()

The loop method represents the main operational loop for the TOF device. Within this loop, the device continually resets the watchdog timer using the resetTimer method, ensuring that the device remains within its expected operational timeframe.

3.2.7 char* get_type

The private get_type method returns a pointer to the type variable of the TOF device seen in 3.2.1, offering a descriptive label for the TOF device's classification.

3.3 TofInitiator

The TofInitiator class is a subclass of TofDevice and represents a TOF initiator device. It extends the base class functionality to handle TOF request transmissions, response processing, and secure communication with responder devices. The class encapsulates frame construction, nonce generation, and specific logic for managing communication with multiple responders.

3.3.1 TofInitiator Variables

Extern Variables

- **extern double distances[NUM_LANDMARKS]**
  The distances variable is an external array declared as extern double distances[NUM_LANDMARKS] in the tof-initiator.h file. This array is initialized in ther main.cpp file and is intended to store distance measurements calculated

by the TOF initiator for each responder in the system. The size of the array, NUM_LANDMARKS, is determined by the number of responders in the system, and each element in the array holds the calculated distance for a specific responder. The actual values are be updated during the processing of TOF responses in the process_tof_response method as described in 3.3.6.

Private Variables

- **uint32_t frame_cnt**
  The frame_cnt variable represents the frame count, tracking the number of frames transmitted by the TOF initiator. It is incremented after each transmission and is used to manage frame sequencing within the communication protocol. This count helps maintain synchronization and order in the communication between the initiator and responder devices.

- **seq_cnt**
  The seq_cnt variable represents the frame sequence number, and it is used to keep track of the sequence of frames transmitted by the TOF initiator. It is incremented after each transmission to ensure that each frame has a unique identifier within the communication protocol. The seq_cnt value is part of the frame's metadata and aids in maintaining order and synchronization between the initiator and responder devices in the TOF communication system.

- **uint8_t nonce[13]**
  The nonce variable is a 13-byte array used as a nonce (Number used Once) in the context of secure communication. It serves as a unique value for each frame transmission and is utilized in the encryption and decryption processes to enhance security by preventing replay attacks. The nonce is generated and updated for each TOF frame transmission, contributing to the integrity of the communication between the TOF initiator and responder devices.

- **dst_address**
  The dst_address variable is a pointer to an array of destination addresses for the responders. It represents the target addresses to which the TOF initiator sends its requests. The array holds the addresses of all individual responder devices, and the TofInitiator class uses this information to selectively communicate with each responder one after another during the TOF communication process.

- **uint8_t total_responders**
  The total_responders variable represents the total number of responder devices that the TOF initiator communicates with. It indicates the size of the array of destination addresses (dst_address) and helps manage the looping through responder devices during the TOF communication process. The TOF initiator iterates through the responders, sending TOF requests and processing responses, and total_responders determines the number of devices in this communication setup.

- **uint8_t current_responder**
  The current_responder variable is an index used to keep track of the current responder being communicated with during the TOF communication process. It is incremented and looped to iterate through the array of destination addresses (dst_address), allowing the TOF initiator to selectively send TOF requests and process responses from different responder devices. This index management facilitates sequential communication with multiple responders within the specified array.

- **double tof**
  The tof variable represents the calculated Time-of-Flight for a particular communication instance. It is used to store the calculated time it takes for a signal to travel from the TOF initiator to the responder and back. The tof value is derived during the processing of TOF responses and is an essential parameter for determining the distance between devices using the speed of light.

- **double temp_distance**
  The temp_distance variable is used to buffer the calculated distance based on the Time-of-Flight measurement during the processing of TOF responses. It represents a temporary variable for storing the distance information before it may be further utilized in the application.

3.3.2 TofInitiator Constructor/Dectructor

The TofInitiator constructor takes parameters representing the source address for the TOF initiator (src), an array of destination addresses for the responders (dst), the watchdog timer timeout value (wdt_timeout), and the number of responder devices (num_of_responders). It sets the type String variable to "Initiator" indicating its role, initializes tof, temp_distance and frame_cnt with 0 and seq_cnt to 10 (0x0A). Latly the MAC-Frame is assembled.

3.3.3 virtual void setup()

The setup method initializes the TOF initiator by calling the base class's setup method to configure basic parameters descibed in 3.2.3. It then sets the expected response delay and timeout using the dwt_setrxaftertxdelay and dwt_setrxtimeout functions. After that the parameters for the AES are set for rx encryption and tx decryption.

3.3.4 virtual void loop()

The loop method serves as the central processing hub, orchestrating the continuous operation of the TOF initiator device. It begins by invoking the base class loop method, decribed in 3.2.6, which includes resetting the watchdog timer to prevent system resets due to potential failures.

Within the loop, the current_responder index is iteratively updated, allowing the TOF initiator to cycle through the array of destination addresses. For each iteration, a TOF

request is sent to the selected responder using the send_tof_request function. After sending the method checks certain bits in the status_reg variable to ensure the message was being sent correctly and if so, updates the seq_cnt and frame_cnt counters.

The method then enters a loop until it detects a response message being received by checking the SYS_STATUS_RXFCG_BIT_MASK masked in the status_reg variable. Then the method processes the response by calling the process_tof_response described in 3.3.6 and puts it in the distances array.

3.3.5 void send_tof_request(uwb_addr dest)

The send_tof_request method is responsible for initiating and sending a TOF request to a specific destination address (responder) and is called in the loop function 3.3.4. It method sets the correction key, assembles the MAC frame and configures and initiates the AES encryption. If the encryption is unsuccessful, a error message will be printed on the serial monitor and the method ends. Otherwise the frame is assembled and the transmission is stated using the dwt_starttx function. By setting the two masked bits DWT_START_TX_IMMEDIATE and DWT_RESPONSE_EXPECTED the message is sent without a delay and the initiator is set to wait for a response.

3.3.6 void process_tof_response()

The process_tof_response method is responsible for handling and extracting relevant information from a TOF response received from a responder device. It method begins by clearing the indication of a good received frame in the status register. It then reads the length of the received frame and proceeds to decrypt the payload of the response using AES decryption after putting it in the rx_buffer variable. The method checks for potential errors in the decryption process and throws error messages over the serial monitor if so.

Upon successful decryption, the method verifies that the received frame matches the expected response from the TOF responder. It extracts the relevant timestamps from the response payload, computes the Time-of-Flight (TOF) and distance based on these timestamps, and updates the tof and temp_distance variables like discussed in the Distance Measurement section 3.1.

3.4 TofResponder

The TofResponder class extends the TofDevice class and is responsible for configuring and operating as a TOF responder device. It includes methods for setup and continuous operation in the main loop. The class utilizes AES encryption for secure communication and employs interrupts to handle successful frame reception. Additionally, it maintains a FreeRTOS Semaphore (responseSemaphore) to coordinate the response process, and it updates RX diagnostics information during operation.

3.4.1 TofResponder Variables

Private Variables

- **uwb_addr dst_address**
  The 32-bit dst_address variable of type long long holds the destination address to which the responder sends its response. This address is set during the instantiation of a TofResponder object and is used to ensure that the response is sent to the initiator device.

- **uint64_t poll_rx_ts**
  The poll_rx_ts variable holds the timestamp when the TOF responder receives a poll frame from the TOF initiator. This timestamp is essential for calculating the time of flight and is put in the response frame.

- **uint64_t resp_tx_ts**
  The resp_tx_ts variable holds the timestamp when the TOF responder transmits the response frame to the TOF initiator. This timestamp is used in conjunction with the poll_rx_ts to calculate the time-of-flight and derive accurate distance measurements in the initiator. Therefore it is also sent back in the response frame

- **DynamicJsonDocument* rx_diagnostics_json**
  The rx_diagnostics_json is a pointer to a DynamicJsonDocument object. This object is used to store diagnostic information, such as timing and status parameters, related to the reception of frames by the TOF responder. The use of a JSON document allows for flexible and structured storage of diagnostic data, facilitating analysis and debugging.

- **static SemaphoreHandle_t responseSemaphore**
  The responseSemaphore is a static semaphore in the TofResponder class, implemented using FreeRTOS. It serves as a mechanism to control the flow of the program, specifically signaling the main loop when it's time to send a response. The semaphore is initially unavailable, and it is given (made available) from within the ISR (rx_ok_cb) when a valid frame is received, allowing the main loop to proceed with handling the response.

3.4.2 TofResponder Constructor/Dectructor

The TofResponder constructor takes four parameters: uwb_addr src (source address of the specific responder), uwb_addr dst (destination address of the initiator), unsigned long wdt_timeout (watchdog timeout in milliseconds), and DynamicJsonDocument* rx_diagnostics (pointer to a JSON document for receive diagnostics), setting it up with the rx_diagnostics_json member variable.

The constructor sets the type member variable to "Responder", creates a static semaphore (responseSemaphore) for synchronization and initializes it as unavailable. If the semaphore creation fails an error message is printed on the serial monitor.

3.4.3 virtual void setup()

The setup method first calls the TofDevice::setup() method described in 3.2.3 to perform general setup tasks. Then, it configures the Transmit (TX) and Receive (RX) Advanced Encryption Standard (AES) jobs for encryption and decryption tasks, respectively. Subsequently it sets the rx_ok_cb method described in 3.4.6 as a callback function for a UWB message being received using the dwt_setcallbacks function and then configures the DWM3000 to throw an interrupt whenever a good frame is received using the dwt_setinterrupt and port_set_dwic_isr functions. Lastly the method configures additional diagnostics using the dwt_configciadiag allowing channel state informations being extracted after each received frame.

3.4.4 virtual void loop()

The loop method is the main operational loop for the TOF responder device. It begins by invoking the TofDevice::loop() method to reset the watchdog timer. Subsequently, it activates reception immediately using dwt_rxenable(DWT_START_RX_IMMEDIATE). The method then waits for a semaphore signal (responseSemaphore) to indicate an active response, and upon receiving the signal, it proceeds to process the received frame as follows.

Within the response processing section, the method checks if the received frame matches the expected poll from the TOF initiator. If the message can not be decypted correctly, for example when the poll message is destined at another responder device, an error message is thrown on the serial monitor and the function returns immediately.

If the check passes, the message is put in the rx_buffer and a response message is assembled by getting the timestamp when the poll was received (poll_rx_ts) and calculating a time when the response is estimated to be sent (resp_tx_ts). This requires the POLL_RX_TO_RESP_TX_DLY_UUS parameter to be well tuned like explained in chapter 7 as well as the antenna delay also being added.

Then the message is assembled by putting the two times poll_rx_ts and resp_tx_ts and generating the MAC frame. With that being done the AES is configured and the message is encrypted. When encrypting is done the UWB message is sent using the dwt_starttx function and its return value is examined if any errors occurred. If so, the device will enter a loop, otherwise the corresponding registers are cleared and the update_rx_diagnostics (referenced in 3.4.5) function is called.

3.4.5 void update_rx_diagnostics()

The update_rx_diagnostics method is responsible for updating the receiver diagnostics information. It extracts relevant diagnostic data from the DWM3000 module and updates a JSON document (rx_diagnostics_json) with this information. The diagnostic data includes parameters such as reception times, status, phase of arrival, power levels, frequency-related information, and other accumulated counts.

The method employs the dwt_readdiagnostics function to retrieve the relevant diagnostic data. It then populates the corresponding fields in the JSON document with the retrieved values. This updated diagnostic information can be utilized for monitoring and analysis of the TOF responder's receiver performance.

3.4.6 static void rx_ok_cb(const dwt_cb_data_t *cb_data)

The rx_ok_cb method is a static interrupt service routine (ISR) for handling the event of a successful frame reception. This callback is registered with the DW3000 IC and is invoked when a good frame is received. Its primary purpose is to release a FreeRTOS semaphore (responseSemaphore) to indicate that an active response is available for further processing in the main loop.

Upon receiving a good frame, the method calls xSemaphoreGiveFromISR to release the semaphore. This allows the main loop, which is potentially waiting for a response, to proceed with processing the received frame. Additionally, the method checks whether a higher-priority task needs to be woken up, and if so, it requests a context switch using portYIELD_FROM_ISR.

3.5 Watchdog Handling

To ensure that the UWB task on all devices keeps being up and running to deliver data for continuing to estimate the tag's position and is not interrupted or halted in an infinite loop a watchdog logic is implied in the TofDevice class. The watchdog is implemented in the "watchdog.h" and "watchdog.cpp" files. The timer used is given by FreeRTOS xTimer functionality.

3.5.1 Watchdog Variables

private Variables

- **unsigned long timeoutMillis**
  Holds the duration in milliseconds after the watchdog will fire.
- **TimerHandle_t timer**
  Holds FreeRTOS timer handle of type TimerHandle_t.

3.5.2 Watchdog Constructor

When constructing a Watchdog object the constructor takes in the timeoutMillis variable of type unsigned long and assigns it.

3.5.3 void resetTimer()

The resetTimer Method takes in no parameters and resets the timer object of type TimerHandle_t to 0 using the FreeRTOS xTimerReset function.

3.5.4 void begin()

The begin Method creates the timer of type TimerHandle_t using the FreeRTOS xTimer-Create fuction. It assigns the name "WatchdogTimer", the timeout period and the timer-Callback method as a callback dunction that is fired once the designated time is elapsed. Then it is checked if the timer creation was successful and the timer is started using FreeRTOS function xTimerStart. If the timer creation as unsuccessful an error message will be outputted via UART.

3.5.5 void stop()

The stop method checks if a timer object is present. If so it is stopped and deleted by the FreeRTOS functions xTimerStop and xTimerDelete.

3.5.6 unsigned long get_timeout()

The get_timeout method, when called, acts as a getter function and returns the current timeout period saved in the timeoutMillis variable of type unsigned long.

3.5.7 static void timerCallback(TimerHandle_t xTimer)

The timerCallback function is called every time duration in timeoutMillis is elapsed. When called the function prints a serial message that the watchdog timer has elapsed and then restarts the esp device using the esp_restart function.

4 EKF Handling

4.1 Estimation Process

In the realm of EKF handling, the primary objective is to estimate the current position coordinates (X, Y, Z). The EKF_Task, exclusively running on the Tag, initializes an instance of the EKF_Filter class. During initialization, it reads anchor positions from EEPROM and stores them in the static landmarkPositions matrix.

The EKF unfolds in two steps: first, estimating the subsequent state vector, and second, the correction step. To commence estimation, a state vector and covariance matrix are created. The Jacobian matrix of the prediction model and the noise covariance matrix are essential for this step. The Jacobian matrix matJacobF initializes as a 3x3 identity matrix (4.1), suitable for predicting constant velocity movements. The noise covariance matrix matQ initializes as shown in (4.2).

$$\text{matJacobF} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

$$\text{matQ} = \begin{pmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{pmatrix} \tag{4.2}$$

The predictEKF method is then invoked for predicting the next state. It accepts the Jacobian and noise covariance matrices along with a callback for a custom prediction model (4.2.9). The measurement vector vecZ is subsequently created, gathering previously measured distances from the TOF_Task.
Preparatory to the correction step, the measurement noise covariance matrix matR is established (4.3). The Jacobian matrix matHj is calculated using the calculateJacobian-Matrix method (4.2.10).

$$\text{matR} = \begin{pmatrix} 0.01 & 0 & 0 & 0 & 0 \\ 0 & 0.01 & 0 & 0 & 0 \\ 0 & 0 & 0.01 & 0 & 0 \\ 0 & 0 & 0 & 0.01 & 0 \\ 0 & 0 & 0 & 0 & 0.01 \end{pmatrix} \tag{4.3}$$

The Extended Kalman Filter correction step uses the correctEkf method (4.2.7) to refine the previously taken estimation.

4.2 EKF_Filter Class

4.2.1 EKF_Filter Variables

Private Variables

- **Matrix<double, DIM_X, 1> m_vecX**
  The m_vecX variable is a 1x3 column vector that holds the X-, Y- and Z-Coordinates of the Tag that the EKF estimated in the most recent calculation. It is initialized as a zero vector.

- **Matrix<double, DIM_X, DIM_X> m_matP**
  The m_matP variable is a 3x3 matrix and represents the state covariance matrix of the last estimation.

- **static uint32_t lastTime**
  The lastTime a static variable is representing the timestamp in milliseconds of the last update. It is used to calculate the time difference between consecutive predictions in the Extended Kalman Filter.

- **static Matrix<double, DIM_X, 1> x_kminus1**
  x_kminus1 is a static variable representing the predicted state vector from the previous time step in the Extended Kalman Filter. It is a key element in the EKF's prediction step, storing the predicted state to be used as the initial estimate for the subsequent prediction. The variable is updated during each prediction to reflect the evolving state of the system.

- **static Matrix<double, NUM_LANDMARKS, 3> landmarkPositions**
  landmarkPositions is a static matrix representing the positions of landmarks in a three-dimensional space. It is of type Matrix<double, NUM_LANDMARKS, 3> and is used within the Extended Kalman Filter implementation to calculate the Jacobian matrix and measurement vector for the measurement model. The matrix is initialized to a zero matrix and later filled with landmark positions read from EEPROM in the read_landmarks_from_eeprom function described in 4.2.8.

4.2.2 Matrix<double, DIM_X, 1>& vecX()

vecX() is a getter function that returns the current m_vecX matrix.

4.2.3 Matrix<double, DIM_X, DIM_X>& matP()

matP() is a getter function that returns the current m_matP matrix.

4.2.4 predict

The predict method performs the prediction step of the filter. It takes as input the state transition matrix (matF) and the process noise covariance matrix (matQ). The prediction

step for the state vector (m_vecX) is cone by multiplying it with the state transition matrix. The presiction of the state covariance matrix (m_matP) is done by multiplying it with the state transition matrix and the transpose of the multiplying it with the state transition matrix then adding the process noise covariance matrix.
The method updates the estimated state vector (m_vecX) and the state covariance matrix (m_matP) based on the provided matrices, representing the prediction of the system's state in the absence of new measurements.

4.2.5 correct

In the correct method . Parameters are the measurement vector (vecZ) represents the actual sensor measurements, the measurement noise covariance matrix (matR) that characterizes the uncertainty in these measurements, and the measurement Jacobian matrix (matH) relates the state vector to the measurement space.

The method computes the innovation covariance (matSk), Kalman Gain (matKk), and updates the state vector and covariance matrix based on the difference between the predicted and actual measurements. This correction step refines the state estimate by assimilating new information from sensor measurements while accounting for measurement uncertainties.

4.2.6 predictEkf

The predictEkf method is almost identical to the predict method. It differs from it in this respect that it does not take the actual measurement vector vecZ but rather uses the predictionModel methodn to predict an vecZ from the state vector m_vecX.

4.2.7 correctEkf

The correctEkf method is almost identical to the correct method. It differs from it in this respect that it does not take the actual measurement vector vecZ but rather uses the predictionModel method to predict an vecZ from the state vector m_vecX.

4.2.8 read_landmarks_from_eeprom

The read_landmarks_from_eeprom method reads landmark positions from EEPROM and populates the landmarkPositions matrix. It iterates over the landmarks, calculates the EEPROM addresses, and retrieves the stored x, y, and z coordinates. The method is part of the EKF initialization process, ensuring the filter has access to landmark positions stored in non-volatile memory.

4.2.9 predictionModel

The predictionModel method is a component of the Extended Kalman Filter (EKF). It predicts the next state of the system based on the current state and a linear motion

model. As input it takes the current state vector and outputs a new state vector, representing the predicted state of the system.

When called he method calculates the time difference dt in seconds between the current state to be estimated and the state that was last estimated. This time difference is crucial for predicting the evolution of the system. The time is obtained by using the millis function.

The new position of the system is estimated linearly. The method takes the three-dimensional directional vector from the position before the last estimated vector to the last estimated vector. It multiplies this directional vector by the elapsed time dt to obtain the prediction for the new state vector. This assumes a simple linear motion model where the system's position changes linearly with time.

Lastly he method updates internal variables, such as the last recorded time and transition vector, to ensure accurate time tracking.

4.2.10 calculateJacobianMatrix

The calculateJacobianMatrix method, as its name implies, is responsible for computing the Jacobian matrix (matHj) used in the correction step of the Extended Kalman Filter (EKF). When operating in three dimensions with the utilization of five anchors, the dimension of matHj is 5x3.

Upon initialization, all values of matHj are set to zero. Subsequently, for each landmark, the offset of the tag's position in each coordinate direction (dx, dy, and dz) is calculated using the latest state vector. These offsets are then utilized to compute the Euclidean distance to each landmark, denoted as range and computed as $\text{range}_n = \sqrt{(dx_n)^2 + (dy_n)^2 + (dz_n)^2}$.

The resulting matHj matrix is populated following the structure shown in equation 4.4:

$$\text{matHj} = \begin{pmatrix} \frac{-\text{dx}_1}{\text{range}_1} & \frac{-\text{dy}_1}{\text{range}_1} & \frac{-\text{dz}_1}{\text{range}_1} \\ \frac{-\text{dx}_2}{\text{range}_2} & \frac{-\text{dy}_2}{\text{range}_2} & \frac{-\text{dz}_2}{\text{range}_2} \\ \vdots & \vdots & \vdots \\ \frac{-\text{dx}_n}{\text{range}_n} & \frac{-\text{dy}_n}{\text{range}_n} & \frac{-\text{dz}_n}{\text{range}_n} \end{pmatrix} \tag{4.4}$$

4.2.11 calculateMeasurement

The calculateMeasurement function, akin to calculateJacobianMatrix, also contributes to the correction step within the Extended Kalman Filter. Its role involves taking the current state vector, which includes the last outputted state vector coordinates, and determining the offset in the X-, Y-, and Z-directions. Subsequently, it calculates the Euclidean distance to every landmark.

Initiating the process, the function sets up variables and matrices for storing calculated values. For each landmark, it computes the offset of the tag's position in each

coordinate direction (dx, dy, and dz) using the current state vector. The Euclidean distance (range) to each landmark is then computed based on these offsets: $\text{range}_n = \sqrt{(dx_n)^2 + (dy_n)^2 + (dz_n)^2}$. Moving forward, the calculated Euclidean distances undergo a transformation. They are divided by the corresponding measured distances to each specific landmark, and the result is multiplied by (-1). The final output is organized into a 1x5 column vector and returned like shown in equation 4.5.

$$\text{measurement} = \begin{pmatrix} \text{range}_1 \\ \text{range}_2 \\ \vdots \\ \text{range}_n \end{pmatrix} \tag{4.5}$$

5 Bluetooth Handling

In this project, Bluetooth Low Energy (BLE) connectivity is utilized to enable the programming of new anchor values onto the tag device from a desktop application, while also facilitating the reception and visualization of the tag's current position. To achieve this, the project includes the "NimBLEDevice.h" header file, which is crucial for configuring and controlling the BLE functionality. It offers important functions and macros for configuring the BLE device, such as specifying the device's name, appearance, and advertising settings.

The structure and behavior of the BLE server used in this project are defined in the "ble-server.h" and "ble-server.cpp" files. These files play a pivotal role in managing the communication and interaction between the tag device and the desktop software, allowing for the seamless exchange of anchor values and real-time positioning data over the BLE connection.

Since the data to be transmitted only affects the tag, it is the only device in the setup to host an BLE server.

5.1 Server structure

In a Bluetooth Low Energy (BLE) server, services are discrete components that house related data and functionality, and each service is composed of one or more characteristics, each representing a specific aspect of that service. In this particular project, the tag device's server is configured with two distinct services: an input service and an output service.

The input service is designated for transferring data between the tag device and an external computer running UWB desktop software, enabling data input from the computer to the device, while the output service facilitates the opposite data flow, allowing the tag device to communicate data back to the computer. This separation of services streamlines the bidirectional data exchange process, effectively handling data transfer in both directions between the tag device and the external computer, enhancing the project's functionality and versatility.

Input Service

The input service serves as a crucial configuration tool for the tag device, allowing the adjustment of various parameters, particularly the positions of multiple anchors. This service is uniquely identified by the UUID "76847a0a-2748-4fda-bcd7-74425f0e4a10" and comprises two essential characteristics.

The DEVICE_POSITION characteristic is responsible for defining anchor positions, represented as strings that contain the x, y, and z coordinates. This characteristic acts as a means to input and update the spatial information of each anchor, facilitating precise location setup.

The SAVE_CONFIG characteristic is employed to initiate the saving process, which is responsible for persistently storing all updated anchor positions in the tag device's EEPROM (Electrically Erasable Programmable Read-Only Memory). This ensures that configuration changes to anchor positions are retained, even after power cycles, maintaining the device's accuracy and functionality.

Output Service

The output service plays a vital role in transmitting both the currently stored anchor positions and the tag device's own estimated position to the UWB configuration desktop software, which acts as the configuration device. This service is uniquely identified by the UUID "76847a0a-2748-4fda-bcd7-74425f0e4a20" and, similar to the input service, comprises two key characteristics.

The ANCHOR_POSITIONS characteristic is responsible for encapsulating all the presently saved anchor positions. This data is structured as a JSON file and serialized into a string. The process involves iterating through the locally stored landmarks and populating the JSON object with this information, along with their corresponding device IDs. This characteristic enables the transmission of anchor positions for configuration or reference.

The OWN_POSITION characteristic holds the tag device's own estimated X, Y, and Z coordinates. These coordinates are bundled into a string format and transmitted. This feature allows the tag device to relay its real-time positional information to the configuration device, which can be valuable for various location-based applications.

5.2 BleServer Class

The BleServer class represents a Bluetooth server, responsible for initializing the Bluetooth device, managing services and characteristics, and handling data communication with connected devices. It organizes BLE services and characteristics using dynamic memory management, allowing for flexibility and extension. The class facilitates advertising, enables data reading and sending, and provides information about the number of connected devices.

5.2.1 BleServer Variables

Private Variables

- **BLEServer *pServer**
  pServer is a private member variable in the BleServer class, holding a pointer to a BLEServer object. This object represents the Bluetooth server for managing BLE

communication on an ESP32 device. Throughout the class, pServer is utilized to create services, manage characteristics, and interact with the Bluetooth server, enabling the establishment of BLE communication.

- **BLEAdvertising *pAdvertising**
  The pAdvertising variable is a pointer to an instance of the BLEAdvertising class. This object is responsible for managing advertising settings and data for the Bluetooth server. It is utilized during the initialization process to configure advertising intervals, associate service UUIDs, and start the advertising process.

- **std::list<BLEService *> mServices**
  The mServices variable is a linked list that holds pointers to instances of the BLEService class. It is used to manage and store the Bluetooth services created during the initialization process. By maintaining a list of services, the server can easily iterate through them for various operations, such as adding characteristics or starting the services.

- **struct Characteristic**
  he Characteristic struct encapsulates information about a Bluetooth characteristic, including its human-readable name, characteristic UUID, and descriptor UUID. This struct is utilized within the BleServer class to organize and represent individual characteristics, contributing to the modular and well-structured design of the Bluetooth server. By providing a clear structure for characteristic properties, it facilitates the creation and management of Bluetooth characteristics within the BLE server implementation.

- **struct Service**
  The Service struct defines a Bluetooth service within the BleServer class, encapsulating a service UUID and an array of characteristics. This struct aids in organizing and representing Bluetooth services, contributing to the modular architecture of the BLE server implementation. By using the Service struct, the code achieves a clear and structured representation of Bluetooth services and their associated characteristics.

- **const std::array<Service, 2> my_services**
  The my_services constant is an array with a fixed size of 2, where each element is of type Service struct. This array represents the services associated with the Bluetooth server and their respective characteristics. It is initalized already with two services that are the inout and output services discussed in 5.1.

5.2.2 void init_server()

The init_server method initializes the Bluetooth server. First it creates a BLEDevice object of the NimBLE library naming it ESP32. Then it creates a server using the createServer method of BLEDevice and puts the pointer to this server in the pServer

variable. After that the init_services described in 5.2.3 method is called. Then an the pAdvertising pointer to the advertising object of the BLEDevice object is set using the getAdvertising method. The advertising is then configured by adding the UUIDs of the input and output services using the addServiceUUID method and the advertising interval uding the setMinPreferred and setMaxPreferred methods. Subsequently the advertising is started by calling the BLEDevice method startAdvertising and a message is put on the serial monitor indicating that the server is now active and can be found by other Bluetooth devices nearby.

5.2.3 void init_services()

The init_services method initializes all Bluetooth services by creating instances of the BLEService class and associating them with their respective characteristics. It iterates through the my_services array described in 5.2.1 and creates a service for each element of type "Service" adding the respective characteristics for that service.

5.2.4 std::string read_value(const std::string uuid)

The read_value method reads and retrieves the current value of a specified Bluetooth characteristic identified by its UUID. It iterates through the list of services, locates the desired characteristic using its UUID, and returns the characteristic's value as a std::string. If the specified characteristic is not found, it prints a debug message to the Serial Monitor. This function facilitates the retrieval of information from specific BLE characteristics during the server's operation.

5.2.5 void send_value(std::string uuid, const std::string data)

The send_value method updates the value of a specified Bluetooth characteristic identified by its UUID and notifies connected devices about the change. It locates the desired characteristic within the list of services, sets its value to the provided data, and triggers a notification to inform connected devices about the update. If the specified characteristic is not found, it prints a debug message to the Serial Monitor.

5.2.6 size_t getConnectedCount()

The getConnectedCount method retrieves the number of devices currently connected to the Bluetooth server. It uses the getConnectedCount method of the BLEServer object to determine the count of connected devices.

5.2.7 void add_Characteristic(BLEService *service, BleServer::Characteristic characteristic)

The add_Characteristic function adds a new characteristic with a specified UUID to a given Bluetooth service that it both takes in as arguments. It utilizes the createCharacteristic method of the BLEService object to create a new characteristic and associates

it with the provided UUID. Additionally, a descriptor is created for the characteristic, allowing for additional information to be attached.

5.3 BleConfigLoader Class

The BleConfigLoader class serves as a fundamental component in the tag device, responsible for initializing and managing the server structure previously discussed in Section 5.1. During its initialization, it activates the server and periodically broadcasts its presence at intervals defined by the constants BLE_MIN_INTERVAL and BLE_MAX_INTERVAL in the ble-server.h file.

Within this class, there are methods for extracting anchor position information from incoming BLE characteristics, saving this data locally, and writing it to the device's EEPROM for persistent storage. Additionally, it includes methods for sending individual characteristics when called. For a more comprehensive understanding of these methods, please consult the DOXYGEN Documentation.

In the main.cpp file, an instance of the BleConfigLoader object is created, and its methods are invoked to enable the tag device's Bluetooth functionality. The actual invocation occurs within the BLE_Task, as detailed in Section 2.5.3. This class encapsulates the core BLE configuration and communication logic, facilitating seamless operation and data exchange in the tag device.

5.3.1 BleConfigLoader Variables

Private Variables

- **BleServer my_server**
  The my_server object is an instance of the BleServer class referenced in 5.2. It is created in the BleConfigLoader class and serves as a Bluetooth server for managing BLE communication. The my_server object is used to initialize the BLE server, handle configuration settings, and facilitate communication between devices over Bluetooth Low Energy.

- **coordinate landmarkAddresses[NUM_LANDMARKS]**
  The landmarkAddresses is an array of coordinate structures in the BleConfigLoader class, with a size specified by NUM_LANDMARKS. Each element in the array represents the coordinates as a double value of a landmark in a three-dimensional space. The array is used to store and manage the positions of landmarks, which can be loaded from EEPROM, updated via BLE communication, and saved back to EEPROM.

5.3.2 BleConfigLoader Construktor

The BleConfigLoader creates an instance of the BleConfigLoader class and calls the init_server method of the my_server object of type BleServer. The method is cescribed

in 5.2.2.

5.3.3 void save_config_to_eeprom()

The save_config_to_eeprom method is responsible for storing configuration settings, particularly landmark coordinates, into the EEPROM. It iterates through the array of landmark coordinates, calculates EEPROM addresses for each coordinate, and writes the X, Y, and Z values separately to the allocated addresses. This function enables the preservation of configuration data across power cycles, ensuring that landmark positions persist even when the device is restarted.

5.3.4 void load_config_from_eeprom()

The load_config_from_eeprom method retrieves configuration settings, specifically landmark coordinates, from the EEPROM. It iterates through the array of landmarks, calculates EEPROM addresses for each coordinate that are all next to each other, and reads the X, Y, and Z values separately. The coordinates are saved into the landmarkAddresses array.

5.3.5 void save_config_to_ble()

The save_config_to_ble method is responsible for broadcasting the current configuration settings, particularly landmark positions, over Bluetooth. It uses the BLE server (my_server) to create a JSON representation of the landmark coordinates and sends this data to a specific BLE characteristic (BLE_CHARAKTERISTIK_ANCHOR_POSITIONS_UUID). This function allows the desktop application to display the currently saved configuration in the GUI.

5.3.6 uint8_t load_config_from_ble()

The load_config_from_ble method retrieves configuration settings, specifically landmark positions, from another BLE-enabled device running the configuration GUI. It utilizes the BLE server (my_server) to read data from a BLE characteristic (BLE_CHARAKTERISTIK_DEVICE_POSITION_UUID) and interprets the received JSON-formatted data. The function updates the local array of landmark coordinates (landmarkAddresses) with the received values, allowing the BLE configuration loader to synchronize its configuration with the data broadcasted by the GUI. If a "save_config" flag is received over BLE, the function returns 1 initiating the saving process and exiting the configuration mode; otherwise, it returns 0.

5.3.7 void send_position(coordinate own_position)

The send_position method is responsible for transmitting the coordinates of the device's own position over Bluetooth. It uses the BLE server (my_server) to send the current coordinates in a specific format to a designated BLE characteristic

(BLE_CHARAKTERISTIK_OWN_POSITION_UUID). This function allows the GUI to receive and visualize use the position information broadcasted by the tag.

5.3.8 void print_config()

The print_config method prints the loaded configuration settings, specifically the positions of landmarks, to the Serial Monitor for debugging purposes. It iterates through the array of landmark coordinates, creates a JSON representation for each set of coordinates, and prints the information, including the landmark ID and its corresponding coordinates, to the Serial Monitor.

6 MQTT Handling

The MqttClient class is designed for managing MQTT communication in an ESP32-based Arduino environment. It encapsulates the functionality for establishing and maintaining a connection to an MQTT broker, handling incoming messages through a callback function, and publishing messages to specified MQTT topics. The class includes methods for checking the connection status, obtaining the MAC address of the associated WiFi client, and ensuring periodic updates to handle MQTT events. The necessary information to establish the connection are set in the main file.

6.1 MqttClient Class

6.1.1 mqtt Variables

Private Variables

- **WiFiClient espClient**
  The espClient is an instance of the WiFiClient class, which is part of the ESP32 Arduino core libraries. This object is used to manage the WiFi connection on the ESP32 device. The WiFiClient class provides methods for establishing, maintaining, and handling communication over a WiFi network.

  In the context of the MqttClient class, espClient serves as the WiFi client for the ESP32, allowing the device to connect to a WiFi network. It is then utilized by the PubSubClient (MQTT client) to establish and maintain the MQTT connection over the WiFi network.

- **PubSubClient client**
  The client is an instance of the PubSubClient class, which is used to handle MQTT communication on the ESP32. The PubSubClient library provides functionalities for interacting with MQTT brokers, allowing the ESP32 to both publish messages to specific topics and subscribe to topics to receive messages.

  In the context of the MqttClient class, the client object represents the MQTT client used for communication with an MQTT broker. It is associated with the WiFiClient instance (espClient), which manages the underlying WiFi connection. The client object is responsible for connecting to the MQTT broker, handling incoming messages, and publishing messages to specified topics.

- **String dev_id**
  dev_id is a variable of type String, representing the device ID associated with the

MQTT client instance. It is passed as a parameter to the constructor and can be utilized for identification or customization purposes within the MQTT communication.

- **const char* topic**
  topic is a parameter representing the MQTT topic to which the client subscribes and publishes messages. It is passed as a parameter to the constructor when creating an instance of the MqttClient. The topic variable is crucial for specifying the communication channel within the MQTT protocol, allowing the client to subscribe to and publish messages on a particular topic within the MQTT broker.

6.1.2 MqttClient Constructor

The constructor of the MqttClient class initializes the MQTT client instance by setting up the MQTT server details, callback function, and buffer size using the respective methods of the PubSubClient class. It then establishes a WiFi connection using the provided SSID and password. Finally, it calls the reconnect method described in 6.1.7 to connect successfully.

6.1.3 update

The update method ensures the continuous functionality of MQTT communication. It includes a call to the private reconnect method described in 6.1.7, responsible for establishing a connection to the MQTT broker if not already connected. Additionally, the method invokes client.loop() to handle MQTT client events, facilitating the processing of incoming messages and other protocol-related tasks. Regularly calling the update method in the main loop of the program maintains the MQTT client's connectivity and responsiveness to incoming events.

6.1.4 publish

The publish method facilitates the publication of MQTT messages to a specified topic. It takes three parameters: the MQTT topic to publish to (topic), the message to be published (msg), and the length of the message (plength). When invoked, this method attempts to publish the message to the MQTT broker associated with the client. If successful, the message is sent to the specified topic; otherwise, an exception is caught, and an error message is printed to the Serial monitor. The publish method allows for the integration of MQTT message transmission in the application, providing a means to communicate data or commands over the MQTT protocol.

6.1.5 is_connected

The is_connected method checks whether the MQTT client is currently connected to the broker. It returns a boolean value, true if the client is connected and false otherwise. To check the connectio it uses the connected method of the private client object.

6.1.6 get_mac

The get_mac method retrieves the MAC address of the associated WiFi client. It returns the MAC address as a String. This method allows the application to obtain the unique hardware address assigned to the ESP32's WiFi interface. It uses the macAddress method of WiFiClass.

6.1.7 reconnect

The reconnect method handles the process of reconnecting the MQTT client to the broker. If the client is not already connected, this method attempts to establish a connection in a loop. If the connection fails, it disconnects the client and retries the connection every 10 seconds until a successful connection is made. Once connected, it subscribes to the specified MQTT topic.

6.1.8 setup_wifi

The setup_wifi method is responsible for connecting the ESP32 to a WiFi network using the provided SSID and password. It initiates the WiFi connection and includes a loop that waits until the ESP32 successfully connects to the specified WiFi network. During this process, it prints dots to the Serial monitor to indicate the connection attempt. Once connected, it prints the IP address of the ESP32.

6.2 MQTT Functions

6.2.1 subscribe_callback

The subscribe_callback function is a callback handler for MQTT subscriptions in the provided code. It is invoked whenever a message arrives on a subscribed MQTT topic. It serves as an interrupt-like routine for processing incoming messages. The function prints the received MQTT message payload to the Serial monitor, allowing for custom message parsing logic to be implemented based on the application's requirements.

7 Tuneable Parameters

7.1 RNG_DELAY_TOF

RNG_DELAY_TOF is the constant representing the delay between consecutive Time of Flight measurements. The value of RNG_DELAY_TOF directly affects the overall round-trip time for TOF measurements and can be adjusted for system performance tuning based on specific application requirements. Obviously it cannot be set infinitely small because each measurement takes its time to be processed. In performance tests plausible measurement results could be obtained with a RNG_DELAY_TOF down to 30ms.

7.2 POLL_RX_TO_RESP_TX_DLY_UUS

The POLL_RX_TO_RESP_TX_DLY_UUS value is a constant defined in the tof-responder header file used to account for the time it takes for the Responder to process the received "Poll" message and prepare the "Response" message for transmission. This delay could include processing time, computation time, and any other factors that contribute to the time between receiving the "Poll" and sending the "Response."

For example, if POLL_RX_TO_RESP_TX_DLY_UUS is set to 1000 microseconds, it means that the system expects the Responder to start transmitting the "Response" message approximately 1000 microseconds after successfully receiving the "Poll" message.

Altering the code for processing the response requires this value to be adapted accordingly.

7.3 POLL_TX_TO_RESP_RX_DLY_UUS

POLL_TX_TO_RESP_RX_DLY_UUS is a constant defined in the tof-initiator header file, representing the delay between the transmission of a poll message and the expected reception of the corresponding response message in the UWB communication system. This delay is specified in UWB microseconds and is a critical parameter for the proper functioning of the communication protocol. The value of POLL_TX_TO_RESP_RX_DLY_UUS influences the timing synchronization between initiator and responder devices, ensuring that the response is expected at the correct time after the poll transmission.

Adjusting this constant may be necessary based on system requirements and environmental factors to optimize the reliability and accuracy of TOF measurements when altering the UWB sequence.

7.4 RESP_RX_TIMEOUT_UUS

RESP_RX_TIMEOUT_UUS is a constant defined in the tof-initiator header file, representing the timeout duration for receiving a response in the UWB communication system. This timeout is specified in UWB microseconds and is a critical parameter for determining how long the system will wait for a response after the transmission of a poll message.

In the context of the code you provided, the RESP_RX_TIMEOUT_UUS constant is used to set the timeout for receiving a response frame. If a response is not received within this specified timeout duration, the system may consider the operation as unsuccessful or trigger error handling procedures.

The value of RESP_RX_TIMEOUT_UUS is crucial for the proper functioning of the communication protocol, and it needs to be carefully chosen based on the expected time of flight, system latency, and other environmental factors. Adjusting this constant allows for fine-tuning the system's responsiveness and reliability in different operating conditions.