École Polytechnique Fédérale
de Lausanne

Laboratoire de Systèmes Robotiques

Development of an ultra-wide band indoor positioning system

Antoine Albertelli

Supervisors
Daniel Burnier
Eliot Ferragni

Professor
Prof. Franceso Mondada

Fall semester, 2017

This page intentionally left blank.

Development of an ultra-wide band indoor positioning system

Antoine Albertelli

Fall semester 2017

1 Context

Absolute positioning is a required component in mobile robotics. In outdoor applications, the Global Positioning System (GPS) fulfills this goal. However, its short wavelength prevents it from being used indoors. Dead reckoning solutions such as Inertial Motion Units (IMUs) or optic flow suffer from drift over time. The goal of this project is to implement a low-cost indoor positioning solution based on Ultra Wide Band (UWB) modules.

2 Working principle

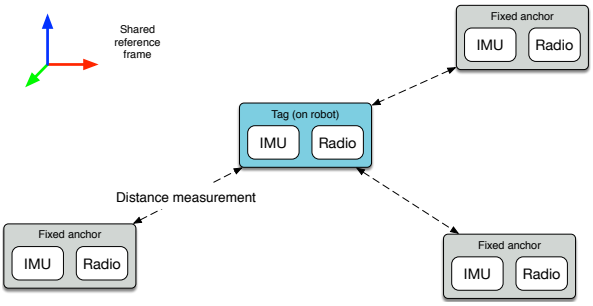


Figure 1: Overview of the system, showing one mobile robot (blue) and three fixed anchors (grey).

The complete system (Figure 1) is made of two types of nodes, tags and anchors. The distance between each node is obtained by measuring the time of flight for a radio packet. The tags are placed on each robot, and estimate their position using the IMU combined with the distance measurements to the anchors. The anchors are at a reference position (fixed) and can be used by several tags. A tag will always use all the available anchors and will be able to seamlessly switch in case a tag is not in range anymore.

3 Tasks

After doing a review of the literature on the topic, I will start by doing a model of the system. This includes both the dynamics of the system and the impact of noise on precision. I should also investigate if the Digital Motion Processor (DMP) unit built in the IMU can be useful. I will then develop a position estimator taking the IMU data as inputs and the distance readings as measurements.

In parallel to this, I will write code for the reference board (Figure 2). Several steps must be done in order to have a working system: 1. Drivers for the UWB module and the IMU chip. 2. Distance measurement protocol. 3. Position estimation algorithm.

I will finish my work by measuring the performance of the system. If time remains, I would like to implement a protocol to communicate over the UWB link, to provide low bandwidth inter-robot communications without additional hardware.

Figure 2: Reference board which includes an STM32F405 processor, a DWM1000 UWB module and an MPU9250 IMU.

4 Results

A working version of the system is implemented. Anchors can communicate with tags and vice versa. The range measurement is accurate (no bias) and has a good precision ($\sigma = 3\,\text{cm}$).

A framework in Python allows for comparison of various Kalman models and tuning. An embedded implementation of the Kalman filter in C++ tracks the position of the tags. Real-world experiments match the precision and accuracy expected from simulated experiments, at least from a qualitative perspective.

The DMP did not match our expecations. Therefore, an attitude determination algorithm (the Madgwick filter) was implemented and shows good static performance (low drift).

Contents

1. Introduction

Precise positioning is one of the key elements of a mobile robot. It allows the software to take decisions and navigate to goals effectively. Therefore, it has been quite extensively studied by roboticists. Nowadays, the most common approaches are Inertial Motion Units (IMUs) or dead reckoning.

However, those approaches tend to suffer from drift over time, due to their integration process. To stay accurate over long period of time, they must be coupled with a reference system to correct the drift. GPS can be used for this role, but does not work indoors due to its frequency range. We wanted to develop a new system that could enable indoor positioning of autonomous robots.

In recent year, Personal Area Network (PAN) radio technologies have received an increasing amount of attention[1], and relevant standards have been appearing, mostly under the IEEE 802.15.4a protocol family. Of particular interest are Ultra Wide Band (UWB) systems, which have a bandwidth of at least 500 MHz or 20% of the center frequency[2]. From a telecommunication point of view, UWB is an interesting technology due to higher range and bandwidth at equivalent power level. However, it also enables new applications, including Time of Flight (ToF) calculation.

We wanted to use this ToF measurement capability to create an indoor positioning system for robots. When looking at examples of positioning system for outdoors, a lot of them used a combination of GPS for global, noisy measurements and IMU (gyroscope, magnetometer and accelerometer) for local position data[3, 4]. Commercial UWB systems have also been used in conjunction with IMU sensors for greater accuracy[5].

1.1. Working principle

The complete system (Figure 1.1) is made of two types of nodes, tags and anchors. The distances between nodes are obtained by measuring the time of flight for a radio wave. The tags are placed on each robot, and estimate their position using the IMU combined with the distance measurements to the anchors. The anchors are at a reference position (fixed) and can be used by several tags. A tag will always use all the available anchors and will be able to seamlessly switch in case a anchors is not in range anymore.

1.2. Requirements

Possible uses of this system include virtual reality and gaming, industrial automation, robotics contest, etc. This work was done for use in robotics contest such as Eurobot or

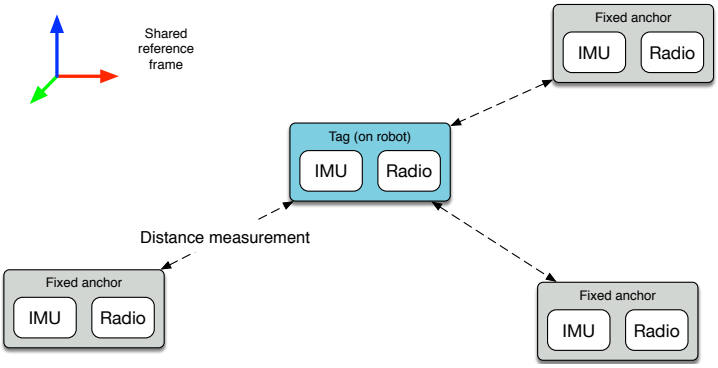


Figure 1.1.: Overview of the system, showing one mobile robot (blue) and three fixed anchors (grey).

EPFL's internal contest. However, we kept in mind the other applications and tried to avoid making choices that would make extension to other applications difficult.

For a robotics contest application, we decided on the following requirements. Those are not to be reached in the framework of this project, but are useful to reason about technical choices.

1. Ability to localize two robots at the same time.
2. At least 3 anchors, uses more if more are available.
3. The robots are moving in the 2D plane, but the anchors must not be in this plane.
4. Reasonable positioning error ($\sigma = 3\,\text{cm}$).
5. Update rate high enough for navigation purposes (about 5 Hz).

2. Hardware

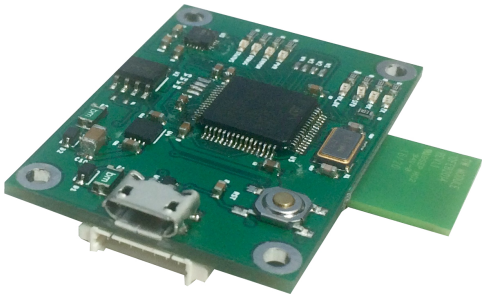


Figure 2.1.: Development board used for this project. The antenna is sticking out the side to avoid interferences with the ground planes.

For this project, we used an Open Source development board (Figure 2.1), which includes all the required components for this project:

- An Invensense MPU9250 IMU.
- A Decawave DW1000 UWB transceiver.
- An STM32F411 microcontroller.
- USB and Controller Area Network (CAN) connectivity.

2.1. Inertial Motion Unit

The MPU9250 is a 9-axis IMU made by Invensense[1]. It was chosen because it combines an accelerometer, a gyroscope and a magnetometer in an easy to use package. It is connected to the main microcontroller through a SPI bus.

An interesting feature of this device is the Digital Motion Processor (DMP). It is a low power co-processor that can be used for a variety of purpose, including orientation estimation, in which it directly outputs a quaternion. Unfortunately, we found out that this feature is poorly documented by the manufacturer, making it impossible to use without reverse engineering. Additionally, the DMP cannot use the magnetometer data. Therefore, we decided not to use it but to implement our own sensor fusion on top of raw sensor information.

[1] `https://www.invensense.com`

2.2. UWB transceiver

The board contains an UWB transceiver which combines a physical interface and a MAC. It has several features useful for our application:

1. IEEE 802.15.4a compliant MAC which can filter incoming frames based on their destination addresses,
2. High range, up to 290 m,
3. High accuracy timestamps for RX and TX of packets,
4. High data rate (up to 6.8 $\mathrm{Mbit\,s^{-1}}$).

3. Positioning algorithm

3.1. Parametric vs. non-parametric filter

State estimation algorithm can be grouped in two major families: parametric and non parametric filters. Parametric filters are filter which take assumptions on the distribution of the state. For example, the Kalman filter makes the assumptions that both the state and the measurement noises are normally distributed. Non-parametric filters, on the other hand, can represent any distribution, and most importantly multi modal ones.

If possible, parametric filters should be used, as they are computationally less expensive. In this project, the code will be running on a microcontroller, therefore it might not be possible to use complex non parametric filters.

Early on the project we decided to use an Extended Kalman Filter (EKF), which is one of the most common state estimator. Like the Kalman filter, which assumes that the distributions of the state and noise are Gaussian. However, it works on non-linear dynamics, by approximating the non linear process through a first order Taylor expansion.

While the EKF is easy to implement and lightweight computationally, it only works if the noises are Gaussian. We can verify this property experimentally for the measurement noise (see Section 5.1), but could only assume it for the process noise.

3.2. Simulation

Testing the model's performance on the real system can be difficult. First, implementing all the models in an embedded language like C or C++ is a tedious task. Then, feeding the algorithm with fixed (recorded or generated) measurements makes the results easier to compare and reproduce. Therefore, we implemented a testing environment in Python.

Python allows us to easily express high level mathematical objects. For example, the Jacobian matrices are automatically computed using a symbolic math toolbox[1], then converted to a numerical version. Theoretically, it should even be possible to generate a C++ implementation from the symbolic representation, although that was not used in this project. The framework allows us to create new EKF models and test their performance in a matter of minutes.

[1] Sympy: `http://www.sympy.org/`

3.3. First model

The goal of this model was to create the simplest thing that could possibly work. It does not account for most issues, but was enough to start work on a simulator for the system.

To simplify the model, the following hypotheses are made:

1. The robot moves in the 2D plane, i.e. its pose is described by (x, y, θ).
2. The IMU's internal motion processor already outputs θ (but it drifts).
3. The IMU outputs the acceleration vector in body frame $\mathbf{a}^b$.
4. The inertial frame is aligned with the world frame, i.e. the robot starts with $\theta = 0$.

The state contains the position and speed, in world frame:

$$\mathbf{x} = \begin{pmatrix} x & y & \theta & \dot{x} & \dot{y} \end{pmatrix}^T \tag{3.1}$$

3.3.1. Motion model

The first step is to transform the acceleration from body frame to world frame.

$$\mathbf{a}^w = R(\theta)\mathbf{a}^b \tag{3.2}$$

The state update equation, using the acceleration as control input

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x} &= \begin{pmatrix} \dot{x} & \dot{y} & 0 & \mathbf{a}_x^w & \mathbf{a}_y^w \end{pmatrix}^T \\ &= \begin{pmatrix} \dot{x} \\ \dot{y} \\ 0 \\ \cos(\theta)\mathbf{a}_x^B - \sin(\theta)\mathbf{a}_y^B \\ \sin(\theta)\mathbf{a}_x^B + \cos(\theta)\mathbf{a}_y^B \end{pmatrix} \end{aligned} \tag{3.3}$$

We can turn it into a discrete state update equation using the forward Euler method:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \Delta_t \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x} \tag{3.4}$$

3.3.2. Measurement model

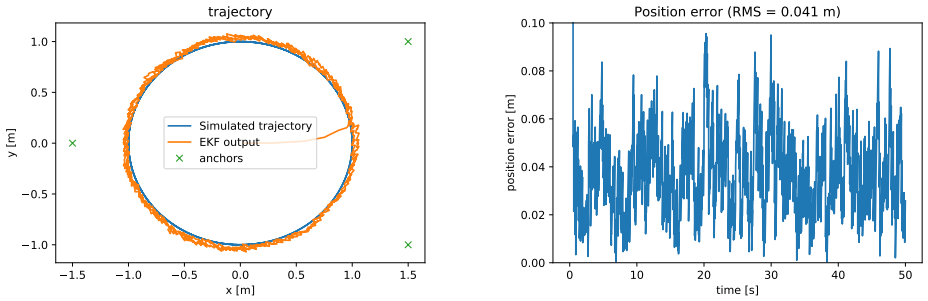
In this model we have two different measurement functions. The first one is the robot's heading, θ, given by the DMP or another attitute estimation algorithm.

$$h_\theta(\mathbf{x}) = \theta \tag{3.5}$$

The second type of measurement is the distance to the UWB anchors. It is actually a family of functions; each beacon defines a measurement function. This function is the distance to the position of the beacon, called $\mathbf{b}$.

$$h_b(\mathbf{x}) = \sqrt{(\mathbf{x}_x - \mathbf{b}_x)^2 + (\mathbf{x}_y - \mathbf{b}_y)^2} \tag{3.6}$$

3.3.3. Simulation results



(a) Trajectory (b) Position error

Figure 3.1.: Simulation results using the simple model. The simulated robot moves at $0.17\,\mathrm{m\,s^{-1}}$. The UWB ranges arrive at 10 Hz, while the accelerometer information was integrated at 200 Hz.

This simple model was converted to an EKF model and implemented using the framework described in Section 3.2. The noise characteristics of the ranging system were measured (see Section 5.1), while the accelerometer noise was taken from the datasheet. The results can be seen in Figure 3.1.

We think this behavior is pretty suboptimal and that the EKF could be better tuned. However, we decided not to investigate this further, as this model required an absolute heading in the shared reference frame. This is not easy to know: the magnetometers can give an absolute heading with respect to the Earth's magnetic frame, but we do not know the rotation between the north and the shared frame. This led us to the creation of the next model.

3.4. UWB only model

This model was developed when I realized that the simple model had two important issues:

1. First, it supposed that we knew the rotation between the world frame and the earth magnetic frame, as the beacons' coordinates are in world frame, but the attitude of the robot is in earth magnetic frame.

2. The model had a lot of different variances to tune, making the tuning hard to do.

This model removes those issues by relying only on the UWB radio ranging for measurement, and no prediction step.

3.4.1. Model

The state contains only the position of the robot.

$$\mathbf{x} = \begin{pmatrix} x & y \end{pmatrix}^T$$

There is no information used for prediction, making the prediction function the identity:

$$\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k, \mathbf{u}_{k+1}) = \mathbf{x}_k \tag{3.7}$$

For the measurement, the UWB system gives us the distance d to a beacon. The beacon's position $\mathbf{b}$ is known and assumed to be fixed. Therefore the measurement model is given by Equation 3.8.

$$h(\mathbf{x}, \mathbf{b}) = \|\mathbf{x} - \mathbf{b}\| \tag{3.8}$$

3.4.2. Tuning

To compute the variance of the model, we estimate that the robot is moving at a constant speed of maximum V_{max} between two measurements updates (which occurs at f_{UWB}). Therefore, the maximum distance that a robot can do is given by Equation 3.9.

$$d = \frac{V_{max}}{f} \tag{3.9}$$

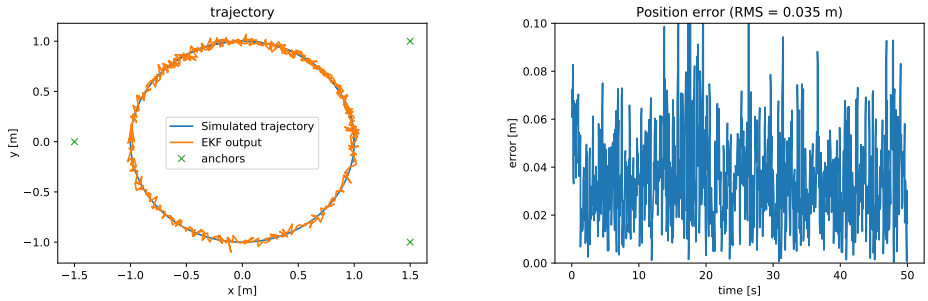
If we assume that $d = 2\sigma$, this means that our hypothesis is valid 97.5% of the time, which is reasonable. Therefore, the process variance can be estimated using Equation 3.10.

$$\sigma^2 = \left(\frac{V_{max}}{2f}\right)^2 \tag{3.10}$$

Assuming an update frequency of 10 Hz and a max speed of $0.14\,\mathrm{m\,s^{-1}}$, typical for a Thymio robot[6], we get a variance of $\sigma^2 = 4.9 \times 10^{-5}\,\mathrm{m}^2$.

3.4.3. Simulation results

The results from the simulation can be seen in Figure 3.2. The simulation parameters were the same as on the previous models to allow for comparison. We can see that this filter has a better RMS error. As it uses less information than our first model, it is probably due to better tuning. As the performance of this model are good enough for our test application, this is the model we decided to implement on the microcontroller.



(a) Trajectory (b) Position error

Figure 3.2.: Simulation results using only UWB range information with 3 beacons. The simulated robot moves at $0.17\,\mathrm{m\,s^{-1}}$. The beacon distances were updated at 10 Hz.

3.5. Differential measurement model

Out of curiosity, we also tried to implement a model where the robot would embed two UWB tags separated by a known distance. This should allow for a better precision, as well as measuring the heading of the robot. We made the following assumptions:

1. Robot moves in the 2D plane
2. Gyro in the yaw axis
3. Gyroscope has constant bias (b_ω) and random noise.
4. UWB receivers are separated by a fixed distance d.

The state vector of the robot contains the position, speed and heading. It also contains a constant term for the gyro bias, used to model the noise on the bias.

$$\mathbf{x} = \begin{pmatrix} x & y & \dot{x} & \dot{y} & \theta & b_\omega \end{pmatrix}^T \tag{3.11}$$

3.5.1. Model

The prediction step uses outputs of the accelerometer (in body frame) and the angular rate as control inputs:

$$\mathbf{u} = \begin{pmatrix} a_x \\ a_y \\ \omega \end{pmatrix} \tag{3.12}$$

The differential equation governing the state evolution is therefore:

$$\dot{\mathbf{x}} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \cos(\theta)a_x - \sin(\theta)a_y \\ \sin(\theta)a_x + \cos(\theta)a_y \\ \omega - b_\omega \\ 0 \end{pmatrix} \tag{3.13}$$

Using forward Euler integration, we get:

$$\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k, \mathbf{u}_{k+1}) = \mathbf{x}_k + \Delta_t \dot{\mathbf{x}} \tag{3.14}$$

For the measurement step, the UWB system gives us the distance d to a beacon. The beacon's position $\mathbf{b}$ is known and assumed to be fixed.

We first compute the position of each UWB receiver in world frame (W). Receiver i is assumed to be at position $\mathbf{x}_i$ in robot frame (R).

$$\mathbf{x}_i^W = \mathbf{x}_{robot}^W + \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \mathbf{x}_i^R \tag{3.15}$$

Then the measurement model is:

$$h_i(\mathbf{x}, \mathbf{b}) = \|\mathbf{x}_i^W - \mathbf{b}\| \tag{3.16}$$

3.5.2. Simulation results

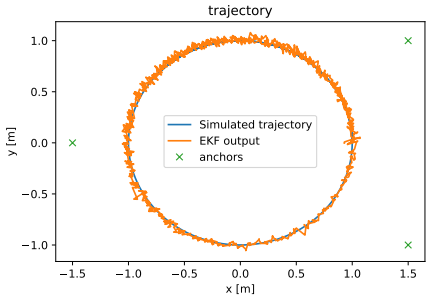
This method was also simulated and the results can be seen in Figure 3.3. To allow for comparison, it uses the same parameters as the previous experiments. We can see that it tracks the angle fairly well and the position has a good performance.

The performance could probably become better with some EKF tuning. We did not want to spend time doing this, as the hardware did not support this configuration. It can however be an interesting extension to allow for better performance and ability to track heading.

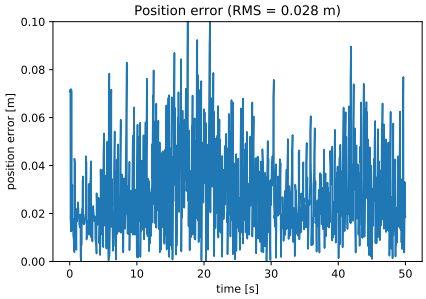
3.6. Attitude estimation

While an attitude estimator could be implemented directly in the EKF, we chose to use a Madgwick estimator instead. This estimator is a relatively novel technique based on gradient descent. It is typically as accurate as an EKF, while being less computationally expensive[7].

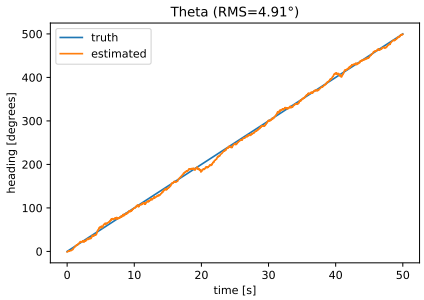
The Madgwick filter takes as inputs the measurements of linear acceleration, angular rates and magnetic fields. It then computes the optimal state estimation, and returns it as a quaternion. We used the reference implementation[7] which was already suitable for microcontroller use.



(a) Trajectory



(b) Position error



(c) Angle estimation

Figure 3.3.: Simulation results using the differential model. The simulated robot moves at $0.17\,\mathrm{m\,s^{-1}}$. The UWB ranges arrive at 10 Hz, while the accelerometer information was integrated at 200 Hz.

4. Implementation

4.1. UWB ranging protocol

Although Decawave provides a description of the protocol[8], the details are still left to the implementor. Therefore, we had to design our own implementation from their description.

4.1.1. Design principles

1. The protocol should require as little state as possible to be stored on the participating beacons. State synchronization between several systems is a common source of error and bugs.

2. We want to maximize the measurement frequency, which means we should have a measurement simple with as little packets as possible.

3. To avoid waking up the microcontroller, the MAC layer integrated in the UWB module must be leveraged as much as possible.

4. The same radio link used for range measurement will also be used as a general-purpose link to transmit data between nodes.

4.1.2. Time measurement

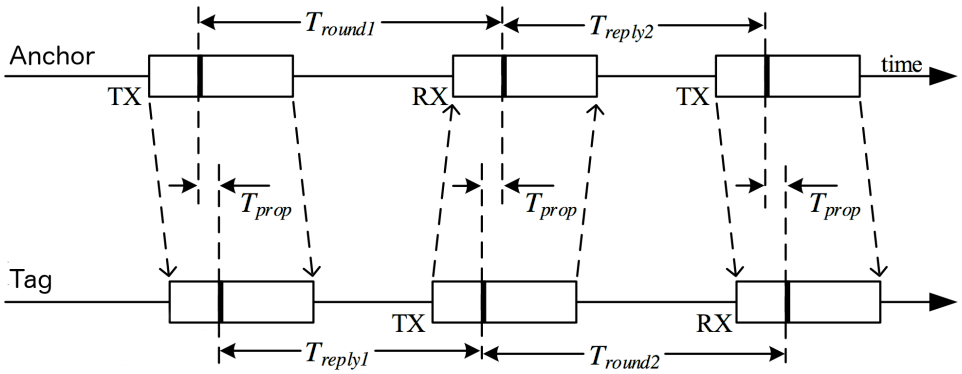


Figure 4.1.: Messages in a time measurement interaction. Source: [8]

The time measurement protocol is composed of a sequence of 3 messages. To initiate a measurement sequence, an anchor broadcasts a measurement advertisement message to

the broadcast address (0xFFFF). Tags may respond with a measurement reply message. Finally, the anchor sends a measurement finalization message and the tag computes the ranging solution:

$$T_{prop} = \frac{T_{round,1} \cdot T_{round,2} - T_{reply,1} \cdot T_{reply,2}}{T_{round,1} + T_{round,2} + T_{reply,1} + T_{reply,2}} \tag{4.1}$$

The timing information is then converted to a distance by multiplying by the speed of light (c).

This 3-way ranging scheme works because transceivers can be programmed to send a frame at a given time. Therefore, $T_{reply,2}$ can be included in the final message.

Measurement advertisement A measurement advertisement message has a sequence number of 0. It is sent by an anchor to the broadcast address (0xFFFF). It only contains the TX timestamp as a 40 bit unsigned integer ($T_{tx,1}$).

Measurement reply A measurement reply message has a sequence number of 1. It is sent by a tag and contains the timestamps of all events so far ($T_{tx,1}$, $T_{rx,1}$ and $T_{tx,2}$)

Measurement finalization Finally, the measurement finalization has a sequence number of 2. It is sent by the anchor and contains the content of the previous message plus the new timestamps ($T_{rx,2}$, $T_{tx,3}$).

Once the tag received the last message, it can compute the ranging solution using Equation 4.1 with the following information:

$$\begin{aligned} T_{round,1} &= T_{tx,1} - T_{rx,2} \\ T_{reply,1} &= T_{rx,1} - T_{tx,2} \\ T_{round,2} &= T_{tx,2} - T_{rx,3} \\ T_{reply,2} &= T_{rx,2} - T_{tx,3} \end{aligned}$$

Here are some notes regarding the implementation:

1. The chosen scheme is compatible with the use of the IEEE 802.15.4a MAC hardware packet filters.
2. The sequence number field is rather used to store the type of the packet rather than an always increasing sequence number.
3. All data in messages are sent in network byte order (most significant byte first).
4. All timestamps in messages are in DW1000 units and referenced to the internal clock (no shared reference).

4.1.3. Antenna delay calibration

In order to obtain correct range readings, we should take in account the delay created by antenna transmission. This delay should be calibrated as it will be different for each design [8]. To do it, we simply measured the distance reported by the modules over a known range (averaged on N samples). The antenna delay (Δ_T) can then be computed by equation 4.2.

$$\Delta_T = \frac{d_{measured} - d_{real}}{2 \cdot c} \tag{4.2}$$

On our system, this value was found to be about 32840 DW1000's internal units (513 ns).

4.1.4. Optimal rate calculation

When using a radio link such as our UWB modules, collisions can occur if two devices try to transmit at the same time. If this happens, at least one of the two transmitted packets will be lost. As the protocol does not include retransmission logic to stay lightweight, this means the whole ranging sequence will drop. This means that increasing the ranging rate past a certain point will *decrease* the rate of successful range transactions, as collisions will occur more frequently. This effect can be seen on Figure 4.2.

To study the probability of having collisions, we can use the theory from queuing systems[9]. Queuing systems are useful to study phenomenon where customers arrive at a certain rate λ, and are served at a rate μ, or equivalently, use the ressources for a time $\frac{1}{\mu}$. In our system, the ressource would be the access to the shared medium, λ the number of packets per seconds sent by the nodes and $\frac{1}{\mu}$ the average time it takes for one full measurement sequence to occur.

We will model our system by a so called "M/M/1" queue, this means that the emission of packets is Markovian (memoryless), the transmission time is Markovian as well, and only 1 packet can be processed at a time. Therefore, the probability of a packet transmission by a node during an interval t is given by the memoryless law:

$$f(t) = \lambda e^{-\lambda t} \tag{4.3}$$

From M/M/1, we know that $p_n = (1-\rho)\rho^n$ is the probability of having n ranging transactions in flight at the same time, where $\rho = \frac{\lambda}{\mu}$ is the load factor of the medium. We are interested in the probability of having $n > 1$, i.e. a collision:

$$\begin{aligned} p(n>1) &= 1 - p(n=0) - p(n=1) && (4.4)\\ &= 1 - (1-\rho)(1+\rho) && (4.5) \end{aligned}$$

The rate of successful measurements (λ') is given by the rate of measurement attemps multiplied by the success rate.

$$\lambda' = \lambda \cdot (1 - p(n>1)) \tag{4.6}$$

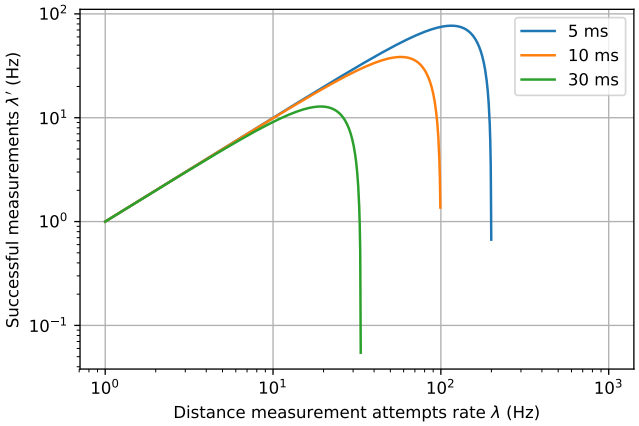


Figure 4.2.: Measurement rates for various values of $\frac{1}{\mu}$. We see that past a certain point, adding measurement attempts per seconds results in too many collisions, decreasing the overall measurement rate. The current implementation needs about 30 ms to complete a measurement. This means the optimal rate of $\lambda = 19.3\,\text{Hz}$, resulting in a succesful rate of 12.8 Hz.

4.2. Software architecture

Continuing on the principles used on the ePuck 2, the software is constructed around a message passing bus (Figure 4.3). This allowed us to validate every subsystem independently during development. It also ensures the software stays loosely coupled: the implementations can be changed without any impact on the rest of the system, as long as the published messages are the same. In addition to the data processing pipeline, a generic parameter service allows modules to define settings which are automatically exposed over USB and CAN, and are stored to flash memory for persistency across resets.

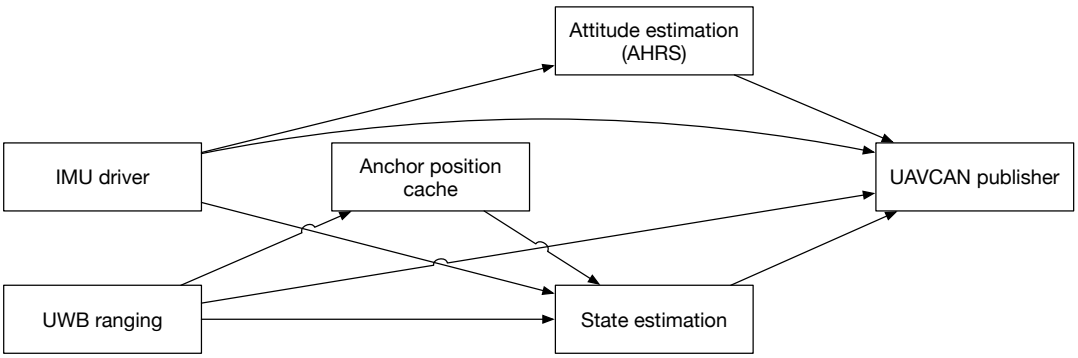


Figure 4.3.: Software architecture. The arrows represent message bus topics. Not shown here is the command shell, which can poke at almost every topic to inspect the data.

While the console is convenient for debugging, a more structured way of communication is required for data logging and integration into a real robot. We used the CAN bus integrated on the board with the UAVCAN[1] protocol on top. UAVCAN is a high-level protocol which defines standard way of encoding messages of various types and different semantics (broadcast or request/response). The messages are defined in `.uavcan` files, from which C++ code is automatically generated. The standard also defines a few conventions including a way to access a device's parameters, and provides a Graphical User Interface (GUI) to interact with the bus. We used a simple USB-to-CAN converter[2] to connect our PCs to the board.

[1] `http://uavcan.org/`

[2] `http://www.cvra.ch/technologies/can_dongle.html`

5. Results

5.1. Range measurement

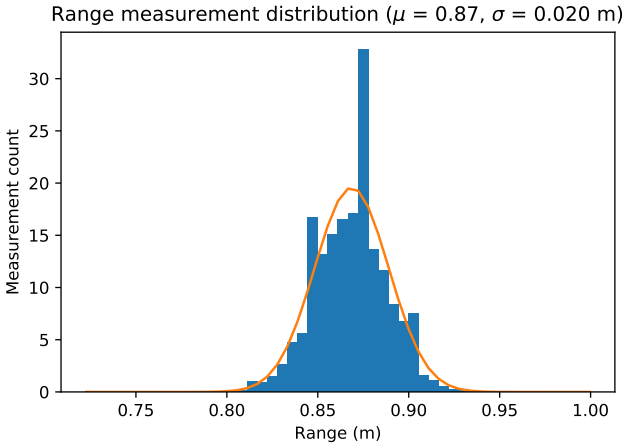


Figure 5.1.: Distribution of the range measured by the UWB module in the optimal orientation. The corresponding gaussian fit is shown. We see that the error is pretty much normally distributed.

The Kalman filter relies on the assumptions that the measurement noise is distributed according to a normal law. As we were not familiar with UWB ranging technologies, we needed to make sure that this was the case. Once the ranging protocol was implemented and working, we separated two boards by a known distance and logged the measured ranges for half an hour. The distribution of the results can be seen in Figure 5.1.

We tried to use a statistical normality tests to determine if the samples were from a normal distribution. However, those tests can only reject the hypothesis that the samples come from the normal distribution and are almost never significant on large sample count. We plotted the distribution that fitted the dataset and decided that the distribution was close enough to a Gaussian for the requirements of the EKF. Although some forum posts described increased noise when running coplanar antennas (instead of parallel), we could not reproduce this effect. It seems that any antenna orientation could work effectively.

This experiment also allowed us to confirm that the antenna delay was properly calibrated since the dataset did not exhibit a bias.

5.2. Attitude determination

As we ended up not using the results from the attitude determination, we did not do a deep evaluation of the performance of the Madgwick filter. The dynamic performances seemed pretty good when testing the algorithm "by hand". To test the algorithm's static drift, we fixed it to a table and let it run until stable, then measured the drift over a 5-minute interval. We found the drift to be very good (Figure 5.2), achieving drifts of less than one degree per hour.

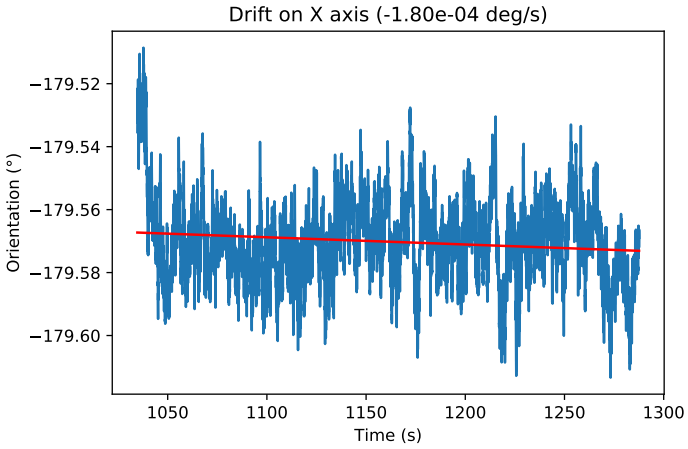


Figure 5.2.: Static drift of Madgwick's algorithm. Only the worst performing axis is show here.

Overall, the performance of Madgwick's algorithm seemed pretty convincing. This, coupled with the ease of use and tuning means that this algorithm can be a very good general purpose attitude estimator. Compared to the DMP, it also does not restrict the choice of sensors.

5.3. Positioning algorithm

The positioning system is implemented and works in the minimal configuration (one tag and two anchors). Real-world experiments confirm the precision and accuracy expected from simulated experiments, at least from a qualitative perspective. We lacked the time to do a proper quantitative analysis of the performance, as well as study the dynamics of the system.

6. Conclusion & Future work

This project was an interesting excursion in the land of UWB systems. A significant amount of time was spent on the design of the state estimator and of the communication system. Unfortunately, a lot of time was also spent writing and debugging simple low-level drivers for the various chips involved.

The current system is already operational and provides an interesting experimental platform. The complete measurement stack (ranging, attitude determination and state estimation) are implemented. Loose coupling with the message bus allow for easy swapping of one component for experimentation. However, the performance of the system could be greatly improved in a second revision of this project.

The development of the beacons will continue as an Open Source project on Github[1]. It could be interesting to add one semester project to enhance the current implementation. Here are some topics that could be covered.

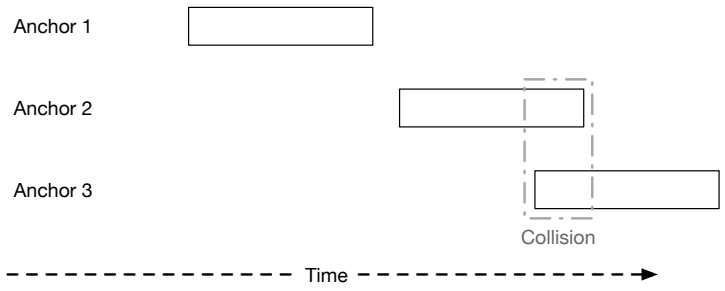
6.1. Radio protocol enhancements

One of the weakest points of the current implementation is the protocol used to share the radio medium. As the current implementation only relies on probabilities to handle collision, its performances are very poor (Section 4.1.4). Using a better protocol to handle interleaved packets could result in a much better media utilization, up to 1742 message (570 ranging) per second[8].

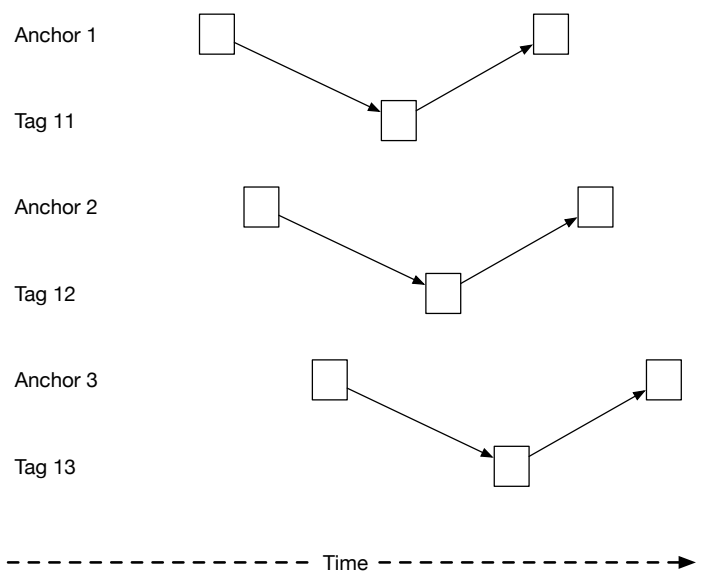
To achieve this level of performance, replacing the broadcast measurement advertisements is needed. The current implementation means that a ranging sequence eventually locks up the whole media while it is completing (Figure 6.1a). This leads to low air utilization, as no transaction can occur even when no messages are being transmitted (during CPU processing). Switching to unicast addressing (Figure 6.1b) would allow other tag and anchors to communicate during processing.

For even greater performance the Time Division Multiple Access (TDMA) protocol could be used. In TDMA, each node is assigned a specific time slot by the coordinator. This allows for very high air utilization: using the fastest data rate and payloads of 12 bytes, each packet takes 103 µs to transmit[8], resulting in about 9700 packets per seconds. However it is very complex to implement, as it requires a perfect synchronization between nodes.

[1] `https://github.com/cvra`



(a) Broadcast (current implementation)



(b) Unicast (proposed change)

Figure 6.1.: Comparison between broadcast measurement advertisement and unicast measurement advertisements.

6.2. Kalman filter enhancements

As discussed in Chapter 3, we implemented the simplest possible Kalman filter. The performance of the system could probably benefit from a better model, carefully tuned to the sensors fitted on board. In addition to the other algorithms described in this report, other simple models could be interesting. For example, adding a constant speed hypothesis to the EKF could probably result in a lower RMS error.

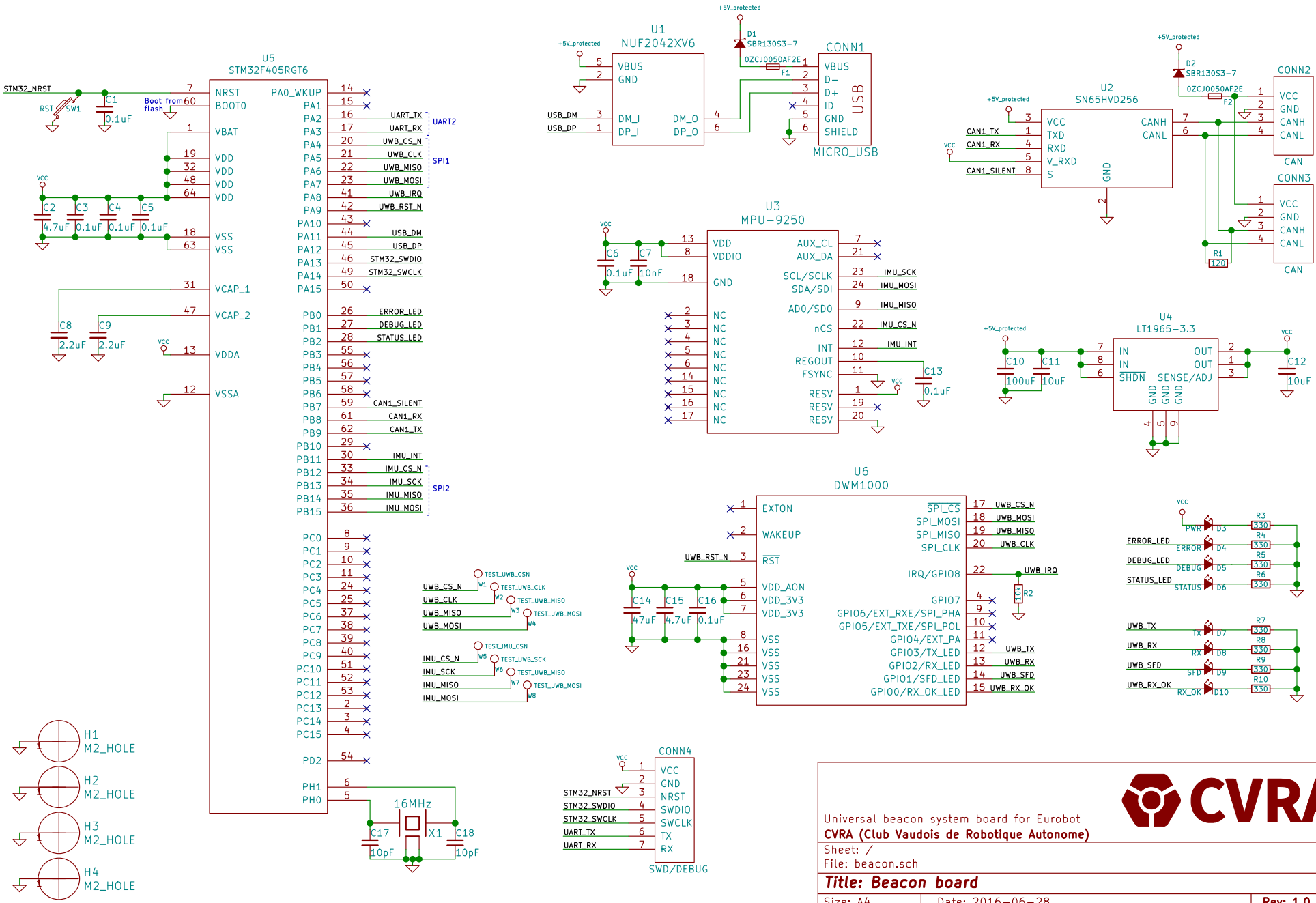
A. Hardware schematic

The following schematic is the one of the board used during this project. However, several flaws were discovered and should be corrected before a new batch is produced.

- There is currently no way to use the USB bootloader built in the microcontroller. This makes updating the board's firmware harder, as it requires a specialized debug adapter. Adding a button on pin BOOT0 would allow the user to enter bootloading mode as well as serving as a general-purpose button.
- The reset pin of the UWB module is connected to a pin used by the USB module of the microcontroller. Although one should be able to disable this pin by software, we could not make it work and had to cut a connection on the board.

All design documents and related issues can be found on the board's Github repository[1].

[1] `https://github.com/cvra/uwb-beacon-board/`


U5
STM32F405RGT6
U1
NUF2042XV6
CONN1
MICRO_USB
U2
SN65HVD256
CONN2
CONN3
CAN
U3
MPU-9250
U4
LT1965-3.3
U6
DWM1000
CONN4
SWD/DEBUG
16MHz
H1
M2_HOLE
H2
M2_HOLE
H3
M2_HOLE
H4
M2_HOLE
Universal beacon system board for Eurobot
CVRA (Club Vaudois de Robotique Autonome)
Sheet: /
File: beacon.sch
Title: Beacon board
Size: A4
Date: 2016-06-28
Rev: 1.0
KiCad E.D.A. kicad 4.1.0-alpha+201606241232+694645ubuntu14.04.1-product
1/1

B. User manual

B.1. Requirements

- A version of arm-none-eabi-gcc compatible with C++11 (Tested with GCC 5.4.1)
- Python 3.5 or greater
- Packager[1], a program used to manage dependencies
- PyUAVCAN[2], which contains the code generator for UAVCAN
- OpenOCD 0.9.0 or greater
- CMake and CppUTest[3] for running the unit tests.

B.2. Quickstart

```
# Updates submodules if the repo was not cloned with --recursive
git submodule update --init --recursive

# Generates the build files
packager

# Creates C++ code from UAVCAN message definitions
make dsdlc

# Builds the project
make

# Uploads the project to target board
make flash
```

B.3. Running unit tests

```
# Creates an empty directory to put the build files
mkdir -p build && cd build

# Configures the build environment
```

[1] https://github.com/cvra/packager
[2] https://github.com/UAVCAN/pyuavcan
[3] http://cpputest.github.io/

```
cmake ..

# Compiles and runs the tests
make check
```

If everything goes well, you should see a message saying something like `OK (174 tests, 174 ran, 1302 checks, 0 ignored, 0 filtered out, 53 ms)`. If there are failures, the relevant line and files will be shown, so you can check them out.

B.4. Board configuration

Once a board has been flashed, they must still be configured. This can be done either using the USB or CAN interfaces (see below). Changes done in one interface are automatically reflected on the other.

The minimum amount of configuration required is to give the board a unique MAC address, indicate wether it is an anchor or not, and give it a fixed position if it is an anchor.

B.4.1. Via USB

The microUSB connection on the board emulates a serial port. You can connect to it using your favorite serial terminal emulator. The baudrate does not matter, as the simulated connection will accept any baudrate. For example, using PySerial[4]:

```
# The name of the port might be different on your computer!
python -m serial.tools.miniterm /dev/ttyUSB0 115200
```

Once connected, you can display the current parameters by running `config_tree`. You can change parameters using `config_set`:

```
# Boolean parameters can be "true" or "false"
config_set /uwb/anchor/is_anchor false

# Scalar and integers parameters
config_set /uwb/pan_id 42
```

Once you are satisfied with the settings, store them to non-volatile memory using `config_save`. Otherwise, they will be lost on reboot. To clear non-volatile memory (which restores the default values for all parameters), run `config_erase` and reboot the device.

[4] `https://pythonhosted.org/pyserial/`

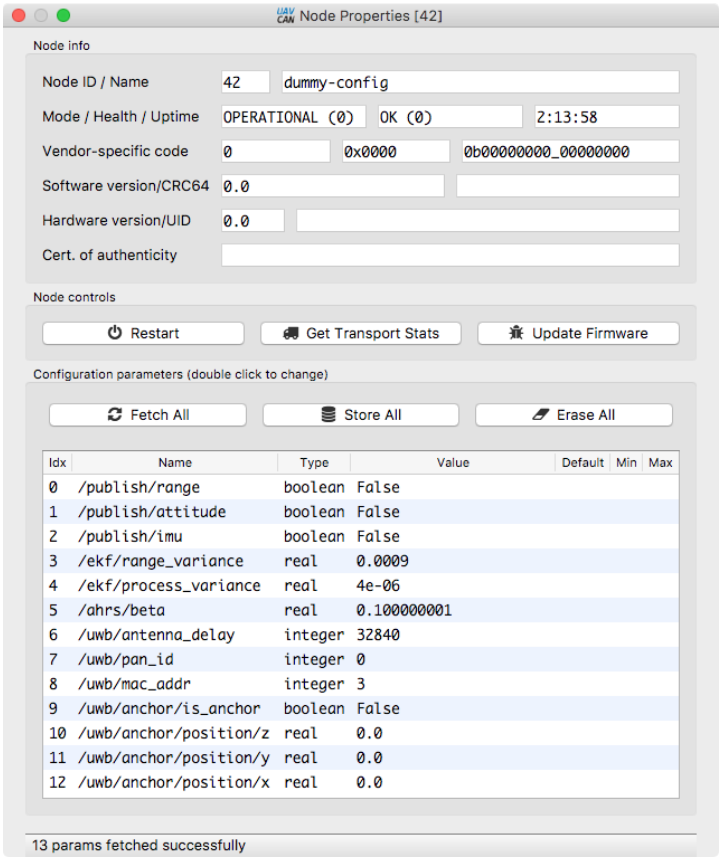


Figure B.1.: UAVCAN parameter GUI

B.4.2. Via CAN

If you have setup UAVCAN correctly, you should be able to run `uavcan_gui_tool`, which is a graphical tool used to interact with UAVCAN devices. Please refer to the tool's documentation[5] for details. Once this is running, you can open the properties of a node by double clicking on it. You should see a screen similar to the one on Figure B.1.

On this screen, you can double click on a parameter to change its value. Do not forget to click the "Send" button to apply your changes.

Once you are happy with your settings, save them to non-volatile memory by clicking the "Store All" button. Clicking the "Erase all" button will clear the non-volatile memory, restoring all parameters to default values on next reboot.

B.4.3. Parameter description

This is the parameter tree, along with default values.

```
publish: # Settings related to the publication of data over UAVCAN
  range: false # Publish raw range measurements
```

[5] `http://uavcan.org/GUI_Tool/Overview/`

```
  attitude: false # Publish attitude quaternion
  imu: false # Publish raw IMU measurements
ekf: # Settings related to the Kalman filter
  range_variance: 0.0009 # Variance of the range measurement
  process_variance: 0.000004 #
ahrs: # Parameters for the attitude (AHRS) algorithm
  beta: 0.1 # Madgwick gain
uwb: # Parameters for the radio module
  antenna_delay: 32840 # Antenna delay, calibrated for this board
  pan_id: 0 # PAN (network) ID. All boards must be in the same PAN
  mac_addr: 0 # Mac address. Must be unique per board in a given PAN
  anchor:
    is_anchor: false # Indicates if this board is a beacon or an anchor
    # Position of this anchor in the shared frame.
    # Ignored if is_anchor is false.
    position:
      z: 0.
      y: 0.
      x: 0.
```

C. Code source organization

The code of this project can be separated into three different categories: libraries, portable code and platform specific code. Libraries are Open Source code that are re-used from other projects. Portable code is generic code that does not contain anything specific to this processor or board, but is only useful in this project. It can be moved into a library if another project requires it later. Portable code is usually well covered by unit tests (see the `tests` folder), where non portable code must usually be tested "by hand".

Libraries

Most of those libraries have READMEs that describe them in much more detail.

- `chibios-syscalls` provides a few extensions for ChibiOS (our RTOS) for malloc and printf mostly.
- `crc` implements some common checksums algorithms.
- `decadriver` is provided by Decawave to interact with their product.
- `eigen` is a C++ matrix manipulation library used for the EKF.
- `parameter` provides a unified interface for the application to declare parameters.
- `cmp`, `cmp_mem_access` and `parameter_flash_storage` are used to store the parameters in non volatile memory.
- `msgbus` implements a software message bus (publisher/subscriber pattern), similar in concept to D-bus or ROS.
- `libuavcan` is the official UAVCAN stack.
- `test-runner` is used only to provide an environment during unit testing.

Portable code

- `MadgwickAHRS.{c,h}` contains the implementation of the Madgwick filter.
- `lru_cache.{c,h}` contains a generic implementation of a Least Recently Used (LRU) cache.
- `mpu9250.{c,h}` contains a portable implementation of a driver for the Invensense MPU9250 IMU.
- `state_estimation.{cpp,h}` contains the code for the state estimator (uses ekf.hpp).
- `ekf.hpp` contains a generic implementation of an EKF using C++ templates.
- `uwb_protocol.{c,h}` contains a portable implementation of the UWB protocol.

Non portable code

- `ahrs_thread.{c,h}` contains the thread that runs the Madgwick filter.
- `anchor_position_cache.{c,h}` contains the thread that stores the position of the anchors and provides them to the other threads.
- `imu_thread.{c,h}` contains the thread that runs the IMU driver.
- `ranging_thread.{c,h}` contains the thread responsible for doing range measurements and other radio operations.
- `state_estimation_thread.{cpp,h}` contains the thread that runs the state estimation.
- `usbconf.{c,h}` contains the USB setup code.
- `main.{c,h}` contains the entrypoint of the application.
- `cmd.{c,h}` contains the command line interface shown on USB.
- `decawave_interface.{c,h}` contains the low level bindings between Decawave's library and our code.
- `board.{c,h}` contains GPIO contains GPIO configuration.
- `bootloader_config.{c,h}` contains code for interacting with the CAN bootloader used on some boards.
- `exti.{c,h}` contains the external interrupt configuration and handlers.
- The `uavcan` folder contains code related to moving messages from the application to the CAN bus and vice versa.

Bibliography

[1] M.-G. Di Benedetto, *UWB communication systems: a comprehensive overview*, vol. 5. Hindawi Publishing Corporation, 2006.

[2] ITU Radiocommunication Assembly, *Characteristics of ultra-wideband technology*, 2006.

[3] M. Roobek, "Motion tracking in field sports using GPS and IMU," Master's thesis, 2017.

[4] D. Titterton and J. L. Weston, *Strapdown inertial navigation technology*, vol. 17. IET, 2004.

[5] J. A. Corrales, F. Candelas, and F. Torres, "Hybrid tracking of human operators using IMU/UWB data fusion by a kalman filter," in *Human-Robot Interaction (HRI), 2008 3rd ACM/IEEE International Conference on*, pp. 193–200, IEEE, 2008.

[6] "Thymio II web site." `https://www.thymio.org/`, 2017.

[7] S. O. Madgwick, A. J. Harrison, and R. Vaidyanathan, "Estimation of IMU and MARG orientation using a gradient descent algorithm," in *Rehabilitation Robotics (ICORR), 2011 IEEE International Conference on*, pp. 1–7, IEEE, 2011.

[8] DecaWave Ltd, *DW1000 User Manual*, 2015. Ver. 2.05.

[9] M.-O. Hongler, *A Guided Tour in Stochastic Processes for Engineers*. 2017.

[10] A. M. Sabatini, "Quaternion-based extended kalman filter for determining orientation by inertial and magnetic sensing," *IEEE Transactions on Biomedical Engineering*, vol. 53, no. 7, pp. 1346–1356, 2006.

[11] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT press, 2005.

[12] S. J. Julier and J. K. Uhlmann, "A new extension of the Kalman filter to nonlinear systems," in *Int. symp. aerospace/defense sensing, simul. and controls*, vol. 3, pp. 182–193, Orlando, FL, 1997.