UAVCAN-enabled indoor postioning solution for unmanned vehicles

Riccardo Miccini
Technical University of Denmark
Centre for Bachelor of Engineering Studies

June 24, 2016

Abstract

Usual abstract stuff.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This introductory chapter will provide background information about Real-time Locating Systems (in short — and throughout the rest of this document — *RTLS*) and the current state of the art, along with a brief description of the partner company, the project description under the form of assignment formulation, and a brief overview of the implementation. The latter will be elaborated further in the following chapters.

1.1 Rationale

This section will strive to explain the importance and relevance of a UAVCAN-based indoor real-time location system, by presenting its two main underlying concepts.

1.1.1 RTLS

From map-making to post tracking, localization has been a fundamental human endeavor which encompassed several ages and civilizations. Although GPS has proven to be a major leap is this regard, its current limitations causes it to be unsuitable for some purposes. First of all, the accuracy lays in the order of a few meters at best, mainly due to the relativistic implication of orbiting satellites [1] (that is without considering more advanced techniques such as *RTK*). Moreover, signal shadowing and multi-path reflections renders it particularly unreliable indoor or in dense urban areas. In order to fulfill the ever-growing demand for automation, more precise location systems have to be employed.

The term RTLS emerged from such needs, and characterizes systems capable of reckoning, tracking, and showing the position of moving objects and persons within a relative frame of reference, mainly a building or enclosed area. The applications benefiting from those systems mainly lay within the logistical/operational areas, and the interested parties range from manufacturing industries to health care operators (management of flows of goods in a warehouse or production plant; assets retrieval e.g. tools, medical equipment, cattle; surveillance and monitoring; personnel deployment and supervision). Consumer-related applications are also starting to arise in the fields of

domotics and electronic devices.

The technologies driving these systems include ultrasounds, infrared radiation, or RSSI (Received signal strength indication) from various sources such as radio frequency modules, WiFi/WLAN, or even Bluetooth; however, each of them presents its own limitations which prevented them from being used in large scale. [2]

The more recent implementation of previously known technologies into positioning systems spawned a vast array of previously unforeseen applications, such as their use along with drones and other unmanned vehicles for maintenance and inspection of facilities, emergency situation management, and whenever human intervention may pose a safety threat.

1.1.2 UAVCAN

Although drones, copters, rovers, and the like are are experiencing a huge rise in popularity in both commercial and hobby use, integrators and final users often have to cope with a vast amount of different protocols and interfaces, some of which directly inherited from the RC modeling world, thus lacking fault tolerance and scalability. For years, the automotive industry has relied on CAN Bus as a means of communication between different microcontrollers and devices, due to its low cost and resilience against electro-magnetic resilience.

UAVCAN is a lightweight open source protocol aimed to provide fast and reliable communication within drone peripherals. [3] A UAVCAN network is composed of a series of nodes which are independent from the others and require no master/controller or host computer. The protocol sits on top of the physical layer of CAN Bus and relies on a reference software stack for easy integration with the user application code. Common standard high-level functions like node health monitoring, network discovery, time synchronization, and firmware update are predefined and taken care of by the library, while message definitions for most drone equipment (speed controllers, navigation data, gimbals, actuators) are standardized and publicly available, making it easy for manufacturers to engineer compliant devices and for integrators to interface with them.

1.2 Company description

UAVComponents ApS is a danish company which develops and supplies components, equipment, and tools for drones and UAVs integrators and hobbyists. Its main products are the *TSLRS* line of transceivers and the *Aeronav One* ground control station. Along with off-the-shelf hardware solutions, the company offers its expertise to clients seeking consultancy or ad-hoc projects. UAVComponents shares its offices and personnel with *Danish Aviation Systems*, a sister company also operating in the drones industry.

The team is composed of roughly eight people, almost all of whom are involved in Research&Development. Besides a solid know-how of piloted and autonomous aerial vehicles, the core competencies include the electronic design of analogic and MCU-based solutions, and software development — both for embedded and computer systems. The company relies on external support for

more skill-intensive tasks such as mechanical and RF design. Manufacture is also mostly outsourced, especially for high volume yields.

As the vast majority of the employees is quite young, the company can boast a very dynamic and informal atmosphere, which in turn resulted in flexible working hours and a high degree of collaboration among all the colleagues.

1.3 Problem formulation

The aim of the project is to design and develop a UAVCAN-compatible indoor positioning system for aerial unmanned veicles (UAVs) based on the *decaWave* devices. Using these radio modules, an algorithm shall be devised and implemented in order to estimate the UAV position in a 2D coordinate system. The resulting information is to be broadcasted through the UAVCAN bus — a message definition shall be chosen amongst the pool of standard definitions, or designed from scratch.

The performance of the system ought to be in line with decaWave's claims in term of maximum accuracy and range, and the device has to be suitable for battery operation. Moreover, the system shall comply with the UAVCAN specifications and support its major standard functionalities.

The project is carried out in collaboration with *UAVComponents ApS*, which will lend its expertise in hardware design and provide the necessary development tools.

1.4 System description

The system is composed of one (or more) *tags* and up to four *anchors*.

The anchors are to be spread across the area of operation, and should all intersect the same horizontal plane. Their coordinates within the frame of reference shall be know beforehand. The tag device is located inside the drone/vehicle and is connected through the UAVCAN bus.

Both type of devices share the same hardware design, but are flashed with different firmwares. It comprises:

- ARM Cortex-M4 microcontroller
- decaWave DWM1000 module
- Inertial measurement unit
- Barometric sensor
- Power supply unit
- CAN transceiver and connectors
- USB interface
- SWD connector for debugging and flashing

The boards can be powered through either the USB or the UAVCAN connectors.

The software infrastructure governing both firmwares is built upon *ChibiOS*, a real-time operating system for embedded devices. Each device — regardless of its role — can be configured through a command line interface accessible through the USB.

Each anchor runs the same software routine, but with different parameters (coordinates and transmission delay). Their role is to respond to the polling message sent by the tags.

The tag firmware is responsible for initiating new communications and processing it (in a way that is thoroughly described in the Theoretical principles). Whenever a new response message is parsed and the distance between the tag and that anchor calculated, the position estimation is updated. Lastly, the location broadcaster transmit the current estimation to the UAVCAN bus, in an asynchronous fashion.

Chapter 2

Problem analysis

The following sections aim to elaborate on the Problem formulation and create a more solid base to use as a guideline and template during the actual development. The adopted framework is based upon the one presented during the course of the studies [4] and bears similarities with the *Unified Software Development Process*.

This analysis does not aspire to be exhaustive, but rather satisfactory in the level of detail required to establish the project. Moreover — in order to avoid resorting to a priori assumption, the term *decaWave* is hereby used to denote the product (IC or module) rather than the company.

2.1 Requirements specification

The *software requirements specification* is the formal description of the system to be developed, thoroughly describing what the product is — and is not — expected to do. It also helps assessing the extent of the endeavor in terms of workload, cost, and other resources. The requirements are commonly laid out in agreement with the client, and provide the basis upon which the product or project is evaluated.

As far as this project is concerned, the company has not set any particular requirements apart from those implied in the project name. The list of requirements is therefore mostly based on former knowledge, common sense and best practices within the industry,

2.1.1 Functional requirements

This section defines specific behaviors or functionalities, and is the basis for the tests. Requirements introduced by the modal verb *shall* indicate mandatory conditions, whilst those introduced by the modal verb *may* indicate optional features.

For conciseness and clarity sake, along with tag and anchor, the concept of *device*is also introduced. Requirements specified for the device are to be fulfilled by both elements.

ID	Description
SRS.F.HW.1	The tag shall include a UAVCAN-compliant interface
SRS.F.HW.2	The tag shall be able to be powered through CAN
SRS.F.HW.3	The device shall be able to be powered through USB
SRS.F.HW.4	The tag shall provide visual feedback for errors and critical states
SRS.F.HW.5	The tag shall provide visual feedback for CAN activity
SRS.F.HW.5	The tag shall be able to fit inside a typical copter drone
SRS.F.SW.1	The tag shall be able to notify its presence and status in the UAVCAN bus
SRS.F.SW.2	The tag shall be able to obtain a node ID from a dynamic node ID allocator server
SRS.F.SW.3	The tag shall be able to respond to node discovery requests
SRS.F.SW.4	The tag shall be able to reboot when requested by another node
SRS.F.SW.5	The tag shall be able to synchronize its time with a UAVCAN master clock node
SRS.F.SW.6	The tag shall be able to emit debugging information and logs, if requested
SRS.F.SW.7	The tag shall be able to be configured through UAVCAN
SRS.F.SW.8	The tag shall be able to discover anchors in the proximity
SRS.F.SW.9	The tag shall be able to periodically poll the anchors for new ranging samples
SRS.F.SW.10	The tag shall be able to update its location estimation when a new sample is available
SRS.F.SW.11	The tag shall be able to publish its location data through the UAVCAN bus
SRS.F.SW.12	The anchors shall be able to respond to tag discovery request
SRS.F.SW.12	The anchors shall be able to respond to tag ranging requests
SRS.F.SW.12	The device shall be able to be configured through USB
SRS.F.SW.12	The device shall be able to be configured through USB
SRS.F.SW.13	The tag may be able to update its firmware through the UAVCAN bus
SRS.F.SW.14	The tag may use the onboard sensors to refine the location data
SRS.F.SW.14	The device may implement IEEE 802.15.4 frame format for communication

Table 2.1: Software requirements specification: functional requirements

2.1.2 Non functional requirements

This section specifies desired characteristics and qualities of the system.

ID	Description
SRS.NF.SW.1	The source shall be writte inn C++
SRS.NF.SW.2	The source shall strive towards modularity
SRS.NF.SW.3	The source shall not allocate or make us heap-based memory
SRS.NF.SW.4	The source shall follow a consistent format style
SRS.NF.SW.5	The source shall be concisely yet extensively documented
SRS.NF.SW.6	The released documentation shall be written in English
SRS.NF.SW.7	The source may be supported by other toolchains and compilers

Table 2.2: Software requirements specification: non-functional requirements

2.1.3 Use cases

This section describes how the actors can interact with the system.

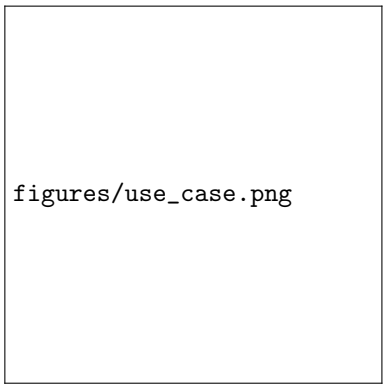

Figure 2.1: CAN Bus: bit sampling

2.2 Theoretical principles

This section will present the theoretical aspects of the technologies employed in the project, and provides a knowledge base for better understanding the problem and the employed solution.

2.2.1 Ultra-wideband

Ultra-wideband (UWB in short) is a radio technology that use a very low energy level for short-range, high-bandwidth communications over a large portion of the radio spectrum.

Although this technique has been know for for decades, is a relatively new entry in the RTLS field [5], with the first affordable development kits having entered the market in the last few years. It gained immediate popularity due to its advantages: blablabla

2.2.2 Ranging algorithm

An overview of different ranging techniques is hereby presented and discussed. The techniques analyzed here mainly strive to calculate the most accurate possible *time of flight* (TOF), which can then be converted into distance with the following trivial equation:

$$distance = ToF \times c \tag{2.1}$$

The devices involved in the process will be referred to as initiator (**I**) and responder (**R**). Furthermore, only solutions that can cope with unsynchronized (i.e. not sharing the same base clock) devices are taken into consideration.

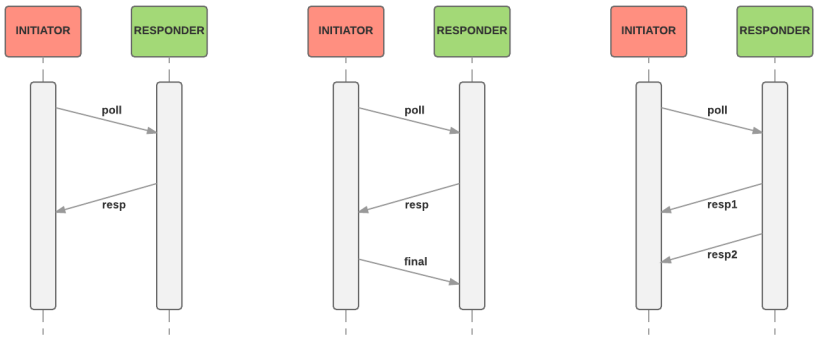

Figure 2.2: Two-way ranging: different approaches

Single-sided 2WR

This is the most straightforward approach:

1. **I** sends poll message and notes time $T_{poll_{TX}}$
2. **R** receives poll message and notes time $T_{poll_{RX}}$
3. After an arbitrary time, **R** sends response message and notes time $T_{resp_{TX}}$
4. **I** receives response message and notes time $T_{resp_{RX}}$

Assuming that **R** includes all its recorded data into the response message, **I** can assume that:

$$ToF = \frac{(T_{resp_{RX}} - T_{poll_{TX}}) - (T_{resp_{TX}} - T_{poll_{RX}})}{2} \tag{2.2}$$

One of the major shortcomings of this scheme is that clock frequency differences between the two devices will cause vast drifts over the tiniest delay.

Double-sided 2WR

This method improves upon the previous ones by trying to compensate for the introduced drift:

1. **I** sends poll message and notes time $T_{poll_{TX}}$
2. **R** receives poll message and notes time $T_{poll_{RX}}$
3. After an arbitrary time, **R** sends response message and notes time $T_{resp_{TX}}$ as well as $T_{reply1} = T_{resp_{TX}} - T_{poll_{RX}}$
4. **I** receives response message and notes time $T_{resp_{RX}}$ as well as $T_{round1} = T_{resp_{RX}} - T_{poll_{TX}}$
5. After an arbitrary time, **I** sends final message and notes time $T_{final_{TX}}$ as well as $T_{reply2} = T_{final_{TX}} - T_{resp_{RX}}$

6. **R** receives response message and notes time $T_{final_{RX}}$ as well as $T_{round2} = T_{final_{RX}} - T_{resp_{TX}}$

Assuming that **R** includes the calculated time intervals in the response message, **I** can assume that:

$$ToF = \frac{T_{round1} \times T_{round2} - T_{reply1} \times T_{reply2}}{T_{round1} + T_{round2} + T_{reply1} + T_{reply2}} \tag{2.3}$$

Although this method is known to be more reliable than the previous, it requires that ranging measurements are transmitted through the radio, which depending on the application may not be desirable.

Single-sided 2WR with drift compensation

This method provides drift correction whilst maintaining the range computation on the initializer:

1. **I** sends poll message and notes time $T_{poll_{TX}}$
2. **R** receives poll message
3. After an arbitrary time, **R** sends response message
4. **I** receives response message and notes time $T_{resp1_{RX}}$
5. After the same amount of time, **R** sends another response message
6. **I** receives response message and notes time $T_{resp2_{RX}}$

In this way **I** need not use any data based on other clocks. The equation becomes:

$$ToF = \frac{(T_{resp2_{RX}} - T_{poll_{TX}}) - 2 \times (T_{resp2_{RX}} - T_{resp1_{TX}})}{2} \tag{2.4}$$

2.2.3 Trilateration

Trilateration consists in determining the location of an object by measuring its distance from reference points. With the advent of electronic means of measuring distances, this technique found widespread usage in the fields of geolocalization, navigation, and surveying. In particular, GPS technology leverages on trilateration for providing its services.

The process of trilateration can be easily visualized in two dimensions by considering the circle projected by the set of points equidistant from a given reference. The radius of such circle is the distance between an anchor whose location is known (the center of the circle) and a tag whose location is to be found.

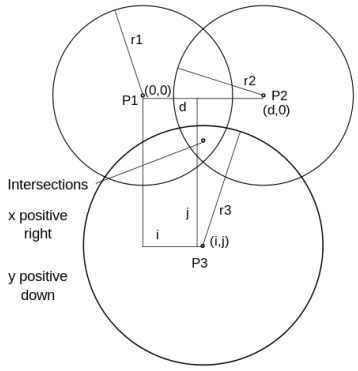

Figure 2.3: Trilateration: two-dimensional example []

Adding another anchor and projecting its distance from the tag brings down the possible location to two points. In the ideal case, a third anchor would project a circle intercepting precisely in one of these points. However, this is rarely the case and may simply be unfeasible in case of a moving object (as the application of this project implies). Trilateration: two-dimensional example [] shows however that an educated estimate can still be achieved.

2.2.4 CAN Bus

A *Controller Area Network* (CAN) bus is a standard designed to allow microcontrollers, sensors, and other devices to communicate with each other without a host computer. It was designed in 1983 by *Robert Bosch GmbH* for automotive usage, but eventually found applications in many other contexts.

A CAN network is made up of nodes connected to each other through a two-wire bus. The wires form a differential pair of 120 Ω nominal impedance. Each node is capable of sending or receiving messages, properly called called frames. A frame consists of an ID which can be 11 bit or 29 bit long, and up to eight data bytes.

CAN transmissions uses a lossless arbitration method in case of multiple nodes simultaneously attempting to send data. Transmitted bit can be recessive or dominant. If two such bits are beind transmitted at the same time, the node transmitting the recessive bits suspends its operation and reattempts the transfer after a given number of clock cycles, effectively implementing a prioritized communication system. For this strategy to work, each node on a CAN network must agree on a bit rate and is required to sample every bit of data at the same time as the others.

Due to the lack of a shared clock signal, a synchronization mechanism is employed. *Hard synchronization* occurs after a recessive-to-dominant bit transition, and simply restart the timing from there. Each bit is divided into time intervals called *quanta*, and the first quantum is assigned to the so-called synchronization segment. Whenever a recessive-to-dominant bit transition occurs outside of the synchronization segment, the sampling time is moved accordingly. To ensure enough transitions to perform the above procedure, a bit of opposite polarity is inserted after five consecutive equal bits, through a process called *bit*

stuffing.

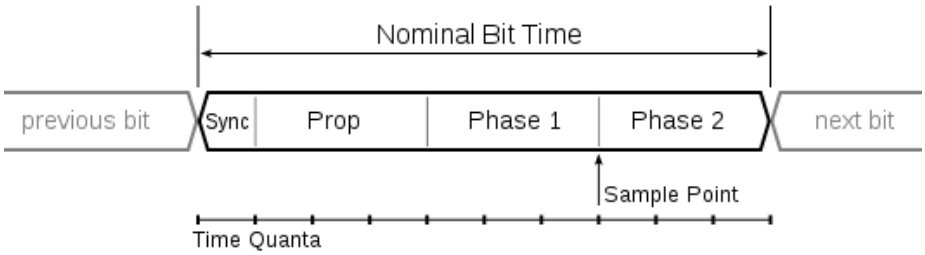

Figure 2.4: CAN Bus: bit sampling

Most CAN interfaces support automatic filtering of undesired frames through acceptance filter. The process consists in performing a bitwise *and* operation between the incoming frame ID and an acceptance bit-mask, and then comparing the result with the content of a filter register: if these match, the frame is added to the reception queue. In this way, a receiving node can choose to accept frames of a specific type, or sent by a particular other node.

2.3 Risk management

Risk assessment and management are well-established disciplines amongst many fields of engineering and other areas of human endeavor, and represent a way of minimizing the probability and impact of adverse conditions in a given context.

In the Risk assessment subsection, possible sources of hindrance are categorized and listed. Preventive measures are then suggested in the Prevention and mitigation subsection.

2.3.1 Description

This section acts as a legend, providing the necessary information for interpreting the subsequent risk assessment. Risks are evaluated according to their composite risk index (CRI):

$$CRI = Severity \times Probability\ of\ occurrence \tag{2.5}$$

The CRI represents the degree of importance of each entry. Entries with a high (ie. red-leaning) CRI shall have maximum priority.

Traditionally, this kind of analysis involves assigning a numeric value from an arbitrarily chosen range to each variable — e.g. 1–5 for the severity and 0–100% for the probability. However, educated estimations require solid data and a tracked history of previous occurrences, all of which were either not available or difficult to obtain with the necessary degree of confidence. [6] Therefore, the final value of the CRI has been neglected in favor of a color-mapped matrix composed of four sub-ranges. Whilst selecting these sub-ranges, higher “granularity” has been given to the more adverse scenarios.

		Probability of occurrence			
		Negligible (1)	Low (2)	Medium (3)	High (4)
Severity	Small (1)				
	Moderate (2)				
	Large (3)				
	Critical (4)				

Table 2.3: Risk impact matrix

2.3.2 Risk assessment

Follows the lists of threats (divided by category) with associated likelihood, impact, and overall CRI. The risk index is accompanied with the affected resource(s), chosen amongst:

- Product **F**unctionality
- Project **C**ost
- Development **T**ime

Whenever possible or necessary, a brief explanation of the predicted consequences is given.

Hardware

The estimations are based on company personnel's knowledge and previous experience with the given components and communication protocols governing them. For the purpose in this analysis the board components have been divided in primary (microcontroller, decaWave module, CAN transciever) and secondary (power supply unit, other sensors, USB interface), based on their importance for the fulfillment of the project.

ID	Risk	Prob.	Sev.	CRI	Res.	Contingencies/Notes
SD	*Schematics design issues (wrong/missing parts, connections...)*					
SD.1	Primary components	3	4	🔴	TC	New PCB manufacturing
SD.2	Secondary components	2	3	🟡	TF	
PL	*PCB layout issues (wrong footprints, overlapping traces...)*					
PL.1	Primary components	3	3	🟠	TC	New PCB manufacturing
PL.2	Secondary components	1	2	🟢	TF	
SC	*Supply chain issues*					
SC.1	Components lead time	1	3	🟡	T	Wait until restocked
SC.2	PCB lead time	3	3	🟠	T	Negotiation/bargaining
SC.3	Defective PCB	1	4	🟡	T	New PCB delivery
PU	*Performance/usability issues*					
PU.1	MCU doesn't boot up	1	4	🟡	T	
PU.2	MCU not detected by SWD	2	4	🟠	T	Tedious troubleshooting
PU.3	High packet loss (USB, CAN)	1	2	🟢	F	
PU.4	High packet loss (decaWave)	2	4	🟠	F	

Table 2.4: Risk assessment: Hardware-related threats

Software

The estimations are based on personal previous development experience, online research, and relevant documentation. The table sub-categories loosely reflect the different software layers described in the Software section.

ID	Risk	Prob.	Sev.	CRI	Res.	Contingencies/Notes
DE	*Development environment issues*					
DE.1	No compiler available	1	4	●	C	Switch MCU vendor/arch. or host system
DE.2	No flashing tools available	2	4	●	C	Switch MCU vendor/arch. or host system
DE.3	Loss of data	2	3	●	T	
LL	*Low-level support issues (peripherals, data busses...)*					
LL.1	No MCU peripherals drivers	2	3	●	TF	Implement from scratch
LL.2	No MCU peripherals docs.	1	4	●	TC	Switch MCU vendor/arch.
LL.2	No comm. with ext. devices	2	4	●	TF	
MW	*Middleware issues (libraries, device APIs...)*					
MW.1	Restrictive/costly license	1	3	●	CF	Purchase; Implement from scratch
MW.2	Lack of platform support	3	2	●	TF	Implement port
MW.3	Poor documentation	2	3	●	TF	
MW.4	Unstable/buggy code	2	2	●	F	
UA	*User application-level issues*					
UA.1	Reached computational limit	1	3	●	F	
UA.2	Middlewares conflict	2	3	●	TF	
UA.3	Poor system accuracy	2	3	●	F	
UA.4	Unimplemented use cases	2	4	●	T	Increased time-to-market
UA.5	Unmet requirements	2	4	●	T	Increased time-to-market

Table 2.5: Risk assessment: Software-related threats

2.3.3 Prevention and mitigation

After having discerned and evaluated as many threats as possible, a number of preemptive measures are suggested and summarized on a sub-category basis.

ID	Description	Approach	Applies to
HW.1	Adopt previously employed components and parts	Avoidance	SD, PL
HW.2	Follow design resources (datasheets, application notes...) from vendors	Reduction	SD, PL, PU
HW.3	Allow multiple people to verify the schematics	Reduction	SD
HW.4	Adopt established naming conventions for pins and signals	Reduction	SD
HW.5	Employ EDA tool's electrical and design rule check (ERC, DRC)	Reduction	SD, PL
HW.6	Verify parts availability and costs beforehand	Avoidance	SC
HW.7	Keep a local stock of commonly used components	Reduction	SC
HW.8	Generate BOM from EDA tool	Avoidance	SC
HW.9	Provision from multiple suppliers	Reduction	SC
HW.10	Pay for expedited delivery	Retention	SC
HW.11	Employ EDA tool's differential pair routing	Reduction	PU
HW.12	Limit bitrate to safer value	Retention	PU
HW.13	Base solution of DWM1000 (integrated module) instead of DW1000 (IC)	Retention	PU
SW.1	Verify tools availability and support	Avoidance	DE
SW.2	Adopt revision control system and host project on dedicated server	Reduction	DE
SW.3	Assess MCU capabilities through vendor's evaluation kit	Reduction	DE, LL, UA
SW.4	Switch to platform/vendor with more extensive support	Retention	DE, LL
SW.5	Provide redundant interfaces whenever possible (e.g. SPI and I2C)	Reduction	LL
SW.6	Connect pin to MCU when in doubt about its configuration	Reduction	LL
SW.7	Employ open-source software solutions	Reduction	DE, MW
SW.8	Acquire fluency with the necessary tools	Reduction	DE, LL, MW
SW.9	Use most tested and stable library version	Reduction	MW
SW.10	Employ reusable components from own/company source base	Reduction	LL, MW, UA
SW.11	Evaluate and test different ranging schemes	Reduction	UA
SW.12	Periodically assess development progress	Reduction	UA
SW.13	Consult online community (user forums, mailing lists, GitHub...)	Reduction	DE, LL, MW, UA

Table 2.6: Risk management: threats prevention and mitigation proposal

2.4 Milestone plan

Here is the project milestone plan as formulated during the *Inception* and *Elaboration* phases. This version of the plan lists a series of achievements deemed necessary to accomplish the major functionalities. It is arranged in a progressive way, with a time axis roughly flowing vertically. Each task depends from its sub-tasks and — more indirectly — on the previous ones within its hierarchal

level.

Although it has been used — together with the previously discussed Use cases — as a reference for the project implementation, unforeseen difficulties have caused the development to drift considerably from the plan.

Please refer to the Possible improvements for the final version of the milestone plan based on the actual progress.

- Extract requirements and use cases
- Test functionality of the modules
  - Simple communication
  - Ranging example
- Schematics design
- Software development
  - Implement support layers (peripherals, tools...)
  - Integrate libUAVCAN
    - Port drivers
    - Implement node
  - Localization system
    - Trilateration
    - Data publisher
- Integration and validation tests
- Documentation
  - Project report
  - User manual
  - Production files

Chapter 3

Design and implementation

This section focuses on the design and implementation of the proposed solution into a physical product. It namely consists in the microcontroller-driven circuit board design, the firmware for the beacons/anchors, and that of the tag.

During the course of the development there have been two major hardware design iterations, with the latter completely breaking compatibility with previous code. This controversial measure has been deemed necessary after having invested a considerable amount of the allocated time trying to troubleshot the decaWave module transmission failures. (refer to Risk assessment ID# PU.4, SC.2 and Prevention and mitigation ID# SW.4)

Nevertheless, the upcoming sections will present an exhaustive description of the system, which may serve as a vademecum for the continuation of the project (outside of its academic scope). Such solution is based on the second, more successful iteration of the project. Hardware design and code frome the former design are included in the deliverables, as described in Deliverables. Adequate considerations about the development process will be drawn in section 5.2.

Amongst the third-party tools and software/hardware components employed in the making of the project, only those directly embedded into the implementation of the aforementioned designs (e.g. UAVCAN stack, ChibiOS, the onboard sensors, etc...) will be thoroughly described, whilst the others are only briefly mentioned (see subsection 3.1.1 and subsection 3.2.1).

3.1 Hardware

This section will present the hardware architecture of the system, focusing on both large-scale aspect and individual components/subsystems.

The board design consists in a single PCB built around the decaWave DWM1000 ultra-wideband radio module. It also includes a with a microcontroller, CAN and USB interfaces, and secondary sensors. At least four of them are necessary for a complete system and since the anchors are stationary, they do not require the secondary sensors to be populated.

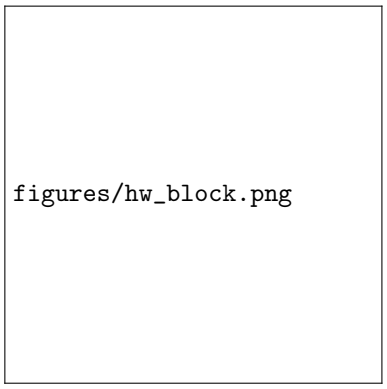

Figure 3.1: Hardware design: block diagram

3.1.1 Setup

The schematic capture, PCB layout, and subsequent export of production files (Gerber,) has all been performed with the aid of
emphKiCad, an open-source ECAD software. Most of the necessary components were available out of the box, with the missing ones imported from the company's *Eagle* libraries. KiCad also include a component library editor for both symbols and footprints.

For hardware troubleshooting and testing, a quad-channel Tektronix digital oscilloscope was used.

Connection of the board to the host system for firmware upload and debug was possible thanks to an *ST-LINK/V2*-compatible SWD in-circuit debugger. Before the custom-made PCB were available, development has been carried out on a *STM32F4DISCOVERY* board, which features an the aforementioned debugger probe, an MCU from a compatible family, and pin headers.

3.1.2 decaWave DWM1000 module

The DWM1000 is an UWB wireless transceiver module designed for easy adoption in two-way ranging schemes. It is based around the DW1000, an integrated low-power, single chip CMOS RF transceiver IC compliant with the *IEEE802.15.4-2011 UWB* standard, and has been chosen because it required no RF design: the antenna and associated RF circuitry are already on the module.

Along with typical RF modules capabilities such as configurable power modes and data transmission rates, the module can use custom, proprietary leading-edge detection algorithm which — along with configurable preamble size and optional, non-standard frame delimiter — can provide accurate timestamping of incoming and departing messages.

The module contains an 38.4 MHz reference crystal used by the SPI interface and the transmitter for signal modulation and timestamps generation. Its accuracy is ensured by a factory trimming process which brings the frequency offset down to about 2 ppm. The on-chip crystal trimming circuit and temperature

sensor allows for dynamic temperature compensation.

Interfacing with the device is achieved through an SPI bus that can run at up to 20 MHz. Other data connections include the interrupt (IRQ) and reset (RST) signal. The bus configuration (sampling polarity and phase) can be customized by manipulating the GPIO pins, which are also routed to the microcontroller.

In laying out the board for PCB manufacturing, attention has been paid towards the guidelines included in the datasheet, which suggests a keep-out area of 1 cm around the antenna section of the module, plus a board-wide ground plane.

3.1.3 Microcontroller

After the previous failed attempt with the *LPC1549* ARM Cortex-M3 microcontroller running at 72 MHz, it has been deemed necessary to move towards a more powerful MCU with faster serial communication ports.

A better device for the project has been found in the *STMicroelectronics STM32F405RG*. It is based upon the 32-bit ARM Cortex-M4 core running at up to 168 MHz, and feature a single-precision FPU, 1 MB of FLASH memory, 192 kB of static RAM, and a wide array of internal peripherals including CAN, SPI, USART, I2C, and USB.

Most these peripherals are commonly available in modern, high-performances MCUs. However, the STM32F405RG has the advantage of being shipped in a 64-pin LQFP package which makes it tiny, inexpensive, and easy to solder manually.

3.1.4 Sensors

Two supplementary sensors are present on the board: a *BMP-280* pressure sensor and *MPU-9250* inertial measurement unit.

The first one measures atmospheric pressure, which can be used for altitude estimation. The latter provides measurements of inertial acceleration, angular rate, and magnetic field along along each of the three axes. It operates thanks to internal *MEMS*-based suspended structures which move relative to the rest of the die, and whose changes in capacitance are then sampled and processed. IMUs are often used to measure the heading of a vehicle in space, and can also be part of a *dead-reckoning system* for position estimation.

As shown on Figure 3.1, the devices are connected through both a SPI and an I2C bus (as per SW.5, SW.6). An additional `INT` pin in the IMU is used to notify the microcontroller when new samples are ready.

3.2 Software

This section discusses the implementation details of the two firmwares involved in the project. The two firmwares share most of the code base, with only the application-level parts performing different operations. Their main components are:

ChibioOS An open-source real-time operating system (*RTOS*) with portable peripherals support and high-performance kernel

Zubax-ChibioOS Middleware providing a more comprehensive command-line interface over USB, watchdog support, and FLASH-based configuration storage

libuavcan Reference implementation of the UAVCAN protocol written in C++

decadriver APIs for controlling the decaWave products

modules High-level board functionalities and RTLS implementation

Figure 3.2 shows a high-level block diagram of all the components involved, and their collaboration.

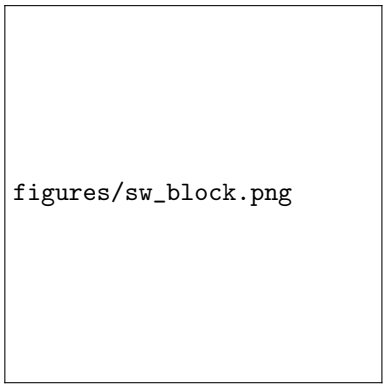

Figure 3.2: Software design: high-level model

Due to the difficulties presented at the beginning of chapter 3, at the time of writing only the anchors software is fully developed. For a detailed report of the missing features and requirements, refer to Acceptance tests.

The following subsections will cover the shared project layout, informations and integration details of the third-party modules, and core application.

3.2.1 Setup

An overview of the host system used for the development is hereby provided for reference purposes.

Development environment

The choice of an integrated development environment has fallen on *Ac6 System Workbench for STM32*, an Eclipse-based IDE tailored around STMicroelectronics line of microcontrollers. The distribution is based on open-source software, and requires little to no configuration. The components shipped with it are:

GCC ARM Embedded GCC-based cross compiler and debugger for ARM Cortex platform

OpenOCD On-chip debugging tool supporting a wide range or generic JTAG/SWD interfaces

Eclipse CDT An extensible multi-language IDE

All of these packages could be installed manually by fetching them from their respective sources, but doing so with a single archive is undoubtedly more convenient. On top of this, it also comes with its own project management facilities like build setting manager, linker script editor, STM evaluation boards library, and so forth.

These latter tools are not taken advantage of — the project is in fact built using `make`, as it is the supported build system for both ChibiOS and libuavcan. Building projects using makefiles is fully supported within Eclipse.

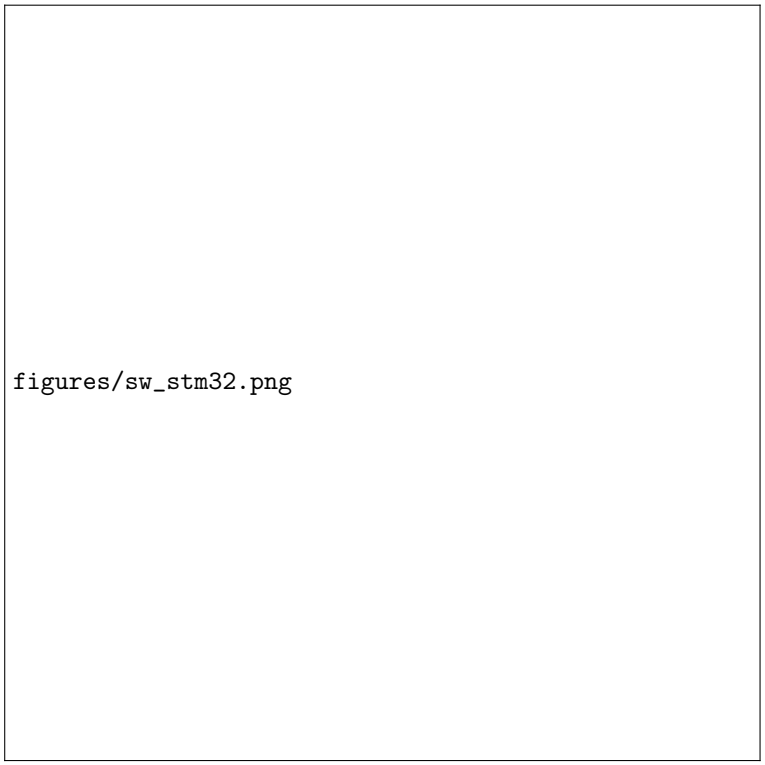

Figure 3.3: Development environment: typical session

Project structure

The firmware folder (folders, more correctly, but here summarized as one since they follow the same system) is a self-contained unit which can be downloaded and build without the need of further operations. This as been achieved by including all the external libraries and modules as `git` submodules, and comes with the benefit of being able to retrieve their upstream commits.

Directory or file	Description
.	Project root folder
./libuavcan/	UAVCAN software stack, submodule
./dsdlc_generated/	UAVCAN data type header files, autogenerated
./zubax_chibios/	Zubax-ChibiOS middleware, submodule
./zubax_chibios/chibios/	ChibiOS folder, submodule
./Makefile	Project-level makefile
./src/	Application-level code
./src/main.cpp	Main file, includes firmware entry point
./src/board/	Board module
./src/sys/	ChibiOS configuration files
./src/usb/	USB descriptors and CLI module
./src/node/	UAVCAN node module
./src/rtls/	Localization class
./src/rtls/dw1000/	decaWave DWM1000 drivers
./src/rtls/twr/	Two-way ranging class
./src/build/	Build directory, autogenerated
./src/build/das_uavcan_rtls.bin	Uploadable firmware, autogenerated

Table 3.1: Software design: folder structure

Table 3.1 shows an overview of the project folder content. Most files have been hidden from the table, as well as submodule subdirectories. It is implied that each application-level module is implemented inside its respective directory.

Other tools

In order to be able to contribute from any given location (work PC, personal laptop...), the source code folder is hosted on a web-based `git` repository manager called *GitLab*. GitLab provides most of the functionalities of GitHub (which was originally used) but can run on premises, making it more suitable for internal research projects. Ordinary maintenance of the repository is performed from command-line, with the aid of graphical `diff`-tool *Meld* for code revision.

Whilst the IDE's powerful code completion is useful for general programming, external text editors such as *SublimeText* or *Atom* are sometimes used as well, especially when large amount of code has to be refactored or restructured.

3.2.2 ChibiOS

ChibiOS is an open-source, real-time operating system for embedded systems. It is written entirely in C and Assembly. It has achieved widespread popularity due to its reliability and compactness in terms of binary footprint and computation overhead.

The ChibiOS endeavor includes two kernel variants (RT and NIL, a variant that focuses on low-performance architectures) and a hardware abstraction layer (*HAL*). Besides the features already mentioned, ChibiOS can boast on a strong community support, vast range of supported architectures and devices (mainly in the STM32 family), and well-documented code base.

Figure 3.4 shows the different modules of a typical ChibiOS-based project. The major ones are explained in the following sections.

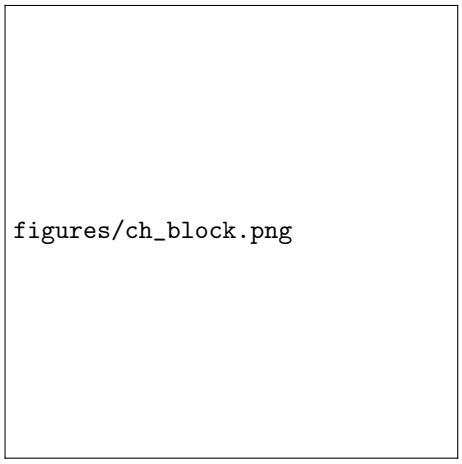

Figure 3.4: ChibiOS: block diagram

RT

RT is the name given to ChibiOS flagship kernel. It designed for size and execution efficiency, and is currently available on ARM7, ARM9, ARM Cortex-M, and PowerPC architectures. These goals are achieved through its particular design choices, such as using double linked circular lists for its internal data structures, synchronous context switching, and almost complete lack of status code return values in the APIs.

It comes with an extensive set of features, including commonly used real-time programming constructs. Follows a list of the most notable ones.

- *Thread synchronization*: Mutexes, semaphores, condition variables, event flags
- *Thread communication*: Synchronous messages, mailboxes, I/O queues
- *Memory management*: Allocation on both heap and static-size pools (safe `malloc`)

Moreover, it complies with *CMSIS RTOS* set of APIs (making ports to and from different compliant system easier), and *MISRA-2012* coding standard, ensuring code quality and reliability.

There are currently three active branches (2.6, 3.0, and 16.1), and the version used in the project is the one tagged 3.0.1. The newest branch is currently not supported by `libuavcan`.

HAL

The hardware abstraction layer included with ChibiOS is extremely comprehensive an includes support for almost all peripherals in a modern microcontroller.

In particular, all devices from the STMicroelectronics STM32 line are supported, with HAL drivers and examples available in the code base and capable of running on the major development boards.

Some of the supported features are:

- *General purpose*: external interrupts, GPIOs, real-time clock
- *Serial protocols*: CAN, I2C, I2S, SPI, UART...
- *Analog functionalities*: ACD and DAC
- *Timer functionalities*: one-shot timer, continuous timer, PWM and ICU, and SysTick timer
- *Abstract interfaces*: streams, channels, files, block devices
- *Support*: circular buffers, stream formatter, built-in streams

Complex drivers can be implemented through one of the abstract interfaces. Available complex drivers include MMC/SD cards interfaces and USB CDC device class.

The APIs are portable across different architectures and devices, and only the bottommost of the several layers of abstraction is affected. It can as well be used alone in bare-metal projects.

Configuration

Most aspects of ChibiOS RT and HAL can be customized by means of precompiler texttt#define directives. These can either be declared in the project `Makefile` or in separate header files within the include path. Due to the amount of configuration needed, the second approach has been chosen. There are three layers of configuration: board, HAL, MCU, and OS. Following ChibiOS conventions, the following files have been used:

- `board.h`
- `chboard.c`
- `halconf.h`
- `mcuconf.h`
- `chconf.h`

The first layer includes the setup of each MCU pin and the frequencies of the external oscillators (HSE for clock generation and LSE for the real-time clock). These definitions are then used in the C source file to initialize the device. If needed, custom initialization instructions can be added there. In this case, it was necessary to configure the CAN pins manually in order to avoid disturbing the bus or triggering interrupts.

The following two files are based on the templates for the specific platform available in the code base. The file `halconf.h` simply defines which HAL driver should be compiled and linked to the project, while `mcuconf.h` includes the

thorough setup of PLL values (performing compile-time checks of their validity) and the enabled peripherals and HAL interfaces.

Lastly, `chconf.h` is the configuration file for the RTOS. Its parameter have been left to default, with the possibility of enabling debug-specific controls depending on build flags.

A ChibiOS project built using GNU Make also requires a properly setup Makefile, which shall include the relevant linker script and port-specific inclusion such as startup, rules, and platform makefiles, It may also specify custom compiler flags and definitions.

3.2.3 UAVCAN

UAVCAN is a lightweight protocol aimed to provide fast and reliable communication within a drone peripherals.

A UAVCAN network is composed of a series of nodes, each one identified by a unique ID. Each node is independent from the others and require no master/controller node or host computer. Communication between the nodes is called *transfer* and can occur through one of these two paradigms: message broadcasting (publisher-subscriber logic) or service invocation (request-response logic).

A CAN frame is composed of an ID and a payload. The payload section contains the encoded message data only, so all the relevant meta-data (source node, message type, destination node in case of service request, etc...) must figure in the frame ID.

Message frame

Field name	Priority					Message type ID													Service not message	Source node ID						
CAN ID bits	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Allowed values																						0	1...127						
CAN ID bytes	3					2								1								0							

Anonymous message frame

Field name	Priority					Discriminator								Lower bits of message type ID								Service not message	Source node ID						
CAN ID bits	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Allowed values																						0	0						
CAN ID bytes	3					2								1								0							

Service frame

Field name	Priority					Service type ID								Request not response	Destination node ID							Service not message	Source node ID						
CAN ID bits	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Allowed values															1...127							1	1...127						
CAN ID bytes	3					2								1								0							

Figure 3.5: UAVCAN protocol: Frame ID formats

Figure 3.5 shows the arrangement of the frame ID for different transfer types. Although UAVCAN frame IDs are 29 bits in size (*CAN 2.0B*), its bus can be shared with protocols using the former 11 bits format (*CAN 2.0A*). *Anonymous message frames* are those sent by nodes which do not have a default node ID

and have not been assigned one yet. In order to avoid CAN collision deadlock, the discriminator section has to be filled with random data.

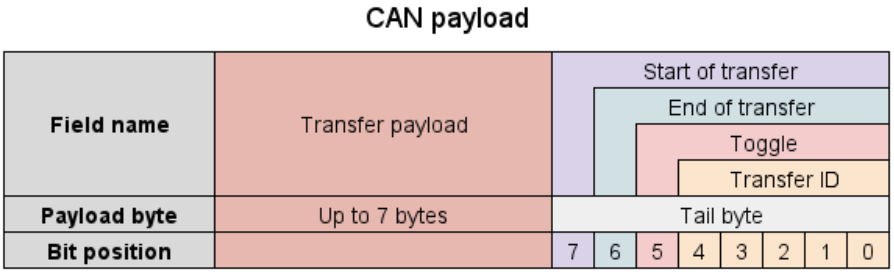

Figure 3.6: UAVCAN protocol: Payload format

The payload structure is the same regardless of transfer type. In single-frame transfers, the *start of transfer* and *end of transfer* bits shown in Figure 3.6 are both set to one.

Since each transfer may contain up to eight bytes of data per frame, multi-frame transfers are also available. In this case, the first two bytes of the first frame will contain the CRC of the complete payload, and the bit-flags in the tail byte are to be set accordingly. The *toggle* bit is set to 0 for even frames and 1 for odd ones. The *transfer ID* bits contain an incremental transfer counter, and its value must be the same for each frame of a given transfer.

Messages and services can be standard or vendor-specific. The standard ones are divided in *protocol* and *equipment*, with the former enabling UAVCAN-related features and the latter providing data definitions for commonly used devices like navigation systems, airspeed sensors, and so forth.

Compliant node are only required to broadcast a heartbeat message on a regular basis to signal their presence. The implementation of other standard functionalities is not mandatory but recommended. For instance, nodes that do not implement `uavcan.protocol.GlobalTimeSync`-enabled time synchronization are condemned to experience time drift. Here follows a table of UAVCAN high-level functions along with the messages/service through it is carried out, and — if available — the use case(s) or requirement(s) it fulfills.

Function	Service/Message definition	Applies to	Notes
Node status reporting	`uavcan.protocol.NodeStatus`		Mandatory
Node discovery	`uavcan.protocol.GetNodeInfo`		Strongly suggested
Time synchronization	`uavcan.protocol.GlobalTimeSync`		Needed by publishers of timestamped data
Node configuration	`uavcan.protocol.param.GetSet` `uavcan.protocol.param.ExecuteOpcode`		
Node restart	`uavcan.protocol.RestartNode`		Recommended, may be used to refresh node settings

File transfer	`uavcan.protocol.file.GetInfo`
	`uavcan.protocol.file.GetDirectoryEntryInfo`
	`uavcan.protocol.file.Delete`
	`uavcan.protocol.file.Read`
	`uavcan.protocol.file.Write`
	`uavcan.protocol.file.EntryType`
	`uavcan.protocol.file.Error`
	`uavcan.protocol.file.Path`		

Firmware update	`uavcan.protocol.file.BeginFirmwareUpdate`		Requires *File transfer*

Debug features	`uavcan.protocol.debug.KeyValue`
	`uavcan.protocol.debug.LogMessage`
	`uavcan.protocol.debug.LogLevel`
			Lowest-priority messages

Command shell access	`uavcan.protocol.AccessCommandShell`		
Panic mode	`uavcan.protocol.Panic`		
Dynamic node ID allocation	`uavcan.protocol.dynamic_node_id.Allocation`		Uses anonymous transfers

Table 3.2: UAVCAN protocol: high-level functionalities

As part of the *Node configuration* function, the following parameters should be available by default:

- Data type ID
- Message publication period
- Transfer priority
- Node ID
- CAN bus bit rate
- Instance ID

Library

UAVCAN provides a software stack written in C++ called `libuavcan`. It consists in a portable, cross-platform library, a package of standard message definitions written in a custom data definition language, and a code generator that converts these into header files for application-level inclusion.

Libuavcan depends on only a very limited portion of the standard C++ library, making it compilable on most moder compilers and embeddable in microcontroller architectures and full-fledged systems alike. Through compile-time configuration, it gives the possibility of ruling out processing or memory-costly or parts, such as exception handling, run-time type identification (RTTI), and dynamic memory allocation. The latter is a desirable feature on many embedded and real-time system.

A major characteristic of libuavcan is that it offers built-in support for all of the high-level functions presented above. These can be individually added to a project by including the relevant header files from `uavcan/protocol/` and initializing the service providers. Although the recommended/suggested functions are included by default, the developer has the possibility of disabling them through the `UAVCAN_TINY` variable, in order to further reduce the memory footprint. This is however only recommended for deeply embedded systems with severe memory constraints, as it will also remove support for transmission diagnostics, event logging, and memory pool allocation.

For the purpose of this project, the library has been configured to compile with `C++11` support and all suggested functions available. All other settings have been kept to their default value.

Depending on the development environment used, the drivers and library can be built using `cmake`, `make`, or any IDE-specific build system, as long as the necessary source files are present and the libuavcan include folder added to the inclusion path.

Platform driver The library is not bounded to any specific CAN interface, and relies on a platform-dependent driver. As of today, the officially supported platforms are:

- *Linux* (requires `libposix4` and `SocketCAN`), most comprehensive driver
- *POSIX*, only abstract interface
- *LPC11C24*, minimal implementation, no multi-threading
- *STM32*, middle-ground support

The drivers are responsible for implementing the low-level, platform-dependent functions, which are abstracted by the following C++ classes:

- `uavcan::ISystemClock`, providing monotonic clock (invariant) and network clock (synchronized and adjusted across the nodes)
- `uavcan::ICanIface`, providing `send` and `receive` functions, as well as CAN filters configuration
- `uavcan::ICanDriver`, which acts as a container for supporting multiple redundant interfaces.

The libuavcan driver for STM32 has no third-party dependencies (except if RTOS support is enabled). It supports all of the major capabilities, including

redundant CAN interfaces (whenever available on the MCU), clock synchronization (through general-purpose timer), and has bindings for several RTOS (ChibiOS, NuttX, and FreeRTOS).

The driver too is configurable via preprocessor definitions. The ones used in the project are reported below.

Parameter	Value	Description
`UAVCAN_STM32_CHIBIOS`	`1`	Symbol defining the platform used; mandatory
`UAVCAN_STM32_TIMER_NUMBER`	`7`	Specifies hardware time used for clock; mandatory
`UAVCAN_STM32_NUM_IFACES`	`1`	Number of available CAN interfaces
`UAVCAN_STM32_IRQ_PRIORITY_MASK`	`4`	Priority level of driver-related interrupts

Table 3.3: UAVCAN driver: compile-time settings

3.2.4 Software modules

The application-level code is contained in the `src` directory. In compliance with NONFUNCTREQ, it is divided into modules based on the underlying hardware component they manage, or the high-level function they implement.

Each module is implemented as a C++ class or namespaced collection of functions, depending on what seemed most semantically reasonable. They may also include or depend on third-party libraries or other modules.

Board layer

This module manages some of the generic device functions and provides useful hooks to commonly used functions. It is not wrapped in a class, due to the inherit *static*-ness of its functions (no object to refer to and state preserved).

It is divided into three namespaces:

`led` Functions to set the color of the on-board RGB LED. Since the LED is used by several modules for signaling their events and state, there are also functions for setting and clearing individual sub-pixels.

`sys` Helpers for system functionalities: low-level initialization (ChibiOS internals and HAL, watchdog, configuration manager), reboot, and fault state handler.

`id` Type-safe functions for sourcing and setting (whenever allowed) identification data: vendor-assigned Unique ID, device signature, hardware platform version (the latter being only a stub)

UAVCAN Node

The node module handles most of the high-level functions presented in the UAVCAN section. It is composed of several parts:

- Configuration parameter variables, handled by the configuration manager described in the following subsection.
- A parameter manager, implemented as a class inheriting from `uavcan::IParamManager` and used to provide support for UAVCAN node configuration.
- A restart service handler, implemented as a class inheriting from `uavcan::IRestartRequestHandler` and used to provide support for UAVCAN node restart.
- A ChiobiOS static thread, implemented as class inheriting from `chibios_rt::BaseStaticThread`, along with internal functions for the setup and initialization of its service providers.
- A set of helper methods to be used by other modules to notify their status.

The UAVCAN functions initialized and managed by the node are:

- Node status reporting
- Node discovery
- Data type info provider
- Dynamic node ID allocation
- Node configuration
- Time synchronization (as slave)
- Node logging
- Node restart

A component status manager takes care of caching the status of the relevant sub-modules and making it available to UAVCAN built-in status provider. Each component is responsible of setting its own initialization and health status. In order to properly notify problems, the health status of the most compromised component is used as the overall health status, and the initialization status will not be suppressed until all components are initialized.

The component status manager is not externally accessible and its methods are relayed into the node class to allow for easy back-end swapping, and to keep interdependency at bay. The same applies to the UAVCAN derived classes and ChibiOS thread, which are all encapsulated in an anonymous namespace.

decaWave drivers

This module contains the software APIs for managing the DWM1000 device, as provided by decaWave, with minor adjustments and completely rewritten porting layer. It is essentially a driver providing support to the main features of the transceiver, hiding the underlying register-map interface. It also includes all the calibration and parameters loop-up table described in the DW1000 user manual and API guide.

The platform-dependent functions are conveniently detached from the rest of the code, and implemented in the following files:

`deca_mutex.c` Functions for atomic operations (e.g. during SPI transfers)

`deca_sleep.c` Function implementing delay

`port.h` Declaration of peripheral-related functions: SPI communication and clock frequency setting, external interrupt channels, pin configuration

`port.c` Implementation of the aforementioned functions

All of the described code has been reimplemented using ChibiOS-specific HAL and RT calls. While the first two functionalities could be tightly mapped to ChibiOS equivalents, the same was not possible with the latter files, and some of the generic initialization operations have been moved out of the decaWave port layer and into the board module. As of today, only polling-based communication with the DWM1000 is supported.

Since the library is written in C, a common practice would be to wrap all the necessary API methods, static variables, and headers in a C++ class. However, due to the staggering amount of functions and the disappointing previous results of such attempt, it has been chosen to employ them as-is, and limit their scope to the ranging module. In order to be able to use them from the C++ ranging class, the relevant header files have to be surrounded by the `extern C¨` directive. This is to avoid C++ compiler name mangling in function calls, which would otherwise result in linker errors.

Care must be taken as the API is not designed to be re-entrant, therefore only a single thread can call them or a synchronization strategy must be employed.

Configuration manager

This module allows the callee to store and retrieve arbitrary data of numerical type in non-volatile memory, in a way similar to associative arrays or dictionaries data structures. The ability to manage settings beyond compile-time constants is needed in order to support some UAVCAN high-level functionalities, and finds applications in debugging and fine-tuning a device.

Initially (i.e. during the first iteration) an EEPROM-based implementation of such module was devised (information on how to find it is given in the Deliverables section). Since the STM32F405 is not equipped with it, the last sector of FLASH memory has been used and the implementation taken from the Zubax-ChibiOS set of reusable components.

This implementation is quite similar to the one of own craft, i.e. both are internally based on a simple table-like structure with access time proportional

to the number of elements. It also has built-in enforcement of parameter boundaries and, a adopt a lesser object-oriented style.

A parameter is declared as a `os::config::Param<type>` object, and it is accessed with `obj.get()` and `configSet(name, value)`.

Command-line interface

A command-line interface (*CLI* in short) is way of interacting with a software or device by mean of entering strings of text.

Although they have been superseded by GUIs in daily compure use, CLIs are still a quite popular tool in embedded systems, where they find applications as configuration, monitoring or debugging tools. They can also be used as back-end for graphical interfaces.

In this case too a CLI has been developed as part of the former set of modules. It was however based on the low-level UART drivers for the LPC1549 platform, which would have been much more daunting to port than the command interpreter itself.

The currently used CLI is the one provided by ChibiOS, with hooks for running through the *USB Communications Device Class*. The CLI configuration is composed of a list of strings representing the command names, and their associated callbacks. The callback functions are internally invoked by the CLI parser, which provides them with eventual parameters entered by the user, using the established `int argc, char **argv` convention. A third parameter is also passed, and allowing the command to output to the same stream

The currently available commands are:

`cfg` Parameter configuration, just as the UAVCAN one

`reboot` Ditto

`threads` Overview of the existing threads along with status, priority and available memory

`info` Overwritten built-in command; System and application infos (ChibiOS version, compiler used, architecture, application version...)

`systime` Built-in command; shows the currently elapsed time

`help` Built-in command; lists all the previous entries

These are subject to extension as new functionalities will arise and require testing or configuration.

Two-way ranger

This module is responsible of performing individual distance measurements between the tag and the anchors. As such, it is available in two, complementary variants: the *initiator* running in the tag, and the *responder* responder running in the anchor.

As of today, the implementation of this module exists only in a proof-of-concept application basis. The following explanation of the class is provided for reference purposes. The relevant example code can be found in the deliverables.

The software module is to be implemented as a C++ class and leverages on the decaWave driver for operating the radio modules. It is meant to be used as part of the *Location estimator* rather than on its own.

Inizialization The class shall provide means of configuring the DW1000 modules appropriately. The decaWave functions responsible for this are `reset_DW1000()`, `dwt_initialise()`, and `dwt_configure()`. The former performs a hard reset of the module. Subsequently, the second function initializes the internal transceiver and the static data structures used by the driver. It may also enable *LDE microcode*, which allows for more accurate timestaping. From this point, the SPI bus can be set to full speed. The latter is then used to apply the configuration values to the radio.

All device types undergo the same configuration process, with the same parameters. The exact values of parameters depend heavily on the ranging scheme adopted.

Antenna delays are also required to be configured for optimal performance, with values stemming from experimental observation or systematic calibration of each board.

Discovery and pairing Once the configuration and calibration process is over, the tag is responsible of performing a network discovery in order to compile a list of available anchors. It will therefore broadcast a pairing message containing an ID (the implementation details of which may vary, but could be the vendor-assigned UID, or could be randomly drawn from an agreed-upon range). Anchors in the area will then respond with unicast messages, revealing their IDs. Further steps may be necessary to define shorter (ideally 16 bit) addresses valid for a single session.

Ranging procedure The ranging procedure involves two different messages: *poll* and *response*. A poll is sent by the initiator/tag to a specific anchor whenever a new distance measurement is required. To do so, it must include its ID as the source and the anchor ID as the destination. A response is sent by the anchor/responder, and contains the timestamp of the last poll arrival, as well as that of its transmission.

In case of successful message exchange (no timeout occurred, matching checksums) the tag can safely calculate and return the new distance sample. Note that the aforementioned procedure describes a *single-sided* ranging scheme. Other schemes may require different packages or data, in accordance the discussion in section Theoretical principles.

Location estimator

Chapter 4

Testing

4.1 Acceptance tracing

This section should be an overview of the tests conducted. A table shall correlate the requirements with the tests that prove their fulfillment, as well as the outcome (passed, not passed, partially passed, feature not implemented).

4.2 Acceptance tests

Detailed view of the hardware-related tests. Should show the following: purpose (which requirements it tests), preconditions, test actions wth relative expected results.

Chapter 5

Conclusions

5.1 Product assessment

Conclusions based on the product, e.g. if it fulfilled all the requirements, if it yielded the desired performances etc ...

5.2 Process assessment

Conclusions based on the development, e.g. how lean it has been, any useful tool discovered, any cumbersome aspect of the workflow that coud have been improved ...

use risk management to explain escalation of issues: e' stato veritiero? e' servito a un cazzo?

5.3 Possible improvements

Suggestions on improvements based on missing features, company and supervisors feedbacks, etc ...

Bibliography

[1] 2016 — Richard W. Pogge — Real-World Relativity: The GPS Navigation System http://www.astronomy.ohio-state.edu/~pogge/Ast162/Unit5/gps.html

[2] 2009 — Joost Koppers — Location Based Services: A general model for the choice between positioning techniques (Dutch) Radboud University Nijmegen

[3] 2014 — Pavel Kirienko — UAVCAN Specification http://uavcan.org/Specification/1._Introduction/

[4] 2013 — DTU Course Base — Object oriented software engineering compendium http://www.kurser.dtu.dk/2013-2014/62432.aspx

[5] 2005 — Faranak Nekoogar — Ultra-Wideband Communications: Fundamentals and Applications ISBN-10: 0-13-146326-8

[6] 2004 — David Hillson & David Hulett — Assessing Risk Probability: Alternative Approaches PMI Global Congress Proceedings

Appendix A

Glossary

MCU Microcontroller Unit

NXP A semiconductor designer and manufacturer company

RTLS Real-time locating system

UAV Unmanned aerial vehicle, commonly known as drone

term definition

another term another definition

Appendix B

Quickstart guide

Setup, CLI and stuff.

Appendix C

Deliverables

This section contains a description of the material found in the hand-in support.

Directory or file	Description
.	Root folder
./iter1/	First project iteration based on LPC1549
./iter1/hardware/	KiCad project including schematics and production files
./iter1/firmware/	Source code folder
./iter1/firmware/lpc1549_uavcan_drv/	Porting of UAVCAN drivers for LPC1549 platform
./iter1/firmware/lpc1549_hal/	C++ hardware abstraction layer for common peripherals
./iter1/firmware/lpc1549_modules/	Reusable high-level modules
./iter1/firmware/lpc1549_rtls/	Application project
./iter1/firmware/lpc_chip_15xx/	LPCOpen libraries
./iter1/examples/	Test code and proof of concepts
./iter1/examples/raw_can/	Transmission of raw CAN frames
./iter1/examples/txrx/	Message transmission with DW1000
./iter2/	Second project iteration based on STM32F405
./iter2/hardware/	KiCad project including schematics and production files
./iter2/firmware/	Application project (including all third-party libraries)
./iter2/examples/	Test code and proof of concepts
./iter2/examples/txrx/	Message transmission with DW1000
./iter2/examples/ss2wr/	Test code with two-way ranging (single sided)
./iter2/examples/ds2wr/	Test code with two-way ranging (double sided)
./docs/	Documentation: datasheets, application notes, manuals etc...
./report/	This document, in pdf as well as LaTeX formats

Table C.1: Further material: folder structure