



**Development of technology and procedures for health
monitoring of UAV subsystems**

Pedro Bernardo Andrade da Silva

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Examination Committee

Chairperson: Prof. João Miranda Lemos

Supervisor: Prof. Agostinho Rui Alves da Fonseca

Member of the Committee: Prof. Bertinho Manuel D' Andrade da Costa

November 2015

Dedicado à minha família
e à Lili.

Acknowledgments

I would like to thank UAVision for giving me this opportunity, for giving me a subject, providing me with the material necessary for this thesis and allowing me to work with them.

I would also like to thank Prof. Agostinho Rui Alves da Fonseca for guiding me through this thesis and his permanent availability towards me.

Last but not least I would like to thank my family and friends, the unsung heroes that have helped me throughout the years. This wouldn't have been possible without you.

Resumo

A indústria dos Veículos Aéreos Não Tripulados (VANT ou UAV do inglês Unmanned Aerial Vehicle) está a crescer. À medida que a indústria cresce, os VANT crescem em complexidade. As aeronaves têm um crescente número de sistemas a bordo e, na ausência de standards, é frequente que cada um necessite de protocolos e ligações diferentes. A consequência é um aumento do peso e da complexidade. O protocolo Unmanned Aerial Vehicle Controller Area Network (UAVCAN) é uma proposta para um standard que utiliza o Controller Area Network (CAN), popular na indústria automóvel e da automação. A utilização do CAN permitiria que a cablagem fosse trocada por apenas dois cabos. A pedido da UAVision, esta tese integra o protocolo UAVCAN com um popular piloto automático chamado ArduPilot, e a subsequente validação. Fazê-lo requer a utilização de uma placa de piloto automático chamada Pixhawk, e de duas outras que servem de nós na rede. O software foi construído e a rede criada. O trabalho foi bem sucedido, com o piloto automático a controlar remotamente um servomotor e a receber a leitura de um sensor de temperatura.

Palavras chave: Veículos Aéreos Não Tripulados (VANT ou UAV), Controller Area Network, piloto automático, ArduPilot, Pixhawk

Abstract

The Unmanned Aerial Vehicle (UAV) industry is growing in size. As the industry grows, so does the complexity of the UAVs. The aircraft have an increasing number of systems onboard and, in the absence of industry standards, it is often that each requires a different protocol and wiring. The consequence is an increase in weight and complexity. The Unmanned Aerial Vehicle Controller Area Network (UAVCAN) protocol is a proposal for a standard that works with the Controller Area Network (CAN) bus, popular in automation and automotive industries. Using the CAN bus would allow the wiring harnesses to be traded with a simple two-wire bus. As requested by UAVision, this thesis integrates the UAVCAN protocol with a popular autopilot software called ArduPilot, and subsequent validation. Doing so required the use of an autopilot board, called Pixhawk, and two other boards that serve as nodes in the network. The software was built and the network created. The work was successful, with the autopilot used to remotely control a servo motor and reading a temperature sensor.

Keywords: Unmanned Aerial Vehicle, Controller Area Network, autopilot, ArduPilot, Pixhawk.

Contents

Acknowledgments	v
Resumo.....	vii
Abstract.....	ix
Contents.....	xi
List of figures.....	xvi
List of Tables.....	xvii
List of acronyms	xviii
1 Introduction.....	1
1.1. Motivation.....	1
1.2. Objectives	2
1.3. Outline.....	3
2 Background	4
2.1. The CAN standard	4
2.1.1. Standard CAN and Extended CAN.....	5
2.1.2. The Bit Fields of Standard CAN and Extended CAN	6
2.1.3. A CAN message.....	7
2.1.4. Message types.....	8
2.1.5. Error checking.....	10
2.1.6. The CAN bus	10
2.2. UAVCAN.....	11
2.2.1. Basic concepts.....	12

2.2.2.	Data structure description language	14
2.2.3.	CAN bus transport layer	18
2.2.4.	CAN frame format	19
2.2.5.	Payload bit and byte order	21
2.2.6.	Node requirements.....	21
2.3.	ArduPilot	21
2.4.	Pixhawk/PX4.....	22
2.5.	Arduino.....	24
2.6.	Olimexino	25
3	Development tools.....	26
3.1.	PX4 toolchain	26
3.2.	IAR Embedded Workbench	27
3.3.	Arduino IDE.....	27
3.4.	MissionPlanner.....	27
3.5.	GitHub GUI for Windows	28
4	Implementation.....	29
4.1.	PX4 / ArduPilot.....	29
4.1.1.	App	29
4.1.2.	Daemon	30
4.1.3.	Communication using the uORB	31
4.1.4.	New CAN message.....	33
4.1.5.	New CAN actuators.....	33
4.1.6.	Sending CAN messages	34

4.1.7.	Receiving CAN messages.....	35
4.2.	Olimexino	37
4.2.1.	Overview	37
4.2.2.	CAN filters.....	37
4.2.3.	Status message	38
4.2.4.	Button action	39
4.2.5.	Version differences	39
4.3.	Arduino.....	40
4.3.1.	Overview	40
4.3.2.	CAN Filters.....	40
4.3.3.	Status message	41
4.3.5.	Version differences	41
5	Results and demonstration	42
5.1.	uORB communication	42
5.2.	Sending CAN messages	43
5.3.	Receiving CAN messages.....	44
5.4.	ArduPilot integration.....	45
6	Conclusions and Recommendations	47
6.1.	Summary of the results	47
6.2.	Future work	48
	Bibliography	49
	Appendix A.....	53
	Appendix B.....	58

Appendix C.....63

Appendix D.....64

Appendix E.....72

List of figures

Figure 1 - The layered ISO 11898 standard architecture	5
Figure 2 - Standard CAN frame.....	6
Figure 3 - Extended CAN frame	7
Figure 4 - The inverted logic of a CAN bus	8
Figure 5 - CAN bus network configuration	11
Figure 6 – New version identifier field	19
Figure 7 – New version payload field	20
Figure 8 - Old version identifier field.....	20
Figure 9 - Old version payload field.....	21
Figure 10 - Payload bit and byte order examples	21
Figure 11 - The Pixhawk autopilot board.....	23
Figure 12 - Arduino diecimila (left). CAN-bus shield (right).....	25
Figure 13 - Olimexino board (left). ST-Link v2 (right).....	25
Figure 14 - Message travel.....	29

List of Tables

Table 1 - Primitive data types 17

List of acronyms

ACK	Acknowledge
ANSI	American National Standards Institute
CAN	Controller Area Network
CD+AMP	Collision Detection and Arbitration on Message Priority
CRC	Cyclic Redundancy Check
CSMA	Carrier-Sense, Multiple-Access
DLC	Data Length Code
DSDL	Data Structure Description Language
EOF	End Of Frame
GCC	Gnu Compiler Collection
GUI	Graphical User Interface
I2C	Inter-Integrated Circuit
ISO	International Organization for Standardization
IAR	Ingenjörfirman Anders Rundgren
IDE	Integrated Development Environment
IDE	IDentifier Extension
IFS	Inter-Frame space
LED	Light Emitting Diode
MAV	Micro-Air Vehicles
NSH	NuttShell
POSIX	Portable Operating System Interface
PWM	Pulse Width Modulation
RC	Radio Controlled
RTOS	Real Time Operating System
RTR	Remote Transmission Request
SCM	Source Code Management

SD	Secure Digital
SOF	Start Of Frame
SPI	Serial Peripheral Interface
SRR	Substitute Remote Request
SWD	Serial Wire Debug
UART	Universal Asynchronous Receiver/Transmitter
UAV	Unmanned Aerial Vehicle
UAVCAN	Unmanned Aerial Vehicle Controller Area Network
uORB	Micro Object Request Broker
USB	Universal Serial Bus

Chapter 1

Introduction

With the advent of smartphones came a push towards miniaturization of electronic systems. Powerful processors and sensors became available in small, cheap and power efficient packages. Taking advantage of that, the Unmanned Aerial Vehicle (UAV), commonly called drones, industry appeared. UAVs were no longer just the domain of militaries. From small and designed for entertainment to bigger ones designed for professional use, UAVs are increasingly common in everyday live. With the increase of the industry, new purposes are being found for these cheap and easy to use flying machines.

1.1. Motivation

With the improvement of any system comes complexity. That complexity prompts researchers to try to find ways of simplifying those systems. That was happening in 1983 when researcher at Robert Bosch GmbH began development of the Controller Area Network (CAN) [1]. Cars were having increasingly more sensors, each sensor had its own communications protocol and required separate wirings [2]. To reduce weigh and complexity the CAN bus was introduced. Requiring only two wires and a simple communications protocol, it was significantly simpler than what was available. It was quickly adopted and in 1993 the International Organization for Standardization (ISO) released the CAN standard ISO 11898 [3].

Something similar is happening today with Unmanned Aerial Vehicles (UAVs). Development is increasing and so is complexity. Currently sensors and actuators communicate using a mix of protocols designed for chip to chip communication and protocols used in the Radio Controlled (RC) hobby industry. Chip to chip protocols, like Serial Peripheral Interface (SPI) [4], were designed for communication with few elements and in close proximity, meaning they don't scale well. While protocols in the RC hobby industry, like Pulse Position Modulation (PPM) [5], were designed for point to point communication between two elements. The addition of more elements requires separate wiring harnesses, which do not scale well.

The logical place to look for suitable communications protocols would be the aerospace industry where weight and simplicity are main concerns. NASA was part of the development of a protocol, designed for general aviation, that uses the CAN bus, called CANaerospace [6].

However, being developed for full scale aircraft, CANaerospace is not adequate for small UAVs [7] because:

- CANaerospace has more than 50% of payload overhead which makes it unfit for high speed data streams like those in light machines where the controlled system requires high update rates and low latency.
- CANaerospace does not provide an easy way to pass multiple values at once.
- CANaerospace does not provide adequate means for common higher-level tasks, such as node configuration handling, firmware update and time synchronization.

Because of this, a new protocol is being developed, which is similar to CANaerospace, called Unmanned Aerial Vehicle Controller Area Network (UAVCAN) [8].

1.2. Objectives

This thesis proposes to create a working example of a CAN bus communication, using the UAVCAN protocol, with an autopilot. Per requested by UAVision the autopilot will be the ArduPilot and will be run in a Pixhawk board.

Such a work will require research about the three open source projects involved, the Pixhawk, the ArduPilot and the UAVCAN projects.

In addition, two nodes of the bus will be created. To demonstrate the flexibility of this communication scheme, two different architectures will be used for each node. One will be an Olimexino board, based on the ARM stm32f103 [9], running at 72 MHz. The other will be based on the Arduino diecimila board. A board based on the AVR atmega168 [10], running at 16 MHz, to provide an example in a resource constrained processor. The use of three different architectures will require different tools and development environments but better demonstrates the flexibility of the communication protocol.

A working example of a CAN bus will be produced that will allow each node to send and receive using the UAVCAN protocol. The Pixhawk will send messages to the Arduino, to control a servo motor, and receive temperature readings.

In the future this network can be used for structural monitoring.

1.3. Outline

After this, the introductory chapter, this thesis is made up of five other chapters, organized as follows:

- **Chapter 2** – Chapter 2 provides background knowledge on relevant topics and concepts necessary to understand the rest of this thesis.
- **Chapter 3** – Chapter 3 introduces the tools used in the development of this thesis.
- **Chapter 4** – Chapter 4 describes the construction of the working CAN bus network.
- **Chapter 5** – Chapter 5 provides details about the tests that were made to validate the correct functioning of the network.
- **Chapter 6** – Chapter 6 concludes this thesis with a summary of the results and gives suggestions for further improvements.

Chapter 2

Background

This chapter provides background knowledge on relevant topics and concepts necessary to understand the rest of this thesis

Firstly, the Controller Area Network (CAN) standard is introduced, with a description of what it is and how it works.

Next is an explanation of the protocol that will be used for communication.

And finally there is an introduction to the software and hardware used in this work.

2.1. The CAN standard

The CAN bus was originally developed for the automotive industry, to replace complex wiring harnesses with a two wire bus [1]. Because of the high immunity to electrical noise and the ability to self-diagnose and repair data errors the technology gained popularity in a variety of industries, from medical to automation and manufacturing.

The CAN bus is a multi-master, message broadcast system that specifies a maximum signaling rate of 1 megabit per second (Mbps) [11]. In a CAN network many short messages are broadcast to the entire network, which provides data consistency in every node of the system.

The CAN communications protocol describes how information is passed between devices on a network and can be defined in terms of layers.

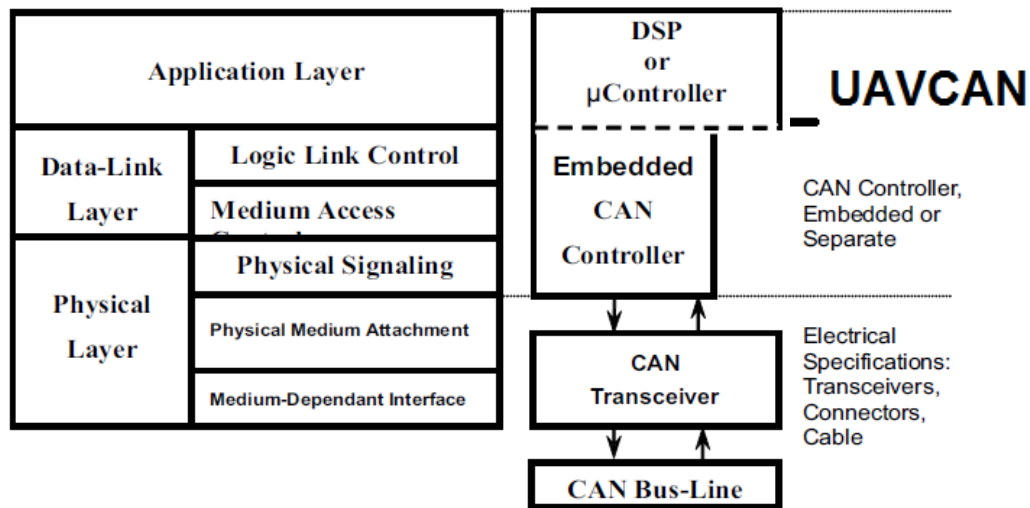


Figure 1 - The layered ISO 11898 standard architecture

As can be seen in Figure 1, the UAVCAN protocol is implemented at the microcontroller layer.

2.1.1. Standard CAN and Extended CAN

The CAN communication protocol is a Carrier-Sense, Multiple Access (CSMA) protocol with Collision Detection and Arbitration on Message Priority (CD+AMP). CSMA means that each node on a bus must wait for a prescribed period of inactivity before attempting to send a message. CD+AMP means that collisions are resolved through a bit-wise arbitration, based on a preprogrammed priority of each message in the identifier field of a message. The standard, initially worked with an 11-bit identifier, provides for signaling rates from 125kbps to 1 Mbps. The standard was later amended with the “extended” 29-bit identifier. The standard 11-bit identifier field in Figure 2 provides for 2^{11} , or 2048 different message identifiers, whereas the extended 29-bit identifier in Figure 3 provides for 2^{29} , or 537 million identifiers.

They are compatible with each other and can function in the same bus.

2.1.2. The Bit Fields of Standard CAN and Extended CAN

Standard CAN

Figure 2 is a representation of a standard CAN frame:

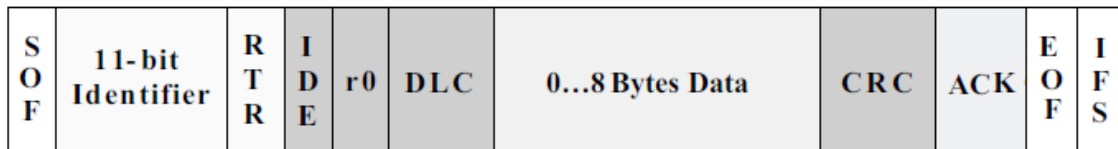


Figure 2 - Standard CAN frame

The meaning of the bit fields of Figure 2 are:

- **SOF** – The single dominant Start Of Frame (SOF) bit marks the start of a message, and is used to synchronize the nodes on a bus after being idle.
- **11 bit Identifier** – The Standard CAN 11-bit identifier establishes the priority of the message. The lower the binary value, the higher its priority.
- **RTR** – The single Remote Transmission Request (RTR) bit is dominant when information is required from another node. All nodes receive the request, but the identifier determines the specified node. The responding data is also received by all nodes and used by any node interested. In this way, all data being used in a system is uniform.
- **IDE** – A dominant single Identifier Extension (IDE) bit means that a standard CAN identifier with no extension is being transmitted.
- **r0** – Reserved bit (for possible use by future standard amendment).
- **DLC** – The 4-bit Data Length Code (DLC) contains the number of bytes of data being transmitted.
- **Data** – Up to 64 bits of application data may be transmitted.
- **CRC** – The 16-bit (15 bits plus delimiter) Cyclic Redundancy Check (CRC) contains the checksum (number of bits transmitted) of the preceding application data for error detection.
- **ACK** – Every node receiving an accurate message overwrites this recessive bit in the original message with a dominant bit, indicating an error-free message has been sent. Should a receiving node detect an error and leave this bit recessive, it discards the message and the sending node repeats the message after re-arbitration. In this way, each node ACKnowledges (ACK) the integrity of its data. ACK is 2 bits, one is the acknowledgment bit and the second is a delimiter.
- **EOF** – This End-Of-Frame (EOF), 7-bit field marks the end of a CAN frame (message) and disables bit-stuffing, indicating a stuffing error when dominant. When 5 bits of the same logic

level occur in succession during normal operation, a bit of the opposite logic level is stuffed into the data.

- **IFS** – This 7-bit Inter-Frame Space (IFS) contains the time required by the controller to move a correctly received frame to its proper position in a message buffer area.

Extended CAN

Figure 3 is a representation of an extended CAN frame:

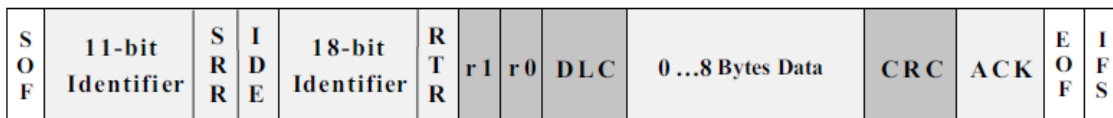


Figure 3 - Extended CAN frame

- **11 bit Identifier** – The first 11 bits of the extended identifier.
- **SRR** – The Substitute Remote Request (SRR) bit replaces the RTR bit in the standard message location as a placeholder in the extended format.
- **IDE** – A recessive bit in the Identifier Extension (IDE) indicates that more identifier bits follow. The 18-bit extension follows IDE.
- **18 bit Identifier** – The last 18 bits of the extended identifier.
- **r1** – Following the RTR and r0 bits, an additional reserve bit has been included ahead of the DLC bit.

2.1.3. A CAN message

A fundamental CAN characteristic, shown in Figure 4, is the opposite logic state between the bus, and the driver input and receiver output. Normally, a logic-high is associated with a one, and a logic-low is associated with a zero. However, in a CAN bus, a one is a logic low. This means the one is a recessive bit, allowing the lowest ID to have the highest priority.

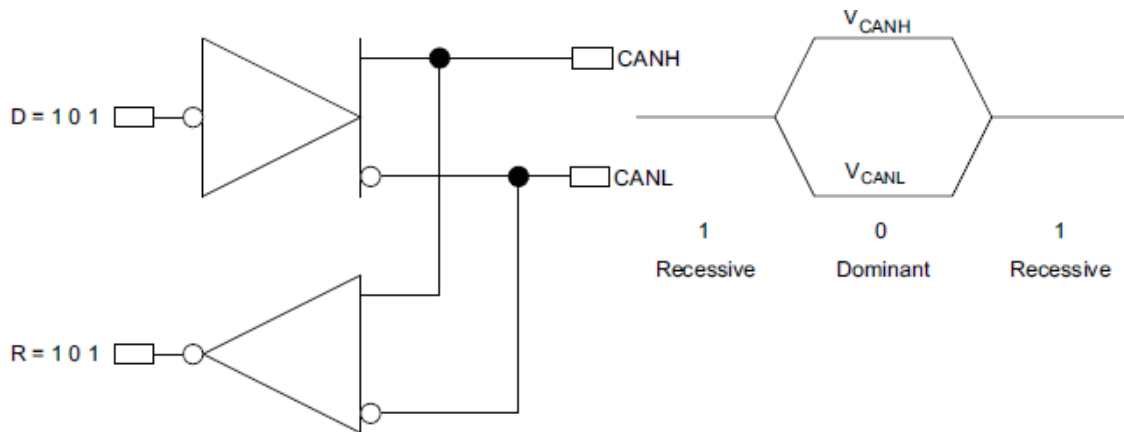


Figure 4 - The inverted logic of a CAN bus

Bus access is event-driven and takes place randomly. If two nodes try to occupy the bus simultaneously, access is implemented with a nondestructive, bit-wise arbitration. Nondestructive means that the node winning arbitration just continues on with the message, without the message being destroyed or corrupted by another node.

The allocation of priority to messages in the identifier is a feature of CAN that makes it particularly attractive for use within a real-time control environment. The lower the binary message identifier number, the higher its priority. An identifier consisting entirely of zeros is the highest priority message on a network because it holds the bus dominant the longest. Therefore, if two nodes begin to transmit simultaneously, the node that sends a last identifier bit as a zero (dominant) while the other nodes send a one (recessive) retains control of the CAN bus and goes on to complete its message. A dominant bit always overwrites a recessive bit on a CAN bus.

Note that a transmitting node constantly monitors each bit of its own transmission.

The allocation of message priority is up to a system designer.

2.1.4. Message types

The four different message types, or frames, that can be transmitted on a CAN bus are the data frame, the remote frame, the error frame, and the overload frame. Both the data and remote frame come from the application layer.

The data frame

The data frame is the most common message type, and comprises the Arbitration Field, the Data Field, the CRC Field, and the Acknowledgment Field. The Arbitration Field contains an 11-bit identifier in Figure 2 and the RTR bit, which is dominant for data frames. In Figure 3, it contains the 29-bit identifier and the RTR bit. Next is the Data Field which contains zero to eight bytes of data, and the CRC Field which contains the 16-bit checksum used for error detection. Last is the Acknowledgment Field.

The remote frame

The intended purpose of the remote frame is to solicit the transmission of data from another node. The remote frame is similar to the data frame, with two important differences. First, this type of message is explicitly marked as a remote frame by a recessive RTR bit in the arbitration field, and secondly, there is no data.

The error frame

The error frame is a special message that violates the formatting rules of a CAN message. It is transmitted when a node detects an error in a message, and causes all other nodes in the network to send an error frame as well. The original transmitter then automatically retransmits the message. An elaborate system of error counters in the CAN controller ensures that a node cannot tie up a bus by repeatedly transmitting error frames.

The overload frame

The overload frame is mentioned for completeness. It is similar to the error frame with regard to the format, and it is transmitted by a node that becomes too busy. It is primarily used to provide for an extra delay between messages.

A valid frame

A message is considered to be error free when the last bit of the ending EOF field of a message is received in the error-free recessive state. A dominant bit in the EOF field causes the transmitter to repeat a transmission.

2.1.5. Error checking

The CAN protocol incorporates five methods of error checking, three at the message level and two at the bit level. If a message fails any one of these error detection methods, it is not accepted and an error frame is generated from the receiving node. This forces the transmitting node to resend the message until it is received correctly. However, if a faulty node hangs up a bus by continuously repeating an error, its transmit capability is removed by its controller after an error limit is reached.

Error checking at the message level is enforced by the CRC and the ACK slots displayed in Figure 2 and Figure 3. The 16-bit CRC contains the checksum of the preceding application data for error detection with a 15-bit checksum and 1-bit delimiter. The ACK field is two bits long and consists of the acknowledge bit and an acknowledge delimiter bit.

Also at the message level is a form check. This check looks for fields in the message which must always be recessive bits. If a dominant bit is detected, an error is generated. The bits checked are the SOF, EOF, ACK delimiter, and the CRC delimiter bits

At the bit level, each bit transmitted is monitored by the transmitter of the message. If a data bit (not arbitration bit) is written onto the bus and its opposite is read, an error is generated. The only exceptions to this are with the message identifier field which is used for arbitration, and the acknowledge slot which requires a recessive bit to be overwritten by a dominant bit.

The final method of error detection is with the bit-stuffing rule where after five consecutive bits of the same logic level, if the next bit is not a complement, an error is generated. Stuffing ensures that rising edges are available for on-going synchronization of the network. Stuffing also ensures that a stream of bits is not mistaken for an error frame, or the seven-bit inter-frame space that signifies the end of a message. Stuffed bits are removed by a receiving node's controller before the data is forwarded to the application. With this logic, an active error frame consists of six dominant bits—violating the bit stuffing rule.

2.1.6. The CAN bus

Signaling is differential which is where CAN derives its robust noise immunity and fault tolerance.

Balanced differential signaling reduces noise coupling and allows for high signaling rates over twisted-pair cable. Balanced means that the current flowing in each signal line is equal but opposite in direction.

The High-Speed ISO 11898 Standard [12] specifications are given for a maximum signaling rate of 1 Mbps with a bus length of 40 m with a maximum of 30 nodes. It also recommends a maximum

unterminated stub length of 0.3 m. The cable is specified to be a shielded or unshielded twisted-pair with a 120Ω characteristic impedance (Z_0).

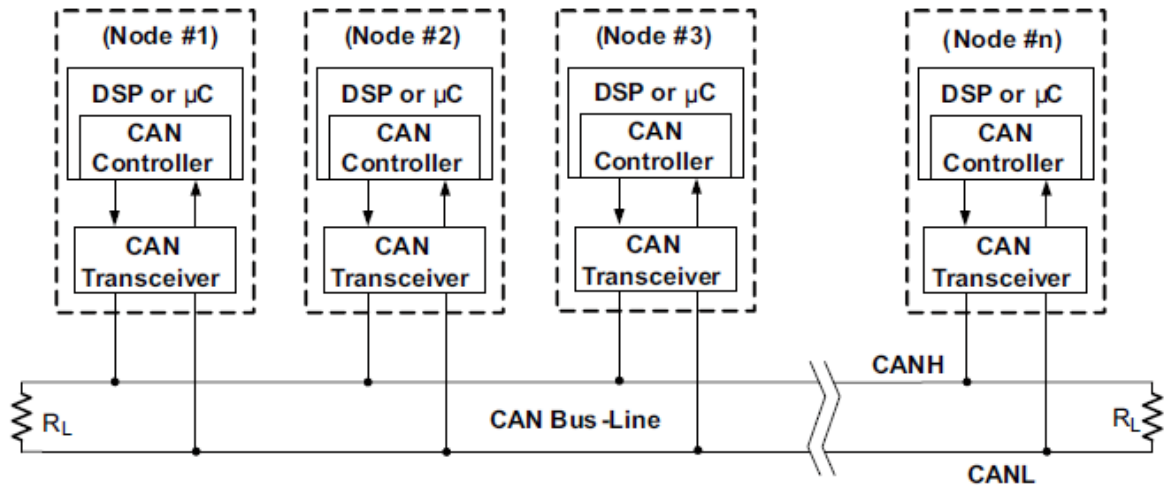


Figure 5 - CAN bus network configuration

As can be seen in Figure 5, the bus must be terminated with two resistances, R_L , of equal value to the cable, 120Ω .

2.2. UAVCAN

The Unmanned Air Vehicle Controller Area Network (UAVCAN) protocol is lightweight and designed for reliable communication in aerospace and robotic applications via the CAN bus. Here only the parts relevant to this work will be mentioned. The full specification can be found in [31]

According to its developer the protocol was developed with the following design goals:

- **Democratic network** - since the network does not require a master node (bus controller), it has no single point of failure.
- **Nodes shall be able to exchange long payloads** - typical use cases for a vehicle bus include the need to transfer sets of related parameters, where each parameter cannot be used independently. Such a set of parameters often does not fit into a single CAN frame, hence the need to split it into several CAN frames with a subsequent reassembly process on the receiving nodes.
- **Support for redundant interfaces and redundant nodes** - it is a common requirement for safety-concerned applications.

- **High throughput, low latency communication** - many applications require high-frequency, hard real-time control loops, which necessitates the need for a low-latency, high-throughput communication method.
- **Simple logic, low computational requirements** - UAVCAN targets a wide variety of embedded systems, from high-performance embedded on-board computers for intensive data processing to extremely resource-constrained microcontrollers. The latter imposes severe restrictions on the amount of logic needed to implement the protocol.
- **Common high-level functions should be clearly defined** - UAVCAN defines standard services and messages for common high-level functions, such as network discovery, node configuration, node firmware update, node status monitoring, network-wide time synchronization, dynamic node ID allocation.
- **Open specification and reference implementation** - the UAVCAN specification is open and freely available for anyone; the reference implementations are distributed under the terms of the MIT License.

There are two versions of the UAVCAN protocol, a new [31] and an old [32] one. When the versions differ, it will be stated.

2.2.1. Basic concepts

The UAVCAN network is a decentralized peer network, where each peer (node) has a unique numeric identifier, node ID. The nodes of the UAVCAN network can communicate using any of the following communication methods:

- **Message broadcasting** - The primary method of data exchange with publish/subscribe semantics.
- **Service invocation** - The communication method for peer-to-peer request/response interactions.

For each type of communication, a predefined set of data structures is used, where each data structure has a unique identifier, the data type ID. Some data structures are standard and defined by the protocol specification and some may be specific for a particular application or vendor.

Since every published message type has its own unique data type ID, and each node of the network has its own unique node ID, a pair of data type ID and node ID can be used to support redundant nodes with identical functionality inside the same network.

Message and service data structures are defined using the Data Structure Description Language (DSDL). DSDL description is used to generate the serialization and deserialization code for a given data structure. The DSDL approach allows to determine the data structure size statically, thus helping to optimize the protocol implementations in terms of memory consumption and performance. This feature is important for deeply embedded systems, where the memory footprint is critical and dynamic memory allocation may not be acceptable.

On top of the standard data types, UAVCAN defines a set of standard high-level functions like node health monitoring, network discovery, time synchronization, firmware update, and more. There's a part of the specification dedicated to the standard data types and application level functions.

Serialized message and service data structures are exchanged by means of the CAN bus transport layer, which implements automatic decomposition of long transfers into several CAN frames, allowing to exchange data structures of arbitrary size.

Message broadcasting

Message broadcasting refers to the transmission of a serialized data structure over the CAN bus to other nodes that are interested in receiving this data structure. This is the primary way of data exchange for UAVCAN.

A broadcasted message includes the following:

- **Payload** - The data structure.
- **Data type ID** - Numerical identifier that indicates how the data structure should be interpreted.
- **Source node ID** - The node ID of the transmitting node.
- **Transfer ID** - A small integer that increments with every transfer of this type of message from a given node.

Service invocation

Since it will not be used in this thesis, service invocation will be defined here but not further explained.

Service invocation is a two-step data exchange between exactly two nodes, the client and the server.

Service invocation is performed in three steps:

- The client sends a service request to the server.

- The server application takes appropriate actions and returns the response data.
- The server sends a service response to the client.

Typical use cases for this type of communication include node configuration parameter update and firmware update among other service tasks.

Both service request and service response include the following data:

- **Payload** – The data structure.
- **Data type ID** - Numerical identifier that indicates how the data structure should be interpreted.
- **Client node ID** - Source node ID during request transfer, destination node ID during response transfer.
- **Server node ID** - Destination node ID during request transfer, source node ID during response transfer.
- **Transfer ID** - A small overflowing integer that increments with every call to this service from a given node.

Both request and response contain exactly the same values for all fields except payload, whose content is application defined. Clients can match the response with a corresponding request using any of the ID fields.

2.2.2. Data structure description language

The Data Structure Description Language (DSDL) is used to define data structures for exchange via the CAN bus. Every data structure is defined in a separate DSDL definition file. The DSDL definition is then used to automatically generate the serialization and deserialization.

The tool that generates source codes from DSDL definition files is called the DSDL compiler.

DSDL definition file

A DSDL definition file defines exactly one data structure. The defined data structure can be used for either message broadcasting or service invocation.

Data type name is defined by the DSDL source file name as follows:

```
<data type name>.uavcan
```

Command.uavcan

The default data type ID, when used, is defined by the DSDL source file name as follows:

<default data type ID>.<data type name>.uavcan

341.Status.uavcan

It is not necessary to explicitly define a default data type ID for non-standard data types. All standard data types have default data type ID values defined.

Data type names are case sensitive and may contain nested data structures.

Syntax

A data structure definition consists of attributes and directives. Any line of the definition file may contain at most one attribute definition or at most one directive. The same line cannot contain an attribute definition and a directive at the same time.

An attribute can be either of the following:

- **Field** - a variable that can be modified by the application and exchanged via the network.
- **Constant** - an immutable value that does not participate in network exchange.

A directive is a statement that provides instructions to the DSDL compiler.

Aside from attributes and directives, a DSDL definition may contain the following entities:

- Comments
- Service response marker

A DSDL definition for a message data type may contain only the following:

- Attribute definitions (zero or more)
- Directives (zero or more)
- Comments (optional)

Attribute definition

Field definition patterns:

```
field_type field_name
field_type[X] field_name
field_type[<X] field_name
field_type[<=X] field_name
```

Constant definition pattern:

```
constant_type constant_name = constant_initializer
```

Each component is discussed below.

Field type

Can be either a primitive data type (primitive data types are defined below) or a nested data structure.

A primitive data type can be referred simply by name, while a nested data structure can be referred by either of these two:

- Short name, e.g., *NodeStatus*, if both the referred and the referring data types are located in the same namespace.
- Full name, e.g., *uavcan.protocol.NodeStatus*. A full name allows to reach the data type from any namespace.

A field type name can be appended with a statement in square brackets to define an array:

- *Syntax* `[X]` is used to define a static array of size exactly X items.
- *Syntax* `[<X]` is used to define a dynamic array of size from 0 to X-1 items, inclusively.
- *Syntax* `[<=X]` is used to define a dynamic array of size from 0 to X items, inclusively.

In the array definition statements above, X must be a valid integer literal according to the rules defined in the section dedicated to constant definitions. Arrays of maximum size with less than one item or multidimensional arrays are not allowed.

Field name and constant name

For a message data type, all attributes must have a unique name within the data type.

Constant definition

A constant is a primitive scalar type. A constant must be assigned with a constant initializer, which must be one of the following:

- Integer literal in base 10, starting with a non-zero character.
- Integer literal in base 16 prefixed with 0x.
- Integer literal in base 2 prefixed with 0b.
- Integer literal in base 8 prefixed with 0o.
- Floating point literal. Fractional part with an optional exponent part,
- Boolean true or false.
- Single ASCII character, ASCII escape sequence, or ASCII hex literal in single quotes.

Primitive data types

The UAVCAN standard assumes that these data types are built-in. The library used in this work provides the following data types:

Name	Bit length	Possible representation in C/C++	Value range	Binary representation
bool	1	bool	{0, 1}	One bit
intX	$2 \leq X \leq 64$	int8_t, int16_t, int32_t, int64_t	$[-(2X)/2, 2X/2 - 1]$	Two's complement
uintX	$2 \leq X \leq 64$	uint8_t, uint16_t, uint32_t, uint64_t	$[0, 2X - 1]$	
float16	16	float	± 65504	IEEE754 binary16
float32	32	float	Approx. ± 1039	IEEE754 binary32
float64	64	double	Approx. ± 10308	IEEE754 binary64

Table 1 - Primitive data types

Data type compatibility

All nodes exchanging some particular data structure must use compatible DSDL definitions. Different DSDL definitions are considered compatible if they share the same type and length.

Binary layout

Compatible data structures must feature the same field types in the same order.

Data type ID

The set of possible data type ID values is limited, so the devices from different vendors may occasionally reuse the same data type ID for different purposes. A part of the data type ID space is dedicated for standard data types. Another part of the data type ID space is dedicated for vendor-specific ID space.

2.2.3. CAN bus transport layer

A transfer that is addressed to all nodes except the source node is a broadcast transfer. A transfer that is addressed to one particular node is a unicast transfer. UAVCAN defines the following types of transfers:

- **Message transfer** – A broadcast transfer that contains a serialized message.
- **Service transfer** – A unicast transfer that contains either a service request or a service response.

Both message and service transfers can be further distinguished between:

- **Single-frame transfer** – A transfer that is entirely contained in a single CAN frame.
- **Multi-frame transfer** – A transfer whose payload is distributed over multiple CAN frames.

In the new version of the protocol, the transfer priority is an integer number that defines the urgency of the data contained in the transfer. In the older version, this priority is defined by the data type ID. Numerically lower priority values indicate higher urgency, and numerically higher values indicate lower urgency. Transfers of higher priority can delay transmission of transfers whose priority are lower.

Single frame transfer

If size of the entire transfer payload does not exceed the space available for payload in a single CAN frame, the whole transfer will be contained in one CAN frame. Such transfer is called a single-frame transfer. Single frame transfers are more efficient than multi-frame transfers in terms of throughput and latency.

2.2.4. CAN frame format

The UAVCAN protocol only uses 29 bit identifiers. The old and new versions differ in how the different field are organized at the bit level. Because of this they are presented separately.

New version

Identifier field

The identifier field for a message frame is composed of the following components:

- **Priority** – 5 bits representing the priority of the message.
- **Message type ID** – 16 bits corresponding to the message type ID.
- **Service not message** – 1 bit that indicates whether the frame is a service or a message.
- **Source node ID** – 7 bits representing the source node ID.

With the fields arranged in the following manner:

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							

Figure 6 – New version identifier field

Payload

The payload of a single frame transfer is composed of up to seven bytes of data followed by the tail byte. The tail byte consists of the following fields:

- **Start of transfer** – 1 bit that indicate the start of a transfer. For single frame transfers this value is always 1.
- **End of transfer** – 1 bit that indicates the end of a transfer. For single frame transfers this value is always 1.
- **Toggle bit** – 1 bit used to avoid CAN frame duplication in multi-frame transfers. For single frame transfers this value is always 0.
- **Transfer ID** - 5 bits representing an integer value that allows receiving nodes to distinguish this transfer from others.

With the fields arranged in the following manner:

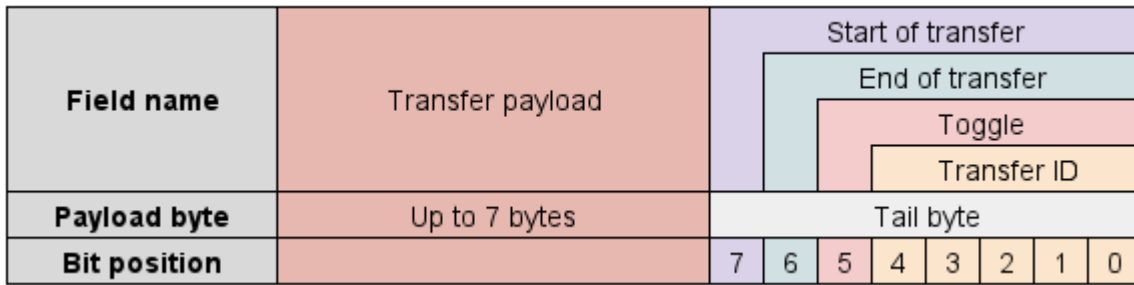


Figure 7 – New version payload field

Old version

Identifier field

The identifier field for a message frame is composed of the following components:

- **Data type ID** – 10 bits corresponding to the message type ID.
- **Transfer type** – 2 bits that determine the type of transfer. 10 for message broadcast.
- **Source node ID** – 7 bits representing the source node ID.
- **Frame index** – 6 bits used to know the order of multi-frame transfers. For single frame transfers this value is always 00000.
- **Last frame** – 1 bit that indicates the end of a transfer. For single frame transfers this value is always 1.
- **Transfer ID** - 3 bits representing an integer value that allows receiving nodes to distinguish this transfer from others.

The fields are arranged in the following manner:

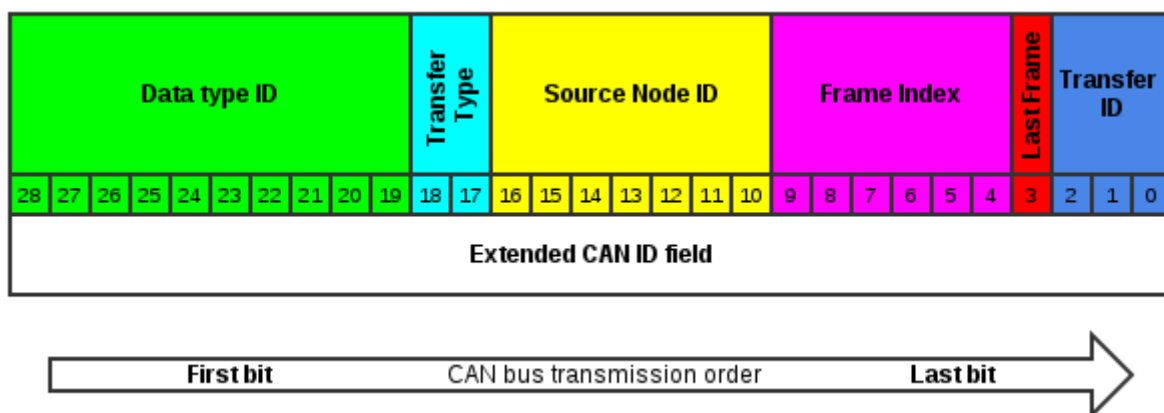


Figure 8 - Old version identifier field

Payload

The payload of a single frame transfer is composed solely of the message to be transmitted, like so:

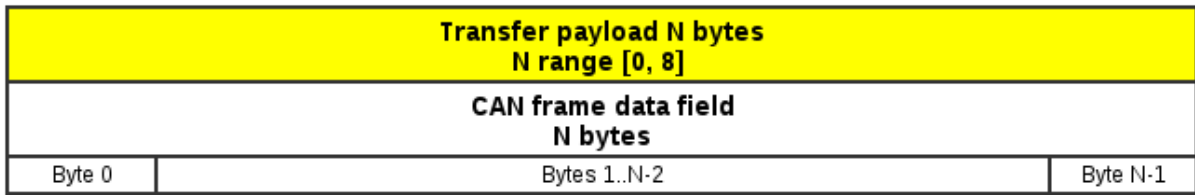


Figure 9 - Old version payload field

2.2.5. Payload bit and byte order

Bits are filled from the most significant bit. Bytes are sent in a little-endian, this means that the least significant bytes goes first. Any pad bits are set to zero.

Figure 10 provides examples to clarify the bit and byte order.

				Byte index																							
				0		1		2		3																	
				0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7								
				7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
				Encoded bit values																							
Value index	Value to encode	Target bit length	Binary representation before truncation	4 most significant bits are truncated; the value is then converted to little endian representation																							
0	0xBEDA	12	101111101101010	Two's complement																							
1	-1	3	111	Two's complement																							
2	-5	4	1011	Two's complement																							
3	-1	2	11	Two's complement																							
4	0x88	4	10001000	4 most significant bits are truncated																							
N/A	Alignment	N/A	All zero	The encoded message must be byte aligned																							

Figure 10 - Payload bit and byte order examples

2.2.6. Node requirements

Every UAVCAN node must report its status and presence by means of broadcasting messages of type *uavcan.protocol.NodeStatus*. This is the only data structure that UAVCAN nodes are required to support. For the most recent version the default data type ID is 341 and for the older one the data type ID is 550.

2.3. ArduPilot

The ArduPilot (or ArduPilot Mega) firmware is an open-source Unmanned Aerial Vehicle (UAV) autopilot [13]. It allows for the control of vehicles, autonomous or remote, of different configurations. It has been used to control boats, rovers, planes and various multirotors.

Developed, originally, for the Arduino open-source electronics prototyping platform, it has been ported to a different hardware platform called Pixhawk/PX4, presented later in this work. This hardware uses a small Real Time Operating System (RTOS) and the ArduPilot runs as an app.

The ArduPilot project also provides a program, called MissionPlanner, to interface the autopilot with the computer [14]. The MissionPlanner serves as a ground station to the vehicle, allowing its control or mission monitorization and configuration.

The source code is downloaded through from GitHub and has the following structure:

- AntennaTracker – Allows the control of the ground station antenna.
- APMRover – Source code for the rover.
- ArduCopter – source code for multicopter. It allows for a configurable number of motors.
- ArduPlane – Source code for the plane.
- Libraries – Contains the libraries necessary for the ArduPilot code.
- mk – Contains the makefiles.
- Modules – Contains the UAVCAN library, the operating system and the Pixhawk firmware.

The operating system that comes with the ArduPilot code is an older version. Because the UAVCAN library comes with the operating system, it is an outdated version as well.

The ArduPilot project is being developed using the toolchain from the Pixhawk project presented next. However, the toolchain downloads the most recent version of the operating system, one that comes with the newer version on UAVCAN. The different versions of the protocol are incompatible. This can lead to some confusion. Because of this, the nodes of the CAN network were made to work with both versions.

When the different versions require different action it will be stated.

2.4. Pixhawk/PX4

The Pixhawk project was initially developed at the Swiss Federal Institute of Technology in Zurich (ETH Zürich) for Micro Air Vehicle (MAV) research [15]. The project produced a MAV with powerful control hardware. The project was eventually open-sourced and development was opened to the community with the PX4 autopilot project [16].

2.4.1. PX4 Hardware

The latest version of the board, presented in Figure 11 and the one used in this work, uses a 32bit ARM processor, based on the Cortex M4F architecture running at 168 MHz [17]. It has 14 Pulse Width Modulation (PWM) outputs, several connectivity standards like Universal Asynchronous Receiver/Transmitter (UART), Inter-Integrated Circuit (I2C) and CAN as well as other Radio Controlled (RC) hobby communication standards. It has a failsafe processor based on the Cortex M3.

For sensing, it possesses two 3 axis accelerometers, two 3-axis gyroscopes, one 3-axis magnetometer and one barometer.

There are also a multicolor Light Emitting Diode (LED), a multi-tone piezo audio indicator, a safety switch and a Secure Digital (SD) card for logging over extended periods of time.

It is programmed over the Universal Serial Bus (USB) port using the MissionPlanner software introduced earlier.



Figure 11 - The Pixhawk autopilot board

2.4.2. PX4 Firmware

The PX4 hardware runs a small, lightweight and efficient operating system NuttX [18], which provides a Portable Operating System Interface (POSIX) style environment.

NuttX

The NuttX is a real time operating system (RTOS) designed for resource constrained systems [19]. It is scalable from 8-bit to 32-bit systems with an emphasis on a small footprint and standards compliance, like the POSIX and American National Standards Institute (ANSI) standards. It also provides a command line called NuttShell (nsh). This command line is usually available through the usb port. However, the ArduPilot code disables this, so a serial-to-USB [20]converter had to be used.

2.4.3. PX4 Middleware

The PX4 middleware runs on top of the operating system and provides device drivers and a micro Object Request Broker (uORB) for asynchronous communication between individual tasks.

uORB

The micro Object Request Broker (uORB) application is used to share data structures between threads and applications, using a simple implementation of the publish-subscribe pattern [21].

The publish–subscribe is a messaging pattern where the senders, called publishers, make the message available as classes, without knowledge of the receivers or if they exist. Likewise, the receivers, called subscribers, express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

2.5. Arduino

Arduino is an open-source hardware and software company, project and user community that designs and manufactures microcontroller-based prototyping boards [22]. The project is based on a family of microcontroller board designs using various 8-bit Atmel AVR microcontrollers or 32-bit Atmel ARM processors.

All Arduino boards are programmed through the Arduino Integrated Development Environment (IDE), using a simplified version of the C++ programming language.

The board used in this work, the diecimila [23], has a serial to usb converter that allows for easy communication and programming using a computer. It uses an AVR ATmega168 running at 16 MHz. The board is designed to give easy access to the microcontroller pins and to allow the use of shields, where a compatible board is plugged directly on top and becomes ready to use.

In this thesis the Arduino will simulate a sensor in the CAN bus.

To communicate in the CAN bus it needs a CAN controller and driver. To do so, a shield is used. The chosen shield is the *CAN-BUS Shield V1.2* [24] produced by Seeed Studio. It uses a MCP2515 CAN Bus controller [25] with SPI interface and MCP2551 CAN transceiver [26].

Both the Arduino board and the shield can be seen in Figure 12.

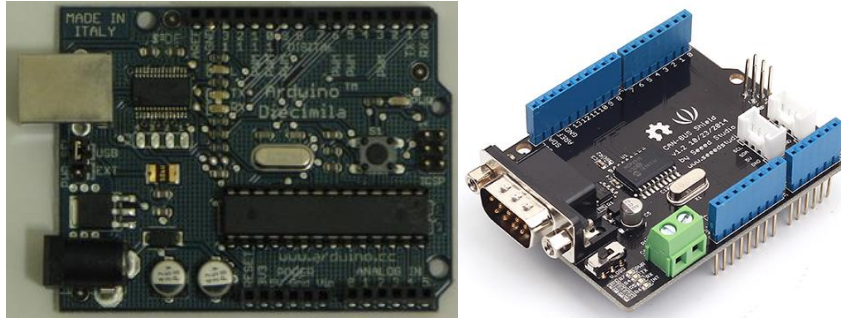


Figure 12 - Arduino diecimila (left). CAN-bus shield (right)

2.6. Olimexino

The Olimexino-stm32 is an open source prototyping board with a stm32 microcontroller, based on the ARM cortex-M3 architecture [27].

The microcontroller has an integrated CAN driver, with the board containing the transceiver, so that no additional hardware is required to communicate in the CAN-bus. It is designed with the same header configuration as the Arduino, and can be programmed with an Arduino-like IDE. However, due to a lack of support for the CAN driver a ST-Link V2 [28] is used for programming, via the serial wire debug (SWD) [29] port. Given the availability of a functional CAN-bus example written for the IAR Workbench IDE [30], that was the chosen programming environment.

Both the Olimexino board and the ST-Link V2 programmer can be seen in Figure 13.

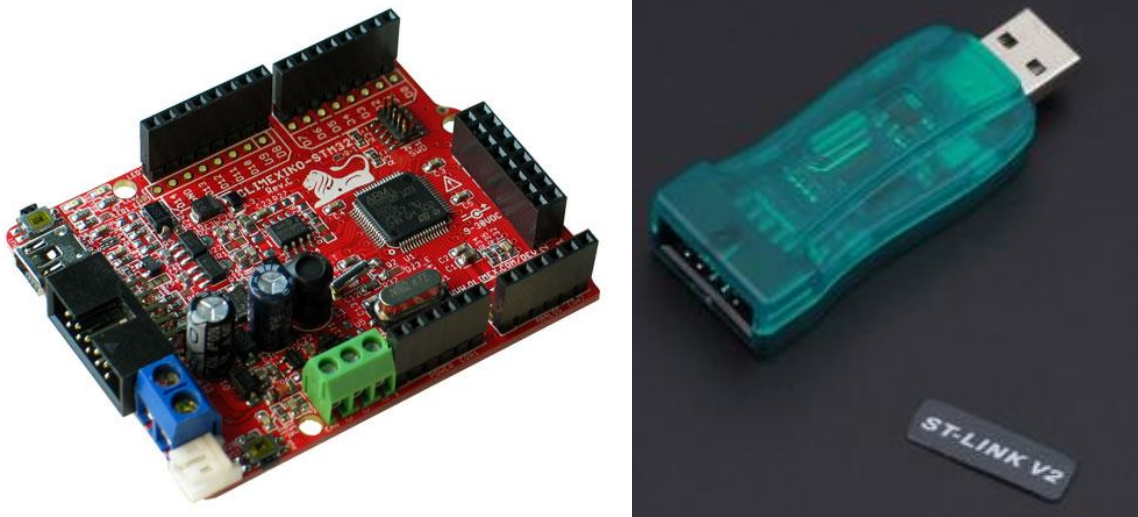


Figure 13 - Olimexino board (left). ST-Link v2 (right)

Chapter 3

Development tools

In this chapter the development tools used in this work are presented. There is a brief explanation of what they are, followed by a description of how they were used.

3.1. PX4 toolchain

The PX4 project has created a software toolchain [33] to facilitate the entry of new developers.

This toolchain installs and configures the software and source code necessary to begin development and to ensure equal configuration among developers.

The toolchain installs the following software:

- **Cygwin** - Cygwin [34] is a free and open source Unix-like environment and command-line interface for Microsoft Windows. It translates POSIX system calls to ordinary Windows system calls, allowing software designed for UNIX systems to run on Windows.
- **MinGW** - MinGW [35] (Minimalist GNU for Windows), is a free and open source software development environment for creating Microsoft Windows applications. It includes a command line and tools like GDB and make.
- **TeraTerm** – TeraTerm [36] is an open-source terminal emulator program. It emulates different types of computer terminals and supports port connections.
- **HTerm** - HTerm [37] is an open-source terminal emulator program. It supports communication serial port connections.
- **Eclipse** - Eclipse [38] is a free and open-source Integrated Development Environment (IDE). It contains a base workspace and an extensible plug-in system for customizing the environment. It allows for the development of C/C++ programs, among other languages.
- **GCC ARM embedded toolchain** – The GNU Compiler Collection (GCC) toolchain for ARM [39] is a free and open-source toolchain that allows for the development of software for Arm Cortex-M and Cortex-R processors (embedded families).

For this work the GCC ARM embedded toolchain needed to be manually updated to the 4.7 version [40] for the correct compilation of the program.

3.1.1. Compiling the code

Development was made in both the ArduPilot source code and the PX4 firmware alone. This requires different command to be run in different places.

- ArduPilot – In the folder of the vehicle configuration desired (ex. ArduCopter) run *make PX4-v2*.
- PX4 Firmware – In the firmware folder run *make PX4fmu-v2_default*.

All commands are to be run in the PX4 console, i.e. the installed MinGW command line.

3.2. IAR Embedded Workbench

The IAR Embedded Workbench is a paid development environment produced by Ingenjörfirman Anders Rundgren (IAR) Systems. The workbench includes a C/C++ compiler and debugger and can be used to program and debug the STM32 family of processors. The trial version of the 6.5 edition was used because examples for the Olimexino board were produced using this environment.

3.3. Arduino IDE

The Arduino IDE is an application built for the Arduino prototyping platform. It includes a C/C++ code editor and is also capable of compiling and uploading programs to the board with a single click

It comes with software libraries and examples to facilitate the development of new applications. Shields for the Arduino boards usually contain a library and examples for the same reason. In this thesis the example application for the *CAN-BUS Shield V1.2* will be modified to serve our purpose

3.4. MissionPlanner

The MissionPlanner is a free and open-source application produced by the ArduPilot project. It serves both as a ground station, receiving telemetry or controlling the vehicle, and allows for the configuration of the mission and the autopilot.

It is used to upload new code to the PX4 autopilot board.

3.5. GitHub GUI for Windows

The GitHub Graphical User Interface (GUI) for Windows is an application for Microsoft Windows that allows for an easier way to use the Git software with GitHub. It offers all of the distributed revision control and Source Code Management (SCM) functionality of Git as well as extra features

It is used to update the ArduPilot and PX4 source code without overriding local changes.

Chapter 4

Implementation

This chapter describes the implementation of the working CAN bus network.

It starts with the creation of applications to test the communication between processes in the NuttX operating system. Then there is the creation of the CAN network, how a new message is created received and handled in the operating system.

The last part introduces the creation of the two nodes that will be used to test the network.

As can be seen in the following diagram, a message travels from the autopilot to the UAVCAN driver through the micro Object Request Broker (uORB), then to the UAVCAN network where it arrives at the nodes.

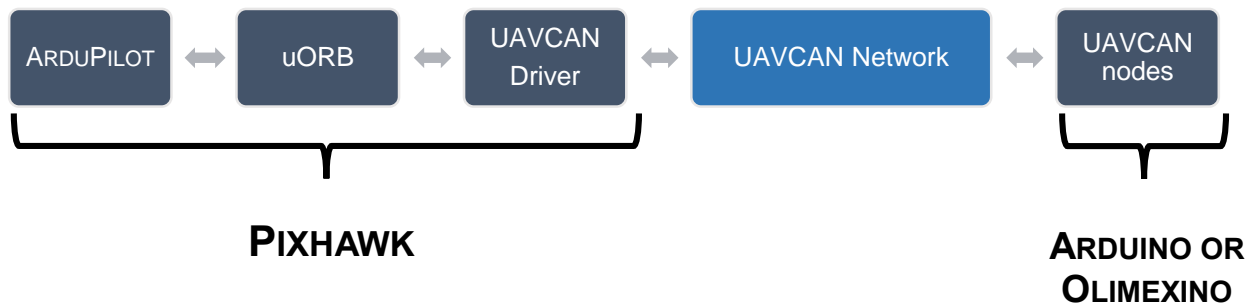


Figure 14 - Message travel.

4.1. PX4 / ArduPilot

4.1.1. App

The simplest process in the NuttX operating system is the app. It differs from the daemon in that, once started, it controls the command line. To create an app one must first create a folder in one of the folders inside the one named *src*. It can be any name but is advisable to be the name of the app. In this case, a folder called *teste_1* was created in the examples folder.

Inside *teste_1* needs to exist a makefile file called *module.mk*. This file must contain the name of the app and the source files like so:

```
MODULE_COMMAND      = teste_1
SRCS                 = teste_1.c
```

For this simple app only one source file is needed.

To create the app only a main function is needed and must have the name on the app followed by *_main* like so:

```
__EXPORT int teste_1_main (int argc, char *argv[]);
```

The *__EXPORT* keyword is needed because it causes the function name to be exported to the linker. Note that the app can be programmed in C++ (or C in a *.cpp* file) but needs the *EXTERN "C"* keyword to prevent the compiler from mangling the function name.

Lastly, one needs to add the new app to the makefile. To do so it is necessary to add the following to the pertinent makefile:

```
MODULES             += examples/teste_1
```

For our target board the makefile is the one with *config_PX4fmu-v2* in the name.

After a successful compilation and upload to the board, the app can now be called from the command line or from the starting script.

Simply writing the name of the app in the command line starts the app.

```
nsh>teste_1
```

4.1.2. Daemon

The daemon differs from the app in that it runs in the background and frees the command line after being called.

To create a daemon one can start from the app source code from before and make the necessary changes.

In NuttX daemons are created using a new thread that is started from the main function. It is customary to include the commands *start*, *stop* and *status* for interaction with the daemon. The main function

from before needs to create a new thread the first time the start argument is given, give the thread the signal to stop in case the argument is stop and print the status if the status argument is given.

A new thread is started with the following command:

```
PX4_task_spawn_cmd (const char *name, int scheduler, int priority, int
stack_size, main_t entry, char * const argv[])
```

The name argument is the name that appears in the running processes list. The *main_t* entry argument is the name of the function that will be run in the thread. Calling this daemon *teste_1* and the new function *teste_1_daemon_app_main*, with the rest of the arguments with default values, the call is:

```
PX4_task_spawn_cmd ("teste_1", SCHED_DEFAULT, SCHED_PRIORITY_DEFAULT, 2000,
teste_1_daemon_app_main, (argv) ? (char * const *)&argv[2] : (char * const
*)NULL);
```

The new function can be considered the new main function and has the following declaration:

```
int teste_1_daemon_app_main(int argc, char *argv[]);
```

To start the daemon it just needs to be called with the *start* argument:

```
nsh>teste_1 start
```

4.1.3. Communication using the uORB

Despite existing other communication methods in NuttX, the uORB was chosen because of its simplicity and because information about the sender or receiver is not necessary.

Because the uORB uses a publisher-subscriber pattern any number of publishers and subscribers can exist for each topic. Every app or daemon can be a publisher, a subscriber or both.

Before communication can exist a topic must be created. To do this one needs to create a file in the *msg* folder, the name of the file will be the name of the topic. Here the file was called *teste_topic.msg* and contains:

```
uint16 inc
```

This message of this topic is a 16 bit unsigned integer called *inc*. At compilation time a new file is created, called *teste_topic.h* with a *teste_topic_s struct* that contains the variable *inc*. Any publisher or subscriber must include this file.

After a topic has been created, it needs to be defined with the uORB. To do so one needs to add to the file *objects_common.cpp* the following:

```
#include "topics/teste_topic.h"
ORB_DEFINE (teste_topic, struct teste_topic_s);
```

ORB_DEFINE is a macro that defines the name of the topic and the structure it passes.

Now that the topic has been created it can be published and subscribed to, in any order.

Publishing

Before a topic can be published to, it needs to be advertised. The function that advertises a topic publishes the first information and returns a file descriptor that is the topic handle. To advertise the topic the function must be called like so:

```
struct teste_topic_s test = {.inc = 0};
orb_advert_t topic_handle = orb_advertise(ORB_ID(teste_topic), &test);
```

Multiple publisher to the same topic can exist but one publisher must close the topic handle before another can advertise and publish.

To publish one just needs to call the *orb_publish* function like so:

```
orb_publish(ORB_ID(teste_topic), topic_handle,&test)
```

Subscribing

Before a topic can receive messages it must first subscribe to a topic. Multiple subscriber can exist at the same time without conflict. To subscribe the following function must be called like so:

```
static int topic_handle=orb_subscribe(ORB_ID(teste_topic));
```

Unlike with the advertisement, where the topic handle is an *orb_advert_t* struct, with the subscription the topic handle is an integer.

A topic can be subscribed to, even if there are no publishers.

The subscriber can check for updates to the topic using:

```
bool updated;
orb_check(topic_handle,&updated)
```

If updates are available they can be copied by calling the function *orb_copy*:

```
struct teste_topic_s test
orb_copy(ORB_ID(teste_receive),topic_handle,&test)
```

The subscriber can also set the interval at which it sees updates using the following function:

```
orb_set_interval(topic_handle,1000)
```

Where *1000* is the interval in milliseconds. In this case the subscriber will see updates once every second, given they are available.

4.1.4. New CAN message

To create new CAN messages a folder was created in the path *uavcan/dsdl/uavcan/equipment*.

Inside this folder two files were created, called *760.TesteCommand.uavcan* and *761.TesteReceive.uavcan*. The number part of the name is the data type ID. The *TesteCommand* file is used for the send variable and contains:

```
uint8 valor
```

The *TesteReceive* file is used for the receive variable and contains:

```
uint8 recebe
```

At compile time a file is created for each message. These new files contain new types with the file names and the declared variables.

4.1.5. New CAN actuators

To send and receive CAN messages it is advisable to create separate logic to handle the send and receive functions. To do that two new files were created, one code file and one header file, called *teste_actuator*. Unless stated otherwise, the code that follows was added to these files.

A new class is created called *UavcanTeste*:

```

class UavcanTeste
{
public:
    UavcanTeste(uavcan::INode &node);
    ~UavcanTeste();

private:
}

```

This class contains the logic necessary to handle both the incoming and outgoing messages. In this thesis the messages is read from a uORB topic called *teste_topic* and sent to the CAN bus using the message *TesteCommand*. The incoming messages *TesteReceive* are read from the CAN bus and written to a uORB topic called *teste_receive*.

4.1.6. Sending CAN messages

To be able to publish messages a new function was created that is then run from the main UAVCAN thread, in the function *UavcanNode::run()*. The main UAVCAN thread is present in the files called *uavcan_main*.

The new function is called *update_outputs*. A new publisher for the required message must also be instantiated, here called *_uavcan_pub_teste_cmd*.

The following code is added:

```

void update_outputs(int output);
uavcan::Publisher<uavcan::equipment::teste::TesteCommand>
_uavcan_pub_teste_cmd;

```

The function *update_outputs* must receive the input, place it in the message to be sent and broadcast the message, like so:

```

void UavcanTeste::update_outputs(int output)
{
    uavcan::equipment::teste::TesteCommand msg;
    msg.value = output;
    (void)_uavcan_pub_teste_cmd.broadcast(msg);
}

```

4.1.7. Receiving CAN messages

In order to receive messages the code first needs to bind the message type to the function that will handle it. Then the order to start receiving this type of message needs to be given. The code to do this is as follows:

```
int init();

void teste_receive_sub_cb(const
uavcan::ReceivedDataStructure<uavcan::equipment::teste::TesteReceive> &msg);

typedef uavcan::MethodBinder<UavcanTeste *,void (UavcanTeste::*)(const
uavcan::ReceivedDataStructure<uavcan::equipment::teste::TesteReceive>&)>Stat
usCbBinder;

uavcan::Subscriber<uavcan::equipment::teste::TesteReceive, StatusCbBinder>
_uavcan_sub_status;
```

The function `init()` gives the order to start receiving messages from the bus.

```
int UavcanTeste::init()
{
    int res = _uavcan_sub_status.start(StatusCbBinder(this,
&UavcanTeste::teste_receive_sub_cb));
    if (res < 0) {
        return res;}
    return 0;
}
```

The function `init()` is run once in `UavcanNode::init()`, in the UAVCAN main thread, with the following code:

```
ret = _teste_controller.init();
if (ret < 0) {
    return ret;
}
```

The variables to publish to the uORB topic need to be initialized:

```
teste_receive_s _teste_receive = {.inc = 0};
orb_advert_t _teste_receive_pub = nullptr;
```

When the messages arrive they are published to a uORB topic. That is done in the `teste_receive_sub_cb` function, like so:

```

void UavcanTeste::teste_receive_sub_cb(const
uavcan::ReceivedDataStructure<uavcan::equipment::teste::TesteReceive> &msg)
{
    _teste_receive.inc=msg.recebe;

    uint8_t node= msg.getSrcNodeID().get();
    printf("node=%d\n",node);
    if (_teste_receive_pub != nullptr) {
        (void)orb_publish(ORB_ID(teste_receive), _teste_receive_pub,
&_teste_receive);

    } else {
        _teste_receive_pub = orb_advertise(ORB_ID(teste_receive),
&_teste_receive);
    }
}

```

The above function also prints, to the console, the ID of the source node of the message, with the following code:

```

uint8_t node= msg.getSrcNodeID().get();
printf("node=%d\n",node);

```

4.1.8. Integrating with ArduPilot

To communicate with the CAN network the ArduPilot code has to publish and subscribe to the uORB topics used before. To test this, the code used before to publish and subscribe to uORB topics was reused. To avoid flooding the system with messages, a one second loop was used.

```

// one_hz_loop - runs at 1Hz
void Copter::one_hz_loop()

```

Due to a bug in the ArduCopter code, any added code leads to a crash in the NuttX operating system. Because of this the ArduPlane code is used. In the ArduPlane the one second loop is:

```

// one_hz_loop - runs at 1Hz
void Plane::one_hz_loop()

```

4.2. Olimexino

4.2.1. Overview

The configuration of the stm32 processor is outside the scope of this work. Since the manufacturer of this board provided the source code for a CAN bus node [41], that code was adapted.

The code provided consisted of the configuration parameters of the microprocessor, like memory mapping and clock speed, and the parameters for the board, like LEDs and buttons.

To serve as a node in a UAVCAN network the board must send a status message once every second. A timer was used that triggered, every second, a function that sent a status message. This timer was also used to track the uptime needed in the status message.

A function was made to handle the incoming messages. In this example the messages were recorded to be later read in the debugger.

The sending of a message was controlled through a push button present on the board. When the button was pressed, a message with an integer was sent. The number is incremented after every press.

4.2.2. CAN filters

To free the processor, the filtering is made in hardware. The filtering is made by setting IDs and masks. The ID is the ID of the message to be received. The mask is a binary number that determines which bits must match between the filter ID and the message ID. If a bit in the mask is 1, the corresponding bit in the IDs must match, if it is a 0 they may differ.

Since this is a 32 bit processor and the extended CAN IDs used are 29 bits, there are extra bits in the register. In this processor the registers for the filter ID and mask are left justified. This means that the three least significant bits of the 32 bit register are not part of the CAN ID.

The configuration code of the filters is as follows:

```
uint32_t IDi=398722056<<3;// Idi = 10111110001000000010000001000
CAN_FilterInitStructure.CAN_FilterIdHigh = IDi>>16;
CAN_FilterInitStructure.CAN_FilterIdLow = (IDi & 0x0000FFFF);
CAN_FilterInitStructure.CAN_FilterMaskIdHigh = 0xFFFF;
CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0xE000;
```


The filter ID used stops all messages except the ones with a data type ID of 760 that corresponds to the *TesteCommand* message. The mask allows any source node ID if the message has the correct type.

If a message passes the filters an interrupt is triggered. This interrupt toggles the state of a LED, for visual confirmation of reception, and stores the CAN IDs in an array that is read later in the debugging session.

```
void USB_LP_CAN1_RX0_IRQHandler(void){
    LED_Toggle(LED2);
    CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
    if(i<200){
        entradas[i]=RxMessage.ExtId;
        i++;}
}
```

4.2.3. Status message

A timer was configured that triggers an interrupt once every second. This interrupt publish a status message with the uptime, in seconds, of the node. This timer also increments the variable that counts the uptime. The publication is made using the following function:

```
void publish_status(uint8_t health, uint8_t mode){
    uint8_t payload[7];

    // Uptime in seconds
    const uint32_t uptime_sec2=uptime_sec;
    payload[0] = (uptime_sec2 >> 0) & 0xFF;
    payload[1] = (uptime_sec2 >> 8) & 0xFF;
    payload[2] = (uptime_sec2 >> 16) & 0xFF;
    payload[3] = (uptime_sec2 >> 24) & 0xFF;

    // Health and mode
    payload[4] = (health << 6) | (mode << 3);
    payload[5] = 0x00;
    payload[6] = 0x00;

    static uint8_t transfer_id;
    transfer_id +=1;
```

```

    static const uint16_t data_type_id = 550;
    uavcan_broadcast_old( data_type_id, transfer_id, payload,
sizeof(payload));
}

```

4.2.4. Button action

For convenience the available button was programmed to send a message when pressed. The message simply has an incrementing number as a payload and 761 as Data type ID, which corresponds to the *TesteReceive* type. For visual confirmation, the button press also toggles the state of the second LED.

The button pressing triggers this function:

```

int publish_frame_old(void){
    static uint8_t carga=0;

    uint8_t payload[1];
    payload[0] = carga;

    carga +=1;
    static const uint16_t data_type_id = 761;
    static uint8_t transfer_id;

    LED_Toggle(LED1);
    int ret = uavcan_broadcast_old( data_type_id, transfer_id, payload, 1);
    transfer_id += 1;
    return ret;
}

```

4.2.5. Version differences

As stated earlier there are two incompatible versions of the UAVCAN standard. The one currently used by the ArduPilot project is the older one. The overall logic of the node is the same with only a few changes. When those changes were necessary to work with both versions, different functions were created and chosen via preprocessor macros.

4.3. Arduino

4.3.1. Overview

Programming in the Arduino environment is extremely simple because the configuration variables are automatically defined. Given that the shield manufacturer provides a library to interface with the CAN driver and an example [42], the example was adapted. The basic functionality is the same as in the Olimexino. It sends and receives messages of data type ID 761 and 760, respectively. In this case the node sends an incrementing value every three seconds and prints any message received to the serial port, which is then read in the computer.

Later a servo was attached to the Arduino and controlled through CAN. A temperature sensor was also read and its value sent to the Pixhawk.

4.3.2. CAN Filters

To receive only the required messages the hardware filters were set. The filters are again based on a filter ID and a filter mask, where the mask determines which bit must match. To do this the following code is used:

```
CAN.init_Mask(1, 1, 0xFFFFFFFF);
CAN.init_Mask(0, 1, 0x1FFFC08);
CAN.init_Filt(0, 1, PX4);
CAN.init_Filt(1, 1, status_);
```

Here two filters and two mask are set. One receives the messages with data type 760 while the other receives the status messages from the Pixhawk. The status messages are received to allow for a workaround explained next.

4.3.3. Status message

Because the libraries in use were already using the available timer, none was available to trigger a status message. So a workaround is used. Every time a status message is received from the Pixhawk, a status message is sent. Since the status message arrives in intervals of roughly one second, it was also used to trigger the message with data type ID of 761. For every three status messages received one message is sent.

4.3.4. Servo and temperature sensor

Both a servo and a temperature sensor were attached to the Arduino. The servo is controlled through the receive messages. The message is received, the value is mapped from a range of 0 to 255 to a range of 0 to 180 and sent to the servo. The value was mapped and sent to the servo using the following code:

```
val = map(val, 0, 255, 0, 180);  
myservo.write(val);
```

The temperature sensor created an analog value corresponding to the temperature, that value was read and sent to the CAN bus.

```
uint8_t temperature =0;  
temperature = analogRead(0);  
publish_frame_new(temperature);
```

The temperature was rounded to the nearest unit.

4.3.5. Version differences

Like the Olimexino, the Arduino node also has different functions for the different versions of the protocol. Here, too, they are chosen via preprocessor macros.

Chapter 5

Results and demonstration

This chapter explains how the various parts of the network were tested.

It begins with the test of the communication between processes using the micro object request broker.

Then there are the tests for the communication in the CAN network. First the ability to, correctly, send CAN messages and then the ability to correctly receive them.

Finally there is an explanation of how the ArduPilot integration was tested.

5.1. uORB communication

To test the communication using uORB topics one app and one daemon were used. The daemon used was the one explained earlier, called *teste_1*. In this test the daemon publishes one integer to the topic every ten seconds, incrementing with every publication. As a receiver is an app called *teste_2* that will receive from that same topic and print to the command line. Both the app and the daemon exit when the message reaches ten.

```
nsh> teste_1 start
teste_1: [daemon] starting
teste_1.c---Inicio
nsh> teste_2
teste_2.c---Inicio
recebido=0
recebido=1
recebido=2
recebido=3
recebido=4
recebido=5
recebido=6
recebido=7
```

```

recebido=8
recebido=9
recebido=10
teste_2.c---Fim
nsh> teste_1.c---Fim
teste_1: [daemon] exiting.

```

5.2. Sending CAN messages

To test sending CAN messages, the Pixhawk was connected to the Arduino and to the Olimexino. As stated before the Olimexino toggles a LED when a message is received and the Arduino prints the data to the serial port. On the Pixhawk side, the UAVCAN daemon receives messages from the uORB topic *teste_topic* and sends them to the CAN bus. To publish to *teste_topic* the daemon *teste_1* was used.

```

nsh> uorb start
nsh> uavcan start
PX4: param_find(UAVCAN_NODE_ID)
PX4: param_find(UAVCAN_BITRATE)
uavcan: Node ID 1, bitrate 1000000
uavcan: SW version vcs_commit: 0x3d7ec370
[Fim do init]
teste_controller ret=0
uavcan: sensor bridge 'gnss' init ok
uavcan: sensor bridge 'mag' init ok
uavcan: sensor bridge 'baro' init ok
nsh>teste_1 start
teste_1: [daemon] starting
teste_1.c---Inicio

```

Both devices received the messages, with the Arduino printing to the serial port:

```

-----
get data from ID:
398722057
Variable ID= 760
Source node ID= 1

```

```
Transfer ID= 1
```

```
1
```

The data was correctly received.

A servo motor was then attached to the Arduino. The Arduino received the values from the Pixhawk and sent a command to the servo motor with a value of zero meaning 0 degrees and 255 meaning 180 degrees of deflection.

5.3. Receiving CAN messages

To test receiving CAN messages, the Pixhawk was again connected to the Arduino and to the Olimexino. The Arduino was programmed to send an incrementing integer every three seconds. The Olimexino also sends an incrementing integer but at the push of a button. The UAVCAN daemon receives the messages, prints node ID and publishes them to a uORB topic. A new topic, called *teste_receive*, was created and the *teste_2* app was adapted to receive from that topic and print the message.

```
nsh> uorb start
nsh> uavcan start
PX4: param_find(UAVCAN_NODE_ID)
PX4: param_find(UAVCAN_BITRATE)
uavcan: Node ID 1, bitrate 1000000
uavcan: SW version vcs_commit: 0x3d7ec370
[Fim do init]
teste_controller ret=0
uavcan: sensor bridge 'gnss' init ok
uavcan: sensor bridge 'mag' init ok
uavcan: sensor bridge 'baro' init ok
nsh>teste_2
teste_2.c---Inicio
node=120
recebido=0
```

A temperature sensor was attached to the Arduino that read the temperature and sent the value to the bus. The temperature was sent in degrees Celsius, rounded to the nearest unit.

5.4. ArduPilot integration

To test the ArduPilot integration, one of the loops was used to publish to a uORB topic. That topic was then read from the UAVCAN daemon and sent to the bus. Initially the ArduCopter code was chosen but due to a bug any code added resulted in an error.

```
Assertion failed at file:armv7-m/up_hardfault.c line: 184 task: ArduCopter
sp:      20013f40
IRQ stack:
  base: 20013f8c
  size: 000005d8
20013f40: 080c4528 000000b8 08091029 00000010 200115d8 00000003 00000000
08097711
20013f60: 080976fd 080a48df e000ed24 080a1815 00000000 20006820 20006824
20000e38
20013f80: 2000bb98 08097681 2001cac4 00000000 00000000 00000000 00000000
00000000
sp:      2001cb98
User stack:
  base: 2001cc18
  size: 00001ffc
2001cb80: 3eaaabae 00000000 00000000 00000000 60000010 0805d2b3 00000000
00000000
2001cba0: 20006810 20000a90 0068f37c 08004491 20000a90 08004515 2000bae8
00000000
2001cbc0: 1000c730 08018c49 08018c9d 00000000 00000000 00000000 006a788b
00000000
2001cbe0: 00000000 00000000 08018c9d 00000000 00000000 00000000 00000000
00000000
2001cc00: 00000000 00000000 00000000 08092301 00000000 00000000 ffffffff
0d0e0fd4
R0: 00000000 2000683c 00000001 200112e4 20006820 20006824 20000e38 2000bb98
R8: 1000c160 2000bbf8 00000000 00000000 00000000 2001cb98 0805d2c9 200112e4
xPSR: 40000000 BASEPRI: 00000000 CONTROL: 00000000
EXC_RETURN: fffffffe9
```

The ArduPlane code was tried next. The one second loop was used to publish an integer that incremented with each publication.

```
NuttShell (NSH)
nsh> [PX4io] mixer sent
<fmuservo> MODE_4PWM
```



```
<fmuservo> set_pwm_rate 0 50 50  
disabling flow control on /dev/ttyS1 _total_written=129  
node=120  
msg = 0
```

This time everything worked as expected and the integration with the ArduPilot code was complete.

Chapter 6

Conclusions and Recommendations

In this last chapter there is a summary of what was done and what was accomplished. It ends with a suggestion for further improvements and future work.

6.1. Summary of the results

This thesis work was intended to integrate a popular autopilot called ArduPilot to a new controller area network (CAN) protocol designed specifically for unmanned aerial vehicles (UAVs) called UAVCAN. To do so three open source projects are used. The Pixhawk project is responsible for the creation of the autopilot board, its operating system, called NuttX, and the UAVCAN protocol. The ArduPilot project is responsible for the creation of the autopilot software and its integration on the NuttX operating system. The UAVCAN project is responsible for creating the protocol and a library as an example implementation. The ArduPilot project uses the toolchain from the Pixhawk project. This creates some confusion because the toolchain downloads and installs the last version of the operation system, a version that is different from the one used by the ArduPilot. The ArduPilot code comes as a special app of the NuttX operating system. It uses an older version of NuttX that includes an outdated version of the UAVCAN protocol, a version that is incompatible with its final one. The final version of UAVCAN was released during the making of this work. Despite coming in a bundle, the ArduPilot code does not use CAN bus.

Being open source projects the documentation was lacking and sometimes inexistent, making development difficult and slow.

Development, and later testing, was begun in the operating system alone, without the ArduPilot code, due to its faster compilation time. However, the code came with the final version, before it was released, so the testing nodes needed to be adapted and work duplicated. Due to some overlap in the projects, part of the UAVCAN team also worked in the NuttX operating system, the UAVCAN protocol is deeply integrated in the operating system. Such integration made a possible update of the protocol version in the ArduPilot code a difficult and protracted endeavor. Despite being incompatible at the bit level, in the bus, both versions of the protocol use the same the same functions. This made possible the unchanged use of the code with both versions, keeping the duplicated work to the nodes only.

The integration of the UAVCAN protocol with the ArduPilot code was achieved. Messages were successfully sent from the ArduPilot to the nodes in the bus where they were used to command actuators. And sensor readings were received.

6.2. Future work

This section addresses some unresolved problems and recommendations for future work, in no specific order.

One major problem was the poor or lack of documentation. It was especially problematic with the ArduPilot project. The main problem was with the integration of in the NuttX operating system. Very little was available about how it was done and how it worked. It makes the entry of new developers unnecessarily difficult and long. The Pixhawk had some error in its documentation that caused some problems and can be improved. The UAVCAN had problems inherent to being a new project and still under development

The lack of documentation contributed to another problem that is the different versions used by the different projects. The ArduPilot project installed a firmware version that was sometimes incompatible with the one it used.

The Pixhawk toolchain installed software that was outdated and not the one necessary for development. The Pixhawk toolchain was created to allow for the easy start of developers and to guarantee that they all used the same configurations. However development was being done in a newer version of the ARM compiler. This meant the compiler needed to be updated to the newer version manually. And since there was no warning, developers only new to do so when they were faced with an error they could not fix and had to ask for help.

Other problem was buggy code. The ArduPilot has a bug in the ArduCopter section that makes it impossible to add code.

For the widespread adoption of the UAVCAN protocol more compatible sensors and actuators are needed. The ArduPilot creators are in a privileged position to do so because they have the platform to advertise and sell, the means of production and the audience. Because the protocol is robust and useful others would follow.

I will be continuing this work with UAVision in order to create a product that will simplify the control of commercial UAVs.

Bibliography

- [1] "CAN in automation: CAN history," [Online]. Available: <http://www.can-cia.de/index.php?id=161>. [Accessed July 2015].
- [2] G. Leen and D. Heffernan, "Expanding Automotive Electronic Systems," *Computer*, vol. 35, no. 1, pp. 88-93, January 2002.
- [3] CAN in Automation, "CAN in Automation (Cia) : History of CAN technology," [Online]. Available: <http://www.can-cia.org/can-knowledge/can/can-history/>. [Acedido em October 2015].
- [4] STMicroelectronics, "TN0897 Technical note: ST SPI protocol," 2013.
- [5] K. T. Wong, "Narrowband PPM semi-'blind' spatial-rake receiver," *Wiley InterScience*, vol. 18, no. 2, pp. 193-197, 2006.
- [6] "Interface specification for airborne CAN applications V 1.7," 12 January 2006. [Online]. Available: http://www.stockflightsystems.com/tl_files/downloads/canaerospace/canas_17.pdf. [Accessed July 2015].
- [7] P. Kirienko, "DIYDrones," 23 January 2014. [Online]. Available: <http://diydrones.com/profiles/blogs/uavcan-can-bus-for-uav>. [Accessed July 2015].
- [8] P. Kirienko, "UAVCAN," [Online]. Available: <http://www.uavcan.org>. [Accessed July 2015].
- [9] STMicroelectronics, "Medium-density performance line ARM®-based 32-bit MCU," August 2015. [Online]. Available: <http://www.st.com/web/en/resource/technical/document/datasheet/CD00161566.pdf>. [Accessed September 2015].
- [10] Atmel Corporation, "8-bit Atmel Microcontroller with 4/8/16K Bytes In-System Programmable Flash," May 2011. [Online]. Available: <http://www.atmel.com/Images/doc2545.pdf>. [Accessed July 2015].

- [11] W. Lawrenz, CAN System Engineering - From Theory to Practical Applications, Springer Verlag, 2013.
- [12] "Road vehicles -- Controller area network (CAN) -- Part 1: Data link layer and physical signalling," ISO 11898-1, 2003.
- [13] "ArduPilot," [Online]. Available: <http://www.ardupilot.com>. [Accessed October 2015].
- [14] "Mission Planner," [Online]. Available: <http://planner.ardupilot.com/>. [Accessed October 2015].
- [15] L. Meier, P. Tanskanen, L. Heng, G. H. Lee, F. Fraundorfer and M. Pollefeys, "Pixhawk: A micro aerial vehicle design for autonomous flight using onboard computer vision," *Autonomous Robots (AURO)*, 2012.
- [16] "PIXHAWK," [Online]. Available: <https://pixhawk.ethz.ch/>. [Accessed September 2015].
- [17] "Pixhawk autopilot," [Online]. Available: <https://pixhawk.org/modules/pixhawk>. [Accessed September 2015].
- [18] "NuttX RTOS," [Online]. Available: <https://pixhawk.org/dev/nuttx/start>. [Accessed September 2015].
- [19] "NuttX," [Online]. Available: <http://nuttx.org/>. [Accessed September 2015].
- [20] "NSH Serial Console Wiring," [Online]. Available: <https://pixhawk.org/dev/wiring>. [Accessed September 2015].
- [21] "uORB - PX4 Autopilot Project," [Online]. Available: <https://pixhawk.org/firmware/apps/uorb>. [Accessed September 2015].
- [22] Arduino, Llc, "Arduino - Introduction," [Online]. Available: <https://www.arduino.cc/en/guide/introduction>. [Accessed October 2015].
- [23] Arduino, Llc, "Arduino - ArduinoBoardDiecimila," [Online]. Available: <https://www.arduino.cc/en/Main/ArduinoBoardDiecimila>. [Accessed October 2015].

- [24] Seeed Development Limited, "CAN-BUS Shield V1.2 - Wiki," [Online]. Available: http://www.seeedstudio.com/wiki/CAN-BUS_Shield_V1.2. [Accessed October 2015].
- [25] Microchip Technology Inc, "MCP2515," 2005. [Online]. Available: <http://www.seeedstudio.com/wiki/images/8/83/MCP2515.pdf>. [Accessed October 2015].
- [26] Microchip Technology Inc, "Mcp2551," 2010. [Online]. Available: <http://www.seeedstudio.com/wiki/images/8/8c/Mcp2551en.pdf>. [Accessed October 2015].
- [27] Olimex, Ltd, "OLIMEXINO-STM32 development board User Manual," November 2014. [Online]. Available: <https://www.olimex.com/Products/Duino/STM32/OLIMEXINO-STM32/resources/OLIMEXINO-STM32.pdf>. [Accessed September 2015].
- [28] STMicroelectronics, "ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32," September 2012. [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/DM00027105.pdf. [Accessed September 2015].
- [29] E. Ashfield, I. Field, P. Harrod, S. Houlihane, W. Orme and S. Woodhouse, "Serial Wire Debug and the CoreSight Debug and Trace Architecture," ARM Ltd, Cambridge.
- [30] IAR Systems AB, "IAR Embedded Workbench® IDE - User Guide," 2009. [Online]. Available: http://supp.iar.com/FilesPublic/UPDINFO/004916/arm/doc/EWARM_UserGuide.ENU.pdf. [Accessed September 2015].
- [31] P. Kirienko, "UAVCAN," [Online]. Available: <http://uavcan.org>. [Accessed October 2015].
- [32] P. Kirienko, "UAVCAN Documentation," [Online]. Available: <http://old.uavcan.org/UAVCAN>. [Accessed September 2015].
- [33] "Toolchain Instalation," [Online]. Available: https://pixhawk.org/dev/toolchain_installation_win. [Accessed July 2015].
- [34] "Cygwin," [Online]. Available: <https://www.cygwin.com>. [Accessed July 2015].

- [35] "MinGW," [Online]. Available: www.mingw.org. [Accessed July 2015].
- [36] "TeraTerm," [Online]. Available: <https://tssh2.osdn.jp/index.html.en>. [Accessed July 2015].
- [37] "Hterm - A terminal program for Windows and Linux," [Online]. Available: www.der-hammer.info/terminal. [Accessed July 2015].
- [38] J. des Rivieres and W. Beaton, "Eclipse Platform Technical Overview," 17 July 2001. [Online]. Available: <http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html>. [Accessed July 2015].
- [39] "GCC ARM Embedded in Launchpad," [Online]. Available: <https://launchpad.net/gcc-arm-embedded>. [Accessed July 2015].
- [40] "GCC ARM Embedded 4.7-2012-q4-major," [Online]. Available: <https://launchpad.net/gcc-arm-embedded/4.7/4.7-2012-q4-major>. [Accessed September 2015].
- [41] Olimex Ltd, "Olimexino-stm32," [Online]. Available: <https://www.olimex.com/Products/Duino/STM32/OLIMEXINO-STM32/open-source-hardware>. [Accessed September 2015].
- [42] Seeed Development Limited, "CAN bus shield," [Online]. Available: https://github.com/Seeed-Studio/CAN_BUS_Shield. [Accessed September 2015].

Appendix A

App and Daemon Files

Teste_1.c

```
/*
 * teste_1.c
 *
 * Created on: 13 de Ago de 2015
 * Author: Pedro Silva
 */

#include <nutt/config.h>
#include <stdio.h>
#include <errno.h>
#include <uORB/topics/teste_topic.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <px4_config.h>
#include <nutt/sched.h>
#include <systemlib/systemlib.h>
#include <systemlib/err.h>

static bool thread_should_exit = false;    /**< daemon exit flag */
static bool thread_running = false;        /**< daemon status flag */
static int daemon_task;                    /**< Handle of daemon task / thread */

/**
 * daemon management function.
 */
__EXPORT int teste_1_main(int argc, char *argv[]);

/**
 * Mainloop of daemon.
 */
int teste_1_daemon_app_main(int argc, char *argv[]);

/**
 * Print the correct usage.
 */
static void usage(const char *reason);

static void
usage(const char *reason)
{
    if (reason) {
        warnx("%s\n", reason);
    }

    warnx("usage: daemon {start|stop|status} [-p <additional params>]\n\n");
}
```



```

}

/**
 * The daemon app only briefly exists to start
 * the background job. The stack size assigned in the
 * Makefile does only apply to this management task.
 *
 * The actual stack size should be set in the call
 * to task_create().
 */
int teste_1_main(int argc, char *argv[])
{
    if (argc < 2) {
        usage("missing command");
        return 1;
    }

    if (!strcmp(argv[1], "start")) {

        if (thread_running) {
            warnx("daemon already running\n");
            /* this is not an error */
            return 0;
        }

        thread_should_exit = false;
        daemon_task = px4_task_spawn_cmd("teste_1",
            SCHED_DEFAULT,
            SCHED_PRIORITY_DEFAULT,
            2000,
            teste_1_daemon_app_main,
            (argv) ? (char * const *)&argv[2] : (char * const
*)NULL);
        return 0;
    }

    if (!strcmp(argv[1], "stop")) {
        thread_should_exit = true;
        return 0;
    }

    if (!strcmp(argv[1], "status")) {
        if (thread_running) {
            warnx("\trunning\n");
        } else {
            warnx("\tnot started\n");
        }

        return 0;
    }

    usage("unrecognized command");
    return 1;
}

```

```

int teste_1_daemon_app_main(int argc, char *argv[])
{
    warnx("[daemon] starting\n");

    thread_running = true;
    printf("teste_1.c---Inicio\n");

    struct teste_topic_s test = {.inc = 0};

    orb_advert_t topic_handle = orb_advertise(ORB_ID(teste_topic), &test); // na
    parte do publisher tem que ser orb_advert_t em vez de static int
    sleep(10);

    while (!thread_should_exit) {

        for (int i=0;i<10;i++)
        {
            test.inc++;
            if(orb_publish(ORB_ID(teste_topic),topic_handle,&test)!=OK){
                printf("Erro orb_publish=%d\n",errno);
                thread_running = false;
                return ERROR;
            }
            sleep(10);
            if(thread_should_exit)
            {
                printf("i=%d",i);
                break;
            }
        }
        break;
    }

    printf("teste_1.c---Fim\n");

    warnx("[daemon] exiting.\n");

    thread_running = false;

    return OK;
}

```

Module.mk

```

MODULE_COMMAND          = teste_1
SRCS                    = teste_1.c
MODULE_STACKSIZE        = 1200

```

Teste_2.c

```
/*
 * teste_2.c
 *
 * Created on: 13 de Ago de 2015
 * Author: Pedro Silva
 */

#include <nuttx/config.h>
#include <stdio.h>
#include <errno.h>
#include <uORB/topics/teste_receive.h>

__EXPORT int teste_2_main(int argc, char *argv[]);

int teste_2_main(int argc, char *argv[])
{
    printf("teste_2.c---Inicio\n");

    bool updated;
    struct teste_receive_s test ;

    static int topic_handle;
    topic_handle = orb_subscribe(ORB_ID(teste_receive));
    if(topic_handle==ERROR){
        printf("Error orb_subscribe (ERROR)=%d\n",errno);
        sleep(10);
        topic_handle = orb_subscribe(ORB_ID(teste_receive));
    }
    else if(topic_handle==-1){
        printf("Error orb_subscribe (-1)=%d\n",errno);
        sleep(10);
        topic_handle = orb_subscribe(ORB_ID(teste_receive));
    }

    if(orb_set_interval(topic_handle,1000)==ERROR){
        printf("Error orb_set_interval =%d\n",errno);
        return ERROR;
    }
    if(orb_copy(ORB_ID(teste_receive),topic_handle,&test)==ERROR){
        printf("Error orb_copy =%d\n",errno);
        if(orb_copy(ORB_ID(teste_receive),topic_handle,&test)==ERROR){
            printf("Error orb_copy =%d\n",errno);
            return ERROR;
        }
    }
    printf("recebido=%d\n",test.inc);

    int i=0;
    while(i<10){
        sleep(1);
        //printf("While-i=%d\n",i);
    }
}
```

```

    if(orb_check(topic_handle,&updated)==ERROR){
        printf("Error orb_check =%d\n",errno);
        if(orb_check(topic_handle,&updated)==ERROR){
            printf("Error orb_check =%d\n",errno);
            return ERROR;
        }
    }

    if(updated){
        if(orb_copy(ORB_ID(teste_receive),topic_handle,&test)==ERROR){
            printf("Error orb_copy =%d\n",errno);
            if(orb_copy(ORB_ID(teste_receive),topic_handle,&test)==ERROR){
                printf("Error orb_copy =%d\n",errno);
                return ERROR;
            }
        }
        printf("recebido=%d\n",test.inc);
        i=test.inc;
    }
}

printf("teste_2.c---Fim\n");
return OK;
}

```

Module.mk

```

MODULE_COMMAND      = teste_2

SRCS                 = teste_2.c

```

Appendix B

UAVCAN Files

Teste_actuator.hpp

```
/**
 * @file teste_actuator.hpp
 *
 * @author Pedro Silva
 */

#pragma once

#include <uavcan/uavcan.hpp>
#include <uavcan/equipment/teste/TesteCommand.hpp>
#include <uavcan/equipment/teste/TesteReceive.hpp>
#include <systemlib/perf_counter.h>
#include <uORB/topics/teste_topic.h>
#include <uORB/topics/teste_receive.h>

class UavcanTeste
{
public:
    UavcanTeste(uavcan::INode &node);
    ~UavcanTeste();

    int init();

    void update_outputs(int output);

private:
    /**
     * TesteReceive message reception will be reported via this callback.
     */
    void teste_receive_sub_cb(const
uavcan::ReceivedDataStructure<uavcan::equipment::teste::TesteReceive> &msg);

    typedef uavcan::MethodBinder<UavcanTeste *,
        void (UavcanTeste::*)(const
uavcan::ReceivedDataStructure<uavcan::equipment::teste::TesteReceive>&)>
        StatusCbBinder;

    /*
     * libuavcan related things
     */
    uavcan::MonotonicTime                _prev_cmd_pub;    ///< rate limiting

```

```

    uavcan::INode                                &_node;
    uavcan::Publisher<uavcan::equipment::teste::TesteCommand>
    _uavcan_pub_teste_cmd;
    uavcan::Subscriber<uavcan::equipment::teste::TesteReceive, StatusCbBinder>
    _uavcan_sub_status;

    teste_receive_s _teste_receive = {.inc = 0};
    orb_advert_t     _teste_receive_pub = nullptr;
};

```

Teste_actuator.cpp

```

/*
 * teste_actuator.cpp
 *
 * Created on: 25 de Set de 2015
 * Author: Pedro Silva
 */

#include "teste_actuator.hpp"
#include <systemlib/err.h>

UavcanTeste::UavcanTeste(uavcan::INode &node) :
    _node(node),
    _uavcan_pub_teste_cmd(node),
    _uavcan_sub_status(node),
{
}

UavcanTeste::~UavcanTeste()
{
}

int UavcanTeste::init()
{
    int res = _uavcan_sub_status.start(StatusCbBinder(this,
    &UavcanTeste::teste_receive_sub_cb));

    if (res < 0) {
        warnx("uavcan sub failed %i", res);
        return res;
    }
    return res;
}

void UavcanTeste::update_outputs(int output)
{
    const auto timestamp = _node.getMonotonicTime();

    if ((timestamp - _prev_cmd_pub).toUsec() < (1000000 / MAX_RATE_HZ)) {
        return;
    }

    _prev_cmd_pub = timestamp;
}

```

```

    uavcan::equipment::teste::TesteCommand msg;

    msg.valor = output;
    printf("msg = %x\n",msg.valor);

    (void)_uavcan_pub_teste_cmd.broadcast(msg);
}

void UavcanTeste::teste_receive_sub_cb(const
uavcan::ReceivedDataStructure<uavcan::equipment::teste::TesteReceive> &msg)
{
    _teste_receive.inc=msg.recebe;

    uint8_t node= msg.getSrcNodeID().get();
    printf("node=%d\n",node);
    if (_teste_receive_pub != nullptr) {
        (void)orb_publish(ORB_ID(teste_receive), _teste_receive_pub,
&_teste_receive);

        } else {
            _teste_receive_pub = orb_advertise(ORB_ID(teste_receive),
&_teste_receive);
        }
}

```

teste/760.TesteCommand.uavcan

uint8 valor

teste/761.TesteReceive.uavcan

uint8 recebe

uavcan_main.hpp

Line 49 : #include "actuators/teste_actuator.hpp"

Line 141

```

UavcanTeste _teste_controller;
int _teste_topic_sub = -1;
teste_topic_s _teste_topic ;

```

Line 145

uavcan_main.cpp

Line 49

```
#include <uORB/topics/esc_status.h>
#include <uORB/topics/teste_topic.h>
```

Line 52

Line 76: `_teste_controller(_node)`

Line 268

```
ret = _teste_controller.init();
printf("teste_controller ret=%d \n",ret);
if (ret < 0) {
    return ret;
}
```

Line 274

Line 330

```
_actuator_direct_sub = orb_subscribe(ORB_ID(actuator_direct));
_teste_topic_sub = orb_subscribe(ORB_ID(teste_topic));
if(_teste_topic_sub==--1){
    printf("Error orb_subscribe (ERROR)=%d\n",errno);
    sleep(10);
    _teste_topic_sub = orb_subscribe(ORB_ID(teste_topic));
}
if(_teste_topic_sub==--1){
    printf("Error orb_subscribe (-1)=%d\n",errno);
    sleep(10);
    _teste_topic_sub = orb_subscribe(ORB_ID(teste_topic));
}
```

Line 345

Line 508

```
updated = false;
if(orb_check(_teste_topic_sub, &updated)==-1){
    printf("Error orb_check =%d\n",errno);
}
if (updated) {
    if(orb_copy(ORB_ID(teste_topic),_teste_topic_sub,&_teste_topic)==-1){
        printf("Error orb_copy =%d\n",errno);
    }
}
if(orb_copy(ORB_ID(teste_topic),_teste_topic_sub,&_teste_topic)==-1){
    printf("Error orb_copy =%d\n",errno);
    return ERROR;
}
else{
    _teste_controller.update_outputs(_teste_topic.inc);
}
}
else {
    _teste_controller.update_outputs(_teste_topic.inc);
}
```


}

Line 533

Appendix C

Message Files

Teste_receive.msg

uint16 inc

Teste_topic.msg

uint16 inc

Objects_common.cpp

Line 259

```
#include "topics/teste_topic.h"  
ORB_DEFINE(teste_topic, struct teste_topic_s);  
  
#include "topics/teste_receive.h"  
ORB_DEFINE(teste_receive, struct teste_receive_s);
```

Line 261

Appendix D

Arduino File

uavcan_px4.pde

```
#include <SPI.h>
#include <Servo.h>
#include "mcp_can.h"

#define __UAVCAN_ANTIGO__

#ifndef __UAVCAN_ANTIGO__
#define __UAVCAN_NOVO__
#endif

const int SPI_CS_PIN = 9;
unsigned long status_ = 288621576;
unsigned long olimex_Id = 0x17CDE008;
unsigned long px4 = 398722056;
unsigned long send_Id = 398722056;
unsigned char ext = 1;
unsigned char len = 0;
unsigned char buf[8];
const int LED = 13;
boolean ledON = 1;
int uavcan_node_id = 120;//111;
MCP_CAN CAN(SPI_CS_PIN);

Servo myservo;
int val

#ifdef __UAVCAN_NOVO__
static const uint8_t PRIORITY_HIGHEST = 0;
static const uint8_t PRIORITY_HIGH = 8;
static const uint8_t PRIORITY_MEDIUM = 16;
```

```

static const uint8_t PRIORITY_LOW      = 24;
static const uint8_t PRIORITY_LOWEST  = 31;
#endif

void setup()
{
  Serial.begin(115200);
  pinMode(LED, OUTPUT);
  myservo.attach(9);
START_INIT:

  if (CAN_OK == CAN.begin(CAN_1000KBPS))
  {
    Serial.println("CAN BUS Shield init ok!");
    Serial.println(CAN.init_Mask(1, 1, 0xFFFFFFFF));
    Serial.println(CAN.init_Mask(0, 1, 0x1FFFFFFC08));
    Serial.println(CAN.init_Filt(0, 1, px4));
    Serial.println(CAN.init_Filt(1, 1, status_));
  }
  else
  {
    Serial.println("CAN BUS Shield init fail");
    Serial.println("Init CAN BUS Shield again");
    delay(100);
    goto START_INIT;
  }
}

void loop()
{ static boolean publish = true;
  static boolean publish1 = true;

  if (publish & publish1){
    uint8_t temperature = 0;
    temperature = analogRead(0);
#ifdef __UAVCAN_NOVO__
    publish_frame_new();
    publish_frame_new(temperature);
#endif
}
}

```

```

#endif
#ifdef __UAVCAN_ANTIGO__
    publish_frame_old();
    publish_frame_old(temperature);

#endif
    publish = !publish;
}
//delay(500);
if (CAN_MSGAVAIL == CAN.checkReceive())
{
    CAN.readMsgBuf(&len, buf);

    unsigned long canId = CAN.getCanId();

    if (canId >> 19 == 760) {
        Serial.println("-----");
        Serial.println("get data from ID: ");
        Serial.println(canId);
        Serial.print("Variable ID= ");
        Serial.println(canId >> 19);
        Serial.print("Source node ID= ");
        Serial.println((canId >> 10) & 0x7F);
        Serial.print("Transfer ID= ");
        Serial.println(canId & 0x7);
        //Serial.println(millis()>>9);

        for (int i = 0; i < len; i++)
        {
            Serial.print(buf[i]);
            Serial.print("\t");
            val = map(val, 0, 255, 0, 180);
            myservo.write(val);
            if (ledON && i == 0)
            {

                digitalWrite(LED, buf[i]);
            }
        }
    }
}

```

```

        ledON = 0;
        delay(500);
    }
    else if (!(ledON) && i == 4) {

        digitalWrite(LED, buf[i]);
        ledON = 1;
    }

}
Serial.println();
}
else if (canId >> 19 == 550){
    Serial.println("Status received");
    publish_status();
    publish= true;
    publish1 = !publish1;
}
}
}

void LED_Toggle(){
    static boolean led=false;
    digitalWrite(LED, led);
    led = !led;
}

void publish_status() {

    uint8_t payload[7];
    uint8_t health = 0;
    // Uptime in seconds
    static uint32_t uptime_sec2 = millis()>>9;
    payload[0] = (uptime_sec2 >> 0) & 0xFF;
    payload[1] = (uptime_sec2 >> 8) & 0xFF;
    payload[2] = (uptime_sec2 >> 16) & 0xFF;
    payload[3] = (uptime_sec2 >> 24) & 0xFF;

```

```

// Health and mode
payload[4] = (health << 6) | ((uint8_t)0 << 3);

payload[5] = 0x00;
payload[6] = 0x00;

static uint8_t transfer_id;
transfer_id += 1;
#ifdef __UAVCAN_NOVO__
static const uint16_t data_type_id = 341;
uavcan_broadcast_new(PRIORITY_LOW, data_type_id, transfer_id, payload,
sizeof(payload));
#endif
#ifdef __UAVCAN_ANTIGO__
static const uint16_t data_type_id = 550;
uavcan_broadcast_old( data_type_id, transfer_id, payload, sizeof(payload));
#endif

}

#ifdef __UAVCAN_ANTIGO__
int publish_frame_old(void){
//Serial.println("Publish frame");

static uint8_t carga=0;

uint8_t payload[1];
payload[0] = carga;
carga +=1;
static const uint16_t data_type_id = 761;
static uint8_t transfer_id;

int ret = uavcan_broadcast_old( data_type_id, transfer_id, payload, 1);
transfer_id += 1;
return ret;
}

```

```

int publish_frame_old(uint8_t carga){

    uint8_t payload[1];
    payload[0] = carga;
    static const uint16_t data_type_id = 761;
    static uint8_t transfer_id;

    int ret = uavcan_broadcast_old( data_type_id, transfer_id, payload, 1);
    transfer_id += 1;
    return ret;

}

int uavcan_broadcast_old(uint16_t data_type_id,uint8_t transfer_id,uint8_t*
payload,uint16_t payload_len){

    if (payload == NULL || payload_len > 8)
    {
        return -1; // In this super-simple implementation we don't support multi-
frame transfers.
    }

    uint32_t can_id = 0x00000000;
    can_id = ((uint32_t)data_type_id << 19)|0x00040000|( 0x0001FC00
&((uint32_t)uavcan_node_id << 10)) |(((0x00F)& (8 | ((uint32_t)transfer_id &
0x00000007))));

    uint8_t payload_with_tail_byte[8]={0x00};

    memcpy(payload_with_tail_byte, payload, payload_len);

    return CAN.sendMsgBuf(can_id, 1, payload_len, payload_with_tail_byte);
}
#endif

#ifdef __UAVCAN_NOVO__

```



```

int publish_frame_new(void)
{

    static uint8_t carga=0;

    uint8_t payload[1];
    payload[0] = carga;

    carga +=1;
    static const uint16_t data_type_id = 1112;
    static uint8_t transfer_id;
    transfer_id += 1;

    int ret =uavcan_broadcast_new(PRIORITY_MEDIUM, data_type_id, transfer_id,
payload, 1);
    transfer_id += 1;
    return ret;
}

```

```

int publish_frame_new(uint8_t carga)
{
    uint8_t payload[1];
    payload[0] = carga;
    static const uint16_t data_type_id = 1112;
    static uint8_t transfer_id;
    transfer_id += 1;

    int ret =uavcan_broadcast_new(PRIORITY_MEDIUM, data_type_id, transfer_id,
payload, 1);
    transfer_id += 1;
    return ret;
}

```

```

int uavcan_broadcast_new(uint8_t priority, uint16_t data_type_id, uint8_t
transfer_id, uint8_t* payload, uint16_t payload_len)
{
    if (payload == NULL || payload_len > 7)
    {

```

```

        return -1; // In this super-simple implementation we don't support multi-
frame transfers.
    }

    if (priority > 31)
    {
        return -1;
    }
    uint32_t can_id = 0x00000000;
    can_id = ((uint32_t)priority << 24) | ((uint32_t)data_type_id << 8) |
(uint32_t)uavcan_node_id;

    uint8_t payload_with_tail_byte[8]={0x00};

    memcpy(payload_with_tail_byte, payload, payload_len);

    payload_with_tail_byte[payload_len] = 0xC0 | (transfer_id & 31);

    return CAN.sendMessage(can_id, 1, payload_len+1, payload_with_tail_byte);
}
#endif

```

Appendix E

Olimexino Files

Main.c

```
#include "includes.h"

Int32U CriticalSecCntr;

#define __UAVCAN_ANTIGO__

#ifndef __UAVCAN_ANTIGO__
#define __UAVCAN_NOVO__
#endif

#define EXTENDED_ID 1028

#define __CAN1_USED__
// #define __CAN2_USED__

#ifdef __CAN1_USED__
#define CANx CAN1
#define GPIO_CAN GPIO_CAN1
#define GPIO_Remapping_CAN GPIO_Remapping_CAN1
#define GPIO_CAN GPIO_CAN1
#define GPIO_Pin_CAN_RX GPIO_Pin_CAN1_RX
#define GPIO_Pin_CAN_TX GPIO_Pin_CAN1_TX
#else /* __CAN2_USED__ */
#error "No CAN2 on the OLIMEXINO STM32!"
#endif /* __CAN1_USED__ */

void NVIC_Config(void);
void CAN_Config(void);
void LED_Display(uint8_t Ledstatus);
void Init_RxMes(CanRxMsg *RxMessage);
void Delay(void);
void Timer_Init(void);
void publish_status(uint8_t health, uint8_t mode);
#ifdef __UAVCAN_NOVO__
int uavcan_broadcast_new(uint8_t priority,uint16_t data_type_id,uint8_t
transfer_id,uint8_t* payload,uint16_t payload_len);
int publish_frame_new(void);
#endif
#ifdef __UAVCAN_ANTIGO__
int uavcan_broadcast_old(uint16_t data_type_id,uint8_t transfer_id,uint8_t*
payload,uint16_t payload_len);
int publish_frame_old(void);
#endif

CAN_InitTypeDef CAN_InitStructure;
```

```

CAN_FilterInitTypeDef CAN_FilterInitStructure;
CanTxMsg TxMessage;
uint8_t KeyNumber = 0x0;
uint32_t uptime_sec = 0x00000000;
uint8_t TransferId =0x0;

#ifdef __UAVCAN_NOVO__
static const uint8_t PRIORITY_HIGHEST = 0;
static const uint8_t PRIORITY_HIGH = 8;
static const uint8_t PRIORITY_MEDIUM = 16;
static const uint8_t PRIORITY_LOW = 24;
static const uint8_t PRIORITY_LOWEST = 31;
#endif
static uint8_t uavcan_node_id;
uint8_t health = 0;
uint32_t entradas[200];
int i=0;

void Clk_Init(void);

int main(void)
{
    ENTR_CRT_SECTION();
    NVIC_Config();
    EXT_CRT_SECTION();
    LED_Initialize();
    BUTON_Initialize();
    Timer_Init();

    for(i=0;i<200;i++){
        entradas [i]=0;
    }
    i=0;

    CAN_Config();

    CAN_ITConfig(CANx, CAN_IT_FMP0, ENABLE);

    LED_Off(LED1);
    LED_Off(LED2);

    uavcan_node_id = 120;

    while(1)
    {
        Delay();
        if(GetButton()){
            while(GetButton()){}
#ifdef __UAVCAN_NOVO__
            publish_frame_new();
#endif
#ifdef __UAVCAN_ANTIGO__
            publish_frame_old();
#endif
        }
    }
}

```

```

    }
}

void Clk_Init (void)
{
    RCC_HSICmd(ENABLE);
    while(RCC_GetFlagStatus(RCC_FLAG_HSIIRDY) == RESET);
    RCC_SYSCLKConfig(RCC_SYSCLKSource_HSI);
    RCC_HSEConfig(RCC_HSE_ON);
    while(RCC_GetFlagStatus(RCC_FLAG_HSERDY) == RESET);
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1,RCC_PLLMul_9); // 72MHz
    RCC_PLLCmd(ENABLE);
    while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET);
    RCC_USBCLKConfig(RCC_USBCLKSource_PLLCLK_1Div5);
    RCC_ADCCLKConfig(RCC_PCLK2_Div8);
    RCC_PCLK2Config(RCC_HCLK_Div1);
    RCC_PCLK1Config(RCC_HCLK_Div2);
    RCC_HCLKConfig(RCC_SYSCLK_Div1);
#ifdef EMB_FLASH
    FLASH_SetLatency(FLASH_Latency_2);
    FLASH_HalfCycleAccessCmd(FLASH_HalfCycleAccess_Disable);
    FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable);
#endif // EMB_FLASH
    RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
}

void Timer_Init(void){

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
    TIM_TimeBaseInitTypeDef timerInitStructure;
    timerInitStructure.TIM_Prescaler = (1099-1);
    timerInitStructure.TIM_CounterMode = TIM_CounterMode_Up;
    timerInitStructure.TIM_Period = (65514-1);
    timerInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
    timerInitStructure.TIM_RepetitionCounter = 0;
    TIM_TimeBaseInit(TIM2, &timerInitStructure);
    TIM_Cmd(TIM2, ENABLE);
    TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);

    NVIC_InitTypeDef nvicStructure;
    nvicStructure.NVIC_IRQChannel = TIM2_IRQn;
    nvicStructure.NVIC_IRQChannelPreemptionPriority = 0;
    nvicStructure.NVIC_IRQChannelSubPriority = 1;
    nvicStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&nvicStructure);
}

void CAN_Config(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
#ifdef __CAN1_USED__
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO_CAN1, ENABLE);
#else /*__CAN2_USED__*/
    #error "No CAN2 on the OLIMEXINO STM32!"
#endif /* __CAN1_USED__ */
}

```

```

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_CAN_RX;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
GPIO_Init(GPIO_CAN, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_CAN_TX;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIO_CAN, &GPIO_InitStructure);
GPIO_PinRemapConfig(GPIO_Remapming_CAN , ENABLE);

#ifdef __CAN1_USED__
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
#else /*__CAN2_USED__*/
    #error "No CAN2 on the OLIMEXINO STM32!"
#endif /* __CAN1_USED__ */

CAN_DeInit(CANx);
CAN_StructInit(&CAN_InitStructure);
CAN_InitStructure.CAN_TTCM = DISABLE;
CAN_InitStructure.CAN_ABOM = DISABLE;
CAN_InitStructure.CAN_AWUM = DISABLE;
CAN_InitStructure.CAN_NART = DISABLE;
CAN_InitStructure.CAN_RFLM = DISABLE;
CAN_InitStructure.CAN_TXFP = DISABLE;
CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;

CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;
CAN_InitStructure.CAN_BS1 = CAN_BS1_3tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_5tq;
CAN_InitStructure.CAN_Prescaler = 4;
CAN_Init(CANx, &CAN_InitStructure);

#ifdef __CAN1_USED__
    CAN_FilterInitStructure.CAN_FilterNumber = 0;
#else /*__CAN2_USED__*/
    #error "No CAN2 on the OLIMEXINO STM32!"
#endif /* __CAN1_USED__ */

#ifdef __UAVCAN_ANTIGO__
    uint32_t IDi=398722056<<3; // a ID começa no primeiro bit, os utlimos 3 estao reservados
#endif
#ifdef __UAVCAN_NOVO__
    uint32_t IDi=268719873<<3; // a ID começa no primeiro bit, os utlimos 3 estao reservados
#endif

    CAN_FilterInitStructure.CAN_FilterMode = CAN_FilterMode_IdMask;
    CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_32bit;
    CAN_FilterInitStructure.CAN_FilterIdHigh = IDi>>16;
    CAN_FilterInitStructure.CAN_FilterIdLow = (IDi & 0x0000FFFF);
#ifdef __UAVCAN_ANTIGO__
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh = 0xFFFF;
    CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0xE000;//os ultimos 3 bits estao reservados (convem meter a 0)
#endif
#ifdef __UAVCAN_NOVO__

```

```

    CAN_FilterInitStructure.CAN_FilterMaskIdHigh = 0x07FF;
    CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0xFFF8;//os ultimos 3 bits estao
    reservados (convem meter a 0)
#endif

```

```

    CAN_FilterInitStructure.CAN_FilterFIFOAssignment = 0;
    CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;
    CAN_FilterInit(&CAN_FilterInitStructure);
    TxMessage.StdId = 0x000;
    TxMessage.RTR = CAN_RTR_DATA;
    TxMessage.IDE = CAN_Id_Extended;
}

```

```

int can_send(uint32_t extended_can_id, const uint8_t* frame_data, uint8_t
frame_data_len)

```

```

{
    if (frame_data_len > 8 || frame_data == NULL)
    {
        return -1;
    }

```

```

    TxMessage.ExtId = extended_can_id;
    TxMessage.DLC = frame_data_len;
    TxMessage.Data[0]=frame_data[0];
    TxMessage.Data[1]=frame_data[1];
    TxMessage.Data[2]=frame_data[2];
    TxMessage.Data[3]=frame_data[3];
    TxMessage.Data[4]=frame_data[4];
    TxMessage.Data[5]=frame_data[5];
    TxMessage.Data[6]=frame_data[6];
    TxMessage.Data[7]=frame_data[7];

```

```

    return CAN_Transmit(CANx, &TxMessage);
}

```

```

void NVIC_Config(void)

```

```

{
#ifdef EMB_FLASH
    NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
#else // VECT_TAB_FLASH
    NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
#endif
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
#ifdef __CAN1_USED__
    NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
#else /*__CAN2_USED__*/
    #error "No CAN2 on the OLIMEXINO STM32!"
#endif /*__CAN1_USED__*/
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

```

```

void Init_RxMes(CanRxMsg *RxMessage)
{
    uint8_t i = 0;
    RxMessage->StdId = 0x00;
    RxMessage->ExtId = 0x00;
    RxMessage->IDE = CAN_ID_STD;
    RxMessage->DLC = 0;
    RxMessage->FMI = 0;
    for (i = 0; i < 8; i++)
    {
        RxMessage->Data[i] = 0x00;
    }
}

void Delay(void)
{
    uint16_t nTime = 0x0000;

    for(nTime = 0; nTime <0xFFF; nTime++)
    {
    }
}

/// Standard data type: uavcan.protocol.NodeStatus
void publish_status(uint8_t health, uint8_t mode)
{
    uint8_t payload[7];

    const uint32_t uptime_sec2=uptime_sec;
    payload[0] = (uptime_sec2 >> 0) & 0xFF;
    payload[1] = (uptime_sec2 >> 8) & 0xFF;
    payload[2] = (uptime_sec2 >> 16) & 0xFF;
    payload[3] = (uptime_sec2 >> 24) & 0xFF;
    payload[4] = (health << 6) | (mode << 3);
    payload[5] = 0x00;
    payload[6] = 0x00;

    static uint8_t transfer_id;
    transfer_id +=1;
#ifdef __UAVCAN_NOVO__
    static const uint16_t data_type_id = 341;
    uavcan_broadcast_new(PRIORITY_LOW, data_type_id, transfer_id, payload,
sizeof(payload));
#endif
#ifdef __UAVCAN_ANTIGO__
    static const uint16_t data_type_id = 550;
    uavcan_broadcast_old( data_type_id, transfer_id, payload, sizeof(payload));
#endif
}

#ifdef __UAVCAN_ANTIGO__
int publish_frame_old(void){

    static uint8_t carga=0;

```



```

uint8_t payload[1];
payload[0] = carga;
/*payload[1] = last;
payload[2] = 0x00;
payload[3] = 0x00;
payload[4] = 0x00;
payload[5] = 0x00;
payload[6] = 0x00;*/

carga +=1;
static const uint16_t data_type_id = 761;
static uint8_t transfer_id;

LED_Toggle(LED1);
int ret = uavcan_broadcast_old( data_type_id, transfer_id, payload, 1);
transfer_id += 1;
return ret;
}
#endif

#ifdef __UAVCAN_NOVO__
int publish_frame_new(void)
{
    static uint8_t carga=0;

    uint8_t payload[1];
    payload[0] = carga;

    carga +=1;
    static const uint16_t data_type_id = 1112;//1029;
    static uint8_t transfer_id;
    transfer_id += 1;
    LED_Toggle(LED1);

    int ret =uavcan_broadcast_new(PRIORITY_MEDIUM, data_type_id, transfer_id,
    payload, 1);
    transfer_id += 1;
    return ret;
}
#endif

#ifdef __UAVCAN_NOVO__
int uavcan_broadcast_new(uint8_t priority,
                        uint16_t data_type_id,
                        uint8_t transfer_id,
                        uint8_t* payload,
                        uint16_t payload_len)
{
    if (payload == NULL || payload_len > 7)
    {
        return -1; // In this super-simple implementation we don't support multi-
frame transfers.
    }

    if (priority > 31)

```

```

    {
        return -1;
    }
    uint32_t can_id = 0x00000000;
    can_id = ((uint32_t)priority << 24) | ((uint32_t)data_type_id << 8) |
(uint32_t)uavcan_node_id;

    uint8_t payload_with_tail_byte[8]={0x00};

    memcpy(payload_with_tail_byte, payload, payload_len);

    payload_with_tail_byte[payload_len] = 0xC0 | (transfer_id & 31);

    return can_send(can_id, payload_with_tail_byte, payload_len + 1);
}
#endif

#ifdef __UAVCAN_ANTIGO__
int uavcan_broadcast_old(uint16_t data_type_id,uint8_t transfer_id,uint8_t*
payload,uint16_t payload_len){

    if (payload == NULL || payload_len > 8)
    {
        return -1; // In this super-simple implementation we don't support multi-
frame transfers.
    }

    uint32_t can_id = 0x00000000;
    can_id = ((uint32_t)data_type_id << 19)|0x00040000|( 0x0001FC00
&((uint32_t)uavcan_node_id << 10)) |(((0x00F)& (8 | ((uint32_t)transfer_id &
0x00000007))));

    uint8_t payload_with_tail_byte[8]={0x00};

    memcpy(payload_with_tail_byte, payload, payload_len);

    return can_send(can_id, payload_with_tail_byte, payload_len );

}
#endif

```

stm32f10x_it.c

Line 35

```

CanRxMsg RxMessage;
extern uint8_t KeyNumber;
extern void LED_Display(uint8_t Ledstatus);
extern uint32_t uptime_sec;
extern uint32_t entradas[200];
extern int i;
extern void publish_status(uint8_t health, uint8_t mode);

```

Line 42

Line 171

```
void USB_LP_CAN1_RX0_IRQHandler(void)
{
    LED_Toggle(LED2);
    CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
    if(i<200){
        entradas[i]=RxMessage.ExtId;
        i++;}
}
```

Line 187

Line 200

```
extern uint8_t health;
void TIM2_IRQHandler(void)
{
    if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
    {
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);

        publish_status(health, (uint8_t)0);
        uptime_sec +=1;
    }
}
```

Line 213