Development of technology and procedures for health monitoring of UAV subsystems

Pedro Bernardo Andrade da Silva
pedro.a.silva@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal
November 2015

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.

1. Introduction

The Unmanned Aerial Vehicles (UAVs) industry is growing. 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 were designed for communication with few elements and in close proximity, meaning they don't scale well. While protocols in the RC hobby industry 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 Controller Area Network (CAN) bus is ideal to replace the wiring harnesses with a simple two wire bus.

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. However, being developed for full scale aircraft, CANaerospace is not adequate for small UAVs [1] because it has more than 50% of payload overhead, it does not provide an easy way to pass multiple values at once and it 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) [2].

This thesis proposes to create a working example of a CAN bus communication with an autopilot. Per requested by UAVision the autopilot will be the ArduPilot and will be run in a Pixhawk board. The end goal is to have network of three nodes communicating with the UAVCAN protocol. And having the Pixhawk send messages to the Arduino, to control a servo motor, and receive temperature readings.

1. 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 introduction to the software and hardware used in this work.

And finally there is an explanation of the protocol that will be used for communication.

1.1. The CAN standard

The CAN bus was originally developed for the automotive industry, to replace complex wiring harnesses with a two wire bus [3]. 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) [4]. 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.

1.1.1. 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 provides for 2^{11}, or 2048 different message identifiers, whereas the extended 29-bit identifier provides for 2^{29}, or 537 million identifiers. They are compatible with each other and can function in the same bus.

1.1.2. A CAN message

A fundamental CAN characteristic, shown in Figure 1, 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.

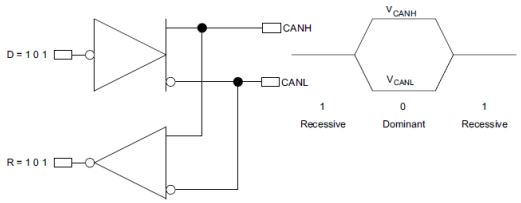

Figure 1 - The inverted logic of a CAN bus

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 message priority is up to the system designer.

1.1.3. 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.

1.1.4. 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.

1.1.5. The CAN bus

Signaling is differential which is where CAN derives its robust noise immunity and fault tolerance. The High-Speed ISO 11898 Standard [5] 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 (Zo).

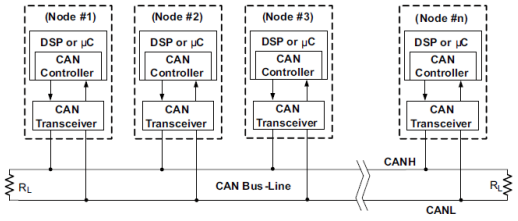

Figure 2 - CAN bus network configuration

As can be seen in Figure 2, the bus must be terminated with two resistances, R_L, of equal value to the cable, 120Ω.

1.2. ArduPilot

The ArduPilot (or ArduPilot Mega) firmware is an open-source Unmanned Aerial Vehicle (UAV) autopilot. It works with the Pixhawk/PX4 hardware, 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. The MissionPlanner serves as a ground station to the vehicle, allowing its control or mission monitorization and configuration.

The operating system that comes with the ArduPilot code is an old version. Because the UAVCAN library comes with the operating system, it is an outdated version as well. The nodes of the CAN network were made to work with both versions. When the different versions require different action it will be stated.

1.3. 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 [6]. The project was eventually open-sourced and development was opened to the community with the PX4 autopilot project.

1.3.1. PX4 Hardware

The latest version of the board, presented in Figure 3 and the one used in this work, uses a 32bit ARM

processor, based on the Cortex M4F architecture running at 168 MHz.

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

Figure 3 - The Pixhawk autopilot board

1.3.2. PX4 Firmware

The PX4 hardware runs a small, lightweight and efficient operating system NuttX, 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 [7]. It has an emphasis on a small footprint and standards compliance. 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 converter had to be used.

1.3.3. 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 [8].

1.4. Arduino

Arduino is an open-source hardware and software company, project and user community that designs and manufactures microcontroller-based prototyping boards.

The board used in this work, the diecimila, 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* [9] produced by Seeed Studio. It uses a MCP2515 CAN Bus controller [10] with SPI interface and MCP2551 CAN transceiver [11].

Both the Arduino board and the shield can be seen in Figure 4.

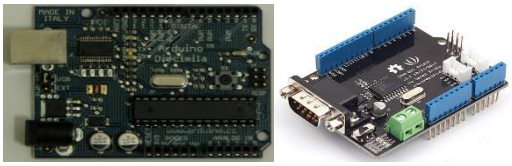

Figure 4 - Arduino diecimila (left). CAN-bus shield (right)

1.5. Olimexino

The Olimexino-stm32 is an open source prototyping board with a stm32 microcontroller, based on the ARM cortex-M3 architecture [12]. 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. A ST-Link V2 is used for programming, via the serial wire debug (SWD) [13] port. Given the availability of a functional CAN bus example written for the IAR Workbench IDE, that was the chosen programming environment.

Both the Olimexino board and the ST-Link V2 programmer can be seen in Figure 5.

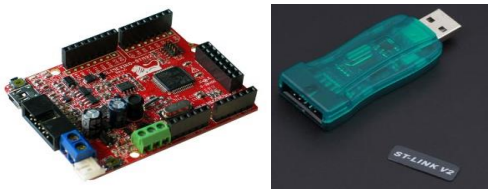

Figure 5 - Olimexino board (left). ST-Link v2 (right)

1.6. 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. It is implemented at the microcontroller level. Here only the parts relevant to this work will be mentioned. The full specification can be found in [14]. There are two versions of the UAVCAN protocol, a new [14] and an old [15] one. When the versions differ, it will be stated.

1.6.1. 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.

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.

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.

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 and 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 request and response contain exactly the same values for all fields except payload, whose content is application defined.

1.6.2. 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.

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.

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.

Field name and constant name

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

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:

- bool
- intX – With X between 2 and 64.
- uintX – With X between 2 and 64.
- floatX – With X being 16, 32 or 64

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.

1.6.3. 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.

1.6.4. 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 the older version is the one used in the tests, that is the one presented here.

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:

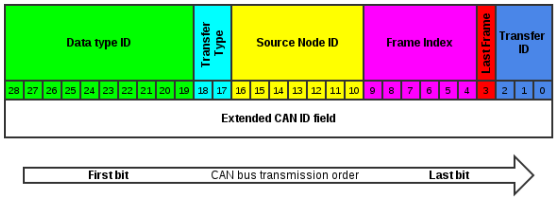

Figure 6 - Old version identifier field

Payload

The payload of a single frame transfer is composed solely of the message to be transmitted.

1.6.5. Payload bit and byte order

Bits a 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.

1.6.6. 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. 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.

2.1. PX4 / ArduPilot

2.1.1. 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.

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

2.1.2. 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 new function that will be run in the thread. 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
```

2.1.3. 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 the can be copied by calling the function *orb_copy*:

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

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.

2.1.4. 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.

2.1.5. Creating 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*.

2.1.6. 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::Te
steCommand>   _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.valor = output;
  (void)_uavcan_pub_teste_cmd.broadcast(msg);}
```

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::equipmen
t::teste::TesteReceive> &msg);
typedef uavcan::MethodBinder<UavcanTeste *,void
(UavcanTeste::*)(const
uavcan::ReceivedDataStructure<uavcan::equipmen
t::teste::TesteReceive>&)>StatusCbBinder;
uavcan::Subscriber<uavcan::equipment::teste::T
esteReceive,                    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));
    return 0;}
```

The function *init()* is run once in *UavcanNode::init()*, in the UAVCAN main thread.

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::equipmen
t::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);
```

2.1.7. 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.

```
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:

```
void Plane::one_hz_loop()
```

2.2. Olimexino

2.2.1. 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 [16], that code was adapted.

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.

2.2.2. 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;
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.

2.2.3. 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.

2.2.4. 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.

2.2.5. 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.

2.3. Arduino

2.3.1. 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 [17], 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.

2.3.2. 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, 0x1FFFFC08);
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.

2.3.3. 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.

2.3.4. 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.

2.3.5. 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.

3. 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.

3.1. 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=10
teste_2.c---Fim
nsh> teste_1.c---Fim
teste_1: [daemon] exiting.
```

3.2. Sending CAN messages

To test the sending of 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.

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
Data = 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.

3.3. Receiving CAN messages

To test the receiving of 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>teste_2
teste_2.c---Inicio
node=120
recebido=0
```

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

3.4. ArduPilot integration

To test the ArduPilot integration, one of the loops was used to publish to a uORB topic. That topic was them read from the UAVCAN daemon and sent to the bus. The ArduPlane code was chosen. The one second loop was used to publish an integer that incremented with each publication.

```
NuttShell (NSH)
node=120
msg = 0
```

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

4. 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. 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.

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

5. Future work

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.

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 all the unsung heroes that have helped me throughout the years. This wouldn't have been possible without you.

References

[1] P. Kirienko, "DIYDrones," 23 January 2014. [Online]. Available: http://diydrones.com/profiles/blogs/uavcan-can-bus-for-uav. [Accessed July 2015].

[2] P. Kirienko, "UAVCAN," [Online]. Available: http://www.uavcan.org. [Accessed July 2015].

[3] "CAN in automation: CAN history," [Online]. Available: http://www.can-cia.de/index.php?id=161. [Accessed July 2015].

[4] W. Lawrenz, CAN System Engineering - From Theory to Practical Applications, Springer Verlag, 2013.

[5] "Road vehicles -- Controller area network (CAN) -- Part 1: Data link layer and physical signalling," ISO 11898-1, 2003.

[6] 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.

[7] "NuttX," [Online]. Available: http://nuttx.org/. [Accessed September 2015].

[8] "uORB - PX4 Autopilot Project," [Online]. Available: https://pixhawk.org/firmware/apps/uorb. [Accessed September 2015].

[9] Seeed Development Limited, "CAN-BUS Shield V1.2 - Wiki," [Online]. Available: http://www.seeedstudio.com/wiki/CAN-BUS_Shield_V1.2. [Accessed October 2015].

[10] Microchip Technology Inc, "MCP2515," 2005. [Online]. Available: http://www.seeedstudio.com/wiki/images/8/83/MCP2515.pdf. [Accessed October 2015].

[11] Microchip Technology Inc, "Mcp2551," 2010. [Online]. Available: http://www.seeedstudio.com/wiki/images/8/8c/Mcp2551en.pdf. [Accessed October 2015].

[12] 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].

[13] 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.

[14] P. Kirienko, "UAVCAN," [Online]. Available: http://uavcan.org. [Accessed October 2015].

[15] P. Kirienko, "UAVCAN Documentation," [Online]. Available: http://old.uavcan.org/UAVCAN. [Accessed September 2015].

[16] Olimex Ltd, "Olimexino-stm32," [Online]. Available: https://www.olimex.com/Products/Duino/STM32/OLIMEXINO-STM32/open-source-hardware. [Accessed September 2015].

[17] Seeed Development Limited, "CAN bus shield," [Online]. Available: https://github.com/Seeed-Studio/CAN_BUS_Shield. [Accessed September 2015].