YRKESHÖGSKOLAN NOVIA

Implementation of Unit Test Cases and Simulated ESM-30 Module for Validating the Integration of a New CANopen Stack into WMAP.

Jonathan Finne

Degree Thesis for Bachelor of Engineering

Electrical and Automation technology

Vasa 2022

BACHELOR'S THESIS

Author:	Jonathan Finne
Degree Programme:	Electrical Engineering and Automation, Vaasa
Specialization:	Automation
Supervisors:	Gerald Villavicencio, Jan Berglund

Title: Implementation of Unit Test Cases and Simulated ESM-30 Module for Validating the Integration of a New CANopen Stack into WMAP

Date: 30.4.2022 Number of pages 45 Appendices 0

Abstract

This Bachelor's thesis was assigned by Wärtsilä's Automation System Lifecycle line in the Automation and Control department in Vaasa. The company is developing a new safety module that will support a new set of features that were not possible with the old CANopen stack, therefore the implementation of a new stack was necessary. Consequentially, the implementation needed to be validated and since the ESM-30 was still in development, there were no integration tests for it yet and the Unit Tests written for the previously used stack become obsolete when the source code is replaced, making new Unit Tests crucial.

The theoretical part of the thesis explores the CAN and CANopen communication protocols, Unit Testing and Integration Testing, what to consider, how to write good Tests and how to simulate communication from a physical module. Unit Tests were written for the main portion of the new WMAP CANopen source code and integration tests that simulate communication with the safety module.

The results of the thesis work were a set of Unit Test cases that verify that the functionality of the WMAP source code remains consistent when code changes are made and ESM-30 safety module integration tests for the UNIC system, which will become a part of Wärtsilä's Integration Test suite that is run daily.

Language: English

Key words: communication stack, Unit Testing, Integration Testing

EXAMENSARBETE

Författare:	Jonathan Finne
Utbildning och ort:	El- och Automationsteknik, Vasa
Inriktning:	Automation
Handledare:	Gerald Villavicencio, Jan Berglund

Titel: Implementering av enhetstester och simulerad ESM-30 modul för validering av integrationen av en ny CANopen stack i WMAP

Datum: 30.4.2022 Sidantal: 45 Bilagor: 0

Abstrakt

Detta examensarbete utfördes på uppdrag av Wärtsiläs Automation System Lifecycle linje på avdelningen Automation and Control. Företaget utvecklar en ny säkerhetsmodul med stöd för nya funktioner som inte var möjliga med den föregående CANopen stacken. Syftet med examensarbetet var att validera integrationen av den nya CANopen stacken i Wärtsilä Modular Automation Platform (WMAP). Eftersom ESM-30 modulen fortfarande är under utveckling så har ännu inga integrationstester gjorts och enhetstesterna för den gamla källkoden blir föråldrade och nya testfall måste skrivas.

Examensarbetets teoretiska del utforskar kommunikationsprotokollen CAN och CANopen, enhetstester för mjukvaran och integrationstestning där man simulerar modulen, vad man behöver tänka på och hur man skriver omfattande tester. Enhetstester skapas för den huvudsakliga delen av CANopen källkoden i WMAP och integrationstester som simulerar kommunikation från ESM-30 modulen.

Slutresultatet av examensarbetet är enhetstester som validerar funktionaliteten av WMAP CANopen källkoden när kodändringar görs och integrationstester för integreringen av ESM-30 säkerhetsmodulen i UNIC systemet, som utgör en del av integrationstestsviten som körs dagligen.

Språk: engelska
Nyckelord: kommunikationsstack, integrationstest, enhetstest

OPINNÄYTETYÖ

Tekijä:	Jonathan Finne
Koulutus ja paikkakunta:	Sähkö- ja automaatiotekniikka
Suuntautumisvaihtoehto:	Automaatio
Ohjaajat:	Gerald Villavicencio, Jan Berglund

Nimike: Laitetestien ja simuloidun ESM-30-moduulin toteutus uuden WMAP:iin integroidun CANopen-pinon validoimiseksi

---

Päivämäärä: 30.4.2022 Sivumäärä: 45 Liitteet: 0

Tiivistelmä

Tämä opinnäytetyö tehtiin Wärtsilän Automation System Lifecycle -jaostolla, Automation and Control -osastolle. Yritys kehittää uutta suojausmoduulia, joka tukee uusia ominaisuuksia, jotka eivät olleet mahdollisia edellisen CANopen-pinon kanssa. Opinnäytetyön tarkoituksena oli validoida uuden CANopen-pinon integrointi Wärtsilä Modular Automation Platform (WMAP) -alustaan. Koska ESM-30 moduuli on vielä kehitysvaiheessa, integraatiotestejä ei ole vielä tehty ja vanhan lähdekoodin yksikkötestit vanhenevat ja uusia testitapauksia on kirjoitettava.

Opinnäytetyön teoreettisessa osassa tutkitaan CAN- ja CANopen-kommunikaatioprotokollia, yksikkö- ja integraatiotestejä, mitä pitää huomioida näissä, sekä miten kattavia testejä kirjoitetaan. Yksikkötestit kirjoitettiin suurimmalle osalle uutta CANopen-pinon lähdekoodia ja integraatiotesteistä, jotka simuloivat viestintää ESM-30-moduulista.

Opinnäytetyön lopputuloksena ovat yksikkötestit, jotka validoivat CANopen-pinon lähdekoodin koodimuutosten yhteydessä, sekä integraatiotestit ESM-30-turvamoduulin integroimiseksi UNIC-järjestelmään, jotka ovat osa integraatiotestipakettia, joka suoritetaan päivittäin.

Kieli: englanti
Avainsanat: viestintäpino, yksikkötestaaminen, integraatiotestaaminen

Table of Contents

1 Introduction 1
1.1 The company in brief 1
1.2 Background 2
1.3 Objective 2
2 UNIC and WMAP 3
2.1 Wärtsilä Unified Controls (UNIC) system 3
2.2 Wärtsilä Modular Automation Platform (WMAP) 5
3 CANopen 6
3.1 The CAN-protocol 6
3.1.1 Explanation and history 6
3.1.2 CAN features 7
3.1.3 Data exchange principles and frame formats 9
3.1.4 Error detection and signaling 10
3.2 CANopen 11
3.2.1 Explanation and history 11
3.2.2 CANopen features 11
4 Software Testing 17
4.1 Testing Types 17
4.2 Testing Methods 18
4.3 Testing Approaches 18
4.4 Testing Levels 19
4.4.1 System Testing 19
4.4.2 Acceptance Testing 20
4.5 Unit Testing 20
4.5.1 Advantages 20
4.5.2 Goals 21
4.5.3 How Unit Testing works 21
4.5.4 How to write good Unit Tests 23
4.6 Integration Testing 24
4.6.1 How Integration Testing works 24
4.6.2 How to write good Integration Tests 26
5 Method 29
5.1 WMAP CANopen Unit Tests 29
5.1.1 Function to be tested 30
5.1.2 Writing the test 31
5.1.3 Stub function example 33
5.2 UNIC – ESM30 CANopen communication Integration Tests 36

5.2.1 ESM-30 Setup test 36
5.2.2 Value mapping and validation 40
6 Results 41
6.1 Unit Tests 41
6.2 Integration Tests 43
7 Discussion 45
7.1 Further development 45
7.2 Acknowledgements 45
8 References 46

1 Introduction

In any large company, making changes to a core piece of software can affect several products and many different people's work. The most important thing to do before taking a new feature into use is to make sure it works as intended and does not break any already existing code. Software Testing exists for this reason, unit tests verify that the code works as intended by the developer and Integration tests validate the operational validity of the entire system.

The practical part of this thesis explores the validation of the implementation of a new communication stack into an automation platform through the use of the first two levels of SW Testing, Unit and Integration Testing. The source code itself needs comprehensive unit tests that immediately tell whether a change to a piece of code alters its functionality and the Automated Test Framework needs a basic simulation of the new ESM-30 safety module for its integration tests. How does one go about writing comprehensive unit tests, and how do you simulate a CANopen module in an integration test? These questions and more will be explored.

1.1 The company in brief

Wärtsilä was established in 1834 and started out as a sawmill in Karelia, later they expanded and became an iron works company. In 1935 the company headquarters moved to Helsinki after gaining significant holdings in Kone- ja Siltarakennus Oy (Machine and Bridge Construction Ltd). The company kept growing and the first Wärtsilä diesel engine was created in 1942 after signing a license agreement with the German company Friedrich Group Germania. Since then, Wärtsilä has evolved into an international Marine and Energy company. (Wärtsilä, 2022)

Today Wärtsilä consists of several different businesses, focusing most of its resources on Marine and Energy where the company is a world leader in creating lifecycle solutions and innovative technologies. Wärtsilä produces products, solutions and services to shape the decarbonization of the marine and energy industry and to meet the increased demand for sustainable energy and smart solutions. Wärtsilä's businesses focus on maximizing the environmental and economic performance of vessels and power plants, aiming for a 100% renewable energy future. (Wärtsilä, 2022)

The company strives to maintain all the highest legal and ethical standards while expecting their employees to act responsibly, honestly and with integrity. Openness and transparency with stakeholders, customers, authorities and other business partners are highly promoted, as is respect for human rights and fair employment practices in all its establishments. (Wärtsilä, 2022)

As of the year 2021, the company's sales reached a net total of 4.8 billion euros, with over 17,000 employees in 68 countries. (Wärtsilä, 2022)

1.2 Background

This bachelor's thesis is done on behalf of the Automation System Lifecycle line in the Automation and Control department of Wärtsilä in Vasa. The purpose is to validate the implementation of a new CANopen stack into Wärtsilä Modular Automation Platform (WMAP). It was decided that Wärtsilä's new safety module, ESM-30, which is still in development, would support a new set of features that are not supported by the old CANopen stack used by WMAP and therefore a new stack needs to be implemented. These new features are network redundancy, flying master and layer setting services. Redundancy ensures network availability in case of device or path failure. Flying Master allows for a second NMT capable device to temporarily take over the role as NMT master in case the original master becomes unavailable. LSS makes it possible for the Network Management Master to change a node's Node ID and baud rate.

1.3 Objective

The objective was to validate the implementation itself by writing Unit Tests for the source code in WMAP and simulating CANopen communication in integration tests. The first is achieved by studying the code and writing comprehensive unit tests which when run tells whether the functionality of the code is as intended or not. The second is achieved by studying the CANopen protocol, figuring out how it works and then simulating the safety module's behavior in the Automated Test Framework for integration tests. Part of the objective is also for the reader of this document to gain an understanding of the CANopen communication protocol and for the two first levels of software testing.

2 UNIC and WMAP

2.1 Wärtsilä Unified Controls (UNIC) system

Wärtsilä's Engine Control system is named UNIC, abbreviated from "Unified Control". It is a modular, embedded control system for monitoring and controlling both gas and diesel Wärtsilä engines, developed to optimize the performance and safety of the engine. The UNIC engine automation system facilitates the fundamental safety functions, monitoring, fuel injection and ignition functionalities while also controlling start/stop logistics and speed/load. (Wärtsilä, 2022)

The UNIC modules, actuators and sensors together with the software program UNITool work together to make up the complete UNIC system. The modules are designed with reliability and modularity in mind which allows the system to be mounted directly onto the engine. All modules, sensors and actuators are then wired with insulated point-to-point cables, which offer the best protection against mechanical forces from the engine, electrical disturbances, heat and even chemicals.

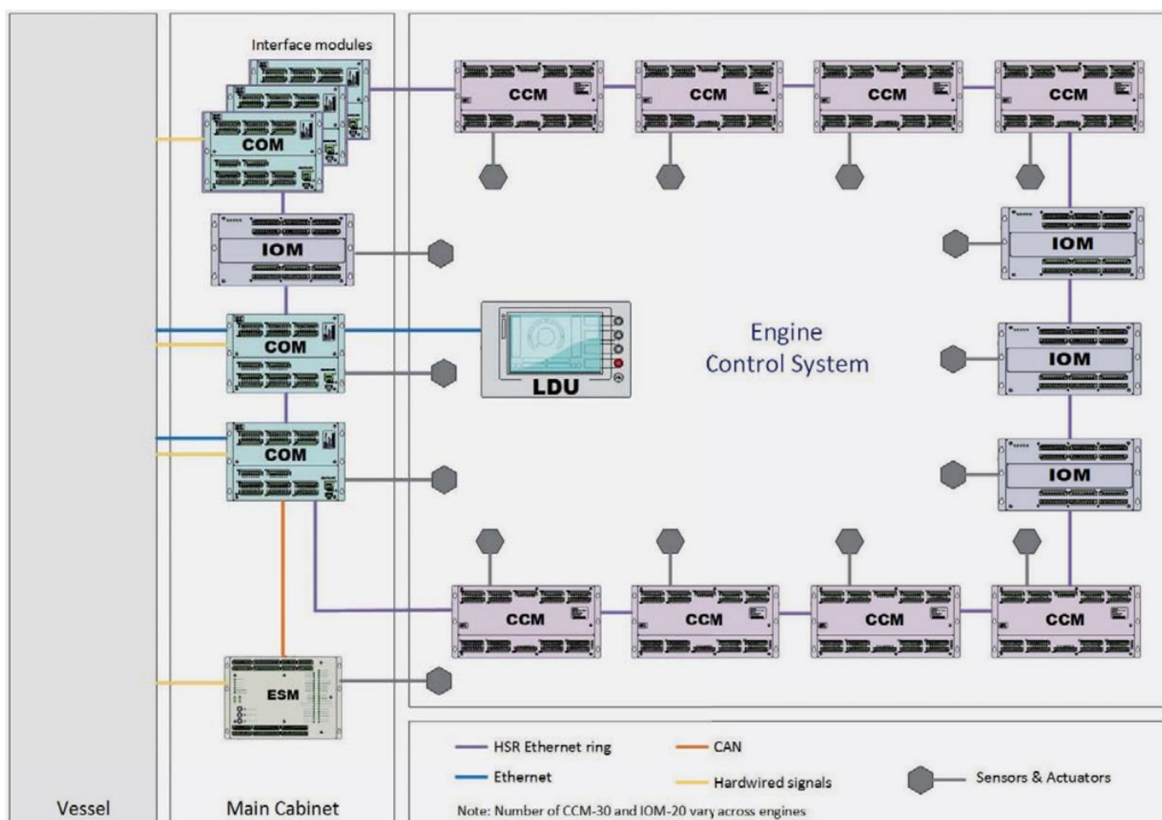

Figure 1: UNIC System overview

UNIC consists of the following building blocks:

COM – Communication module

The communication module is the main gateway between the UNIC system and the vessel systems, the COM module supports multiple interfaces such as hardwired I/O, Modbus, WECAN, CANopen and a few more. The COM module is responsible for several control functions and manages updating of software and configuration of the entire UNIC system. For redundancy's sake, there are generally at least two COM modules in a UNIC network setup.

CCM – Cylinder Control Module

The main responsibility of the Cylinder Control Module is combustion control, it also monitors and controls the inlet valve timing and all the injection and combustion functions for the cylinders. Engines with more cylinders require more CCMs.

IOM – Input/Output Module

The IOM-modules are placed close to sensors and measurable devices and handle all measurements in that specific area of the engine. The engine type, application and number of cylinders influence how many IOM modules are needed in the network.

LDU – Local Display Unit

UNIC's Local Display Unit allows visualization of the information coming from the modules/sensors and operation of the engine through its touch display and physical push buttons. The LDU's key features include status information, local start & stop, trip & shutdown reset, local/remote control selection and emergency speed settings. The physical emergency stop button is also located on the LDU module.

ESM – Engine Safety Module

Lastly, and most relevant to the thesis, the Engine Safety Module provides and handles safety functions ensuring the safety of the crew in case of engine failures, such as shut down in case of cylinder overspeed or low oil pressure among others. The ESM is normally located in the main cabinet with the COM modules. The practical part of this thesis includes

writing integration tests for the communication between the COM module and the new ESM module.

(Wärtsilä, 2017)

2.2 Wärtsilä Modular Automation Platform (WMAP)

A software platform is like an ecosystem, consisting of a selected set of software and resources that work together. The beginnings of WMAP go all the way back to the 90s when the UNIC 2 systems precursors WECS and UNIC 1 were used. Nowadays with UNIC 2, WMAP takes on a more modular approach regarding control of the engine. The overall structure of the entire system is as shown below.

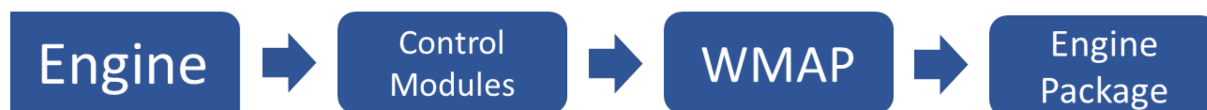

Figure 2: UNIC structure overview

The system begins with a Wärtsilä engine, typically running on diesel, used for power generation in ships or power plants. Control modules are installed on the engine, usually there are multiple modules like COM-10, CCM-30 and IOM-20 to name a few. These modules run on Linux with Xenomai, a framework adding real-time function to the Linux OS. Real-time means that data and events are processed according to limited time constraints.

WMAP is a software platform that builds Application Programming Interfaces (APIs) and a set of primitives on top of Xenomai. WMAP does not do any actual engine control by itself, for that it needs an Engine Package. The Windows application UNITool is used for configuring and maintaining an instance of WMAP. The created engine package contains the configuration of the engine and a set of applications that use the WMAP APIs and are run as Xenomai tasks.
(Matias Muhonen, Wärtsilä, 2019)

The CANopen source code and its accompanying unit tests are contained in WMAP together with the rest of the necessary technical implementations.

3 CANopen

In this section I will explain both the lower-layer CAN-protocol and the higher-layer CANopen protocol. For the purposes of the practical part of the thesis, it is not needed to go quite this in-depth about the communication protocols, but this document was written with the intent of giving the reader an understanding beyond what is strictly necessary.

3.1 The CAN-protocol

CANopen is a communication system based on the CAN-protocol. To explain CANopen, we first need to get familiar CAN.

3.1.1 Explanation and history

A communication protocol is simply a set of rules that govern how a particular technology communicates, kind of like how humans speak by using different languages, devices can't understand each other unless they are using the same communication protocol. (W3Schools, u.d.) Since CAN is only the base upon which CANopen is running, I will not go into too much detail regarding this lower-layer protocol.

CAN is an abbreviation of "Controller Area Network" and is a communication protocol mostly used by embedded electronics in vehicles. Its creation was started in 1983 by Robert Bosh and was first taken into use 3 years later in 1986, improving reliability, fuel efficiency – and most importantly – safety in automobiles. One of the most important changes that made this new communication protocol superior was the fact that many electronic modules could now be connected and communicate with each other using a single common communication bus. By 1993 CAN was adopted as an international standard with ISO-11898.

(Parikh, 2021)

In cases where only two electronic devices need to communicate, other protocols work just fine, but a vehicle contains a whole network of devices that need to share information with each other. Before CAN was introduced, every device had a separate wire connected to every other device it needed to communicate with, this became more and more inefficient as electronics got more advanced and vehicles started using more modules and sensors at

the same time with more complex information shared between them. CAN solved this problem by connecting every, or at least several, modules to a common serial bus, reducing physical wiring and the overall complexity of the entire system. Today CAN is the dominating serial network for embedded control systems in most passenger cars and other automotive applications. (Parikh, 2021)

3.1.2 CAN features

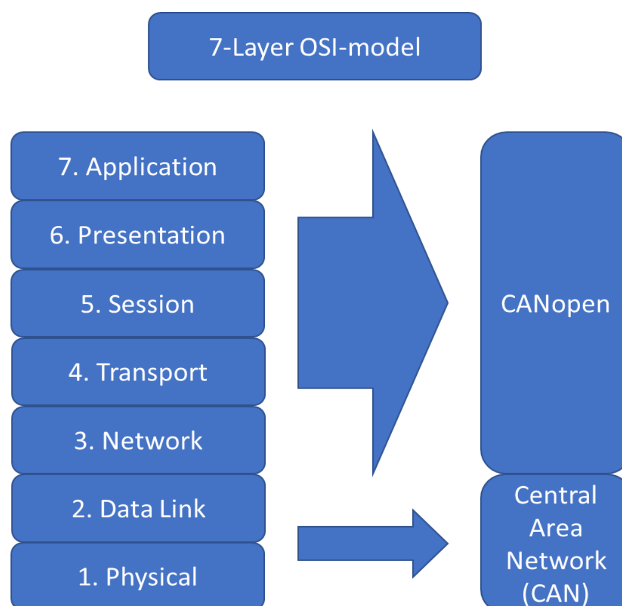

Figure 3: CAN/CANopen OSI-model

The CAN protocol defines most of the two lowest layers in the seven-layer Open System Interconnection (OSI) model. The OSI-model is a conceptual model describing communication functions across communication technologies. The lower layers describe more basic communication while the higher layers describe more specific aspects regarding how the communication is handled. The OSI-models seven layers consist of, in descending order, the Application, Presentation, Session, Transport, Network, Data Link and Physical layers.

CAN defines the Data Link layer and most of the Physical layer, the higher layers can be defined by the system designer, or an existing higher-layer protocol can be used to implement the remaining layers, for example CANopen. The physical CAN bus itself consists of two wires, CAN low and CAN high and use baud rates up to 1Mbit/s for Classical CAN. The higher the baud rate is, the shorter the physical wiring should be with a recommended length being below 40 meters for 1Mbit/s and 500 meters for 125kbit/s. These wires also

need to be terminated at both ends of the bus using 120 Ohm termination resistors. In the physical layer, Physical Signaling (PS) is defined by CAN while Physical Medium Attachment (PMA) and Medium Dependent Interface (MDI) are not defined by CAN. This means that the system designer can use any driver/receiver and medium of transport as long as they meet the Physical Signaling requirements. The CAN specification defines Logical Link Control (LLC) which manages message filtering, overload control and recovery management and the Medium Access Control (MAC), which performs data encapsulation, bit stuffing, error detection and serialization/deserialization.

(Richards & Microchip Technology, 2002)

The Classical CAN data link layer has been internationally standardized in ISO 11898-1 (2015) and supports several very important features, some of the most important are:

- Multi-drop capability. All components are connected to the same electrical circuit, allowing the systems designer to create a distributed and redundant communication system. Distributed, meaning a system with multiple components located on different machines that coordinate their actions and communicate in a way that makes it seem like a single coherent system to the user. (Gibb, 2019) Redundant, meaning that the network has a backup mechanism for moving network operations onto a redundant infrastructure in case of an unplanned network outage. (techopedia, u.d.)

- Broadcast communication. This means that a single message is received by several devices at once instead of sending a separate message to each device, this reduces bandwidth requirements and makes the system more flexible.

- Sophisticated error detection functions and automatically re-transmitting erroneous frames, making the network highly reliable.

- Fault confinement, stopping malfunctioning devices from repeatedly sending erroneous frames by identifying hardware or software disfunction and switching off defective devices, guaranteeing data consistency on the network.

(CiA: CAN in Automation, u.d.)

3.1.3 Data exchange principles and frame formats

The Classical CAN-protocol is based on the mechanism of "broadcast communication". Every frame that is broadcast on the network contains an identifier that indicates what kind of information the message contains and its access priority, this means that every identifier in the network needs to be unique. It's important to know a message's priority when several nodes are competing for network access. Adding a new receiver CANopen node to an existing network is very easy and can be done without any hardware or software modifications to the already existing network. This way you can have a modular network and keep the distributed processes synchronized. (CiA: CAN in Automation, u.d.)

Table 1: Classical Base Frame Format (CBFF)

SOF	Identifier	RTR	IDE	R0	DLC	DATA	CRC	ACK	EOF	IFS
1	11	1	1	1	4	0 – 64	15	1	7	3

The Classical CAN protocol supports two different frame formats, what separates them is the length of the identifier. "Classical Base Frame Format (CBFF)" supports an 11-bit long identifier while the "Classical Extended Frame Format (CEFF)" supports a much longer 29-bit CAN identifier. The data frame starts with the "Start of Frame (SOF)" bit followed by the "Arbitration field", which provides the identifier and a "Remote Transmission Request (RTR)" bit. This bit indicates whether the frame is a CAN data frame or a CAN remote frame. Next up is the "Control Field" which contains both an "Identifier Extension (IDE)" bit and the "Data Length Code (DLC)", the IDE shows whether the frame uses the base frame format or extended frame format.

The "Data Length Code" indicates the size of the frame's data field in number of bytes, in a Remote Frame, the DLC contains the number of requested data bytes. The "Data Field" itself contains the application data, which can vary from zero up to eight bytes. To guarantee the integrity of the frames, a "Cyclic Redundant Check (CRC)" sum is used, receiving nodes confirm correct reception by transmitting a dominant bit in the ACK slot of the "Acknowledge field (ACK)". Every frame ends with an "End of Frame (EOF)" field followed by an "Inter Frame-Space" field containing a 3-bit intermission, bus-idle time and an 8-bit suspend transmission in case of an error passive node.

Table 2: Extended Base Frame Format (EBFF)

SOF	Identifier	SRR	IDE	Identifier	R0	R1	DLC	DATA	CRC	ACK	EOF	IFS
1	11	1	1	18	1	1	4	0 – 64	15	1	7	3

The extended frame format differs in its length and contents of the identifier. The 11-bit "base identifier" is still there, but there is also an 18-bit "identifier extension". The drawbacks are that EBFF frames require about 20% more bandwidth, increasing network latency and making error detection slower since the CRC's correctness takes longer to verify. All Classical CAN implementations must be capable of handling both frame formats since the year 2003. (CiA: CAN in Automation, u.d.)

3.1.4 Error detection and signaling

The CAN protocol's Data Link layer makes use of five error detection mechanisms, achieving an extremely high reliability:

- Cyclic Redundancy Check. A frame check sequence is calculated and added in the CRC field of the frame, the CRC is calculated depending on the contents of the frame itself and when a device receives the frame it immediately computes its own CRC sequence and compares it with the CRC given in the received frame. If the sequences do not match, a CRC error has occurred.

- Frame checking. The received frame is compared against a fixed format and frame size, making sure the structure is correct by checking the bit fields. Errors detected by this method are classified as "format errors".

- Acknowledgement error. When a device receives a frame it acknowledges this by sending a dominant bit in the ACK field, if the transmitter does not receive this acknowledgement, it indicates an ACK error.

- Bit monitoring. Since every node is constantly monitoring the network signals, this means that the transmitting node can read back its own data from the CAN bus to ensure that the signal has been transmitted correctly. If the node reads a bit that is not the same as the transmitted bit the node will report a Bit-error.

- Bit stuffing. This is a method the nodes use to make sure long rows of bits with the same logical value is received correctly. Whenever there are 5 consecutive bits with the same value, the node inserts a stuff bit with opposite logical value, the receiving nodes automatically remove the stuffed bit from the received data.

If one or more of these errors are detected, an error frame is sent causing all other nodes in the network to detect an error themselves and the currently ongoing transmission is aborted. The receivers now patiently wait for a retransmission of the aborted Data Frame. To avoid that the network is permanently disrupted by a device repeatedly sending erroneous frames, the aforementioned “fault confinement” causes the faulty node to switch of its CAN interface. (CiA: CAN in Automation, u.d.)

3.2 CANopen

That’s it for the CAN-protocol, this next chapter covers the higher layer protocol CANopen.

3.2.1 Explanation and history

CANopen is a communication protocol based on CAN, which means that CAN bus serves as the ‘transport vehicle’ for CANopen messages. Development of CANopen started in 1993 and the first version of the CANopen specification was published by CiA (CAN in Automation) in November of the following year. Originally CANopen was designed for motion-oriented machine control systems but today it is used broadly in application fields such as vehicles, maritime electronics, medical equipment and building automation. It was developed as a standardized embedded network to be highly flexible regarding configuration capabilities and to enable modular and flexible combinations of existing manufacturing cell units. CiA’s latest CANopen specification, “CiA 301, CANopen publication layer and communication profile 4.2” was released to the public in 2011.

(CiA: CAN in Automation, u.d.)

3.2.2 CANopen features

The CANopen protocol enables interoperability between devices of-the-shelf which is one of its most useful features along with providing standard methods for configuring devices. While the CAN bus represents the two lowest layers of the OSI-model, enabling

transmission of frames with an 11-bit identifier, an RTR bit and 64-bits of data, CANopen covers mainly the transport and application layers leaving the presentation, session and network layers unused. The application is implemented via a set of standards, adding several important concepts to the communication system.

Table 3: CANopen frame

Function Code	Node ID	RTR	Data Length	DATA
4	7	1	4	0 – 64

In CANopen, the 11-bit CAN-ID is referred to as the Communication Object Identifier or COB-ID and is split into two parts. By default, the first four bits work as a function code and the seven following bits contain the CANopen node ID. The function code describes what kind of message is being transmitted, from NMT to HEARTBEAT and many others in-between.

Table 4: CANopen communication protocol codes

NMT	0000
SYNC	0001
EMCY	0001
TIME	0010
Transmit PDO 1-4	0011-0101-0111-1001
Receive PDO 1-4	0100-0110-1000-1010
Transmit SDO	1011
Receive SDO	1100
HEARTBEAT	1110

The CANopen protocol adds six new core concepts on top of the basic CAN functions.

- Communication models: A CANopen communication system can use any of these three different models for device/node communication: master-slave, client-server and producer-consumer.
- CANopen communication protocols: Different protocols depending on what kind of data is being transmitted, e.g., SDO for configuring nodes, PDO for sending real-time data, NMT for controlling node states. These and more will be explained further later.
- Device states: The master node can change the operational states of slave nodes, e.g., resetting it or stopping it in case of malfunctioning.
- Object dictionary: Every node contains an object dictionary with entries specifying, for example, the device configuration, these entries can be accessed via SDOs.
- Electronic Data Sheet: Every node type has a standard EDS containing the object dictionary entries for that type.
- Device Profiles: Describing standards for e.g. I/O modules and motion control for vendors.

(CSS Electronics, 2021)

3.2.2.1 Communication models

The three different communication models used by CANopen are very similar in the end. The Master-Slave model is used for Network Management (NMT) commands, one node acts as host controller/application master. The master node sends or requests data to or from a slave. In a standard application there can be 0-127 nodes in the same network.

Client-Server is used e.g., when the application master needs object dictionary data of a slave. The master acts as the client and the slaves as servers. The client sends a data request to a server which then responds with the requested data. Reading from a server is called “uploading” while writing to a server is called “downloading”, the terminology makes sense when thought about from the server’s perspective. SDO’s use this model of communication.

Consumer-Producer is the third model used by CANopen. Here the producer of a message broadcasts data on the network, which the consumer nodes read. The producer sends its data either on request, called the “pull model”, or without a request, called “push model”. This model is used when nodes broadcast their heartbeats, every node in the network acts as both consumers and producers depending on the message.

(CSS Electronics, 2021)

3.2.2.2 CANopen Communication protocols

Network Management (NMT) is one of the most important protocols in the CANopen network. Using NMT commands like start, stop and reset, nodes can be set to different states like pre-operational, operational or stopped state. The NMT master sends a 2-byte message with the first byte containing the requested state and the second byte containing the node ID of the target node. This message is sent with CAN-ID 0, indicating it as a broadcast command for all nodes in the network to read.

Synchronization (SYNC) messages are typically triggered by the application master and are used e.g., for synchronizing actuation of CANopen devices and sensing of inputs. A SYNC-message is broadcasted on the network with COB-ID 80, nodes configured to react to sync messages can respond by transmitting their latest captured real-time data or by setting their output at the exact same moment as the other nodes participating in the synchronous operation.

Emergency (EMCY) messages are sent when a device experiences a fatal error, for example a sensor failure. The emergency message has a COB-ID of 80 + Node ID and contains information about the error experienced.

The Timestamp (TIME) is used to distribute a global network time. The master sends a frame containing a CAN-ID of 100 and 6 bytes of date and time information. The first 4 bytes give the time in ms since midnight and the last two bytes give the number of days since January 1, 1984.

Process Data Objects (PDO) are used for sharing real-time data between devices, e.g. sensor data like temperature or pressure and are often seen as the most important

protocol used in CANopen. A PDO frame contains 8 bytes of data and can even contain several object values within the same frame, making them ideal for real-time sensor data transmitting. PDO's use the consumer/producer model, a node produces data, transmitting it to the consumer via "Transmit PDO" (TTPDO), typically the NMT master. The producer can also request data by sending a "Receive PDO" (RPDO). Producers can for example be configured to respond to a periodic SYNC trigger broadcasted by the consumer. The producers then broadcast a PDO containing data from one or several parameter values. The consumer needs to be configured in advance to be able to interpret these parameter values, e.g., when a node sends a PDO1 with COB-ID 180 + node ID, the master needs to know already what the incoming data describes. That's why they are preconfigured as PDO1, PDO2, PDO3 etc.

Table 5: CANopen TPDO

TPDO COB-ID	Value from Data Parameter 1	Value from Data Parameter 2	Value from Data Parameter 3
11	8	16	32

Utilizing a client/Server model, the Service Data Objects (SDO) can access and change values in a CANopen devices object dictionary. The "client" initiates communication with a "server", in simple CANopen networks, the NMT master acts as the client and the slaves act as servers. The purpose of the SDO can be updating an object dictionary entry e.g., when the master needs to change the configuration of a slave. The master sends a frame with COB-ID 600 + node ID, only the node with that specific node ID will accept the frame. The rest of the frame consists of different instructions and the data for the receiving node to download.

Table 6: CANopen SDO

COB-ID	CCS	Empty bit	n	e	s	OD 16-bit index	OD 8-bit sub index	Data
11	3	1	2	1	1	0-16	8	32

As mentioned earlier, the COB-ID consists of a 4-bit function and a 7-bit node ID. The Client Command Specifier (CCS) decides the transfer type, 1 for download to the module, 2 for Upload to the client. N indicates the number of bytes of the data bytes that do not contain data. E, “Expedited transfer” indicates whether all data was able to fit in a single frame. “S” indicates whether data size is shown in “n”. The 16-bit index and 8-bit sub-index gives the object dictionary address to be read or written to. And finally, the data bytes contain the data to be downloaded to the node. The master node first sends a “SDO receive” frame with COB-ID 600 + node ID to which the targeted node responds with an “SDO Transmit” frame with COB-ID 580 + node ID, if the master requested an upload, then this frame also contains the data from the OD entry requested by the master. If the master indicates a download, then the “SDO Transmit” just contains the index and sub-index and 4 empty data bytes. Technically, SDO’s could also be used for real-time data sharing but, since they also need to contain a command and the OD addresses, they can only carry 4 bytes of data, making PDO’s far more useful.

Node Monitoring (Heartbeat) is a crucial SDO service and has two purposes: To transmit periodic “alive” messages, telling the NMT-master that it is operational and to confirm the NMT command. The heartbeat is produced periodically, e.g., every second, and consist of a CAN-ID 700 + node ID and the node’s operational state in the frame’s first data byte. The NMT master acts as a consumer and reacts if no heartbeat is received in a certain amount of time. Nodes also use the heartbeat to answer NMT commands, when the NMT master changes the state of a node it can e.g., answer with a pre-operational heartbeat or an operational heartbeat.

(CSS Electronics, 2021)

4 Software Testing

According to the Institute of Electrical and Electronics Engineers ANSI/IEEE 1059 standard, testing is – “A process of analyzing a software item to detect the differences between existing and required conditions and to evaluate the features of the software item.” In layman’s terms, software testing is the process of evaluating whether the functionality of a software application meets the specified requirements or not and identifying these potential defects to ensure the production of a quality product.

Agile and DevOps as software development approaches are the biggest trends in software development as of 2021 and they are changing the face of Software Testing. Agile emphasizes continuous learning and planning, team collaboration and incremental delivery, making Software Testing an inevitable and important part of the development process. DevOps is a combination of Development and Operations working to shorten the distance from the start of development to delivery and to improve the quality of the product. Software testing is an integral part of the development cycle.

(STM Rajkumar, 2021)

Software testing can be split into many different types, methods, approaches and levels. The world of software testing is very broad and deep so I will not explore every different aspect of software testing, but I will give an overview of the important aspects and dig deeper into the areas relevant to this thesis, namely Unit and Integration Testing.

4.1 Testing Types

Firstly, tests can be either manual or automatic, these are relatively self-explanatory, but manual testing is when a person manually tests the software application, taking the role of an end-user and searching for unexpected behaviors and bugs in the application, a manual tester does not use any automated scripts or tools.

Test Automation or automation testing is when the testers or developers use testing software or write scripts to create tests that can then be run by themselves to test the application. Even though the tests are automated, they still have to be created manually by developers and the test results need to be evaluated for manual troubleshooting. Therefore, the tests should be written in a way that makes it clear what is being tested and

error messages should be sufficiently descriptive. Test automation automates a manual process, thereby saving time in instances where one or several tests need to be run and re-run repeatedly and quickly. A computer works insanely fast compared to a human and does not make any mistakes, neither does it disregard small errors in the test results. In Software Development Life Cycle (SDLC), you can technically start writing tests already from the requirements gathering phase all the way until the software is finally deployed.

In some scenarios, a good optioncan even be to write the test for the software before writing the software itself. For example: Suppose you need to create a function that takes a specific range of inputs and gives predetermined outputs depending on said inputs, then you can quickly write a test that ascertains that the given output is the desired one. When writing the code for the software you can now immediately check whether it works as intended or not by running the test. Software testing is a continuous process, it is said that no software can ever truly be 100% tested, there is always another test case to write or an opportunity to increase the code coverage.
(STM Rajkumar, 2021)

4.2 Testing Methods

Testing "method" can be either static or dynamic, also known as Verification and Validation respectively. Static testing, or Verification, is the act of verifying whether the product is being built correctly by comparing the product to its requirements and checking the documents and files. The goal is to determine if the product after a development phase satisfies the criteria given at the start of said phase. Dynamic testing, Validation, is done by testing the real product and actual output of the software application at the end of the development process to determine whether the product satisfies the specified requirements given at the start of development.
(STM Rajkumar, 2020)

4.3 Testing Approaches

The different testing "approaches" are White Box Testing, Black Box Testing and Gray Box Testing. White Box Testing can also be called Glass Box, Clear Box or Structural Testing and is based on the internal code of the application. For White Box Testing, you need

perspective from the inside, access to the code itself and programming skills to write the test cases, therefore White Box Testing is usually done by the same developers that are working on the software.

Black Box Testing requires no access or knowledge about the underlying code structure and is sometimes called Behavioral Testing, Specification-Based Testing or Input-Output Testing. Black Box Testing tests the software by evaluating the functionality from an end-user perspective, creating test cases based on specifications and requirements.

Gray Box Testing is, as the name suggests, a combination of the two earlier approaches. The tester aims to search for defects caused by improper structure or usage of the application by studying the design documents.
(STM Rajkumar, 2020)

4.4 Testing Levels

Software Testing is generally split into four levels with different purposes: Unit Testing, Integration Testing, System Testing and Acceptance Testing. Each one of these four levels verifies the quality, performance, correctness and functionality of the software. Since this thesis work is in regards to writing Unit and Integration Tests, I will write more in-depth about those two and just briefly go over the later levels, System and Acceptance testing, first.

Figure 4: Testing Levels

4.4.1 System Testing

Also known as “End to End Testing”, is the penultimate testing level and uses a Black Box approach. System Testing is done near the end of development when ensuring that the fully integrated application works as intended together with the target systems. This is done by testing every input in the application and verifying that we get the desired outputs, in other words, testing how the user will experience the application.

4.4.2 Acceptance Testing

Acceptance testing is the final level of testing and is done to gain customer sign-off on the software. There are three types of acceptance testing: Alpha, Beta and Gamma testing. Alpha testing is done by performing usability tests which are most often done by the in-house developers who worked on the software or by the client together with testers. Beta testing is done before software delivery by an appropriately sized number of end-users. Beta testers give feedback, improvement ideas and reports any defects they find which are then fixed by the developers. Lastly, we have Gamma testing, this is the last phase before the software is released. This is to confirm that the product is indeed ready for market release according to the specified requirements with a special focus on security and functionality.

(STM Rajkumar, 2019)

4.5 Unit Testing

With those out of the way, now back to the first level in the software testing process, Unit Testing, also called Module Testing or Component Testing. This is where individual pieces of code are tested separately, "unit" meaning the smallest testable part of an application. Unit Tests are white-box and can be either manual or automated tests written by the software developers in the same environment as the software. In later levels several parts of the system are tested together, whereas in Unit Testing every component of the system is tested individually, this is to make sure every component is working correctly before testing them together.

4.5.1 Advantages

There are many advantages to having Unit Testing be a part of the development cycle. Firstly, it finds problems early in the development, reducing the cost of later testing since the cost of fixing a bug earlier in the development is always lower than having to fix it later after it has started affecting other parts of the system. Secondly, it simplifies the process of debugging, which is when you find and resolve defects preventing the desired operation of the software. When a unit test that usually passes suddenly fails, you only need to debug the most recent code change to find what went wrong. Also, when changing existing functionality, unit tests reduce defects due to them being run after every change to confirm

the changes are working correctly. Lastly, it improves the coding practices and standards, making the code easier to read and document.

4.5.2 Goals

The goals of using Unit Testing in your development cycle is to save on testing costs and to find bugs as early on as possible as well as isolating every section of the code and making sure these individual parts are working correctly. Tests being "isolated" means that they can be run in whichever order and does not affect each other in any way, e.g., one test failing should not have any bearing on the other tests. Speed and stability are also very important, test cases should not include any randomized variables or unnecessary operations, every test case should be concise and deliberate for testing one precise thing. Unit Testing also allows developers to more easily upgrade or refactor the code further on into the development.
(STM Rajkumar, 2019)

4.5.3 How Unit Testing works

Unit Testing is automated testing, but the test cases themselves need to be written manually. A test case is simply a chunk of code that calls the function you want to test with parameters of your choice and checks if you get the desired output. Generally, you follow the principle *"test only one thing per test case"*. To give a very simple example, say you have a function that takes as a parameter an integer between the numbers 0–99 and returns a value from a list. To test this function, you simply create a loop that calls the function with every parameter and asserts whether the correct value is returned from the list, for further testing you can test a parameter outside the 0–99 range, a decimal number or even a string of characters. As you can see, this means that the person writing the test cases needs to understand what the function is supposed to do and what it's supposed to return given certain parameters as input. As the functions to be tested get more advanced, the test cases will also naturally get more advanced.

4.5.3.1 Fakes

When performing Unit Testing, you might come across situations where you don't have access to the full code due to it simply not being written yet, or maybe the function interacts with an external system, such as a database. To be able to test every function

separately, both the tests and the functions to be tested need to be isolated from other functions and parts of the software. To allow for this, test doubles such as fakes, stubs, and mocks should be used. Fakes are simplified versions of the production objects. For the database example we can create a fake implementation that uses a simple in-memory storage of data which allows us to test the function quickly without starting up a real database. The function we are testing does not notice any difference.

4.5.3.2 Stubs

For stubs, say for example we have a function that takes a sensor value as input, calls two other functions, one which calculates and returns the value's byte size and the other the values difference from the sensor's average, and then returns those two values. Now, if something happens to be wrong with one of those other two functions, our test will fail, which can at first make us think there's something wrong with the main function. In order to get the function completely isolated, we use mocks and/or stubs. In this example we create two stubs that act as those two other functions. Now when the first function tries to call one of those other two functions, the test framework redirects the call to the stub function instead. This stub function can be made as simple or as advanced as needed, when using stubs, you always define exactly what data will be returned when called. This way we can test that the main function works, without relying on whether those other functions are working correctly or not, those two will be tested separately by their own test cases.

4.5.3.3 Mocks

Mocks are used to register function calls. When asserting a test, you can verify with the mock function that the expected actions have been performed. An example can be a function that prepares an automated email to be sent, you do not want a real email to be sent every time you run the test and the function itself will probably not return anything. With the mock function you can verify that the "send email" command has been executed or the email sending service has been called with the right parameters without sending a real email. Observe that this test does not check whether the email is sent correctly, but that is not the responsibility of this test, that is the responsibility of the test for the email service software. Our test is checking that the email service is called with the right parameters from the function we are testing.

(Software Testing Magazine, 2017)

4.5.4 How to write good Unit Tests

Even after learning what Unit Tests are and what they are supposed to accomplish, it might still be difficult to actually write useful test cases. Here I will bring up some points made by professionals regarding what properties unit tests should have. Some of these have been brought up earlier in the text and some are more important than others, the writer can eventually sacrifice a property if this leads to the test receiving a more valuable property in return.

4.5.4.1 Properties

- Firstly, Isolated. This property has been mentioned earlier but it is one of the most important ones. What this means in practice is that tests do not affect each other and return the same results regardless of in what order they are run.

- Fast and Composable. The tests should be straight to the point and run quickly, there is no limit to the amount of Unit Tests that can belong to a test suite.

- Writable and Readable. The writing of the tests should be cheap compared to the cost of the code that is being tested. The tests should also be clear and comprehensible for the reader, motivating the writer to design understandable code.

- Specific and Deterministic. If a test fails, it should be clear what the reason for failing is. Deterministic means that if the input remains the same, the output also remains the same, there are no uncertainties or random outcomes.

- Inspiring and Predictive. Passing tests should inspire confidence in the code and the product. At the stage when all tests pass without issues, that should indicate that the code under testing is suitable for production.

- Behavioral and Structure-insensitive. Test cases should be sensitive to any change in the behavior of the tested code, if a developer accidentally changes even just a small behavioral aspect of the code, the test result needs to catch that. Although if only the structure of the code is changed, but the functionality remains the same, then the test results should not change.

- Lastly, Automated. After the test cases have been written, they should run without need of human intervention. No need to manually enter parameters or check outputs.

On occasions where a large amount of test cases are needed, it is oftentimes favorable to sacrifice “inspiring and predictable” in order to make the tests fast, writable and specific. Looking outside of Unit-Testing, in acceptance testing there might be a requirement to make the tests more readable for non-programmers, then speed and specificity will most likely suffer. Real-time monitoring can also be considered testing, monitoring is neither predictive nor automated, but you get fast feedback and can manipulate the inputs to simulate real use.

(Beck, 2019)

That’s it for Unit Testing, the first level of Software Testing, next chapter will cover the following level, Integration Testing.

4.6 Integration Testing

Software applications consists of multiple modules conversing with each other through an interface. When integrating a new software module to an already existing system, they all need to be tested together, which is where Integration Testing comes in. Integration Testing is the second level of the software testing structure, being the next step after Unit Testing and has become increasingly more important over the years. I&T can be both white-box and black-box depending on what aspect is tested. Generally, there is a test plan for what needs to be tested and what is prioritized the highest, not just regarding I&T but regarding all testing during the products development. I&T is usually significantly slower than Unit Testing due to it covering a larger part of the application and interacting with many different parts such as databases, network endpoints or file systems.

4.6.1 How Integration Testing works

Integration Testing, also known as “Integration and Testing” or “String Testing” tests the connectivity and data transfer between modules of the software. Where Unit Testing sees each component as an independent system, Integration Testing sees all integrated modules as a single system. I&T does not focus on individual modules but on the communication

between modules, all software modules should be unit tested before they are integration tested. Like with all testing, the objectives of Integration Testing are to reduce risk and find defects as early on in development as possible, preventing them from causing resources to go to expensive bug fixes in later stages of the product's development. Integration Testing is also done to build confidence in the quality of the product and to verify that the functionality of the product is as specified by the requirements.

Imagine we are creating a product consisting of four modules, e.g., a web application consisting of four webpages: the main page, login page, profile and settings. These four modules will not be completed at the same time, in order to test one of them independently we need to use fake modules which are called Stubs and Drivers, these are explained a bit later in the chapter.

4.6.1.1 Approaches

I&T can be divided into four different approaches: Top-Down Approach, Bottom-Up Approach, Big Bang Approach and Sandwich/Hybrid Integration Approach which is a combination of the Top Down and Bottom-Up approaches. I&T also makes use of test fakes, namely stubs and drivers.

Top-Down approach integrates and tests the most important modules first. Here, stubs are used in replacement of unfinished submodules to test the main module. Once the most important parts work correctly, you then integrate the submodule and test them together. In our web application example, this means we would test the main page first, using stubs and drivers in place of the other pages.

Bottom-Up is, as the name implies, the other way around. Submodules are tested first using a driver to simulate the main module. When all submodules are working correctly, you integrate the main module and verify its functionality. This would test the login, profile and settings pages first, using a driver instead of the main page in our web application.

In the Big Bang approach, all individual modules are made ready before any of them are integrated, they are then tested all at the same time. This approach can make it difficult to trace the origin of failures due to the late integration. It's always best to catch errors and mistakes as soon as possible in the development, making the other approaches safer

options, although this approach does have the significant advantage that no stubs or drivers need to be written.

(STM Rajkumar, 2020)

4.6.1.2 Stubs and Drivers

Stubs and drivers have similar objectives and features, acting as alternatives to the real software, existing to substantiate the essential requirements of inaccessible modules. Drivers are similar but instead of simulating the called function, they simulate the function doing the calling. Mostly used in Bottom-Up testing, they simulate the behavior of the main module calling the sub modules. Drivers establish the test environment, taking care of the communication and sending reports, making them more complex than Stubs.

From the Unit Testing chapter, you probably remember stubs as fake functions that get called in place of the real function and simply return a predetermined value. In Integration Testing, stubs can be a bit more advanced, they are used in the Top-Down approach and simulate the behavior of the sub modules. You can also store data with Stubs, for example, to keep track of the number of times a message has been sent. Stubs can also show trace messages, display parameter values, return consistent values or specific parameters to the main module. For the integration tests for the UNIC system, the ESM is still under development and therefore its behavior needs to be simulated.

(javaTpoint, u.d.)

4.6.2 How to write good Integration Tests

Integration Testing, as mentioned earlier, is more complicated and takes more time than Unit Testing, depending on how good coverage you're aiming for. I&T can be considered an extension of Unit Testing since it focuses on cross-cutting concerns, meaning parts of the program relying on or affecting other parts of the system, such as performance, security or data validation. With I&T you make sure that the entire system, as a whole, is working, which is something that cannot be confirmed with Unit Testing alone, no matter the number of tests written. To get the most out of integration testing, you need to know the best practices.

Remember to keep Unit test and Integration test suites separate. Since Unit tests are much faster, combining both into one suite will hurt productivity. A suite of Integration tests can easily take up to 15-20 minutes to run, while the Unit test suite's runtime is down in single digits, if you are working on the Unit tests and have both combined, then you lose a lot of time every time you execute the tests, which is very often. It is incredible easy to get distracted by, for example, social media when waiting for tests to finish, the faster you can make the test suite run, the better, provided that quality does not suffer.

Don't use Integration tests for everything. It's important to know what cases belong to which testing level, since Unit tests are much faster, it is important to use those as much as possible and only use I&T for things that cannot be Unit Tested. Having a good combination of both levels of testing will make it a lot faster and easier to iterate on the application. Make sure that each test case adds value by testing a separate thing, don't for example, have many cases that test similar things just with different parameters.

Use a descriptive naming convention. This is vital in all testing levels, a good naming convention makes sure that the tester immediately knows where and what has gone wrong. A solid convention to use is the *Feature_ExpectedBehavior_ScenarioUnderTest* template. **Feature** represents the module or unit that is being tested. **ExpectedBehavior** is the output we are expecting. **ScenarioUnderTest** represents the circumstance or scenario that is being tested. An example of using this convention could be *TempSensor_EMCY_Overheat*. The tester will immediately know, just from the name, which feature is not working. For my integration tests a simpler convention is sufficient, *test_module_feature*. **Test** is just to indicate that it is a test case, **module** tells which module is concerned and **feature** is just the feature that is being tested, for example, *test_ESM30_txPDO* tests the transmit process data object.

Include test execution as a part of your continuous integration/development process. The tests are meant to identify issues with integration of new software, create a dedicated environment where you can run the tests without having to spend time waiting for them to finish, hindering your work on development. In Wärtsilä, integration tests are run daily on a physical testing rack to ensure newly integrated code changes don't cause problems. This environment needs to be as similar to the real product as possible to give best results.

Reset all test data between execution of test cases. Like with Unit Testing, Integration tests should also be independent and isolated from each other, which makes one of the most important things to remember being to reset test data after every case execution. This ensures that test cases are not dependent on other parts of the system and helps with debugging since it's easier to know where a bug originates from when you know that the state is clean before each test. Test cases that randomly fail due to accidental dependency on other parts of the system are called "flaky" tests.

(Kralj, 2022)

Comprehensive error logging. Integration tests have bigger scope and can span multiple software modules and devices, so when a test fails, it is much harder to know where exactly things went wrong. By extensively logging every step of the test, it's much easier to find the cause. Unfortunately, exhaustive logging will affect performance, so it's important to keep that in mind when considering which is more valuable.

Here's a counterintuitive idea, begin Integration Testing from the get-go together with Unit Testing. I've mentioned several times how Integration Testing is the second level and done after Unit Testing, but with Agile development, work is done iteratively and with great flexibility. This means that there is so reason to save I&T until later, it can be done alongside the product development, testing each module as it gets finished.

(Honig, 2015)

5 Method

This chapter will describe the practical work done for the company. First, the Unit tests for the WMAP CANopen source code, written in the C programming language. Second, the integration tests for CANopen communication between the new ESM-30 module and the UNIC communication module COM-10, written in Java. Lastly, I will quickly go over some points regarding the writing of the thesis itself.

5.1 WMAP CANopen Unit Tests

Development on Wärtsilä Modular Automation Platform WMAP, is done in an IDE, Integrated Development Environment, called Eclipse. Opening up WMAP in Eclipse greets the user with a dizzying amount of folders and files, making it easy to get lost unless you know exactly where you need to go. Thankfully, everything is sorted and structured to make navigation easier. To find the CANopen source code, we need to go a few levels down the subdirectories of the "code" folder. The source code itself is split up in a couple different files, the one we'll be focusing on is named *cano.base.c*. This is the biggest of the bunch consisting of 3778 lines of code.

The new CANopen stack was implemented into WMAP in a "proof of concept" development branch, which means that the product version of WMAP and even the main development branch remains untouched. When the PoC branch is finished and everything is working correctly, it is then merged into the master branch which is continuously integration tested.

For my development of the Unit tests, I created yet another separate branch split from the PoC branch. This branch is stored locally on my machine which means that I can edit any code and make any other changes I want without it affecting anyone else. I started by generating an automated test case file, this is done automatically by a python script, what it does is create a new file with skeleton test cases for each function in the source file and some setup for when running the tests. These skeleton test cases are just empty functions, but they save a lot of time since the developer doesn't need to write the name of every function manually.

We now got a new file where we can write the tests, a stub file was also created where we can put our fake functions that can be called in place of the real functions when we are testing a function that calls other functions in turn. These new additions need to be included in the Makefile to compile correctly, at this step I got stuck for a while until a teammate figured out the problem. The Makefile is a file that contains information about what actions are performed during compiling and dependencies for a the files in the program.

I had only done some basic coding in C before, so I decided to start with the shortest and simplest functions and then work my way up, improving my skills as I go. After some struggling, a lot of learning and valuable help from my co-workers, by the end 40 test cases had been written, so understandably I will not be documenting every single one, but I will walk through some examples.

5.1.1 Function to be tested

```
static u32 calculateSizeOfTypeInfosVector(const TypeinfosCfg* pInfo)
{
    u32 size = 0;
    size = sizeof(TypeInfosCfg) - 4          //TypeInfosCfg without StructTypes vector
            + ((pInfo->NumOf + 3) / 4) * 4  //StructType of u8 vector
            + ((pInfo-> +3) / 4) * 4;       //actual vector in CANODataMappings


    return size;
}
```

This is a seemingly straightforward function. It takes as parameter a pointer, named “pInfo”, to a TypeInfosCfg struct, calculates the size, in bytes, of a variable by adding together values contained in the struct. A pointer is a memory address to a variable, a struct is a custom-made collection of values of different types.

A variable named “size” is declared with the data type unsigned 32. On line 3182 we then calculate the size of a struct of the type TypeInfosCfg, except, this struct also contains another struct, named “StructType”, which we remove by subtracting its size, 4 bytes.

```
            + ((pInfo->NumOf + 3) / 4) * 4  //StructType of u8 vector
```

On line 3183 we take the value of the member NumOf, then we add 3 and divide/multiply by 4 to round off the number to a multiple of 4 above the original number. The reason for why this is done will be explained a bit later when I'm describing the Unit test for this function.

```
            + ((pInfo-> +3) / 4) * 4;          //actual vector in CANODataMappings

    return size;
```

The next line is just the same principle as the one above except with the "TotalSize" variable instead. Finally, we return the "Size" variable, containing the sum of these calculations.

5.1.2 Writing the test

Creating a Unit test for this function is relatively easy, we only have three variables that are relevant: the struct TypeInfosCfg and two of its members: NumOf and TotalSize. So to test that this function works, we give these members whichever values we want them to have, then we manually calculate what the function should return and compare that to the value actually returned by the function. The value we expect to be returned is the size of the TypeInfosCfg struct, plus the value of the NumOf variable and the value of the TotalSize variable, the two latter being rounded up to the closest multiple of four.

```
static void Test_CalculateSizeOfTypeInfosVector(int runMode)
{
    START("Test_CalculateSizeOfTypeInfosVector.");

    u32 ret = 0;

    localTypeInfosCfg.NumOf = 1;
    localTypeInfosCfg.TotalSize = 15;
    ret = CalculateSizeOfTypeInfosVector(&localTypeInfosCfg);

    ASSERT_U_INT(ret, 28);      //8 + 4 + 16
}
```

First, we create a variable "ret" for the return value, then we give some values to the struct members TotalSize and NumOf. Next, we call the original function and pass the memory address to the struct we have created, putting the return value into the "ret" variable. To assert whether the function works as intended, we just go through the function manually,

calculating what we expect the function to return and compare it to the value in our "ret" variable.

```
/** Object Dictionary (inner struct) TypeInfosCfg **/
typedef struct
{
    u32         TotalSize;      /*!< TotalSize */
    u16         NumOf;          /*!< (0..64) */
    u8          Dummy1;         /*!< Dummy1 */
    u8          Dummy2;         /*!< Dummy2 */
    u8          StructType[1];  /*!< StructType */
} TypeInfosCfg;
```

The code above shows the TypeInfosCfg struct. On a 32-bit processor, memory is accessed in chunks of four bytes per cpu cycle called a *word*. To make the memory access as fast as possible, we want to align the members along these four byte words. When these members are declared, they are stored in memory, the u32 member takes up four bytes of memory, making the second member conveniently aligned with the next word. The u16 datatype uses two bytes of memory, ending in the middle of a word, that's why two bytes of memory padding is added to align the StructType member with the next word of memory. Seeing that the StructType struct is declared as u8, you might think that we should only need to subtract one byte, but the way the memory works means that there is padding created after the StructType to fill out three more bytes. Below picture illustrates why it is so.

These three darker u8 allocations are created by the compiler as padding to fill out the memory because the stored variables need to line up without leaving empty parts in the memory. So even though that last member is one byte, four bytes will be occupied in memory.

Figure 5: Memory allocation illustration

The reason we add three and divide/multiply with four is also for memory allocation purposes like the explanation above, unsigned data types don't use decimal numbers so when dividing, the result will be rounded down to a whole number. For example, if we have the number 13: ((13 +3) /4) *4 = 16. If we have the numbers 14, 15 or 16, the result will still be 16. If we don't add three at the start, the result would always round down to a

multiple of four below the original number, except if the original number already is a multiple of four, since the remainder is always discarded.

Now, back to calculating the size. The TypeInfosCfg consists of one u32, one u16 and two u8 variables after subtracting 4 bytes for StructType. To calculate the size we simply add 4 bytes for the u32, 2 bytes for the u16 and one byte for each u8. Next we add the value of NumOf, which we have declared as 1, add 3 divided by 4, and multiply by 4 to get the final value. Since 1 + 3 = 4, the rest is unnecessary in this case, which is always true when the original value is any multiple of four + 1. For the next line we just do the same thing with the TotalSize, in this case being 15.

15 + 3 = 18
18 / 4 = 4.5 Ignoring the remainder we get 4.
4 * 4 = 16 Now we have a multiple of 4 above the original number.

Then we just call the function, saving the returned value into "ret" and compare it to what we have manually calculated that we should get. If the values match up, the test passes, otherwise the test fails and the user is alerted via the logging interface.

5.1.3 Stub function example

As is demonstrated in the example above, even a seemingly simple function consisting of only a few lines can get very complicated. Some of the later test-functions I wrote, upwards of 130 lines long, ended up being more straightforward to write than this one, but here I got to bring up some important concepts about memory and structs in C. Below I will demonstrate how a stub function is used with an example.

```
static u8 USMPreopReqReceived(u8* semErr)
{
    u8 ret = FALSE;
    UNIC_OS_RET err;

    *semErr = FALSE;

    // Wait notification from USM
    UNIC_OSSemPend(gCANOTriggerNetworkInit, CANO_USM_WAIT_TIMEOUT_MS * 1000U, &err);

    if (err == UNIC_OS_NO_ERR)
    {
        ret = TRUE;
    }
    else if (err != UNIC_OS_TMEOUT)
    {
        *semErr = TRUE;
```

```
    }
    else
    {
        //empty
    }

    return ret;
}
```

Above we can see the function that will be tested, USMPreopReqReceived. Now the thing of interest in this function is that it calls yet another function before returning a value. On the ninth line, after the comment, the function UNIC_OSSemPend is being called. The test we are writing is for the main function, a separate test will be written later for testing the second function. Since we want unit tests to be isolated, this means we need to remove any uncertainty about the second function affecting our results when testing the main function. If we are testing the main function but the second one is broken, our tests will fail and show that the main function is broken, which is not true. Therefore, we create a stub function that gets called instead of the real function.

```
void UNIC_OSSemPend(UNIC_OS_SEM* pSem, u16 timeout, UNIC_OS_RET* pError)
{
    static int count = 0;

    if(count == 0){
        *pError = UNIC_OS_NO_ERR;
        count = 1;
    }
        else if(count == 1)
    {
        *pError = UNIC_OS_ERR_EVENT_NULL;
        count = 2;
    }
        else
    {
        *pError = UNIC_OS_TIMEOUT;
        count = 0;
    }
}
```

The code snippet above is the stub function taking the place of the real function. The actual UNIC_OSSemPend function is a lot more complex but this stub function simply pretends to be the real function and just follows the set of instructions we want it to. It is declared as a “void” function which means that it does not return a value back to the main function, but instead it changes the value at a certain memory address.

The way it works is that it takes in all the parameters that the usual function takes, but then ignores them and simply follows the instructions we have given it. Here we declare the variable “count” as zero, then by using an if-statement we set the value at the memory address of the “Error” variable to different values depending on whether the count variable is 0, 1 or 2. Whenever this stub function is called, it changes the value of “count”, so that it sets another error when it is called. When it has been called three times, the “count” variable goes back to zero and we start over. It is done this way because we have three different scenarios that we want to test by setting the “Error” variable to different values.

The original function declares an “err” variable and passes its memory address “&err” along when calling the stub function. The stub function takes this address but renames it to “pError”, it then sets the value at that address to one of three alternatives depending on which if-statement is triggered.

```
static void Test_USMPreopReqReceived(int runMode)
{
    START("Test_USMPreopReqReceived");

    u8 semErr = FALSE;
    u8 ret = USMPreopReqReceived(&semErr);
    ASSERT_U_INT(semErr, FALSE);
    ASSERT_U_INT(ret, TRUE);

    ret = USMPreopReqReceived(&semErr);
    ASSERT_U_INT(semErr, TRUE);
    ASSERT_U_INT(ret, FALSE);

    ret = USMPreopReqReceived(&semErr);
    ASSERT_U_INT(ret, FALSE);
    ASSERT_U_INT(semErr, FALSE);
}
```

At last, we arrive at the test function itself, in this test case the stub function does a lot of the work, leaving the main test function to simple assert whether the values have been set correctly. All we need to do is pass the memory address to the function we are testing, check that it returns the right value and also that the value at the location of the memory address has changed as expected. We call the function three times, which means the stub function is also called three times, using its three different if-statements once each. After each function call, the test asserts that the values have changed accordingly.

5.2 UNIC – ESM30 CANopen communication Integration Tests

After having written a sufficient amount of unit tests for the source code, it was time to proceed to the next level of testing. Integration tests already exist for other modules in the system, but since the ESM-30 is still in development, no tests regarding that module have been made yet. These already existing tests can be used as a base when creating the new ones, since all modules need to be tested similarly.

My task was regarding validating the new CANopen stack, this means I only need to focus on the CANopen communication integration tests. The first thing I did was to study some Java and look at the existing tests to figure out what they do and how they work. Conveniently, there are tests written involving a WAGO CANopen module, these tests could be used as a base for the ESM-30 tests. Then a custom Engine Package containing the ESM-30 needed to be created, to be able to test a system with the module present. The latest "platform release test" package was used as a base and the ESM-30 was added as a CANopen module.

I will go through one of the ESM-30 tests as an example, to walk through the code. The making of these tests quickly started following a clear structure: study how the existing test works, reapply that knowledge and make an ESM-30 specific version, get the test to pass and check that it fails when it's supposed to fail and passes when it's supposed to pass. In the CANopen theory Chapter I brought up the different communications that are used in the protocol: NMT, SYNC, TIME, EMCY, PDO, SDO and Node Monitoring (Heartbeat). Out of these, tests were written for all but SYNC and TIME. In the setup test below, both NMT and Heartbeat are tested.

5.2.1 ESM-30 Setup test

Below I will go through the ESM-30 setup test. First there are some steps for the setup of the test, like engaging CAN reception to be able to read the CAN data and rebooting the network management master. The ATF-system is doing a lot of stuff in the background that we do not need to know about for the purpose of this thesis, all relevant lines of code will be explained. For reading the CAN traffic I am using CANrunner, a free CAN diagnostic tool created by Wapice Ltd.

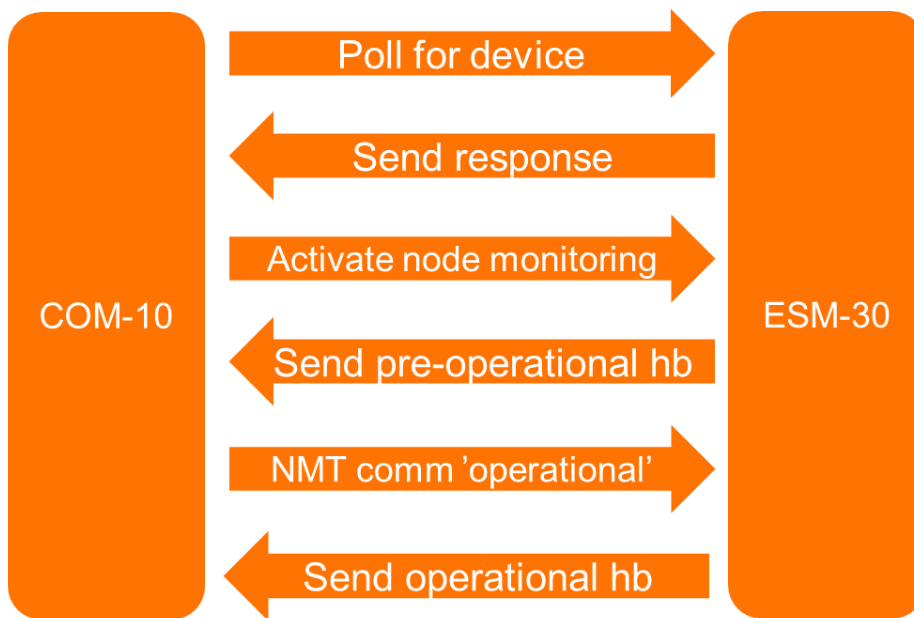

Figure 6: ESM-30 setup sequence

The picture above demonstrates the communication between UNIC and the ESM-30 during setup, the communication frames sent from the ESM-30 to UNIC will be manually created in the test.

```
//Wait for the NMT master to start polling for the device type
CANopenUtils.waitForDeviceTypeQuery(can, ESM30_1.canNodeID, 60, TimeUnit.SECONDS);

//Send response
CANopenUtils.sendCANOMessage(can, CANopenUtils.CANopenSDOResponseID + ESM30_1.canNodeID,
                                         new CANData(0x43, 0, 0x10x, 0, 0, 0, 0, 0));
```

This first line simply waits for the Network Management master to boot up and send a broadcast on the CAN-bus that requests an answer from the ESM-30. The master in this case being a virtual COM-10, when it boots up, it reads the configuration file that we have created in UNITool and starts polling for every device that is supposed to be part of the network. Here I am using the modified “Platform Release Test” package where I have manually added the ESM-30 to the configuration.

You can see the NMT master sending boot up message and going into pre-operational mode first. On line 5447, you can see the Receive SDO being sent from the master to the device with Node ID 81. In the CANopen theory chapter I explained the COB-ID, consisting of a 4-bit function code + the 7-bit Node ID, here we can see that in action. In the “CAN ID”

column you can see the COB-ID “651”, this number is in hexadecimal. The ESM-30’s Node ID is 81, in hexadecimal that is 51_h. By removing the Node ID from the COB-ID we get the function code $651_h – 51_h = 600_h$. 600_h in binary is 110 0000 0000, remove seven digits from the right for the Node ID and we get 1100. As you can see from the CANopen function code index on page 12, function code 1100_b is the code for Receive SDO. If you do the same with the next line: $5D1_h – 51_h = 580_h$, which in binary is 101 1000 0000, remove seven rightmost digits and we get 1011_b which according to the table on page 12 is the Transmit SDO function code.

This Transmit SDO on line 5448 is the response sent by the second line of code from the screenshot above and confirms to the NMT master that the ESM-30 is indeed connected to the network.

Ch	Counter	T/R	Time	Flags	CAN ID	Len	Data	Node ID	Message Type
6	5442	RX	0		0x00000701	1	0x00	1	NMT Error Control (Heartbeat: Boot Up)
6	5443	RX	1003		0x00000701	1	0x7f	1	NMT Error Control (Heartbeat: Pre-Operational)
6	5444	RX	1992		0x00000602	8	0x40 00 10 00 00 00 00 00	2	Receive SDO
6	5445	RX	2007		0x00000701	1	0x7f	1	NMT Error Control (Heartbeat: Pre-Operational)
6	5446	RX	2196		0x00000602	8	0x80 00 10 00 00 00 04 05	2	Receive SDO
6	5447	RX	2201		0x00000651	8	0x40 00 10 00 00 00 00 00	81	Receive SDO
6	5448	RX	2251		0x000005d1	8	0x43 00 10 00 00 00 00 00	81	Transmit SDO

Figure 7: Boot up and poll for device CANopen frames

After having replied to the request from the NMT master, we wait for the master to send a Heartbeat time write. Upon receipt of which we then send a response.

```
//Wait for HB time write
CANopenUtils.waitForHeartbeatTimeSetting/can, ESM30_1.canNodeId,
                                             GENERAL_MAX_WAIT_TIME_S, TimeUnit.SECONDS);

//Send response
CANopenUtils.sendCANOMessage(can, CANopenUtils.CANopenSDOResponseID + ESM30_1.canNodeID,
                                             new CANData(0x60, 0x17, 0x10, 0, 0, 0, 0, 0));
```

Below are respective frames as seen in CANrunner. Frame 5451 is NMT masters receive SDO sent to ESM-30, frame 5452 is the response frame sending the same data bytes you can see in the test code.

Ch	Counter	T/R	Time	Flags	CAN ID	Len	Data	Node ID	Message Type
6	5451	RX	2473		0x00000651	8	0x2b 17 10 00 de 03 00 00	81	Receive SDO
6	5452	RX	2519		0x000005d1	8	0x60 17 10 00 00 00 00 00	81	Transmit SDO

Figure 8: Heartbeat time write and acknowledgement CANopen frames

Then we wait for the NMT master to send the NMT-command "start remote node" for the ESM-30.

```
//Wait for the 'start remote node' command
CANopenUtils.waitForCANOMessage(can, ESM30_1.canNodeId, 0x0, 2,
            new CANData(0x01, ESM30_1.canNodeId), null, releaseTest.getOperationalTimeoutSecs(),
            TimeUnit.SECONDS, "Did not receive 'start remote node' CANopen request",
            CANopenUtils.CANopenState.CANopen_PREOP, true);

//Send operational HB
CANopenUtils.sendCANOMessage(can, CANopenUtils.CANopenHeartbeatID + ESM30_1.canNodeId,
                                new CANData(CANopenUtils.CANopenState.CANopen_OPER.state()));

waitForNmtMasterOperational();
can.stopReception();

EDLMessage expectedMessage = new EDLMessage()
    .setModule(nmtMaster)
    .setLogId(CANO_NODE_STATE_CHANGE_LOG_ID)
    .addParameter(new EDLParameter("Node ID", Integer.toString(EDLMessage.canNodeId)))
    .addParameter(new EDLParameter("New node state", "Operational"));

waitForExpectedEdlMessage(expectedMessage);
```

We are looking for a frame with data length "2", where the first data byte is 0x01 and the second data byte is the target module's Node ID. NMT commands have function code 0000 and a 0 also as the Node ID, which means it is a broadcast message for all nodes to read. If the second data byte is also 0, then all nodes in the network must perform the command. The relevant frame can be seen on line 5484.

Lastly, we send an "operational heartbeat" message, this heartbeat message simply tells the NMT master that the node is operational. The heartbeat frame has the function code 700_h i.e., 1110_b and holds a single byte of data indicating the node's operational state which can be "Boot Up: 0x00", "Stopped: 0x04", "Operational: 0x05" or "Pre-operational: 0x7f".

Ch	Counter	T/R	Time	Flags	CAN ID	Len	Data	Node ID	Message Type
6	5482	RX	4201		0x00000558	8	0x00 00 00 00 00 00 00 00	88	RxPDO 4
6	5483	RX	4201		0x00000559	8	0x00 00 00 00 00 00 00 00	89	RxPDO 4
6	5484	RX	4203		0x00000000	2	0x01 51	0	NMT Node Control
6	5485	RX	4210	X	0x18eeffea	8	0x00 00 00 00 00 00 00 40	0	Unknown Frame
6	5486	RX	4260		0x00000751	1	0x05	81	NMT Error Control (Heartbeat: Operational)

Figure 9: Operational Heartbeat CANopen frame

After that the test waits for the NMT master to send its own operational heartbeat and then stops the CAN reception.

We have now successfully simulated the CANopen communication from the setup sequence of the ESM-30 module and verified that UNIC understands and responds correctly.

5.2.2 Value mapping and validation

Transmit PDO, Receive PDO, SDO and EMCY are tested similarly to each other, by sending a message of said type and making sure that the value is received correctly. In the case of the tPDO test, we send a PDO message from the ESM-30, containing a set of data values which upon receival are mapped to dedicated DC-codes in the UNIC module.

When creating the Engine Package containing the ESM-30, we map the different PDO values to different DC-codes. Then when a PDO is received, the UNIC module saves the value to the corresponding code. The ESM-30 has 5 PDOs but the code snippet you see below is for the testing of PDO 1.

```
/**
 * Test verifies functionality of UNIC module
 * receiving PDO from ESM-30 and mapping to DC
 */
@Test
public void testESM30TxPDO(){
    can.startReception();
    waitForNmtMasterOperational();
    can.stopReception();

    byte[] expectedValues = new byte[]{0x12, 0x34, 0x56, 0x78, (byte) 0x91, 0x23, 0x45, 0x67};

    final int TPDO_TEST_COUNT = 10;
    for (int count = 0; count < TPDO_TEST_COUNT; count ++) {
       //Send RxPDO, generate a CANopen frame with COB-ID 0x1D1 and send 8 bytes of data.
       for (int i = 0; 1 < expectedValues.length; i++){
           expectedValues[i]++;
       }
       ATFUtils.startStep("Round one: Generate a PDO1 message");

       CANopenUtils.sendCANOMessage(can, CANopenUtils.CANopenTxPDO1ID + ESM30_1.canNodeId,
                                       new CANData(expectedValues));
```

```
        //Wait for the first DC value to update.
        waitForDcValueUpdate(ESM30_1.MappedDcCodes.PDO1_1, ecpectedValues[0]);

        //Check that values were updated to mapped codes
        assertThat(nmtMaster.getDCCode(ESM30_1.MappedDcCodes.PDO1_1).readValue()).as(
                "Value for ccode mapped to PDO1 index 1 is not
correct").isEqualTo((int)expectedValues[0] & 0xff);
        assertThat(nmtMaster.getDCCode(ESM30_1.MappedDcCodes.PDO1_2).readValue()).as(
                "Value for ccode mapped to PDO1 index 2 is not
correct").isEqualTo((int)expectedValues[1] & 0xff);
        assertThat(nmtMaster.getDCCode(ESM30_1.MappedDcCodes.PDO1_3).readValue()).as(
                "Value for ccode mapped to PDO1 index 3 is not
correct").isEqualTo((int)expectedValues[2] & 0xff);
        assertThat(nmtMaster.getDCCode(ESM30_1.MappedDcCodes.PDO1_4).readValue()).as(
                "Value for ccode mapped to PDO1 index 4 is not
correct").isEqualTo((int)expectedValues[3] & 0xff);
        assertThat(nmtMaster.getDCCode(ESM30_1.MappedDcCodes.PDO1_5).readValue()).as(
                "Value for ccode mapped to PDO1 index 5 is not
correct").isEqualTo((int)expectedValues[4] & 0xff);
        assertThat(nmtMaster.getDCCode(ESM30_1.MappedDcCodes.PDO1_6).readValue()).as(
                "Value for ccode mapped to PDO1 index 6 is not
correct").isEqualTo((int)expectedValues[5] & 0xff);
        assertThat(nmtMaster.getDCCode(ESM30_1.MappedDcCodes.PDO1_7).readValue()).as(
                "Value for ccode mapped to PDO1 index 7 is not
correct").isEqualTo((int)expectedValues[6] & 0xff);

     }
 }
```

We declare the values we will be sending as expectedValues, then we declare the TPDO_TEST_COUNT variable as 10 and create a loop that runs ten times, sending ten PDOs and checking that the DC code has been correctly updated each time. Inside this loop we have also created another loop that just increments every data byte so that it doesn't just send the same PDO ten times in a row.

6 Results

The concrete result is 40 unit tests and 6 integration tests. The suite of unit tests covers necessary functions in the base file of the new WMAP CANopen source code. The Integration tests simulate the four main functionality of CANopen communication between the new ESM-30 module and UNIC, namely PDO, SDO, EMCY and NMT. Together, these make for a solid first two levels of testing, but improvements can still be made.

6.1 Unit Tests

Every Unit tested file in the WMAP source code has its own separate file with the relevant Unit tests. The Unit tests are supposed to be run whenever someone makes a change to the relevant source code, for example refactoring or improvements. After a code change, assuming the functionality of the code remains the same, the tests will still pass. If a change

is made that impacts the functionality, whether accidentally or on purpose, the relevant test will fail and signal the developer. This collection of 40 Unit tests runs in less time than a minute, making it a very fast way of checking that code functionality remains consistent.

The CANopen stack source code consists of several separate files where "cano_base.c" is the biggest one, being 3779 lines long. The target was to get Unit tests written for this file, which was achieved. Not every single function needed to be tested. The picture below shows the eclipse terminal output after running the "cano_base.c" Unit test suite.

```
[WTP] Results [2022-03-28 16:20:46]:
[Canopen]
1 [Canopen external API]
1.1) Test_CANOGetCommDataPtr (PASS)
1.2) Test_CANOGetEmcyDataPtr (PASS)
1.3) Test_CANOManageGlobalCanErrorTS (PASS)
1.4) Test_CANOGetTypeInfoTypesPtr (PASS)
2 [Canopen internal API]
2.1) Test_CANOManageGlobalCanError (PASS)
2.2) Test_init_Library (PASS)
2.3) Test_InitNetworkChannels (PASS)
2.4) Test_SetNetworkState (PASS)
2.5) Test_CalculateSizeOfTypeInfosVector (PASS)
2.6) Test_PortStackLoop (PASS)
2.7) Test_InitCPS (PASS)
2.8) Test_CANOHandleIncomingMessage (PASS)
2.9) Test_SetNewDriverState (PASS)
2.10) Test_CalculateSDOItemSize (PASS)
2.11) Test_MonitorNodeStatuses (PASS)
2.12) Test_GetSysDataMappingPtr (PASS)
2.13) Test_CANOPostInit (PASS)
2.14) Test_CANOInitializeDiagData (PASS)
2.15) Test_CANOUSMStateCheckCB (PASS)
2.16) Test_CalculateRPDOItemSize (PASS)
2.17) Test_CalculateODItemSize (PASS)
2.18) Test_CalculateEMCYItemSize (PASS)
2.19) Test_CANOChEnabled (PASS)
2.20) Test_UpdateCfgLookups (PASS)
2.21) Test_InitSdoPerNode (PASS)
2.22) Test_CalculateTPDOItemSize (PASS)
2.23) Test_CalculateCPSRuntimeRecordSize (PASS)
2.24) Test_RunCycle (PASS)
2.25) Test_CANOInit (PASS)
2.26) Test_InitCanoBase (PASS)
2.27) Test_USMPreopReqReceived (PASS)
2.28) Test_CANOPostInitCPS (PASS)
2.29) Test_USMOpReqReceived (PASS)
2.30) Test_CANOShouldMsgBeInserted (PASS)
2.31) Test_CANOInitializeQueueRt (PASS)
2.32) Test_CalculateIndexMappingDataSize (PASS)
2.33) Test_ClearUnusedParams (PASS)
2.34) Test_CANOUpdateHwChannelStatus (PASS)
2.35) Test_CANOIndicateFault (PASS)
3) Global asserts [id:test_Global] (PASS)
Verdict: PASS (40 tests, 184 asserts)
```

Figure 10: Unit Test suite execution results

6.2 Integration Tests

The UNIC network consists of several different hardware modules, we are focusing on the still in development safety module ESM-30. Six integration tests were written covering the ESM-30's CANopen communication with UNIC. These tests cover setup of the module, transmit PDO, receive PDO, emergency, SDO initialization and PDO supported datatypes. The integration tests are run every day, verifying that no one has pushed a change that breaks the system's functionality.

The continuous integration system downloads the most recent source code from the version control system, builds the software and executes the tests via the Automated Test Framework. Some tests run on actual hardware and some are run on virtual modules, the CANopen integration tests run on virtual since there are no physical CANopen modules connected to the test rack. If a test fails, the concerned people will be informed by an automated e-mail.

Below are log messages when running the ESM-30 setup test, when everything runs successfully, the logger gives simple updates regarding the test.

```
2022-03-28 16:44:56,774 INFO  ATFLogger - === Starting test com.wartsila.wmap.tests.canopen.ESM30_1:testESM30Setup ===
2022-03-28 16:44:56,774 DEBUG ATFLogger - Powering down device COM10_1.
2022-03-28 16:44:56,774 DEBUG ATFLogger - Stopping virtual module COM10_1.
2022-03-28 16:44:59,331 INFO  ATFLogger - Waiting until module COM10_1 (COM10, 0x51) is in state: None. Max wait time 13 s.
2022-03-28 16:44:59,354 DEBUG ATFLogger - COM10_1 Current state: None.
2022-03-28 16:44:59,354 DEBUG ATFLogger - Time waited 22 ms.
2022-03-28 16:44:59,354 DEBUG ATFLogger - Powering up device COM10_1.
2022-03-28 16:44:59,354 DEBUG ATFLogger - Starting virtual module COM10_1.
2022-03-28 16:45:00,433 DEBUG ATFLogger - Waiting for COM10_1 to go pre-operational on the CANopen bus.
2022-03-28 16:45:07,222 DEBUG ATFLogger - Found UNIC2 module in system. Operational timeout: 180 s.
2022-03-28 16:45:08,929 DEBUG ATFLogger - Waiting for COM10_1 to go operational on the CANopen bus.
2022-03-28 16:45:09,734 DEBUG ATFLogger - Waiting for expected EDL messages. Max wait time 65000 milliseconds.
2022-03-28 16:45:10,546 DEBUG ATFLogger - All expected EDL messages received. Time waited 792 milliseconds.
2022-03-28 16:45:10,546 INFO  ATFLogger - === Test com.wartsila.wmap.tests.canopen.ESM30_1:testESM30Setup passed. ===
```

Figure 11: ESM-30 setup test log messages

Below is a picture how it might look when an error is encountered. In this instance, we are running the same ESM-30 setup test but failed to receive the NMT command from UNIC to start the node.

```
2022-03-28 18:46:22,697 DEBUG ATFLogger - Powering up device COM10_1.
2022-03-28 18:46:22,697 DEBUG ATFLogger - Starting virtual module COM10_1.
2022-03-28 18:46:23,763 DEBUG ATFLogger - Waiting for COM10_1 to go pre-operational on the CANopen bus.
2022-03-28 18:46:30,536 DEBUG ATFLogger - Found UNIC2 module in system. Operational timeout: 180 s.

java.lang.AssertionError: [Did not receive 'start remote node' CANOpen request]
Expecting:
 <false>
to be equal to:
 <true>
but was not.

    at com.wartsila.wmap.tests.canopen.CANopenUtils.waitForCANOMessage(CANopenUtils.java:187)
    at com.wartsila.wmap.tests.canopen.ESM30_1.testESM30Setup(ESM30_1.java:110) <16 internal lines>
    at org.testng.SuiteRunnerWorker.runSuite(SuiteRunnerWorker.java:52)
    at org.testng.SuiteRunnerWorker.run(SuiteRunnerWorker.java:84) <4 internal lines>
    at com.intellij.rt.testng.IDEARemoteTestNG.run(IDEARemoteTestNG.java:66)
    at com.intellij.rt.testng.RemoteTestNGStarter.main(RemoteTestNGStarter.java:109)
```

Figure 12: Error message during test run

The picture below shows the results from the IntelliJ terminal when running the ESM-30 Integration test suite. Since the tests are isolated from each other, they can be run in any order.

```
Tests passed: 6 of 6 tests – 5 min 14 sec

Default Suite                              5 min 14 sec
  wmap-auto-tests                          5 min 14 sec
    ESM30_1                                5 min 14 sec
      testESM30EMCY                        1 sec 600 ms
      testESM30PDO_SupportedDatatypes      3 sec 536 ms
      testESM30RxPDO                       735 ms
      testESM30SDOinitialization           11 sec 563 ms
      testESM30Setup                       14 sec 387 ms
      testESM30TxPDO                       1 min 45 sec

===============================================
Default Suite
Total tests run: 6, Failures: 0, Skips: 0
===============================================

Process finished with exit code 0
```

Figure 13: Integration Test suite execution results

Beyond these tests, another valuable result is this document itself, which can function as a CANopen and SW Testing quick guide for readers.

7 Discussion

This thesis was incredibly rewarding for me. When I started writing the Unit tests I had only a very basic understanding of the C language, but with every test I worked on, I stumbled upon a new concept or functionality of the language, learning more things with natural progression. I learned a lot about software testing, why it is needed and how it is done, gaining significant interest in the entire process.

There were some hiccups along the way, at one point during the creation of the integration tests my local virtualization server stopped working and a couple days were spent on trying to get that working again. I was also a bit skeptical about using two new languages so soon after each other but this has taught me to not be afraid of code, it may seem confusing and difficult at first, but figuring out how it works is very rewarding.

7.1 Further development

Further development on the work done in this thesis is relatively straightforward. Unit tests were created for the main chunk of functions in the CANopen stack but there are still some smaller files containing more functions that could use some tests. As described in the Software Testing chapter, SW testing consists of four levels, where this thesis has seen work being done on the first two levels, a natural development for the future is of course proceeding on to the third and fourth levels of testing, namely System Testing and Acceptance Testing.

7.2 Acknowledgements

Finally, I would like to thank some people who have helped me and taught me a lot during these months. All my teammates in Team Spark, especially Peeter who helped me with the unit tests, Tomas who helped with UNITool- and ATF-related stuff, Enn who solved the Makefile problem and of course Gerald, who voluntarily took it upon himself to be my supervisor and mentor me through the whole process, becoming not only someone I respect as a colleague, but also a good friend. Also, my manager Markus, for giving me the opportunity to work in A&C and Jan, my Novia supervisor for helping with the school side of things. And finally, Anders, my study buddy who kept me company during these long days.

8 References

Beck, K., 2019. *Test Desiderata.* [Online]
Available at: https://medium.com/@kentbeck_7670/test-desiderata-94150638a4b3
[Accessed 25 03 2022].

CiA: CAN in Automation, 2022. *CAN: From physical layer to application layer and beyond.* [Online]
Available at: https://www.can-cia.org/can-knowledge/

CiA: CAN in Automation, n.d. *CANopen history.* [Online]
Available at: https://www.can-cia.org/can-knowledge/canopen/canopen-history/

CiA: CAN in Automation, n.d. *Classical Controller Area Network (CAN).* [Online]
Available at: https://www.can-cia.org/can-knowledge/can/classical-can/

CSS Electronics, 2021. *CANopen explained - A simple intro.* [Online]
Available at: https://www.csselectronics.com/pages/canopen-tutorial-simple-intro

Gibb, R., 2019. *What is a Distribued System?.* [Online]
Available at: https://blog.stackpath.com/distributed-system/

Honig, R., 2015. *6 best practices for integration testing with continous integration.* [Online]
Available at: https://techbeacon.com/app-dev-testing/6-best-practices-integration-testing-continuous-integration
[Accessed 26 03 2022].

javaTpoint, n.d. *Difference Between Stubs and Drivers.* [Online]
Available at: https://www.javatpoint.com/stubs-vs-drivers

Kralj, K., 2022. *5 Qualities Your Best Integration Tesets Should Have.* [Online]
Available at: https://methodpoet.com/integration-test-best-practices/
[Accessed 25 03 2022].

Matias Muhonen, Wärtsilä, 2019. *Platform Software Development.* [Online]
Available at: -
[Accessed 23 03 2022].

Parikh, B., 2021. *CAN Protocol. Understanding the controller area network.* [Online]
Available at: https://www.engineersgarage.com/can-protocol-understanding-the-controller-area-network-protocol/

Richards, P. & Microchip Technology, 2002. *A CAN Physical Layer Discussion.* [Online]
Available at: https://ww1.microchip.com/downloads/en/appnotes/00228a.pdf

Software Testing Magazine, 2017. *Unit Testing: Fakes, Mocks and Stubs.* [Online]
Available at: https://www.softwaretestingmagazine.com/knowledge/unit-testing-fakes-mocks-and-stubs/

STM Rajkumar, 2019. *Levels of Testing.* [Online]
Available at: https://www.softwaretestingmaterial.com/levels-of-testing/

STM Rajkumar, 2019. *Unit Testing Guide.* [Online]
Available at: https://www.softwaretestingmaterial.com/unit-testing/

STM Rajkumar, 2020. *Black Box and White Box Testing.* [Online]
Available at: https://www.softwaretestingmaterial.com/black-box-and-white-box-testing/

STM Rajkumar, 2020. *Integration Testing.* [Online]
Available at: https://www.softwaretestingmaterial.com/integration-testing/

STM Rajkumar, 2020. *What is Verification and Validation in Software Testing.* [Online]
Available at: https://www.softwaretestingmaterial.com/verification-and-validation/

STM Rajkumar, 2021. *Software Testing Trends 2021.* [Online]
Available at: https://www.softwaretestingmaterial.com/software-testing-trends/
[Accessed 24 03 2022].

STM Rajkumar, 2021. *What Is Software Testing.* [Online]
Available at: https://www.softwaretestingmaterial.com/software-testing/

techopedia, n.d. *Network Redundancy.* [Online]
Available at: https://www.techopedia.com/definition/29305/network-redundancy

W3Schools, n.d. *Types of Network Protocols and Their Uses.* [Online]
Available at: https://www.w3schools.in/types-of-network-protocols-and-their-uses/

Wärtsilä, 2017. *Wärtsilä UNIC: Engine Control System Overview.* [Online]
Available at: https://www.wartsila.com/docs/default-source/product-files/engines/brochure-o-e-unic-ms.pdf?sfvrsn=7ab39545_4

Wärtsilä, 2022. *About Wartsila.* [Online]
Available at: https://www.wartsila.com/about

Wärtsilä, 2022. *History of Wärtsilä.* [Online]
Available at: https://www.wartsila.com/about/history

Wärtsilä, 2022. *UNIC engine automation system.* [Online]
Available at: https://www.wartsila.com/encyclopedia/term/unic-engine-automation-system