
Master Thesis

# Simulation of a TCU Node on a Virtual CAN Bus

**Johan Viklander**

A thesis submitted in fulfilment of the requirements for the degree of Master of Science in Engineering Physics.

Department of Physics
Umeå University
January 2016

**Simulation of a TCU Node on a Virtual CAN Bus**


**Abstract**

Electrical Control Units (ECUs) communicating on Controller Area Networks (CAN buses) are widely used in vehicle electrical systems. Being able to simulate these circuits and buses in a computer environment is beneficial during the development phase when access to test benches is limited and expensive. Simulations can also give a very detailed view of the entire system which in an ordinary lab setup would be practically impossible.

BAE Systems Örnsköldsvik SIL Lab department works in the simulation environment CANoe supplied by Vector Informatik GmbH. BAE Systems have a simulation model of their central communication circuit. Unlike the less complicated circuits on the bus it cannot be integrated in the CANoe simulation environment. The less complicated nodes are modelled to usable extent but this is not possible with the central communication circuit. This report presents a possible solution to facilitate communication between the simulated ECU and the CANoe simulation environment under certain real-time constraints.

A solution was achieved with a combination of an external program which handled shared memory with callback functions and Vector's Fast Data eXchange protocol (FDX).


**Simulering av en TCU-nod på en virtuell CAN-buss**


**Sammanfattning**

Elektriska styrenheter (ECUs) som kommunicerar på ett Controller Area Network (CAN-buss) används ofta inom elektriska system i fordon. Att ha möjligheten att simulera dessa kretsar och bussar i ett datorsystem är fördelaktigt under utveckling när tillgång till testbänkar är begränsad och användning av dem är kostsamt. Simulering kan också ge en mer detaljerad vy av hela systemet, vilket en vanlig labbuppställning inte kan.

BAE Systems Örnsköldsviks SIL Lab-avdelning arbetar i simuleringsmiljön CANoe som tillhandages av Vector GmbH. BAE Systems har en simuleringsmodell av deras viktigaste kommunikationskrets. Till skillnad ifrån deras mindre avancerade kretsar kan denna ECU inte integreras i simuleringsmiljön i CANoe. Denna rapport beskriver en möjlig lösning för att skapa kommunikation mellan den simulerade ECU:n och CANoe som uppfyller vissa realtidskrav.

Lösningen består av en kombination av externa program som hanterade delat minne med "callback"-funktioner och Vectors Fast Data eXchange protocol (FDX).

# Contents

# 1 Introduction

In recent years a large part of vehicle development has been digitalization and automated control. The demand from consumers, regulations of fuel efficiency and safety and global market pressure has driven this research development. It is common now that a vehicle has several integrated computers and Electrical Control Units (ECUs).

At first most vehicles only had a single ECU but as demand grew more were introduced. The communication went through one wire for each signal. This system was impossible to scale up as the amount of wires grew with the amount of ECUs and signals. This increases manufacturing price and complexity.

Bosch started developing the Controller Area Network (CAN) protocol in 1983 as it became obvious that some kind of reliable serial bit communication was required. It was later released in 1986. Now communication on several CAN buses are common in modern vehicles.

## 1.1 The Need for Simulation

The buses can have upwards of a hundred nodes each with hundreds of signals. The signals are sent very quickly as very often as well. It is not uncommon of signals being sent cyclically as often as every ten milliseconds. It is quite obvious that simulation of these buses are essential in development and testing. The fact that simulations are cheap while giving more detailed view of the system is also an advantage.

## 1.2 BAE Systems Hägglunds and Vector GmbH

BAE Systems Hägglunds is a manufacturer of military vehicles. They produce Combat Vehicle 90 (CV 90) and the all terrain armoured vehicle BvS 10. In the development of theses vehicles they have several stages of testing ranging from pure simulations and Hardware In the Loop (HIL) testing to real lab testing.

After a migration to the new environment, CANoe, supplied by Vector GmbH they are no longer able to simulate the system any more as the main node of their buses, developed by BAE Systems, is incompatible with CANoe. Although it is possible to simulate the node outside of the CANoe environment. The reason for the migration was extended testing features which were not available in the old environtment.

If the node would be able to send and receive information to and from CANoe then a simulation of the network would be possible. Since the CANoe environment is proprietary software outside of BAE Systems control an implementation will have to work around pre-made standards of communication defined by Vector. This fact limits the flexibility of a possible solution to the problem.

This work has been done with a model of BAE Systems' TCU (Traction Control Unit) node but it is not limited to it. Any node of the type that BAE Systems have developed is now compatible. Henceforth in this report the node will be referenced as the TCU.

A solution was achieved by using a combination of shared memory space on the computer and socket communication between the programs. The socket communication was handled by callback functions. The callback functions were executed in response to a new signal being available and their function was to redirect the data to another application through the socket.

# 2 Background

The following chapter is based on the information from Vector's website cited in [1].

## 2.1 Hardware

Before the use of CAN buses, wires were drawn for each signal. This can, for example, be a light on the dash board in a car. This is impractical for vehicles with a lot of information exchange since the cost of the material and the weight of the vehicle will be high, the construction time will be long and the complexity will be high during service of the vehicle. Since modern vehicles can have several hundreds of signals it is obvious how important the CAN technology is.

### 2.1.1 The CAN Bus

The CAN bus is made up of unshielded twisted pair wires. The twisting reduces the resulting magnetic field from the wires. These wires are called CAN high and CAN low. They are usually between 0.34 to 0.60 square millimetres in thickness with less than 60 mΩ resistance. Terminating resistors of 120 Ω prevents reflection in the wires.

The communication is symmetrical signal transmission. The wires each have a voltage which can be high or low. Their difference is interpreted as a dominant or recessive bit. This facilitates communication in binary form.

If several nodes communicate at the same time the bus use AND logic to determine the bit being sent. That is only a single dominant bit is needed for the CAN bus to be dominant. A maximum speed of 1 Mbit/s can be achieved in a high speed CAN bus. Low speed CAN is 125 kbit/s.

Every node on the CAN bus is connected to these wires in a multi master serial connection as can be seen in figure 1.

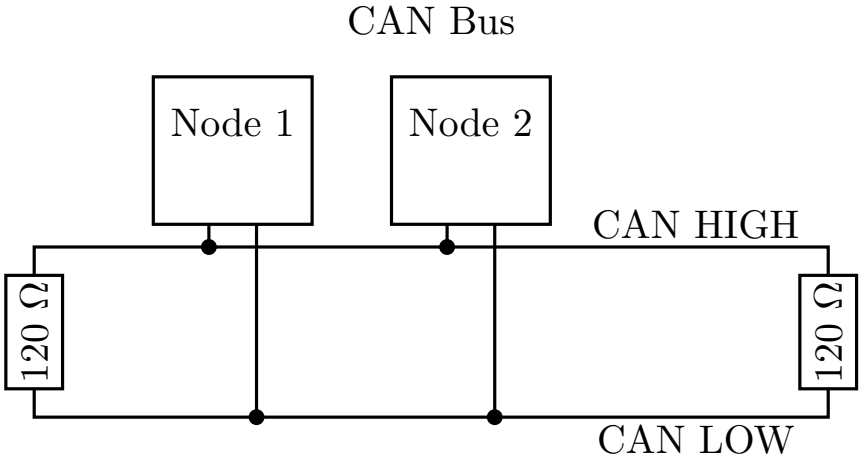


Figure 1: A CAN bus with two nodes.

### 2.1.2 The ECU

Each ECU, called node, in the CAN bus is required to have at least one Central Processing Unit (CPU), one CAN controller and one transceiver. They can both

receive and send messages but not simultaneously. Early CAN nodes consisted usually only of the CAN controller and the CAN transceiver but now they usually have a CPU running an operating system with network management and diagnostics. An illustration of a node is shown in figure 2

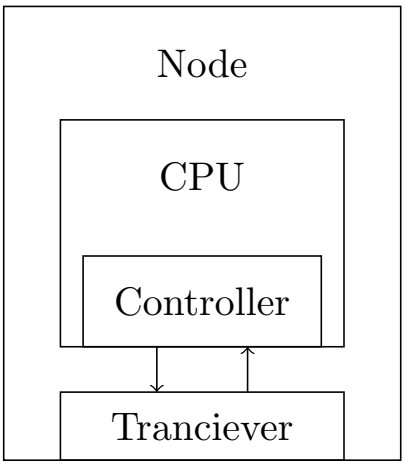


Figure 2: A CAN node.

When receiving messages the transceiver converts the voltage levels from the CAN bus to the levels used by the CAN controller and vice versa when transmitting a message.

The CAN controller stores the bits that are incoming from the CAN bus and sends complete messages to the CPU by an interrupt. When transmitting a message the CAN controller sends the bits of the message serially to the transceiver. There are both integrated and standalone CAN controllers. CAN controllers integrated in the CPU are faster but less flexible than a standalone version.

The CPU processes the message and decides what commands (if any) to execute in response to received message.

The CPU and the transceiver follow the CAN communication protocol when reading data from the bus and from the resulting CAN messages.

## 2.2 The CAN Communication Protocol

Information on the CAN bus is transmitted in frames, often called messages. There are four kinds of frames. The most common one is the data frame which function is to transmit signal data. The remote frame requests signal data from other nodes and is less common than the data frame. If any error occurs then an error frame is sent and if some kind of overload on the bus is detected then an overload frame is sent.

The communication works by letting every node monitor the bus and react to any event that is relevant to the node. The protocol is therefore event driven. Data is not sent unless asked for by another node or is a predefined cyclic behaviour of the node.

### 2.2.1 The Data Frame

The data frame is the most common frame sent in a CAN bus. It transports signal data to the rest of the network. It is either sent cyclically or from a request from another node. The frame is 52 bits or 72 bits long depending on if it is in standard

format or extended format. The extended format allocates more bits for the frame identification field. There are seven main fields of information that make up a data frame, identification (ID), remote transmission request (RTR), identifier extension bit (IDE), data length code (DLC), data bytes (DATA), cycle redundancy check (CRC) and acknowledge field (ACK). The full structure of the standard CAN frame and the extended CAN frame can be seen in appendix A and appendix B respectively with tables explaining each range of bits.

The ID identifies the frame. The nodes in the network knows which frames are relevant to them and will stop processing a message when it has an ID not defined in its database. It also identifies what signals are contained in the data field. There are two kinds of ID, standard ID and extended ID. Their sizes are 11 bits and 29 bits respectively. The extended ID is divided into a base ID which is 11 bits and corresponds to the normal standard ID field together with the extended ID field which is 18 bits. The standard ID field is followed by the RTR bit and IDE bit. The IDE bit indicates that the frame is of extended format if it is recessive. This tells the node that what follows the IDE bit is the extended ID field.

The RTR bit determines if the frame is a data frame or remote request frame. Remote request frames are frames that request signal data from other nodes. Usually data frames are sent cyclically but remote request frames enables data transmission almost on the fly.

The DLC indicates how much data bytes are contained in the frame. The node needs to know how many bytes it should read since there is no effective alternative way for the node to know what is data and what is other parts of the frame. The data is sent in a discrete amount of bytes. The node knows what signals each bit in the data corresponds to since it has identified the corresponding signals by referencing the frame ID to its database of frames and their data content.

The CRC field is a checksum which is constructed by the sender node from the part of the message leading up to the CRC field. This enables the receiver node to check instantly if the message has been corrupted during the sending process. If it has been corrupted then the ACK bit is sent as recessive by the receivers.

The ACK bit is written by the receiving nodes and overrides the senders recessive ACK bit if at least one node considered it a valid frame. Therefore the sender node cannot know at first if all nodes has received the message correctly. If the ACK bit is written recessively then the sending node stops sending the frame and sends an error flag in the bits after the ACK delimiter bit. Receiving nodes which did not send a dominant ACK bit also sends an error flag to ensure consistency if there was other nodes which sent a dominant ACK bit. At this point the sender can determine if every node got the message correctly and act upon that information.

### 2.2.2 The Remote Request Frame

The remote request frame is almost identical to the data frame. The difference is that it has no data field. A recessive RTR bit indicates the presence of a remote request frame. In practice they do not need to be defined as a separate frame in the node since it will have the same ID as the corresponding data frame. Only the separate behaviour in the node needs to be defined for remote request frames. It is possible that several frames can be sent between the request frame and the resulting data frame if there are other frames with higher priority being sent at the same time. Therefore request

of data should not be considered instant.

### 2.2.3 The Error Frame

The error frame consists of two fields, the error flag which is six to twelve dominant or recessive bits followed by the error delimiter which is 8 recessive bits. Recessive bits in the first field indicates that it is a passive error and dominant bits indicate active error.

### 2.2.4 The Overload Frame

The overload frame is similar to the error frame in that it contains two fields and both are a flag field and delimiter field. The bits are the same as the ones of a active error flag. When transmitted the other nodes identify the error and themselves sends the overload frame. Identification of an overload can either be an internal condition in the node or detection of a dominant bit between frames.

### 2.2.5 Error handling

There are five logical errors which can happen during a transmission of a frame. Bit monitoring and ACK check is made by the sender node while the receivers perform the format check, stuffing check and the CRC check.

Bit monitoring is checking that the value sent from the sender node is the same value broadcast by the CAN bus. The ACK slot and the ID field is not included in these checks. The ACK check fails if the ACK bit is returned from the bus as recessive. In that case no node has overwritten it as dominant indicating that no node received the frame successfully. If any of these errors occur an error flag is sent by the sender node.

Stuffing check, format check and CRC check is the receiver node's work. Stuff checking fails if the receiving node registers six homogeneous bits in a row. Any frame with five homogeneous bits in a row must be stuffed with a inverse bit after which the frame information is resumed. This is done for synchronization purposes. Format check is done to ensure the frame follows the basic layout of a CAN frame, mainly by checking that the delimiter bits have the correct values and that there are no dominant bits within the EOF. A CRC checksum is made of the frame as soon as the data field is broadcast to the receivers. When the CRC field is broadcast a comparison is made and if it differs the ACK bit is sent as recessive and an error flag is sent to the bus by the receiving node.

There is error tracking in the CAN protocol to ensure that the CAN bus does not get blocked. Each node starts in a active state. This means that when they detect an error they send an active (dominant) error flag. If a node detects a large amount of errors its state changes to passive. This means that when they detect an error they now send a passive (recessive) error flag. This means that their error flag will be overwritten by other nodes on the CAN bus. If the amount of errors on a node increases even more then it is suspended.

# 3 CAN Bus Simulation

The ability to simulate a CAN bus and its nodes is very beneficial. In the early development stages of a new CAN system it would be very impractical to have to implement it to test it. It would also be very expensive as well, besides time consuming.

In testing of the CAN system simulation is required to simulate unusual condition in which the vehicle has some predefined behaviour which relies on the CAN system. This could for example be behaviour in collisions or engine failure, both of which would be very expensive conditions to create in real life.

There is also more information to gather from a simulation than it is from an experiment. For example when searching for errors a simulation will probably generate logs of every single frame being sent on the bus and be able to pin point exactly where the problem occurred.

As stated before, simulation is very cheap when compared to the alternative. Generally real testing should only be a last resort as long as simulation can achieve the same result.

When doing simulations it is ideal to have a bus which can be simulated in real-time. Therefore one must consider what computer equipment is used. If it is a pure simulation without hardware nodes involved then real-time is not a requirement but as soon as real hardware nodes are used then real-time is very important otherwise the hardware might execute time out errors. This mode of operation is called hardware in the loop (HIL).

There are several different CAN simulation software and one can always create one's own. One must take care to ensure that all software is compatible with each other when mixing different software. Otherwise some kind of coupling between the software has to be made to ensure compatibility between the simulation environments. This is a common problem when migrating to new simulation environments. CAN has several different standards as well. So care has to be taken such that all simulation software has support for the needed standard.

BAE Systems has several requirements for their simulation environment. The simulation environment has to be in real-time, therefore an effective coupling mechanism is needed between different programs. Preferably this mechanism should be purely event driven so that events are processed as soon as possible. Polling algorithms would be too performance intense.

The different applications should be easy to work in and should easily integrate into each other. This means that the user of the software should need to do as little work as possible before initiating a simulation.

BAE Systems has recently migrated from a simulation environment which they developed and maintained themselves to the CANoe simulation environment. Most of the nodes needed in the simulation can be modelled in CANoe and MATLAB but the main node is still only compatible for the previous environment. Therefore a coupling algorithm needs to be created between the node and the CANoe environment so they can exchange information during run time.

# 4 Pre Existing Software

Several important software tools were already available to use at BAE Systems. These programs all need different configuration files which are created by Python scripts. The files and scripts are presented in the next chapter.

## 4.1 Universal Configurator

Universal Configurator (UC) is a database program which defines the information exchange in the vehicle. It is divided into three parts, SCDR (unknown abbreviation), simulation and test benches.

SCDR has all the information which will be present in the actual vehicle. It defines both the hardware and software aspect of the CAN buses present in the vehicle. For the actual hardware it defines how each node should be connected to the buses, what I/O ports it uses on the physical node and so on.

The program also tracks which kind of information is exchanged in the network. For the CAN buses every frame and every signal contained in the frames is defined and also how each signal value should be interpreted. Many signals are only a variable in a function which produces the quantity of interest. Most formulae are in linear form with a factor and an offset. The program also defines what should be logged and what error codes there are to report problems in the networks.

The simulation branch defines the simulation exclusive signals needed for the system. An example is the signal "TCU_ON" which indicates if the TCU node is turned on or not. It is used start and stop the node in simulation environments.

The test benches branch handles data required to use the node at the test benches. It has the file names of several different configuration files, which CAN buses are physically present and which IO signal each physical IO controller corresponds to. The test benches module also connects to external Python scripts which parses the database and generates different configuration files such as database files in XML format or CAPL code files used by CANoe.

## 4.2 CANoe

CANoe, produced by Vector GmbH and released in 1996, is a CAN bus simulation tool for ECU development. It supports several different bus types. Nodes on the buses can be imported from MATLAB Simulink[2][3] library files or be defined in CAPL code. CAPL code is Vector's own interactive, event based, scripting language. It is very similar to C in syntax.

Each node can also have access to a library file written in C with functions that the CAPL code can access. Although there are limitations on what kind of functions and what kind of arguments in the functions the library file can have. Only the very basic data types can be passed from the CAPL code to the library functions and back. The function cannot use any calls to the operating system either and dynamic memory allocation is discouraged.

Vector also has a standard for socket transmission of data. It is called Fast Data eXchange (FDX). It facilitates data exchange of both frames and IO signals.

### 4.2.1 CANoe Testing scripts

CANoe has a testing interface[4] where the user creates scripts which are designed to simulate different use of the nodes and checks that it behaves as the real node should. An example of a small test could be closing the ramp behind the tank and checking that the node has registered that the ramp is closed afterwards.

BAE systems has a wide variety of tests which they use during the HIL simulation. These tests will be the basis on which the proposed solution will be judged.

## 4.3 DataDistributed

DataDristribution (DD) is a library of functions which facilitates shared memory between applications on a computer. The use of shared memory is made through semaphores. DD is transparent source code which is used at BAE Systems. It creates connections between applications where each connection is a memory space on the computer. The memory space could either be for a CAN frame or an IO signal. The former is 20 bytes in size and the latter is 4 bytes in size. The structure of the DD CAN frame can be seen in appendix C. It also has built in error detection.

The initiating program creates a DD realm. A DD realm is an environment where connections and clients can be stored. The initiating program will usually also create a DD client from which it can create the connections. A DD client is an object which can create connections, subscribe to connections and change the values of connections. Usually each program that is somehow connected to DD controls one client.

When a client subscribes to a connection it links the subscription to a callback function. This callback function can be any type of valid C function and its purpose is to be executed for the subscribed client when the value of the connection changes. The only restriction is that it cannot return anything and the arguments are set beforehand. Fortunately at subscription time it is possible to pass a void pointer token to the callback function. This enables the callback function to have a dynamic behaviour even when it is static in every other respect.

The creator of the realm can choose whether to use polling of the connections or not. If polling is used then the callback functions will only be executed after a client has requested a check on all connections. Not using polling lets the callback functions be executed as soon as a connection value changes.

## 4.4 The TCU and Rubus

The central node in these CAN buses, the TCU node, is executed under the simulated real-time operating system Rubus. The Rubus simulation is developed by Arcticus Systems. In the Windows environment it is simulated under best effort real-time. Rubus is a real-time operating system which means that it has to be predictable in the amount of time it uses to complete certain instructions. Real-time operating systems are common where predictability in response time is critical, for example medical equipment need real-time operating systems. Non real-time operating systems, such as those used by personal computers are not predictable in response time

Rubus also handles shared memory. It creates a DD realm and connects to it as a DD client. It then creates shared memory spaces for all signals and buses contained in the dd.xml file and attaches a callback function to them so that the TCU gets updated as soon as new data is available.

# 5 Proposed Solution

As stated earlier, the TCU node with its Rubus operating system is separate from the CANoe simulation environment. Therefore a coupling scheme had to be created between them. Since the broadcast frequency of the system is so high the solution would need to have event based characteristics. This is to ensure that unlike a polling solution, where a program reads the messages in intervals, this solution will not miss a message if they arrive very close to each other in time.

The event based behaviour can be achieved by using the program DD in conjunction with the FDX-protocol[5]. In this solution the information flow from CANoe will be handled by DD and the information flow to CANoe is handled by DD and FDX together. The reason for introducing FDX is to achieve event based behaviour. The CAPL code API can access DD function in order to write to the node as soon as information is available from CANoe. On the other hand CAPL functions are not accessible from outside of CANoe. This is a problem since CAPL functions need to be accessed if a new message arrives from the TCU node. With the FDX protocol it is possible to save the data to system variables inside CANoe and have CAPL code execute whenever these are overwritten by a new message.

A diagram of the solution can be seen in figure 3 with the four CAN buses present in the Mk1 tank.

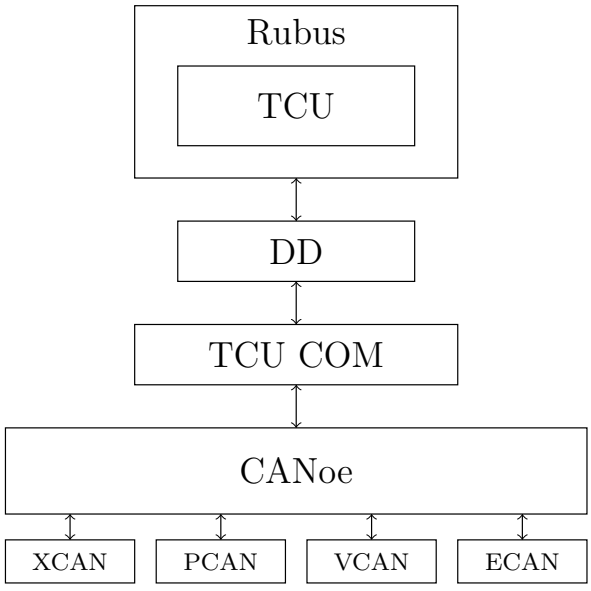


Figure 3: Program structure of the proposed solution.

## 5.1 Contributions

In this section all the programs and scripts created to implement the above solution is presented. The function library TCU Communication (TCU COM) is the backbone of the solution which handles the sending and receiving of data during the simulation.

There is also several Python scripts that generate configuration files and CAPL code files. The creation of these files are done in UC where all the information about the node is stored.

### 5.1.1 TCU Communication

TCU COM is a function library which facilitates the sending and receiving of messages in CANoe. It defines functions which can be called by the CANoe CAPL code.

TCU COM is a CAPL-dll[6] which is a way to inject functions written in C or into the CAPL/CANoe environment. The functions which can be called by the CAPL-code are limited in what kind of argument data types they receive and which data type is being returned. They can only use basic data types such as char, short, long, double, integer array and string.

The most important functions which are provided by TCU COM are described below.

#### 5.1.1.1 Initiation of DD

The initiation of DD requires the path to the DD dll file and the path to the DD xml file. These need to be passed from the CAPL code. The initiation creates a DD realm and a DD client for TCU COM to interface with DD realm.

#### 5.1.1.2 Memory Space Handles

Handles to each memory space can be requested by name. These are stored in order to avoid potentially expensive lookups each time a new value needs to be stored of retrieved.

#### 5.1.1.3 Memory Space Subscription

The DD interface lets the user attach callback functions to the DD memory spaces. These functions are called when the memory space is written to. TCU COM lets the CAPL code subscribe to memory spaces with callback functions that creates a datagram for either a 20 byte CAN message or a 4 byte I/O signal and sends the data through a socket to CANoe with the FDX protocol.

#### 5.1.1.4 Write to DD

In order to write CAN messages or I/O signal to the TCU node the program has two function, one to write a CAN message to a CAN bus and one to write to a I/O signal.

### 5.1.2 Configuration Files and Python Scripts

The simulation environment requires several configuration files to function. These are created based on the information in the UC database which defines the system. The procedure is done by Python scripts. It also uses a CAPL code file which handles event based procedures during the simulation. There is one CAPL code file for doing the HIL simulation and one for the pure simulation.

#### 5.1.2.1 DD

DD require a configuration file, dd.xml, which defines which memory spaces need to

be created and their names. These are usually 20 byte spaces for each CAN bus and 4 byte spaces for each IO signal. These names have to match the names which the TCU node uses to ensure that the information from the node is stored at the same space that is read by other applications.

#### 5.1.2.2 FDX

The FDX protocol needs to know what kind of data will be sent. Therefore it requires a configuration file before the start of data transmission which defines what kind of datagrams are sent. Each datagram requires a group identifier and a predetermined size. For each block of information in the datagram it also requires byte offset, data type and identifiers for the information blocks. These identifiers need to correspond to the data variables in the CANoe environment.

#### 5.1.2.3 System Variables

CANoe uses system variables to store information from the simulated nodes. These are also used to facilitate the information transfer between the TCU node and the CANoe environment through the FDX protocol. The FDX protocol stores all incoming information in system variables.

#### 5.1.2.4 CAPL Code SIM

The CAPL code for the simulation consists of four different kind of blocks. The first block declares all variables to be used. These variables are the DD path and its handles to the variables in DD. The second block is the "on start" block which executes itself at the beginning of the simulation. This block initiates DD and requests handles from DD which are stored in the handle variables. The third block type is the event handlers. They execute their code when certain events occur. These are typically a change in a system variable or a certain message is broadcast on the bus. These blocks enables the transmission of information from CANoe and detection of new incoming information from the TCU. The last block is the "on stopMeasurement" block which releases the DD instance when the simulation is over.

#### 5.1.2.5 CAPL Code HIL

The CAPL code for the HIL simulation consists only of event handlers. They are responsible for detecting changes in IO signals to and from the IO hardware.

## 5.2 How the Simulation Works

There are several stages in the start of the simulation and during the simulation which will be described in more detail in this chapter.

### 5.2.1 Pre-simulation

Before the simulation starts all the configuration files described above needs to be created from UC. The files are created and placed automatically in the folders specified

in UC.

The DD file needs to be placed in an appropriate folder which is predefined by the TCU simulation program. This is important since that folder path is not controllable by the user.

The FDX folder path needs to be added in CANoe in the preferences window. The FDX file is not required for the TCU COM function library. Therefore no specific folder path is required in relation to TCU COM. The IDs which link memory space names to FDX groups is provided by the CAPL code. The port number in CANoe has to match the port number in the TCU COM function library.

The system variables file is added to CANoe in the system variables preference tab. They include the variables in which the FDX protocol save the incoming information.

The TCU COM dll file is also added in the preferences window in CANoe. This allows the CAPL code to use its functions. No imports are necessary in the CAPL code.

The Rubus TCU simulation does not need to be started since the CANoe simulation environment is in control if the TCU is turned on or not. Therefore the TCU will be started during the simulation.

The CAPL code is added to the virtual TCU node's CAPL code window. The code's main function is to receive and send information to and from the TCU node and the CAN buses. Therefore to code defines no real behaviour of the node except turning it on or off which is controlled by a event handled system variable.

With these configurations the simulation can be started with BAE Systems' provided CANoe configuration for the other nodes in the system.

### 5.2.2 Simulation

At the start of the simulation a realm instance of DD is created and a DD client is connected to it from CANoe's CAPL code. All memory spaces defined by the DD file is created and CANoe requests handles to every memory space. Afterwards each memory space is subscribed to with a callback function which is defined to send data through FDX.

During the simulation the user starts the vehicle from inside CANoe using a virtual dash board. The TCU simulation software is executed as a separate process on the computer. The TCU subscribes to all signals in DD in order to see all outgoing information from CANoe.

When a message is put on a CAN bus from the TCU the callback function provided by TCU COM is executed and creates a FDX message and sends it to CANoe. In CANoe the data is saved to system variables. The writing of these variables creates and interrupt in the CAPL code which indicates that a new message has arrived. Then a callback function in the CAPL code executes and packages this data into a CAN message and outputs it on the bus. The reason why FDX is used in the sending of messages to CANoe is to get the event based behaviour mentioned earlier. There is no other easily accessible way to interrupt the CANoe simulation from outside other than to create a CAPL interrupt by sending data to system variables which is a main use of the FDX protocol. A visual interpretation can be seen figure 4.

Similarly when a message is broadcast from CANoe to a CAN bus an interrupt is created in the CAPL code. Then a callback function reads all relevant information from the message and sends it to the memory space in DD which represents the CAN

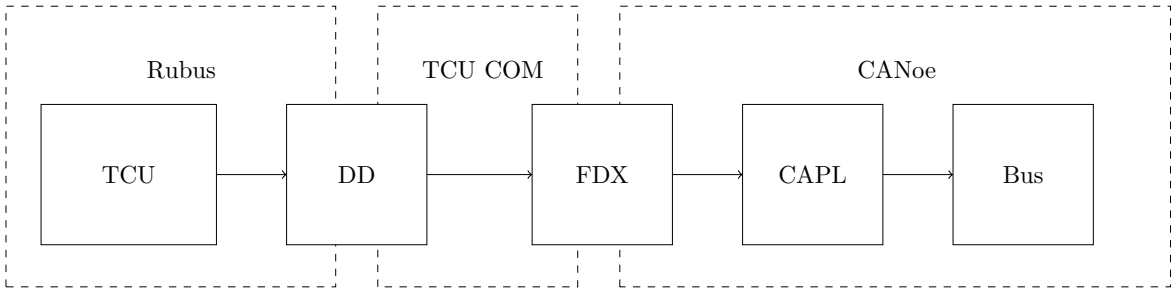


Figure 4: A message from the TCU to CANoe

bus in question. When this space is written to in DD a callback function provided by the TCU simulation is executed. This callback function sends the data to the TCU and the TCU determines if the message is a relevant message to it and if so reacts accordingly. The reason why FDX is not used with messages from CANoe is that the DD interface with its callback functions is enough in that particular direction of data flow. A visual interpretation can be seen figure 5.


Figure 5: A message from CANoe to the TCU

# 6 Results and Discussion

To test the quality of the proposed solution CANoe's built in test feature was used.

To illustrate the practical performance of the solution a series of pre built tests were used which were provided by BAE Systems. These test simulate realistic use of the node. They are meant to simulate how a normal user would use the node. Thses are the most important kind of tests.

Unlike the realistic testing a synthetic test does not measure normal use of the node. The synthetic test measures if the node meet specifications of performance. The test measure time between each cyclic CAN frame and checks whether the timespan is within the specified ranges. A message which arrives 50 percent too early or too late is considered a failure. This test is smaller and less important than the realistic test.

## 6.1 Realistic Testing

The realistic testing measures all basic characteristics of the system. An example of a test could be that the battery is turned off and the testing environments checks that the voltages in the system decreases towards zero. These tests measure the behaviour of the system. The speed of the system is not the primary attribute which is tested.

The proposed solution passed practically all of these tests. This concludes that the simulation implementation behaves accurately for normal use and testing.

## 6.2 Synthetic Testing

The synthetic testing measured the times for all cyclic messages on all four CAN buses. The two buses which the most traffic failed the test. The two buses with lower amount of traffic passed.

A majority of the messages on the buses that failed are still inside the acceptable limit. A minority of the messages fail and a small fraction of those are outliers with very large errors.

## 6.3 Discussion

Although the solution does not meet all specifications it is not necessarily a problem since the issue is undetected during normal use. As long as the realistic tests pass the solution is considered success.

The fact that not all practical tests passed is not of concern since the tests themselves can have errors or are not up to date with the newest version of the TCU. The creator of the test scripts considered all test to be passed even though some tests reported a failure.

Whilst the solution works for one node the results from the synthetic test might be problematic for the requirement of full deployment of newer tanks. The Mk3 tank uses several nodes of this type. Since one node does not work perfectly it brings up the question if several nodes might not work at all. Worst case scenario would be if even the practical tests would fail for Mk3. The synthetic testing might be an indicator of a limitation in the solution.

The importance of the synthetic tests pales in comparison to the practical tests. As long as the practical tests pass, the solution is considered to all intents and purposes a success for the intended use and application of it.

# 7 Conclusions and Future Work

Although the synthetic tests failed the proposed solution is considered a success for systems of one node of the central communication circuit type. The reason it can be considered a success is because the practical parts of the simulation is the most important for BAE Systems when testing the node on a desktop computer.

The behaviour of the simulated node is not expected to show less nuanced problems which are detected on real hardware anyway. The point of desktop simulation is to be able to develop automatic test scripts early in the development process. Being able to find logical errors in the scripts early, even before having access to real hardware, saves a lot of time. Therefore the simulation can be considered accurate for the practical uses it is intended for.

Future work in that area include:

- Modifying existing UC configuration files to facilitate generation of Mk3 configuration files from Python scripts.
- Modifying existing Python scripts to generate Mk3 configuration files.
- Eliminate bottle necks in any of the programs if needed.
- Bug fixes of other simulated nodes.

The behaviour of full deployment for the Mk3 tank remains to be seen. That will be the real test of the proposed solution. In the worst case scenario the solution is practically incompatible with several nodes of this type on a desktop computer and should be confined to single node use.

A qualified guess for the reason behind the failures of the test cases is that there is unecessary load on the CANoe simulation environment. The main communication subroutines can be exported outside of the CANoe environement and be used by a seperate process which can be executed on another core. If this provides a significant improvement remains to be seen but it should be the first thing to explore. Further work would be to explore different optimizations if the above mentioned is not enough. There is no reason at this point to believe there is a fundamental problem with the solution. Any modern computer should theoretically be able to handle the work load.

Although if that is not the case the porting to the Mk3 tank, which uses several nodes, should be possible and pretty straight forward given some time.

In either case this solution will be implemented in the offices of the SIL-lab and used in the single central node projects. This will help them in the early stages of testing, test script design and decrease the use and queue time for real test benches.

Hopefully it will be developed to several central node projects as well and used there too.

# 8 References

[1] Vector Informatik GmbH, (2016, November 13th). E-LEARNING - CAN Controller Area Network. Retrieved from: http://elearning.vector.com/vl_can_introduction_en.html/

[2] Vector Informatik GmbH, *AddOn MATLAB Interface - User Guide*, v.2.2, (Vector Informatik GmbH)

[3] Vector Informatik GmbH, Mark Schwager, *Using MATLAB with CANoe*, v.1.0, (Vector Informatik GmbH, 2008)

[4] Vector Informatik GmbH, Stefan Krauss, *Testing with CANoe*, v.1.0, (Vector Informatik GmbH, 2009)

[5] Vector Informatik GmbH *CANoe FDX Protocol*, v.1.4 English (Vector Informatik GmbH, Stuttgart, 2013).

[6] Vector CANtech Inc, Jun Lin *Implementing and Integrating CAPL DLLs*, v.2.0 (Vector CANtech, Inc, 2005).

# A Standard CAN Frame

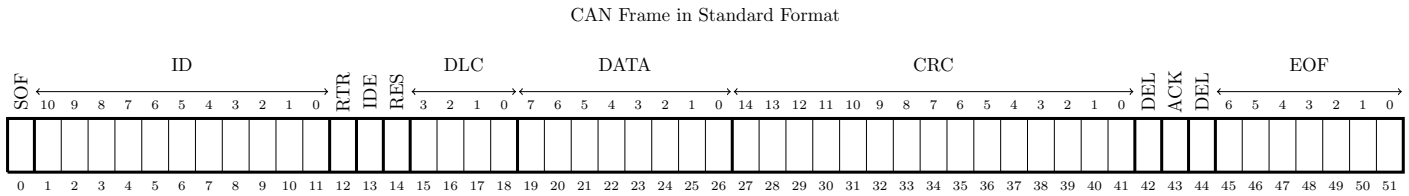


Figure 6: A CAN frame in standard format with one byte of data.

| Name | Length (bits) | Description |
|---|---|---|
| SOF | 1 | Start of frame, (dominant). |
| ID | 11 | An identifier representing name and priority of the frame. |
| RTR | 1 | Remote transmission request. Dominant for data frames and recessive for remote frames. |
| IDE | 1 | Identifier extension bit. Dominant for standard format and recessive for extended format. |
| RES | 1 | Reserved bit, should be dominant. |
| DLC | 4 | Data length code. Number of data bytes in the frame. |
| DATA | 0 - 64 | The data bytes. The length is specified by the DLC. |
| CRC | 15 | Cycle redundancy check. |
| DEL | 1 | Recessive delimiter between CRC and ACK. |
| ACK | 1 | Sent as recessive and returned as dominant if no errors occured. |
| DEL | 1 | Recessive delimiter between ACK and EOF. |
| EOF | 7 | End of frame. Seven recessive bits in a row to indicate the end of frame |

Table 1: The standard CAN frame data fields.

# B Extended CAN Frame

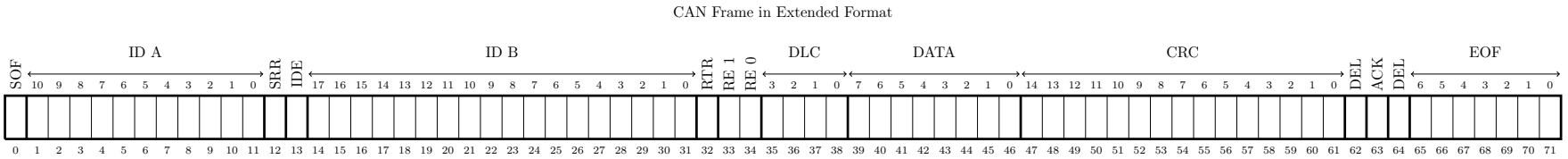


Figure 7: The extended CAN frame.

| Name | Length (bits) | Description |
|---|---|---|
| SOF | 1 | Start of frame (dominant). |
| ID A | 11 | First part of the identifier representing name and priority of the frame. |
| SRR | 1 | Substitute remote request (recessive). |
| IDE | 1 | Identifier extension bit. Dominant for standard format and recessive for extended format. |
| ID B | 11 | Second part of the identifier representing name and priority of the frame. |
| RTR | 1 | Remote transmission request. Dominant for data frames and recessive for remote frames. |
| RE | 2 | Reserved bits (should be dominant). |
| DLC | 4 | Data length code. Number of data bytes in the frame. |
| DATA | 0 - 64 | The data bytes. The length is specified by the DLC. |
| CRC | 15 | Cycle redundancy check. |
| DEL | 1 | Recessive delimiter between CRC and ACK. |
| ACK | 1 | Sent as recessive and returned as dominant if no errors occured. |
| DEL | 1 | Recessive delimiter between ACK and EOF. |
| EOF | 7 | End of frame. Seven recessive bits in a row to indicate the end of frame |

Table 2: The extended CAN frame data fields.

# C DD CAN Frame

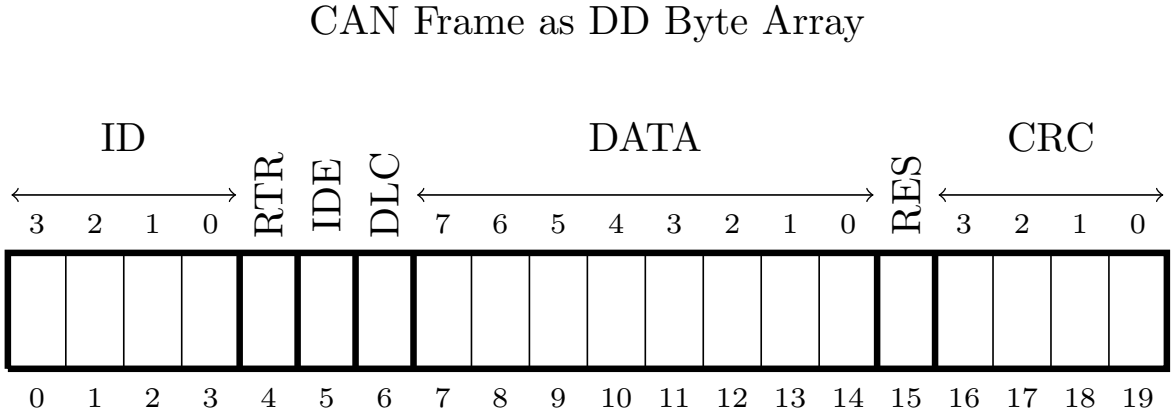


Figure 8: The DD CAN frame.

| Name | Length (bytes) | Description |
|---|---|---|
| ID | 4 | The CAN frame identifier. |
| RTR | 1 | The remote transmission request bit. |
| IDE | 1 | The identifier extension bit. |
| DLC | 1 | The data length code. |
| DATA | 8 | The data bytes. |
| RES | 1 | Reserved byte. |
| CRC | 4 | The cycle redundancy check. |

Table 3: The DD CAN frame data fields.