

Options handling

Niklas Blomqvist, Robin Gustafsson

Contents

Revisions	3
Abstract	4
Introduction	4
Motivation	4
Purpose	4
Problem	5
Delimitations	5
Theory	5
The forklifts	5
CAN	6
Protocols	6
Worst case CAN latency	7
Method	7
Feasibility study	7
Implementation of prototype	8
Evaluation	9
Result	9
Feasibility study	9
Current system	9
PDO Interface	11
Implementation of the prototype	14
Software flow	14

Implement an option	16
Evaluation	16
Bench prototype	16
Interview	17
Truck prototype	17
Performance	19
Discussion	20
Result	20
Method	21
Future work	22
Workload Distribution	22
Conclusions	22
Citations	23
Vocabulary	24
Appendix A, test options	25
Turn lights	25
Lift height restriction	25
Speed and steer angle restriction	25
Horn	25
drive speed reduction	25
hydraulic assistance	25
Appendix B	26
Questions asked to our mentor	26

Revisions

Version	Date	Sign off	Change note
0.1	2015-02-23	Robin, Niklas	First draft

Abstract

Unique customizations (options) of features in forklifts are often requested by customers. When new options are created or present options have to be modified in the main software the complexity increases, the firmware revision pool gets large and with the increasing code size the memory limit is threatened. This affects the software development since the frequent modification of the option handler software is very resource consuming. Therefore it is desirable to have a highly modular system for the option handler to reduce the development process. Although the market value of this improvement is negligible the possible long term savings is the desirable effect. The purpose of this thesis is to explore the possibility of migrating the option handling software to a dedicated hardware module. This will help the development process by increasing the modularity of the system architecture and thus reducing the development scope. Methods for model based development will be utilized to explore ways to efficiently speed up the software development process. The terms of inclusion and the tools to accomplish this option handler is analyzed. A system model of the resulting approach will be designed and a prototype will be developed to validate the result.

Introduction

We have conducted our thesis work at a big forklift manufacturer located in Östergötland as a part of our Bachelor degree in Computer Science. In this report the forklift manufacturer will be called “The company”.

Motivation

A large quantity of the sold forklifts is equipped with non-standard options requested by the customer. An option might be anything ranging from turn indicators to an advanced hydraulic sequence with height, weight and speed restrictions.

These options are all implemented in the firmware that controls the truck. Currently, The company has no way to decouple the option implementation from the main firmware. This means pollution of the source code tree as separate branches has to be created for each customer specific option. This also means that there are multiple variants of the same version of program code that needs to be maintained.

Purpose

In order to satisfy the increasing customer demand of new features (options), The company needs a faster, more reliable and testable way to develop them. Currently, options are added to the main firmware.

This thesis aims to validate the vision of an external option handler. The option implementation will be decoupled from the main firmware and a dedicated unit for managing options separately will be added. By doing this we will achieve a more modular system and this will speed up development of new features aswell as decreasing the number of potential bugs in the main firmware.

Problem

- What needs to be taken in consideration when designing the options controller?
- How is performance of the CAN bus affected, if additional controllers are added, in regard to bus-load and roundtrip-time?
- Do we need to make modifications on the current CAN-bus communication protocol?

Delimitations

The time will not be sufficient to develop a full scale version of the options handling. With respect to that, we have chosen to spend most of the time developing a working architecture, and a prototype. The prototype will be written in such way that it should be easy to extend with new features.

The fundamental part of this thesis is the development of an architecture as general as possible. It is therefore not vital that we implement all the existing options, as long as the architecture can be deemed good enough to handle them. This will be tested by implementing a few option that utilizes all of the different part of the truck; hydraulics, drive, steer and display.

Further, one possible delimitation might be to hand off the MCU side of the development to The company. This option, however, depends on how much time they can spare.

Theory

In this chapter, we explain the theory needed to fully grasp the method and result.

The forklifts

The truck is divided in different function domains, controlled by different hardware. These domains are drive, steering, hydraulics and other miscellaneous peripherals. Among the latter we have the SSU¹ and BCU², but it could also be internet connectivity or other sensors of any kind.

All of these domains are controlled by the ICH which delegates commands over the CAN bus. Some controllers, such as the SSU tells the ICH when something is wrong. The ICH then has to take action; it can be reducing the speed to a halt or steer in a certain direction.

This means that when developing the options controller, we still have to manage critical functions in the ICH. It is not safe to rely on an *external* device controlling this. The ICH must have a full non-overrideable fail safe mode.

The option controller will therefore only ask the ICH to execute tasks. The ICH will always have the last word over the action being requested.

¹Shock Sensor Unit

²Battery Control Unit

CAN

The CAN protocol utilized in the trucks is *CANopen* which has support for network management and device monitoring. Messages are being sent in frames where the data is divided as following:

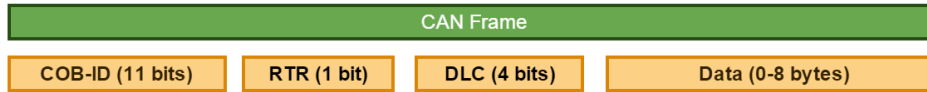


Figure 1: CAN frame

The communication object identifier, COB, consists of 11 bits of data, where the frame with the lowest value (ID) has the highest priority. This means that in the case of bus collision, the packet with the highest priority wins.

The RTR, or remote transmission request, is not used by us, but it can be used to request data. Normally this is set to 0 as the data objects usually transmits ciclycally.

The data length code (DLC) tells the recieving end how many bytes of data to expect. The maximum bytes of data you can send in one frame is 8.

Protocols

In the truck system multiple communication protocols are used; Network Management (NMT), Service Data Object (SDO), Process Data Object (PDO) and Emergency Object (EMCY).

NMT: The NMT protocol is used to change device states. You will notice that our implementation does not follow this protocol. It was not a first hand priority to implement. The states which may be requested are (hexadecimal address and *name*):

- 0x01 *operational*
- 0x02 *stopped*
- 0x80 *pre-operational*
- 0x81 *reset node*
- 0x82 *reset communication*

SDO: The Service Data Object protocol is used when a client needs to get or set a value on the server. A use case for this might be reading the driver profile from the ICH in order to compensate options to driver preferences.

PDO: This is the most used protocol in the truck. There are multiple PDOs being sent to and from the ICH and other units. These objects are typically relatively time critical as they contain information ranging from drive speed to steer angle.

Our implementation builds fully on this protocol.

EMCY: A device can send an error message on internal fatal error. They are sent with high priority which makes them usable as interrupts if the recieve routine is adapted.

Worst case CAN latency

The PDOs are sent with a 20 ms interval making the worst case roundtrip 60 ms. This is because different PDOs can have different priority, depending on the receive address. If the CAN bus becomes temporary congested due to previous transmit error or other reason, the PDO queue of each unit tries to re-send. When this happens, packet latency will occur. Packet latency will occur all of the time when two units tries to send data at the same time. Often, this latency is negligible as it normally lags behind only a couple of milliseconds.

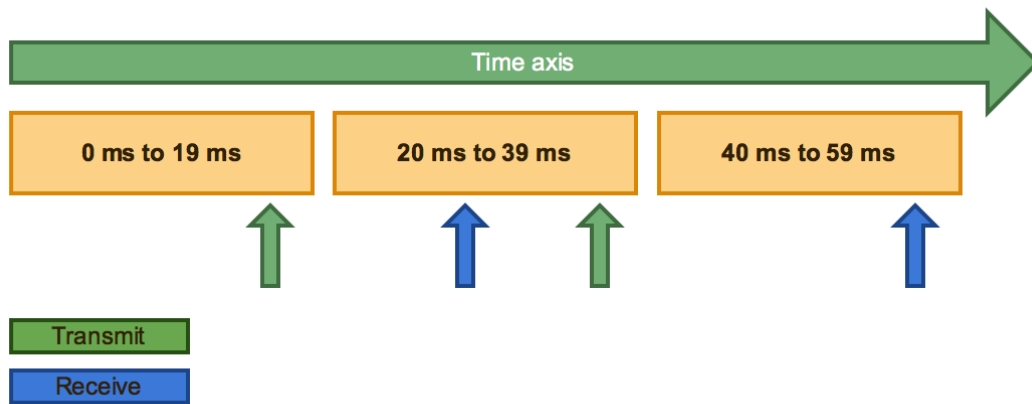


Figure 2: Illustration of worst case PDO latency

This is something you would have to consider when implementing time sensitive operations, such as emergency stop. The system has been tested by The company and remains very stable even at periods of busloads far above 100 %³.

Method

In this chapter, we describe the approach and how we concluded the results of this thesis.

Feasibility study

We began by establishing the criteria for the options handling. This included asking our mentor how it was supposed to function, but also reading up on how the options are implemented currently.

In the beginning we worked mostly without coding, discussing possible valid solutions and put them through theoretical dry-runs. By trying to get our thoughts on paper directly, we managed to avoid failures but we also got some documentation *for free*. Continuously, we asked questions as they came up.

³A packet queue larger than the possible packet rate

To implement an external option handler, the embedded software design had to be studied in order to establish the new system prototype. The vital parts of the current option handler were to be identified and the expandability of the CAN interface explored. Tools for implementing the prototype were identified.

Before implementing the OCU prototype we designed a complete system model including details about hardware aspects, software flow and CAN interface.

Implementation of prototype

The system model gave way for the iterative implementation process of the prototype. The implementation started with very simple sub-prototypes mostly aimed towards testing our understanding of the CAN-bus.

We were introduced to the MCU2B which was perfect for the purpose of representing the external OCU. Together with the MCU2B we would have the standard ICH hardware to represent the original system. These hardware modules would link together using a CAN-bus fork harness. Additional hardware needed to establish the prototype included a 24 volt power supply and two CPC-USB; one for debugging of the CAN-bus and one for firmware download.

First, we started with the classic “Hello world!”, by making a LED blink. From there, we began removing unneeded code and started creating a foundation for the options to build on.

The first fundamental prototype, the bench-prototype, forked into two systems where the OCU was implemented to the extent possible in a standalone state. To complement the standalone OCU, A debug tool were developed to represent all the sub-systems not available at this stage of the implementation process. The final prototype implementation was a real truck application where all the compromises, introduced by the debug tool, were eliminated.

The company have CAN bus interfaces with competent software, but they cost a lot, and there is a limited access to them, as they are used by other developers. This lead to us the creation of our own CAN-debug tool. Our custom CAN-debug tool used the EMS CPC-USB hardware, which is much more affordable and available for use in the development team. Thanks to Volkswagen Research⁴, the Linux kernel has support for CAN. This made it possible for us to build a relatively complete test bench application for simulating the truck with a small Python GUI. The debug-tool could be used to represent the abscent hardware functionalities of a truck. E.g. a slider in the GUI could represent the fork height since no actal forks were available in the bench-prototype.

Test option 1, 2 and 3⁵ were specified before implementation of the first prototype started. These were of great assistance since the test options worked as milestones when implementing the OCU prototype and also help us locate flaws early in the implementation process. These three options would also be used to validate the system at the half-time presentation of the prototype.

To sum up, the neccessary essentials to start the actual implementation included the following: A complete system model including CAN PDO interface and software flow, The rigg was established and a few test options for validation were specified.

⁴<https://www.kernel.org/doc/Documentation/networking/can.txt>

⁵See Appendix A

Evaluation

Evaluation occurred somewhat successively inline with the several sub-prototypes were finished to be able to move on to the next step. Several tools to validate the system, identified in the study phase, were utilized. The main evaluation strategy were to involve test-options⁶ to easier identify limitations of the prototype.

The first test bench prototype were implemented to demonstrate the possibility of an external option handler. At this stage we used our debug interface and test option 1, 2 and 3 to validate the system. Some output were fully functional and some were simulated in the debug GUI. This prototype was used in the half-time demonstration of the system as well.

We conducted an experiment where we let a system developer at the company implement a couple of options, not aware of the embedded system design. He was only instructed to follow the user-manual and later interviewed to evaluate the system by answering a handful of questions.

With the completion of the test bench we utilized the spare project time to further develop the prototype. At this point the goal was to get the prototype to function in a live truck with few modifications.

The grand final of the evaluation process had all the test-options active on a real truck. Final results were mostly based on this phase.

We made sure to collect performance data such as bus-load and roundtrip-time during the whole project and especially every time we added traffic to the bus.

Result

In this chapter, we present the result divided into the result of the feasibility study and the result of the implementation and evaluation of the prototype.

Feasibility study

From our feasibility study, we were able to conclude that an external options handler represented on the CAN bus would be not only feasible, but also the best solution.

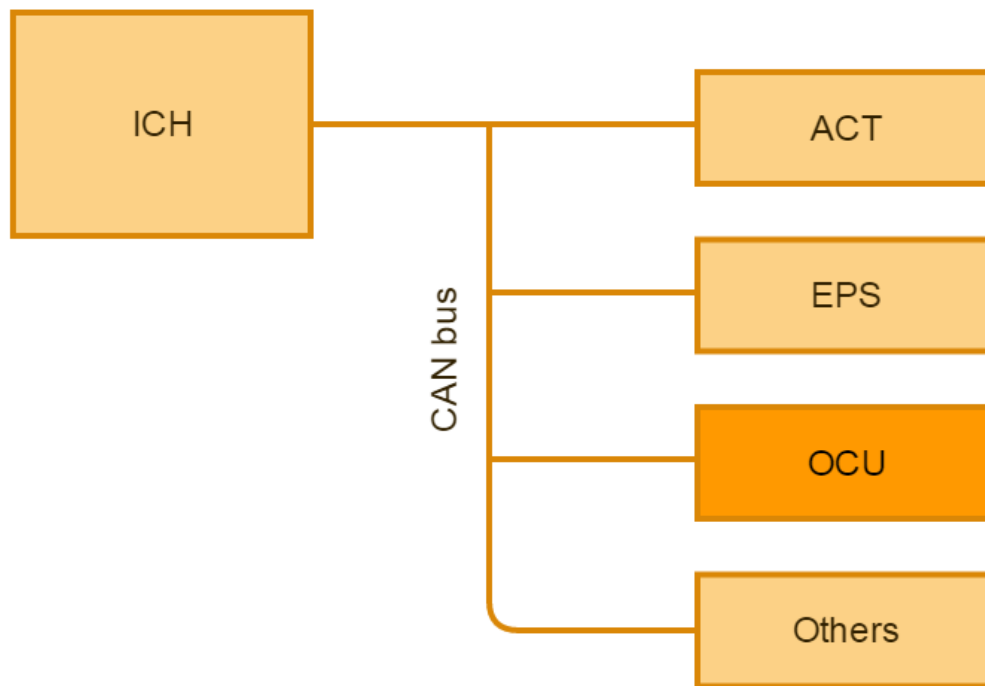
Current system

The company has an options handler where the options runs tightly coupled in the main loop. The options are setup with a parameter table, one row with multiple columns per option. This table has to be modified for each truck as some parameters differs between different truck models.

This makes it *dangerous* and non-trivial to implement new options as a bug in one option might cause the truck to fail. The truck does implement fail detection in the kernel, and there is also a watchdog⁷ which triggers if the code stops responding. However, the code in the unit responsible of controlling the truck should ideally be modified as little as possible. Having all of the options integrated in the main firmware makes it harder to test the functionality.

⁶See Appendix A

⁷Hardware timer which restarts the system if the timer itself has not been restarted



The current option handler as it sits has direct access to a lot of the internal hardware- and software functionality of the ICH. Thus the individual options have a lot of freedom towards modifying the truck functionality. To migrate the option handler to an external unit the current level of access had to be kept in the new system.

Theoretically since the functionality available inside the ICH had to be accessed from outside the ICH we had to add additional CAN communications to access these from the new option handler, connected to the bus. Some of the operations available internally on the ICH can be recovered from the raw CAN traffic already present on the bus; functions such as reading the current speed or steerangle.

The main challenge of the project was to find the correct balance and identifying the vital parts for the option handler to operate outside the ICH. The CAN communication we had to add came with the price of risking to flood the bus. This is not desired since the overall response of the system would decrease. We only considered adding traffic if it was absolutely necessary. The signals already present at the CAN-bus were prioritized and utilized to the full extent. These signals could simply be sniffed by listeners without adding traffic to the bus.

The kernel of the ICH ticks⁸ every 1.25 ms. In the MCU2B, the kernel ticks at 1 Hz. In both cases, the main application loop is called once every 20 ms. If the loop is not finished in that time, the truck will stop.

This is roughly how the interrupt routine for the kernel works. The code is simplified to reflect the gist.

⁸Generates interrupt in which kernel and application functions are called

```

void runKernel(void)
{
    DisableInterrupts;

    handleCanTransmit(); // Send queued CAN messages

    // Every other kernel tick
    if (CurrentKernelTick & 1) {
        handleX();
    } else {
        handleY();
    }

    // On the tick before the application loop, start A/D conversion
    if (CurrentKernelTick == KERNEL_LAST_TICK) {
        startAdConversion();
    }

    EnableInterrupts;

    if (CurrentKernelTick == KERNEL_RUN_APPLICATION) {
        preApplication(); // Prepare various input/control status
        handleCanReceive(); // Make recieved messages available to application
        runApplication(); // Run application based on input (CAN/HW)
        postApplication(); // Set output based on runApplication()
        resetWatchdog(); // Reset watchdog timer to indicate successful run
    }
}

```

The majority of system modules consists of a pre-application, a run-application and a post-application to statically direct the flow of operations. By this software architecture the overlap of several system modules can be predefined to easier prevent timing issues. The memory is statically handled as well so no dynamicaly allocated memory is allowed within the system, the memory is pre-loaded and this setup works well with this type of embedded system.

The pre-application often includes handling of newly received CAN signals. E.g. scaling of raw values etc. The run-application is the main software routine for the module where all the operations is specified. The post-application is last step before leaving the module's software routine. In this step typically instances the CAN transmission routine.

PDO Interface

Together with our mentor we identified the vital signals and specified the new CAN interface for the OCU. The OCU adds a total of three (3) PDOs when configured to full truck interaction. The address of the OCU is 0x1B (decimal 27).

The OCU only requires one PDO to be sent from the ICH. This PDO includes signals required by the OCU to operate. This PDO includes option button bit field among others. As an example, the option buttons are available to read as hardware functionality on the ICH. There are a total of six option buttons on the handle and therefore fits as a bitfield inside one byte. This is perfect

since we can dedicate one byte in the PDO for all the button status where a high bit represent a button at a index beeing pressed. This is the only receive object on the OCU, called PDORx1.

The OCU does recieve other PDOs as well, but they are not addressed specifically to it. E.g. the ACT addresses its data to the ICH, but the OCU has the possibility to read the same data from the bus. For this purpose, we have setup a way to relatively easy listen to PDOs not addressed to the OCU.

<i>Controller</i>	<i>Folder</i>	<i>Getter</i>
ACT	sm_act	tCanMsg* get_CanAct(int index)
EPS	sm_eps	tCanMsg* get_CanEps(int index)
Other	sm_other	tCanMsg* get_CanOther(int index)

Table 2: Folder structure for CAN sniffers

This method makes it relatively easy and clean to implement new listeners. The tCanMsg* get_Can<xyz>(int index) returns a pointer to the CAN message, which makes the recieved data available. The index is used to select from different PDOs sent by a controller. All core controllers sends more than one PDO. The first PDO of the ACT is accessed this way:

```
uint8_t example_data = get_CanAct(0)->Data[0];
```

PDORx1: The PDORx1 is the most important PDO the ICH must implement and send to the OCU. This data object gives the OCU the information that is not already existing on the CAN bus.

<i>Byte</i>	<i>Variable</i>	<i>Data type</i>	<i>Unit</i>
0..1	BflyRamped ⁸	S16	100 %/32508
2	AdLift ⁹	S8	100 %/127
3	AdLift ¹⁰	S8	100 %/127
4	Digital Button bitfield	Bitfield	-
5	Option Button bitfield	Bitfield	-

Table 3: PDORx1

The data frame send the current state of the throttle, *Butterfly*, and the analog value of the forklift controls. The handle has an array of buttons, some for controlling core functionality and some for options.

Among the core buttons, there are the *horn*, the *Belly button* which makes the truck brake and reverse if pressed, and two switches for hydraulic funtions; *DiLift* 1 and 2. They control the high lifting forks and the low lifting forks respectively.

⁸The value of the throttle

⁹The value of the first analog lift control

¹⁰The value of the second analog lift control

Table 4: Button bitfields

Table 5: Digital buttons

<i>Bit no</i>	<i>Alias</i>
0	DiLift2Up
1	DiLift2Down
2	DiLift1Up
3	DiLift2Down
4	Horn
5	BellyButton

Table 6: Option buttons

<i>Bit no</i>	<i>Alias</i>
0	Opt6
1	Opt5
2	Opt4
3	Opt1
4	Opt2
5	Opt3

Set bit indicates a button being pressed. Depending on implementation, a button can act as a switch which toggles on or off, or active high. Multiple buttons might be active at any time.

PDOTx1: This is the object which can control most of the ICH. Functionality can be requested or restricted. Also, combinations can be sent.

<i>Byte</i>	<i>Variable</i>	<i>Data type</i>	<i>Unit</i>
0	Command ¹¹	Bitfield	100 %/32508
1	Drive Speed ¹²	S8	100 %/127
2	Drive Speed Change ¹³	U8	100 %/127
3	Hydraulic command ¹⁴	S8	-
4	Hydraulic function	Bitfield	-
5..6	Steer angle	S16	1/182°

Table 7: Command

Table 8: Bitfields

<i>Bit no</i>	<i>Alias</i>	<i>Bit no</i>	<i>Alias</i>
0	Request drive	0	1 st function
1	Restrict drive	1	2 nd function
2	Request hydraulic	2	3 rd function
3	Restrict hydraulic	3	4 th function
4	Request steering	4	5 th function
5	Restrict steering	5	6 th function
6	Request power	6	7 th function
7	Not used	7	8 th function

Table 9: Command

Table 10: Hydraulic function

It is up to the implementation in the ICH to determine if, for example, multiple functions can be requested simultaneous: speed and steer angle can be relevant to restrict at the same time. Some function combinations are moot; request and restrict uses the same field for the speed.

¹¹ A zero indicates no request/restriction

¹² Range -12.5 to +12.5 km/h. Sign determines direction. If value is zero, ICH controls speed change

¹³ Positive value corresponds to lowering. Hydraulic function must be set.

¹⁴ Only one hydraulic function can be active at any given time.

Request power is not required on any of our prototypes. Due to wiring and other circumstances, we did not implement this.

PDOTx2:

<i>Byte</i>	<i>Variable</i>	<i>Data type</i>	<i>Unit</i>
0	Display 1 ¹⁵	U8	-
1	Display 2	U8	-
2	Display 3	U8	-
3	Display 4	U8	-
4	LED indicators bitfield	Bitfield	-

Table 11: Display and LED data

This process data object is only used to control the display and its surrounding LEDs. If the first byte is set to 0, the display will be handled by the ICH.

<i>Bit no</i>	<i>Alias</i>
0	Time indicator
1	Pot indicator
2	Battery indicator
3	Tool indicator

Table 12: LED indicator bitfield

Implementation of the prototype

inledande text

Software flow

First we modified the ICH and OCU software to run `benchbuild #define BENCHBUILD TRUE`. This mode allows us to run the ICH standalone without the need of the several external modules e.g. EPS or ACT modules which are required by the ICH if running non-benchbuild. If external modules are missing the ICH goes into asserted mode from waiting for the modules to respond and does not initialize other modules or any operations what so ever.

The internal storage of options is dealt with in an object oriented manner in order to keep the structure as organized as possible. The option objects is kept in an array which stores details about each individual option. The array is built in the `gw_initializeMcu(void)` function and the CAN PDO-interface is initialized here as well using the standard hardware function. This function is called upon by the standard set-up routine. The OCU will not instance the receiving of PDO-messages, this is handled automatically by the hardware and the received data can be directly used in the assigned register.

When the CAN signals have been received all the options execute their corresponding function in a loop inside `gw_runMcu(void)`. This means that every option will run its function regardless of if the signals, that each option is dependent of, has changed or not.

¹⁵Display 1 is the leftmost digit. When set to 0, ICH controls the display.

During the option loop the CAN send buffer will be filled with function calls for the ICH. The buffer will have a static capacity and set of functions to call. The standard CAN protocol will be used, meaning that every 20 ms packets in the buffer will be transmitted to the ICH.

Unlike the receiving of PDO data, the transmission have to be instanced and this is conveniently done from the post-application to ensure that the correct data from the main option loop is transmitted. The transmission will be instanced by the `gw_postApplicationMcu(void)` meaning that it will be called each 20ms but after the run loop has completed. These packets will be cleared before the options run, which means that the “non-triggering” cases not have to be dealt with.

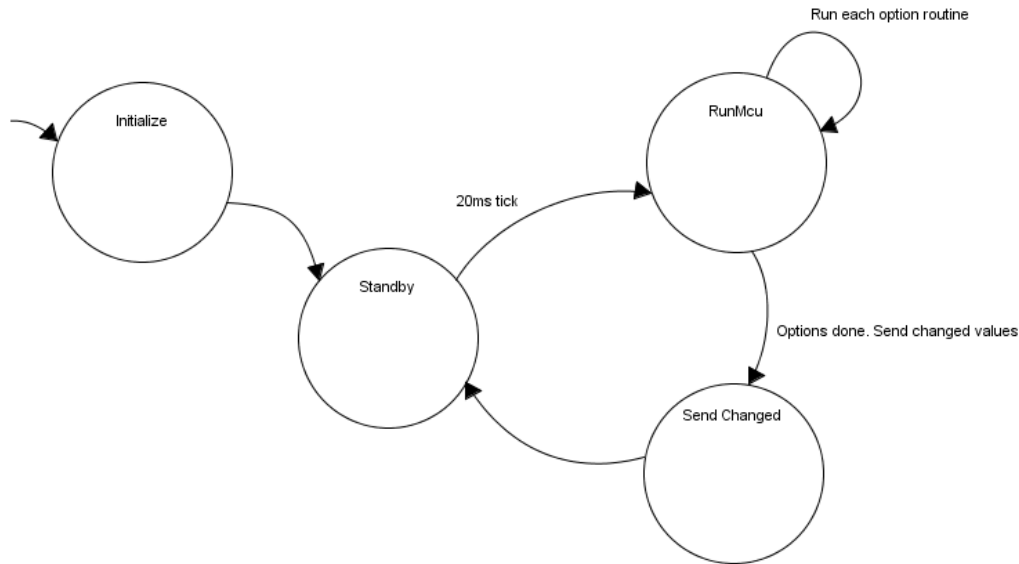


Figure 3: OCU

/newpage

The option folder structure is designed so that the individual options are implemented in private files. This makes option development file oriented which helps during script assisted option implementation as well as organize the active options. An option is to be created in the `/sm_options` directory. The preferred way is to create an option by using the `generate_option_template.py` script which will create a template header and source file. The header file will include a function declaration and the source file will include an empty function skeleton. There is no need to include anything besides what the template generator includes. In `mcu.c`, all that needs to be included is `sm_options/option_functions.h`. The options “main” function is active by assigning it to the `OptionArray` in `mcu.c`.

The ICH side of the system is not that sophisticated and could use some more implementation. We considered it not to be of that importance for validating the vision of an external option handler therefore we only implemented the vital parts to validate the OCU. Some function calls were hard to test in the early stages of the prototype since sub-systems of the truck were missing. E.g. the steering potentiometer was not connected to the handle, thus was not implemented fully until the prototype was installed on a real truck. Basically the ICH sends signals depending

on its hardware input, the signals packed in the PDOtx1 interface, E.g. Option button being pressed. Upon receiving function calls, on one of the PDOrx, the ICH calls the appropriate hardware functionality assigned to this function. E.g. activate an output.

Implement an option

förklara hur man använder toolbox referera till appendix med nuvarande optioner

Evaluation

inledande text för result evaluation

Bench prototype

The majority of the system were implemented under the bench prototype stage. The bench prototype were installed in a rig with only the separate forklift master controller handle, which contain the ICH, and the separate OCU. All the primitives needed for this prototype were not available given that all the hardware modules were not available. Thus we were somewhat limited and forced to compromise.

To evaluate the bench prototype using test options 1.2 and 3 [länk till appendix a] we added two small LEDs to represent physical indicators driven by a low current output and a pull-up resistor. Other than that the majority of functionality needed to handle the user interaction for validation of the system were already embedded in the ICH, Like buttons and a display. Although some signals had to be simulated virtually to get the system to function fully, this is where our debug-GUI had its prime.

The debug-interface gave us the possibility to simulate a lot of the missing system characteristics. For the three options implemented at this prototype stage to work good enough to function as evaluation samples we had to compromise and add some of the missing utilities to the debug-interface GUI.

Without the steering potentiometer from the OEM trucks we had to manually disengage the turn indicators. A simple horizontal slider, -100% (left) to 100% (right), in the debug-GUI allowed us to simulate the disengagement of the turn indicator by turning the slider.

Without any actual forks in the rig we had to simulate the lifting of the forks and the weight loaded on the forks in our GUI as well. Two vertical sliders solved this problem by displaying the actual height (in centimeters) on one slider and applying weight (in kilograms) with the other.

Some other visuals were added to the GUI to give even more useful output from the system, like dashboard indicators and throttle value and others, but the above are the vital ones as extension to the prototype to fully function.

Some issues we had with the debug-GUI involved mainly the sliders. The resolution of the input sliders were a bit dodgy of unknown reason. This caused us to miss some desired values when testing. E.g. if we wanted to load the forks in the prototype with 1000 kg and placed the slider at 1000, some times the actual value of the sliders were off by up to 100 units.

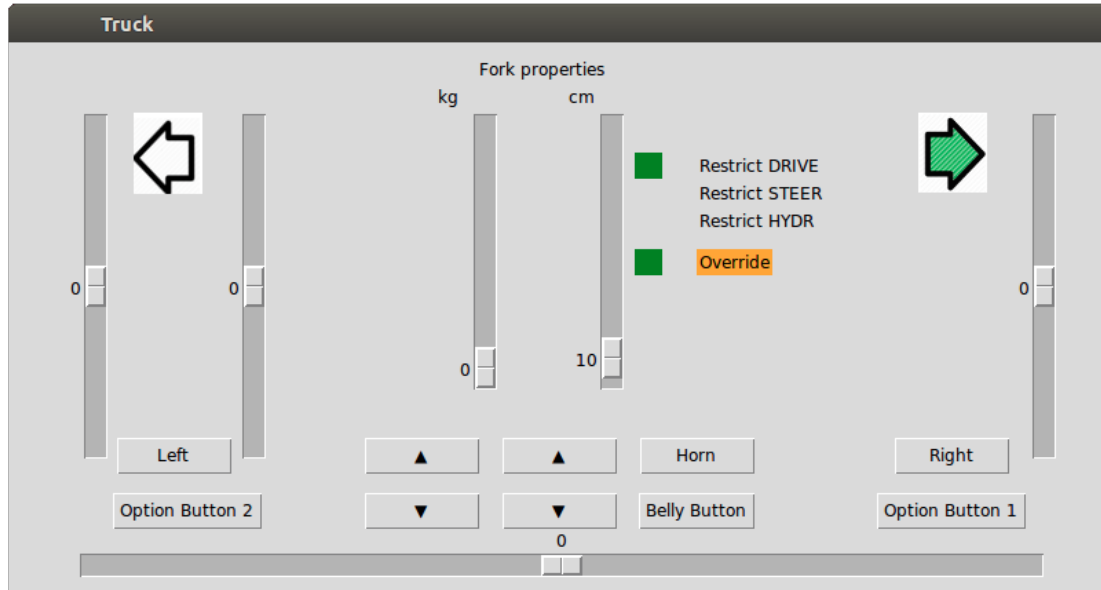


Figure 4: Picture of the interface. Right turn indicator and the override button are active

Interview

The experiment where we let our mentor, in the roll of a system developer, implement a couple of options given the user-manual resulted in options 4, 5, 6 and 7 [länk till apendix A]. Upon finishing the experiment he was asked to evaluate the system by awnsering to a handfull of interview questions. For the result of these questions, see Apendix B [länk till apendix b].

One of the flaw detected by the experiment was that the script that instances the new option code skeleton had to be executed from within the development environment. Some other minor design flaws were identified thanks to this experiment aswell.

Truck prototype

During the last week of the project we got the permission to implement our system on a real forklift. Since we had time to spare, the company supplied us with one of their prototyping forklifts and we modified the bench prototype to run on a live forklift.

The second prototype edition was implemented on a real forklift to remove compromises made in the bench prototype. In this prototype the new system with the external option handler is compleatly abstract to the feel of the product.

All the specified test options was used to evaluate this prototype and for the first time we had all the hardware modules available and with a live forklift we could really se the options modifying the behaviour of the forklift.

One issue we detected at this stage was our interface for sniffing already present signals of the CAN-bus was initialized wrong, the interface was instanced as send/receive PDO. This caused the interface to not respond. One of the signals dependent of this interface was the steering

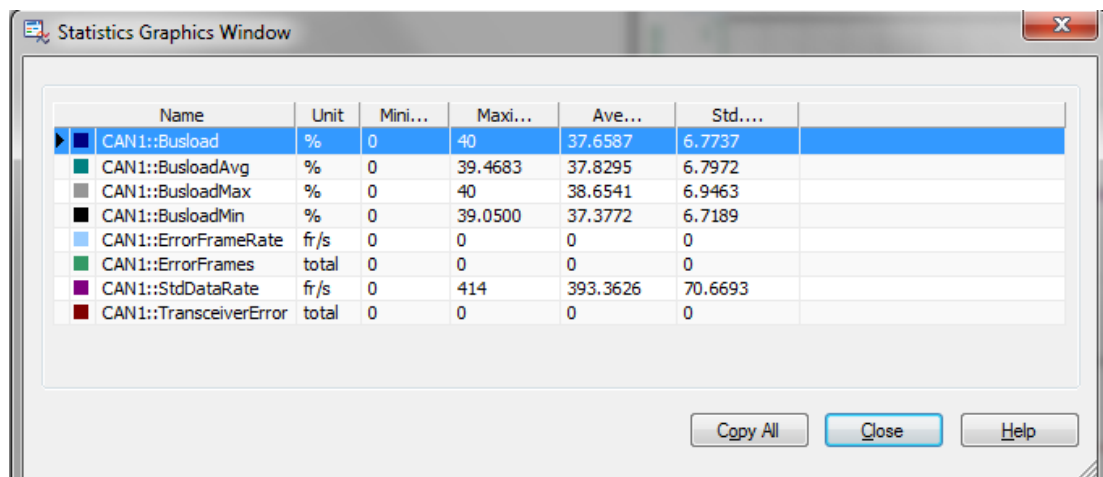
angle, which previously had been simulated in our debug-interface. We simply had to change the software to instance the sniffer interface as purely receive PDO so this was an easy fix.

Previous prototype utilized a permanent power supply and the forklifts had a battery. Since these prototyping forklifts are frequently used, we had to charge the battery to be able to work on this prototype. Otherwise our system was bolt-on with only a few adjustments such as not running the software in bench build and other minor details like such.

Performance

We made sure to collect some performance result of the prototype given that the system is real-time dependent. Since we modified the CAN-bus by adding traffic we had to make sure the additional traffic did not overload the bus and thus affecting the system response time.

To collect the performance samples we used the Canalyser [[länk till förklaring](#)] tool which had embedded functionality to measure CAN busload. The CAN busload unit is measured in percent of available bandwidth used, where 100% indicates that the bandwidth buffer is maximized. The busload can exceed 100% without direct harm to the system but indirectly this implies that some packets may suffer additional and unknown delay which is dangerous in a real-time system.



The screenshot shows a window titled "Statistics Graphics Window" with a table of CAN busload statistics. The table has columns for Name, Unit, Mini..., Maxi..., Ave..., and Std.... The first row is highlighted in blue. Below the table are buttons for "Copy All", "Close", and "Help".

Name	Unit	Mini...	Maxi...	Ave...	Std....
CAN1::Busload	%	0	40	37.6587	6.7737
CAN1::BusloadAvg	%	0	39.4683	37.8295	6.7972
CAN1::BusloadMax	%	0	40	38.6541	6.9463
CAN1::BusloadMin	%	0	39.0500	37.3772	6.7189
CAN1::ErrorFrameRate	fr/s	0	0	0	0
CAN1::ErrorFrames	total	0	0	0	0
CAN1::StdDataRate	fr/s	0	414	393.3626	70.6693
CAN1::TransceiverError	total	0	0	0	0

Figure 5: CAN busload: Standard truck

A fully operational forklift with all the standard components outputs a busload of approximately 40%.

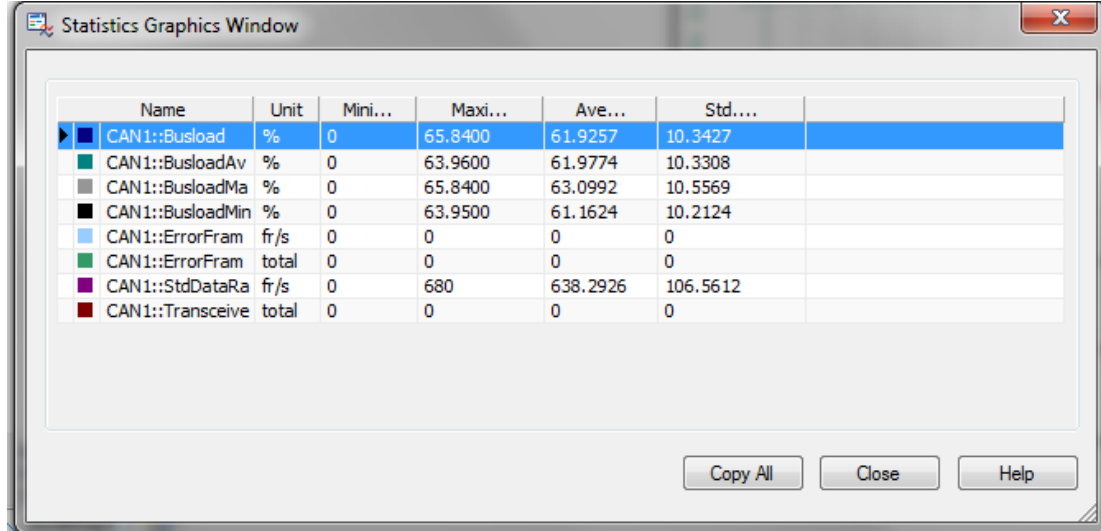


Figure 6: CAN busload: Standard truck with OCU prototype

With our OCU prototype the resulting busload is approximately 60%. Although this is with our debug interface present and this PDO interface is technically not needed. Without the debug interface the resulting load would presumably land at an average of 55%, given that each PDO interface stacks approximately 5% to the busload. This presumption is valid due to the statically driven can interface, no dynamically instanced traffic appear and thus the load is always stable.

This indicates that even with the debug interface the busload criteria is within acceptable range.

Round trip time was taken in consideration and calculated to 60ms during a worst case scenario given that busload does not exceed 100% limit.

Discussion

This chapter provides a discussion of the resulting outcome of this project and how well it copes with our initial vision of system requirements specified in the introduction. We will also discuss how well the method of work was functioning, or what could be done differently to achieve better results

Result

The fundamental part of the project was to explore the possibilities of extracting the current option handler software to an external hardware unit. We also aimed to improve the system by creating an universal and flexible system which is easy to expand and use.

The resulting prototypes introduces a foundation for the hardware and communications technologies to receive further development as well as validates the vision needed by the company to go through with it. It is hoped that the project will continue receiving development and more prototypes and finally an finished product to embed in the the main system which should be quite possible given our research.

A more testable rendition of this would be having the options in a separate hardware unit and making it listen to data existing on the CAN bus. This way, testing is done by giving it input A and verifying that output B comes out. There is no need to sprinkle in debug blocks if you can test the output from the code. Our solution was essentially to simulate an environment with an external option handler by making two hardware modules communicate over the CAN bus.

The technical challenge of this project has been very enjoyable. It ranged from identifying the problem to modelling a solution to actually implementing prototypes.

The huge scope of this project soon revealed itself and made us rethink our priorities. When this was realized we shifted our work to the more vital objectives. Unfortunately some research got misplaced because of this and ultimately was not included in the thesis. But we don't see this as a problem since the time spent supplied us with more understanding of the system and most definitely helped during development.

The greatest challenge we faced was to design the new CAN PDO interface as described in the result chapter. As the protocol for CAN communication already was available in the software, supplied by the company, we chose to use this and design the new interface by following the current protocol. With our new interface we were able to identify the vital information, needed to be transmitted, well and managed to package the signals into three PDO packets. This resulted in a fairly small addition of bus-load.

The finished prototype had all three test options active and fully functional and with plenty of user feedback and interaction the test bench validated the initial idea of an external option handler well.

Liknar spider I busload expandability

Python tool expandability

interview result disadvantages

Method

The method used in the feasibility study of the project mainly involved analysis of the current software. Since we used model based development we tried to visualize the complete system with all its components in advance. This method was very rewarding since we got a good start in the project and it helped seeing which components to start implementing first. This also helped during the early stages in the project since we could display our system approach to our mentor and get instant feedback. We chose to conduct several mini prototypes iteratively on the identified components of the system. These could be implemented alongside each other and later be assembled to the main prototype.

In order to collect successful and good results from our project, given that the project was purely implementation based, we decided to conduct an experiment. The purpose of the experiment was to validate the system by letting a system developer implement options on our prototype given a user manual to the system. The output of our prototype is somewhat abstract since the system does not act different with the external option handler but the fact that it works just as the original system is the goal. Therefore we chose to let the subject validate the system with an interview upon finishing the implementation experiment. This is actually a legitimate conclusion because if this system were to receive further development, the goal is that the task of administering options would be transferred to another department within the company. Therefore the implementation process needs to be as simple as possible.

Future work

Workload Distribution

The several different areas that have been developed during the project are dispatched between us to even out the workload. We soon indirectly specialized in different fields which increased the effectiveness of our work. This is mainly built on an open dialogue between us developers and our mentor. But both kept each other well informed and some aspects were designed collectively to keep a good overview of all the project details.

robin OCU

niklas DEBUG TESTBENCH

Conclusions

Citations

Vocabulary

Word	Meaning
ACT	Motor controller
EPS	Electric servo controller
ICH	The control unit controlling all units of the truck
SEU	Spider expansion unit, hardware with I/O
OCU	“Spider version 2.0”, I/O and beefier CPU
MCU2B	The name of the hardware we implemented our OCU on
MCU2B	Grapiical user interface

Appendix A, test options

Here are the, from The company given, options we implemented. These options lets us control a big part of the truck functions; speed, steer and hydraulics.

Turn lights

They should work like in a car. A press on option button 1 will make an output on the OCU toggle a 1 hz, which will correspond to left turn light. When the steer angle has passed a predefined value to the left, and then returned to another predefined value smaller than the first, the light should turn of. The same thing should happend when using option button 6, but with a right turn.

Lift height restriction

When lifting more than 1000 kg, it should not be possible to lift the forks above a predefined height. When pressing option button 5, an override should be possible allowing the forks to go higher. There should also be a second predefined height which you should not be able to lift above, no mather regardless of fork load. This restriction should also be possible to bypass when pressing option button 5.

Speed and steer angle restriction

If the truck load is above 2000 kg, the maximum speed should decrease linear in the range 6 to 4 km/h. When loaded 2000 kg, the max speed is 6 km/h and max 4 km/h at a load of 2500 kg or more. Similar, the maximum steer angle should be restricted linear from 90° to 70° in the same load intervall; 90° at 2000 kg and 70° at 2500 kg or more.

Horn

Pressing the horn button should sound the horn and display “horn” on the display.

drive speed reduction

Pressing option button 4 toggles slow mode, displaying SLOW on the display and limits the speed to 1.25 km/h, and slow mode is disabled by pressing option button 4 again.

hydraulic assistance

Pressing option button 2 toggles a mode where low-lifting assistance arms engages. This mode is disabled by pressing option button 4 again.

Appendix B

Questions asked to our mentor

1. **During the implementation of the test option, did you find the step by step user-manual helpful and clear enough? If not what parts were unclear?** It was clear and helpful. Only problem was that the auto generation script had to execute through the Eclipse environment which was not noted.
2. **During the implementation of the test option, did you ever feel too limited within the toolbox we specified in the user-manual? If so, what tools did you miss? E.g. signals, function primitives or others.** The options I selected were covered by the toolbox. In the future, if we were to go live with this architecture, a lot more tools would be needed.
3. **At first encounter, without knowing the embedded design of the option handler and given the user-manual, did you feel like implementing the option was too complicated or just straight forward?** For me it was very straight forward. I was up and running within minutes. I believe that even an “outsider” would find it easy to start implementing options.
4. **The current option generating script simplifies option development some, but the developer still needs basic knowledge of C-programming to fully complete the option. The next step in the automation of option development would be to fully generate the option algorithm with the script, in order to add a GUI which does not need programming skills of any sort. Do you think this is possible given the variety in complexity of the current options?** Yes, this is the ultimate goal, and I think it is feasible. By using small, primitive tools and combining them to obtain large options is definitely something I think will work.
5. **Are there any details about the script you would want to change?** Some steps (adding the function pointer and increase the NUMBER_OF_OPTION) was manual, and it would be better to automate this as well. This way, only options selected to run are called, and no risk of introducing call to null pointer.
6. **Does the prototype validate the vision of an external option handler well?** Yes, that is my opinion. At least parts of the vision. I am more confident now then I was 10 weeks ago that this is a possible way forward.
7. **Are there any details about the option implementation process that would be problematic in a real system?** Yes, especially details about functional safety; How do we allow “endless” of possibilities in modifying the behaviour of a machine without adding to the risk of causing injury? Also details about security; How do we create a modular, bus-based system without adding the risk of being hacked?
8. **Do you see this prototype and the idea receiving further in-house development?** If it were up to me, yes.
9. **Are there any other comments you would like to add about the result of the prototype?** I think you have done a great job!

First, we started with the classic “Hello world!”, by making a LED blink. From there, we began removing unneeded code and started creating a foundation for the options to build on. We quickly realized we needed a way to communicate with the MCU2B. The company have CAN

bus interfaces with competent software, but they cost a lot, and there is a limited access to them, as they are used by other developers. This lead to us creating our own CAN bus software.

For hardware, we used EMS CPC-USB, which sells for less than €200. Thanks to Volkswagen Research¹⁷, the Linux kernel has support for CAN. This made it possible for us to build a relatively complete test bench simulating the truck with a small Python application.

With the test bench, we were able to build a prototype working good enough to function in a live truck with almost no modifications.

¹⁷<https://www.kernel.org/doc/Documentation/networking/can.txt>