Options handling

Niklas Blomqvist, Robin Gustafsson

Contents

Revisions

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

Abstract

Unique customizations (options) of a fork lift trucks features are often requested by customers. When new options is 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 fork lift manufacturer located in Östergötland as a part of our Bachelor degree in Computer Science. In this report the fork lift manufacturer will be called "The company".

Motivation

A large quantity of the sold fork lifts are 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 decouple the options implementation from the main firmware and dedicate a separate unit to handle the options in order to speed up development of new features, and decrease the number of potential bugs in the main firmware. By doing this we achieve a more modular system.

Problem

- What needs to be taken in consideration when designing the options controller?
- How is the CAN bus affected if additional controllers are added as it runs at 125 kbit per second.
- 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 fork lifts

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 SSU[1] and BCU[2], but it could also be internet connectivty 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.

[1] Shock Sensor Unit
[2] Battery 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:

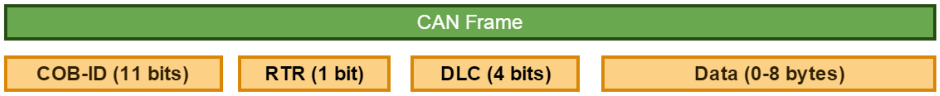


Figure 1: CAN frame

The communication object identifier, COB, consists of 11 bits of data, where the frame with the lowest ID value 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 are 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 realtively 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.

Worst case CAN latency

The PDOs are sent with a 20 ms interval making the worst case roundtrip 60 ms.

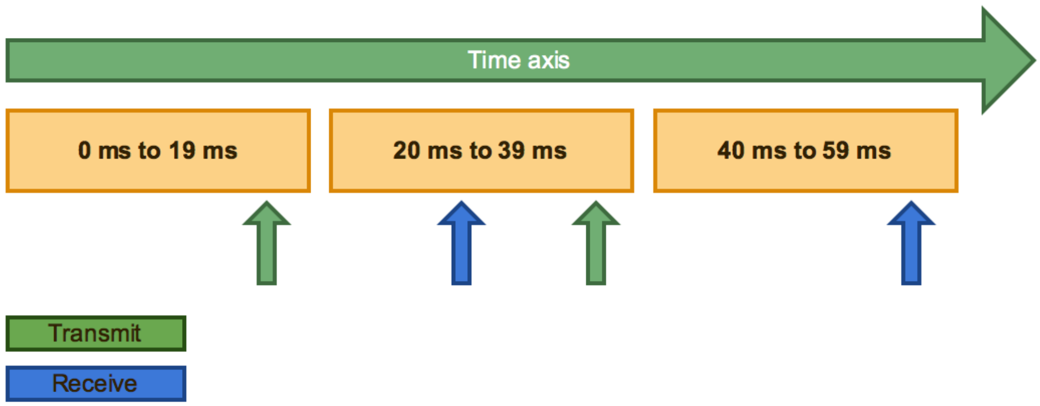


Figure 2: Illustration of worst case PDO latency

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 in negligible as it normally lags behind only a couple of milliseconds.

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 bus loads far above 100 %[3].

Method

In this chapter, we describe 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.

[3]A 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 indentified 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.

The first fundamental 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 avilable 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.

Test option 1, 2 and 3[4] 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.

Result

Feasibility study

Implementation of prototype

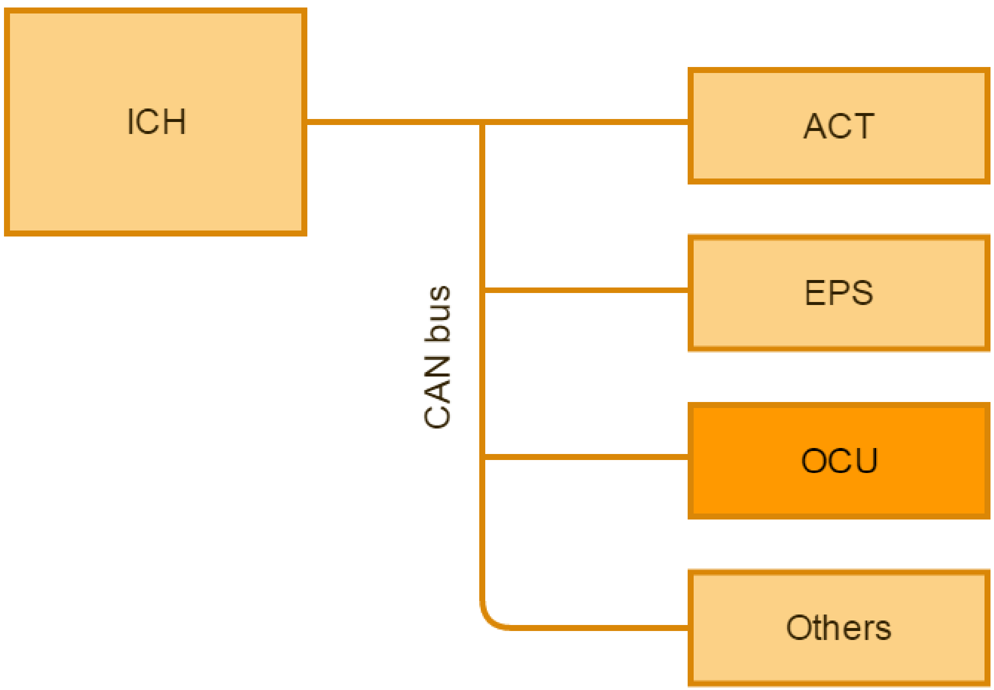
Nedan text ska flyttas runt!

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.

[4] See Appendix A

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 watchdog[5] 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.

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.



The current option handler as it sits has direct access to alot of the internal hardware- and software functionality of the ICH. Thus the individual options has alot of freedom towards modifying the truck functionality. If we are to migrate the option handler to an external unit it is therefore important to keep the current level of access 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 thesefrom 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

[5]Hardware timer which restarts the system if the timer itself has not been restarted

the system would decrese. We only considered adding traffic if it was absolute necessary. The signals already present at the CAN-bus was 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 ticks[6] each 1.25 ms. In the MCU2B, the kernel ticks at 1 Hz. In both cases, the main application loop is called once each 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.

```
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.

[6] Generates interrupt in which kernel and application functions are called

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 package on the OCU, called PDO RX1.

In order to restrict or request drive/hydraulic/steer functions from the OCU to the ICH one additional PDO is added. This PDO is for the more operational demands wich the OCU needs to be able to instruct the ICH. These operation primitives are frequently used by the individual options. As an example a option may want to restrict the drive speed. This PDO is the first of two transmit PDOs needed by the OCU, called PDO tx1.

A third PDO is required in order to control the display and the surrounding LEDs located in the fork lift handle. The reason for needing a whole PDO just for the display is simply that the four didgit display needs a set of four bytes for receiving characters (1 `char` = 1 byte). This PDO is the second transmit PDO called PDO tx2.

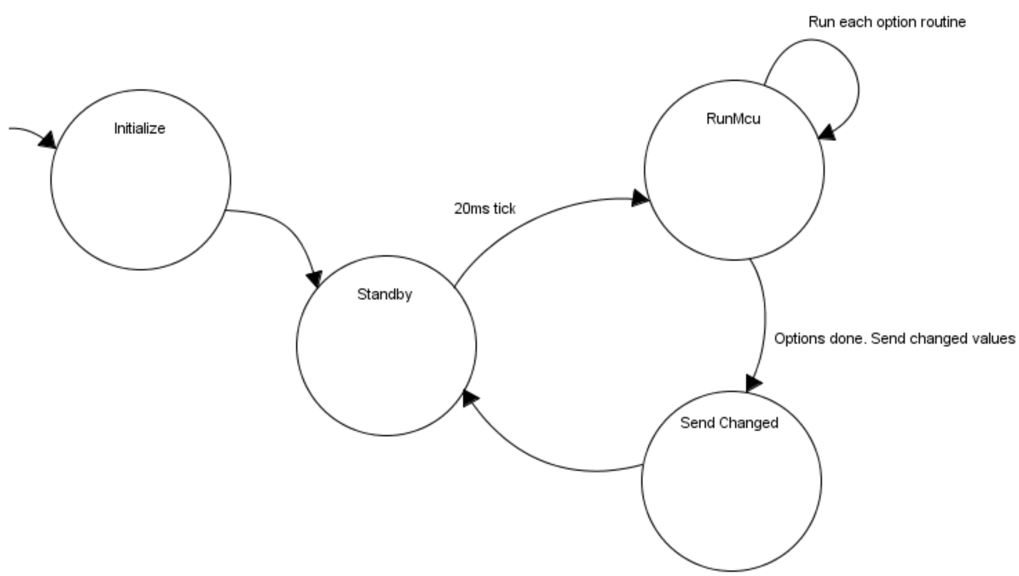


Figure 3: OCU

BILD på mcu2b?

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 wich are required by the ICH if running non-benchbuild. If external modules are missing the ICH goes into aserted 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.

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.

Evaluation

Evaluation occured somewhat successivley 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-options[7] to easier identify limitations of the prototype. The grand finale of the evaluation process had all the test-options active on a real truck. Results were mostly based on this phase.

[7]See Appendix A

Result

Feasibility study

Implementation of prototype

Evaluation

Discussion

Result

Method

Future work

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

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.

Appendix B

Sådant vi ska lägga in på ett bra ställe

Today, The company handles a big quantity of customer specific options on their fork lifts. The options are all being built in into the main controller (MCU) of the truck. This leads to problems:

- Developing the options requires a lot of resources, this because it began as a "one-off job".
- Because all of the features exists in the main firmware, the code becomes very complex and hard to follow. Many of the features are also inactivated for most of the customers.
- Many options have different parameters for different customers and trucks.
- The available code memory will soon be filled. Adding additional code will require a larger on chip memory.
- It is not possible to easily handle over the development of customer options to a another department, as it requires vast knowledge of the firmware.

The company is looking for an options handling solution which allows the functionality to be moved from the MCU to a separate controller.

We will work out a solution where the options handling will allow development of functionality, independent of the main firmware, in a more modular fashion. This will allow parameter based configuration without the need of rewriting code. It is important that all existing options are handled properly. This will imply a reduction in time needed to develop new features.

A graphical interface is desired in order to simplify the administration of options without the need of deep programming knowledge. A *PLC representation* would give the user a good overview of active options and also the possibility to customize parameters. It is also important that the options handling are secure in a way that ensures that no unauthorized person can tamper with it.

Currently there is no significant market value in the options handling it self, but in the long term there will be. Both in time savings as well as in product quality. This will give The company a better foundation to decide on the possibility to include this in the control units of the trucks.

During the thesis work, we utilized an iterative software development model. We started out with the given hardware (named MCU2B) which, compared to the ICH, is very powerfull. The hardware also contains various I/O, some capable of driving high currents.

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 Research[8], 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.

[8] https://www.kernel.org/doc/Documentation/networking/can.txt