Distribution Ärende/Subject

DME Documentation of the OCU

**Revision history**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Revision* | *Date* | *Issued by* | *Sections* | *Comments* |
| 0.1 | 2015-04-23 | Robin Gustafsson, Niklas Blomqvist |  | First draft |
| 0.2 | 2015-04-29 | RG, NB | Added information to section 2. Added section 3 and 4. Old section 3 and 4 are now section 5 and 6 | Second draft |
| 0.3 | 2015-05-05 | Niklas Blomqvist | 2, 3 | Added information about the OCU and the truck data simulator. |
|  |  |  |  |  |
|  |  |  |  |  |

**Table of contents**

[1 Introduction 3](#_Toc418620728)

[1.1 Glossary 3](#_Toc418620729)

[2 The OCU 3](#_Toc418620730)

[2.1 CAN 3](#_Toc418620731)

[2.2 PDO Interface 3](#_Toc418620732)

[2.3 Additional PDO 3](#_Toc418620733)

[2.4 System Model 3](#_Toc418620734)

[2.5 Initialize 3](#_Toc418620735)

[2.6 RunMcu 4](#_Toc418620736)

[2.7 Send Changed 4](#_Toc418620737)

[2.8 Handling requests on the ICH 4](#_Toc418620738)

[2.9 Option hierarchy 4](#_Toc418620739)

[3 Extra: CAN-Debug tool (truck data simulator) 4](#_Toc418620740)

[4 User manual 6](#_Toc418620741)

[4.1 Generate option stub 6](#_Toc418620742)

[4.2 Create option stub manually 6](#_Toc418620743)

[4.3 Option requirements 6](#_Toc418620744)

[5 Toolbox 7](#_Toc418620745)

# Introduction

This document describes the concept of the OCU, and how to implement new options.

## Glossary

|  |  |
| --- | --- |
| Term | Explanation |
| ACC | AC combi regulator, motor controller. |
| ACT | AC traction regulator, motor controller. |
| ICH | Integrated Control Handle (master) |
| MCU | Main Control Unit (master) |
| OCU | Option Control Unit |
| TBD | To Be Defined |

# The OCU

The OCU is meant to be used in scenarios where there is a specific demand for a truck feature customization and where the main truck software is to be left untouched.

## CAN

At the moment the OCU adds a total of three (3) PDOs when configured to full truck interaction. The address of the OCU is 0x1B (27 decimal).

The OCU only requires one additional PDO to be sent from the ICH. This PDO includes signals required by the OCU to operate, E.g. Option button bitfield among others. In order to restrict or request drive/hydraulic/steer functions from the OCU to the ICH one additional PDO is added. A third PDO is required in order to control the display and the surrounding LEDs.

## PDO Interface

The interface and protocols for transmission on the three PDOs are specified in the document *OptionCANopen\_mapping.docx*

## Additional PDO

The OCU can be configured to listen to the CAN bus for messages already present at the bus. This may be extended in order to listen to additional units (SSU, BCU, etc.). As of now, the ACT and EPS listeners are active but needs to be polished. E.g. a nice interface to easily extract data with is to be implemented. At the moment the EPS listener extracts the steering potentiometer angle and the ACT listener extracts the drive direction and speed. These types of signals were already present at the CAN thus instancing these signals in new PDO would mean a loss in system response.

These listeners are represented in *sm\_xxx*, where *xxx* stands for e.g. act, eps, etc.

## System Model

In this section we will present and describe the initial system model.

The initialize routine is instanced from the main software in the same way as the other units. The OCU software routines run each 20ms and the general software flow is the following:

* Receive updated system signals from CAN bus
* RunMcu (Execute all option unique routines)
* Send Changed on PDOtx1 & PDOtx2

The receive state can be misleading since no receive call is instanced. The reason for this is that the PDOrx1 interface is predefined and data updates to the memory is handled automatically.

## Initialize

The internal storage of options is dealt with in an object oriented manner in order to keep the structure as organized as possible. This will also aid in the event of further development as the structure can be extended without the need to rewrite all of the existing code. 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 established here as well. This function is called upon by the standard set-up routine. The OCU will not instance the receiving of PDO-messages, this is handled elsewhere and the received data can be directly used in the assigned register.

## RunMcu

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.

## Send Changed

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. 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.

## Handling requests on the ICH

The ICH side of the system is currently 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 since sub systems of the truck were missing. E.g. the steering potentiometer was not connected to the handle, thus was not implemented fully. 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.

## Option hierarchy

An option is to be created in the g*oldwing/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*.

A solution where a constant in the created options header file, e.g. *OPTION\_IS\_ACTIVATED* set to *TRUE*, is used by a script to populate the OptionsArray in *mcu.c* would be an improvement.

# Extra: CAN-Debug tool (truck data simulator)

We developed the OCU using a custom made fork lift truck simulator. As we did not have unlimited access to the very much expensive CANalyzer CAN interface, we created our own using the much cheaper CPC-USB CAN interface. The simulator is written in Python and is multi-threaded. It is acceptably easy to implement new PDOs by extending the main PDO class in the code (named CANibal).

In the application we rely on SocketCAN developed by VW, which was later contributed to the Linux kernel, which makes communication over CAN bus possible almost out of the box.

To read data from the CAN bus, we use the (in SocketCAN) included tool called *candump*. This makes our application quite resource hungry, as text is piped in to our server daemon, and from there relayed (using sockets) to our GUI. In the GUI, the messages are dispatched to the GUI components through a built in message queue – which also increase the overhead some.

The data is transmitted from the interface to the server via sockets. From the server, we use a small binary called *cdump* which takes a CANOpen ID and the data to be transmitted as argument. *Cdump* will automatically set the correct DLC, and ship the data over *can0*.

The prerequisites for the interface and the server is Linux with SocketCAN, python-2.7 with python-tk-inter and gevent. This will require python-dev (on most Linux distributions) to be installed.

Correctly configured, the simulator allows us to simulate all of the truck components from a standard PC, just as with CANalyzer.

To start, run *can.sh* followed by *crank\_server.sh* followed by *interface.py*.

# User manual

## Generate option stub

You can have the options c and h files created automatically for you by running the script *generate\_option\_template.py* located in the *sm\_options* folder inside the g*oldwing* folder.

The script will ask for the name of the option to be created and generates the files from it, according to the BT Goldwing template.

## Create option stub manually

To create the option files manually, create the options h and c files in the sm\_options folder: <*your\_option.c/h*>. In the *your\_option.c* file, make sure to add #include “goldwing.h”.

You do not need to include *your\_option.h*, as this is taken care of by g*oldwing.h*.

## Option requirements

In order for the options to function, you must meet the following criteria:

* The name of the option routine function should be identical to the file name (except for .c/.h)  
  E.g. if the file is named warning\_lights.c/h the main function of the option should be called warning\_lights
* Your .c-file must include g*oldwing.h*
* In your .h-file, be sure to declare your main function of the option under *GW\_PRIVATE*, as it must be accessible from other parts of the system

The script *generate\_option\_template.py* does all this for you and is the recommened method.

To add an option to the *run queue*, you add the function pointer to *OptionArray* defined in *goldwing/sm\_mcu/mcu.h/c* file. Increase the *NUMBER\_OF\_OPTION* constant in *mcu.h* and add the function pointer to the *OptionArray[n].run*-member.

All should now be configured to start implementing the option. All the tools made available by the ICH are listed in section 5. Use these to access signals and function calls. In the case of a vital signal needed by your option not present in the list, check if the signal is already available on the CAN-bus. These signals can be acquired by the sniffing-interface (the signals can occur in un-scaled format) without any penalty in CAN-traffic.

**Remember to *make clean* if an option routine is removed manually.**

# Toolbox

|  |  |
| --- | --- |
| Function | Description |
| UWord **get\_currWeight**(**void**); | Gets the current weight on forks in mV |
| Bool **get\_height1**(**void**); | Returns TRUE if forks are higher than height sensor 1 |
| Bool **get\_height2**(**void**); | Returns TRUE if forks are higher than height sensor 2 |
| SWord **get\_currSpeed**(**void**); | Gets current speed in engine rpm |
| SWord **get\_currSteerAngle**(**void**); | Gets current steer angle as analog value |
| SWord **get\_bflyRamped**(**void**); | Gets Butterfly value |
| SByte **get\_adLift1**(**void**); | Gets adlift value |
| SByte **get\_adLift2**(**void**); | Gets adlift value |
| UByte **get\_digitalButtonBitfield**(**void**); | Returns bitfield with pressed digital buttons. 1 indicates button press |
| UByte **get\_optionButtonBitfield**(**void**); | Returns bitfield with pressed option buttons. 1 indicates button press |
| **void** **write\_display**(UByte, UByte, UByte, UByte, UByte); | Make a write to display call to ICH. If first byte is zero, ICH takes control of display. Use constants defined in jura\_display\_const.h (included in goldwing.h) |
| **void** **restrict\_hydraulic**(UByte); | Restrict hydraulic function |
| **void** **request\_hydraulic**(UByte, SByte); | Request hydraulic function |
| **void** **restrict\_steer**(SWord); | Restrict to given steering angle |
| **void** **request\_steer**(SWord); | Request given steering angle |
| **void** **restrict\_drive**(SByte); | Restrict to given drive speed |
| **void** **request\_drive**(SByte); | Request given drive speed |
| **void** **request\_power**(**void**); | Request power (main contactor) |
| tCanGoldwingPdo\* **get\_CanMsg**(UByte CanIndex); | Used to get a can object to manipulate freely. Use with caution. |