|  |  |
| --- | --- |
| 199.jpg | HANTUNE |
|  | BitsChipsRC30Target_HANtune.jpg |
| Version 1.0  11/12/2012 | Error handling, Technical design |
|  | This document contains the technical design for the implementation of visualizing errors with HANtune.  By Scott Wagemakers |

Copyright (c) 2020 [HAN University of Applied Sciences]

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

HANTUNE

Error handling, Technical design

Contents

[Introduction 3](#_Toc340500515)

[1 Schematic view 4](#_Toc340500516)

[2 formats 6](#_Toc340500517)

[2.1 Error struct 6](#_Toc340500520)

[2.2 Error list 6](#_Toc340500521)

[3 Starting process 7](#_Toc340500522)

[3.1 Sequence diagram 7](#_Toc340500524)

[3.2 Function definition 7](#_Toc340500525)

[4 Reading proces 8](#_Toc340500526)

[4.1 Sequence diagram 8](#_Toc340500528)

[4.2 Function definition 10](#_Toc340500529)

[5 Summary all processes 12](#_Toc340500530)

[6 HANtune 13](#_Toc340500531)

[6.1 Sequence diagram (protocol) 13](#_Toc340500534)

[6.2 Proposal lay-out 14](#_Toc340500535)

[7 Other hardware platforms 15](#_Toc340500536)

[7.1 Schematic view 15](#_Toc340500538)

[7.2 Formats 16](#_Toc340500539)

[7.3 Processes 17](#_Toc340500540)

[8 Test Cases 18](#_Toc340500541)

[8.1 Test cases A&B 18](#_Toc340500543)

[8.2 Test case C 19](#_Toc340500544)

# Introduction

This document contains the designs needed for developing error handling in HANtune.

# Schematic view



#### Error list buffer

Located on a pre-defined location in RAM (needed for XCP). It contains all occurred errors and is the complete list read by HANtune. Containing a size parameter, a semaphore to control its use, a flag to check for interruption of the HANtune reading of the buffer and all errors. The size will depend on the usable RAM size of the hardware platform.

#### HANtune

Capable of reading and deleting of the complete error list.

Simulink model

Blocks in the model which are capable of error related actions (i.e. read/write/delete).

#### API

The programming environment of the Bodas RC30 series defines complete methods for usage of supported ECU’s, one method being error handling.

#### Stored errors / EEPROM

Is a complete list of all errors occurred since the usage of the controller or resetting of the list.

#### Active errors / RAM mirror

Is the actively used list for errors, being updated for every error event.

#### Internal error handling

The Bodas RC30 series have the ability to report internal errors (like short circuits on input), using the same method as user related error can be used.

#### Buffer sync

Since errors are already stored in RAM, the buffer sync function will copy this list (active or stored errors, depending on parameter) to the error list buffer.

# formats



## Error struct

The error code format from the format (named “ErrorData\_ts”, found in “api\_lib\_basic.h”) used in the “Bodas RC30” API, it contains;

|  |  |  |  |
| --- | --- | --- | --- |
| **Field nr** | **Field name** | **Data type/size** | **Function** |
| *1* | *Code\_u16* | unsigned int, 16 bit | Unique identifier. |
| *2* | *Param\_u8* | unsigned int, 8 bit | Freely usable (i.e. severity). |
| *3* | *Occurrence\_u8* | unsigned int, 8 bit | Nr of occurrences. |
| *4* | *Timestamp\_u32* | unsigned int, 32 bit | Time of last occurrence. |

## Error list

Needs to start at a pre-defined memory address in order to be obtained through XCP. To ensure compatibility with multiple hardware platforms, the array is written in pages of 4 bytes (32 bit). The number of maximum errors is determent per hardware platform. The error list is constructed from a struct array, where the struct is defined by the RC30 API. The first row, even tough still the same error format, is used to store info. The following data is stored;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ***Page A*** | | | ***Page B*** |
| *Field nr* | *1* | *2* | *3* | *4* |
| *First row* | reserved | controlFlag | semaphore | size |
| *Next rows* | ID | parameter | occurrence | timestamp |

#### controlFlag

Use to check for interruption of reading by HANtune;

1 = HANtune is busy reading.

0 = flag has been reset by a function, HANtune has ended reading or by resetting the ECU.

#### semaphore

Used to indicated the buffer is in use and should not be used until available again.

1 = buffer is available.

0 = buffer is in use.

#### Size

The current number of contained errors in the buffer.

# Starting process

In order to read by the means of XCP, the address where the error list is located must be available. To start using error handling in HANtune, the base address of the error list is retrieved during initialization of XCP.



## Sequence diagram



## Function definition

|  |  |
| --- | --- |
| **Name** | getErrorListAddress |
| **In** | *void* |
| **Out** | *uint8\** (startAddress) |
| **Function** | Returns the start address of the error buffer (row 0) |

# Reading proces



## Sequence diagram



#### HANtune

Sends a user command (freely definable command in XCP) to request the preparation of the error data. Then it waits for the XCP handler to reply. When received, it start with setting the start address and then reads the first row. After computing the info in the first row, it reads the error codes in a while loop, stopping when the array size is reached. The last action is to send the user command to release the error buffer.

#### XCP handler

Consists of two parts; the standard XCP code and the custom user command handler, called by the standard XCP code.

When the user command is received, the buffer sync is called. Then the standard XCP code is used to retrieve data from memory addresses. Afterwards when the user command is received again, the function to release the buffer is called.

#### Error list

Consist of functions (like buffer sync) and the array (placed on a pre-defined memory address), containing 1 row with parameters and the rest for errors.

When buffer sync is called the semaphore is set to 1 to indicate the use of the buffer, the controlFlag is set and all pending changes are applied to the error list buffer.

Afterwards the parameters are reset when the release buffer function is called.

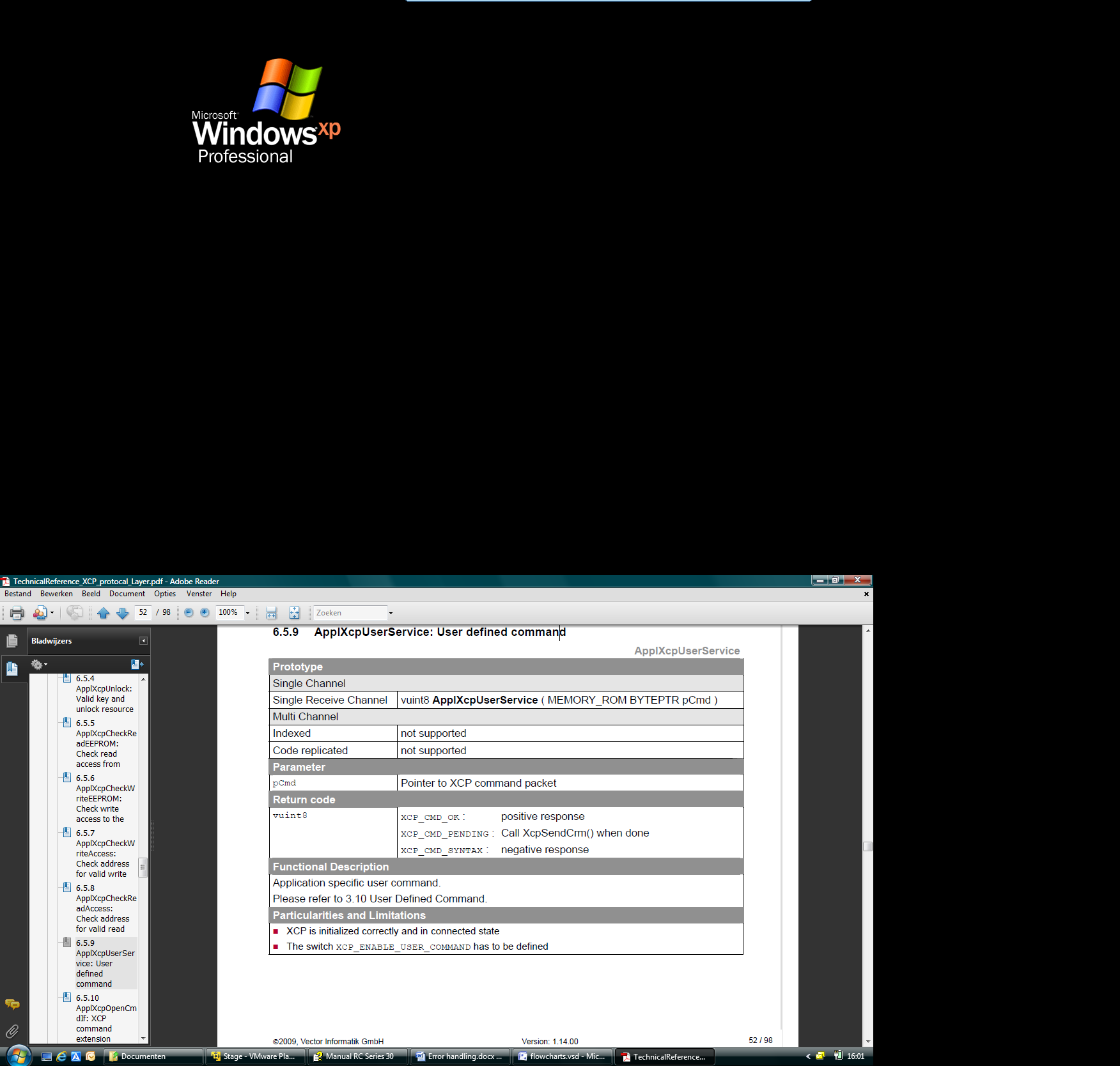
## Function definition





#### XCP user command handler

From the document “TechnicalReference\_XCP\_protocal\_Layer.pdf”;



#### Other functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | | | bufferSync |
| **In** | | | *uint8* parameter |
|  | ***Nr*** | | *Function* |
|  | *0* | | Read active errors |
|  | *1* | | Read stored errors |
|  | *2* | | Delete active errors (form RAM buffer) |
|  | *3* | | Delete stored errors (form EEPROM) |
| **Out** | |  | *void* |
| **Function** | | | Synchronize 2 error arrays, set semaphore & controlFlag |

|  |  |
| --- | --- |
| **Name** | bufferRelease |
| **In** | *void* |
| **Out** | *bool* (bufferOK) |
| **Function** | Reset semaphore & controlFlag |

# Summary all processes

#### Writing to error list buffer

To prevent errors changing while HANtune is reading the list, only one process can access the buffer at once. The process wanting to read/write can only do so when the semaphore is 1, it then sets the semaphore on 0 as long as it is busy with the buffer.

All processes, besides HANtune, writing to the buffer will set the controlflag to 0. Since HANtune is the only process (including rebooting the ECU) that set the controlflag to 1, reading a value of 0 from the controlFlag after reading means a other process has interrupted and the read values are possibly inaccurate (and therefore are discarded).

#### Reading aN error list

The entire array of errors is read as [ while current row ≤ size parameter ].

#### Deleting An error list

Emptying an error list is done by setting its size parameter to 0. The actual data in the list remains unchanged and is later on overwritten by new data.

#### Buffer overflow

When a new error would overflow the (temporary) error list [ size parameter < max errors ] the error is discarded and the last error in the list is overridden with an error. In case of the error list buffer when the last error is error ID 0xFFFE, the error before it is overwritten.

The errors for buffer overflow are;

|  |  |
| --- | --- |
| **Error ID** | **written when** |
| 0xFFFD | temporary error list buffer overflow |
| 0xFFFE | error list buffer overflow |

# HANtune



## Sequence diagram (protocol)



## Proposal lay-out



The standard lay-out is extended with a extra tab to the left. This tab contains the error handling library. The library contains elements for all supported hardware platforms

(like the Bodas RC30 seen in the picture). The hardware platforms might be subdivided into different error types (like active and stored for the Bodas RC30). Each library contains parameters and signals, with the same properties as the user defined data of the “Project data” tab (multiViewer, booleanViewer, etc). The signals and parameters lists might contain signals like; error data, a parameter for selecting the currently viewed error and a help file viewer (which shows the help file associated with the current error).

Error handling and user-defined parameters and signals can be used on the same lay-out tab.

# Other hardware platforms



## Schematic view



#### Other error source(s)

A error handling function programmed in the ECU to deal with internal errors. It might not be present on certain hardware platforms.

#### Simulink model

Blocks in the Simulink model which are capable of error related actions

(i.e. read/write/delete).

#### Temporary error list

Located in RAM (no pre-defined address needed), it stores occurring errors when the error list buffer is used by HANtune. It contains a size parameter, delete parameter and the actual errors with desired action flag. It might contain room for only a few errors, depending on the usable RAM size of the hardware platform.

#### Buffer sync

Either copies errors from the temporary error list to / deletes errors from the error list or deletes the error list if the delete parameter is set.

## Formats

#### Temporary error list

Because the temporary error list is just for internal use, its memory address is irrelevant. The array will use the same error format as the error list buffer (except for the first ‘info’ row) with an added flag to store the needed action (store/delete). Beside the array, other info is stored in loose variables;

|  |  |  |
| --- | --- | --- |
| **Field** | **Data type/size** | **Function** |
| *code* | unsigned int, 16 bit | Unique identifier. |
| *param* | unsigned int, 8 bit | Freely usable (i.e. severity). |
| *occurrence* | unsigned int, 8 bit | Nr of occurrences. |
| *timestamp* | unsigned int, 32 bit | Time of last occurrence. |
| *actionFlag* | unsigned int, 8 bit | Desired action of error, 0 = store 1 = delete |
|  |  |  |
| **Variable** | **Type** | **Function** |
| *deleteFlag* | boolean | Indicates (being 1) the order to delete the array list buffer before synchronizing it with the temporary buffer. |
| *size* | unsigned int, 32 bit | The number of new errors (0 when no errors are unsynchronized). |

## Processes



# Test Cases

Because the HANtune implementation of the error handling will be done after the implementation of the hardware part, these test cases are for later use.

Test case C is used to confirm the functioning of the hardware implementation, it is used while/after prototyping.



## Test cases A&B

The next configurations will be used for case A;

• The ECU will run the “DemoProject V0.6.0” model, adjusted to store errors defined by HANtune-parameters.

• The ECU will be read by HANtune with the signals and parameters configuration from the HANtune-project provided with the “DemoProject V0.6.0” package, adjusted to set errors for testing.

• HANtune will run at the default rate of 10 Hz.

• Requesting the error list will be done at a rate of 1 Hz and 10 Hz.

• Setting 20 different errors.

• Missing or delayed reactions from the hardware will be reported.

Additionally to case A, when the initial tests are successful, the amount of errors will be increased until ‘breaking point’, this point will be reported.

Case B will exists of the following configurations;

• The ECU will run a custom, stripped model, it will include no more than 5 HANtune-signals and parameters (the CAN bus can’t handle much more at a rate of 1000 Hz).

• A custom HANtune-project will be build.

• HANtune will run at a rate of 1000 Hz.

• Requesting the error list will be done at a rate of 1 Hz.

• Setting 5 different errors.

• Missing or delayed reactions from the hardware will be reported.

#### Acceptable results

For both cases, not retrieving the error list in 10% of all attempts is considered acceptable, but only when the loss of data can be made noticeable in HANtune. Steps following the failure of the tests will be considered, when needed, afterwards.

## Test case C

Using PCAN-view, the sequence diagram from paragraph 6.1; “Sequence diagram (protocol)” will be executed (where PCAN-view replaces HANtune) and the responses will be checked against these diagrams. The resulting trace file will be saved, together with a screen shot from a HANtune project showing the read error as signals.