FUTURE TUNING PROCESS FOR EMBEDDED CONTROL SYSTEMS

Muhammad Arsalan

This thesis is presented as part of Degree of Master of Science in Electrical Engineering

Blekinge Institute of Technology
September 2009
Future tuning process for embedded control systems.

Master’s Thesis

Muhammad Arsalan
Abstract

This master’s thesis concerns development of embedded control systems. Development process for embedded control systems involves several steps, such as control design, rapid prototyping, fixed-point implementation and hardware-in-the-loop-simulations.

Another step, which Volvo is not currently using within climate control is on-line tuning. One reason for not using this technique today is that the available tools for this task (ATI Vision, INCA from ETAS or CalDesk from dSPACE) do not handle parameter dependencies in a satisfactory way. With these constraints of today, it is not possible to use online tuning and controller development process is more laborious and time consuming.

The main task of this thesis is to solve the problem with parameter dependencies and to make online tuning possible.
Acknowledgment

I would first like to thank my supervisor Björn Mårdberg (Climate Area Manager) at Volvo Technology. The door to his office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work, but steered me in the right the direction whenever he thought I needed it.

I offer my sincerest gratitude to my thesis supervisor at Blekinge Institute of Technology, Anders Hultgren, who has supported me throughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way.

I would like to thank Dr. Mats Andersson for providing me an opportunity to work in a very professional environment and making sure that I get all required tools to do my work without any delay.

I would also like to thank Martin Hiller, Björn Fridholm, Vincent Andreasson and Christian Axbrink for their help, support and very important suggestions about my work. I have learned a lot from them. Roger Carlsson has helped me a lot with ATI VISION interfacing, without his help this work could have been very difficult for me. I am cordially thankful to Roger.

Beyond my work, Markus Ottosson, a very nice colleague and a very good friend introduced me to very famous Swedish sport “Innebandy”. It was very nice experience to play Innebandy with very friendly colleagues although I was always the first one to get out of breath and tired but that was fun. So I want to thank them all, especially Markus Ottosson, David Berg, Henrik Weiefors and Stefan Börjesson.

Lastly, and most importantly, I wish to thank my parents. They bore me, raised me, supported me, taught me, and loved me. To them I dedicate this thesis.
# Table of Contents

1 INTRODUCTION ................................................................. 11

1.1 Background ........................................................................... 11
1.2 Goals and objectives ............................................................. 13

2 BACKGROUND ....................................................................... 15

2.1 EMBEDDED SYSTEMS ......................................................... 15
  2.1.1 History of Embedded Systems ........................................ 15
  2.1.2 Common Characteristics .............................................. 16

2.2 Model Based Design ............................................................ 17

2.3 ECU DEVELOPMENT ........................................................ 18
  2.3.1 Conventional Approach for ECU Development .................. 18
  2.3.2 Modern ECU Development ......................................... 19

2.4 Universal measurement and Calibration Protocol (XCP) .............. 20

3 PROBLEM INVESTIGATION .................................................. 23

3.1 Complete process for developing embedded control systems ....... 23
3.2 Parameter Dependency ........................................................ 25
3.3 Reasons for introducing parameter dependencies ..................... 26
3.4 Statistics about parameter dependency .................................... 26
3.5 Parameter dependency problem in development process ............. 27
3.6 Effect of parameter dependencies on development process ........... 30

4 POSSIBLE SOLUTIONS ...................................................... 33

4.1 Parameter dependencies and MATLAB .................................. 33
4.2 Parameter dependencies and TargetLink .................................. 34
4.3 Parameter dependencies and C-code ....................................... 35
4.4 Parameter dependencies and Calibration tool ............................ 35
4.5 Separate windows application

5 SELECTED SOLUTION

  5.1 Reasons for selecting this solution
  5.2 Overview of solution
  5.3 Required Softwares
  5.4 Project file
  5.5 Parser
    5.5.1 Comments Removal
    5.5.2 Multiple line parameter definitions to single line
    5.5.3 Separating parameters with dependencies
    5.5.4 Position of dependencies in a parameter definition
    5.5.5 C-code parsing
    5.5.6 Replacing parameter names
  5.6 Calibrator
    5.6.1 Parameter values from VISION to MATLAB
    5.6.2 Parameter values from MATLAB to VISION
  5.7 Calibration of non-calibratable parameters
  5.8 Dependency Calibrator in a Nutshell

6 RESULTS

  6.1 Verification

7 CONCLUSION

8 FUTURE WORK

REFERENCES

APPENDIX

Calibration tools
# Terms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2L</td>
<td>ASAM MCD-2MC Language</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface. An interface for applications, operating systems, or hardware to access functionality.</td>
</tr>
<tr>
<td>ASAM</td>
<td>Association for Standardization of Automation and Measuring systems</td>
</tr>
<tr>
<td>CAL</td>
<td>Variable class type in TargetLink Data Dictionary. Calibration Data</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network. Serial communication with emphasis on reliable transmission. Used in vehicles.</td>
</tr>
<tr>
<td>CANH</td>
<td>CAN High. One of the two wires used to send data in the CAN bus</td>
</tr>
<tr>
<td>CANL</td>
<td>CAN Low. The second of two wires used to send data on the CAN bus</td>
</tr>
<tr>
<td>CCM</td>
<td>Climate Control Module</td>
</tr>
<tr>
<td>CCP</td>
<td>CAN Calibration Protocol</td>
</tr>
<tr>
<td>DD</td>
<td>Data Dictionary</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>MBD</td>
<td>Model Based Design</td>
</tr>
<tr>
<td>MC</td>
<td>Measurement and Calibration</td>
</tr>
<tr>
<td>SAE</td>
<td>Society of Automotive Engineers</td>
</tr>
<tr>
<td>Vbf</td>
<td>Volvo Binary Format</td>
</tr>
<tr>
<td>XCP</td>
<td>Universal Measurement and Calibration Protocol</td>
</tr>
</tbody>
</table>
1

INTRODUCTION

1.1 Background

Volvo technology (VTEC) is an innovation company that provides expert functions and develops new technology for “hard” as well as “soft” products within the transport and vehicle industry. Among other things VTEC is working with embedded control systems. For one of the embedded control systems particularly “Climate Control Module (CCM)”, VTEC is working with the whole chain. VTEC does this for Volvo Cars, Volvo Trucks, Volvo Construction Equipment, Renault Trucks and Land Rover.

The work process for embedded control system development is typically as follows:

→ Control Design
→ Rapid Control Prototyping
→ Fixed-Point Implementation
→ Hardwar-In-the-Loop Simulation
→ Online Tuning.

It is an iterative process, but there is one problem for the last step, which limits the possibilities of working iteratively. Control design is typically made in MATLAB/Simulink and Fixed-Point implementation is typically made with a tool such as TargetLink. During these steps the parameters may be handled in an m-file. When going to the on-line tuning step however, the parameters are handled in a tool such as ATI Vision, INCA or CalDesk. Once you have taken this step the connection to the m-file is lost. Therefore the last step is somewhat of a one-way step. It is not completely impossible to go back to the earlier steps in the development chain, but the iterative process is not well supported by available on-line tuning tools of today.
The following m-script instructions are examples of parameter dependencies that will cause the mentioned problems:

```
Heating = [-100, -20, 0, 20, 100];
BlowerHt = [ 12, 5, 4, 5, 10 ];
Blower_min = min[ BlowerHt];
Defrosting = [ 0, 20, 100 ];
BlowerDef = [ Blower_min, Blower_min, 10 ];
```

Using the above vectors in interpolation tables, one table with Heating as input vector and BlowerHt as output vector and another table with Defrosting as input vector and BlowerDef as output vector would cause problems during on-line tuning process. Three of the elements are meant to have identical values, but the tools, as it is today would allow them to be tuned individually. This is just one of many constructs, which may be very useful as long as you are in the MATLAB environment but causes problems during the on-line tuning process.
1.2 Goals and objectives

The main goals of this master’s thesis are:

- To investigate the problem of parameter dependencies.
- To find possible solutions.
- To make online tuning possible for dependency parameters in the development process of embedded control systems.
2 BACKGROUND

2.1 EMBEDDED SYSTEMS

2.1.1 History of Embedded Systems
In the era of earliest development of computers i.e. 1930-40s, generally computers were capable of doing a single task. Over time with the advancement in technology, traditional electromechanical sequencers presented the concept of programmable controllers using solid state devices.
“One of the first recognizably modern embedded systems was the Apollo Guidance Computer, developed by Charles Stark Draper at the MIT Instrumentation Laboratory.”[1]
After the early applications in 1960, the prices of embedded systems have come down and their processing power has been increased dramatically. A standard for programmable microcontrollers was released in 1978 by National Engineering Manufacturing Association. This standard was for almost any computer-based controllers for example event-based controllers and single board computers.
When the production cost of microprocessors and microcontrollers fell, it became feasible to replace old, big and expensive components like potentiometers and varicaps with microprocessor read knobs. With the integration of microcontrollers, the application of embedded systems has further increased. The embedded systems are being used into areas where generally computers would not have been considered. Most of the complexity is contained within the microcontroller itself and very few additional components are needed that’s why most effort is done in software area.
2.1.2 Common Characteristics

Embedded Systems have several common characteristics.

- *Uni-Functional*: Embedded systems are usually designed to execute only one program but repeatedly. For example, an ordinary scientific calculator will always do only calculations. While on the other hand, a laptop computer can execute an enormous number of different programs, like web browsers, word processors, programming tools and video games. New programs or softwares are added very frequently.

- *Tightly constrained*: All computing systems have constraints on design metrics, but these constraints can be very tight for embedded systems. A design metric is defined as, “a measure of an implementation’s features, such as cost, size, performance, and power”. Embedded systems are often desired to cost just a few dollars, they must be designed for minimum size to fit on a single chip, they must be able to perform fast processing in order to process real-time data, and they must consume minimum power in order to extend battery life or may be to prevent the requirement of a cooling fan.

- *Reactive and real-time*: Many embedded systems should be able to continually react to changes in the system’s environment. They must also compute certain results in real time without too much delay. For example, a cruise controller in cars have to monitor and react to speed and brake sensors continuously. It must compute acceleration or decelerations repeatedly within quite limited time; a delay in computation of results could result in a fatal failure to maintain control of the car. On the other hand, a desktop computer generally focuses on computations with comparatively infrequent reactions to input devices. In addition, a delay in those computations may perhaps be inconvenient to the user but that does not result in a system failure.
2.2 Model Based Design

Model-Based design in short MBD is a mathematical and visual method of addressing problems associated with designing complex control systems. It is used in many industrial equipment designing, automotive and aerospace applications. Here in this thesis our focus is on climate control of new vehicles. This methodology is used in designing embedded software.

Embedded software development consists of four steps:

1. Modeling a plant.
2. Analyzing and synthesizing a controller for the plant.
3. Simulating the plant and controller.
4. Integrating all these phases by developing the controller.

Model-based design is quite different from the conventional designing method. In this methodology designer use continuous and discrete time building blocks instead of using long and complex software coding.

This model based design enables designer to fast prototyping, testing and verification. Along with all these advantages, dynamic effects on the system can also be tested in hardware-in-the-loop (HIL) simulation mode.

Some important steps in model-based design approach are:

1. By choosing appropriate algorithm and acquisition of real-world system data, various types of simulations and analysis can be performed before producing a real controller.
2. The model produced in step one is used to identify characteristics of the plant model. Then a controller can be made based on these characteristics.
3. Using this model, the effect of time varying inputs can be analyzed. In this way the possible errors can be eliminated and it is very convenient to change and test any other parameters.
4. Last step is deployment.
Advantages of model based design compared with the conventional approach are as follows:

- Model based design provides common design environment which is important for development groups from the view point of general communication and specifically for data analysis and system verification.
- Model based design enable engineers to detect and correct errors in early phase of development. This is crucial point for minimizing time and financial impact of system.
- Model based design can be reused later for upgrading and for derivative systems which are capable to expand.

2.3 ECU DEVELOPMENT

2.3.1 Conventional Approach for ECU Development

The conventional approach for electronic control unit (ECU) development is summarized in following four steps:

1. Some experienced personal define the functions and system architecture and then the hardware engineer design the hardware circuit.
2. Control engineer design the control algorithms and a programmer generate a handwritten code for that algorithms.
3. Then these control algorithm program codes and hardware are integrated and tested by system engineer or maybe hardware engineer.
4. Then on the engine test bench the complete system is tested.

There are few problems with this conventional approach for ECU development. First and very major problem is that the hardware circuits are made before the confirmation of control rules and results. Only this factor adds a big risk in the process of ECU development. Secondly if some error is encountered during the program code testing, it is very difficult to judge whether the error is because of software codes or in the control algorithms. This programming of the control algorithm is itself a very time consuming process and it take additional time when some errors
are encountered and the process of debugging. Since many people from different field of work are involved in this process so coordination between them also take time and it makes the development cost to increase. [2]
That’s why the conventional development process can not satisfy the demand of modern age and its requirements.

2.3.2 Modern ECU Development
On the bases of integrated development environment, the modern development of electronic control units can be efficiently completed and tested. Using model based simulation and hardware-in-the-loop simulation it is very easy and convenient to eliminate software errors and to modify the control algorithms. Due to this the development cost is reduced and development efficiency is improved. This modern development process is called V-cycle development process.
This process is illustrated in Fig. 1

![Fig. 1. The V-Cycle of model-based software development.](image-url)
• Using very sophisticated tools like MATLAB/Simulink/Stateflow and dSPACE TargetLink, the control algorithms are modeled. These control algorithms are confirmed using off-line simulations.
• The ANSI C code is produced using code generation tool. The one we are using is dSPACE TargetLink.
• The code produced in above step is compiled and downloaded into the control module and simulation is done in Hardware-in-the-loop mode, which confirms the credibility of the control algorithms.
• This tested program code of control algorithms is downloaded into the electronic control unit for further test and modification.
• Finally calibration of the whole control system is done.

2.4 Universal measurement and Calibration Protocol (XCP)
XCP is a standardized and universally applicable protocol with much rationalization potential. It is not only used in ECU development, calibration and programming, it is also used to integrate any desired measurement equipment for prototype development, functional development with bypassing and at SIL (software in the loop) and HIL (hardware in the loop) test stands. [10]
For calibration and measurements, it is common practice to connect electronic control units in a CAN* network. For this purpose CAN calibration protocol is used extensively. With increasing demands of more sophisticated controllers, new electronic control units are becoming more and more complex and for that reason new networks are being developed such as, FlexRay, TTCAN etc.[10]

To meet the needs of new networks, the measurement and calibration protocol should be more generalized and flexible. This generalized and flexible protocol is XCP (Universal measurement and calibration protocol).
XCP is independent of transport layers. So in XCP, “X” generalizes the various transport layers that

* Details about CAN are provided in Appendix A.
are used by the members of the protocol family e.g. [4]

- XCP on CAN
- XCP on FlexRay
- XCP on Ethernet
- XCP on USB and so on

This is summarized in Fig. 2.

![Fig. 2 XCP support for different transport layers](image)

**Fig. 2 XCP support for different transport layers [5].**
This chapter will give answers to the following questions:

- What is parameter dependency problem?
- What is the effect of parameter dependency problem on tuning of embedded control systems?
- What are the difficulties to solve the problem at different platforms?

Note: All examples used in this report are only for illustration purposes and are NOT the actual parameters used in climate control module of Volvo Cars and Volvo Trucks.

### 3.1 Complete process for developing embedded control systems

The complete process for developing embedded control systems is illustrated in Fig. 3. First step of this development process is to define parameters and that can be done in the m-file. These parameter values are loaded into MATLAB base workspace from where TargetLink/Simulink model fetches these values to simulate the process.

After checking the simulation results and doing some modifications if required, C-code is generated by TargetLink. That C-code contains all the information about the control algorithm and input values. In the next step the auto-generated C-code is compiled using a Green Hills Suite.
Green Hills' software together with GNU Make and VBF converter is used to generate a map file and VBF file (Volvo Binary Format). This vbf file is downloaded in the embedded controller. The map file is used to generate A2L file using TargetLink. This A2L file is required by the calibration tool (for this project ATI VISION is used for calibration) and then using this calibration tool we can do parameters modifications in ECU. These modifications are also called tuning.
3.2 Parameter Dependency

As all parameters are defined in a m-file, some parameters depend on the values of some other parameters. It may also be possible that the values obtained as a result of calculation between two or more parameters are used in the definition of other parameters. So, all those parameters which contain some other parameters or calculations of some other parameters in their definitions are called dependent parameters e.g.

1. `Maximum_Fan_Speed_Mode4 = [114 133 144 151 158 176];`

2. `Max_Fan = max(Maximum_Fan_Speed_Mode4);`

3. `Minimum_Fan_Speed = 20;`

4. `Noise_Comp_FanSpeed = [0
   Minimum_Fan_Speed
   70
   130
   Max_Fan ];`;

5. `Fan_Voltage = [5 9 12 ];`

6. `OutsideAir_NoiseComp = [5 10 15;
   Minimum_Fan_Speed Minimum_Fan_Speed Minimum_Fan_Speed;
   55 60 65;
   105 110 115;
   Max_Fan Max_Fan Max_Fan ];`

In above example parameters:

- Parameter 2 is dependent on parameter 1.
- Parameter 4 is dependent on parameter 2 and 3.
- Parameter 6 is dependent on parameter 2 and 3.
3.3 Reasons for introducing parameter dependencies

Thinking of parameter dependencies a question may arise in minds that, “Why do we need to introduce parameter dependencies in the first place?”

Answer to this question is that, when designing a control algorithm in a tool such as Simulink, it is convenient to use named parameters instead of hardcoded numbers.

For instance, if the highest fan available corresponds to a voltage of 13.5V. Designer may want to have a parameter for this, so that instead of using the value 13.5 at many instances of algorithm, the name of parameter specified for that value can be used. If one day that hardware is needed to be changed and for new hardware 13.4V is the maximum that can be used for highest fan level, then it is easier to change one parameter value rather than changing many hard coded values at different instances.

Sometimes it is good to have one parameter depending on another. For instance in a look-up table, there are several values in each vector and these values may depend on other parameters. It would be rather limiting if a vector or a matrix could only contains hard coded numbers.

So, the use of dependent parameters helps keeping a good structure in the algorithm. It makes it easier to work with the parameters.

3.4 Statistics about parameter dependency

There are quite significant numbers of parameters which are dependent on other parameters. For instance in Climate Control P3, total number of parameters is 1618 out of which 227 parameters are dependent on other parameters and 1391 parameters are independent. We call independent parameters as “Base Parameters”.

This is illustrated in the following pie chart.

- Total number of dependencies used at different instances in parameter definitions = 378
- Number of whole dependencies (i.e. complete parameter definition depends on some other parameter(s)) = 154
- Number of fractional dependencies (i.e. dependencies used at different instances inside the parameter definitions which are vectors or matrices) = 224
To analyze the problem of parameter dependency, let’s walk through the development process of embedded control systems and find out what exactly is the problem with parameter dependencies.

As the process starts with parameter definitions in m-file, so the investigation starts from m-file, see Fig. 5. To visualize this process, an illustration with an example of parameter with dependencies in its definition is shown as follows:

### 3.5 Parameter dependency problem in development process

To analyze the problem of parameter dependency, let’s walk through the development process of embedded control systems and find out what exactly is the problem with parameter dependencies.

As the process starts with parameter definitions in m-file, so the investigation starts from m-file, see Fig. 5. To visualize this process, an illustration with an example of parameter with dependencies in its definition is shown as follows:
After defining all parameters, the m-file is run in MATLAB. In this step all the values of dependency parameters are evaluated by MATLAB and are loaded into MATLAB base workspace. Precisely during this loading process the dependencies are replaced by their values and any information about the relation of a parameter with dependency parameter is lost.

This is shown in Fig. 6.
Fig. 6. Dependency loss in MATLAB base workspace.

As now the dependency information is lost, so this loss will propagate through all the further steps, for example in C-code generation, A2L file and in strategy file (a file created by the ATI VISION).

Fig. 7 shows that the propagation of dependency information loss. So in C-code there is no information with the help of which we can trace dependency parameters.
Fig.7. Propagation of dependency loss from MATLAB to C-code.

3.6 Effect of parameter dependencies on development process

The problem caused by parameter dependencies comes to the surface during the calibration step.

During calibration the values of parameters are tuned. When the information of parameter dependencies is lost, then we have to tune each parameter value individually. This is shown in Fig. 8.
So if a parameter is used, for instance, in the definitions of five different parameters, then we have to tune the value of that parameter at those five locations individually. If there is any calculation involved in any parameter definition, then we must do it manually and update the value. This process of changing values manually is very time consuming and error prone.

There is another possibility that to avoid doing these calculations and tuning parameter values individually. We can change the parameter values in the original m-file, where we have all parameter definitions and repeat the complete process again. This is very laborious work and it also takes a lot of time, so this possibility is not so feasible.
POSSIBLE SOLUTIONS

As the complete process for developing embedded controllers is a multistage process and it depends on four highly sophisticated software platforms. So there can be different approaches to solve the dependency information loss. Following are the possible platforms for doing modifications in order to handle the dependency loss problem.

- MATLAB
- TargetLink model
- C-code
- Calibration tool
- Separate windows application

Following is the in depth analysis of above mentioned platforms and possibility of finding a feasible solution.

4.1 Parameter dependencies and MATLAB

When m-script, containing all parameter definitions, is run in MATLAB, all parameter values are evaluated and stored in MATLAB base workspace. Right at this first step dependency information in m-script is lost. The reason of this loss is that MATLAB base workspace support values belonging to only one class type. That can be “char”, “double”, “struct” or any other class but the values can not belong to a mixture of two or more class types, i.e., values cannot consist of two elements of an array belonging to “char” class and other elements of array belonging to “double” class. Although using the
data type “struct” it is possible to add cells and arrays in one definition but all inputs should be given in pairs, i.e. “Name of field” and “Values”. So that does not give the required freedom.

![Supported Class types in MATLAB base workspace.](image)

In our case of parameter dependency for example, we have an array of eighth elements. Second element and eighth element of our example array are names of some other parameters, so these names belong to char class and rest of elements of that array are numerical values belonging to double class. So MATLAB evaluates the values of dependency parameters and replace all names with their corresponding values and our dependency information is lost.

Although there is a function in MATLAB called “eval” and this function can be used instead of dependency parameter name but this does not solve our problem because this function will evaluate the values of those parameters and eventually it’s the value of parameter which is updated in the base workspace and dependency information is still filtered out.

Moral of the story is that we can not do anything in MATLAB to save our dependency information until unless MathWorks do some changes in MATLAB so that base workspace would be able to support values belonging to different classes in same definition.

### 4.2 Parameter dependencies and TargetLink

In TargetLink we can use custom lookup tables and we can include custom code. Let us suppose for a moment that by adding these custom lookup tables and using some extra blocks we manage to
introduce lost dependency information in TargetLink model. But when TargetLink will generate C-code, most probably it will evaluate all those values and resulting values will be included in C-code. There are two reasons for this behavior of TargetLink:

- First reason is that, TargetLink work inside MATLAB so all the calculations are done in MATLAB and we face the same problem as described previously.
- Second reason is that, dSPACE claims that TargetLink generates C-code in the most efficient way, because this C-code is flashed into controller in binary format, so it is the maximum effort of TargetLink to keep C-code as small as possible because of the limited memory of ECU and demand of high operational speed.

So TargetLink does not generate extra variables and pointer in C-code until unless some significant changes are done in TargetLink by dSPACE.

4.3 Parameter dependencies and C-code

C-code generated by TargetLink can be modified and it is possible to add any kind of extra information but there are two reasons which make this possibility impracticable.

- First reason is that, this C-code will be flashed into ECU and there is very limited memory in the control unit and bigger C-code will result into a less efficient embedded controller.
- Second reason which makes this possibility impracticable is that it requires a lot of manual labor every time we change something. This is also error prone.

4.4 Parameter dependencies and Calibration tool

In calibration tool like ATI VISION, there is an option to use script written in Vision scripting language or in Visual basic. Instead of doing manual calibration we can automate calibration using the script.
In our case, we have matrices with dependencies. So in order to do calibration using the scripting option we have to write function for doing matrix calculations and then that script must be able to evaluate dependencies according to new values. So this option is not so feasible.

### 4.5 Separate windows application

After analyzing all possibilities only one option is left. That is to develop a separate windows application which will extract dependency information from m-script, calculate the values of dependency parameters according to the values tuned in calibration tool and will implement those new values of dependencies back in calibration tool.
SELECTED SOLUTION

After analysis of all possible solutions, it is deducted that the most feasible solution to the dependency loss problem is a separate windows application which:

- Extracts dependency information from m-file.
- Gets tuned parameter values from calibration tool.
- Calculates all values corresponding to those tuned parameter values.
- And implements updated values of dependency parameters back in calibration tool.

5.1 Reasons for selecting this solution

Among other solutions we have selected development of “separate windows application”, as a feasible solution. Major reasons for selecting this solution are as follows:

- Selected solution which is developing a separate windows application does not need any modification of present softwares. This solution is fast, no extra licenses are required for this and it works just according to our requirements.

- If we choose any solution which includes modification in software tools, then that involves the involvement of tool makers. That process of convincing toolmakers to modify their software according to our requirements and if they agree then the process of developing and releasing new version of software may take very long time.

- Tool makers would charge a great sum of money to make specified changes or for making an add-on application for the softwares.
5.2 Overview of solution

The solution is an application named “Dependency Calibrator”. It works in two steps.

In the first step the m-file is parsed and the information of dependency parameter along with their location in parent parameter is extracted and rearranged in a way that it can be used in the second step that is calibration.

During the second part of the process, first of all the application will import data from VISION so that if user has tuned any value in calibration tool, that data will be updated in MATLAB and then the application will do calculations in MATLAB after that new values obtained as a result of those calculation will be updated again back to VISION. This cyclic process from VISION to MATLAB and back to VISION will update parameter values. If user has changed values which was used by other parameters, those new values will be updated on all locations where they are used. This is shown in Fig. 10.

![Fig. 10 Overview of solution.](image)
The application “Dependency Calibrator” is divided into two parts.

- Parser
- Calibrator

Detailed explanation of how this application is working is as follows.

5.3 Required Softwares
Parser works without any requirement of external software but in order to run “Calibrator” following softwares must be installed on your system:

- MATLAB R2007b
- ATI VISION 3.5.3

MATLAB is automatically launched by the application but make sure to launch ATI VISION before you use “Calibrator” part of “Dependency Calibrator” application.

5.4 Project file
Project file is a key to control the “Dependency Calibrator” application. Instead of using hard coded paths for different files used in this application, an option is given to the users to select their desired locations. These locations can be specified in a separate file which is named as project file.

In this project file the instructions can be given after certain tags. One must be very careful because these tags should not be altered. While user inputs can be given after the symbol “@”.

“Dependency Calibrator” application is in fact capable of handling multiple m-files and multiple c-files. Directory path for these files can be specified in project file.

Project file contains following tags:

- VISION's Device Name @ : After this tag, name of the hardware device which is used in the VISION device tree, should be given. For example,

  VISION's Device Name @ PCM
Or

VISION's Device Name @ CCM

- **Path of m File @**: After this tag the full path for m-file should be given. If number of m-files is more than one, then this tag followed by file path of those m-files should be given on a new line. Parser will read all these files and will merge them into one file. For example,

  Path of m File @ C:\FolderName\subFolder\File_Name.m

  Path of m File @ C:\FolderName2\subFolder2\File_Name2.m

- **Root directory for c files @**: In general practice c files can be generated in different folders but their root directory remains same. So in order to avoid repeating same address and to minimize the chances of error this tag is introduced in project file. So after this tag path of root directory for c files should be specified. Please note that there should be no "\" at the end of root directory path. For example:

  Root directory for c files @ D:\ABC_XYZ\subFolder\subSub

- **Folders containing c-files @**: After this tag the names of folders which contain c files should be specified. If there are more than one folder containing c files then those folders names should be added after a comma ",". The parser will then search these folders for all c files contained in them.

  For example:

  Folders containing c-files @ FolderMedCfiles,subFolder\cFolder

- **Root Output Directory @**: This tag should be followed by the path for required location where the user wants the application to generate all files. For example:

  Root Output Directory @ C:
• Extra File for calibrating non-calibratable parameters @    : After this tag, there should be the path for the file containing names of those parameters which are not calibratable but they are desired to be calibrated in VISION. Those names should be exactly the same as defined in m-file, followed by underscore “_” and followed by any desired word or character. For example:

```
Extra File for calibrating non-calibratable parameters @ C:\ExtraParNames.txt
```

5.5 Parser

The first part of the complete dependency calibration process is the parser. When “Parser” is executed, a window appears showing two options, “Load Project File” and “Parse”. It is required to load the project file before hitting the “Parse” button. Once the project file is loaded, the parser will have all the information to start parsing. Parser application is shown in following fig.

![Parser application](image)

Fig.11 Parser application
There are six operations done by “Parser” on input m-file(s) and c-file(s) which are explained as follows.

5.5.1 Comments Removal
First operation done by parser on m-script is comment removal. It removes all the comments from m-file(s). Some comments start from the beginning of line and other are at the end of parameter definitions. If there is more than one m-files, then all those files will be merged into one file as a result of this step. The output file produced in this step will be without any comment.

The reason for removing comments is that, in next steps we have to convert multiple line parameter definitions to single line. For that it is required that there must be no comments. Second reason is that some comments contain the same structure as the parameter definitions in fact those are old values of the same parameters. So in order to minimize any possibility of error, we have to remove comments.

After removing comments, the parser also removes empty lines and extra white space inside the parameter definitions.

5.5.2 Multiple line parameter definitions to single line
It is required that all parameter definitions should be single lined. There are two reasons for this operation on parameter definitions.

First reason is that parser is reading complete file line by line, so it is important to read the complete information about a parameter in one step.

Secondly we have to separate all dependencies present in parameter definitions. It is also possible to read multiple lines but in doing that we face big problem of setting a new line record (in programming record is a tag which tells about the end of line by default it is “\n”) which tell the parser that current parameter is finished here and new parameter definition has started. In that case we must add any symbol or specific number of white space or something like that which must repeat after each
parameter definition. In our case there is no such pattern repeating periodically and symmetrically in the m-file and it is not practical to modify all m-files by putting a symbol after each parameter. So, this parser application handles this problem by converting all parameter definitions into single line definitions.

5.5.3 Separating parameters with dependencies

Up till this point all parameter definitions are converted into single line definitions and all comments are removed from the m-file(s). Next step in the parsing process is separating those parameters which are dependent on some other parameters. This goal is achieved by using regular expressions.

The regular expression searches for any parameter name in the parameter definitions and if there is any parameter name found in the definition, it saves that parameter in a separate file.

After this step we have all parameters with dependencies filtered out in a separate file.

5.5.4 Position of dependencies in a parameter definition

Going one step ahead, now we have to parse each parameter to find where exactly the dependency lies. So this is a very crucial moment in the whole parsing process.

According to our m-file there can be three major groups of parameters.

- Complete dependency
- Dependency in an array or vector
- Dependency in a matrix

Complete dependency can be defined as, when whole parameter definitions depend on some other parameter or some calculations of other parameters. e.g.,

\[
\text{Max} \_\text{Fan} = \max (\text{Maximum} \_\text{Fan} \_\text{Speed} \_\text{Mode}4);
\]
Dependency in an array or vector can be defined as if there is some element of array or vector depending on other parameter or some parameter calculations. In this case we have to know, where precisely the dependency lies in that array or vector. For example:

\[
\text{VentFan\_Speed} = [0, \text{Min\_Fan\_Speed}, 30, 45, 110, \text{Max\_Fan}];
\]

In this example, VentFan\_Speed is dependent on Min\_Fan\_Speed which is 2\textsuperscript{nd} element and Max\_Fan which is 6\textsuperscript{th} element.

Third group of parameters can be dependencies in a matrix. This is even more complicated because in this case we have to keep track of two things, column index and row index.

The output of this step is according to following format.

**Keyword; Parameter Name; Dependency Name; X-Offset; Y-Offset;**

So this format is a semicolon separated string in which:

**Keyword:** can be any word but in our case it is defined as “Parameter”. The sole purpose of this keyword is to distinguish this string from any other information in the file that can be comments or some other information. So to be sure that this is the information of parameter dependency it must start with the specific keyword.

**Parameter name:** represents the actual name of parameter which has dependency in its definition.

**Dependency name:** is the name of parameters on which the parent parameter is depending. This can be only name of other parameter or it can be the result of some calculation of other parameters.

**X-Offset:** In case of a 1D array or a vector X-Offset will be the location of dependency i.e. the number or element in the array. In case of a matrix X-Offset is column index of the dependency element.

**Y-Offset:** In case of 1D array or vector y-Offset will always remain “y”. This indicates that the respective parameter is a vector. In case of 2D array and matrices Y-Offset indicates the row index of the dependency element.

Zero based indexing is used in this format for X and Y-Offset. When the value of both X-Offset and Y-Offset is “d”, that means the complete definition of that parameter is a dependency.
This conversion of parameter dependency information from MATLAB format to new format is shown in Fig. 12.

\begin{verbatim}
Max_Fan = max(Maximum_Fan_Speed_Mode4);
Noise_Comp_FanSpeed = [ 0
    Minimum_Fan_Speed
    70
    130
    Max_Fan ];
OutsideAir_NoiseComp = [ 5 10 15:
    Minimum_Fan_Speed Minimum_Fan_Speed Minimum_Fan_Speed;
    55 60 65;
    105 110 115;
    Max_Fan Max_Fan Max_Fan ];
\end{verbatim}

<table>
<thead>
<tr>
<th>KEYWORD</th>
<th>PARAMETER_NAME</th>
<th>DEPENDENCY_NAME</th>
<th>X-Offset</th>
<th>Y-Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Max_Fan</td>
<td>max(Maximum_Fan_Speed_Mode4); d</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Noise_Comp_FanSpeed</td>
<td>Minimum_Fan_Speed</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Parameter</td>
<td>Noise_Comp_FanSpeed</td>
<td>Max_Fan</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Parameter</td>
<td>OutsideAir_NoiseComp</td>
<td>Minimum_Fan_Speed</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Parameter</td>
<td>OutsideAir_NoiseComp</td>
<td>Minimum_Fan_Speed</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Parameter</td>
<td>OutsideAir_NoiseComp</td>
<td>Minimum_Fan_Speed</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Parameter</td>
<td>OutsideAir_NoiseComp</td>
<td>Max_Fan</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Parameter</td>
<td>OutsideAir_NoiseComp</td>
<td>Max_Fan</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Parameter</td>
<td>OutsideAir_NoiseComp</td>
<td>Max_Fan</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Fig. 12. Extracted parameter dependency information.

### 5.5.5 C-code parsing

In the previous step we have generated the file which contains information about the parameter name, dependency name and the location of dependency in parameter definition. The problem here is that in
the calibration tool, the names of parameters are not the same as were defined in the m-script. These names are changed by adding different tags during the C-code generation in TargetLink. So in order to find the respective parameter names we need to parse C-code.

The good news here is that TargetLink only change the actual names according to a certain pattern which can be selected and modified in the TargetLink model. So according to that pattern we can extract the corresponding names of parameters.

5.5.6 Replacing parameter names
At this step we have dependency information of parameters from m-script and we have their corresponding names in C-code which can be found in calibration tool.

In this step parser will replace the names of parameters as defined in m-script with their corresponding C-code names.

The output of this final step of parsing is ready to be used for calibration process. These new parameter names are the same as defined in the strategy file of ATI VISION. So in the file generated in this step we have all information of dependency parameters and their positions in the definitions of parent parameters.

After completing this process the application will show a message informing about completion of the parsing process. By clicking “OK” this application will exit.

5.6 Calibrator
Second part of dependency calibration process is another application which is interfaced with MATLAB and ATI VISION. Make sure that before running “Calibrator” a project is open in VISION otherwise the application will display an error message.

Calibrator is shown in Fig. 13.
Fig. 13. Calibrator application.

If a vision project is open a calibration application is run then, at the start window two options are shown.

- Load project file
- Calibrate
It is required to first load the same project file that is used for parser. Completion of this operation will be confirmed by displaying a message box. After this, the calibrator has got all information required to run and it is ok to press “Calibrate” button.

When calibrate button is pressed the application export parameter values tuned in VISION to MATLAB and update corresponding parameters in MATLAB base workspace. If the value which is tuned belongs to dependency parameter then all the parameters depending on that value will be updated according to their relation defined in m-file.

After updating all parameter values, those values are sent back to VISION at their appropriate locations and in this way we get our desired results.

Detailed explanation of how this happens is as follows.

For explanation we further divide calibrator into two parts,

- Parameter values from VISION to MATLAB
- Parameter values from MATLAB back to VISION

### 5.6.1 Parameter values from VISION to MATLAB

In this step the application will use MATLAB and ATI VISION as COM servers. When MATLAB is invoked as a COM server, it looks like as in Fig.14. Original m-file which contains all the parameter definitions is required in this step. The path for that m-file is specified in the project file. The application commands the MATLAB server to change the current directory of MATLAB to the root directory of the specified file. After that it runs the m-file and that all parameter definitions are loaded in MATLAB base workspace.
Now to find out which parameters may be tuned in calibration tool, it is required to look into C-code generated by TargetLink for the variables belonging to the variable class “CAL”. Type of variable class for any variable can be changed from “Data Dictionary” of TargetLink.

As this information is saved into a file during parsing, so the application will get this information from that file.

When the values of these calibratable parameters are tuned in VISION, the application gets these new values for updating them in MATLAB. But the parameter names in MATLAB are different. So the application will translate the names.

The program now knows the names of parameters as defined in m-file and their corresponding names in VISION.
To read the values of calibrated parameters from VISION, first of all the application will check the data type of those parameters, whether they are scalar, 1D array, 2D array, 2D table or 3D table. The method for getting values of parameters belonging to different data type is different. Once the data item type is known the program will send that parameter information to its respective method.

- For a scalar the actual value of that scalar in VISION is transferred to MATLAB.
- For a 1D array the actual values of Y-axis are transferred to MATLAB, because in case if 1D array x-axis values are just the index number.
- For a 2D array the actual values of z-axis are transferred to MATLAB.
- For a 2D table, actual values of y-axis are the concerned values.
- For a 3D table, the actual values of z-axis are transferred to MATLAB, because on x-axis and y-axis the actual values belong to some other parameters which are generally 1D array and are handled separately under DataItemType1DArray.

After doing this process the program will run the file containing parameter dependency information. This file was generated during the parsing process. According to this file all dependency parameters are updated with new values got from VISION. Now, as all dependency parameter values are updated so, program will save all this information in a “mat” file which is named as “calibration. mat”. This file is a binary file and all information present in the MATLAB base workspace is saved in it.

At this point the process of updating parameter values from VISION to MATLAB is completed and the next step is to update all the changes caused by changing the values of parameters which are dependent.

Here another very important point is that if a value is changed in VISION and that value actually was a dependency then, according to the requirement the value must not be allowed to change until the change is made in the base parameter value. This calibration application does exactly that. Tuning of independent parameters is not affected by the application.
5.6.2 Parameter values from MATLAB to VISION

Now coming to the second part of calibrator, it is now required that the values of all parameter dependencies should be updated back in VISION.

For this purpose the application will load the “calibration.mat” file that was saved with new values in previous step. Another file required by this part is the final output file obtained in the parsing process. According to that file the program will find the name of a parameter as it is in the calibration file. The program will find the value of dependency parameters from the mat file in MATLAB and it also has the information about the location of the dependency in the parameter definition, so it will update the corresponding value of dependency in VISION.

To update the value of dependency at the right location the program will first check the data type of the parameter and according to the data item type it will send the information to the appropriate method. That method will check first that if the dependency is a part of array or a matrix or is it some resultant value of calculation between some other parameters. So according to that information the program will do all required calculation in MATLAB and then import the value to the right place in the ATI VISION.

When all the values are updated it will show the message that the values are updated and if there is any parameter that did not belong to variable class “CAL”, the program will show all these parameters with warning in a list box.

This process is iterative and it can be repeated as many times as the user wants. When all the parameters are calibrated then this application can be closed. By closing the application, the command window of MATLAB, which was opened as a COM server, will also be closed.

This process of updating the parameter values can be monitored in VISION using screen window and control items.

5.7 Calibration of non-calibratable parameters

In the system there can be some parameters which are used indirectly. Indirectly means their values are used in some other parameters but they them self are not used anywhere in the TargetLink model due
to which they can never appear in the C-code and as a result those parameters are not available in the calibration tool for tuning.

The application “Dependency Calibrator” handles this kind of parameters as well. For tuning these parameters we can create new data item in calibration tool and then add names of those data items into a text file. Path of this text file should also be specified in the “Project File” after the tag “Extra File for calibrating non-calibratable parameters”.

For creating new data item in ATI VISION, go to “Data Item Manager”. In “DataItemGroups” go to device name e.g. “CCM”. Open the “Characteristics” folder and click on “Values”. The window on the right side of this panel will display different data items and some other information about those data items. In this window by right clicking and then selecting “New” will show a dialog box titled as “Select Data Item Type”. Now select the type of data item according to the type of parameter. After selecting data item type, a dialog will appear asking the name of parameter. The name of data item should be selected as, **actual name of parameter as defined in m-file followed by an underscore “_” and some other name according to your wish. By choosing this kind of name, the data item will be connected to the original parameter as defined in m-file.** After that a dialog will appear showing the properties of that data item. Make sure that the “Base address” of this data item does not coincide with the base address of any other predefined data item. Memory type should be selected to “RAM [adjust and monitor]”, then you will be able to change the value(s) of this data item. After doing all these adjustments, click “Apply” and then click “OK”.

By following this procedure a new data item is created which was not calibratable by default, but now this parameter can also be tuned in the calibration tool.

**5.8 Dependency Calibrator in a Nutshell**

All the steps of “Dependency Calibrator” are summarized and depicted in Fig. 15.
Fig. 15. Dependency Calibrator in a nutshell.
6 RESULTS

Results of this project are demonstrated with the help of an example of some parameters which contain dependencies.

Following graphs are made in ATI VISION using control objects in Screen file.

Graphs in Fig.16 show the values of parameters before tuning. So here it is required that if we tune any parameter value on which other parameters are dependent, then all the value of that dependency parameter should be changed at all instances where that dependency is used.

In this example there are two parameters on which values of some other parameters are dependent, we call these two parameters as “base parameters” which are as follows:

Minimum_Fan_Speed = 16;

Maximum_Fan_Speed_Mode4 = [114 133 144 151 168 173];

The other parameters in our example which are dependent on these base parameters are as follows:
Max_Fan = max(Maximum_Fan_Speed_Mode4);

Noise_Comp_FanSpeed = [ 
    0 
    Minimum_Fan_Speed 
    70 
    130 
    Max_Fan 
];

OutsideAir_NoiseComp = [ 
    5 10 15 
    Minimum_Fan_Speed Minimum_Fan_Speed Minimum_Fan_Speed 
    55 60 65 
    105 110 115 
    Max_Fan Max_Fan Max_Fan ];

These initial values of parameters are shown in the following fig.
The parameter values to notice are outlined with red blocks.
Now when we tune the values of base parameters, the application will change these values at all instances where they are used as dependencies.
New values of base parameters are as follows,

\[
\begin{align*}
\text{Minimum\_Fan\_Speed} & = 20; \\
\text{Maximum\_Fan\_Speed\_Mode4} & = [114 \ 133 \ 144 \ 151 \ 176 \ 173];
\end{align*}
\]

Now as,
\[
\text{Max\_Fan} = \max(\text{Maximum\_Fan\_Speed\_Mode4})
\]

So the value of \text{Max\_Fan} will become 176 according to the changed value of \text{Maximum\_Fan\_Speed\_Mode4}

The calibrator application will update these new values of \text{Minimum\_Fan\_Speed} (i.e. 20) and \text{Max\_Fan} (i.e. 176) in all other parameters.

Updated values are shown in Fig. 17.
Fig. 17. Values of parameters after calibration.

Changed values are outlined with red blocks.
6.1 Verification

The application “Dependency Calibrator” is used for the “Seat Ventilation Control” and “Cabin Climate Control” Volvo Cars and Volvo Trucks. It has successfully updated the values of all the dependency parameters tuned in the ATI VISION environment. That means the application has successfully completed the parsing process and the calibration process. This application does not allow the values of dependency to change until that value is changed in the base parameter. This functionality helps to keep the value of dependency parameter same throughout the system. Which was the main idea behind this thesis work.
CONCLUSION

This thesis report concludes that the application “Dependency Calibration” handles the parameter dependencies quite efficiently. The fact that all calculations are done in MATLAB makes it possible that all kind of parameter operations which are supported by MATLAB are also supported by this application and we can use the capabilities of TargetLink/Simulink model to the maximum extent.

So this application contributes to following:

- Makes online calibration possible for dependency parameters.
- Less error prone.
- Is efficient and saves valuable time.
- Requires minimal manual labor.
FUTURE WORK

Current application saves updated parameter values in a “.mat” file which is a binary file. The application does not write or update parameter values in the original m-file. It would also be interesting if, when user would tune a dependency parameter the application would inform user about all the parameters that will be affected by that certain change.

Future work related to this project can be:

- Developing a text editor application specifically for updating m-file. That application should save old parameter values by commenting it and then write a new parameter with updated values.

- Displaying more information about the parameters affected by tuning dependency parameter(s) in calibration tool’s environment.

For updating m-file, it may be helpful to use new MATLAB Editor API, available in MATLAB 2010a which is expected to be released in March 2010. This new MATLAB Editor API provides programmatic control over opening and saving files, navigating and modifying file contents, and querying file properties [6].

Now the problem has been solved for when we are using ATI VISION. It would be good to have solutions (or a generalized solution) for other calibration tools such as, INCA and CalDesk.
REFERENCES


APPENDIX

Calibration tools
Major calibration tools used at Volvo Technology are:

- ATI VISION
- CalDesk by dSPACE
- INCA by ETAS

ATI VISION
ATI VISION software is an integrated calibration and data acquisition tool that collects signals from ECU and external sources, measures relationships between inputs and outputs, enable real-time calibration and modification to closed loop control systems, time aligns and analyzes all information, manages calibration data changes and programs the ECU. [7]

VISION software can do following:

- Flash the ECU
- Monitor and measure signals in the ECU and external signals
- Calibrate ECU parameters in real-time
- Compare, import and merge calibration data
- Analyze data
- Perform ECU algorithm rapid prototyping

VISION has an integrated script manager and it is possible to write automation scripts for iterative processes using VISION scripting language. VISION includes an Application Programming Interface
(API) enabling the integration of application-specific components from different vendors and data exchange between VISION and other applications. [7]

**CalDesk**

CalDesk by dSPACE is a tool for different stages of the ECU development process like prototyping a control strategy, calibrating an ECU, or validating vehicle behavior. Rapid prototyping systems, ECUs and the vehicle bus can be accessed in parallel. Data from different sources may be recorded and analyzed as a whole. [9] Functionality overview of CalDesk is shown in Fig. A1.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
</tr>
<tr>
<td>Simultaneous measurement, calibration, and diagnostics</td>
<td></td>
</tr>
<tr>
<td>Support of dSPACE rapid prototyping platforms</td>
<td></td>
</tr>
<tr>
<td>ECU flash programming</td>
<td></td>
</tr>
<tr>
<td>Application programming interface (API) for automation, e.g., with Python or MATLAB®</td>
<td></td>
</tr>
<tr>
<td>Compatibility with ASAM-MCD standards</td>
<td></td>
</tr>
<tr>
<td>Integrated project and experiment management</td>
<td></td>
</tr>
<tr>
<td>Python Editor to create automation scripts</td>
<td></td>
</tr>
<tr>
<td>Quick configuration of experiment environment via drag &amp; drop or keyboard control</td>
<td></td>
</tr>
<tr>
<td>Wizard mechanism for user guidance</td>
<td></td>
</tr>
<tr>
<td>Instrument library with custom-specific default instruments (p. 282)</td>
<td></td>
</tr>
<tr>
<td>Easy keyboard-only operation</td>
<td></td>
</tr>
<tr>
<td><strong>Measurement</strong></td>
<td></td>
</tr>
<tr>
<td>Recording measurement data and calibration values over time</td>
<td></td>
</tr>
<tr>
<td>Time-correlated data acquisition from different sources</td>
<td></td>
</tr>
<tr>
<td>ECU master synchronous measurements and polling measurements according to PC timer</td>
<td></td>
</tr>
<tr>
<td>Polling measurements according to PC timer</td>
<td></td>
</tr>
<tr>
<td>Integrated measurement data analysis for online and offline use cases</td>
<td></td>
</tr>
<tr>
<td>Options to display signals over the time (TT plot) and over another signal (XY plot)</td>
<td></td>
</tr>
<tr>
<td>Creating new variables based on calculation formulas</td>
<td></td>
</tr>
<tr>
<td>Powerful trigger options for data recording</td>
<td></td>
</tr>
<tr>
<td>Acoustic signals for trigger and threshold conditions</td>
<td></td>
</tr>
<tr>
<td>Bookmarks (manual and automatic) and bookmark navigator</td>
<td></td>
</tr>
<tr>
<td><strong>Calibration</strong></td>
<td></td>
</tr>
<tr>
<td>Online and offline calibration</td>
<td></td>
</tr>
<tr>
<td>Support of single-page and two-page concepts with working and reference page</td>
<td></td>
</tr>
<tr>
<td>Calibration of ECUs without dedicated data segments</td>
<td></td>
</tr>
<tr>
<td>Underflow of parameter changes and display of change history</td>
<td></td>
</tr>
<tr>
<td>Comparing and merging multiple data sets</td>
<td></td>
</tr>
<tr>
<td>Report generation on data sets in XML, HTML or PDF format</td>
<td></td>
</tr>
<tr>
<td><strong>Interfaces and protocols</strong></td>
<td></td>
</tr>
<tr>
<td>AUD, NBD, JTAgNexus, JTAg/OCDS and JTAg/SDI</td>
<td></td>
</tr>
<tr>
<td>CCP (CAN Calibration Protocol)</td>
<td></td>
</tr>
<tr>
<td>XCP on CAN (p. 298)</td>
<td></td>
</tr>
<tr>
<td>XCP on Ethernet (TCP/IP and UDP/IP)</td>
<td></td>
</tr>
<tr>
<td>XCP on FlexRay</td>
<td></td>
</tr>
<tr>
<td>CAN and LIN bus monitoring</td>
<td></td>
</tr>
<tr>
<td>LIN monitoring</td>
<td></td>
</tr>
<tr>
<td>Diagnostic protocols (p. 293)</td>
<td></td>
</tr>
</tbody>
</table>

*Fig. A1. CalDesk functionality overview [9].*
CalDesk Application Programming Interface (API) is based on Windows COM/DCOM components and is accessible by various programming languages such as C#, Python or Visual Basic.

CalDesk has an integrated Python editor with syntax high-lighting and automatic code completion. In conjunction with the python interpreter it is possible to write automation scripts. [9]

**INCA**

The INCA base product comprises the system core with its measurement and calibration functionality. It supports online and offline adjustments of characteristic values, lines and tables. Simultaneously to the parameter optimization, INCA allows to acquire measurement signals from ECU’s as well as easy and quick reuse of existing calibration data and hardware configurations. [8]

In addition to the measurement and calibration core system, the INCA base product includes tools for managing the configurations of ECU projects and calibration parameters, for analysis and measured data and for reprogramming the ECU flash memory. [8]

Besides interactive optimization of parameters, INCA supports access via remote control interfaces for automation of experiments. For example, test benches access INCA via the ASAM MCD-3MC remote control interfaces for MATLAB and for other windows applications via the Microsoft COM-interface. [8]

Main functions of INCA are summarized in Fig. A2.
### Functions at a Glance

**Calibration**
- Online/offline calibration, 2 page (working/reference page)/1 page concepts, multiple ECU/processor access, management of limited calibration RAM by tool, support of dependent and adaptive parameters
- ECU calibration via ETK, CAN (CCP, XCP, KWP, SAE J2534), K-Line (KWP2000, McMess), USB (XCP), Ethernet (XCP), and FlexRay (XCP)
- Integration of ETAS ECU and Bus Interfaces (ES500, ES1000)
- Integration of virtual devices that connect to simulations

**Calibration Data Management**
- Database for the consistent management of projects, parameter data sets, experiment and hardware configurations with import and export features supporting team work
- Compare and copy of calibration data, function view, management of calibration dataset versions

**Measurement**
- Acquisition and online display of measurements, calibration parameters (scalars, maps), comments, online calculation and display of derived signals, xt/xy-oscilloscope and tabular online display
- Bus monitoring on CAN, LIN and FlexRay, sending of CAN messages
- Integration of Ethernet (ES400, ES600), ES1000, CAN, and SMB (L4) measurement devices
- Unattended measurements with ES715 Drive Recorder

**Measurement Data Analysis**
- Xt/xy-oscilloscope, tabular data representation, cursors, offline trigger, signal calculation, statistical analysis

**Flash Programming**
- Flash programming via ETK, K-Line (KWP2000), and CAN (KWP, UDS, CCP, XCP) with integrated PROF flash tool
- ODX based Flash Programming via CAN (KWP, UDS) and K-Line (KWP2000) with integrated ODX flash tool

**Test bench connection and automation interfaces**
- ASAM MCD-3MC V2.0/V2.1.0 on RS232, ASAM MCD-3MC V2.1.1 on Ethernet, ASAM MCD-3MC V1.0, MATLAB®-API, and COM-API

**Supported data exchange formats**
- ASAM MCD-2MC V1.5.1 ECU description
- ASAM MCD-2D V2.0.1 (ODX) diagnostic data description
- Intel Hex, Motorola S19 hex formats for calibration data exchange
- DCM and CVX for calibration data exchange using physical representation
- MDF, ASCII, DIadem-ATF, FAMOS, MATLAB®-M measurement file formats

**Supported bus description formats**
- FIBEX (FlexRay)
- LDF (LIN)
- CANdb bus descriptions and CANalyzer logfiles

---

*Fig. A2. INCA functions at a glance [8].*