Master’s Thesis

Integrated Test Environment

Daniel Andersson

Reg Nr: LIU-IDA/LITH-EX-A--13/032--SE
Linköping 2013

Linköping University
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden
Master’s Thesis

Integrated Test Environment

Daniel Andersson

Reg Nr: LIU-IDA/LITH-EX-A--13/032--SE
Linköping 2013

Supervisor: Sandahl, Kristian
IDA, Linköpings universitet
Jemander, Thorbjörn
Autoliv Electronics
Tomas Hellberg
Autoliv Electronics

Examiner: Rezine, Ahmed
IDA, Linköpings universitet

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden
Title
Integrated Test Environment

Abstract
To implement a command line interpreter is normally an easy task. The task gets harder when adding requirements of multi-instance functions and the system is to run on a multi-processor security critical embedded system. This thesis describes a first iteration of the system development. The project behind the thesis consists of requirement elicitation, design, implementation and unit testing. The result from the project is a working first version of the system.

Keywords
Integrated Test Environment, CLI, Distributed Command Line Interface
Abstract

To implement a command line interpreter is normally an easy task. The task gets harder when adding requirements of multi instance functions and the system is to run on a multi-processor security critical embedded system. This thesis describes a first iteration of the system development. The project behind the thesis consists of requirement elicitation, design, implementation and unit testing. The result from the project is a working first version of the system.

Sammanfattning

Acknowledgments

I would like to thank all my supervisors and my examiner, this could not have been done without you. Many thanks to everyone at Autoliv who have supported and helped me in my work. I also want to give thanks to Bengt-Arne Nyman for giving me the opportunity to do this thesis work.

Special thanks to mom and dad for their support during both this thesis work and during my entire university time.

Daniel Andersson
# Contents

1 **Introduction** .................................................. 1
   1.1 Abbreviations ............................................. 2
   1.2 Objective ............................................... 2
   1.3 Problem statement ........................................ 2
   1.4 Limitations .............................................. 3
   1.5 Outline .................................................. 4

2 **Background** .................................................... 5
   2.1 The System ............................................... 5
      2.1.1 Autoliv Vision System ................................ 5
      2.1.2 ECU .................................................. 5
   2.2 Previous work ............................................ 7
      2.2.1 Telematic ........................................... 8
      2.2.2 hwtest ............................................. 8
   2.3 Third Party Software ..................................... 8
      2.3.1 Tiny Shell ........................................... 8
      2.3.2 LUA .................................................. 8

3 **Requirements** ................................................ 9
   3.1 Requirements Elicitation process ......................... 9
   3.2 The Requirements Document ................................. 10
   3.3 Analyzing design requirements in brief .................. 10
      3.3.1 Two CLI’s per processor ............................ 10
      3.3.2 Host Program or scripting language on host ....... 10
      3.3.3 List of command structures ......................... 10

4 **Design and Architecture** ..................................... 11
   4.1 Splitting the system into two parts ....................... 11
   4.2 The hardware ITE-system design ............................ 11
   4.3 Design goals and questions for the software ITE-system .. 12
   4.4 Patterns .................................................. 12
      4.4.1 Command pattern .................................... 12
      4.4.2 Broker pattern versions and Data transfer object .. 14
      4.4.3 CSI/CSU model ...................................... 18
      4.4.4 Facade and Adapter .................................. 19
Chapter 1

Introduction

This thesis describes the development of a software application. It mainly handles the system being developed, but it also contains information about the process of development.

The software is supposed to run on external hardware, a digital box with a size a bit smaller than a book. This box has a camera which gives frames as input to the system. It is placed in a vehicle and can send signals to the vehicle, for example to the breaks. Since it can control things in the car, it can thus be called an Electronic Control Unit – ECU, which it will be called in the rest of this thesis. What this box does is described later, but is somewhat irrelevant to the work of this thesis.

Normally the only input to the system is the frames from the camera. This input is processed in the unit and the system outputs signals to the vehicle. In this work a backdoor into the system is being developed. This is desired to be able to test the system. It is supposed to help the developers to see that the system is working correctly. The backdoor to the system is created by connecting a cable between the ECU and a computer. When this is connected one can access a command line interface – CLI, a way to textually write commands for the system to execute. These commands could for example be commands to see a state in the code executing, for example which is the current frame number being processed from the camera. One example of what one could enter to the CLI is shown below.

```c
while (actualCameraFrame() < 20) do
    print("frames have been inputted to the system");
i=i+1;
end
```
1.1 Abbreviations

ECU – Electronic Control Unit, the device that contains the hardware and software and are to be installed in a vehicle.

CLI – Command Line Interface.

MCU – Micro Controlling Unit.

ASIL – Automotive Safety Integrity Level. The safety classed functions, they should not share any memory with the not safety-classed functions.

ITE – Integrated Test Environment. Also the name of the system to develop.

RPC – Remote Procedure Call, invoke a function on another processor or address space.

VP – Video Processor

API – Application Programming Interface

CSI – Computer Software Interface, an independent specification for an interface which one or more modules implement

CSU – Computer Software Unit, the basic “module” of a system, which implements at least one CSI

1.2 Objective

The objective of this thesis work is to construct a command line interpreter and a set of functions associated with commands in the CLI, to be integrated in the ECU. The prime reason of the CLI is to execute tests on the ECU. The CLI-system will have users from different departments, the departments which are stakeholders to the system is Software development, Hardware development, Test department and Production department. The hardware department wants this system to be able to do initial tests on the systems. This is tests to see that the hardware is working correctly and to verify the hardware design. The software department wants to see different states of modules in the code, get logging information or trigger certain executions. The test department wants for example to simulate errors in the system to see how it reacts. The production department can use this system to test the units with errors which are returned from the customer, for example to see what things were identified as objects by the system.

This system does not exist before the thesis work, neither are the requirements for the product elicited. Thereby the project will contain requirement elicitation, architecture, design, implementation and verification.

1.3 Problem statement

From the beginning, what is known is that Autoliv wants a CLI for testing on the ECU, but not what functionality there shall be. The requirements for the test system must be elicited and written in such a way that they are testable, implementable and able to fit into the existing requirements database system of Autoliv. Since the stakeholders are from both software and hardware department
it is not guaranteed that the requirements will be consistent with one design, there
might be a need for two systems.

The ECU consists of one MCU and some other processors. It is a real-time
embedded system. The scheduling of functions to different processors is done at
compile-time. A deterministic processor-scheduling is used, but for every recom-
pile the developer might decide to have a function in another processor. This rises
of one design problem for the CLI. Also, it exist functions which are safety classed
(ASIL-functions), these functions may not be interfered by non-safety classed func-
tions. To address this the memory managing unit is set up to separate the address
spaces of the two categories of functions. This must be considered at CLI design.
The multiprocessor architecture gives rise to a way of looking at the system, when
communication between processors works, they form a distributed system, but
when the communication is not up, they become lone entities. The system shall
also be used in different stages in the development cycle, which means that all
driver-software is not available at all times of running the CLI.

When designing the CLI, these dynamic aspects have to be considered. For
example the CLI could be a distributed program, it could be central and connected
to via a client from each processor, or there could be one CLI per processor. Also
the registration of functions to the CLI must work for the dynamic environment.

The underlying software has a layered architecture. The CLI will be in the
framework layer, with application layer above and OS, drivers, hardware abstrac-
tions and hardware below. The CLI has to be able to call functions in all layers.
The top layer, which is made of an abstraction called jobs, is temporal dependent
on the arrival of messages, propagating from processing of camera frames. The
functions implemented in the CLI will typically be able to listen to events, or read
states from jobs when they are being run. This results in the CLI functions being
sometimes temporal dependent.

The CLI will also be in the ECU at delivery, so there is a limit on what impact
it might do and its size.

The registration of one command can be done from all the instances of the code-
structure having the command. This forces the CLI to handle multiple instances
of the same function, which are separable in what code-structure they work on.

1.4 Limitations

To finish the whole system is not in the scope of this work. The workload of
implementing all of the CLI functions is estimated to 10 man-years. The imple-
mentation phase of this work will be very limited in time and will not contain all
the to-be functionality of the system. Further, what is being focused on is not the
functions the CLI are to provide but how it will provide it. The security is not
implemented in the first release. The requirements which have been implemented
are the once focusing on the CLI design and functionality, not the CLI functions.

The requirements elicitation phase will cover the whole CLI including its func-
tions. The design phase will cover the context of the CLI and how it shall operate
in sense of recognizing commands and being able to execute them. This is also
the focus of the implementation phase. The modules recognized as needed in the requirements elicitation phase is a future work of this project.

1.5 Outline

This thesis describes the whole process of developing a first version of ITE-system. It begins with what has been done and what can be used, and continues with describing the requirements elicitation phase. The complete requirements document can be found in appendix A.

When the requirements are elicited, they need to be analyzed for the design and after this can the design phase begin, as well in reality as in this document. In the design chapter, chapter 4, different applicable design patterns are presented which fulfill some desired characteristics of the system. In the design chapter, the system context is also presented.

Chapter 5 describes some security designs related to the CLI, while chapter six describes some interesting details from the implementation.

A small chapter regarding testing of the system is found in chapter 7, right after the Implementation chapter, chapter 6. As done in the project timeline. Concluding remarks and discussion about further work are finally presented in chapter 8.
Chapter 2

Background

This chapter presents the system which will use the ITE, what previous work has been done and the third party software that can be used.

2.1 The System

2.1.1 Autoliv Vision System

The system in context is the Autoliv Vision System [4] which is a driver assistance system for recognizing pedestrians and other objects. The system is mounted in a car. The system is a camera with belonging hardware which runs algorithms for the detection.

2.1.2 ECU

The ECU is built with several processors, one of them is an MCU and the others are called the video-processors, see figure 2.3. The underlying operating system running on the video-processors is a safety-critical real-time operating system. Every processor will have some dedicated tasks to run, which are triggered from the arrival of frames from the camera. Thereby the processors have a sort of cyclic scheduling [5], dependent on the arrival of camera frames. This system shall be extended with an ITE-system, with a CLI to take commands to execute tests on the system. When the CLI is running there will be different numbers of drivers implemented depending on where in the development cycle the ITE-system is used. Every processor is a multicore processor and also have a division into ASIL (safety-classed) functions and non-ASIL functions, see figure 2.4. These two types of processes may not share any memory between each other. Figure 2.1 and figure 2.2 show how the information of the functions need to propagate to the CLI.
Figure 2.1. The functions which the ITE shall be able to run are located on different virtual processors. They all need to be known by the CLI.

Figure 2.2. The functions registered on different processors need to be registered to the CLI.

The MCU will have the only connection to the system when it is complete, but during development all processors are accessible and shall therefore be able to make a connection to the CLI. The software for the system is written in C89(ANSI C) with a use of an Autoliv inhouse developed version of Object-Orientation with interfaces from C-structs of function pointers and corresponding implementations. This is further described in section 4.3.3. The software is also structured in a Layered Architecture[1][6], where the highest layer – The application layer – is the one most temporal dependent on the frame arrivals from the camera.
Figure 2.3. The system is built from many cores which can have different ways of communicating with the outside world.

Figure 2.4. Every processor is a multicore processor (SMP, Symmetric Multi Processor) and also has a division into ASIL (safety-classed) functions and non-ASIL functions. This division is made by a memory region wall (the dotted lines).

2.2 Previous work

Some previous work have been done by Autoliv in this area which is helpful for the project.
2.2.1 Telematic

Years ago Autoliv had a similar system as the ITE. This was running on a telematic product. Since it was long time ago, the knowledge of how it worked and how it was designed is partly lost. However, the knowledge remaining about the concept eases the requirement elicitation phase because of people knowing more what the system will do and what they want from it.

The design of the ITE for the Vision System will also differ a lot from the design of the ITE for Telematic system. The telematic system was a single core, single processor system while the Vision System is a multiprocessor, multicore system.

2.2.2 hwtest

hwtest is a smaller hardware test-shell written to the MCU of the current Vision project. It is written at Autoliv and hence it is possible to use code from it in the ITE project. The hwtest program is based on Tiny Shell [8].

2.3 Third Party Software

The following third party software have been used, the prime reason for selecting these softwares is their small size.

2.3.1 Tiny Shell

Tiny Shell [8] (TinySH) is an open source minimal shell implementation in C available under LGPL license [7]. It is based on registration of commands with a command-struct of a function-pointer. It is easy to use with an UART connection though it processes one character at a time in its input. Tiny Shell has a very small source, only 20kB, this makes it very applicable for using if the memory-footprint has to be kept low. Tiny Shell can be used and extended for the project, because of the LGPL license and its dynamic command adding behavior. Tiny Shell provides no means of scripting, it is just a shell to add a command, find it and execute it.

2.3.2 LUA

LUA [14] is a lightweight scripting language. The LUA source is about 500kB large and provides good scripting possibilities. It is a scripting language based on associate arrays, it is widely used in gaming industry. It is easy to find references online how to use LUA, thereby a big amount of documentation can be saved by using such a language instead of defining an own scripting language. LUA is licensed under MIT license [18].
Chapter 3

Requirements

This chapter first describes the elicitation technique used and then presents the requirements document.

3.1 Requirements Elicitation process

The eliciting of the requirements differs somewhat from the normal elicitation process [1]. The stakeholders for the ITE-system are developers, where some of them are potential developers of the ITE system itself. Having stakeholders who are context aware and have good implementation knowledge makes the requirements given to be more design and implementation constraining than usual. This is to be regarded as both an advantage and a disadvantage, since restricting requirements give constraints in the solution space. However also the stakeholders are experienced in developing and designing similar systems and hence are likely to give design and implementation requirements that are beneficial for the system design. The requirements are given both for the desired functionality and the implementation constraints.

The elicitation process was made much as described in Software Engineering [1] and by help from Autoliv-employees [2]. The elicitation was made by interviews of different stakeholders. After the interview, the requirements were formulated and sent to be audited by the stakeholder. After the requirements were stable the stakeholder were asked to prioritize the stated requirements. Stakeholders were interviewed in groups per department. All requirements were merged and stated in one requirements document. There was a review process [1] of the requirements with all the stakeholders and project leaders.

Having stakeholders from different departments giving the requirements separately raises the possibility of conflicting requirements. In general for ITE it is not a big problem but some problem might rise with the driver dependency. Software department requirements have a much higher demand on implemented drivers then the hardware department does. It is a possible conflict with the requirement from the hardware department to have the system to be able to work early with as few implemented drivers as possible.
3.2 The Requirements Document

The full document with requirements, use-cases and requirements module assignments is found in Appendix A.

3.3 Analyzing design requirements in brief

Some of the requirements given make direct design constraints on the ITE-system, these constraints are discussed below. COMMONX and SWX refer to requirements from the requirements document found in appendix A.

3.3.1 Two CLI’s per processor

Reading the requirements one can deduce that two CLI’s per processor is needed. First COMMON1 says that if processor communication is not working one CLI shall work per processor. This obviously forces one CLI per processor. SW15 states that non-ASIL functions shall not interfere ASIL functions. If there were only one CLI per processor, it would be either ASIL or not. In both cases it would need to point to functions in the other space, and since this is not allowed we cannot have only one CLI actuating the functions. By this there is a need to have two CLIs per processor, one ASIL and one not, or one non-ASIL CLI plus a component in the ASIL part that is able to execute the command given the name. These two CLIs, or CLI and actuator, have to communicate, but they do not share the same address space by the requirement. So this is the same situation as communication between processors – they too do not share the same address space. When the communication is developed it has to be general enough to allow communication between ASIL space and non-ASIL space.

3.3.2 Host Program or scripting language on host

COMMON2 states that it shall be possible to run scripts. To be able to run scripts either a scripting language has to be implemented on host, this would preferably be done by using a third-party solution, or a program needs to be done on host that can sequentially send commands to the CLI.

3.3.3 List of command structures

The developers need to be able to add commands dynamically to the CLI, this is stated in SW14. To apply to this requirement it is feasible to use lists of command structures. Encapsulating commands into list nodes it is possible to add more commands for the CLI to search from.
Chapter 4

Design and Architecture

4.1 Splitting the system into two parts

Looking at the hardware requirements, for example to be able to write to a pin or a memory address, makes it impossible to run the hardware tests at the same time as the system is running in normal mode. When the system is running in normal mode it will have an operating system and drivers. An operating system will block the write of certain memory regions, ex. it is in kernel space. The drivers will be handling certain pins and will thereby block general purpose input and output on those pins.

Another requirement given by the hardware department is ”It shall be possible to access all the drivers which are implemented”. This is not compatible with the requirements to read and write pins. Two options arise: make two startup modes for the same system, or split it into two systems, one for using when system is running and one for using directly on the hardware. It would be a goal in itself, for the convenience of the user, to have only one system, also if the system is divided the hardware department have to use both of the systems. However the advantages for splitting the system are too big. A split should be made. This is justified by a study of the requirements: the hardware functionality need no means of distributed functionality of the system, it needs not to conform to the CSI/CSU standard, and even if it is the same system it would need to be two distinguished modes. Bringing all the software required parts to the hardware part would only make it harder to implement and make the system available later in time for the hardware department. This is discussed with and agreed upon by the Senior Software Architect responsible for the project [12].

4.2 The hardware ITE-system design

The hardware ITE-system is to give the functionality enforced by requirements HW1 to HW5 and HW7 to HW9, which can be found in Appendix A. Further, the system needs not to be distributed, it shall work on every processor running
alone, that is, there shall not be any dependency for any other processor to work. The hardware department will need to use both the software and the hardware versions of the ITE-system, thereby the hardware ITE-system needs to be as close to the software ITE-system as possible in how to interact with the system.

For the hardware ITE-system it may not be possible to run LUA [14], it will be an advantage if possible because the hardware ITE-system must support at least sequential scripting possibilities and LUA would provide on target scripting possibilities. If it is not possible to use LUA, Tiny Shell will be used [8]. This is used in the hwtest program already implemented. It fulfills the requirements on command-structures. There will be a host program developed to support on host sequential scripts, CAN communication and file transfers, so the scripting possibility for the hardware ITE-system can be taken from there.

4.3 Design goals and questions for the software ITE-system

From the understanding of what needs to be done, and from the requirements, some design questions rise. The main design questions are listed below and discussed in section 4.4.

- Make a design that facilitates the implementation of the requirement to be able to dynamically add functions to the CLI.
- Given three nodes with dedicated commands, how does the system know where the commands exist and how to call them?
- Make a structure of the ITE-system, which components shall exist and in what context is the CLI.
- A way to implement remote procedure calls (RPC).

4.4 Patterns

This section identifies and presents some patterns that can be used to solve the goals and questions stated in section 4.3.

4.4.1 Command pattern

Requirements say that there is a need of dynamical commands, a possibility to add commands and then invoke them. The command pattern is an object-oriented design pattern which fulfills the command structure desired by the requirements, to encapsulate commands and to be able to store them. Figure 4.1 and 4.2 is the general version of the command pattern [3].
This can without big effort be modified to an ITE-system version in C.

- Invoker = CLI
• Receiver = The environment of the function-pointer, this comes implicit with the function-pointer

• Command = A struct with one field being the function-pointer, one being the name etc.

• concreteCommand = One variable instance of the Command. This is then to be viewed as the instance of the concreteCommand.

### 4.4.2 Broker pattern versions and Data transfer object

One question to solve is how to make the system know where the commands exist and how to call them, for handling this, the broker pattern is considered.

The broker pattern is a pattern for distributed systems. It is a pattern describing how to locate services [9]. This can be combined with the data transfer object pattern which is a pattern for wrapping data to transfer only one object with all the information needed, instead of sending multiple times what is asked for [10].

In the broker pattern a singleton class named broker is located on a known node, and the broker is responsible to locate the services. The client who wants to make a remote call, uses a proxy that hides the distribution and allows the client to view the system as central. The proxy calls the broker to locate the service. A pure version of this in the ITE-system domain is shown in figure 4.3.

![Figure 4.3. The sequence for a call using the pure broker pattern, one central singleton broker located at a known node. For a bigger picture see Appendix B, figure 1.](image)

For this structure the space needed to store the commands would be twice the
number of commands in the system, each CLI will have its own commands and the broker will have all the commands. For a command stored on processor A to be executed from the CLI of processor B four interprocessor sends have to be performed, wrapperA → Broker → wrapperB → Broker → wrapperA.

<table>
<thead>
<tr>
<th>Dataspace:</th>
<th>2 ( \times ) number of commands in system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication:</td>
<td>4 sends/command</td>
</tr>
</tbody>
</table>

**Table 4.1.** Costs of central broker knowing all the commands.

**Modifying the design, broker version 2**

Now think outside the pattern, how can it be modified and what can be gained? First, the total space used was twice the number of the commands, if instead the broker does not know the commands, it would have to ask all the other processors for the command, figure 4.4 illustrates this.

**Figure 4.4.** The sequence for a call using a modified broker pattern, central singelton broker with no commands. For a bigger picture see Appendix B, figure 2.

This version will reduce the dataspace needed but use more communications as the broker needs to ask all other CLI’s if they have the command. If the broker sends one per processor and the processor splits it to the two CLIs it will be 6 sends, but if it sends once per CLI it will be 12 sends.
### Design and Architecture

<table>
<thead>
<tr>
<th>Dataspace:</th>
<th>1</th>
<th>× number of commands in system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication:</td>
<td>6 or 12</td>
<td>sends/command</td>
</tr>
</tbody>
</table>

**Table 4.2.** Costs of central broker not knowing the commands.

**Modifying the design, broker version 3**

Next step of modifying violates the broker pattern but might be a design option. For the sake of consistency, the class will still be named broker. Here the broker is not central anymore, instead its one broker per CLI, having this we can also collapse the dataWrapper with the broker on the sending side, this gives the diagram shown in figure 4.5.

![Diagram showing modified broker pattern](image)

**Figure 4.5.** The sequence for a call using a modified broker pattern, the broker is local. For a bigger picture see Appendix B, figure 3

By having the broker local we reduce the communication worst case numbers, the call does not have to go the extra way to the broker. Instead the number of commands to store grows bigger as every broker will store all commands.

<table>
<thead>
<tr>
<th>Dataspace:</th>
<th>6</th>
<th>× number of commands in system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication:</td>
<td>2</td>
<td>sends/command</td>
</tr>
</tbody>
</table>

**Table 4.3.** Costs of using local brokers knowing all the commands.
Modifying the design, broker version 4

The local version of the broker can also be used without knowing the commands as shown in figure 4.6.

Now the dataspace used goes down to its minimum while the communications used again go up, here again the number depends if the broker sends once per processor or once per CLI. Note that for the brokers not knowing the commands the communication used is expressed exactly (that is $\Theta(n)$ [11]) while for the brokers knowing the commands it is the worst case (that is $O(n)$ [11]).

<table>
<thead>
<tr>
<th>Dataspase:</th>
<th>1</th>
<th>× number of commands in system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication:</td>
<td>4 or 8</td>
<td>sends/command</td>
</tr>
</tbody>
</table>

Table 4.4. Costs of using local brokers not knowing the commands.
Choosing the broker design

The communication numbers in the above tables is worst case for the broker knowing the commands, which is when the broker, the calling CLI and the executing CLI are in different processors. However have in mind also the distinction of address-spaces in one processor giving that we have six CLI’s on six “virtual” processors. Different designs can be used in that aspect as well, even having a local broker it can still be global for that processor, that is one broker for the two CLIs. If the broker call also encapsulates to whom the other end shall answer the communications is reduced.

Regarding the computational complexity [11], the options with a broker knowing the commands will do first one search in the local CLI commands, then the broker will do a search in all the other commands, and at last the CLI containing the command will do one search for it. If the broker does not know the commands, the local CLI will search its commands, after that the broker will send the request to all other CLIs to search for it, thereby the same amount of commands are searched in both cases giving the same computational complexity, but notice that for the broker not knowing the commands the workload will be distributed. What has been investigated above is the cost of executing a command, other operations as listing commands and adding commands have not been regarded yet. The operation to list commands uses much communication in option one and two since the CLI will ask the remote broker for it, while using no communications at all in case three with the local broker already knowing the commands, and in case four the local broker will ask all others, so there too it will be much communication. Instead when adding a command, case one needs one send to add it to the broker, case two needs no communication, case three will need a lot of sends to distribute the command to all the brokers. The list-commands command will probably be used more frequently then operation to add a command. This gives some favor to option three.

The whole system will be more constrained in communication bandwidth than space, therefore minimizing the communication overhead would be considered most important, which favors option three. So in agreement with the Senior Software Architect responsible for the project[12], option three is chosen to be the version to use.

4.4.3 CSI/CSU model

The CSI/CSU model can be seen as an Autoliv in-house developed version of object orientation. Using the CSI/CSU model makes the code to be more modularized, the structure reminds of object orientation. A CSI is an interface showing the operations that can be called while a CSU is an object containing different interfaces and providing the underlying functionality. This endorses the good design practice known as the dependency inversion principle [13]. This is implemented with C-structs as shown in the code fragment below, and modeled as shown in figure 4.7. From a CSU it is then possible to call the functions from the interface.
4.4.4 Facade and Adapter

Since the CLI can possibly be implemented with Tiny Shell or in LUA there is a need to wrap the CLI. This is done by putting a facade[3] around the implementation. The facade is though mostly an adapter of LUA to look like Tiny Shell, this wrapping is described in section 4.7.

4.5 CLI environment

The CLI is to be put into its context. As mentioned above, the broker version three is to be used, so there will be local brokers. It is beneficial to have as little code as possible running in ASIL-mode. The broker can be put into the non-ASIL mode only because the ASIL-CLI will only be asked to perform commands which it knows, this is because no user can connect to the ASIL-CLI. This gives figure 4.8 which is a coarse view of the system. The ITE-system also has to adapt to the CSI/CSU model.
Figure 4.8. The CLI in its context. The ASIL-part is pure C while the CLI not in the ASIL part might be LUA.

Decision for implementation in C or LUA

- The CLI shall be possible to use whether it is implemented in LUA or C. In the video-processors it should be no problem running LUA, but on the MCU, LUA might require too much memory. The MCU is very constrained in memory. Thereby being able to change at compile time if to use LUA or C is desired.

- ASIL-CLI shall be implemented in C, there is no gain of using LUA for that part and the ASIL-code shall be kept as small as possible.

- Communication module shall be implemented in C.
- The CLI proxy will be different for the ASIL part and the non-ASIL part, but in both cases it will be implemented in C.

- For the broker it would be easier to implement it in LUA. However a LUA implementation would both be larger in size and memory footprint. Also if the broker is implemented in C, it is possible to use it for both LUA-CLI and tinySH-CLI version. Thereby, the broker is implemented in C.

The CLI will need to have a wrapper to make it invisible if it uses Tiny Shell or LUA. A facade should be made to hide the implementation. The interface to the facade is chosen to be close to Tiny Shell, thereby it is like an adapter making LUA look like Tiny Shell but also slightly modifying the Tiny Shell interface first.

The communication between the processors can be done in two ways, shown in figure 4.9 and 4.10.

![Diagram](image)

**Figure 4.9.** Version A: The broker can send to the ASIL part of the other processor.
**Figure 4.10.** Version B: The broker can only send to the non-ASIL part.

Version B is chosen to keep as few communication channels as possible to the ASIL-parts.

The refined model can be seen in figure 4.11
4.6 Broker

There will be one broker per processor, and the basic principle of the broker can be found in figure 4.5. As explained in the previous section the broker will be implemented in C. The responsibility of the broker will be to, given a string, find the processor having the command and send the execute request to that processor. This is not far off from what the CLI does, therefore the broker code can be a modification of the C version of the CLI, the Tiny Shell version. The broker will contain one command-list per processor in the system. When a command is to be registered to the broker the register command shall both state the command itself and the processor number of the virtual processor containing the command. The processors could for example be numbered as shown in table 4.5.

The broker will expose the following functions:

- void broker_addCmd(IBroker* me, ITE_cmdType *cmd, int processorNum)
  - add the command located at processor processorNum to the broker.
• void broker_exec(IBroker* me, ITE_outputFnType output,) - execute the command given in the string. The string can contain arguments, either as foo arg1 arg2 or as foo(arg1,arg2).

When using LUA-CLI there will be some adaptions to make. This is explained in section 4.7 but influences the broker in a way that it will have a private function broker_execFromFnCall(ICSU* me, ITE_outputFnType output, int argc, char** argv) which treats the first argument as the name of the function to call. It will also have a function to tell its function-set to the LUA-CLI: giveBrokerNamesToLuaCli(). The broker object is shown in figure 4.12.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MCU</td>
</tr>
<tr>
<td>1</td>
<td>MCU-ASIL</td>
</tr>
<tr>
<td>2</td>
<td>VP0</td>
</tr>
<tr>
<td>3</td>
<td>VP0-ASIL</td>
</tr>
<tr>
<td>4</td>
<td>VP1</td>
</tr>
<tr>
<td>5</td>
<td>VP1-ASIL</td>
</tr>
<tr>
<td>N</td>
<td>VP(N/2)-ASIL</td>
</tr>
</tbody>
</table>

Table 4.5. Example of virtual processor numbers of the broker.

4.7 Wrapping of the CLI

To wrap the CLI to be both LUA and Tiny Shell compatible gives rise to some problems. The C-functions which they can call are type-defined by

```c
//LUA:
typedef int (*lua_CFunction) (lua_State *L);
```

```c
//TinySh:
```
4.7 Wrapping of the CLI

typedef void (*tinysh_fnt_t)(int argc, char **argv);

//Which means that the user functions are to be defined like
int f(lua_State *L)
void f(int argc, char** argv)

To call a function in LUA, the syntax f(arg1,arg2) is used, while in Tiny Shell
the syntax f arg1 arg2 is used. The user must not see the difference between the
two interfaces, neither shall the one registering the functions. Thereby the facade
interface will abstract the interfaces so that they become the same. The Facade
interface will be:

- void charIn(unsigned char c) - input character from user of CLI, a newline
  or carriage return will trigger the execution of the inputted string.

- void addCommand(ITE_cmdType *cmd) - the command cmd will be added
  and should have a function to which it points.

Now the interface looks similar to Tiny Shell interface, to use the Tiny Shell
CLI version a forward of the request can be made. The work-effort will be in
making this abstraction work for the LUA CLI version.

4.7.1 ‘add’ and ‘execute’ command to LUA CLI version

When a call to a registered function shall be made it will be inputed as foo("arg1")
to the system, thereby LUA has to know the existence of the function and be able
to call it. LUA has a built in register function which can register C-functions to
be able to call from the LUA interface. However using this we would have to use
the lua_CFunction functions, and what is required is to use ITE_cmdFunction.
Thereby it is not possible to register the function as normal to LUA. If instead
there would be just one C-function which would be called by LUA there would not
be the problem of different function types. That is, when a registration is made
its made with the name of the function but with the pointer to the wrapping
C-function. The problem here is that when the C-function gets called it will not
have access to the real function pointer that is to be called.

The solution to the problem is to make the registration not as a C-function
but as a LUA function. At initialization the C wrapper function is registered and
also a LUA table containing function data. Then when a call to add command is
invoked a string is built and executed in LUA which is the definition of a LUA
function pushing the needed name and function pointer to the LUA stack and
then calling the C-function. The C-function does then have enough knowledge to
proceed with the operation. Then when an operation is invoked it will be a call
to the C-function which will get the real function pointer and execute it. This is
shown in figure 4.13.
4.7.2 Use of the broker

Problems arise when a function is not found, LUA returns a message of the form 'attempt to call global "foo", a nil value.' It might be the case that the operation is in another processor, this has to be checked with the broker.

- If a check is to be done before the LUA call to the function, parsing of LUA commands would have to be done in C which is not feasible.
• If a check was to be done afterwards, that is, capturing the error and then asking the broker, we would have destroyed possible scripts running. As an example if we run the script
while(!stop) stop = foo()
and foo is located at another processor the script would break, return the error and then do one call only to foo.

• If one should rewrite the LUA function call to handle the error in LUA, and from there call the broker, it would work. However it is not desired to do any changes to the LUA core.

• If the broker tells the LUA-CLI what functions it has, in a similar way as the addCommand does, then the command names would be duplicated. However instead achieve a nice and easy design which is also similar to the way normal commands are added, thereby being consistent with the current design.

The broker implements a function where it calls the addCommand for all its commands. Before doing this it sets the function pointer of the command to point to function broker_exec_from_fnCall which is consistent with the ITE_cmdFunction type.

4.8 Remote Procedure Calls

If processor A is to invoke a command on processor B, this call is remote. Remote procedure calls needs to be implemented. This is not a new problem but in the domain where it shall be used there are no implementations of RPC. When doing a RPC from the CLI it needs to be performed in an ordered, and preferably synchronous manner. For example if we do two consecutive calls we do not want the answer from the first one to arrive after the second one. Also the transmission of the call must be relied upon.

First approach - non-working way

If the CLI starts being asleep and wakes up on messages, that is, works as a normal job, then at system initialization the system shall send a message to the CLI to wake up and accept input. The CLI reads the input, and calls the function by sending a message. When the message is sent it goes asleep. The message is received, the function invoked and the function answer is sent back as a message to the CLI. When the message comes back the CLI wakes up, shows the answer and then accepts new input.

An advantage of this is that all calls would be handled in the order they arrive, also events could use the same message way to communicate to the CLI. So using this design would also help in designing for event handling.

This approach does not work with CLI on target scripting. Assume we have a script:
If the CLI goes asleep after doing the first `rpc()` it would not next time remember were in the script to execute and that it should execute. A workaround for this might be possible but other ways of implementing RPCs will be better.

**How do the CLI wait for answer**

The CLI cannot be awake and busy-wait for the answer, it has to sleep, also as said above it cannot be sleeping and wakening on messages, so it has to be blocked by someone in the chain of call forwarding. The sequence of sending the message is seen in figure 4.14.

![Call sequence for RPC](image)

**Figure 4.14.** Call sequence for RPC.

Here the CLI must be blocked by one of `c_call`, Broker or Transmitter.

**Using the job abstraction combined with semaphores**

There are many existing ways of solving the problem of Remote Procedure Calls (RPC), but looking at what the system already provides a design can be done using the system-job abstraction. This will benefit in the way of code consistency, the other developers will more easily understand the way RPC works in the system.

A job is an abstraction to be in a chain, connected to other jobs. The typical job takes one or more messages as input and produces one or multiple messages as output, a job is triggered when all the input message connections have a message. Of course there can be start-jobs (named trigger-job’s) and end-jobs. Start-jobs

```plaintext
for i = 0..40 do
    rpc()
end
```
take no input and end-jobs give no output. A trigger-job has a trigger function that will be called in every loop of the dispatcher, when the trigger job returns the action-function of the job will be called. The implementation of remote procedure calls is made using three jobs and two semaphores.

- The call-transmitter is a trigger-job responsible of sending the call. It has one interface function for doing the RPC function-call, and one more interface function to handle the reception of the answer.

- The call-receiver job is the job receiving the call and responsible to execute it and return the output, hence this job will be a normal job taking in-messages and producing out-messages.

- The call-answer-receiver is the job receiving the answer from the calling job. This cannot be the same job as the call-transmitter since this would make a feedback loop in the system, which is not desired in this case.

Letting the trigger-job make a pend for a semaphore in the trigger-function will make it sleep directly from system startup and not take any CPU time. Then when a RPC is to be made, a function in the call-transmitter job is called, this function then posts on the semaphore after setting the internal command variable to be the RPC to be made. This makes the trigger-function to come alive and the action function to start its execution.

The action-function of the call-transmitter will send a message to the processor which has the function to invoke, after this it pends on a semaphore initialized
to 0 to make it sleep. The message it sends will trigger the execution of the call-
receiver and the function will be called. The function result will be accessible by
the call-receiver which sends this answer as a message to the call-answer-receiver.

The call-answer-receiver will be triggered by the answer message, it then calls
a function from the call-transmitter job to set the answer data and posts on
the semaphore on which the call-transmitter function is waiting. Now the call-
transmitter function continues and returns the answer of the call. The behavior is
shown in figure 4.16 and the job graph in figure 4.15.

**Figure 4.16.** Sequence diagram for the implementation of remote procedure calls. For
a bigger picture see appendix B
Chapter 5

Security

The CLI will be in the ECU’s which are sold, but the CLI is intended for internal use only. Thereby the CLI needs security to prevent unauthorized use of it. The potential ”hackers” of the system will be the customers of the ECU. They can be assumed to not be dedicated attackers. The requirements of the security part is because of that not extremely high. Also the impact of a security break in is not devastating, but a successful attack can lead to misuse of the system which in the end can lead to a safety problem. In figure 5.1 a tree is shown with possible impact of a break-in.
The potential attackers have access to the bus talking to the ECU, but they do not have direct access to the hardware inside the ECU. Therefore, for example storing a password in the ECU memory can still be done in plaintext without risking a read from the attacker.

5.1 User convenience versus Security

It needs to be weighted, user convenience versus security of the system. Adding more security to a system often reduces the user convenience, and the other way around, making the system more convenient often reduces security. This is known as the "curse of convenience" [25]. As mentioned above the severity of a security breach is moderate. The likelihood of an attacker hacking the system is proportional to the security used, not just the likelihood of a successful attack, but the likelihood of an attempt to attack the system. With some security it can be regarded as low likelihood of an attack. Risk = likelihood x impact = low-moderate.

The user experience of the CLI is regarded high. If it is too much security required for using the CLI, it will not be used. So the user cannot be forced to authorize himself too often. This must be regarded in doing the security solution.
A problem to look into is when the system restarts. If the user of the ECU was authorized for CLI operations at that point in time he will be after restart too. It would harm the user experience having to log in again after restart.

5.2 Where to authenticate

The check of privilege could be deferred to the registered functions of the CLI. This is however not a good idea. This would force all the providers of commands to the CLI to implement their own security, which introduces many different places of possible vulnerabilities. Also, if LUA is used it would not be any restriction to the CLI user to use LUA scripting without being authenticated. This is a misuse of the system which shall not be allowed. As in Privilege Separation [23], the authentication should be placed in a module on its own. This design helps to implement the security in one place only and therefore adheres to the security design principle to “keep it simple” [24]. If an upgrade shall be done to support multiple roles, this should also be handled by the module of authentication in the beginning. But then the user credentials have to be passed along since the authentication module cannot know if the logged in user is authorized for that special command. A good way to have user control of the commands is to have the credentials required to be part of the command struct, and then the CLI to check for the rights passed to it when asked to execute a command.

5.3 How to authenticate

Two normal types of authentication are “what you know” and “what you have” [25]. One way of adding security to the system would be to just have a password which gives the user the credentials needed to use the CLI. With this approach the password could be securely stored in the ECU and the developers would have to remember it. Here many people need to know the password and the password needs to be easy to remember. This introduces a high-risk vulnerability of a social engineering attack. If many people know a password that is seldom or never changed it will eventually leak out. Instead of this approach, a challenge-response solution should be used [25][26][27]. This changes scope of the protection to “what you have”. Now a program has to be developed to run at the computer. Still a password is used in the implementation of the security, but the user needs not to know it. The computer program and the ECU both know the secret password. When a login is to be made the ECU gives a random character or number sequence which the user shall input into the computer program. From this sequence called challenge, the computer program uses a one-way hash function on the challenge combined with the password to generate a response message. Input to this hash-function can also be some sort of salt[28]. Now the computer program presents this response on the screen so that the user can input it to the ECU. The ECU has done the same calculation and because of that it can compare its result to what the user enters. This is illustrated in figure 5.2
This approach both helps against the social engineering attack and bruteforcing the password. Now for an authorized user to give an unauthorized user access to the CLI he would need to send the program. This can more easily be traced and is more effort to do. Still it is not a guarantee it will not happen. One more layer of security for that risk is to add an Autoliv login to the computer program. This can guarantee that only people working on Autoliv computers can access the program. For bruteforcing of the password the hacker now also needs the challenge, the possible other salt, and the hash-function. This makes bruteforcing much more inconvenient.

5.4 System restart

When testing the system it might be some system restarts done by different reasons. Having to log into the system at every restart will harm the user convenience. A solution for the case when the CLI user want the system to do a restart would be to implement a function in the CLI telling the system to restart. This command could then write to the memory that a shutdown was made authenticated so that the system shall start in authenticated mode. This approach will however not work if the system restarts due to some failure. When in production this might still happen so often that it harms the user convenience. One approach to solve this problem is to have some bit set in the permanent memory as soon as successful log in has been made. However then if the user do not log out of the system the next one who starts the ECU will be logged in, this is a harm of security. When "production" and "release" modes are available, the CLI could automatically use...
the secure version when in production mode and the more user convenient method when in production mode. Another option is to set an amount of restarts ok to do without re-authentication. Doing this there needs not to be a separation between development and release mode.
Chapter 6

Implementation and Low-level Design

6.1 Communication and output

To be able to use command structures the function type which the command structure includes must be type-defined. This makes that the return value of the functions that can be used in the CLI must be predefined to a single type. The most convenient and generic type to return from a function is then to return a string. Returning a string the user of the function can know what return-type to expect and parse the string as that return-type. Nevertheless a function in a CLI has a main goal of providing information to the invoker, thereby it will be necessary to be able to use prints in the functions. A function invocation can be made from remote. Thereby a standard print cannot be used, since this would print the answer to the standard output stream of that processor.

A function needs to be able to print output and to answer a string. This shall be done remotely. The remote communication has only one channel of sending data. To conform to only having one channel one option is to not allow the functions to have a return-type, another is to only allow returns and not prints. Both options are somewhat limiting but can be worked around. The requirements enforce scripting to be possible, ex. while(fn(1))fn2(). Where fn2() prints useful information. So there is a need to both be able to use the output as result and to use the output as prints.

To solve the problem of interpreting data a model of user decision to interpret data was considered. That is, the user of the CLI makes an invocation and after that decides how to interpret the data. This would be easy to implement but would be more confusing for the user, the way to do the above mentioned while loop would be while(interpretBool(fn1())) output(fn2()). This is not a nice syntax and also the user needs to have perfect knowledge of what the function invoked prints and in what order.

Instead a model of both returns and prints was adopted. The trouble for this
approach is how to get the data separated when it comes back to the CLI. The CLI must be able to print the strings printed from the function and use the returned data. For separating this the functions registered to the CLI have to take an output function as parameter and returning a string. So if a local function is invoked it will answer the string and directly use the CLI’s output function. This can be seen in the code below:

```
//CLI contains:
OutputFunctionType output;  //Can be set by the invoker of the CLI
//in the invocation part:
cmd->functionPointer(output, argc, argv);

//the function to be invoked
char* foo(OutputFunctionType out, int argc, char** argv){
  //do stuff
  out("state is X");
  return "3";
}
```

Here the function asking the CLI to invoke a function can first set the output of the CLI, then use the CLI to invoke the function. In the CLI the user can get the result from the function directly. Thereby while(fn(1))fn2() needs only to be changed to while(fn(1)=="true") fn2().

The solution for the remote invocation is that the broker takes the output function as parameter from the CLI, just as a function would have done, and then calls the call-transmitter with the output function as parameter. The call-transmitter will make the remote call, get an answer, separate print and return, and then use the output function to print the string to be printed and return the answer to the broker. On the remote end, the receiver of the remote call will accept the call and set its processors CLI output-function to print to an own buffer. Then make the call, get the answer, pack the answer with the buffer printed to and send back the answer.

**Adaptions**

When using Tiny Shell the return value cannot be used since Tiny Shell only is an invoker, so when Tiny Shell is used the answer will just be printed. When using the CSI/CSU model, the interface cannot be a variable, thereby one more level of indirection is introduced. The CLI provides a function output which just uses the output variable.

### 6.1.1 Print from LUA

When a print is made from inside LUA the standard output stream is used. To redirect it, it is possible to either redirect it from C or in LUA code. The redefinition was made from inside LUA because it is no difference to the used and the redefinition from inside LUA is shorter in code. The print was made to be a
call to C with variable amount of arguments. Format strings cannot be used for the new print-function but this is not a restriction since LUA provides a function string.format which can be used in combination with print. See code below.

```c
#include

int luaPrint(lua_State *Lu) {
    int argc=lua_gettop(Lu);
    int pos=0;
    int i=0;
    char outputBuf[MAX_PRINT_BUF];
    for (i=1; i<=argc; ++i) { /* lua counts 1..n so 1..argc */
        if (!lua_isstring(Lu, i)) {
            CLI_output("Error: Not possible to make string of argument to print\n");
            return 0;
        }
        sprintf(outputBuf+pos, lua_tostring(Lu, i));
        pos=strlen(outputBuf);
    }
    CLI_output(outputBuf);
    return 0;
}

//LUA:
function print(...) luaPrint(...) end

//Example use
print(string.format("h%c llo",'e'))
```

### 6.2 Initial phase

#### 6.2.1 Function register to the CLI

The system is set up in the so called glue-code, a code that connects the different software components. This is also where the jobs are connected to each other and the CSUs are being initialized. When commands are to be registered to the CLI, some requirements have to be fulfilled, they are stated below:

- The commands need to be registered to the CLI after the CLI is initialized.
- A command needs to be fetched from the CSU having the command. That is, the CSU which is to provide a function to the CLI must be initialized before the command is added to the CLI.
- When framework->start() is run, the CLI needs to have all the commands, that is, no command adds shall appear after framework->start().

From the above requirements it is seen that the registration of the commands must be coupled to the glue-code of the framework.
If all CSUs with commands to add had an interface for this, an automatic collection of the commands could be done. It could be done in a way with pointer arithmetic so that a CSU needs only to provide the interface if it has commands to add. The problem with this automatic approach arises when the command adding function is to iterate over all CSUs. There is no way of getting all CSUs from the system so this cannot be done.

A possible approach is to make the command registrations from the glue-code. So that the glue-code asks the CSU for the commands to add, then iterate over them and add them to the CLI. However, this approach would bloat the glue-code with a lot of command adding. Also everyone wanting to add commands would then have to edit in the same file.

If instead a middle registration CSU is used we can keep the dynamics and not bloat the glue-code. The CSUs having commands to add need to implement an interface ICmdProvider which has a supplyCmds function. The framework needs to have a function registerCmdProvider to keep track of all the CSUs having commands to add. This way, the CSUs having commands to add will register themselves to the framework, the framework will then, when all is set up, do one function which iterates over all the CSUs who added themselves and ask them for their commands. That is, the CSUs push themselves into the framework which later pulls out the information from the CSUs. This structure is shown in figure 6.1 and 6.2.

![Class diagram for the alternative of the command provider knowing which interface to register to. The framework has to implement the ICmdRegister Interface.](image-url)

**Figure 6.1.** Class diagram for the alternative of the command provider knowing which interface to register to. The framework has to implement the ICmdRegister Interface.
6.2 Initial phase

Figure 6.2. Sequence diagram for this alternative. The class which will provide CLI commands registers as a command provider.

To sharpen the above design a bit, one can make the framework not needing to implement the CommandRegister interface but instead have a list or array of command providers, and when a module is added to the framework it makes a query if the module contains the ICmdProvider interface. This design favors composition over inheritance, which is a good design principle known as the Composite Reuse Principle[20]. This is shown in figure 6.3 and in the code fragment below.
**Figure 6.3.** The framework contains a list or array of ICmdProviders over which it can iterate.

```c
void FW_deploymentCreate(Framework *my, const ICSU_ParamHeader *params)
{
// ...
myCSU = MY_getCsuFactory()->create(&myParams.header, &internalMem);
// ...
handle = FW_addModule(my, brokerCSU, NULL, FW_MTYPE_SERVICE);
// ...
}

static FW_Handle FW_addModule(Framework *my, ICSU *theCSU, const FW_JobConfig *config, FW_ModuleType type)
{
// ...
theCSU->getFactory(theCSU)->queryInterfaces(&csiIdArray, &sizeOfArray);
foundInterface = AlvFalse;
i = 0;
while((i < sizeOfArray) && (foundInterface == AlvFalse))
{
foundInterface = (csiIdArray[i] == CSIIdentifier_ICmdProvider) ? AlvTrue : AlvFalse;
i++;
}
if(foundInterface)
{ /* It was a cmd provider */
ALV_VERIFY(foundInterface, CSIIdentifier_ICmdProvider, 0);
my->cmdProviders[my->numCmdProviders++] = PTR_CAST_UNSAFE(ICmdProvider*, theCSU->cast(theCSU, CSIIdentifier_ICmdProvider));
}
```
6.2.2 Broker initializing

At system startup the commands needs first to be registered to the CLI on their target processor. When all commands are registered to their corresponding processor the information about where the commands are located needs to be sent to the brokers on each processor. This enforces two points of synchronization in the startup sequence. First point to say that all commands are registered on all processors so that the CLI can start its broadcasting of commands to the remote brokers. The other point will say that all brokers have got the commands and the system can now start.

The last synchronization is for starting to use the CLI. If an assumption is made that the user cannot type anything to the CLI at the short time from the CLI-start until the command-sends have finished it is possible to just start use the CLI directly and not having the last synchronization point, but, if a script is auto-started on system startup it might be a problem. If instead the fact that all the other processors brokers have successfully sent their commands to the CLI connected to is used as synchronization-point to start, even a script cannot fail. This is because even if the other brokers have not got the commands the connected to CLI was sending those brokers will not be used. The only important broker is the one connected to, the others exist for the possibility to connect to the CLI on different processors. This argument is based on the fact that bullet number three from the broker command distribution techniques below is used. The states of the CLI is shown in figure 6.4
The broker is supposed to know where all the jobs are located. In order to do this the brokers will have to communicate their commands in the initial phase. This has to be done after all the commands have been registered to the CLI.

- One way of making the brokers to have the same command set is to do a `broker_addCommand` in the startup code on all other processors than the one having the command. Using this way the effort to do the solution is none, but the convenience of the user is harmed.

- Another way of doing it is to redefine a sequence of command sends, for example processor A sends its command set to processor B which sends the commands to processor C, processor C merges the commands and sends them back to processor A, finally A sends the merged commands to processor B again so that B also has the commands of C. This can be seen like the way of a token-ring[19]. The benefit of this approach is if a new processor is added, the number of sends is just increased by two. The negative is the complexity of implementing the ring with the send functions and synchronize it with the
other startup-phase of the system. A graph for this solution would look like figure 6.5

![Figure 6.5](image)

**Figure 6.5.** The ring version with three nodes.

- By using a job as a sender, each processor will have one receiving job and one transmitting job. The Transmitting job of the processors will have output to the other two brokers receiving units and will post a message containing its commands. The receiver can then add all the received commands to its broker. This is shown in the job-graph of figure 6.6.

![Figure 6.6](image)

**Figure 6.6.** Job-graph for sending and receiving commands in the brokers.

It will be one connection from every processor to every other processor, that is \( n(n-1) \) number of sends. If three processors are used, this is not a problem, then it is only \( 3(3-1) = 6 \) sends. But adding more processors here will have a squared impact on the number of sends.
6.3 Commands for multiple instances

A CSU is a unit made to be able to have many instances, one can think of it as a class. So if a CSU implements the ICmdProvider interface, and has commands to register, how shall then these commands from different instances be separated in the CLI? Since it is a class which provides the commands, this class can be instantiated many times in the system, for example the dispatcher of every job. So if the developer wants to add a getter in the dispatcher to get some statistics from every dispatcher when asked for in the CLI, the different dispatchers have to be separable. It is the same function but it should be invoked with different instances of the class as parameter. And another problem is, how shall it be separated if the same name is on two processors? The user might want to connect to one processor executing a command on the other processors which has the same name as a command on its own processor.

Starting from the question of how to implement it, the "how" question propagates all the way to the user. How does the user know what instances of a function there is, and how does the user chose which one to invoke? Starting from this end, the problem cannot be solved in many different ways, the user needs to identify the instance by a unique name. How this should be appended to the command the user should invoke has more options. The one decided to use is to append the instance name in the end of the function name which are registered to the CLI. So if a function named getState() is registered from myClass, the name to invoke from the command line could be getState_inst1() to separate the instances. To invoke the command on a different processor, the instance names could either be unique per processor or global. If locally unique one can also append the processor name first to the call, that is, to do the above invocation on processor2 instead, do proc1_getState_inst1().

6.3.1 Unique instance name

So what is needed then is to give all the instances a unique name, this could be generated or provided. If it is generated it could be rather inconvenient for the user to get a grip of what name goes to what instance. Therefore it is a better idea to force the developer to provide it. Also the developer can make an own generator which matches his code.

The name needs to be stored and provided somehow. In the way the system is built the create function is the best option. So when create is called it needs to get the unique instance name somehow. Three ways of adding this is:

- All creates takes an instance name.
- Add a redirectCreate which calls create and then sets the instance name
- The parameters which create already takes, can have the instance name.

The two first bullets enforce a lot of change to the already existing code. Thereby let us look further to the last bullet. The parameters sent to the factory’s create method is a param* which is a pointer to a param structure. When
the create method starts it will cast this param\* to be an own myCSU_params\*. This can be done by forcing the myCSU_params to have the first member to be a param-struct and then send the pointer to that member. Thereby the cast is just a reinterpretation and one can see the adding of information as a version of the decorator pattern[3].

The instance name could thereby be added either to the param-struct or to the myCSU_params-struct as a decorator of the param-struct. Adding it to the real param-struct is a change to the complete system, so for the time of implementation the decision is made to do it the decorator way. This structure is shown in the below code fragment and in figure 6.7

```
typedef struct BRKR_Params{
    ICSU_ParamHeader header; /* The header used for type safety of parameters. */
    AlvChar instanceName[MAX_INSTANCENAME_LEN]; /* Name for identifying the instance */
} BRKR_Params;

BRKR_Params bp;
// set bp
create((ICSU_ParamHeader *)bp, ...)

void create(ICSU_ParamHeader p, ...){
    // ...
    strcpy(my->name, ((BRKR_Params*)p)->instanceName);
}
```

**Figure 6.7.** The parameters decorated with instance name is a way of not modifying all the code while bringing the new instance name parameter to the CSU.
6.4 Miscellaneous implementation details

6.4.1 No CLI-Proxy CSU needed

When receiving a call made from remote, the call-receiver will wake up, then it needs just to set the output correctly, make the local call and get the answer from the call. It was decided to not make an own CSU for this, the receiver-side CLI-proxy. Therefore the proxy functionality was divided between the call-receiver and the CLI. The call-receiver sets the output to its own buffer-printing function and then calls CLI_proxyInput function. The CLI_proxyInput function skips the charIn step since it already has the complete command. It checks whether it is LUA or Tiny Shell being the CLI in use, and then calls luaL_dostring or exec_command_line depending on the CLI in use.

If LUA is used the output is captured by appending "__t=" to the command to be executed. Then when execution is done, the output stream is redirected and then a print(__t) is made to get the result.

For Tiny Shell, the source was modified so that the exec_command_line returns the function answer. Thereby it can be captured in different ways if the call was made from remote or local, since a remote call must return the answer while a local call shall print it.

6.4.2 How to access the call-transmitter from the broker

The broker can be entered in the exec_fromFnCall function called from c_CommandCall in the following way: The c_cmdCall gets the pointer to exec_fromFnCall as it would have been any local function and executes it as an ITE_functiontype. Then in the exec_fromFnCall there is a need to call the call-transmitter, but if the running environment is just a function how is the call-transmitter accessed? It could be a global variable for recognizing the transmitter which are to be called. But looking at the requirements, the functions registered to the CLI shall be able to "read internal states" that is, it must be able to access the CSU. Thereby all functions must take an ICSU as parameter. So this solves the problem of finding the transmitter as well. There is no need to keep it as a global but the brokers exec_fromFnCall will get the broker ICSU pointer and can thereby find its call-transmitter.

6.4.3 What to put into the API

The structure of the files for the system are divided into layers, where the ITE-system will be in the framework layer, using functions from the framework layer and the OSAL (operating system abstraction layer) and providing functionality to the application layer through an API. When a developer, either for the framework or for the application layer wants to use the CLI they need to implement the ICmdProvider interface, which says that they have commands to add. What more they have to do, and how, is further described in the ITE User Manual found in appendix C.
So it is clear that the ICmdProvider file has to be in the API delivered by the framework. Moreover the provider of CLI commands need to know the structure of a command, the typedef of the functions and some constants. This could be included in the ICmdProvider interface to expose fewer things in the API, which is a good thing. The disadvantage with doing this is that the cohesion of that file would be low. Some classes, ex. the CLI needs not to know about the ICmdProvider interface, but they do need to know about the command structs. Thereby it was decided to keep the files separated and put them both in the interface.

### 6.4.4 Getting the commands into LUA

Using LUA the user input is forwarded to LUA. LUA needs then to have knowledge of the registered commands. The commands will be in LUA as LUA functions, and the important information belonging to a function will be stored in a table. The structure of the function-information table is shown in the code below:

```lua
function funcType (name, help, use, fnPointer, ICSUpointer )
    return { name=name, help=help, use=use, fnPointer=fnPointer, ICSUpointer=ICSUpointer }
end
functions = {}
```

This makes it possible to insert things to the functions table as functions[name] = funcType(name, help, use, fnPointer, ICSUpointer), and access the individual fields just as accessing in a struct: functions['foo'].name.

So when adding a command first the function table is filled in using the LUA-script shown below which fetches the command information from a stored C-variable.

```lua
name, help, use, fnPointer, ICSUpointer=luaFn_getCmd ()
if ( functions [ name ]== nil ) then
    functions [ name ]=funcType ( name, help, use, fnPointer, ICSUpointer )
else error ( 'function ' .. name .. ' already exists' )
end
```

After the information is filled in, the LUA-function must be made. It is made from a string which has been put together in C. Every registered command to LUA will look similar, they all return the answer from calling the same C-function. The LUA function code is shown bellow together with the creation of the string.

```lua
//LUA function
function name(...) 
    local arg={...}
```
```lua
return c_commandCall(functions[name].fnPointer, functions[name].ICSUpointer, functions[name].name, table.unpack(arg))
end

//way to create the function from C
char* part1 = "function ";
char* part3 = "(...) local arg={...} return c_commandCall(functions[" "] . fnPointer , functions[" ";
char* part5 = ""] . ICSUpointer , functions[" ";
char* part7 = ""] . name , table.unpack(arg)) end ";
int bufSize = 4*strlen(cmd->name)+strlen(part1)+strlen(part3)+
strlen(part5)+strlen(part7)+strlen(part9)+1;
char luaFunctionRegister[FN_REG_SIZE];
if (FN_REG_SIZE<bufSize) {
    CLI_output("Error: Sent too long string to register to lua \n");
    return;
}
sprintf(luaFunctionRegister,
    "%s%s%s%s%s%s%s%s ",
    part1, cmd->name, part3,cmd->name, part5 , cmd->name, part7, cmd->name
    , part9);
luaL_dostring(L, luaFunctionRegister);
```
Chapter 7

Testing

Autoliv makes products that handle human safety. A product that for example can access the breaks of the car might impact human life. Thereby the requirements of testing is highly implemented in the process of developing software. The systems are not security critical but safety critical. The means of security still exist in the context of the ITE-system, the security of the ITE-system is discussed in the security chapter, chapter 5.

Developing the ITE-system unit-testing[1][29] is done and when doing code check-in to the code-revision control system[1][30] the commit is only accepted if 100% of the code is covered by the unit-tests. Also the code will be reviewed by at least one other employee to guarantee good standards of the code.

Also MISRA C rules are followed to promote safety. MISRA (Motor Industry Software Reliability Association) is, as described on their homepage [31] ”MISRA was originally established as a collaboration between vehicle manufacturers, component suppliers and engineering consultancies, and seeks to promote best practice in developing safety-related electronic systems in road vehicles and other embedded systems. To this end MISRA publishes documents that provide accessible information for engineers and management, and holds events to permit the exchange of experiences between practitioners.” MISRA C is a set of safety promoting rules that are used for the C language. MISRA-C:2004 document contains 142 rules.

7.1 Testing of code

When using the predefined system code structure of CSI’s and CSU’s there exists code generators for making mocks and some test-code. This helps the developer to perform earlier testing. The mocks will reflect the interfaces of the CSI’s dependent on and thereby make the dependency to other modules to be simulated when testing. The tests which shall be done for a component is Factory Unit Tests, Unit Tests and Integration Tests. The tests are performed on host.

It is also possible to test code on target and on a simulator of the kernel which are running the real-time operating system. Tests should be done on both these parts.
7.1.1 Testing of Remote Procedure Calls

The factory tests for RPC can be used just as they are generated. The unit tests are almost not applicable, some small functions can be tested. The main test for RPC is integration tests. The integration testing of the remote procedure calls is a bit harder than the normal integration testing. To test code that depends on semaphores and different thread calls also the testing program has to be done threaded.

For multi-threading boost was used [15]. The tests as well as the system is normally done in C, however for using boost C++ is needed. Thereby the test functions using boost have to be extracted to another file and called from the C test file. When calling C++ from C be aware of the name wrangling made in C++. To make the program compile use "extern "C"" for the code in the header file for the C++ function. This will make the linker find the C++ functions.

The test sequence can be seen in figure 7.1. The entry function calls a C++ thread organizer that spawns a thread to do the actual user call after some time. Then it calls the function to trigger a send when user input arrives. After the message has arrived the user-call function will be waiting on a semaphore, the send of the message is done and when answer arrives a post is made to the semaphore. This makes the user-call function to return and the test is complete.
7.2 LUA on simulated system

After consideration had been done to use LUA, it had to be tested if it was possible to run it on the kernel to be used. Thereby the simulator was used to try to get LUA to work. First tests failed in linking. After some investigation it came down
to that no file system was provided by the OS at the time of compilation. After adding this, only one linking error remained from the LUA source. This was a call to the stdlib function system(cmd) [17]. This was used only once from lua, in the LUA standard operating system library loslib.c [14]. This was removed and after that LUA was compiled to run on simulated target with the following gluing code between C and lua:

```c
#include <INTEGRITY.h>
#include <stdlib.h>
#include <stdio.h>

#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

lua_State *L;

void luaConsole () {
    int s;
    char input [100];
    int maxInLen=100;
    while (1) {
        printf ("> " );
        fgets (input, maxInLen-1, stdin);
        s = luaL_dostring (L, input);
        if (s){/*error, print message and pop*/
            printf ("%s\n", lua_tostring(L, -1)); /*print error*/
            lua_pop(L, 1); /*remove from stack*/
        }
    }
}

int main () {
    int s;
    lua_State *Lu = luaL_newstate();
    L=Lu;
    luaopen_io(L);
    luaL_openlibs(L);
    s=luaL_dofile(L, "myScript.lua");
    if(s){/*error, print message and pop*/
        printf ("%s\n", lua_tostring(L, -1)); /*print error*/
        lua_pop(L, 1); /*remove from stack*/
    }
    luaConsole();
    Exit (0);
}
```
Chapter 8

Discussion

This is the final chapter where I discuss what has been done, what could have been done differently and what should be done in the future.

8.1 Conclusions

The project was done mostly according to what was planned and I believe the created software will be highly valuable and helpful in the development of the ECU functionality.

8.1.1 Design

Pretty early in the process it was decided to split the ITE-system into two parts. After the implementation has been done, I would say that this was definitely a good choice. The system is now not restricted by hardware requirements and the implementation follows more design patterns taken from the object-oriented world. The software and hardware systems have almost nothing in common. Also the hardware system needs not to conform to any of the software requirements which would only constraint the solutions for that project.

The decision to adopt the system to both LUA and TinyShell was also a good decision made. This had one more benefit which was not considered before, now, if the code is reused in another project with less memory constraints, another scripting language could be included.

The discovery that the ITE-system needs to handle many instances of the same function was made pretty late in the project. This made the plan change a bit, it was planned to implement event handling and event listening to the system but when the fact about multi instance functions appeared it was considered to be of much higher importance, so event handling had to be deferred to later work in the project. I don’t know why this was not discovered earlier. If an investigation of how the users intended to use the system would have been done, this discovery might have been found earlier.
8.1.2 Process

In the process of the project I was intended to spend one week of getting to know the context, then four weeks of requirements elicitation, four weeks of design, four weeks of implementation and four weeks of integration and verification. In the beginning this process seemed to be easy to fulfill, but after some weeks of requirements work I could not help starting with some design. I would say that a totally strict model of developing is not user convenient. If instead of following a model like this, one would have worked agile in the way of iterations, still this strict way comes in all of the iterations. In my opinion the best way of developing software, is to as soon as possible start with the work that can be done in whatever phase it is. That means when some requirements are settled, start to do some design, keep working on the requirements and redo the design as the requirements evolve. Same goes with design and implementation, its when implementation is started or half way through, most of the design problems rise. To figure them all out in advance is hard and requires a lot of experience. My approach might need some rework but I believe that it will eventually give better software.

8.1.3 Planning

Unit tests should definitely have been made before the code itself. That would have helped in the development of the functions, but most of all I believe the tests would have been better. What I did now was to first do all the code and then start with the testing. This made the tests more adapted to the code existing instead of coming from the requirements.

Also when planning the project, I estimated time to design and time to code pretty well. However, I constantly underestimated the time to integrate the code with the code already in the system. This is a non-trivial and time consuming task and enforces many tools to be used. It took almost one week of work complete my first commit to the version control and analyzer system.

8.2 Future work

This is just a first release of a bigger product. The implementation done can probably be sharpened a bit, but most important the functionality which are to be provided are still not implemented. Now an interface for providing and calling commands have been made so the developers need to create functions to put into the CLI. Lots of what shall be done in the future can be found in the requirements document. Many features are highly prioritized and still not started. This project was a first release and it was decided already before the project started that it should and could not be totally completed in the limited time of a master thesis project.

What I would say can be made better with the CLI parts, is to make it easier for a developer to add commands, now a lot have to be added to the CSU which are to provide commands. Also there are no means of synchronization between
job’s action functions and the CLI functions. The CLI functions are purely asynchronous and forces the command provider to be sure that no race conditions appear and that no data is accessed in a way without locks.

The work closest in time is to get the developers of Autoliv to know about the ITE-system so that they can start fill the CLI with commands to use.
Bibliography

[1] Shari Lawrence Pfleeger, Joanne M. Atlee (2009), Software Engineering, Pearson Education


[12] Thorbjörn Jemander, Senior Software Architect at Autoliv Electronics, oral source


[19] Llorenç Cerdà Alabern (2009), Xarxes de Computadors - Conceptes bàsics, Edicions UPC


Appendix A

Requirements Document

This appendix shows the complete Requirements document for the ITE project.
ITE Requirements

Background
The system has three processors and each processor have two address-spaces, thereby being two virtual processors. Requirements are given for a ITE system which will be a system on target connected to from a host. The systems interface will be a command line interpreter (CLI).

High prioritized functional requirements
COMMON1: The test system shall work even without inter-processor communication working. It shall then be possible to see and execute commands on the processor connected to.
That is, even if the MCU doesn’t work or communication drivers are not yet implemented, it shall be possible to connect to any processor to execute tests on it.

COMMON2: It shall be possible to write scripts to run at the CLI.
This is to be able to run test sequences.

COMMON3: It shall be possible to connect to the CLI through an UART.

HW1: It shall, through the program, be possible to read and write to pins on the processors.
For example to read from the A/D converter or lighten a LED.

HW2: It shall be possible to read and write to the memory.
This will be tested early since memory errors are hard to solve late in the process. Since this requires some drivers, they will be essential to implement early in the system development.

HW3: There shall be delay functions implemented, the accuracy of the delays shall be 1ms+- 0.1ms.

HW4: It shall be possible to perform memory tests, either through scripts or through a function.

HW9: It shall be possible to read a frame from video-source and to publish a frame on the video-interfaces available at the system.

SW1: The CLI shall contain functions for file-handling, like ls, rm, etc.

SW2: The CLI shall be able to list all functions which it can currently invoke.

SW3: The CLI shall be able to list all other processors to which it can communicate.

SW4: It shall be possible to, independent from which processor you connect to, if network is established, see and execute all able commands for all processors, in a uniform way.
This is to be able to run the same script file independent from what processor you are connected to.

SW5: It shall be possible to get system statistics printed through the CLI, for example latency, processor workload, number of frames since start, etc.

SW6: It shall be possible to create a message from a job, and to execute a job’s action function.

SW7: It shall be possible to handle an application-level graph, for example show the graph or its connections.
SW8: The CLI shall be able to store and search from 200 commands. *That is the total amount of commands in the system shall be able to reach 200.*

SW9+MAINT7: It shall be possible to get and set states (internal variables) in a CSU or job.

SW10: It shall be possible to handle the system state manager for example to see the latest events.

SW11: The CLI shall be able to execute functions in both the application layer and the framework layer.

SW12: The CLI shall be a modular entity, possible to extend with modules and itself only contain methods for storing and executing commands, not the command-implementations.

SW13: It shall be possible to, when listening to something, decorate the output with timestamp and wave number.

SW14: A developer shall be able to dynamically add functions to the CLI. *By adding a registration of the command in the system startup sequence.*

SW15: The processes which are not safety classed may never interfere the safety classed (ASIL) ones, they can’t share memory.

SW16: The ITE hardware functionality shall be implemented as not safety classed.

FD1+TST1+MAINT4: The CLI shall be able to start the event-logging functionality and make it decorated with global timestamps. *It shall do extensive logging when it is on, the log can be filtered afterwards on the PC. For test: It must be able to do it for long time, ex to save a log while doing a drive recording or long time heat-test.*

FD1.1: The logging functionality must be able to log internal states, start/stop time of the jobs and intermediate results.

FD1.2: The CLI shall be able to start transfer of the log to PC.

TST2: The CLI must be able to simulate and inject errors, to be able to ease DTC-testing. *Ex.: The camera wire has broken or the DSP have crashed.*

COMMON4: It shall be possible to connect to the MCU-CLI through CAN. *This is the only access point for test department. Not that important for the others.*

MAINT1: It shall be possible to synchronize, for logging purpose, the time in the ECU with the time on the computer connected to the CLI. *This will be used in temperature tests to know the environment temperature.*

MAINT2: It shall be possible to get the data which triggered an algorithm to execute. *To be able to debug when no error is in the log, after a “wrong action”. Ex the car seems to always brake when entering a tunnel.*

MAINT3: It shall be possible to trace a sequence of messages leading to a decision. *To be able to debug when no error is in the log, after a “wrong action”.*
SW18+MAINT5: All commands shall not be able for everyone to use, there shall be a login and access rights for commands. *This prevents the customer from bloating the device.*

MAINT6+TST3: It shall be possible to test parts of algorithm chains by inserting input to entities in the chain and trigger their execution.

**Other functional requirements**

FD2: The CLI shall be able to take input from a file. This is for making the CLI to be able to generate messages, in a stream, from a file *for example to fake camera input, or to start an algorithm chain from a specific point.*

HW5: It shall be possible to flash memory from a file. For example to get boot-loaders to the memory.

HW6: It shall be possible to access all the drivers which are implemented.

HW7: It shall be possible to create bus traffic; this is to make measurements on for example magnetic fields possible.

HW8: It shall be possible to do CPU load tests. *Have a program that does intensive calculations for a period of time.*

SW17: There shall be a debug-mode when the commands are in the CLI, and when system is not started in debug mode, the commands shall not be in the CLI. *The reason is that at production we do not want all the commands in the CLI.*

SW19: It shall be possible to be connected to two CLI’s in different processors at the same time if one is dedicated to output. *This is to be able to listen from one CLI and invoke functions from the other.*

SW20: The CLI shall be able to react to events that happen. *For example trigger something when frame 23 starts.*

SW21: The CLI shall be able to distinguish between immediate schedulable commands, and deferred commands to do when certain conditions become true.

SW22: The CLI shall be able to stack deferred commands and execute them at their desired point of execution.

SW23+MAINT10: The CLI shall be able to start a transfer via Ethernet.

MAINT8: It shall be possible to, when a reset has been forced because of a failure see the function and/or line where the error arose. *This is to be able to locate the root cause of errors more easily. This is a requirement on how to write to the system event log.*

MAINT 9: It shall be a command log in the system, to be able to see if some malicious commands have been executed.
Comments
The prioritized requirements from the hardware department (marked HW or COMMON 1, 2 and 3) are both the important functions to implement as well as the functions they need really early in the process. The main advantage from the ITE-system for the hardware department is early testing.
### Use cases

#### Hardware

**Name:** Test memory  
**Actor:** Hardware developer  
**Flow of events:** The hardware developer connects to the system via UART, he then uses a sequence of read and write to memory commands to do a memory test.
**Name:** Add a function and use it  
**Actor:** Software developer  
**Flow of events:** The developer of a job writes code to trigger his action function, he then registers this function. After the register he lists the available functions on the system and sees he’s function there. He invokes it.
Name: DTC-testing
Actor: System tester
Flow of events: The system tester logs in to the system through the CAN interface, injects an error and observes the system reaction to the error.
Name: System logging
Actor: Feature developer
Flow of events: The feature developer starts the logging of events. The system runs for some time, and then the feature developer downloads the log to investigate it.
Modularization and distribution of requirements

The ITE system will be divided to the following modules:

Core CLI-functionality modules
These are the modules on which the other modules shall be plugged in. They will provide the core CLI functionality and the distributed way of access. This will be the modules to focus on in start of development.

- Communications module
- CLI
- Broker (manager of calling functions on another processor)

Functional modules
These are the modules which will be plugged into the CLI system and thereby constructing the ITE.

- Hardware functionality (can be implemented without parts of the core functionality)
- Host program
- File-handling
- System statistics
- System state
- Job and message handler
- Logging module
- Event handling (Own requirement entity but tightly coupled to other parts of code)

Communications module
The communication module is responsible to take input and redirect it to the CLI in a uniform way independent of the incoming connection. It is also responsible to provide the functionality to give output to the correct place.

Requirements: COMMON3, COMMON4, SW19, and eventually functions to have a checksum on the communication.

CLI
Responsible to find and call the given function name if it exists and to otherwise forward the call to the broker – if we have connection to other processors.

Requirements: COMMON1, SW2, S28, SW11, SW12, SW13, SW14, FD1.2, SW18+MAINT5, MAINT9
And by forwarding to the broker: SW3, SW4
This is only requirements from software, the requirements from hardware is:
To be able to dynamically add commands and to be able to find and execute them.

Broker
The broker is responsible to know which processors we are connected to, which functions able to call and how to call the functions. This is the core of the distribution. The broker is not needed by hardware requirements.

Requirements: SW3, SW4, SW8
**Hardware functionality**
The functionality required for the hardware tests. This is a less design, more functionality entity. This is needed to be implemented early.

Requirements: HW1-HW9

**Host program**
The requirements force a host program, but when connecting through UART there shouldn’t be a need to use the host program. It will be needed for having a CAN connection and to transfer files between the host and the CLI.

Requirements: COMMON2, FD2, FD1.2, HW5, COMMON4, MAINT1

**File-handling**
The system needs file-handling, the CLI itself shall not implement this, thereby the need to make it an own requirement entity. This also includes the requirements to be able to with a command start a transfer of a file via another interface, ex Ethernet.

Requirements: SW1, FD1.2, SW23+MAINT10

**System statistics**
A module to provide system statistics such as latency, processor workload, number of frames since start, etc.

Requirements: SW5, MAINT8

**System state**
A module to see the system state, for example latest events that happened.

Requirements: SW10

**Job and message-handler**
This module shall be able to handle the graph and message structure, for example to show connections, insert messages, start action-functions etc.

Requirements: SW6, SW7, FD2, TST3+MAINT6

**Logging module**
Module to log system events, job and CSU states etc. Connected with system state module. And to event handling.

Requirements: FD1.1, TST1, MAINT8, MAINT9, MAINT2, MAINT3,

**Event handling**
It shall be possible to listen to events in the system, therefore there must be means to create events and to listen to them.

Requirements: SW20-SW22, SW13

**Requirements not assigned**
The requirements not assigned to any module are SW9+MAINT7
Appendix B

Larger scale diagrams

The following pages contain larger versions of the diagrams.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Daniel Andersson