



---

# Report

Embedded debugger

---

Place & Date	Enschede, January 24, 2019
Version	1.1
Document name	Report
Created by:	Marijn Feijten, 408818



## Change log

Version	Date	Author	Description of change(s)
V0.1	14-05-2018	Marijn Feijten	Initial version
V0.2	03-06-2018	Marijn Feijten	Changes according to the feedback from T. Mentink & C. Slot
V0.3	18-06-2018	Marijn Feijten	Changes according to the feedback from T. Mentink & C. Slot and added chapters 5, 6 and 7
V1.0	18-06-2018	Marijn Feijten	Changes according to the feedback from T. Mentink, C. Slot & F. Vlaardingerbroek
V1.1	24-01-2019	Marijn Feijten	Removed appendices

---

## Foreword

This document is the final report for the Embedded Debugger graduation project, executed at DEMCON. This report is meant to give insight in how the project was formed and executed. The project plan and conceptual design document are included as appendices.

This report was written on behalf of DEMCON and Saxion. If anything is unclear after reading this report, the intern can be contacted. For contact information, see the front page.

This document is written using  $\text{\LaTeX}$ (la-tech), an open-source mark-up language<sup>1</sup>. This template has been created by Foitn and is free for anyone to use (Foitn, 2016).

The results of this graduation internship have been greatly influenced by employees of DEMCON. I would like to thank my colleagues for helping me during my internship and sharing their knowledge and experience.

In particular I want to thank my graduation coaches from DEMCON:

- Timon Mentink
- Bonne Zwaga (for the first week of the internship)

From Saxion I want to thank my graduation coach:

- Christiaan Slot

Marijn Feijten

Enschede, January 24, 2019

---

<sup>1</sup>For more information, see <https://en.wikipedia.org/wiki/LaTeX>, for an easy 'Get started' guide, see <http://www.maths.tcd.ie/~dwilkins/LaTeXPrimer/> and for the used  $\text{\LaTeX}$  distribution, see <https://www.tug.org/protext/>.

---

## Summary

This report provides an overview of the carried out graduation project, the "Embedded Debugger". The Embedded Debugger is a C#/.NET tool which is used to debug embedded systems at a high level. The tool has been developed at DEMCON and has been actively used in many projects. This tool allows engineers to write to registers, read from, plot and log values from registers. This has been worked on from many sides, whereby some parts have not been implemented correctly. The goal of this project is to streamline the Embedded Debugger and to add some new features.

The methodology used in this project is feature based, which is a combination between the V-model and Scrum. Each feature is seen as a subproject, which is approached from a V-model perspective. This means defining requirements, conducting applied research, implementing the feature and testing it against these requirements. All features together can be seen as the backlog of Scrum.

The result of the project is a completely renewed application written in WPF, inspired by the old Embedded Debugger. In this application all of the old functionality has been implemented. The new application contains almost all of the features which were in the backlog, with some exceptions to a couple of could have features. An example of a new feature is an RPC (Remote Procedure Call) library, which allows external scripts to manipulate the Embedded Debugger for rapid testing and prototyping.

These features that have not been implemented have been placed in the recommendations, thereby ensuring that future work can be done. An example of these features is a new version of the proprietary DebugProtocol, which allows for a more versatile tool. This new version has been designed, but is yet to be implemented.

---

---

# Contents

<b>Foreword</b>	<b>ii</b>
<b>Summary</b>	<b>iii</b>
<b>Contents</b>	<b>iv</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vi</b>
<b>Abbreviations</b>	<b>vii</b>
<b>Glossary</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The company . . . . .	1
1.2 Background of the assignment . . . . .	1
1.3 Problem definition and objectives . . . . .	2
1.4 Scope of work . . . . .	2
1.5 Approach and methodology . . . . .	3
1.6 Outline of the report . . . . .	4
<b>2 The Embedded Debugger</b>	<b>5</b>
2.1 Background . . . . .	5
2.2 DebugProtocol . . . . .	6
2.3 Overview . . . . .	9
2.4 Project goals . . . . .	14
2.5 Issues . . . . .	15
2.6 Conclusion . . . . .	16
<b>3 The new Embedded Debugger</b>	<b>17</b>
3.1 Embedded Emulator . . . . .	17
3.2 Design phase . . . . .	18
3.3 Implementation phase . . . . .	21
3.4 Test phase . . . . .	25
<b>4 Additional functionality</b>	<b>29</b>
4.1 Embedded configuration over DebugProtocol . . . . .	29
4.2 Embedded configuration from compiler output . . . . .	31
4.3 Simulink parsing . . . . .	32
4.4 RPC . . . . .	33
<b>5 Results</b>	<b>36</b>
5.1 GUI . . . . .	36
5.2 Embedded Emulator . . . . .	44

---

<b>6</b>	<b>Conclusions</b>	<b>45</b>
6.1	Debug the current debugger, or make a new design . . . . .	45
6.2	Implement a serial port interface . . . . .	45
6.3	Create the Embedded Emulator . . . . .	45
6.4	Send embedded configuration from the embedded platform to the Embedded Debugger . . . . .	46
6.5	Create an RPC interface . . . . .	46
6.6	Generate embedded configuration from compiler output . . . . .	46
6.7	Add a trace window to the GUI, which acts as a logging mechanism, containing features like logging levels . . . . .	46
6.8	Update the debug protocol, to allow a maximum of 256 debug channels .	47
6.9	Create an interface to other languages . . . . .	47
<b>7</b>	<b>Recommendations</b>	<b>48</b>
7.1	Update the debug protocol . . . . .	48
7.2	Simulink . . . . .	48
7.3	Trace window . . . . .	48
7.4	Embedded target . . . . .	49
	<b>References</b>	<b>50</b>

---

## List of Figures

2.1	Overview of communication . . . . .	6
2.2	Protocol Message Layout . . . . .	6
2.3	Protocol Message Escaping . . . . .	8
2.4	Global class diagram old Embedded Debugger . . . . .	10
2.5	Connect tab . . . . .	11
2.6	Registers tab . . . . .	12
2.7	Terminal tab . . . . .	12
2.8	Log tab . . . . .	13
3.1	The Embedded Emulator . . . . .	18
3.2	Global class diagram Embedded Debugger . . . . .	19
3.3	DataTreeView example (WinForms) . . . . .	22
3.4	TreeDataGrid example (WPF, custom control) . . . . .	22
3.5	Decoding a protocol message . . . . .	25
3.6	Encoding a protocol message . . . . .	26
3.7	ConnectorChooserUserControl - result of listing 3.1 . . . . .	28
4.1	The parsing user control . . . . .	31
4.2	Overview of RPC connection . . . . .	33
4.3	RPC interface & RPC resolver . . . . .	34
5.1	Main window Embedded Debugger . . . . .	37
5.2	The Embedded Debugger: Connections . . . . .	37
5.3	The Embedded Debugger: Registers . . . . .	39
5.4	The Embedded Debugger: Terminal . . . . .	39
5.5	The Embedded Debugger: Logging . . . . .	40
5.6	Log file name helper window . . . . .	41
5.7	The Embedded Debugger: Parsing . . . . .	42
5.8	The Embedded Debugger: Parsing . . . . .	43
5.9	The Embedded Emulator . . . . .	44

## List of Tables

2.1	Part of command list (cmd-byte) . . . . .	7
2.2	Escape characters . . . . .	8
4.1	Embedded Configuration overview . . . . .	30
4.2	Embedded Configuration commands overview . . . . .	30



---

## Abbreviations

Abbreviation	Description
ACK	Acknowledge message
CAN	Controller Area Network
CDD	Conceptual Design Document
CPU	Central Processing Unit
dll	dynamic-link library
DSP	Digital Signal Processor
e.g.	exempli gratia (for the sake of example)
ETX	end byte
GUI	Graphical User Interface
i.e.	id est (that is to say)
I/O	Input / Output
LED	Light Emitting Diode
LDR	Light Dependant Resistor
OO	Object Oriented
OS	operating system
PC	personal computer
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
STX	start byte
UDP	User Datagram Protocol
WinForms	Windows Forms
WPF	Windows Presentation Foundation

---

---

## Glossary

Term	Description
microcontroller	A control device which incorporates a microprocessor
optomechatronic	Fusion of optical and mechatronic engineering
refactor	To rewrite existing source code in order to improve its readability, reusability or structure without affecting its meaning or behavior
svn	Subversion, Apache Subversion is a software versioning and revision control system distributed as open source under the Apache License
.NET	A framework by MS-Windows, comes with a large set of (graphical and non-graphical) sw-libraries, makes use of a virtual machine

---

# Chapter 1: Introduction

This report is the final document of the graduation period. The graduation is the last module in the studies Applied Computer Science at Saxion University of Applied Sciences. The main goal of the graduation period is for the student to prove that he/she can work on a professional level in a company. This means that all of the competences set by Saxion have to be met and these will be graded. These competences are not described here, since this is part of the learning report and not the final report.

## 1.1: The company

DEMCON is a high-end technology supplier of products and systems, with as focus areas high-tech, medical, embedded, optomechatronic and industrial systems & vision. DEMCON is a fast-growing business that supports clients with a wide range of competences. As a system supplier, DEMCON can meet the entire needs of its clients, from proof of principle, prototype and pre-production to serial production. In more than 25 years, the business has grown to become the DEMCON Group (with more than 440 employees in 2018). The head office is situated in Enschede, with subsidiary locations in Son, Oldenzaal, Groningen, Delft and Münster (Germany). DEMCON has clients worldwide, from the Netherlands and Germany to Asia and the US.

## 1.2: Background of the assignment

DEMCON designs embedded systems based on a wide range of target systems. This includes microcontrollers, application processors and digital signal processors (DSP). The embedded systems are used in mechatronic systems, thus typical functions are: motor control, signal processing, user interfaces etc. For these embedded projects, a general, reusable tool has been developed for high-level debugging: the "Embedded Debugger". This tool is used for standard reading/writing of variables, has a command-line interface, but also includes graphical plotting and logging. The embedded debugger consists of two parts, a PC-application which is written in C#/.NET and a protocol for communicating with an embedded system.

---

### 1.3: Problem definition and objectives

The tool is actively used in multiple projects. Because of this, the embedded debugger has been branched a couple of times. In each of these branches different features have been implemented. One of the goals of this project is to make these branches obsolete and have all of the already implemented features in the main branch. This ensures quality and removes redundant code, code that might work in a different way in different branches. The project members have suggested several improvements, which will be called features from now on. After merging the different branches, objectives are to implement these features.

### 1.4: Scope of work

The scope of this project is to refactor or rebuild the Embedded Debugger. Afterwards new features can be implemented. The target side (embedded platform) is not part of this project and will not be edited. The new Embedded Debugger should allow for fast improvement and for new features to be implemented within a short amount of time. The to be implemented features are described in the conceptual design document (CDD), see ???. Any features that are not described in the CDD, will not be implemented.

---

## 1.5: Approach and methodology

The first step in this project is for the intern to get acquainted with the current software. The software was written in 2012 and a lot of features have been implemented in a short period of time without proper testing. For this reason a decision has to be made whether to continue with the current software or to start over, keeping the old software in mind.

To determine which feature was to be implemented first, a CDD is made, in which the features are described. Afterwards these feature got a score of 0-10 on how important they were and whether or not it is a core feature.<sup>1</sup> The CDD contains a planning for the entire graduation period and contains as many features as possible. Some features require applied research to determine the best way to implement this feature. After the research the feature is implemented after which it is tested.

The used methodology can be seen as a subset of the V-model with a combination of scrum. This is because every new feature is approached as a mini V of the V-model. This means that for each feature the following procedure is performed:

1. Determine requirements
2. Conduct applied research, to determine the best way to implement the feature
3. Implement the feature
4. Test the feature against the requirements

The other part of the methodology is the scrum part. Because of the feature based methodology each feature is a small project on its own. This usually happens in scrum as well. In scrum the backlog is used to determine which features are left to be implemented. All of the features that have been defined in the CDD can be seen as this backlog.

For each subproject a certain amount of time is estimated. This can vary from half a day plus half a day of testing to three weeks plus half a week of testing. Each new feature is tested against the previously defined testing material, which can be hardware/software tests or simply letting a colleague test for user-friendliness.

For the normal V-model a document is written for each step of the way, which means specifications, functional design and technical design. Since these is a mini v per feature, these documents are not written, even though some research is conducted. A pro about this is that writing these kinds of documents takes time, which is saved and can be used to implement more features. A con about this approach is that if the project or part of the project is ever repeated, this research has to be conducted twice. For this project, the decision was made to skip the writing of these documents, since more features implemented was of higher importance.

---

<sup>1</sup>Without a core feature the entire Embedded Debugger doesn't function properly

---

## 1.6: Outline of the report

This report describes the entire graduation period of the project Embedded Debugger at DEMCON. For this reason the chapters are written in chronological order.

Chapter 2 describes the old Embedded Debugger, explaining how it was implemented and what its problems are. Chapter 2 concludes that the old Embedded Debugger cannot be used properly anymore and that it is more time efficient to start over.

The Embedded Emulator, which was created first, is described in section 3.1. Chapter 3 describes how the new Embedded Debugger came to be. Section 3.2 explains how the new design for the Embedded Debugger was created.

After the new design was made, it was to be implemented, which is described in section 3.3. After the implementation phase the application had to be tested. This ensures that the new Embedded Debugger performs to a high standard. The testing is described in section 3.4.

Once the new Embedded Debugger was up and running, additional functionality was implemented, this is described in chapter 4. Each new feature is described in terms of design, implementation and testing, where possible.

Chapter 5 describes the resulting products from this graduation period. It creates a complete picture of what happened and which goals were achieved.

In section 2.4 the goals for this project are stated. From the goals and the results from chapter 5 conclusions can be drawn. These conclusions are described in chapter 6.

Chapter 7 recommendations, is the final chapter of this report. It is meant as a sort of advise for the next person that is going to be working on the Embedded Debugger. It gives insight in which features can still be implemented and for some of these features it gives a recommended way to do this.

---

---

## Chapter 2: The Embedded Debugger

This chapter describes the Embedded Debugger, starting with some background information in section 2.1. Afterwards the DebugProtocol is explained, giving insight in how the debugger is implemented. This insight is enlarged in section 2.3, which describes the class diagram and some functionality. The Embedded Debugger had some issues, these are described in section 2.5. Based on these issues, a conclusion was drawn, which is described in section 2.6. The final section of the chapter, see section 2.4, describes the project goals, explaining what the main goals of the internship were.

### 2.1: Background

The Embedded Debugger is a C#/.NET application, which can be used to debug embedded platforms. This tool is generic and can be used on any embedded platform. On the embedded platform the embedded side of the Embedded Debugger has to be implemented, ensuring calls are handled correctly. Using the embedded side, the Embedded Debugger can manipulate the embedded platform. This consists of reading a register value, writing to one, sending debugstrings (just like a printf call) and more. This allows for fast debugging without any additional tooling required. Do note that this is not a hardware wise debugger, therefore stepping through code, settings breakpoints, etc. is not possible.

The biggest advantages of the Embedded Debugger compared to a hardware debugger are:

- No additional hardware is required
- A hardware debugger can not be used on every embedded platform, which can be an entire system with multiple microcontrollers/FPGAs/DSPs
- No restrictions to what the capabilities in IDE of the manufacturer of that particular microcontroller are
- Any additional feature can be added at any given time
- The Embedded Debugger has a plotting environment, where signals can be analysed
- Logging is implemented, which means analysis can be done after the debugging sessions

The Embedded Debugger was originally written in 2012. This version was targeted at efficiency and smaller microcontrollers. The communication protocol between the C#/.NET application and the microcontroller has been developed at DEMCON and will henceforth be called "DebugProtocol". To enable the microcontroller to correctly handle calls from the DebugProtocol, a small piece of software has to be included in the firmware, which is called "Embedded Debugger Target Side (EDTS)". A graphical overview of the communication between the C#/.NET application and an embedded platform can be found in figure 2.1.

---

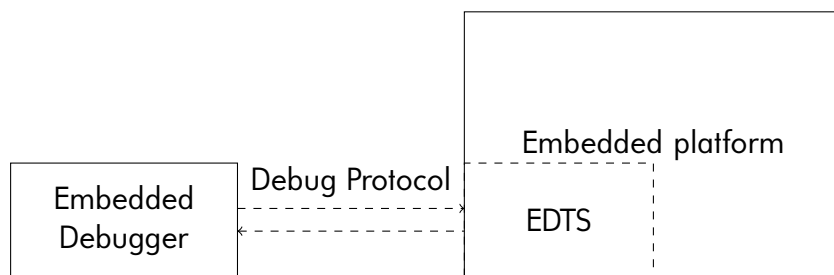


Figure 2.1: Overview of communication

## 2.2: DebugProtocol

The DebugProtocol is protocol used by the Embedded Debugger. This protocol has been set up at DEMCON and is developed in a minimalistic way. Therefore it has limited overhead while being able to ensure that all data has been send. This section gives some insight in how the DebugProtocol was set up and how it should be used. The complete explanation and details of the DebugProtocol can be found in ??.

### Message

Each message send over the DebugProtocol has the following layout, of which a graphical layout can be found in figure 2.2. The message starts with a start byte (STX) and ends with an end byte (ETX). The second byte of the message is the ID of the embedded platform the message is send to or from, this allows multiple nodes to be connected simultaneously.

The third byte is the message ID, this message ID id used to identify if this is a new or an older message. Whenever the PC sends a message where the ID is set (not 0x00), the embedded platform has to send an acknowledge (ACK) message. This message ID is omitted whenever the embedded platform sends a message to the PC, this is because the PC never has to send an ACK. If the ACK message is not send within a certain amount of time, the original message is resend to the embedded platform.

Each message contains a cmd byte, which lets the receiving side know how to handle the message. Examples of this cmd byte can be found in table 2.1

STX	μC	msg-ID	cmd	cmd-data	CRC	ETX
1 byte	1 byte	1 byte	1 byte	... bytes	1 byte	1 byte

Figure 2.2: Protocol Message Layout



---

cmd	Description
'V' = 0x56	Version, retrieve the version and name of the embedded platform
'I' = 0x49	Info, retrieve sizes of types of the embedded platform (think of the size of an int on x86 or x64 platforms)
'W' = 0x57	WriteRegister, write a certain value to a register on the embedded platform
'Q' = 0x51	QueryRegister, read a value from a certain register on the embedded platform
'S' = 0x53	DebugString, send debugstrings, just like a printf statement, only it will print on the Embedded Debugger

---

Table 2.1: Part of command list (cmd-byte)

The cmd-data consists of an N amount of bytes. This N is determined by which cmd byte is received. These bytes contain the actual data, which is send from or to an embedded platform.

Last in the message is the cyclic redundancy check (CRC), which is used to check if the message has been send completely, without any loss of data. This CRC changes drastically if a single bit changes in the message. Therefore this is a small and quite secure way to validate the messages integrity.

## Debug channels

Each embedded platform has 16 so called 'Debug channels'. These Debug channels are set up by the Embedded Debugger and are responsible for sending register data periodically. Each debug channel (when enabled) is set to a certain register and sends this data, which can e.g. be sensor data at a certain frequency. The possible frequencies are:

- High, every time the value is updated
  - Low, every second
  - Upon, which means that it is send upon request of the Embedded Debugger
-

## Escape characters

As described in section 2.2, the DebugProtocol starts with an STX byte and ends with an ETX byte. For STX the value of 0x55 is used and for ETX the value of 0xAA is used. These values might be used inside of a message, example is that the value of a register is 0xAA. Because of this, escaping these characters is required. Escaping means that an escape character is placed in front of the byte that has to be escaped. Moreover, the to be escaped byte changes value. By performing this action, there can never be an STX or ETX byte between the start and stop bytes. The escape character for this was chosen to be 0x66. The STX, ETX and escape-char are chosen to have a minimum of equal consecutive bits. This minimizes bit-stuffing in cases where the transport layer makes use of this (like CAN-bus or wireless communication).

To escape the character, the escape character (ESC) is place in front and the value changes. This changed value is the XOR value of  $\text{ESC} \oplus 0xXX$ , an overview of this can be found in table 2.2. To decode a message, the XOR can be taken again, thereby resulting in the original value again. A graphical overview of message escaping can be found in section 2.2.

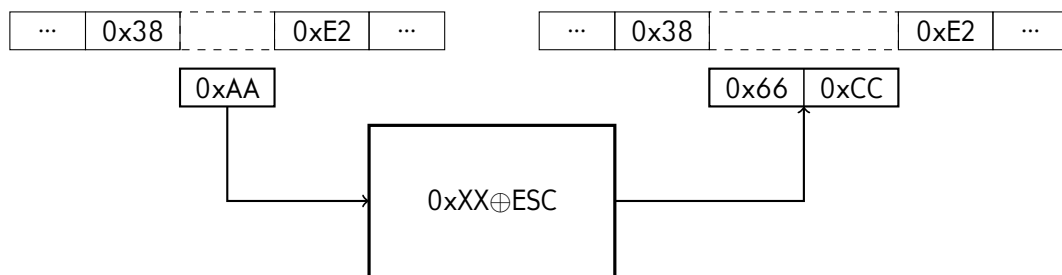


Figure 2.3: Protocol Message Escaping

Original byte	XOR	Result
STX 0x55	$\text{STX} \oplus \text{ESC} = 0x33$	0x66 0x33
ETX 0xAA	$\text{ETX} \oplus \text{ESC} = 0xCC$	0x66 0xCC
ESC 0x66	$\text{ESC} \oplus \text{ESC} = 0x00$	0x66 0x00

Table 2.2: Escape characters

## 2.3: Overview

This section gives an overview of the old Embedded Debugger and its functionality.

### Global class diagram

An overview of the software can be found figure 2.4. This overview is incomplete, since its purpose is to give an idea of how the software was set up. This means that not all classes have been drawn and not all dependencies have been set.

As can be seen in figure 2.4 the software consists of three layers. The first layer is the connector. This layer contains the communications manager, application protocol and the actual connectors<sup>1</sup>. This layer is responsible for the sending and receiving of data from an embedded platform. This includes removing application protocols <sup>2</sup>, which can be on top of the debug protocol.

The second layer is the model, this layer is responsible for storing and manipulation of data. This includes logging received data to a file or preparing the data for a plotting environment.

The third layer is the view, which contains the main form and the usercontrols. The main form is an empty shell, in which the usercontrols are loaded, with whom the user can interact with the application.

Each layer contains a so called manager. This manager is responsible for keeping track of which objects are made in the layer and stores objects for other layers to access.

The EmbeddedDebuggerApp is the base of the application, which contains the managers and is responsible for startup of the application. Since this class contains all managers, it spans across all three layers.

The complete class diagram can be found in ??.

---

<sup>1</sup>In this report, the term connector is used quite often, all of these are software connectors, examples are TCP, UDP or Serial connectors

<sup>2</sup>An application protocol is an additional protocol used for sending messages, simply put, applying the DebugProtocol twice can be seen as an additional application protocol. Examples of possible application protocols are HTTP, SFTP, SMTP.

---

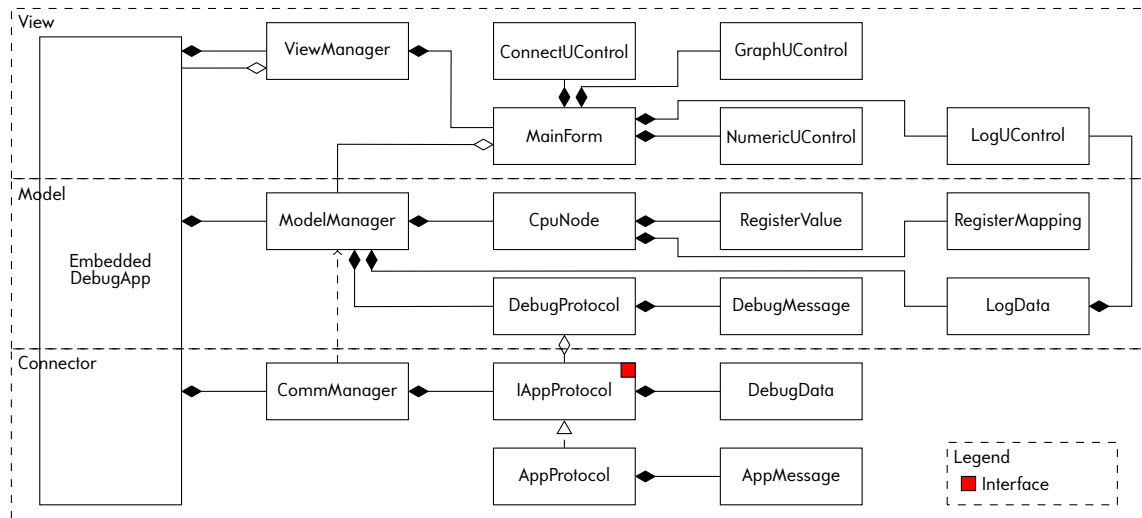


Figure 2.4: Global class diagram old Embedded Debugger

## Functionality

This subsection gives an overview of the user interface functionality of the Embedded Debugger.

### Connection tab

The first tab of the Embedded Debugger is the connection tab. This tab can be used to choose the connector and manipulate the chosen connector. This includes connecting, disconnecting and opening the connectors settings window. In this tab a list of connected embedded platforms can be found as well. A screenshot of the connection tab can be found in figure 2.5.

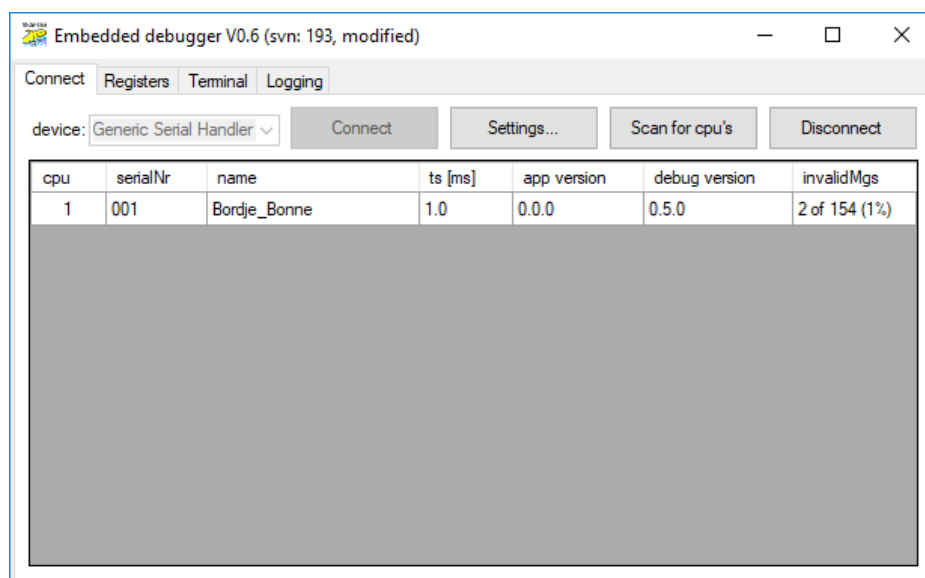


Figure 2.5: Connect tab

### Registers tab

The second tab of the Embedded Debugger is the registers tab. This tab contains two lists, a list for read registers and a list for write registers. A read register can not be written to, but can be set up as a debug channel, which is explained in section 2.2. Such registers can also be set up to plot or to log. A write register can be queried (read once) or written to.

Furthermore, the registers tab contains the plotting environment. This environment is used to plot the selected read registers. The plot autoscales, but can be zoomed whenever this is preferred. A screenshot of the registers tab can be found in figure 2.6.

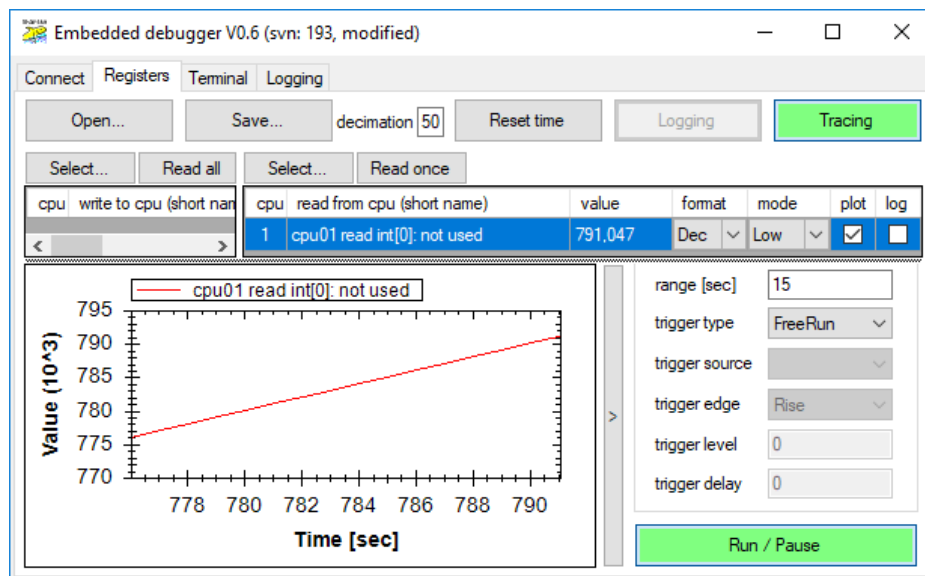


Figure 2.6: Registers tab

### Terminal tab

The third tab of the Embedded Debugger is the terminal tab. In this tab an embedded platform can be selected using the combobox. After that, debugstrings can be exchanged using the terminal. A screenshot of the terminal tab can be found in figure 2.7.

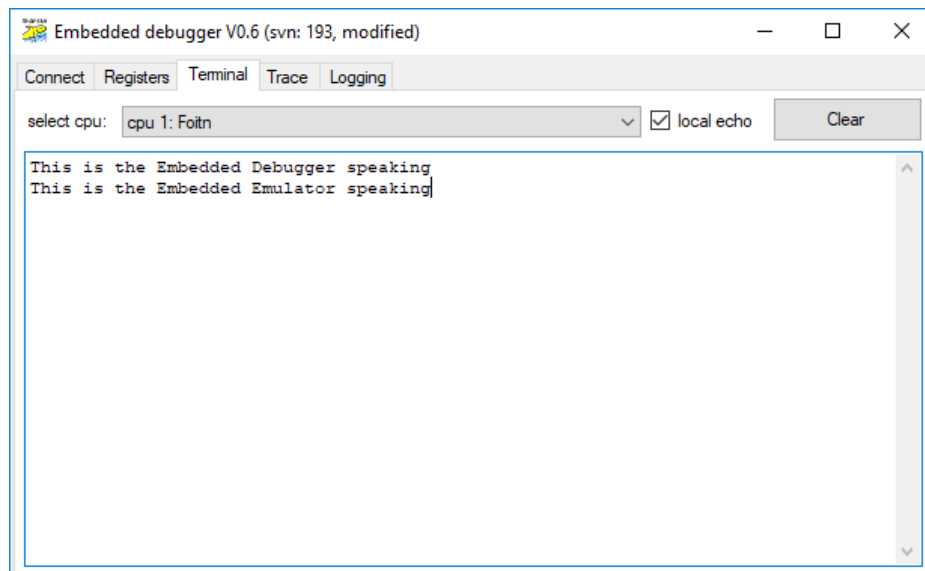


Figure 2.7: Terminal tab

## Logging tab

The fourth tab of the Embedded Debugger is the logging tab. This tab can be used to configure logging of debugchannels to a .log file. When configured, logging can be started from the logging tab or from the registers tab. A screenshot of the registers tab can be found in figure 2.8

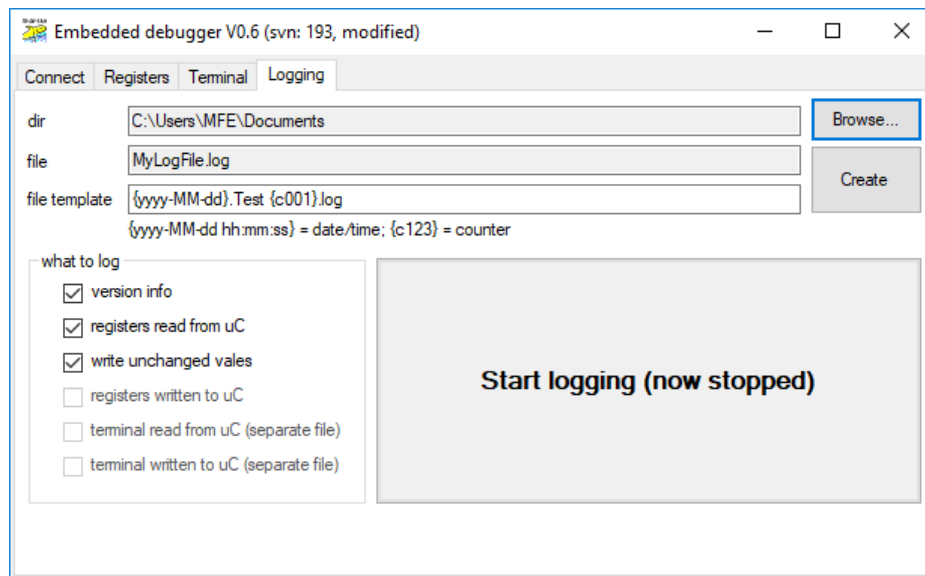


Figure 2.8: Log tab

## 2.4: Project goals

The main goal of this project is to update the current Embedded Debugger. This means that some of the branched features (in the svn repository) have to be merged into the trunk. Moreover it means that the usability of the Embedded Debugger has to be updated. This is because the user interface has some limitations, e.g. buttons that do not function. Furthermore, some new features are to be implemented in the Embedded Debugger. The complete list of project goals and their description can be found in **??**. The project goals are:

1. Debug the current debugger, or make a new design
  2. Implement a serial port interface
  3. Create the Embedded Emulator
  4. Send embedded configuration from the embedded platform to the Embedded Debugger
  5. Create an RPC interface
  6. Generate embedded configuration from compiler output
  7. Add a trace window to the GUI, which acts as a logging mechanism, containing features like logging levels
  8. Update the debug protocol, to allow a maximum of 256 debug channels at once, this is currently 16
  9. Create an interface to other languages
-



## 2.5: Issues

This section describes the issues with the old Embedded Debugger that contributed to the choice to redesign the Embedded Debugger.

### Connectors

The Embedded Debugger has been used in multiple projects, for which a connector was designed. Examples of these connectors are TCP, Serial, CAN. Some had an application protocol on top of the debug protocol. For this reason these connectors required a unique implementation. For one of the projects, a connector was build using a serial connection. This project required the connector to remove the project specific application protocol. Because of this, it is impossible to use this exact same connector for a different project. This resulted in a long time required to implement a serial connection for a next project.

### Core runs in a minimized form

The core of the Embedded Debugger, which consists of the CommunicationManager, ModelManager and ViewManager, are created in a minimized Windows Form, called the EmbeddedDebuggerApp, see section 2.3. This Form is removed from the taskbar. When the Form is forced to be visible from task manager, the Form shows up for a moment on screen and on the task bar, where after the Form is made invisible again. This consumes some recourses, moreover this can be seen as a bad programming practice.

### Graphical User Interface (GUI)

Parts of the design of the user interface does not provide a good user experience, making it difficult for a non-software engineer to work with the Embedded Debugger. An example of this, is the fact that there are two lists in the registers tab, see section 2.3. There is no explanation in the GUI itself to explain what the lists are and what the difference is between the two.

Some buttons have been added, without any proper function. An example of these buttons are the "Open..." and "Save..." button in the registers tab, see section 2.3. These can be clicked, but have nothing behind the UI element, therefore once clicked, nothing happens. Also some of the tasks are performed, taking some time and freezes the GUI, without giving feedback to the user on whether the Embedded Debugger is working or has crashed.

---

## No proper multi threading

The Embedded Debugger has a couple of threads running in the background. The main thread is the main GUI thread, which is responsible for the updating of the user interface. Another thread is also a GUI thread, even though it is never used as such. This is due to the fact that this thread runs in a minimized window, as described in section 2.5. Additional threads are used in some of the connectors, to handle sending and receiving of messages on a low level.

Whenever a message is received, the main GUI thread will take the message, decode it and run it through the DebugProtocol. If a message is send, the GUI thread will take this message, encode it and send it to the connector thread. Because this all happens in the same thread (or at some time it goes through the minimized form thread) no other tasks can be performed at the same time. An example of this is whenever the GUI is refreshed, during the repainting time, no messages can be handled. This could cause the debug protocol to not be robust, leading to ACK resending to not be send in the same period of time. Moreover it could cause an ACK not to be received before resending the message, which could lead to resending a lot of messages for no reason.

## 2.6: Conclusion

After listing the issues described in section 2.5 it was concluded that repairing the Embedded Debugger was not a good option. This would consume a lot of time and would not fix all issues, since some of these, example is the core in a minimized form, are at the core of the application. A better option was to redesign the entire application, keeping these issues of the previous design in mind. This redesign should use the DebugProtocol, to ensure backwards compatibility. Moreover, the basic design of keeping different layers separate should be used.

---

---

## Chapter 3: The new Embedded Debugger

A description of the Embedded Emulator, which was designed before the new Embedded Debugger, can be found in section 3.1. After the Embedded Emulator was finished, the new Embedded Debugger was designed, which is described in section 3.2. The implementation phase of the new Embedded Debugger is described in section 3.3. This section describes what obstacles were overcome during the implementation phase. The testing phase is described in section 3.4, which came after the implementation phase. This section goes into detail on which tests have been performed and how these were set up.

### 3.1: Embedded Emulator

To be able to design and test a new Embedded Debugger, the decision was made to start by creating the Embedded Emulator. The Embedded Emulator is a mirror like application, which simulates an embedded platform. This allows for fast and precise testing. Before any code was written or any design was made for the new Embedded Debugger, the Embedded Emulator was made. This was tested against the old Embedded Debugger, thereby ensuring the debug protocol behaved in the same way as an embedded platform would. Using the Embedded Emulator during the implementation of the new Embedded Debugger allowed for fast and precise testing and development. Because this project does not contain the target side (EDTS), no hardware had to be used. Therefore a software solution which emulates a piece of hardware was an elegant and cheap solution.

The Embedded Emulator is a mirror like application of the Embedded Debugger. It simulates an embedded platform while using the generic connectors from the Embedded Debugger. The main purpose of the Embedded Emulator is to test the Embedded Debugger, without having to use any hardware. By using `com0com` (*Null-modem emulator (com0com) - virtual serial port driver for Windows*, n.d.), two virtual COM ports are opened, thereby the Embedded Emulator can also emulate a connection over serial. By implementing a serial connection as the first connector, project goal 2 (as defined in section 2.4), was solved.

For a class diagram of the Embedded Emulator, see ??.

The Embedded Emulator contains functionality to rapidly test parts of the Embedded Debugger by auto-responding to messages. If this checkbox is checked, any of the incoming messages is responded to as the Embedded Debugger would expect from an embedded platform. When this checkbox is not checked, all of the incoming messages is responded with an ACK, thereby ensuring that the Embedded Debugger will not resend this message. The buttons on the Embedded Emulator can be used to send certain messages, such as Version, Info and the Config.

A screenshot of the Embedded Emulator can be found in figure 3.1.

---

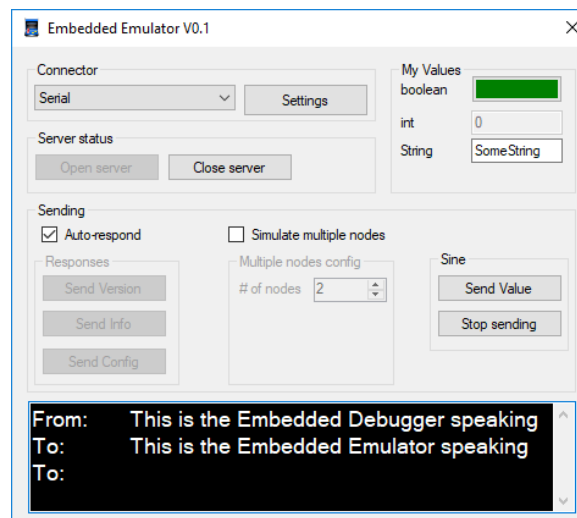


Figure 3.1: The Embedded Emulator

## 3.2: Design phase

The design phase is the first phase when setting up a software project. During this phase most of the features are laid out and a class diagram is set up. Since the use cases and sequences of the Embedded Debugger are the same as for the old Embedded Debugger and these diagrams have already been made, these are not discussed.

### Class diagram

To be able to setup the new Embedded Debugger, a global design of the software has been made. This design shows all of the basic features. If this design is properly implemented, the new Embedded Debugger should function equally to the old Embedded Debugger. A global class diagram can be found in figure 3.2. Most of the dependencies have been placed in this diagram, but since this is only to get a general idea on how the design works, some have been left out, the complete class diagram can be found in ??.

In this design it is clear to see that the application has been set up in five layers. The bottom three layers (ProjectConnectors, Connectors & DebugProtocol) are in different assemblies (.dll files). Because these are build independent of the application, these can be reused for different applications. An example of this is the Embedded Emulator, which uses both the Connectors and DebugProtocol assemblies. As described in section 2.3, the old Embedded Debugger consisted of 3 layers. Because each of these layers contained some issues, none of these have been copied to the new Embedded Debugger.

The top two layers are in the main application and have been kept separate as much as possible. Reason for this is whenever the view or the model do not function properly or it has to be updated, this can easily be done. This without having to edit anything in the other layer. An example of this is half way through the internship, the Windows Forms (WinForms) application had some limitations. Because of this, the project had to be ported to the Windows Presentation Foundation (WPF). By removing the top layer (View), which used to be WinForms, a new layer could be placed on top with WPF. This saved a lot of time and made sure the model worked exactly the same as it did before.

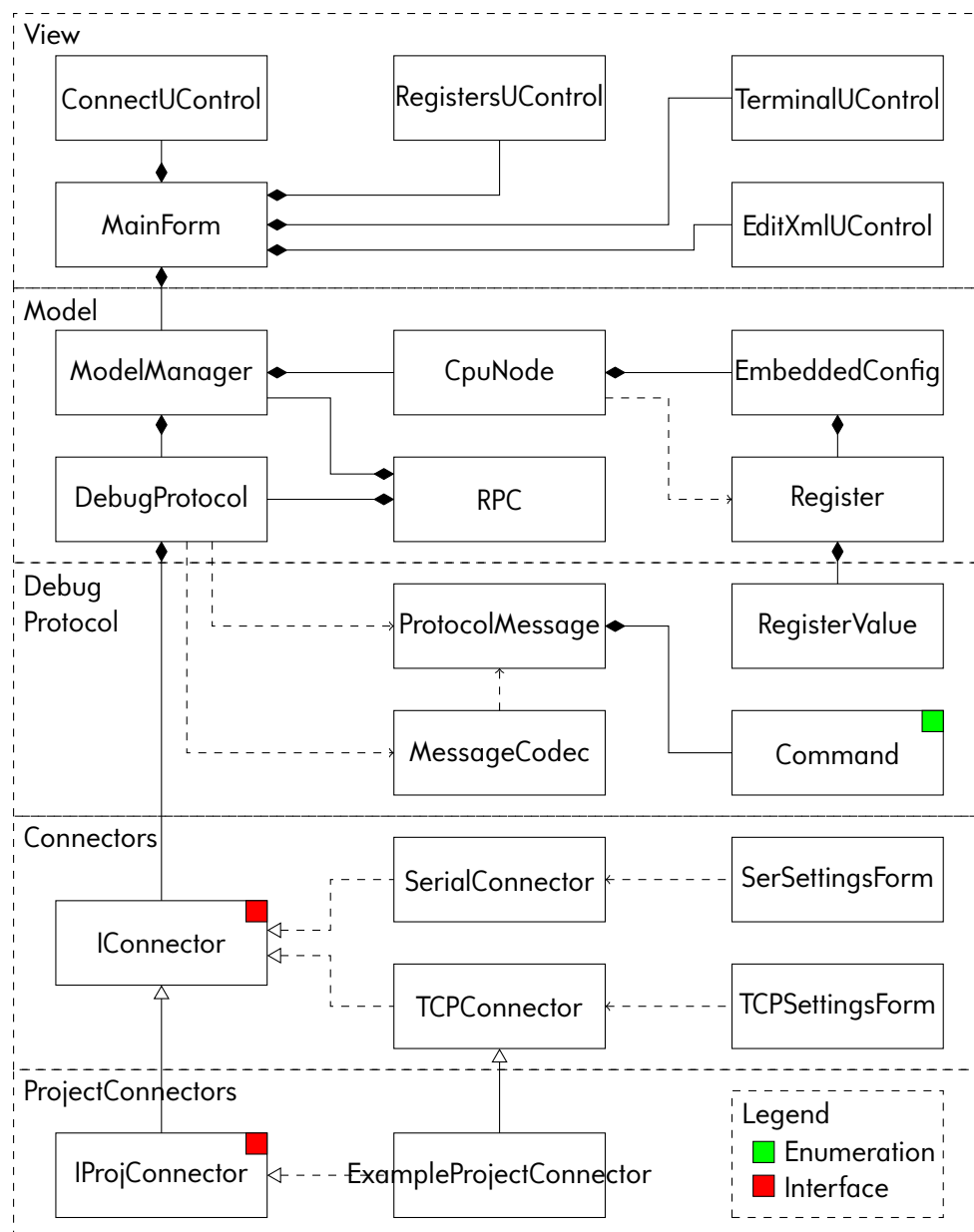


Figure 3.2: Global class diagram Embedded Debugger

## Connectors

As described in section 2.5 the connector implementation in old Embedded Debugger was not ideal. For this reason the IConnector and the IProjectConnector interfaces were created. The IConnector interface should be used to create generic connectors, examples are serial, TCP, UDP etc. These generic connectors should be tested very heavily, ensuring they are robust and cannot lead to issues. Whenever a new project comes along and no application protocol is used on top of the debug protocol, this generic connector can be used. By doing so, little time is required to start debugging with a new embedded platform.

Some projects do require an application protocol on top of the debug protocol. In this case, a new connector can be created, implementing the IProjectConnector and extending one of the generic connectors. One can then override the SendMessage and the ReceiveMessage methods, as is suggested by the IProjectConnector. Within this overridden method, the application protocol can be removed or added. Afterwards, the base class can be used to either send or receive the message. By doing this, the underlying properly tested generic connector is used for sending and receiving the data. This provides a stable connection with correct error handling.

## DebugProtocol as separate assembly

Part of the debug protocol can be used in the Embedded Emulator, which is described in section 3.1. An example is the MessageCodec class, which is used to encode and decode messages. This class is also used to calculate the CRC of a message. The Embedded Emulator has to encode and decode these messages to be able to communicate with the Embedded Debugger. By adding this class to a separate assembly, the formed .dll file can be included in the Embedded Emulator. This means that the properly tested class is not to be rewritten or copied upon change.

In future projects where the Embedded Debugger might be used in a C#/.NET application, this .dll file and the connectors can be included, which saves quite some implementation time. Another advantage is that this does not have to be compiled every time, even though that is only while debugging. In production the tool is never recompiled.

---

### 3.3: Implementation phase

For the implementation, most of the usercontrols, the windows and the look & feel of the application have been inspired by the old Embedded Debugger. This is to keep the same usability, though some of the user-friendliness has been updated. Whenever features had to be taken over, the old Debugger was taken as the base and build from that point. This has been done where possible and where this was the best solution. Example of this is the DebugProtocol, which is explained in section 2.2. The DebugProtocol is already used in some projects and there were no major flaws in it. Because of this, the DebugProtocol was kept the same, thereby ensuring backwards compatibility.

Some of the features were implemented in a different way in the old Embedded Debugger and could be improved upon. An example is the MessageCodec, where in the old Embedded Debugger the encoding and decoding was performed in the ProtocolMessage itself. Thereby containing the array of bytes and the decoded parts of the message. When lots of messages were send at the same time, this would consume more memory. This could happen whenever a lot of Debug channels are enabled and set to high frequency. The MessageCodec on the other hand "parses" the messages to byte arrays and the other way around. Therefore the original data is cleaned up, once the parsing is finished.

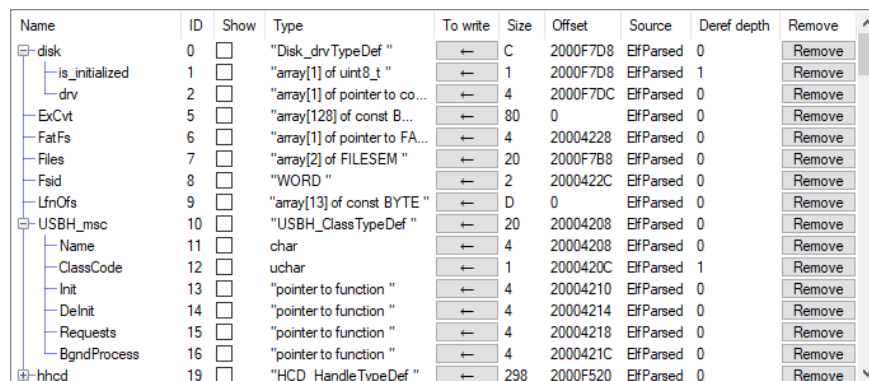
#### TreeDataGrid

During the implementation phase it became clear that in some cases it is very useful to display the embedded configuration not in just a normal list, but in a sort of treeview. This is because some values, like arrays or objects, cannot be displayed, while their elements can be. But some of these objects or arrays may not be needed for display, when generating the configuration or during operation. If the complete array is shown in the list, this could result in a very large list, with values that are not important for that moment. Therefore it is interesting to display this in a treelike manor. This means that the array can be collapsed or expanded when needed.

For WinForms, there is a free library to create a listview combined with a datagrid, called ObjectListView (Piper, 2016). This library contains several controls, examples are ObjectListView, TreeListView, DataListView and DataTreeListView. For this project the control "DataTreeListView" was used, this is a listview and datagrid combined. This means that a register can contain childregisters, which can, in the usercontrol, be expanded. An example of this can be found in figure 3.3. This figure shows part of an embedded configuration. The "USBH\_msc" object contains some properties/fields, these can be found under the object. This cleans up the list and makes it more structured.

Halfway through the internship, the WinForms application was ported to WPF. For WPF there are no free listview libraries with the same functionality as the TreeDataListView.

---

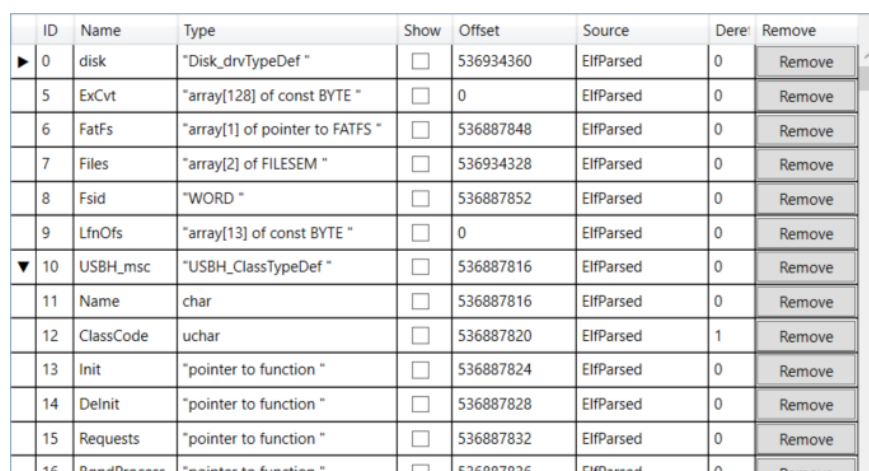


Name	ID	Show	Type	To write	Size	Offset	Source	Deref depth	Remove
disk	0	<input type="checkbox"/>	"Disk_drvTypeDef "	←	C	2000F7D8	ElfParsed	0	Remove
is_initialized	1	<input type="checkbox"/>	"array[1] of uint8_t "	←	1	2000F7D8	ElfParsed	1	Remove
drv	2	<input type="checkbox"/>	"array[1] of pointer to co...	←	4	2000F7DC	ElfParsed	0	Remove
ExCvt	5	<input type="checkbox"/>	"array[128] of const B...	←	80	0	ElfParsed	0	Remove
FatFs	6	<input type="checkbox"/>	"array[1] of pointer to FA...	←	4	20004228	ElfParsed	0	Remove
Files	7	<input type="checkbox"/>	"array[2] of FILESEM "	←	20	2000F7B8	ElfParsed	0	Remove
Fsid	8	<input type="checkbox"/>	"WORD "	←	2	2000422C	ElfParsed	0	Remove
LfnOfs	9	<input type="checkbox"/>	"array[13] of const BYTE "	←	D	0	ElfParsed	0	Remove
USBH_msc	10	<input type="checkbox"/>	"USBH_ClassTypeDef "	←	20	20004208	ElfParsed	0	Remove
Name	11	<input type="checkbox"/>	char	←	4	20004208	ElfParsed	0	Remove
ClassCode	12	<input type="checkbox"/>	uchar	←	1	2000420C	ElfParsed	1	Remove
Init	13	<input type="checkbox"/>	"pointer to function "	←	4	20004210	ElfParsed	0	Remove
Delnit	14	<input type="checkbox"/>	"pointer to function "	←	4	20004214	ElfParsed	0	Remove
Requests	15	<input type="checkbox"/>	"pointer to function "	←	4	20004218	ElfParsed	0	Remove
BgndProcess	16	<input type="checkbox"/>	"pointer to function "	←	4	2000421C	ElfParsed	0	Remove
hhcd	19	<input type="checkbox"/>	"HCD_HandleTypeDef "	←	298	2000F520	ElfParsed	0	Remove

Figure 3.3: DataTreeListView example (WinForms)

For this reason the challenge arose to design a custom control. Since the Embedded Debugger is not a project for a customer, functionality and user-friendliness were the only requirements. Looks were not taken into consideration.

The standard WPF DataGrid from Microsoft comes very close to what this component requires. It shows values and can draw itself, based on a list of objects. The only feature that it lacks is the treeview part. Therefore a new component was made, the "TreeDataGrid". The TreeDataGrid extends the DataGrid component, therefore having all of its functionality. Within the constructor of the component, a column is created, which is responsible for the treeview functionality. If a Register (one Register per row) contains any childregisters, an arrow '▷' is shown, thereby indicating that there are childregisters. When this arrow is clicked, the childregisters are inserted below the Register in the list. Moreover, the arrow is changed into a '▽'. When this arrow is clicked again, these childregisters are removed from the list again. An example of the TreeDataGrid can be found in figure 3.4.



ID	Name	Type	Show	Offset	Source	Dere	Remove
▶ 0	disk	"Disk_drvTypeDef "	<input type="checkbox"/>	536934360	ElfParsed	0	Remove
5	ExCvt	"array[128] of const BYTE "	<input type="checkbox"/>	0	ElfParsed	0	Remove
6	FatFs	"array[1] of pointer to FATFS "	<input type="checkbox"/>	536887848	ElfParsed	0	Remove
7	Files	"array[2] of FILESEM "	<input type="checkbox"/>	536934328	ElfParsed	0	Remove
8	Fsid	"WORD "	<input type="checkbox"/>	536887852	ElfParsed	0	Remove
9	LfnOfs	"array[13] of const BYTE "	<input type="checkbox"/>	0	ElfParsed	0	Remove
▼ 10	USBH_msc	"USBH_ClassTypeDef "	<input type="checkbox"/>	536887816	ElfParsed	0	Remove
11	Name	char	<input type="checkbox"/>	536887816	ElfParsed	0	Remove
12	ClassCode	uchar	<input type="checkbox"/>	536887820	ElfParsed	1	Remove
13	Init	"pointer to function "	<input type="checkbox"/>	536887824	ElfParsed	0	Remove
14	Delnit	"pointer to function "	<input type="checkbox"/>	536887828	ElfParsed	0	Remove
15	Requests	"pointer to function "	<input type="checkbox"/>	536887832	ElfParsed	0	Remove
16	BgndProcess	"pointer to function "	<input type="checkbox"/>	536887836	ElfParsed	0	Remove

Figure 3.4: TreeDataGrid example (WPF, custom control)



## DebugProtocol V1.0

To be able to implement goal 8, "Update the debug protocol, to allow a maximum of 256 debug channels at once, this is currently 16" as stated in section 2.4, a part of the DebugProtocol needs to be changed. The ConfigChannel option in the DebugProtocol contains one byte for the channel number, for which 0x0F is set as the maximum value. This 0x0F is 16 in decimal, which is the maximum number of channels. Since half of this byte is still available, it is easy to upgrade this to 256 channels.

However, when looking at the ReadChannelData command, it can be seen that there are two bytes reserved for the mask. The ReadChannelData is send by the embedded platform, whenever new register data for a register is available. The mask, these two bytes, contains 16 bits, one for each debug channel. If a value is send for e.g. channel five, bit five is set to 1. This means that if the DebugProtocol needs to be able to handle 256 channels, 256 need to be reserved. This means that 32 bytes have to be reserved for the mask. This is an increase of 1600% for the mask, which is a lot. Smaller microcontrollers might not be able to handle this amount of debug channels in the first place. Because of this, the decision was made to redesign the DebugProtocol.

This redesign is V1.0 of the DebugProtocol and takes the previous version, V0.7, as a basis. The redesign was made to be as modular as can be. This means that it can be made small for normal microcontrollers, but can be made quite big when needed for entire systems. One of the ways this was designed, is to send the maximum amount of DebugChannels for that specific embedded platform upon initialization. In this way the previously described mask, is modular and the size is determined on the amount of channels.

Another example is the number of registers, this was previously set to 4 bytes, which limits the DebugProtocol to 4GB of data. For a normal embedded platform this is quite enough, but possibly for a complete system, running an operating system and much more, 4GB is not enough. The maximum number of registers can also be send during initialization, thereby it is also modular.

The users of the Embedded Debugger had some additional features which could be implemented in the DebugProtocol. Currently any register is either a readable or writable, meaning that a write register can never be plotted or that a read register can never be written to. This is correct in most cases, but in some cases these are both true or neither is true. An example of both readable and writable a timer\_tick, which needs to be set to a certain value or reset, after which is could be plotted. An example of neither can be an array of integers, these integers can be readable or writable, but it is not needed to read/write the array itself. Because of this, the ctrl byte, which is used in a couple of commands, had to be changed. Currently the ctrl byte contains three properties of a register.

---

These properties are:

- If the register is readable or writable (bit 7)
- What the source of the register is (absolute offset, relative offset, etc.) (bits 6-4)
- The number of dereferencing that has to be performed (pointer to pointer to pointer, etc) (bits 3-0)

The second one, the source of the register is no longer needed, since the new approach is for the embedded platform to decide where exactly the register is placed. Therefore three bits are free for use. In version 1.0 of the DebugProtocol, bit 7 is used when a register can be read from and bit 6 when this register can be written to.

The third one, for the dereferencing can still be useful according to colleagues and because there is nothing (yet) to fill up this half a byte, this has been left in. Since in the end of the internship there was not enough time left to implement V1.0, this is added to the recommendations.

---

## 3.4: Test phase

### Unit testing

Parts of the Embedded Debugger have to perform to a certain standard. By using unit tests, most bugs and minor mistakes can be filtered out. An example of this is the MessageCodec class, which can be found in **??**. This class is responsible for encoding and decoding messages send from and to the embedded platform.

### Decoding

For decoding, this class receives an array of bytes. It takes out any message by finding all corresponding startbyte (STX) and endbyte (ETX). This message is checked against the CRC at the end of the message, if this is correct, this is a valid message. The next step for decoding is to place all received data from the message in a ProtocolMessage object. This includes the CPU-id, MSG-id, command and the actual data. If any additional bytes have been send in this buffer, these are stored for the next message, since it is possible that a message send over different buffers. A graphical overview of decoding a message can be found in figure 3.5.

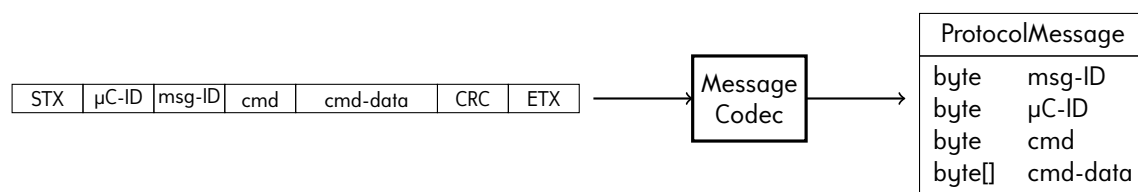


Figure 3.5: Decoding a protocol message

### Encoding

For encoding, this class receives a ProtocolMessage object and converts it into an array of bytes. The MessageCode creates an array, one byte for the STX, ETX, μC, cmd and CRC and the number of bytes of the cmd-data. The μC, msg-ID, cmd and cmd-data are places in the array, after which the escape characters are added, which is explained in section 2.2. Once all of this is in place, the CRC is calculated and placed in the second to last place in the array. Last of all, the STX and ETX are place. A graphical overview of encoding a message can be found in figure 3.6.

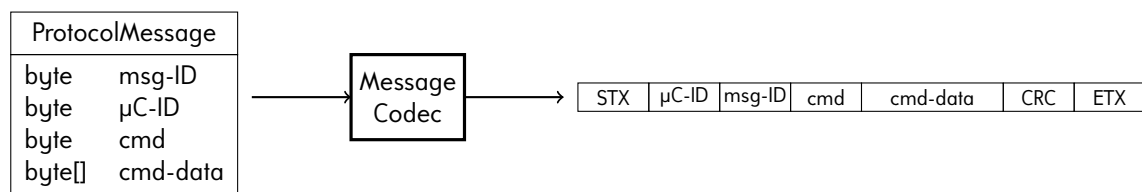


Figure 3.6: Encoding a protocol message

Testing for this class is very useful, since the decoding and encoding has to be reliable when this is used in the Embedded Debugger. For this class unit testing is the best method, since this is mostly comparing input against output, which can be tested with unit tests. These tests consist of most (if not all) scenarios that can happen during operation. Examples of these are to decode an array with only one message and no other bytes, which is a happy flow test. One of the more advanced and less common scenarios is that one or more messages are spread over multiple buffers. Another scenario is when a message (the byte array) contains not escaped characters (STX, ETX or ESC). As explained in the debug protocol these should be escaped whenever used, therefore these not escaped characters may never appear in a message byte array.

One of the main advantages of unit testing, aside from verifying that the code works, is that after any change in the code, these tests can be rerun. With every rerun, the developer can verify that the recent changes did not change anything to this part of the application or that the changes did not break anything. Therefore no bugs are introduced in these specific parts of the application.

## Winforms limitations

Initially, the application was written using WinForms. After the initial version and testing by colleagues it became clear that some parts of the application were reacting very slowly and made it difficult to work with. An example of this is the list of registers in the Registers tab. This lists refresh rate dropped depending on the amount of rows it contained. When the amount of rows was below 10, it would perform as expected and this was not an issue at all. If this amount got above 1000 entries, the lists refresh rate would drop to the point where the application would not function properly. This made it difficult to work with the application.

Because this was a large issue, some applied research was performed. This research mostly entailed searching through Stack Overflow and other forums. In the discussions in these forums, it became clear that solving this issue with WinForms would take a lot of time and had a high failure rate. Most of the advice that was given in these discussions was to start over (only for the view) with WPF. WPF is the newer variant made by Microsoft and contains more features.

When using WPF, the windows and user controls are designed by writing XAML instead of dragging and dropping elements. These WinForms elements would have to be configured after setting it to the correct position. Some of these elements only had limited functionality, where this functionality was implemented in WPF. WPF also gives more possibilities and more freedom to the programmer. An example of this is the ability to create a template column for a DataGrid. This allows the programmer to design a column containing e.g. a combobox (dropdownmenu) with all entries of an enum without having to add this programmatically. Programmatically means that the programmer would have to write C# code to configure the DataGrid, even though the .NET framework is set up to do this in the XAML/Control editor. By binding this column to a property of a list of objects, this property is set whenever the combobox set to a different value. This was a good reason to port the project to WPF.

Because the WinForms project was setup with different layers (as is described in section 3.2), the view could be replaced. This without having to change anything to the model. After the replacement of the view, the model and the view had to be connected and the entire application was up and running again.

Testing the WPF application against the WinForms application is not an easy task. The main reason for this, is that the main problem was that the WinForms was hard to work with for a user. For this reason this test was performed by rolling out an initial version of the WPF application to some colleagues. These colleagues use the Embedded Debugger often (some on a daily basis) and could easily spot an improvement. After a week of testing, the feedback from these colleagues was very positive, because the applications performance was as expected. Because of this it can be concluded that porting the application to WPF was a success.

---

An example of XAML (the ConnectorChooserUserControl) can be found in listing 3.1. The result of this XAML can be found in section 3.4. This is what every line of code does.

- Line 1 states that this is a user control, which class it belongs to and sets the initial height and width of the usercontrol
- Line 2 adds a grid to the usercontrol with a margin all around of 3 pixels
- Line 3 starts the column definitions for the grid
- Line 4 sets the width of the first column to 75
- Line 5 sets the width of the second column to \*, which means all remaining space
- Lines 6-8 sets the width of the remaining columns to 90
- Line 9 closes the column definitions
- Line 10 adds a label saying "Connector" to the usercontrol in column 0<sup>1</sup>
- Line 11 adds a combobox in column 1, with an eventhandler "ConnChanged", which changes when the selected item changes<sup>2</sup>
- Line 12 adds a button for connecting in column 2, with an eventhandler upon click
- Line 13 adds a button for settings in column 3, with an eventhandler upon click
- Line 14 adds a button for disconnecting in column 4, with an eventhandler upon click
- Line 15 closes the grid
- Line 16 closes the usercontrol

Listing 3.1: ConnectorChooserUserControl.XAML

```

<UserControl x:Class="EmbeddedDebugger.View.UserControls.ConnectorChooserUserControl"
    ↪ d:DesignHeight="35" d:DesignWidth="600" >
  <Grid Margin="3">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="75"/>
5      <ColumnDefinition Width="*/>
      <ColumnDefinition Width="90"/>
      <ColumnDefinition Width="90"/>
      <ColumnDefinition Width="90"/>
    </Grid.ColumnDefinitions>
10  <Label Grid.Column="0" Margin="3 0" Content="Connector"/>
    <ComboBox Grid.Column="1" Margin="3" Name="ConnectorChooserCB" SelectionChanged="ConnChanged"/>
    <Button Grid.Column="2" Name="ConnectButton" Content="Connect" Click="Connect_Click"/>
    <Button Grid.Column="3" Name="SettingsButton" Content="Settings" Click="Settings_Click"/>
    <Button Grid.Column="4" Name="DisconnectButton" Content="Disconnect" IsEnabled="False"
    ↪ Click="Disconnect_Click"/>
15  </Grid>
</UserControl>

```



Figure 3.7: ConnectorChooserUserControl - result of listing 3.1

<sup>1</sup>Columns and rows are 0 based.

<sup>2</sup>These eventhandlers are available in the C# code

## Chapter 4: Additional functionality

This chapter describes the additional functionality that was build into the Embedded Debugger after the basics were implemented. Section 4.1 describes sending the Embedded Configuration over the DebugProtocol and how this was implemented. Section 4.2 describes how the Embedded Configuration was retrieved from compiler output. The embedded configuration can contain some Simulink variables, which have to be generated from the C API, this is described in section 4.3 Section 4.4 describes what RPC is, how it was implemented in the Embedded Debugger and it gives an example.

### 4.1: Embedded configuration over DebugProtocol

The old Embedded Debugger used XML files to store an embedded configuration. This embedded configuration describes which register can be written to or read from and at what memory address this register is. It also describes which type and size this register is. An example of this is can be found in listing 4.1. This example contains two registers, one read register and one write register.

Listing 4.1: Embedded configuration XML

```
<Read>
  <Register type="int" size="4" name="myReadInt" address="0x000384"/>
</Read>
<Write>
5  <Register type="int" size="4" name="myWriteInt" address="0x000385"/>
</Write>
```

One of the problems with this approach is that multiple engineers need to work with this configuration. If this configuration has to be e.g. send over the mail or transferred via a usb-stick, this takes a lot of unnecessary time. Another problem is that the programmer needs to write this XML file for every embedded platform, which takes a lot of time. Moreover, after each recompile of the software, the configuration needs to be checked whether the registers are at the same location as before.

To tackle both of these problems, the idea came to life to send the Embedded configuration over the DebugProtocol. This means that the embedded platform sends its own configuration upon request. The result is that no XML file has to be created manually and that it doesn't have to be available on server.

To implement this in the DebugProtocol, the command 'E' was chosen, since 'C' was already in use. The Embedded Debugger requests the configuration by sending a message, with command 'E' and an empty cmd-data. The embedded platform responds with the entire configuration, one register per message and ends with the number of registers send. If the number of registers send does not equal the number of messages send, the Embedded Debugger will request the configuration again.

For an overview of this message exchange, see table 4.1. Row 1 is the Embedded Debugger, requesting the configuration. Row 2 is a template of how a register is send, for a list of what all these commands mean, see table 4.2. Row 3 is the final message send, which contains the number of registers send, which finalizes the transfer of the configuration.

Direction	cmd	cmd-data										
PC → $\mu$ C	'E' = 0x45											
PC ← $\mu$ C	'E' = 0x45	rRN3	...	rRN0	nmS	nmN	...	nm0	idN	...	id0	size type ctrl
PC ← $\mu$ C	'E' = 0x45	trRN3	...	trRN0								

Table 4.1: Embedded Configuration overview

Command	Description
rRN4 ... rRN0	The running number of the register in this message <sup>1</sup>
nmS	The number of bytes used for the name of the register
nmN ... nm0	The name of the register
idN ... id0	The ID of the register, size N is determined in GetInfo as 0x0B <sup>2</sup>
size	The size of the register
type	The type of the register (int, double, float, etc)
ctrl	Control byte: bit 7: Readable register bit 6: Writable register bit 3-0: pointer depth, how deep to dereference

Table 4.2: Embedded Configuration commands overview

<sup>1</sup>This number must be incremental and must be a running number

<sup>2</sup>This can be an absolute memory address, but is irrelevant for the Embedded Debugger



## 4.2: Embedded configuration from compiler output

In case the embedded platform is not capable of sending the configuration over the protocol, the compiler output can be used to generate this configuration. For the sake of simplicity and because it has been used for the configuration before, the parsed configuration will be stored as an XML file. Parsing is more intensive than reading an XML file, therefore the configuration is not parsed every time the debugger requires the configuration.

There are multiple possibilities to generate the configuration. There are some open source options, one of these is elf-parser by TheCodeArtist<sup>3</sup>. After some trying, it became clear that this library only allowed some of the user defined registers to be listed.

A better possibility is the fromelf application provided by the KEIL IDE. This application takes a .ELF or .AXF file and generates a .txt file. Each line contains a variable, described in the variable name, type, size in bytes and memory address. By capturing and parsing the output of this tool, this can be turned into an embedded configuration in which the required information about the registers and their use is known. This configuration is shown to the user, in which the user can select the desired registers that will be shown in the configuration once the embedded platform is connected. Afterwards, this configuration is written to a .XML file, storing it for future use. The fromelf application lists every register that is build in the .ELF file, even listing some of the registers basic registers, including configuration, this allows for on the go configuring the embedded platform.

As can be seen in figure 4.1, the usercontrol contains a the TreeDataGrid, see section 3.3. This allows for an array to be collapsed and selected/deselected as a whole, while still being able to only select one element. The structure is build on the output of the fromelf tool.

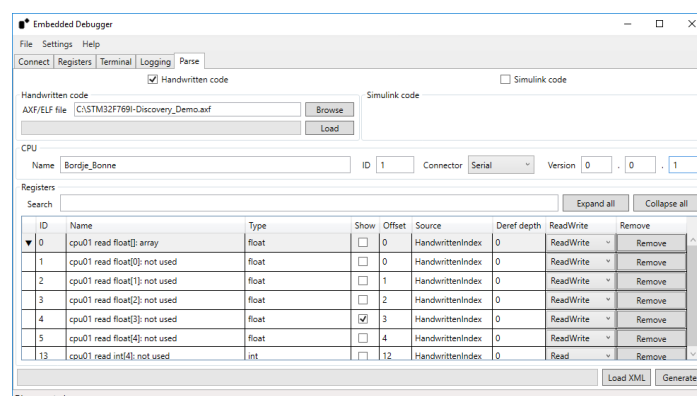


Figure 4.1: The parsing user control

<sup>3</sup><https://github.com/TheCodeArtist/elf-parser>

### 4.3: Simulink parsing

For some projects Simulink is used to create control loops. A mechatronic system engineer creates a control loop and generates C code. This C code is afterwards implemented in the embedded platform software. To be able to use the naming scheme used in Simulink, the C API needs to be parsed. This is because the variable names are different from the strings used in the C API.

During this internship, a little research has been conducted on this C API. The best way of going about this is to compile the C API and request the names from the API. Within the C#/.NET application, it is not an easy task to start compiling this API and request features. It is possible to extract the information by going through the source code. The downside to doing this is that whenever MATLAB (creators of Simulink) change anything to the C API, this parser might not work anymore. This means that the parser would not be robust and the Embedded Debugger might need to be reconfigured. Because of this, further research has been postponed. More about this can be found in the recommendations.

---

## 4.4: RPC

The Embedded Debugger is a useful tool for debugging, but it can also be used to test an embedded platform. An example of this could be a small electrical circuit which consists of a microcontroller, an LED and an LDR. Using the Embedded Debugger, the register responsible for the LEDs state can be set to on, thereby turning the LED on. Afterwards the value for the LDR can be requested, thereby verifying that the LED was turned on. To automate this process, an RPC library can be used.

An RPC library opens a port (in this case a TCP port) on the application, where procedures can be called upon. This allows another program (independent of programming language, since this uses a standard protocol) to manipulate the Embedded Debugger. An overview of this can be found in figure 4.2. By using RPC, automated testing can be implemented from e.g. a Python script.

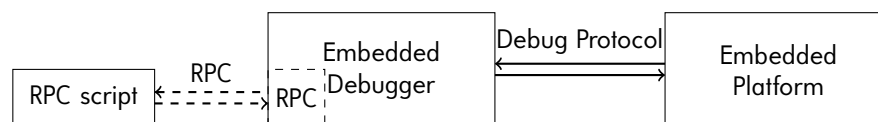


Figure 4.2: Overview of RPC connection

For RPC libraries there are two options for the type of communication structures. These two structures are:

- XML
- JSON

The XML is the older and more readable variant, while JSON is newer and more optimized for faster communication. For this reason, JSON is preferred.

After some research on C#/.NET RPC libraries, it became clear that there were a couple of JSON libraries applicable. The three that have been tested during this project are

- json-rpc-csharp <sup>4</sup>
- json-rpc.NET <sup>5</sup>
- vs-streamjsonrpc <sup>6</sup>

---

<sup>4</sup><https://github.com/adamashton/json-rpc-csharp>

<sup>5</sup><https://github.com/Astn/JSON-RPC.NET>

<sup>6</sup><https://github.com/Microsoft/vs-streamjsonrpc>

---

The main focus of the RPC is to enable a python script to connect to the Embedded Debugger. This means that the available python libraries are a leading factor in the choice of the C#/.NET library. The main libraries that use JSON for the RPC, use HTTP for the communication protocol. Examples of these are:

- python-jsonrpc <sup>7</sup>
- jsonrpclib <sup>8</sup>

The before mentioned C#/.NET libraries do not support HTTP for communication, only the standard TCP.

For this reason the focus was shifted to an XML RPC library. Python has a standard library which is installed by default, called xmlrpclib. This library is very straight forward and allow the user to call a procedure as if it was an actual function of the object. Example of this can be found in listing 4.2. This example would retrieve the current value of register 12 on the embedded platform with id 0.

Listing 4.2: Example of retrieving a register value

```
1 server.GetRegisterValue(0, 12)
```

For the C#/.NET XML RPC library, a very largely used library is the XML-RPC.NET created by Cook Computing (Cook Computing, 2011). This library is widely used, frequently updated, has a large community and has a lot of examples. The basics of the library are described in the examples and these can be implemented without a lot of required changes. This class can be found in `??`. The `RPCInterface` class uses the `RPCResolver` to forward the procedure calls into the Embedded Debugger. By separating these two classes, the functionality of the Embedded Debugger is only described in the `RPCResolver`, which can be found in `??`. This leaves the `RPCInterface` to be an abstract class, which can easily be used in future projects, by simply copying this `RPCInterface` and writing the `RPCResolver` for the logic of the application. A graphical overview of this can be found in figure 4.3.

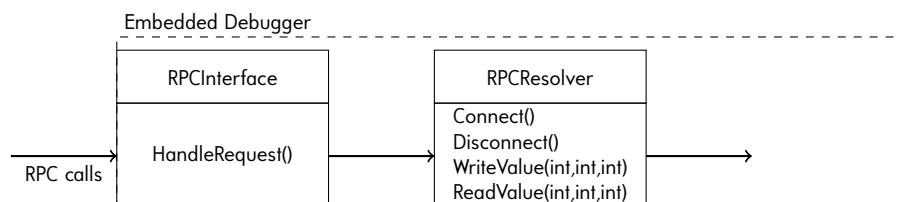


Figure 4.3: RPC interface & RPC resolver

<sup>7</sup><https://github.com/gerold-penz/python-jsonrpc>

<sup>8</sup><https://github.com/joshmarshall/jsonrpclib>

An example of the complete system can be found in listing 4.3. A breakdown of this example line by line:

- Line 1 imports the required part of the xmlrpc library (this library has been renamed to xmlrpc in Python 3)
- Line 2 imports time, which enables usage of delays
- Line 3 creates a connection to the Embedded Debugger using localhost, port 11000 over HTTP
- Line 4 connects the Embedded Debugger to an embedded platform over a serial interface, using port COM50 and baud rate 9600
- Line 5 lets the Embedded Debugger set the value of 38294 on the embedded platform with id 0 on register with id 200
- Line 6 sets channel 5 up for logging
- Line 7 lets the Embedded Debugger know that it is time to start logging, including the path, filename, if it has to be a different file for each cpu, the separator, timestampUsage and if a header is used
- Line 8 lets the script sleep for 10 seconds
- Line 9 tells the Embedded Debugger to stop logging values
- Line 10 disconnects the Embedded Debugger from the embedded platform(s)

Listing 4.3: Example of setting a register value

```
1 from xmlrpc import client
2 import time
3 server = client.ServerProxy("http://localhost:11000")
4 server.ConnectSerial("COM50",9600)
5 server.SetRegisterValue(0, 200, 38294)
6 server.SetLogging(0, 5, True)
7 server.StartLogging("C:/Temp", "MyCsvFile", False, ";", 1, True)
8 time.sleep(10)
9 server.StopLogging()
10 server.Disconnect()
```

As shown in listing 4.3, using the RPC library it is very easy to manipulate the Embedded Debugger. This can be used in such a way that this script is expanded, enabling it to log to a file, read some register values and afterwards analyse these. If these values and logs are found to fulfil the test requirements for the embedded platform, it can be considered good enough. This allows for automated testing in which an operator has to press only a single button, all values are checked and the operator is told the result. The result of this is a very fast and accurate test, which is not prone to human error (except for reading the result wrongly).

The complete manual of the Embedded Debugger RPC can be found in ??.

## Chapter 5: Results

This chapter describes the result of the internship project starting with the GUI in section 5.1. Afterwards, the Embedded Emulator is described in section 5.2.

### 5.1: GUI

This section gives an overview of what the GUI looks like. The WinForms variant is the old Embedded Debugger, as described in chapter 2, while the WPF is the new Embedded Debugger, as described in chapter 3.

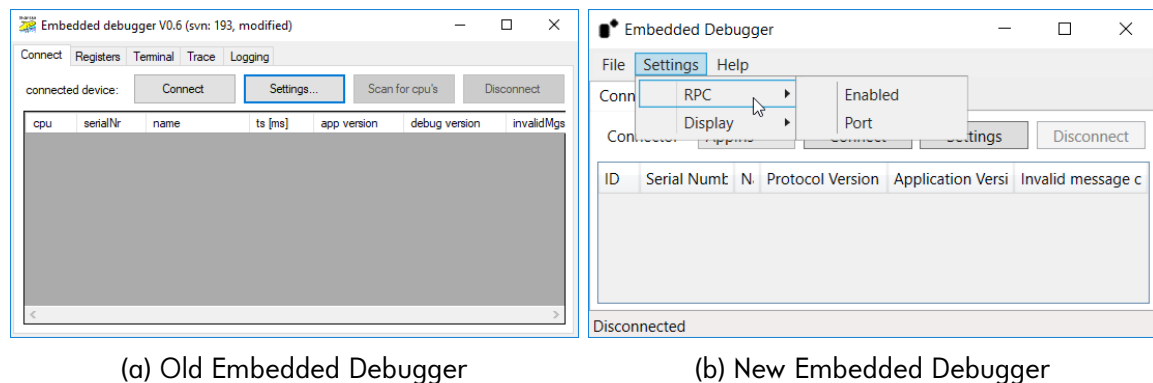
The main window, the shell of the application is described first. After which the tab controls are explained. The tabcontrol contains tabs which contain the actual pages of the application.

#### The main window

The main window of the old Embedded Debugger was empty, so no settings, no status-bar, nothing. The new Embedded Debugger has a status-bar, indicating whether the Embedded Debugger is currently connected and which connector it is using. This can be useful when looking at the Registers tab, to know which connector is being used and whether a disconnect happened.

The new Embedded Debugger also contains a couple of menu-bars, File, Settings and About. File currently has an "Exit" button, which can be used to shutdown the application. This can be expanded upon when required. Settings contains a submenu for the RPC, which contains an enable/disable button and a button, which opens a window, to set the targeted port for RPC. Moreover it contains a Display submenu, in which currently only the displaying of variable types can be set. When this is enabled, all variable types are set to their C++ names, instead of their C# names (e.g. short becomes int16\_t). Last of the menus is the Help submenu, which contains an About button, clicking this opens an aboutwindow, in which some about states are described. An overview of the main window of the new Embedded Debugger can be found in figure 5.1b. To compare this with the old Embedded Debugger, see figure 5.1.

---



(a) Old Embedded Debugger

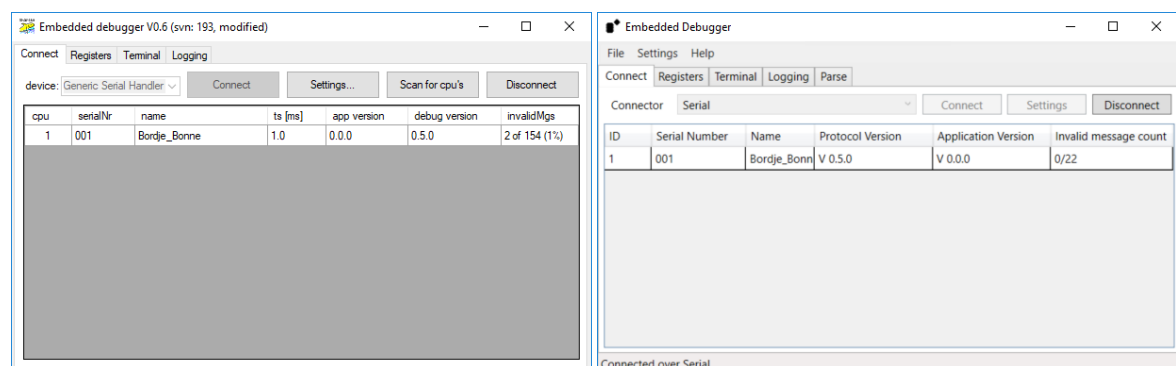
(b) New Embedded Debugger

Figure 5.1: Main window Embedded Debugger

## Connections

The connections tab allows for the user to manipulate the connector. This connector can be chosen using the dropdownmenu, configured via the settings window, which opens by clicking the settings button, connecting and disconnecting. After the connect button has been clicked, the connector connects to the given port and searches for embedded platforms. Any embedded platforms that respond to the search are listed in the datagrid.

The button "scan for cpu's" has been removed in the new application, since users (almost) never used it, one can easily reconnect when needed. The connector list used to be filled by a hardcoded list of connectors, which means that whenever a new connector is added to the application, this connector needs to be added to the list as well. In the new Embedded Debugger this has been updated to a list that uses reflection. This means that any class in the application that implements the IConnector or IProjectConnector interface is added to this list. Result of this allows the programmer to add a new connector, which is automatically added to the list.



(a) WinForms

(b) WPF

Figure 5.2: The Embedded Debugger: Connections

## Registers and plotting

The registers tab is responsible for displaying the list of registers and their current values. Whenever a value is changed, the list will update, thereby showing the newly received value. The search bar on top can be used to filter the list of registers by name. The user can turn on logging, plotting, set the channel mode and set the formatting for any register in the list. The button at the end of a row can be used to retrieve the current value of that certain register at any moment. In the bottom half of the user control, the plotting environment can be found. If plotting is turned on for any of the registers, the plot will be displayed in here. The plot can be paused whenever a closer look at a certain moment is required.

In the new application the two lists have been merged into one, since there is not only read or write, but read, write, both or none. An example of both readable and writable is a `timer_tick`, which needs to be set to a certain value or reset, after which it could be plotted. An example of neither can be an array of integers, these integers can be readable or writable, but it is not needed to read/write the array itself. This difference in `ReadWrite` is displayed to the user by enabling or disabling certain aspects of this list. For example, if a register is writable, the textbox in the value column is writable, otherwise it is read-only. The same goes for the channelmode, plotting, linecolor and logging, these are only available for readable registers. The refresh button is available when a register is readable or writable, but disabled if `ReadWrite` is set to none.

Furthermore the top part of the usercontrol has been cleaned up. The "Open" and "Save" button in the old Embedded Debugger had no function and are therefore removed. These have been replaced by a dropdownmenu in which the processor can be selected or when required, all connected processors can be displayed in the list. Moreover, a searchbar has been added, which can be used to filter registers in the list, which is useful when a large list is present.

A button has been added with which all `DebugChannels` can be turned off all at once. This is useful since the Embedded Debugger turns on as many `DebugChannels` as possible at once upon connect. Whenever another register is to be added as `DebugChannel`, this is not possible and another has to be turned off first.

Another update to this usercontrol is the plotting environment. The old environment was not available for WPF and colleagues pointed out that this is not a userfriendly environment to work with as a programmer. Therefore a new one was selected, that is used in a lot of projects, is highly maintained and is userfriendly to edit. A pause button has been introduced, which allows the user to pause the plotting environment at any given moment and analyze the plot.

---



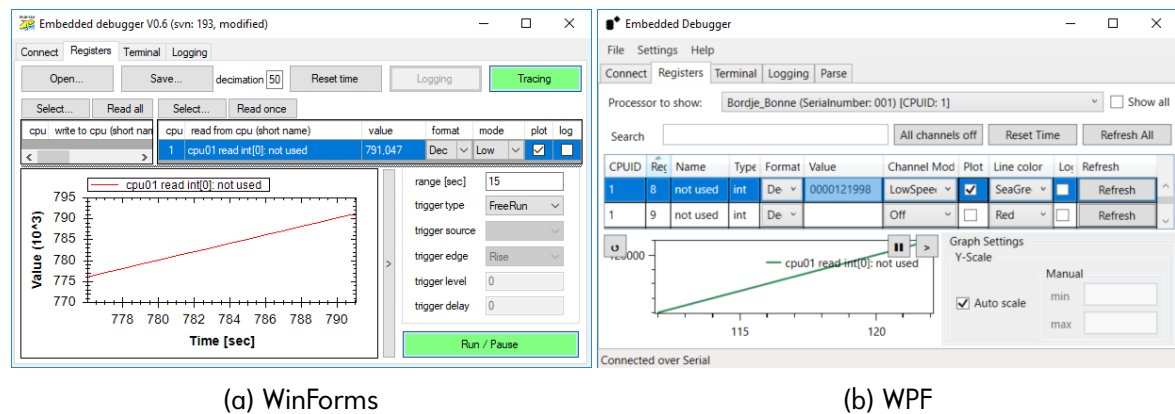


Figure 5.3: The Embedded Debugger: Registers

## Terminal

The terminal tab is the interface for the debug strings of the debug protocol. It can be used to send debug strings to the embedded platform and to display the debug strings from the embedded platform.

In the old Embedded Debugger, any input given to the terminal control would be send instantly to the embedded platform. This means that if there was any mistake in the input, this could not be fixed. In the new Embedded Debugger, the debugstring is only send when the enter key is pressed. Moreover, the input would only be shown when "local echo" was selected. When this is selected, it is quite unclear what is send to the embedded platform and what is received. The new Embedded Debugger shows which side sends the message using "To" and "From".

The new usercontrol also allows the user to zoom in, making the text larger and more readable when desired.

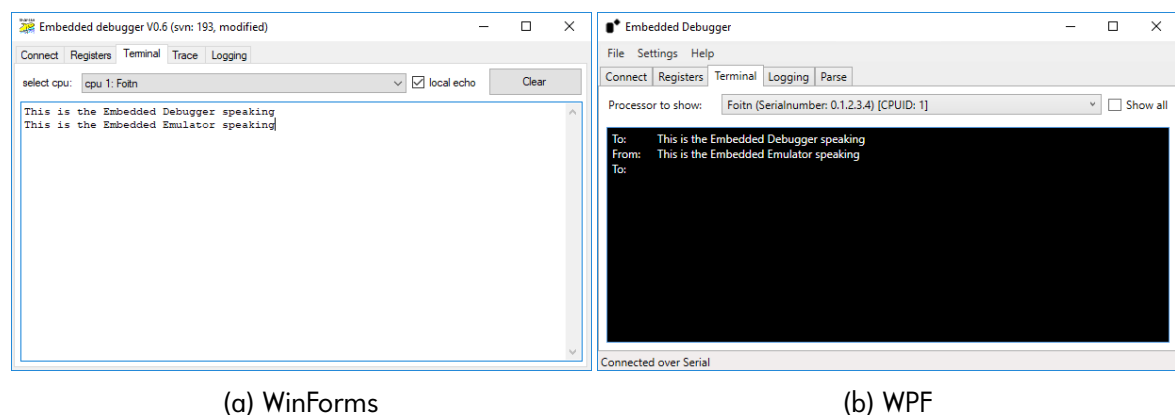


Figure 5.4: The Embedded Debugger: Terminal

## Logging

The logging tab can be used to configure logging for the Embedded Debugger. In the register tab registers can be set to log to file. Once this configuration is set and the start logging button is pressed, the Embedded Debugger starts to log these register values to the file.

The old Embedded Debugger only supported logging to a .log file, the new Embedded Debugger allows logging to a .txt file and a .csv file. A .csv file can be analyzed by external tools, to create graphs, calculate means, maximums, minimums, etc. The Embedded Debugger allows for the user to set the sort of timestamp, absolute, relative (from the moment logging was started) or none. This allows plotting against time. Moreover the new Embedded Debugger allows the user to define which character(s) is to be used for separating values. This is useful since some applications, such as Microsoft Excel, interpret a comma as a decimal sign when region is set to e.g. dutch or german. When using the semicolon, this is not an issue at all.

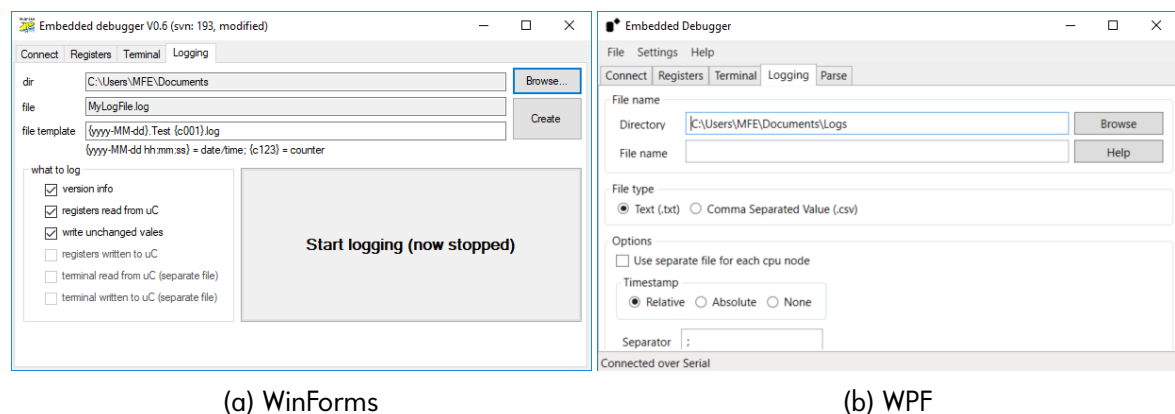
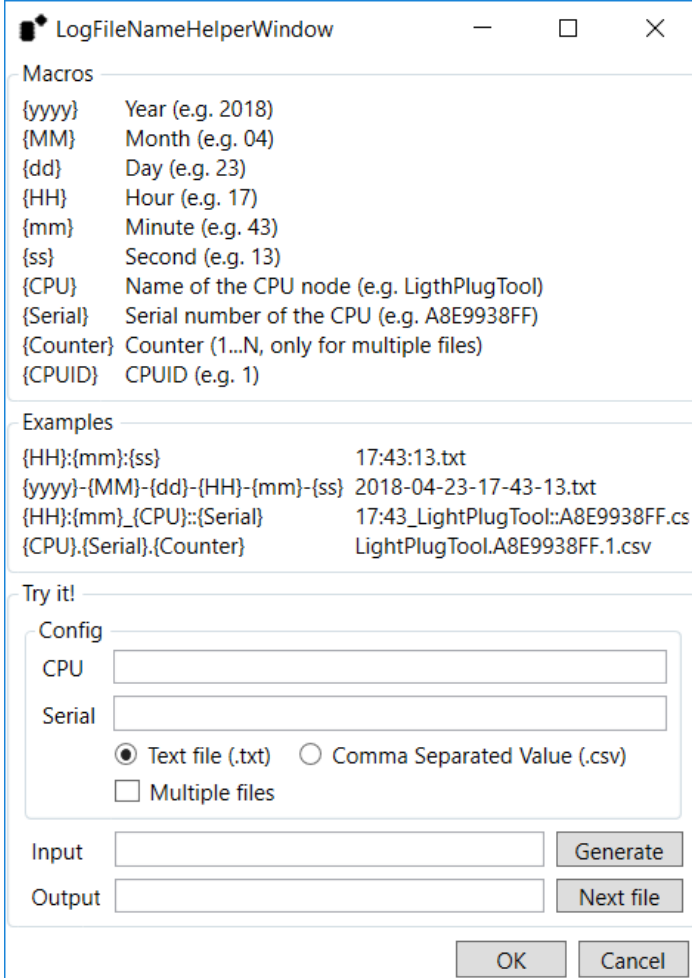


Figure 5.5: The Embedded Debugger: Logging

As can be seen in figure 5.5a there is a file template field. This can be used to give parameters for the filename. Thereby allowing the user to use the same template for each different logging session, while generating a different file every time. The new Embedded Debugger allows for a lot more template features. Because this did not quite fit in the logging window, this has been moved to a separate window. This window can be opened by clicking the "Help" button, of which a screenshot can be found in figure 5.6. This window allow a user to try the template and view what the output would be during logging.



The screenshot shows a Windows-style dialog box titled "LogFileNameHelperWindow". It contains three main sections: "Macros", "Examples", and "Try it!".

**Macros:** A list of placeholders and their corresponding values:

- {yyyy} Year (e.g. 2018)
- {MM} Month (e.g. 04)
- {dd} Day (e.g. 23)
- {HH} Hour (e.g. 17)
- {mm} Minute (e.g. 43)
- {ss} Second (e.g. 13)
- {CPU} Name of the CPU node (e.g. LigthPlugTool)
- {Serial} Serial number of the CPU (e.g. A8E9938FF)
- {Counter} Counter (1...N, only for multiple files)
- {CUID} CPUID (e.g. 1)

**Examples:** A list of example file names generated using the macros:

- {HH}:{mm}:{ss} 17:43:13.txt
- {yyyy}-{MM}-{dd}-{HH}-{mm}-{ss} 2018-04-23-17-43-13.txt
- {HH}:{mm}\_{CPU}::{Serial} 17:43\_LightPlugTool::A8E9938FF.cs
- {CPU}::{Serial}::{Counter} LightPlugTool.A8E9938FF.1.csv

**Try it!:** A section for testing the macro configuration. It includes a "Config" group box with the following controls:

- CPU: A text input field.
- Serial: A text input field.
- File format: Two radio buttons, "Text file (.txt)" (selected) and "Comma Separated Value (.csv)".
- Multiple files: A checkbox.

Below the config group box are two more text input fields labeled "Input" and "Output", each followed by a button: "Generate" for the Input field and "Next file" for the Output field. At the bottom right are "OK" and "Cancel" buttons.

Figure 5.6: Log file name helper window

## Parsing

The parsing tab can be used to parse an .ELF or .ALF file to a .XML file. This .XML file can be used for future embedded platforms when these connect. Since this feature was not present in the old Embedded Debugger, a screenshot can not be provided. More about the parsing can be found section 4.2.

This usercontrol allows a user to load an .ELF or .AXF file, which is parsed and loaded into the datagrid. The CPU name, ID, connector and version can be set here, which will be placed in the generated XML. The DataGrid allows a user to decide which of the registers should be shown when the embedded platform connects. Since the parsing can take a while for larger .ELF or .AXF files, a progressbar has been added, to indicate the progress. For loading and generating XML another progressbar has been added. This usercontrol allows a user to generate XML, but also edit XML by reloading a file from system.

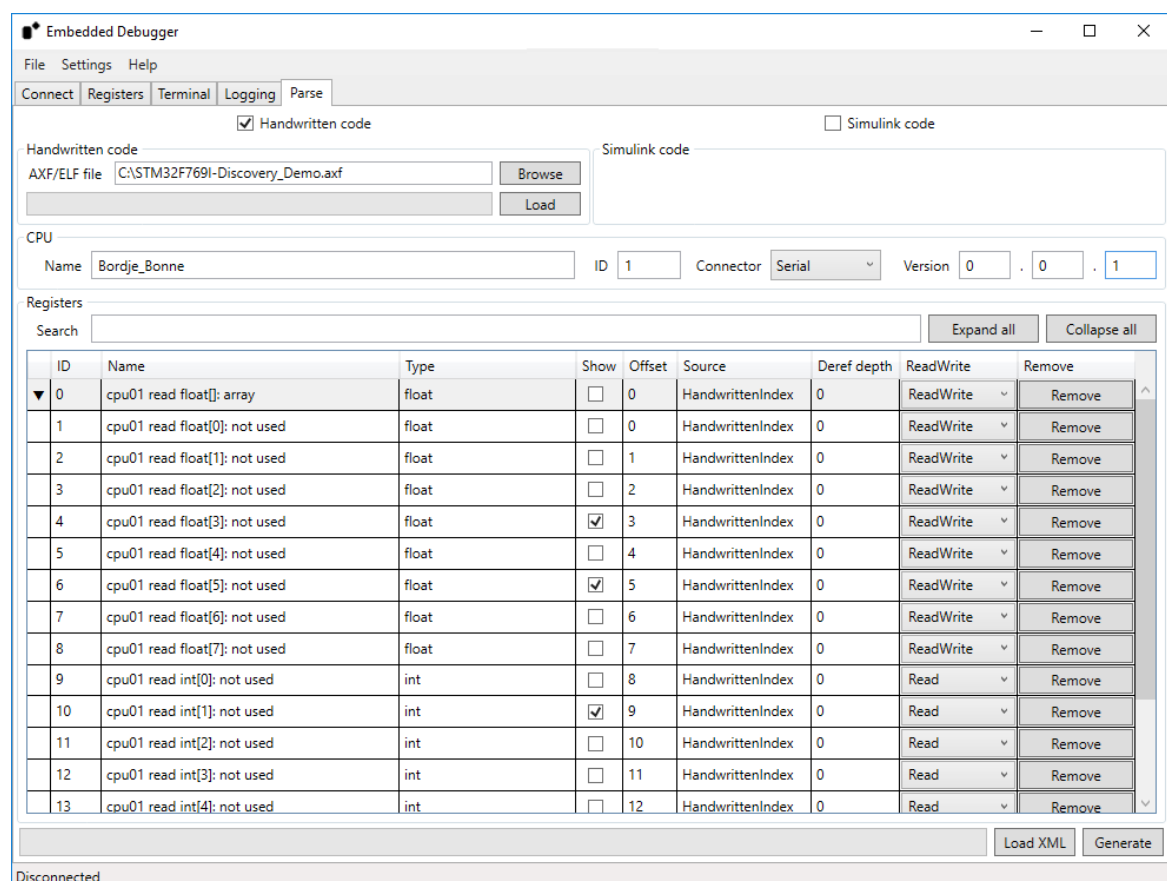


Figure 5.7: The Embedded Debugger: Parsing

Whenever a new register should be added, this would normally have to be done in the .XML file. To help users avoid this, the EditRegisterWindow has been implemented, of which a screenshot can be found in section 5.1. This allows the user to set all required information about this register, even letting him choose the parent register, whenever required. This edit button opens a new window, which shows all Registers, from which a Register can be chosen as parent.

The screenshot shows a dialog box titled "Edit Register" with a red close button in the top right corner. The dialog contains the following fields and controls:

- Name:** A text box containing "SomeNewRegister".
- Type:** A dropdown menu showing "Int".
- Size:** A text box containing "4" with up and down arrow buttons.
- Show:** A checked checkbox.
- Offset:** A text box containing "1073875626" with up and down arrow buttons, followed by an unchecked checkbox and the label "Hex".
- Source:** A dropdown menu showing "ElfParsed".
- DerefDepth:** A text box containing "0" with up and down arrow buttons.
- ReadWrite:** A dropdown menu showing "ReadWrite".
- Parent:** A text box containing "0. SomeParentRegister" with an edit icon (pencil) to its right.
- Buttons:** "OK" and "Cancel" buttons at the bottom.

Figure 5.8: The Embedded Debugger: Parsing

## 5.2: Embedded Emulator

The Embedded Emulator, which is described in section 3.1, allowed rapid development of the Embedded Debugger. This mirror of the Embedded Debugger emulates an embedded platform without requiring any hardware.

The final application allows for connection over the generic connectors, as these have been designed in the Embedded Debugger. Two of the three assemblies of the Embedded Debugger have been included in the Embedded Emulator. These assemblies are the Connectors and the DebugProtocol. This makes sure that the MessageCodec and the Connectors are available without redesigning these.

The Embedded Emulator can auto-respond to messages, but can also be set to only send ACK messages. The user can send the Version, Info and Configuration using the dedicated buttons.

The Embedded Emulator can also be used to simulated multiple nodes. This by sending the same configuration and version, but using a different  $\mu$ C-ID.

The GUI contains three fields:

- Boolean, the red/green button
- Int
- String

These fields can be manipulated from the Embedded Debugger, to test if the writing or reading works properly.

A screenshot of the Embedded Emulator can be found in figure 5.9.

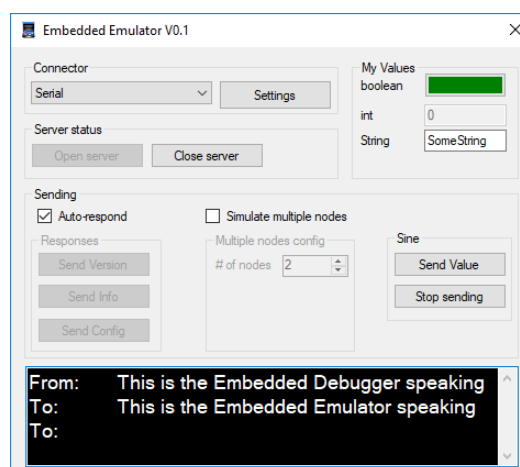


Figure 5.9: The Embedded Emulator

## Chapter 6: Conclusions

This chapter looks back at the goals which were set in section 2.4 and describes whether these goals have been achieved and why.

### 6.1: Debug the current debugger, or make a new design

As described in section 2.6, the old Embedded Debugger had a couple of issues that were difficult to work around. Therefore the decision was made to redesign the Embedded Debugger and keep the DebugProtocol as is. This has been described in chapter 3.

The new design contains almost all of the features which the old Embedded Debugger contained. Only part that was not implemented is the entire plotting environment. The plotting environment of the old Embedded Debugger also had some capability of triggering on certain events. For example, whenever a Debug channel got above a certain threshold, the plotting would start or stop. This has not been implemented, since users reported not to use this feature often or at all.

As a result, this project goal has been achieved, because a redesign has been made, which can do all the tasks of the old Embedded Debugger and more.

### 6.2: Implement a serial port interface

Since the implementation of a serial port interface was one of the main project goals, this was the first generic connector that was implemented. Multiple generic connectors have been implemented, to allow the tool to be more versatile. This means that this goal was achieved early on in the project, while creating the Embedded Emulator, which is described in section 3.1.

### 6.3: Create the Embedded Emulator

As has been described in section 3.1 and section 5.2, the Embedded Emulator was created in the beginning of the project. This allows for rapid testing and development of the new Embedded Debugger, whereby ensuring the same communication protocol as the old Embedded Debugger. This project goal has been achieved.

---

## **6.4: Send embedded configuration from the embedded platform to the Embedded Debugger**

This project goals has been achieved, as is described in section 4.1. During the project, another project of DEMCON used the Embedded Debugger and the decision was made to implement this feature in both projects. The original Embedded Debugger was already configured to received the Embedded Configuration, though this was not over the Debug Protocol. After a couple of iterations, a properly working implementation of the Embedded Configuration over the Debug Protocol was found and implemented in both projects. This concludes that this project goals has successfully been resolved.

## **6.5: Create an RPC interface**

As described in section 4.4, the RPC interface has successfully been implemented in the Embedded Debugger. Thereby it allows another program, independent of programming language to interface with the Embedded Debugger, and manipulate the embedded platform. This project goals has successfully been achieved.

## **6.6: Generate embedded configuration from compiler output**

By using the "fromelf" application from the Keil IDE, this project goal was resolved. This application parses a .AXF or .ELF file and outputs the register names, their memory addresses, types and sizes. All of this is described in section 4.2. This project goals has been successfully achieved.

## **6.7: Add a trace window to the GUI, which acts as a logging mechanism, containing features like logging levels**

This trace window is yet to be implemented. This has been on the backlog, because this feature was a nice to have. Even though it has not yet been implemented, a proposal has been made to the person who requested the feature. This proposal includes an addition to the DebugProtocol and a terminal like user interface for the Embedded Debugger. Therefore this project goals has not been achieved, but has been set in motion.

---



## **6.8: Update the debug protocol, to allow a maximum of 256 debug channels**

As explained in section 3.3, the DebugProtocol allows for up to 16 debugchannels, which has to be to maximum of 256 channels. To allow this, the DebugProtocol has to be updated. This is a quite drastic change, because it requires the Embedded Debugger to be able to handle different version of the DebugProtocol. To implement this in a generic way, was not implemented. The Version 1.0 of the DebugProtocol has been designed, therefore this project goal has been achieved.

## **6.9: Create an interface to other languages**

Since the RPC interface, which is described in section 4.4, worked very well, this feature has been shut down. If any other programming language is desired to manipulate an embedded platform, either the RPC library can be used or a language specific library is to be written. The RPC library is already language independent and works very well. For this reason, this project goal has not been achieved, since the feature was shut down.

---

## Chapter 7: Recommendations

This chapter describes the recommendations for parts of the Embedded Debugger for a follow up project. Since not all of the features have been implemented (yet), these recommendations can be seen as guidelines for another student or colleague.

### 7.1: Update the debug protocol

Version 1.0 of the DebugProtocol has not been implemented, due to a limited amount of time remaining for this large task. Therefore it is recommended for a next student/colleague to implement this. Most of the information about this is described in section 3.3. The entire implementation can be found in [??](#). It is highly recommended to implement this V1.0 in a generic way, thereby allowing easy upgrading to even a newer version in the future.

### 7.2: Simulink

As described in section 4.3, the Simulink parser has not been implemented. Some research has been conducted in this, but this has proven not to be useful. Therefore it is recommended to implement as correct parser. This parser should preferably compile the C API and request all of the memory addresses plus names. Afterwards, these memory addresses could be compared with the registers retrieved from the elf parsed Embedded Configuration. Then the names of these registers can be set to the once retrieved from the Simulink API. This allows for the engineer that created the Simulink code to see the same names that have been put into Simulink in the Embedded Debugger.

### 7.3: Trace window

As described in section 6.7, the trace window has not been implemented. The DebugProtocol V1.0 does allow for trace messages to be send with a log level. This trace message could be displayed in the trace window in the user interface. This trace window can be based upon the existing terminal window, where only the "From" and "To" can be replace by the timestamp and loglevel. Moreover this trace window could allow the user to turn on/off a specific log level. Thereby e.g. only displaying error logs.

---

## 7.4: Embedded target

This project has been limited to the C#/.NET application of the Embedded Debugger. This means that none of the code of the target side has been changed. In order for an embedded platform to utilize all of the new features, this target side of the Embedded Debugger should be updated. This includes features like:

- Version 1.0 of the DebugProtocol
- Embedded configuration over the DebugProtocol
- Using the trace window

Furthermore it is advised to set up this updated target side in such a way that it is generic. This is mostly aimed so that the target side can be implemented on any sort of microcontroller or embedded platform. During this internship, an obstacle was that the current target side was written primarily for STM32 boards. This made it really difficult to get it to properly work with a simple platform, for example an Arduino. In the most ideal case, the Embedded Debugger (target side) becomes so generic, that only one .h file is to be included. Maybe create an object and setting the correct USART pins or registers, but no further configuration or device specific code would be required. Though since no research on this has been conducted during this internship, there are no recommendations on how to actually implement this.

---

---

## References

- Cook Computing. (2011). *Xml-rpc.net*. Retrieved 2018-05-23, from <http://xml-rpc.net/>
- Foitn. (2016). *Foitn.com*. Retrieved 2017-02-07, from <https://foitn.com/LaTeX>
- Null-modem emulator (com0com) - virtual serial port driver for windows*. (n.d.). Retrieved 2018-05-25, from <http://com0com.sourceforge.net>
- Piper, P. (2016). *Objectlistview*. Retrieved 2018-06-06, from <http://objectlistview.sourceforge.net/cs/index.html>
-