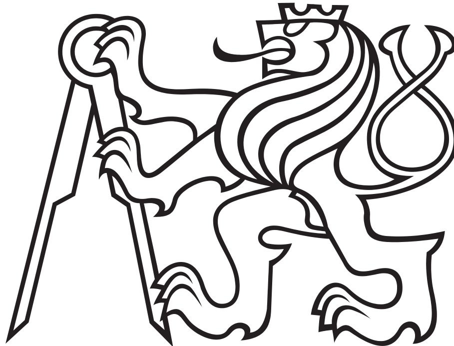
Czech Technical University in Prague

Faculty of Electrical Engineering

MASTER'S THESIS



Andrey Albershteyn

Processor-in-the-Loop Development System for e200 Core Microcontrollers

Department of Control Engineering

Supervisor of the Master's thesis: Ing. Michal Sojka, Ph.D.

Study programme: Cybernetics and Robotics

Specialization: Cybernetics and Robotics

Prague 2018

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Albershteyn Andrey** Personal ID number: **420194**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

Processor-in-the-loop Development System for e200 Core Microcontrollers

Master’s thesis title in Czech:

Vývojový systém pro processor-in-the-loop simulace s mikrokontrolerem řady e200

Guidelines:

1. Make yourself familiar with NXP MPC5748 hardware platform and the Rapid prototyping platform (RPP) project developed earlier at the university.
2. Design and implement the PIL system for MPC5748 MCU based on Matlab Simulink with (roughly) the following features:
- Layered software architecture: HAL-DRV-RTOS-API-TLC
- Basic HW support: GPIO, SPI, SCI, LIN, ADC, FlexRay, CAN, PWM, (ENET)
- Consider AUTOSAR OS as an RTOS
- Single rate model execution with the granularity of 500 ?s
- Basic diagnostic function: Overrun
- External mode support (SCI or ENET interface)
- Integration with Matlab R2016b (64 bit) for Windows
3. Test your system by developing demonstration applications (Simulink models) for each of the peripheral and a more complex application involving multiple peripherals.
4. Document the resulting system both from user and developer point of view.

Bibliography / sources:

[1] Jenkins, Horn, Sojka: Simulink code generation target for Texas Instruments TMS570 platform, Technical report, ČVUT v Praze, 2015.
[2] M. DiNatale: An introduction to AUTOSAR (https://retis.sssup.it/sites/default/files/lesson19_autosar.pdf)

Name and workplace of master’s thesis supervisor:

Ing. Michal Sojka, Ph.D., Department of Control Engineering, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **31.01.2018** Deadline for master's thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

Ing. Michal Sojka, Ph.D. Supervisor’s signature	prof. Ing. Michael Šebek, DrSc. Head of department’s signature	prof. Ing. Pavel Ripka, CSc. Dean’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgments
I would like to express my gratitude to my supervisor, Ing. Michal Sojka, Ph.D., for his supporting and given advice. My thanks also go to Ing. Pavel Kučera, Ph.D. for his excellent management of this project.

I hereby declare that I have completed this thesis with the topic "Processor-in-the-Loop Development System for e200 Core Microcontrollers" independently and that I have included a full list of used references. I have no objection to the usage of this work in compliance with the act §60 Zakon e.121/2000 Sb. (copyright law).

In .... date .... signature of the author

Abstract: The aim of this thesis is to develop a processor-in-the-loop system based on e200 core microcontroller from NXP MPC5748G [36]. The use of visual programming tools such as Simulink [26] allows to utilize Model-Based Design development process and significantly speed up the integration and validation of the software pieces. Holding this purpose Eaton corporation required a tool to create a processor-in-the-loop system comprised mainly of Simulink, Autosar OS, and e200 based microcontroller. We designed a basic prototype system made up of the Autosar operating system and Simulink library. We described and documented the structure of the designed system and the way it is implemented. Furthermore, we provided detailed instructions of the process of configuration of the new components and the process of adding new functionality to already existing Simulink library. We developed a testing application to simplify verification process and created four demonstration models to provide a starting point in the development of the more complex models.
Keywords: Model-based design, Processor in the loop, Matlab/Simulink, Autosar

Abstrakt: Hlavní cíl této diplomové práce je vytvořit processor-in-the-loop systém založený na mikrokontroléru od firmy NXP s jádrem e200 – MPC5748G. Použití vizuálních programovacích nástrojů, například Simulink, umožňuje využití principu Model-Based Design během vyvíjení nového produktu a významně zrychluje proces validace a verifikace vytvořených algoritmů. Pro vývoj jednoho ze svých produktů firma Eaton potřebovala vytvořit nástroj pro processor-in-the-loop testování založeného na Simulink, Autosar OS a MPC5748G. Tato práce se zabývá vývojem a návrhem zmíněného systému. Detailně jsme popsali a zdokumentovali celou strukturu a realizace. Dál, vytvořili jsme návody pro přidání dalších funkce do Simulink knihovny a pro konfigurace nových hardwarových jednotek. Vyvinuli jsme testovací aplikace pro ověřování existujících funkce operačního systému a čtyři demonstrační Simulink modely, které slouží ukázkou implementovaných charakteristik systému a můžou se použít pro vývoj složitějších modelů.
Klíčová slova: Model-based design, Processor in the loop, Matlab/Simulink, Autosar

Contents

1 Introduction **5**

1.1 Objectives 5

1.2 Outline 6

2 Concepts and Technologies **7**

2.1 Model-Based Design 7

2.2 Brief description of Processor-in-the-loop simulation 7

2.3 Autosar 8

2.3.1 Autosar configuration 9

2.4 Autosar MCAL 9

2.5 Matlab & Simulink 9

3 PIL System Design **11**

3.1 System Architecture 11

3.2 Operating System Configuration 13

3.2.1 Autosar OS Configuration 13

3.2.2 MCAL modules Configuration 13

3.3 The use of RTE 14

3.4 User Application 15

3.4.1 Simulink Skeleton of the application 16

3.4.2 Architecture 16

3.5 Simulink Library 18

3.5.1 Block internal structure 18

3.6 PIL simulation flow 19

3.7 Run-time Model behavior 19

4 Environment Set-up **21**

4.1 Hardware Set-up 21

4.2 Software Tools 22

4.2.1 S32 Design Studio IDE for Power Architecture based MCUs . . . 22

4.2.2 Wind River Diab Compiler 5.9.6 22

4.2.3 NXP Software Packages 23

4.2.4 EB Tresos Studio . . . 23
4.2.5 Cygwin & GNU Make & Putty . . . 23
4.2.6 Matlab + Simulink . . . 23

5 Implementation Details **25**
5.1 Folders and Files description . . . 25
5.1.1 Folders structure . . . 25
5.1.2 Makefiles . . . 26
5.1.3 Compilation & Execution . . . 27
5.1.4 Simulink Folder Description . . . 29
5.2 User Application . . . 32
5.2.1 ert_main.c . . . 32
5.2.2 model.c . . . 34
5.3 OS & MCAL Configuration . . . 34

6 User Manuals **37**
6.1 Configure Makefile.config . . . 37
6.2 The use of Testing application . . . 37
6.3 Assemble and install Simulink library . . . 38
6.4 Create new model . . . 39
6.5 Adding new functionality . . . 39
6.5.1 Configure hardware . . . 40
6.5.2 Creating Interface to Simulink . . . 40

7 Evaluation **43**
7.1 Demonstration of IO blocks . . . 43
7.2 Timing and Multirate Simulink model . . . 44
7.3 Granularity . . . 45
7.4 SCI Echo Server . . . 46
7.5 Overrun demonstration . . . 46

8 Conclusion **47**

List of Figures **53**

Acronyms **55**

CD content **57**

Appendix A **59**

Appendix B **61**

Appendix C **63**

Appendix D **65**

1. Introduction

Development of a new product with strict requirements on safety and reliability is a long and challenging task. Making this process quick and effortfully optimal is financially beneficial. One of the techniques applied in engineering is Model-Based Design (see 2.1). It is a method used to deal with non-trivial model design problems [7] via applying visual and mathematical tools. The use of graphical diagramming tools dramatically speeds up development and verification of the existing systems and new ideas [24].

One of a currently developing product in the Eaton Corporation is hydraulically-operated electronically controlled differential. Its primary goal is to provide superior vehicle cross-country mobility with a rapid response for stability and traction control. Electronic control eliminates drivers interface but requires stable controlling algorithms and communication with other components of the car [5].

Eaton Corporation is an international power management company with many consumers products and solution for a variety of industrial sectors. One of the first Eaton domain of interest is automotive. Nearly every automotive, vehicle and engine manufacturer in the world make use of Eaton's high-quality products [4]. Eaton offers a long list of traction modifying devices for many applications ranging from personal cars to military machines.

One of the well-known tools for Model-Based Design of embedded software is Simulink (see 2.5). For our purposes, there already exists a relevant Model-Based Design toolbox[1] oriented toward the development of the software compatible with the microcontroller used in this project. This toolbox allows creating models which can interact and control low-level peripherals such as GPIO pins, ADC, PWM, SCI, CAN, etc. However, this toolbox is not compatible with Autosar which is one of the main requirements of this work.

1.1 Objectives

The objective of this project is to create a prototype of the system with similar functionality for the Processor-in-the-loop simulation. The system should be based on Autosar operating system and utilize Simulink for Model-Based Design. The integration with Simulink includes a library with basic hardware blocks such as digital input/output, SCI, PWM, etc. In terms of timing, the generated application have to be precise with granularity down to $500\mu S$. As a tested model can be quite complicated, the time required to calculate one step of the control loop can not fit within the execution step

[1] `https://www.nxp.com/support/developer-resources/run-time-software/automotive-software-and-tools/model-based-design-toolbox:MC_TOOLBOX`

of the processor. The system should have functionality for execution time overrun diagnostic of the model's control loop (*overrun* later in the text). Next, to make it even more flexible, we should implement external mode support to allow run-time communication between Simulink and hardware target. This system needs to be integrated with Matlab R2016R running under MS Windows operating system. The last objective is to document every part of the project for the further development.

1.2 Outline

The thesis is divided into eight chapters. After this introduction and problem description, Chapter 2 – Concepts and Technologies explains the essential core of this work and gives a description of primary tools and frameworks. Chapter 3 surveys the design of the suggested system architecture. It should create a background of the way system is constructed and function. The chapter also deals with explanation of the particular parts of the system and how they work and should be configured. The next Chapter 4 lists all the software tools recommended for work with this project and describes hardware configuration. Chapter 5 extends Chapter 3 with a real implementation of the designed architecture. It contains most of the practical details which should help during project investigation. In Chapter 6 we provide several instructions on how to work and extend this system. Chapter 7 contains the description of the evaluation models which were developed to demonstrate working ability and correctness of the suggested solution. Finally, in Chapter 8 we recapitulate major results of the project and briefly describe future improvments.

2. Concepts and Technologies

In this chapter, we will give a relevant background on the main concepts applied in this work. Then we will describe the key technologies used in the development.

2.1 Model-Based Design

Model-Based Design (MBD) is a methodology broadly used in motion control, goods industry, aerospace, and automotive fields [7]. It is the mathematical and visual approach used throughout the whole process of the product development. It dramatically speeds up prototyping, testing, and verification of software development [7] [24] [30].

Model-Based Design is comprised of the four main phases – modeling, analyzing, simulating and deployment. In this work, we are mainly focused on the last step – deployment. In terms of MBD, it means the execution of a developed model on real hardware via code auto-generation. This process helps to catch problems which were missed or can not be found during a software simulation [7].

The Model-Based Design is mostly applied via utilization of graphical tools as it is much quicker to develop a model and find mistakes in the algorithm than in text-based modelling tools [29]. In this project, we used Matlab + Simulink modeling environment which is described in section 2.5

2.2 Brief description of Processor-in-the-loop simulation

Processor-in-the-loop simulation is the process of executing developed model on real hardware. This methodology helps to expose problems at the beginning of the product cycle, in the design phase. The hardware tests help to detect problems which can be missed during the software model simulation. For example, we can carry out overrun diagnostics to determine if designed control loop fits within execution step of the processor [25].

One of the most popular and known tool for Model-Based Design (section 2.1) with PIL verification is Simulink [26]. PIL simulation functionality provided by Simulink lets us to generate, compile and execute tests on target hardware automatically. The results of the execution can be then analyzed in the powerful Matlab environment [25].

2.3 Autosar

Autosar (AUTomotive Open System ARchitecture) it is an open and standardizied software architecture for automotive electronic control units established by the partnership of many large companies such as BMW, Bosch, Ford, Toyota, etc. [1] [9]. Its purpose is to manage growing complexity of automotive software through reuse and exchangeability of the software modules between different projects and companies [1].

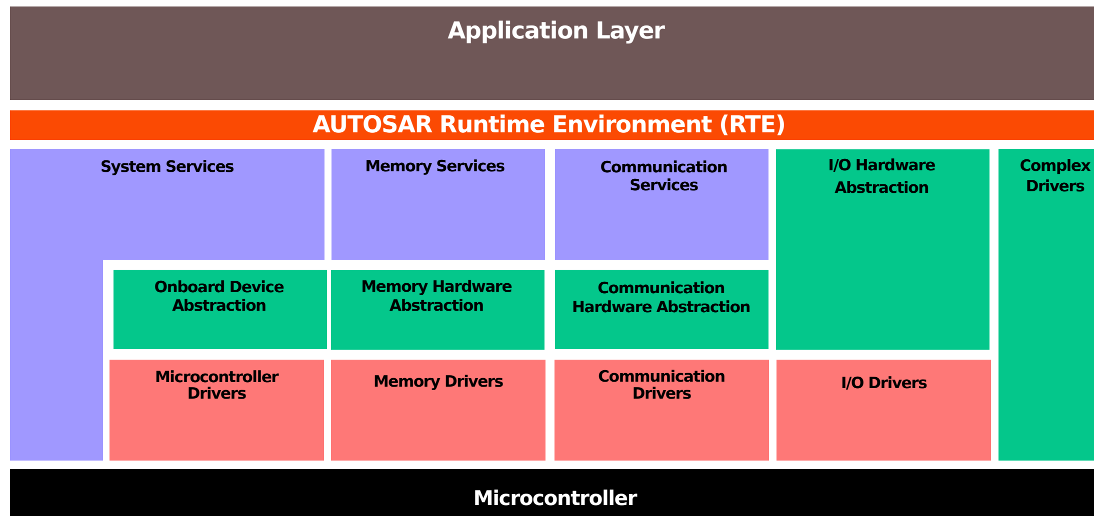


Figure 2.1: Autosar OS Layers Structure

The primary goal of Autosar is to make hardware and software highly independent of each other. The layered structure depicted in figure 2.1 is the main advantage of Autosar in comparison to the custom solutions. It allows customers to reuse software in numerous project without modifying it. The strict definition of application interfaces and so-called Basic Software Modules (BSW) provides a unified interface on all of the hardware platforms [1].

Autosar Specification defines what should be implemented and how it should behave (in terms of software interface). Moreover, specification formulates the guidelines on how it should be implemented (e.g., file structure) but each implementation can differ. Therefore, Autosar specification is an interfaces description and not an implementation instructions.

In this project, we used Autosar implementation provided by NXP. This realization do not contains the full complect of modules, only the System Services and Microcontroller Abstraction layer (the red blocks and the purple one in figure 2.1). Its documentation rigorously describes every deviation from original specification.

Autosar in its nature is moduled system [2] [11]. Staying with this ideology most of the companies and OEMs distribute their software in the separate pieces. For example, in our case NXP provides two software packages Autosar OS and Autosar MCAL [33].

Operating System package provides System services only; the purple blocks in figure 2.1. Whereas, MCAL/MAL package provides modules for microcontroller abstraction; the red blocks in the figure 2.1.

2.3.1 Autosar configuration

The every component of the system as OS as particular MCAL module can be divided into two parts – implementation and configuration. The realization of a module is a static set of files which do not require any modification. Whereas configuration is distributed in the form of templates that are later parsed by special tools to generate modules' configuration.

Therefore, the assembling process of the system is divided into two parts. Collecting all the necessary realization related source code and header files and then generating configuration of the whole operating system via special tool (see section 4.2.4) [39].

2.4 Autosar MCAL

MCAL/MAL stands for MicroController Abstraction Layer. It is the first layer of abstraction in Autosar operating system. MCAL/MAL implements interface for on-chip MCU peripheral modules, and external devices mapped to the memory. This layer creates an abstraction for Basic Software Modules and a few OS services. It makes the top and middle layers first abstraction of the underlying hardware. In figure 2.1 MCAL layer is shown as the red blocks and consist of I/O Drivers, Communication Drivers, Memory Drivers and Microcontroller Drivers. As a more concrete example this layer contains drivers for LIN, GPIO, ADC, etc.

2.5 Matlab & Simulink

Matlab and Simulink are products of the MathWorks Inc. It is software with many different extensions and worldwide community. These tools create an environment for mathematical model development, analysis, and verification. It is widely used in most of the engineering fields such as automotive, aerospace, signal processing, communication, etc.

Matlab is computing environment with its proprietary programming language. It is intended primarily for numerical computation and matrix manipulations. But it is highly extensible with many toolboxes and interfaces to the other programming languages such as C/C++, Java, Python, Fortran, and C#.

Simulink is a graphical programming environment for model development. The development process is based on manipulation with graphical blocks. It allows to prototype and test new ideas rapidly while decrease chance of making the mistakes as in text-based tools. Moreover, it is tightly integrated with Matlab which offers the broad range of possibilities for further analysis.

The flexibility of the Simulink allows to modify and extend created models quickly. With the addition of Simulink, Matlab and Embedded Coder it became a perfect tool for Model-Based Design for embedded devices. These extensions offer powerful functionality for code generation for real-time and nonreal-time applications. The Matlab coder is a tool for generating C and C++ code for a variety of hardware platforms. It generates readable and portable source code which can be then integrated into a custom project. Embedded Coder is enhancement above the Matlab coder for producing target-specific optimization, code customizations and Processor-in-the-loop simulations [18] [15]. The Simulink coder allow to generates C and C++ code from the Simulink charts and diagrams [27].

3. PIL System Design

In this Chapter, we will give the description of the software architecture developed during this project. The system can be divided into two main parts. First one is low-level embedded software made up of Operating System, MCAL modules, Autosar configuration and user application. The second one is Simulink related resources. It consists of the scripts exerting process of model code generation, specific target configuration rules of a model and blocks' templates composing Simulink library.

Firstly, we will give a brief description of the system architecture as a total; which components made up the system and how they interact with each other. Then we will describe each of them in more details.

3.1 System Architecture

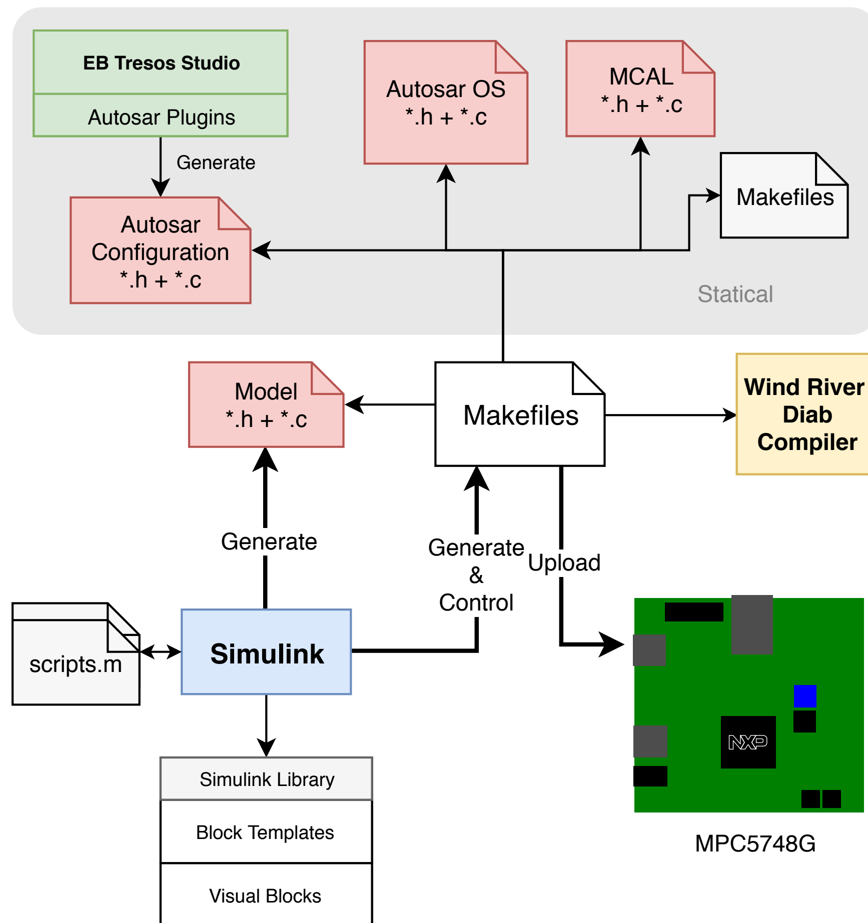


Figure 3.1: System Architecture

This section describes the architecture of the project and the way its components interact with each other. Figure 3.1 shows a schematic diagram of the system arrangement and overview of the processes happening in the system. The chart can be roughly divided into two parts. One is statical and is not affected by Simulink (highlighted with a grey rectangle). The second one is partially generated and mostly controlled by Simulink.

Let's focus on the first part. By statical, we mean that these components are not affected or modified during model development. The red blocks in the grey area named "Autosar OS" and "MCAL" are the source code of Autosar Operating System and MCAL modules accordingly. These sources are provided by NXP (for more information see section 4.2.3).

The left-side red block in the grey area is a configuration of Autosar OS and MCAL modules. It is generated via EB Tresos Studio (see 4.2.4) indicated as a green box. The IDE outputs files with many parameters used by Autosar components to carry out hardware initialization, to define which API is required and to create interface entities (e.g., channels, ports). The last light-grey box on the right symbolize the Makefiles which defines the key variables for the complation process.

The core component of the second part is Simulink. Simulink has a few purposes. Firstly, it is used for model development. To create a model which utilize microcontroller's peripherals you need to assemble and install a custom library. The special script collects blocks found in a project folder compiles them and then assemble them into the library. Next, the library is added in the library browser and blocks can be freely used in the model.

The fact that model will be used for PIL simulation require additional manipulations with model's configuration. The Matlab environment allows to automate it via scripts. When the model is created user specify target platform which defines all required parameters such as target language, Makefile template, make utility, model solver, solver's parameters, etc.

The next purpose of the Simulink is to convert the model created by a user into the source code. To accomplish this, we use the extension named Embedded Coder. In addition to source code, it generates Makefile to compile and link all the necessary files automatically.

The model is converted into the C source code by utilization of special templates attached to every block. These templates describe how a block should be integrated with the other parts, how it should be initialized and terminated, and inner algorithm of the block [15].

To summarise, before developing a new model Autosar components should be brought to required configuration. That means that in this phase all the peripherals, channels,

system clocks, system tasks, etc. are created and configured. After that, the whole process of the code generation, compilation and uploading is controlled by Simulink. Embedded Coder generates source code of the model using built-in blocks and blocks from the custom library. The code is then compiled and uploaded to the target hardware. Finally, Simulink indicates that process is successfully finished and the model is continue executing on the board.

3.2 Operating System Configuration

The Autosar components have three different variants of configuration: *Pre-Compile*, *Post-Build*, and *Link-Time* [12]. In this project, we used two types – *PreCompile* and *PostBuild*. The first one means that all of the componenet's parameters are generated in the form of source and header C files before the compilation of the system and can not be changed in the run-time. The second one stores multiple ECU configuration in the memory areas, and the system can switch between them in run-time. Moreover, it allows to change some of the BSW properties and add other elements, but it is limited only to a specific set of parameters [40].

3.2.1 Autosar OS Configuration

The Autosar operating system allows only pre-compile time type fo the configuration parameters. It means that it is configured and scaled statically. Therefore, most of the system parameters and functions such as alarms, scheduling tables, number of tasks, stack sizes, interrupts are configured before compilation and can not be changed in a run-time.

3.2.2 MCAL modules Configuration

A microcontroller consists of many hardware components such as clocks, CPU cores, communication modules, etc. Before running any operating system or user application, these peripherals need to be configured. Autosar MCAL is microcontroller abstraction layer for these hardware components.

Even though Autosar is a highly flexible system, it is accomplished by limiting the system configuration making it less dynamic. For example, it is impossible to change pin mode from a digital output to PWM-driven one (NXP safety implementation does not support this API). The API offered by the operating system also relies on the configuration and need to be enabled or disabled depending on the project requirements. Moreover, some of the parameters are strictly limited by Autosar specification and can have only *Pre-Compile* type of configuration.

NXP's MCAL modules are distributed in the form of plug-ins for EB Tresos Studio (see 4.2.4). Plug-in folder includes templates in the form of Tresos Studio markup language. These templates are used to generate C language headers and source code files which contains the configuration of individual MCAL modules.

Plug-in folder also contains a statical implementation of MCAL module. In other words, it implements all structures and functions which provides an interface for interaction with hardware. This implementation is also need to be included in the project.

3.3 The use of RTE

Autosar is a widely used system. Nowadays most of the car parts have their built-in microcontroller for monitoring, communication, and control. Most of the OEMs do not produce the whole set of the parts required to assemble a car. They focus on one specific set of products and supply it to bigger companies [41]. It becomes important to offer compatible software for interact and control these parts. The primary purpose of Autosar is to standardize this software via abstraction layers. In Autosar ideology these applications should be placed at the application layer (top layer in figure 3.2).

AUTOSAR 4.0 (CLASSIC PLATFORM)

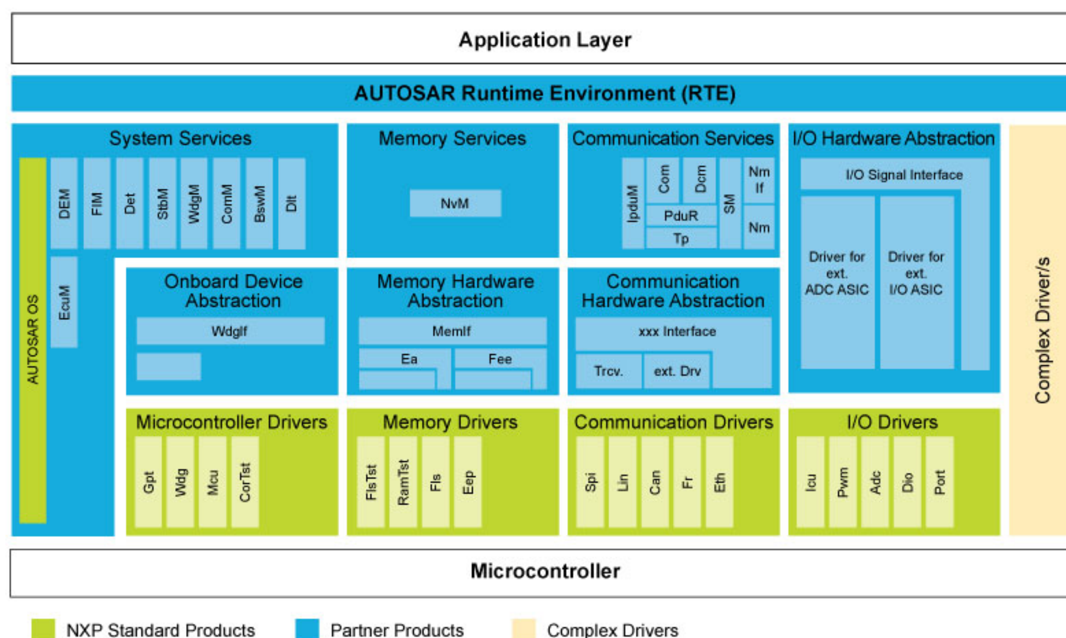


Figure 3.2: Autosar implementation provided by NXP [36]

In an Autosar system, the user application is called Software Components. These are located in the application layer of Autosar architecture. They can communicate with each other and with operating system services via Run-time Environment. RTE with other Basic Software Modules implements Virtual Functional Bus concept [3]. It is a

pivotal idea in Autosar architecture that assures standardized communication between components. That, in turn, allows OEMs to ship the same controlling application for different variation of hardware configuration.

Simulink software already has tools for validation, development, and simulation of the software components [28]. Unfortunately, implementation of the Autosar provided by NXP does not implement **I/O Hardware Abstraction** basic software component (see figure 3.2). It is an interface between microcontroller abstraction layer and RTE [3].

Even though it is not the part of the problem what we are solving, it would bring some model portability and reliability of the MathWorks tools. As a result, we couldn't use VFB for data acquisition and control over peripheral modules. Instead, we used interface provided by MCAL/MAL layer to interact with on-chip modules.

3.4 User Application

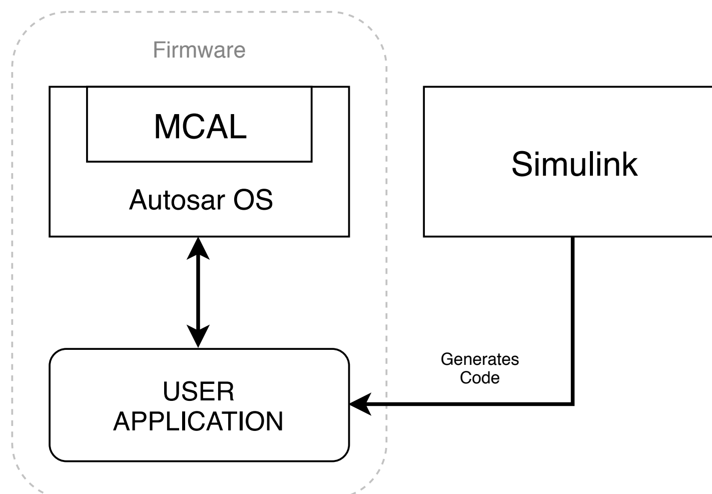


Figure 3.3: Simulink generates only user application (model's code)

The user application is meant to be a software piece which utilizes the operating system for its custom behavior. In our case, it represents software generated from a developed model. As we do not use RTE and can not utilize its interface this application can not be placed at the application layer of Autosar layer structure (see 2.1). We can assume that our application is above MCAL abstraction layer with tight integration with system services.

The structure of the application is fixed and have a skeleton which later is filled up with code generated from a control loop of the Simulink model. The skeleton already contains integration with Autosar OS and a few important MCAL modules initialization. It defines all the tasks and their inner structure.

3.4.1 Simulink Skeleton of the application

The skeleton is not a regular C language file; it is template written in special TLC language. This template is parsed by Simulink and then filled up with necessary code constructions. Code which will be placed in the template depends on many parameters defined by Simulink. For example, although template has a construction which needs to be replaced by code for external mode support if this support is disabled in model settings (in Simulink), it will not be generated.

The model's control loop is placed in a separate file completly generated by Simulink. It mainly consists of three functions which are called from the main application file. These functions are responsible for initialization, one control loop step, and termination of the model. The control loop of the model is placed under *one step* (*modelname_step(void)* in the code) function. The *initialization* (*modelname_initialize()* in the code) function contains intial source code of the blocks. Every block template describes the process of the block intialization and termination. The last function *termination* is similar to *initialization* function, but it is called at the model termination stage.

The generated control loop is made up of parts of the code corresponding to separate blocks. These code pieces interact via code structure generated on the basis of the blocks' description (e.g., number of the inputs/outputs, types of the inputs). Simulink knowns the way to handle its built-in blocks. We implemented similar templates for our custom blocks. These blocks TLC templates were based on RPP project implementation. More information about blocks implementation can be found at [10], [16], [21] and [14].

3.4.2 Architecture

The application architecture has a few primary criteria which need to be satisfied: application unable to change OS parameters, execution rate need to be precise, minimal execution rate of the model can vary, and application should detect control loop overruns.

As mentioned earlier Autosar OS is statically configured and can not be changed in run-time, the application should be structured in the way that it does not require any OS configurations. For example, we can not manipulate with task's priority or length of the alarms' cycle to obtain required task execution behavior.

Minimally required execution rate of the model is $500 \mu S$. We used it as a fundamental frequency for the whole application. It is accomplished by a hardware counter and alrarm attached to it. The alarm is system object which can be configured to activate specific task every **n** ticks. The alarm is attached to the counter which is incremented by hardware clock (FXOSC) configured to specific frequency, in our case $40 MHz$.

Between the hardware clock and the software counter there is clock STM0 which inputs is connected to FXOSC and output is configured as input for the counter. It has prescaler which decrease count frequency to $2MHz$. The alarm's period length is 1000 ticks. Therefore, every $500\mu S$ alarm activates a task which is responsible for the control of the application. This task also has the highest priority.

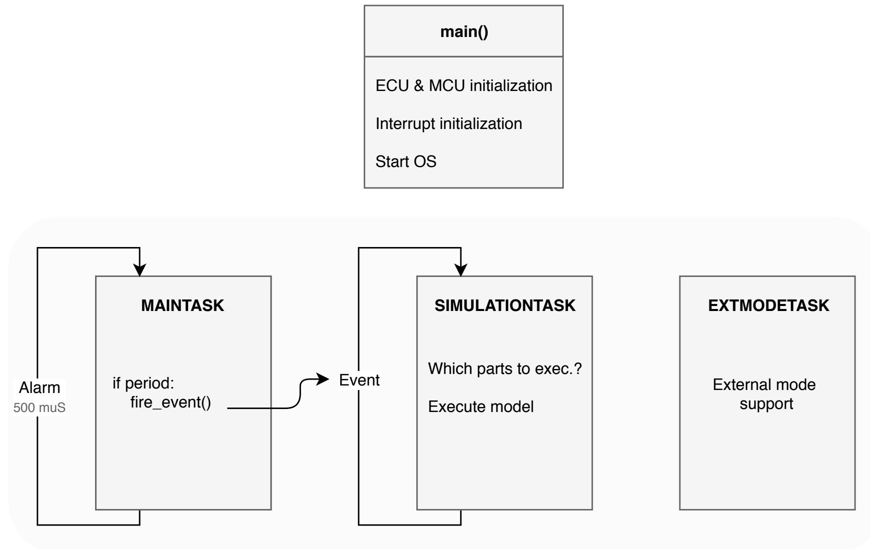


Figure 3.4: Internal structure of the user application

This periodic task (in implementation it is named MAINTASK) has two main purposes – to carry overrun diagnostics and activate model task responsible for model execution. Main and model tasks are synchronized via system events (similar to semaphores). Model task unlimitedly waits for an event to occur. Only the periodic task decide when event need to be set.

Model task need have to be executed with fundamental rate defined by Simulink. This rate is calculated as a greatest common divisor of all sample rates of all control loops presented in the model. The main periodic task, mentioned earlier, uses this rate to calculate period at which execution cycle it needs to activate the model task. For example, if the fundamental rate is 0.05 and the main task always has rate 0.0005, then, the main task will set event every 100 cycles.

When the model has multiple control loops with different sample rates they are handled as follow:

1. Simulink calculates fundamental rate – the smallest step required to all loops were in phase
2. Main periodic task executes the model task with this smallest step
3. Function generated by Simulink marks parts of the model which need to be executed in this step

4. Parts which should not be executed are skipped

In this type of architecture called singletask – multirate overrun detection reduces to control of a single task. As main periodic task has the highest priority, it can preempt execution from the model task and check if it is within time limits or not.

There also exists a third task named EXTMODETASK which is used for External Mode support. This task is filled up with code generated by Simulink Coder.

3.5 Simulink Library

We created a custom Simulink library with blocks which represent an interface to hardware peripherals. In other words, these blocks are the visual representation of the code which will be placed in the final sources.

Visual models of the blocks are attached to templates. These templates contain the description of C-code construction for block initialization, execution, and termination.

More information about the process of creating a custom library and the way libraries works can be found at [22].

3.5.1 Block internal structure

S-function is a Simulink interface which allows using custom blocks in Simulink models. In order to utilize it, we need to define a few block's attributes which will determine the way block will be function and presented in Simulink.

The particular blocks are defined by three files – visual model, Matlab executable (MEX) and TLC template. The visual model is built-in Simulink block attached to MEX file. It can have a mask manually created by a developer with port labels, parameters help, an icon on the block, etc. to make the block more recognizable. The Matlab executable file is written in a compilable language (in our case it is C) and presents executable part of the block. The Simulink calls function from MEX file to obtain block attributes such as the number of inputs, number of outputs, number of parameters, etc. The C MEX files are designed to be flexible in its implementation; it can be, for example, complex algorithm. In this project, we used it as the description of block's attributes (number of inputs/outputs/parameters). The last file is TLC template. It is code-representation of the block. These files are written in TLC language. These templates are not used in model development, only in code-generation phase. For more information about block structure see [14], [13] and [10].

3.6 PIL simulation flow

The application can be roughly divided into two parts – OS and User Application (figure 3.3). As the OS does not require any modification, Simulink will treat it as user's additional sources. We can automate the OS compilation process by modifying Makefile generated for model compilation.

Makefile is created from a template which is parsed and filled up with model's parameters. This template includes Makefiles for OS compilation. Also, these Makefiles specifies compiler, building flags, output folder, rules for other extra sources (e.g., UART), etc. In result, we get a classical C project which can be compiled even without Simulink by using building control tool (for example **make** utility; see 4.2.5).

The user application (model) is transformed into code. The source code of the blocks is based on TLC templates. Simulink uses code structures based on blocks description provided in TLC and s-function templates to interconnect blocks inside of the application.

The whole PIL simulation workflow including code-generation and compilation can be seen in figure 8.7. More information about PIL simulation can be found at [32] and [15].

3.7 Run-time Model behavior

In a run-time Simulink can connect to the target hardware via a serial link or TCP/IP. If the connection is successfully set up, the user can change blocks parameters and view dataflow of the model, like in software simulation. This is done by step-nature of the model execution. Communication between Simulink and target hardware is based on server-client architecture where target is server and Simulink is client. Unfortinately, there is no official description of the protocol used for communication.

The Mathworks, Inc. provides library written in C language which need to be included into the target's firmware. This library implements communication protocol and handling of the model modifications.

4. Environment Set-up

4.1 Hardware Set-up

The hardware set-up consist of the hardware target, Windows PC, power adapter, JTAG debugger and a few cables. First of all, we need to power up the board with 5V min. 1A power supply. This power adapter is not shipped within NXP board package.

Then, connect JTAG debugger to PC via USB cable and to board via 14-pin JTAG connector. As a debugger, we used USB MULTILINK FX Advanced Debug Probe[1] distributed by one of the NXP partner P&E Micro. The probe has two LEDs one of which is power, and the other is an indicator of target connectivity. Therefore, if your board and debug probe are correctly set up you will see both LEDs glowing. Documentation and specifications of the probe can be found at the official "Download page"[2].

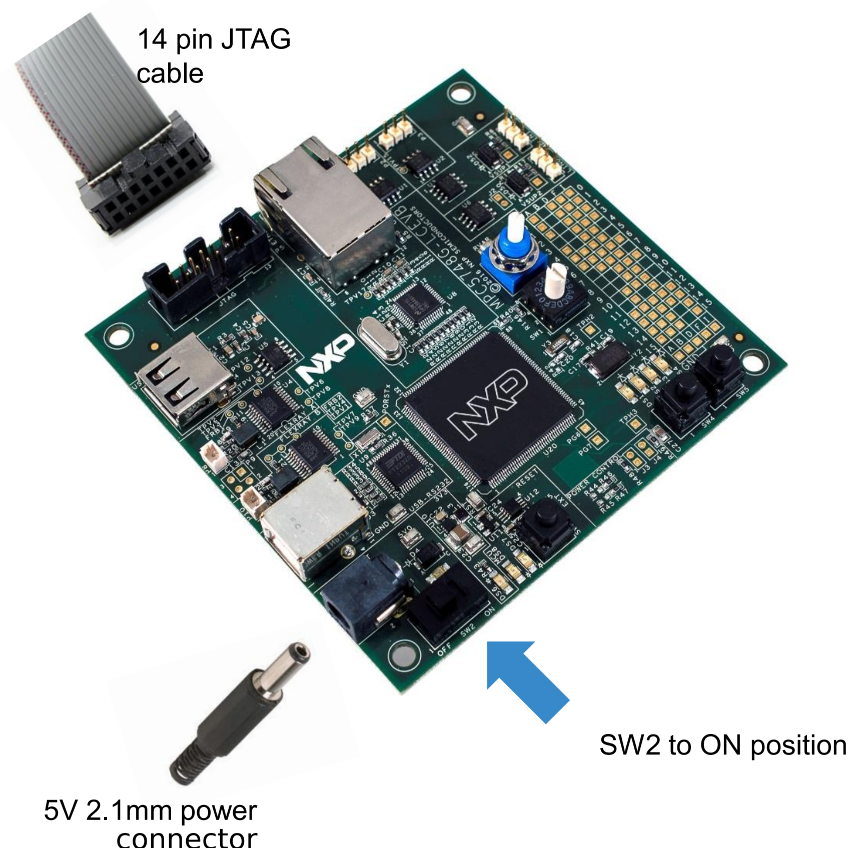


Figure 4.1: Photo of the evaluation board taken from [35]

[1] http://www.pemicro.com/products/product_viewDetails.cfm?product_id=15320143&productTab=1

[2] http://www.pemicro.com/products/product_viewDetails.cfm?product_id=15320143&productTab=3

Lastly, we need to create a serial connection between board and laptop via USB cable. Evaluation board already has the USB-RS232 serial interface, so there is no need for additional hardware. The FTDI FT2232D USB to Serial interface chip which is placed on the board will create two serial ports on your PC. Note that only one of these ports is used. To determine which one is working you can try to connect a serial monitor to one of them and reset the board. It will boot a hard-coded firmware and send a message via serial port; often it is port with the higher number.

4.2 Software Tools

This project was developed and tested on the Windows 10 machine. In this section, we will list recommended applications and tools which you will need to work with the developed project.

4.2.1 S32 Design Studio IDE for Power Architecture based MCUs

S32 Design Studio is an integrated development environment for Automotive and Ultra-Reliable MCUs. It is officially distributed by NXP. S32 Studio is based on multiplatform Eclipse open source IDE. It includes GNU toolchain with GCC Compiler 4.9 for e200 power architecture microprocessors. Also, it has support for Green Hills and Wind River Diab compilers, built-in GDB debugger with integration for P&E Multilink/Cyclone hardware. We used it to test basic hardware peripherals with the utilization of SDK provided by NXP and to use P&E Micro GDB server for firmware debugging and uploading.

The end project does not use S32 Design Studio itself, only its tools. In other words, we are using only P&E Micro GDB Server, and drivers installed for probe support.

S32 Desgin Studio download, installation instruction and documentation can be found on official NXP web-pages in [37].

4.2.2 Wind River Diab Compiler 5.9.6

Wind River Diab compiler is a compiler with many supported target architectures including PowerPC. Due to some source code limitation of NXP's Autosar OS and Autosar MCAL implementations we were forced to use one of the officially supported compilers (Green Hills Compiler or Wind River Diab Compiler or CodeWarrior compiler). This product has evaluation license which can be found at [31]. Installation instruction and user manual will be sent via email after registration.

4.2.3 NXP Software Packages

In this project, we used implementation of Autosar OS and Autosar MCAL[3] provided by NXP. These are software packages comprised of source codes, documentation, integration manuals, makefiles and EB Tresos Studio plugins. These packages are required to be installed as they contains source code of the OS and MCAL modules.

4.2.4 EB Tresos Studio

EB Tresos Studio is configuring tool for Autosar OS and Autosar Basic Software Modules (including MCAL/MAL modules). It is based on Eclipse open source IDE and can be easily extended via plugins. The software packages described in section 4.2.3 will automatically install plugins for all provided software modules. As Elektrobit and NXP are partners, NXP offers evaluation license for EB Tresos Studio. Detailed description and features can be found at [8].

4.2.5 Cygwin & GNU Make & Putty

Cygwin is the large set of GNU and Open Source tools which attempts to migrate utils available on the most Linux platforms to the Windows systems. In other words, it is set of Linux utils such as cp, mv, rm, cd, etc. modified to work on Windows operating system. We used them in the build process (mainly in Makefile) to create folders, remove files, find text in the source code, etc. Installation tool and documentation can be found at [6]. To be able to use these tools inside of the Windows command prompt (cmd) add `CYGWINROOT/bin` folder to your system path.

The make utility is automatization tool for the building process. It controls and executes the compilation, linking and other source code manipulation. You can either install stand-alone GNU Make[4] utility or use one shipped with Matlab (in our version of Matlab it is GNU Make 3.81).

Putty is open source software which can be used as serial link monitor. Installation instruction and documentation can be found at [38].

4.2.6 Matlab + Simulink

Matlab/Simulink is a powerful toolset for analysis and model design applying code-based and visual approach. These tools are widely used for Model-Based design (section 2.1) in many engineering fields. We used Simulink, Matlab Coder, Simulink Coder [27]

[3]Autosar 4.0 https://www.nxp.com/autosar
[4]http://gnuwin32.sourceforge.net/packages/make.htm

and Embedded Coder [15] to create a model and then generate C/C++ code. The code is later compiled and executed on the target hardware.

To install Simulink and required toolboxes follow the official installation guide available at [26] and [17].

```
----------------------------------------------------------------------
MATLAB Version: 9.1.0.441655 (R2016b)
MATLAB License Number: *****
Operating System: Microsoft Windows 10 Home Version 10.0 (Build 16299)
Java Version: Java 1.7.0_60-b19 with Oracle Corporation Java
HotSpot(TM) 64-Bit Server VM mixed mode
----------------------------------------------------------------------
MATLAB                                    Version 9.1         (R2016b)
Simulink                                  Version 8.8         (R2016b)
Control System Toolbox                    Version 10.1        (R2016b)
Embedded Coder                            Version 6.11        (R2016b)
MATLAB Coder                              Version 3.2         (R2016b)
MATLAB Compiler                           Version 6.3         (R2016b)
MATLAB Compiler SDK                       Version 6.3         (R2016b)
Simulink Coder                            Version 8.11        (R2016b)
Simulink Control Design                   Version 4.4         (R2016b)
```

Listing 4.1: Output of the Matlab's **ver** command

5. Implementation Details

Chapter 3 explains the design of the developed system. This chapter gives description of implementation details of the project. The result of this work is distributed in the form of a folder with all configurations and sources included. However, this is not stand-alone working example and requires additional software described in chapter 4.

5.1 Folders and Files description

5.1.1 Folders structure

The following listing shows folder structure of the developed system. We separate the project into two parts – one contains all the content necessary for the developing and running firmware on the target hardware and second one for integration with Simulink.

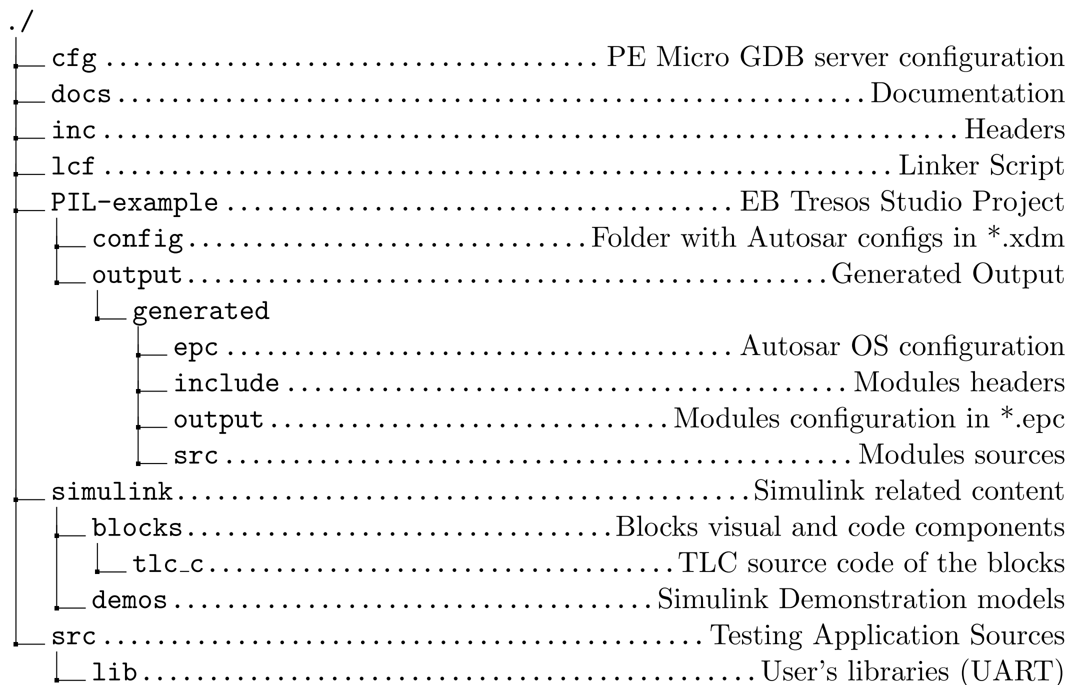


Figure 5.1: Directory tree of the project

The root folder contains a few files with configuration for software tools such as linker or GDB. The other part of the folder contains sources for a testing application.

The testing application means to be the application which has the same structure as software generated by Simulink, except it doesn't have Simulink's injections of code. This can be useful for testing newly added Autosar module or for verification of code which hard to debbug using Simulink modeling tool.

Let's discuss these folders in more details:

- **cfg** - contains configuration file for P&E Micro GDB server which was taken from the S32 Design Studio workspace.
- **docs** - this folder contains this document and its sources
- **inc** - folder with headers used in testing and Simulink applications. These headers defines, for example, variable types, system version, UART library interface. Some of them were taken from NXP's Sample Application, and some were created for the testing application located at `PROJECT_ROOT/src`. Note that these headers are also used for application generated by Simulink.
- **lcf** - folder contains linker script used in the compilation process. This script was taken from NXP's sample application and modified for this project.
- **PIL-example** - contains EB Tresos Studio project with configuration used in this example. The project can be imported into Tresos Studio workspace. To be able to modify the project you need to install plug-ins shipped as part of NXP software package (see section 4.2.3).
- **simulink** - folders with scripts, templates and models for integration with Simulink.
- **simulink/blocks** - blocks' templates and their visual models
- **simulink/blocks/tlc_c** - TLC description of the blocks
- **simulink/demos** - prepared Simulink demonstration models
- **src** - testing application code sources
- **src/lib** - testing application libraries. It contains a library for work with the UART which was taken from one of the NXP's sample application and modified with the necessary functions for our purposes. This library is also used in the application generated by Simulink.

5.1.2 Makefiles

The compilation process of the testing and Simulink's application is controlled by Makefiles (Simulink executes them). The project root folder contains there makefiles - makefile, Makefile.config and Makefile.rules.

- **makefile** - is a Makefile which is used only for compilation of testing application. This file also contains a few rules for uploading firmware into target hardware, cleaning the project and printing some debug information. This file utilizes both of configuration and rules makefiles.

- **Makefile.config** - this file contains the configuration of the compilation process. These are, for example, paths to Autosar OS, S32 Studio, Diab compiler, compilation flags, output folder, etc. Makefile generated by Simulink for model compilation includes these configurations to replace default ones. An Autosar application compiled by this file can be made up of various configuration of MCAL modules. Therefore, to optimize and speed up the compilation process, this Makefile parses Autosar configurations to obtain a list of enabled MCAL modules. The file with the list of enabled modules is located at `PROJECT_ROOT/PIL-example/generated/include/modules.h`. The inclusion of a few key modules is hard-coded (Base, ECUM, RTE). The list is obtained via **Cygwin** (see 4.2.5) utilities such as cat and grep. The code can be seen in listing 5.1.

- **Makefile.rules** - this file contains rules for the compilation of particular files. Note that Autosar OS has separated makefiles located at `OS_DIR/mak/os_defs.mak` and `OS_DIR/mak/os_rules.mak`. If you include this file in another makefile (e.g., in the `PROJECT_ROOT/makefile`), it should be included after Makefile.config. This file is used by Simulink Makefile to compile Autosar OS, MCAL modules and user's libs (located at `PROJECT_ROOT/src/lib`).

```
# Get list of enabled modules
AUTOSAR_MODULES := $(shell cat "$(TRESOS_OUTPUT_FS)/include/modules.h" |
    grep '^\#define USE_' | grep 'STD_ON' | cut -d'_' -f2)

# Base modules which are not included in modules.h
AUTOSAR_MODULES += Base
AUTOSAR_MODULES += ECUM
AUTOSAR_MODULES += RTE

# Get folders from NXP MCAL software package
MOD_FOLDERS := $(addprefix $(MCAL_TRESOS_PLUGINS)/,$(foreach mod_name,$(
    AUTOSAR_MODULES),$(shell ls $(MCAL_TRESOS_PLUGINS) | grep -i "$(
    mod_name)_")))
```

Listing 5.1: Part of the Makefile.config which obtains list of used MCAL modules

5.1.3 Compilation & Execution

The application generated by Simulink contains Makefile in it (the name depends on the model's name $< modelName >$ *.mk*). This Makefile is generated from template **makefile.tmf** located at `PROJECT_ROOT/simulink`. Template includes configuration and rules Makefiles from the project root (see section 5.1.2).

The Wind River Diab compiler is used not only for the compilation of Autosar sources but also for compilation of sources generated by Simulink. Due to the fact that it

is a commercial compiler, it requires a license. The license is located at the path `DIAB_ROOT/license`. The path to this folder is need stored in environmental variable $WRSD_LICENSE_FILE$. It is used by the compiler to determine is license is valid.

As testing application can be compiled from the command line (Command Prompt in Windows), these environmental variable can be set via a **batch** script. The official documentation for the Diab compiler provides instruction how to generate all required variables. We provided the **env.bat** file as an example.

However, Simulink doesn't have access to command line's environment. Therefore, variables should be defined inside of the Matlab/Simulink environment. We defined only licensing path in the `PROJECT_ROOT/simulink/start.m`.

The **Makefile.config** contains a path to the folder that is used for store compiled binaries and end firmware file. By default, this folder is `PROJECT_ROOT/output` (the variable is $OUTPUT_PATH$). However, the firmware generated by Simulink is stored in its project folder. The output folder is used only for storing separate parts (e.g., Dio.o, OS_alarm.o).

The assembled firmware is uploaded via P&E Micro GDB server shipped in the form of Eclipse plug-in for S32 Design Studio IDE. The Simulink's firmware is uploading automatically via Makefile's target **upload**. The testing application, in turn, should be uploaded with the following command **make -f makefile upload**.

The uploading process is carried out by communication between P&E Micro GDB server and GNU PowerPC GDB. The GNU Debugger is also shipped as part of S32 Design Studio.

```
target remote localhost:7224
monitor _reset
load firmware.elf
detach
```

Listing 5.2: GDB script for firmware uploading

We automated GDB uploading process via ***.gdb** script. In case of the testing application, this script is hard-coded into **makefile** under $(FLASH_SCR) target. In case of Simulink projects, it is located in `PROJECT_ROOT/simulink/autosar_pil_make_rtw_hook.m`. This is done due to the fact that Makefiles generated by Simulink have different names depending on the model's name. The *RTW hook* calls this Makefile with **upload** target. After the first uploading, the GDB's script can be found in the project folder. As it is standard GDB debugging session, it is possible to manually debug the target.

The listing 5.2 shows the process of the firmware uploading. First of all, PowerPC GDB [client] connects to the P&E Micro GDB server; the port can be changed in

command line arguments we set it to 7224. Then, debugger resets the target hardware and uploads firmware. Lastly, it detaches from the GDB server and leaves firmware to execute on the hardware.

To assure that application is running it prints starting message on the serial port. The message contains model's name and compilation datetime. The example of the message is shown in the listing 5.3. The serial link can be monitored via Putty (see section 4.2.5).

```
==============Model Started===============
'model_name' - Wed Jun 19 14:10:44 2018 (TLC 8.3 (Jul 20 2012))
```

Listing 5.3: Example of the model start message

5.1.4 Simulink Folder Description

```
PROJECT_ROOT/simulink
├── autosar_pil.tlc ........................................ System Target file
├── generate_lib.m ................................... Library assembling script
├── start.m ................................................ Entry point script
├── file_process.tlc ......................... mrmain's initialization template
├── slblocks.m ........................................ Block library control file
├── autosar_pil_make_rtw_hook.m ................................... RTW hooks
├── makefile.tmf ........................................... Makefile's template
├── sl_customization.m........Customization of MATLAB env. in current folder
├── mrmain.tlc ........................ Template of the main model's source file
├── compile_blocks.m.............................Script for block compilation
├── autosar_pil_select_callback_handler.m.....Callback for System Target file
├── rtiostream_serial.c ............ Serial interface for External Mode support
├── setup.m...........................................Library installation script
├── demos ............................................... Demonstration models
└── blocks ................................................... Library's blocks
    ├── header.c................................S-Functions' common header
    ├── sfunction_din.c ........................................ DIN S-Function
    ├── sfunction_xxx.c.......................................xxx S-Function
    ├── sfunction_pwm.c.....................................PWM S-Function
    ├── trailer.c...............................S-Functions' common trailer
    ├── din.slx ................................DIN visual model for Simulink
    ├── xxx.slx.................................xxx visual model for Simulink
    ├── pwm.slx .............................. PWM visual model for Simulink
    └── tlc_c .............................. TLC template for particular blocks
        ├── sfunction_din.tlc..................................DIN TLC template
        ├── sfunction_xxx.tlc ............................... xxx TLC template
        ├── sfunction_pwm.tlc.............................PWM TLC template
        └── common.tlc ..................... Common function for TLC templates
```

Figure 5.2: Shorten directory tree of the Simulink folder

Figure 5.2 shows the list of the files in Simulink folder. The part of these scripts compiles and assemble the blocks library. The other part sets and configures the environment. The following list describes purposes of these files in more details:

- **autosar_pil.tlc** - is so-called System Target file. It exerts the process of code-generation during model build. It defines variables which are pivotal to the building process. For example, we defined only a few important parameters - the name Makefile's template, the name of code-generation utility (*make_rtw*), language, code format, target type, etc. For more information see [20].
- **generate_lib.m** - this script creates library file and adds blocks to it.
- **start.m** - it is an entry point to the library compilation. This file were created mainly to separate parameters which need to be defined before usage.
- **file_process.tlc** - wrapper for *mrmain.tlc* template. This file is used by *autosar_pil_select_callback_handler.m* to set it as ERT custom template.
- **slblocks.m** - Simulink block library control file. It defines a few library's metadata for library browser.
- **autosar_pil_make_rtw_hook.m** - hook file[1] that invokes custom functions in the specific phases of the build process.
- **makefile.tmf** - template of the Makefile for the projects generated by Simulink.
- **sl_customization.m** - the function which is called to customize Matlab's environment in the current folder[2].
- **mrmain.tlc** - template of the main model's source file. More details can be found in section 5.2.1.
- **compile_blocks.m** - all blocks listed in the array in the *start.m* script and which's corresponding files are presented in the *blocks* folder are compiled via Matlab's MEX compiler.
- **autosar_pil_select_callback_handler.m** - callback function which called when *autosar_pil.tlc* is set as model's System Target file. This callback function sets recommended parameters of the model. These parameters can be changed via graphical user interface in model's preferences.
- **rtiostream_serial.c** - abstraction layer between target serial link implementation and external mode libraries provided by MathWorks.
- **setup.m** - installation script which sets a few important variables. Then, it compiles blocks via *compile_blocks.m* and assembles them into library via *generate_lib.m.*

[1] `https://www.mathworks.com/help/rtw/build-process-customization.html`
[2] `https://www.mathworks.com/help/simulink/ug/registering-customizations.html`

- **blocks/header.c** & **blocks/trailer.c** - common header and trailer for blocks' sfunctions.
- **blocks/sfunction_xxx.c** - sfunction for *xxx* block. These files are just data sources/sinks which describe a number of block's inputs, outputs, parameters and carry a few additional operations such as parameters' validation.
- **blocks/xxx.slx** - visual representation of block *xxx*. It is create a mask of the block with port labels and icon on the block. In other words, it is an appearance of the block in Simulink.
- **blocks/tlc_c/common.tlc** - common header for TLC templates.
- **blocks/tlc_c/sfunction_xxx.tlc** - TLC template for *xxx* block. This template is written in TLC language and used by Simulink to generate C code. Therefore, these files are code-description of the blocks.

The current library set contains 6 blocks: ADC, Digital Input (din), Digital Output (dout), PWM, SCIR (SCI Receive), SCIS (SCI Sender), Overrun. These are basic blocks with a minimal number of parameters. The Autosar MCAL modules have a broad range of function to configure and interact with hardware peripherals. The required functionality can be easily added to the existing implementation.

The configuration of the Autosar system is a quite complicated process of architecture planning and setting up all the components together. Moreover, Autosar configuration is highly depended on project requirements and necessary functionality (see section 3.2.1). Even though it is possible to implement a simple Simulink wrapper for the provided interface, it is time-consuming to handle all the situations of the driver utilization. Simulink is not a configuring tool for Autosar, and it can not offer the same level of functionality. Therefore, we agreed with the assignment submitter that this set of blocks is enough for the demonstration of the system working capacity and switched priority to the other parts of the project.

- ADC - converts a value from a chosen channel
- Digital Input - reads boolean value (LOW/HIGH) on hardware pin
- Digital Output - writes boolean value (LOW/HIGH) to hardware pin
- PWM - sets PWM signal on a chosen channel with a variable duty cycle
- SCIR - receive one byte from the serial port
- SCIS - sends one byte to the serial port
- Overrun - overrun detection block

The serial port is attached to the LIN2 hardware channel. The configuration of this channel can be found in Autosar OS configuration. The parameters of the serial link are hardcoded in the UART library located at `PROJECT_ROOT/src/lib`. The parameters are 115200-8-N-1.

Note that we also added **overrun** diagnostic block to the library (except visual model). Unfortunately, due to the fact that we failed to integrate External mode support in our target, this block does not work as we expected. It is a simple block with one boolean output which indicates an occurrence of the overrun. However, as the target hardware can not communicate with Simulink, it can not send an event that overrun occurred. Therefore, we change it in the way that when overrun is detected the code corresponding to this block sends message on a serial port.

The library can be assembled by running **start.m** script. It in turn calls **compile_blocks.m** and **generate_lib.m**. The **start.m** contains a few important parameters which need to be set before a run.

5.2 User Application

We implemented single task – multirate executing model. In this type of architecture, all the particular model parts (with different sampling times) created in Simulink are periodically executed in only one task. To decide which part of the model execute and which not Simulink provides a mechanism in the form of the function *rate_scheduler()*. This scheduler is executed with fundamental model rate (a greatest common divisor of all sample times) and marks all the control loops which need to be activated in this execution step.

The program consists of three task – main periodic task (*MAINTASK*), simulation task (*SIMULATIONTASK*) and external mode task (*EXTMODETASK*). All model related code is periodically called from the simulation task. The main periodic task is responsible for overrun detection and synchronization of the simulation task. The a mechanism for single tasking architecture is shown in figure 5.3.

The code of the user application is automatically generated from the model developed via Simulink. There are two primary files which need to be mentioned – *ert_main.c* and *modelname.c* (filenames depend on the template and model names, therefore, they can be different):

5.2.1 ert_main.c

ert_main.c - it is the main entry point to the user application, and it contains C *main* function. This file is generated from a template (*mrmain.tlc*) which is written in the

form of markup language used by Simulink for code modification. *mrmain.tlc* contains function calls and variables definition which will be replaced by corresponding C code.

The **ert_main.c** is a file which defines application structure in the form of tasks. It contains tasks functions and *main()* function. *main()* is the initialization of basic modules (MCU, ECU), interrupts, serial link and modules which added by Simulink. Simulink add only those modules which it found in the model. For example, if the model uses ADC, then Simulink will add ADC initialization into *main()* function. This is done by the following command: $\% < LibCallModelInitialize() >$ (in *mrmain.tlc*). In final step *main*() function pass control to the operating system.

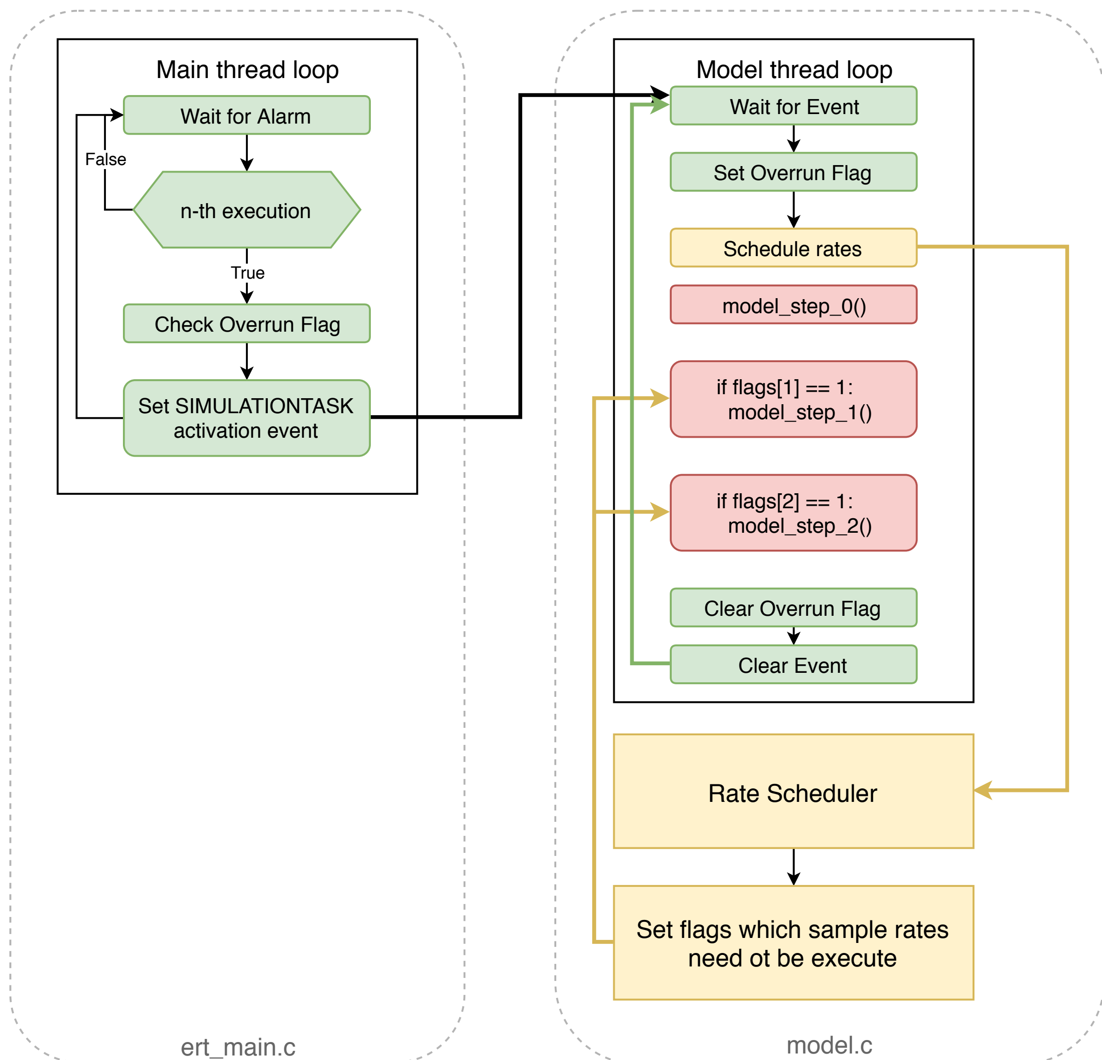


Figure 5.3: Application inner structure

There is three task function corresponding to each task – MAINTASK, EXTMODE-TASK, and SIMULATIONTASK. The MAINTASK is running with sample time equal $500\mu S$ via waiting for an event from the alarm. It is done for make execution of the all task precise. This task runs on the highest required frequency and starts SIMULATIONTASK with period calculated by Simulink. Also, MAINTASK have highest

priority and controls if SIMULATIONTASK overran or not.

5.2.2 model.c

model.c - this file is completly generated by Simulink. The file is made up of three function – initialization, one step of the model and termination of the model. These functions are filled up with code in accordance with model's utilization of the blocks. This means that every block has its initalization, step and termination procedures. Simulink Coder collects all those procedures combine them and put into *model.c*.

The most important part is *one step* function. This function implements control loop of the model. The custom blocks created in this project will be described in accordance with their TLC templates.

5.3 OS & MCAL Configuration

The OS is configured separately from particular MCAL modules. The textual configuration files of the operating system can be found at *PROJECT_ROOT/PIL-example/config*. It is stored in ***.xdm** format which a proprietary format is providing enhanced usability features during work with EB Tresos Studio [12]. The following list contains main points about OS configuration:

- OS has one mode ("Mode01") and one application ("Application01")
- We configured three interrupts – UART RX, UART TX and end of ADC conversion
- System has three tasks - MAINTASK, EXTMODETASK, and SIMULATIONTASK (described in section 3.4.2)
- OS has two counters. Both are attached to hardware clocks. First one is a system counter attached to a clock with a frequency of 80MHz. The second one is responsible for the timing of the tasks' execution, and frequency of its clock is 40MHz.
- The counter used for task activation is actually connected to the FXOSC via STM0 clock (see figure 5.4) with prescaler which decrease input frequency to 2MHz
- The system has one alarm attached to the second 2MHz counter. This is cyclic alarm with period of 1000 ticks. At the overflow point it activates MAINTASK.

The more information about the configuration of the clock, counter, alarm can be found in Appendix C (section 8).

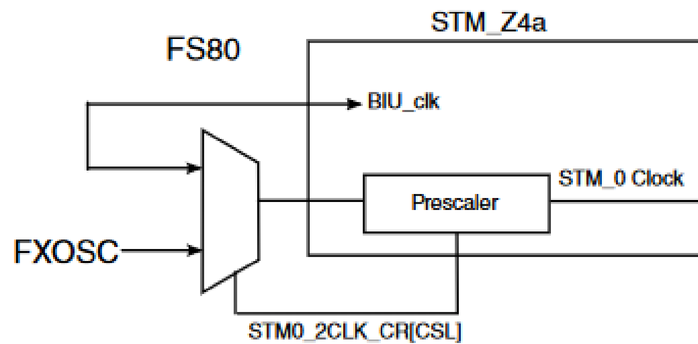


Figure 5.4: STM0 clock generation (taken from reference manual at [36])

Most of the MCAL modules are configured for its minimal working state. These settings can be investigated by using project for EB Tresos Studio (see section 4.2.4) placed at *PROJECT_ROOT/Tresos Studio* or by examining ***.epc** configuration files located at *PROJECT_ROOT/generated/output*.

A few key configuration points which need to be mentioned:

- ADC module is configured to convert a value from one channel (connected to a potentiometer)
- PWM module has only one channel connected to LED1 (DS1)
- One of the LIN channels is used for UART communication (LIN2)
- LED2 - 4 (DS2 - 4) are configured as outputs

6. User Manuals

In this chapter you find several instruction which should help properly configure the project for further work. The previous Chapter 4 – Environment Set-up listed all the tools required for work with the project. In these instruction we assume that these tools are already available in the system.

6.1 Configure Makefile.config

The first step before compiling of any project is to configure **Makefile.config**. As it is explicit from the filename it is file wich configuration such as paths, compilation flags, and files lists. The file is divided into section via comments to make it clearly readable and intuitive. The section named "**User defined paths**" contains paths which need to be changed in accordance with your installation paths.

- **PROJECT_ROOT** - this path is generated automatically via *cygpath* utility (it is part of the Cygwin set; see 4.2.5). In case of the problem with the *cygpath* it can be filled manually.
- **OS_ROOT** - path to the NXP Autosar OS folder.
- **S32_STUDIO** - path to the root folder of S32 Design Studio.
- **NXP_MCAL** - path to the NXP Autosar MCAL folder.
- **DIABDIR** - path to the Diab compiler. Note that it is not the root folder of the Wind River Diab package. It is path to compiler folder, for example, located at `DIAB_ROOT/compilers/diab-5.9.6.2`

These are the necessary modifications which need to be applied. Feel free to modify any of the other parameters such as output folder, cygwin utilities, compiler flags etc.

6.2 The use of Testing application

The build process of the testing application is automated via Makefiles. The first step, as described in the preceding section 6.1 is to assign correct paths. The rules makefile **Makefile.rules** does not require any modifications. The entry point to start the building process is **makefile**. It has a few key targets:

- **all** - compiles the testing application. By default, it takes only `PROJECT_ROOT/src/main.c` file. To add more sources append them into **CC_FILES_TO_BUILD** variable. An assembled firmware will be located at `PROJECT_ROOT/output/bin/firmware.elf`.

- **upload** - uploads the firmware into the hardware target.

- **clean** - remove firmware and all of the object files including the OS and MCAL modules related.

The **all** target is default one. After successful compilation use **upload** target to flash firmware into target hardware.

The target **showflags** is used for debugging purposes and can help to check if all paths and files are correct.

On the Windows machine *make* utility can be run with the following command: **make -f Makefile target**, where *Makefile* is the name of the makefile and *target* is the name of the target. The commands are execution from the *Windows Prompt* or *cmd.* Note that to run *make* without leading path it should be added to the environment path variable.

6.3 Assemble and install Simulink library

The Simulink library itself is the collection of blocks. An assembled library can not be modified. To modify existing blocks or add a new one you need to reassemble the library. The process is of the installing library for the first time or reassembling it with modified blocks is following:

1. Open Matlab and go to the `PROJECT_ROOT/simulink` via **cd** command

2. Open **start.m** script

3. Specify path in the **compFolder** variable

    (a) **compFolder** is a folder with Wind River Diab compiler (see section 4.2.2). The Makefile generated by Simulink calls project related Makefiles which in turn specify this compiler as the main one. Wind River Diab compiler expects a few environmental variables to be defined. These variables contain list of folders including licensing one. Matlab use the **compFolder** path to define a variable in its internal environment to make it visible for the compiler.

4. Run this script. It will compile blocks, assemble them into the library and load it into the Matlab/Simulink environment. The library file will be located at `PROJECT_ROOT/simulink/blocks/pil_mpc5748g.slx`.

Now, the library should be available in the library browser under the name of **Autosar PIL - MPC5748G**. In the end, the script opens one of the demonstration model specified in the **demo_name** variable.

Note that as TLC templates of the blocks are not a part of the library and used only for code-generation you can freely change them without a need to reassemble the library.

6.4 Create new model

The process of creating a new model for Processor-in-the-loop testing is almost the same as in a normal case. However, there exist a few differences. We automated all the model's parameters set up via defining custom target description file. In other words, we simplified the process of creating a new model to only specifying a correct System Target file.

1. Open Simulink and create a new model
2. Open model preferences
3. Go to **Code-Generation** tab
4. In the top area (named **Target selection**), in the field **System Target file** click **Browse...**.
5. Go to the `PROJECT_ROOT/simulink` folder. There will be custom **System Target file** named *autosar_pil.tlc*.
6. After that, all settings should update our default values. Now the model is ready for compilation.

6.5 Adding new functionality

During this project, we implemented a set of basic blocks for work with the microcontroller's peripherals. These are Digital Input/Output, ADC, and PWM, SCI. But MPC5748G has much reacher set of the peripherals which can be used in an advanced project. In this section, we will describe how to add new functionality to already created project and where to find documentation for correct configuration.

The process can be divided into two steps – hardware configuration and adding interface to the Simulink library. The following sections explains both steps separetly.

6.5.1 Configure hardware

The first step is to configure hardware and create entry points. MCAL/MAL is abstraction layer right above the hardware. Autosar defines a specification for most of the hardware peripherals. That means that high-level configuration almost does not depend on low-level hardware configuration. In other words, the configuration of parameters of different peripherals does not require knowledge of microcontroller's low-level registers which need to be changed. For example, we did not work explicitly with the hardware registers to configure hardware pins mode or direction. However, there are many parameters which are platform depended and requires some knowledge of the microcontoller implementation, for example, hardware clocks.

NXP's plug-ins for EB Tresos Studio have the documentation of all of the fields offered in the graphical user interface. Moreover, the plug-in folder has two stand-alone documentation files - module description and integration manual. The last one is especially useful as it contains information about module dependencies.

Moreover, EB Tresos Studio has problem view window which is quite useful and helps to create all required entities. It monitors all the changes made by the developer and analyzes the configuration for conflicts or missing declarations. There exist a good webinar about Autosar MCAL software from NXP available at [34].

6.5.2 Creating Interface to Simulink

The next step after hardware configuration is to create according blocks in the Simulink library. As described in section 3.5.1 the particular blocks are made up of three files.

Firstly, to describe block's attributes you need to create a Matlab executable file. It is written in C language and compiled via MEX wrapper for a compiler. The MEX shipped as part of the Matlab environment (for more information see [19]). The implementation of your newly added block should be placed at `PROJECT_ROOT/simulink/sfunction_name.c`, where *name* is the name of your block; it is good practice to choose short and memorizable abbreviatures. You can freely use already implemented blocks to get an insight of basic structure of those files. More details about available framework can be found at [13].

```
MATLAB_ROOT/bin/mex sfunction_name.c
```

Listing 6.1: Command to compile MEX from the Matlab

Note that you should define *S_FUNCTION_NAME* constant (in the begging of the file) which will be used for attaching it to the visual model. After the block is complete compile it with Matlab **mex** utility; it can be done by calling command in the listing 6.1

Secondly, you need to create visual model of the block:

1. Open Simulink and create New empty model
2. Open Library Browser
3. Search for **s-function** and add it to the model (see figure 6.1)
4. Open preferences of s-function block (Right Click - Preferences)
5. In **S-Function name** field type name defined in your sfunction source code; it is a value of the *S_FUNCTION_NAME* constant.
6. If you defined any parameters type them into **S-Function parameters** field (for example, double(SParameter1)).
7. Press *apply* and close this window. Now, you should see the block with the corresponding number of inputs/outputs as in your MEX file.
8. To make block recognizable and mnemonical you can apply a mask on it. This can be done by *Right Click - Mask - Create Mask*

Finally, we need to TLC template which will be used to generate the source code. The templates should be placed at `PROJECT_ROOT/simulink/blocks/tlc_c/sfunction_name.tlc`. Inspect our implementation to get an insight of the way to write those templates. For more information check official documentation available at [23].

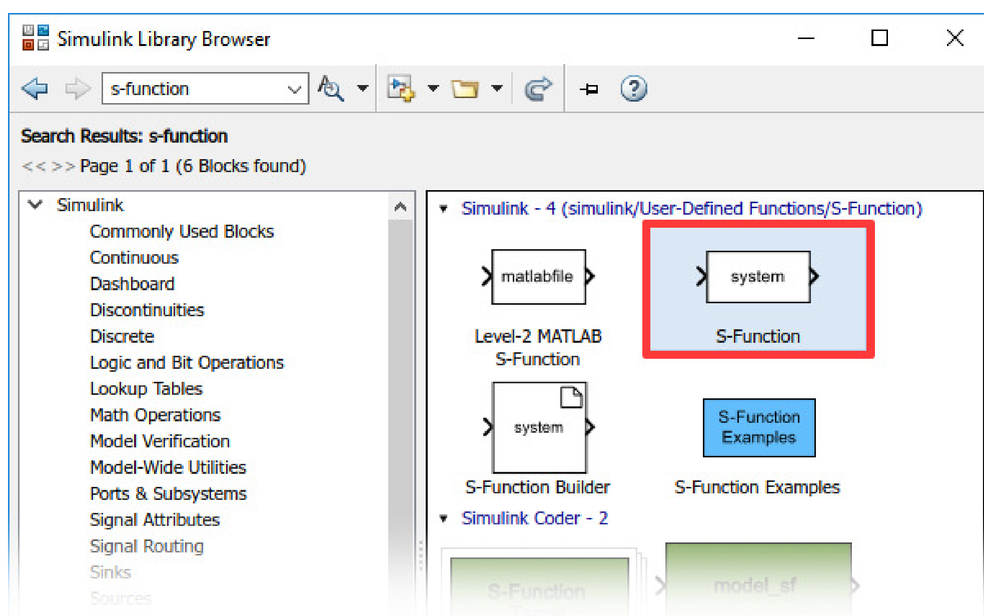


Figure 6.1: The block is point up by the red rectangle

At this phase, the block is ready to be added to the library. In the **start.m** script add a name of your block into the list **blocks** (the name of the block is the same as the filename of the **.slx* file). After running the script, block will be added to the library.

7. Evaluation

In this chapter, we present four demonstration models which utilize blocks from the developed Simulink library to show functionality and characteristics of the implemented solution.

7.1 Demonstration of IO blocks

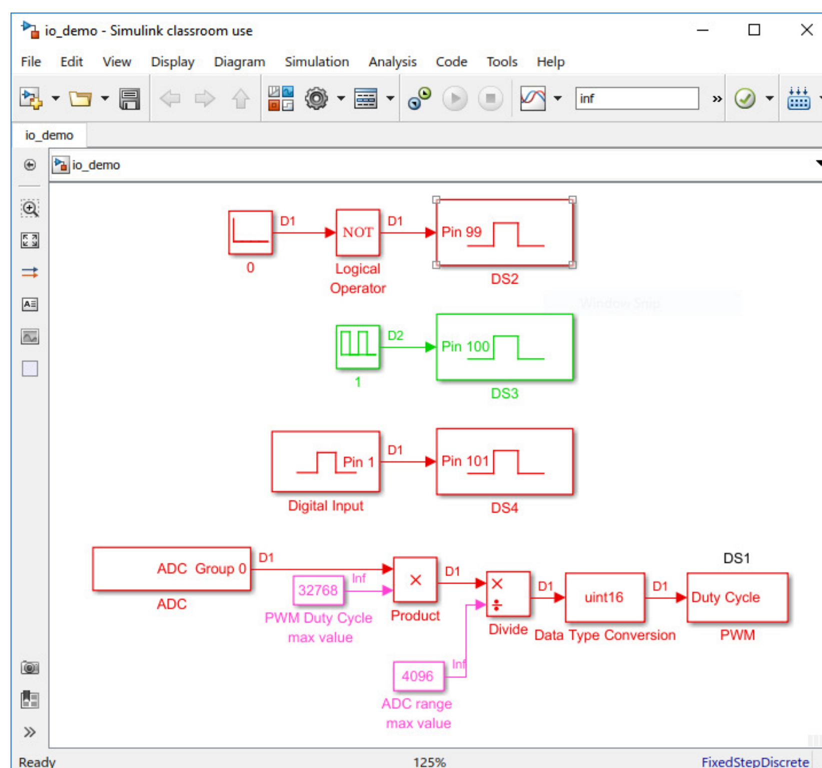


Figure 7.1: Demonstration model of all implemented IO blocks

The first model utilizes the low-level peripherals blocks available in the library. These blocks are – Digital Input, Digital Output, ADC block and PWM block. This model uses all four on-board LEDs to indicate that application is running. All four control loops have different colors which represent different sampling rates. The second control loop (green one) is running on the sample time equals to 0.5. Therefore, LED (under pin 100 – DS3) will be toggle every half of a second. The others loops (red ones) are running on the sample time of 0.01 fraction of a second. The fourth control loop reads the analog value on potentiomentr (the blue one, used for ADC tests), calculates the corresponding value in the range of PWM's duty cycle and write it to the LED (first one on the board – DS1).

7.2 Timing and Multirate Simulink model

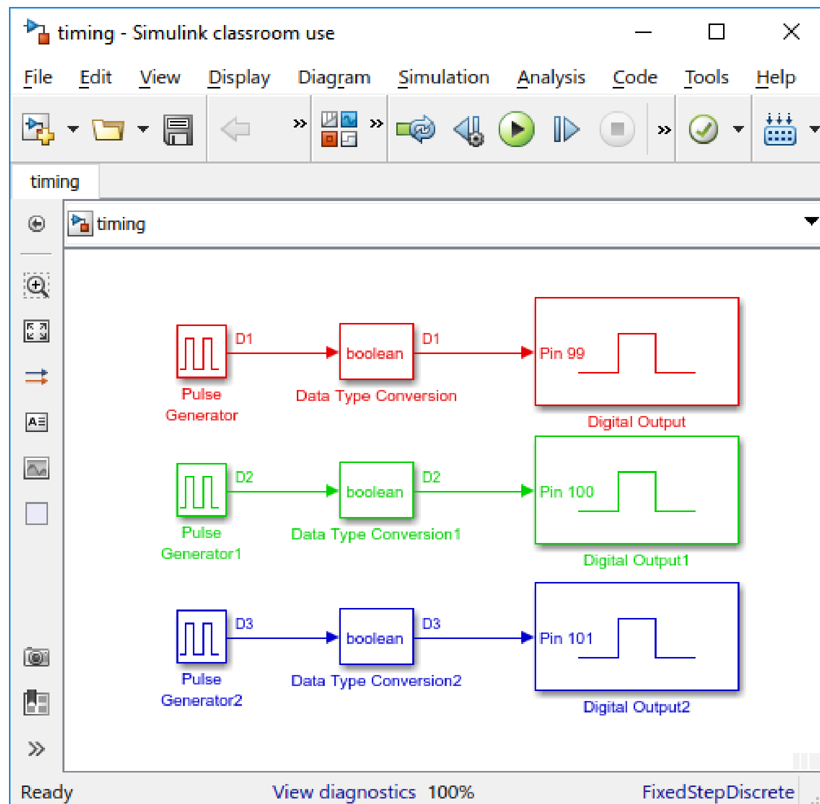


Figure 7.2: Demonstration model of control loops with different freqencies

This model demonstrates correct work of timing and multirate mechanism. It uses three different frequency to toggle three LEDs. Different colors denote different frequencies. In non-trivial models, it can be hard to follow all the model sample times, to make it more convenient Simulink has the legend which can be opened via menu: **Display - Sample time - All**. This window is shown in figure 7.3.

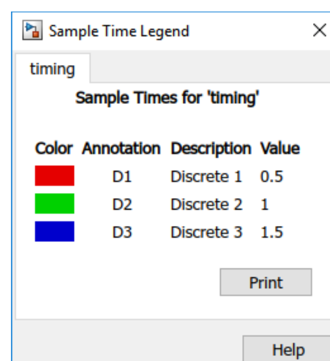


Figure 7.3: Freqincies of the model shown in figure 7.2

7.3 Granularity

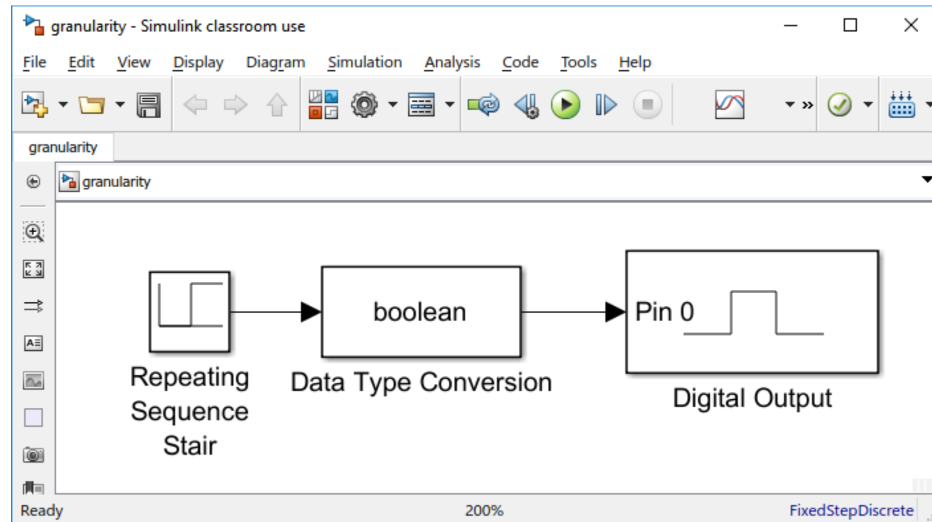


Figure 7.4: Model used to show correctness of timing mechanism

It is important for control application to have precise timing. To proof that our implementation has required the precision of the execution rate, we created demonstration model with maximal possible frequency (2000 Hz). The repeating sequence stair, shown in figure 7.4, has sample time 0.0005 and output vector [0 1]. The generated signal is converted into boolean value (LOW/HIGH) and then written on one of the pins (in this case it is PA[0]). We recorded this pin with Oscilloscope and measured a frequency of 1000 Hz which corresponds to our expectations (pin is toggled every cycle therefore frequency will be two times lower). The frequency can be seen in the left-bottom corner in the figure 7.6.

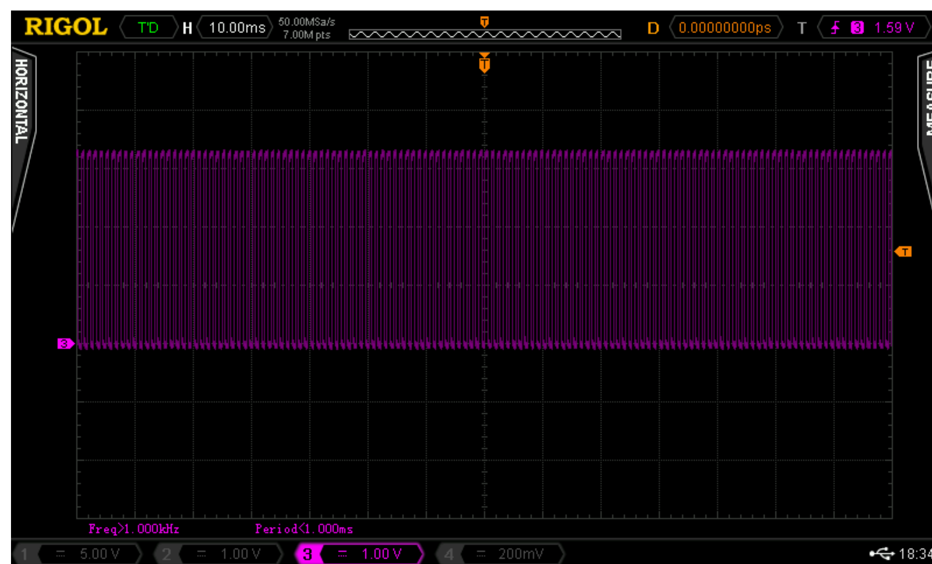


Figure 7.5: Signal recorded with osciloscope for model in figure 7.4

7.4 SCI Echo Server

The last demonstration model is so-called Echo-server. The hardware target sends back all the data it receives. The principle is simple: the SCIR block receives one byte from the serial port, and the SCIS block sends this byte back. However, the model is constantly executed which means that block SCIS attempts to send character every execution step. This, in fact, creates constant spamming of empty characters on the serial port. We solved this issue by wrapping the SCIS block into subsystem which is activated only if SCIR successfully received a character. For monitoring serial port on Windows machine you can use *Putty* (see section 4.2.5).

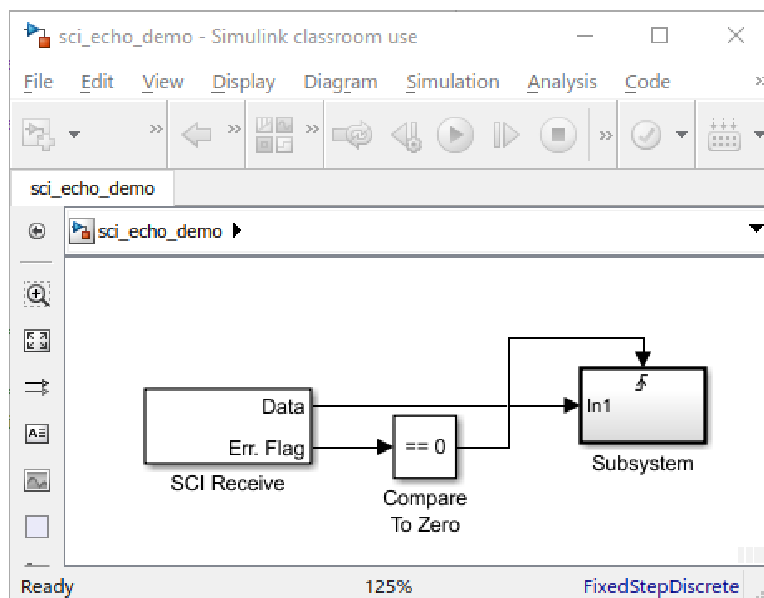


Figure 7.6: Model of the Echo server

7.5 Overrun demonstration

For the demonstration of the overrun diagnostics, we used the testing application. We did not find an elegant solution to create an artificial overrun from the Simulink. Therefore, we added infinite while loop in the testing application which is activated when a pushbutton (SW4) on the board is pressed.

```
// Wait while button is pressed
while(Dio_ReadChannel((Dio_ChannelType)DioConf_DioChannel_SW4) == STD_HIGH){
    ;
}
```

Listing 7.1: Fragment of the code for creating overruns

This hang-up leads to long task execution and creates overrun. The overruns are signalled on serial port by message outputed by the MAINTASK.

8. Conclusion

The primary objective of this thesis was to create a basic prototype of the processor-in-the-loop system running Autosar operating system on the MPC5748G-LCEVB Development Board. The project is similar to the Rapid prototyping platform (RPP) developed earlier at the CTU. The key difference is the utilization of the automotive industry standard – Autosar.

The main objective was to investigate and make use of Autosar operating system with further integration with Matlab/Simulink environment. The Autosar open architecture is quite a massive software standard with a plethora of specifications and rich set of configuration parameters. The operating system was configured to satisfy the suggest system design. The microcontroller abstraction layer modules were configured to create a set of basic blocks to demonstrate capabilities of the system. Thesis assignment states more extensive list of the blocks, however, as was shown in the section 5.1.4 and section 3.2.2 by the agreement with the submitter we decided to make only a demonstration set of essential blocks. Our configuration of the system and its internal design make it possible to assure necessary granularity of $500\mu S$. Moreover, the overrun diagnostics was included in the system setup. The developed Processor-in-the-loop system is integrated with Matlab R2016B (64 bit) running under MS Windows operating system. The setup of the environment for the model development and PIL simulation is mostly automated and only requires some essential modification of the configurational files. The thesis contains the list of the tools recommended for work this project with links to datailed documentation. We described suggested design of the system and provided several instructions describing the use and ways of improving the system. Lastly, we created evaluation models to demonstrate the system's features and to help getting started with the project.

External Mode The development or prototyping of a new model can be even more productive if there would be External Mode support. It allows to change parameters of the blocks in run-time and view data-flows inside a model. Unfortunately, the source code provided by MathWorks, Inc. does not have any documentation at all. There exist a few examples on the internet with external mode integration into the custom targets. However, they are outdated and does not provide enough information for implementing it in another system. The correct integration of the external mode support apparently involves some reverse engineering of quite complicated and old code.

However, during this project, we carried out some experiments and collected information and records about the integration of the external mode into this or any other project. These materials are available as an attachment to this work.

Fixed Configuration of Autosar + GNU Compiler Implementation of Autosar OS and Autosar MCAL modules require compilation with compiler officially tested and supported by NXP. However, Eaton Corporation would like to use free or custom compiler for the further work. Due to the fact that user application code is written/-generated in C language and doesn't contain compiler-specific constructions or syntax, there is a possibility to compile it with GNU Compiler for PowerPC architecture[1].

Therefore, the compiled version of the pre-configured Autosar components for specific project will allow to use a developed system without additional compilers such as Wind River Diab Compiler or Green Hills Compiler.

[1]https://gcc.gnu.org/onlinedocs/gcc/RS_002f6000-and-PowerPC-Options.html

Bibliography

[1] Autosar. *Autosar - The standardized software framework.* [Online; accessed January, 2018]. URL: https://www.autosar.org/.

[2] Autosar. *Specification of Operating System.* Version: 5.0.0. Revision: 3. [Online; accessed January, 2018]. URL: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_SWS_OS.pdf.

[3] Autosar. *Specification of Run-time Environment.* Version: 3.2.0. Revision: 3. [Online; accessed January, 2018]. URL: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_SWS_RTE.pdf.

[4] Eaton Corporation. *Eaton Web Pages - Automotive.* [Online; accessed May, 2018]. URL: http://www.eaton.com/Eaton/ProductsServices/Vehicle/markets/automotive/index.htm.

[5] Eaton Corporation. *Eaton Web Pages - EGerodisc Differentials.* [Online; accessed May, 2018]. URL: http://www.eaton.com/Eaton/ProductsServices/Vehicle/Differentials/egerodisc-differentials/index.htm.

[6] The Cygwin. *Cygwin collection.* [Online; accessed February, 2018]. URL: https://www.cygwin.com/.

[7] Ewen Denney. "A Software Safety Certification Plug-in for Automated Code Generators". In: *NASA Ames Research Center, Moffett Field* (November 29, 2006).

[8] Elektrobit. *EB Tresos Studio.* [Online; accessed February, 2018]. URL: https://www.elektrobit.com/products/ecu/eb-tresos/studio/.

[9] Elektrobit. *Latest in-car technologies - Autosar.* [Online; accessed March, 2018]. URL: https://www.elektrobit.com/products/ecu/technologies/autosar/.

[10] Carlos Jenkins, Michal Sojka, and Michal Horn. *Code generation for automotive rapid prototyping platform using Matlab/Simulink.* Technical report, Czech Technical University in Prague, [Online; accessed January, 2018]. 2015. URL: http://rtime.felk.cvut.cz/rpp-tms570/rpp_simulink.pdf.

[11] M.DiNatale. *An introduction to AUTOSAR.* ReTiS Lab, [Online; accessed February, 2018]. URL: https://retis.sssup.it/sites/default/files/lesson19_autosar.pdf.

[12] Freescale Marius Rotaru. *Hands-on Workshop: Autosar Training.* [Online; accessed May, 2018]. 2015. URL: https://www.nxp.com/docs/en/supporting-information/ftf-acc-F1243.pdf.

[13] MathWorks, Inc. *Basic C MEX S-Function.* [Online; accessed April, 2018]. URL: https://www.mathworks.com/help/simulink/sfg/example-of-a-basic-c-mex-s-function.html.

[14] MathWorks, Inc. *C/C++ S-Functions Documentation*. [Online; accessed April, 2018]. URL: `https://www.mathworks.com/help/simulink/c-c-s-functions.html`.

[15] MathWorks, Inc. *Embedded Coder*. [Online; accessed April, 2018]. URL: `https://www.mathworks.com/products/embedded-coder.html`.

[16] MathWorks, Inc. *Introduction to the Target Language Compiler*. [Online; accessed May, 2018]. URL: `https://www.mathworks.com/help/rtw/tlc/what-is-the-target-language-compiler.html`.

[17] MathWorks, Inc. *Matlab & Simulink*. [Online; accessed February, 2018]. URL: `https://www.mathworks.com/products/matlab.html`.

[18] MathWorks, Inc. *Matlab Coder*. [Online; accessed January, 2018]. URL: `https://www.mathworks.com/products/matlab-coder.html`.

[19] MathWorks, Inc. *Matlab/Simulink Documentation - Build MEX functions from C/C++ or Fortran source code*. [Online; accessed March, 2018]. URL: `https://www.mathworks.com/help/matlab/ref/mex.html`.

[20] MathWorks, Inc. *Matlab/Simulink Documentation - Controlling Code Generation With the System Target File*. [Online; accessed April, 2018]. URL: `https://www.mathworks.com/help/releases/R2011b/toolbox/rtw/ug/bse3c7m-1.html`.

[21] MathWorks, Inc. *Matlab/Simulink Documentation - S-Function Basics*. [Online; accessed April, 2018]. URL: `https://www.mathworks.com/help/simulink/s-function-basics.html`.

[22] MathWorks, Inc. *Matlab/Simulink Documentation - Simulink Libraries*. [Online; accessed April, 2018]. URL: `https://www.mathworks.com/help/simulink/libraries.html`.

[23] MathWorks, Inc. *Matlab/Simulink Documentation - TLC Files*. [Online; accessed March, 2018]. URL: `https://www.mathworks.com/help/rtw/tlc/tlc-files.html`.

[24] MathWorks, Inc. *Model-Based Design Benefits*. [Online; accessed January, 2018]. URL: `https://www.mathworks.com/solutions/model-based-design.html`.

[25] MathWorks, Inc. *Processor-In-the-Loop Simulation on Embedded Linux Boards*. [Online; accessed March, 2018]. URL: `https://www.mathworks.com/company/newsletters/articles/processor-in-the-loop-simulation-on-embedded-linux-boards.html`.

[26] MathWorks, Inc. *Simulink - Simulation and Model-Based Design*. [Online; accessed March, 2018]. URL: `https://www.mathworks.com/products/simulink.html`.

[27] MathWorks, Inc. *Simulink Coder*. [Online; accessed April, 2018]. URL: `https://www.mathworks.com/products/simulink-coder.html`.

[28] MathWorks, Inc. *Simulink Software Tools for Autosar*. [Online; accessed May, 2018]. URL: https://www.mathworks.com/solutions/automotive/standards/autosar.html.

[29] Charles J. Murray. "Automakers Opting for Model-Based Design". In: *Design News* (November 5, 2010). [Online; accessed May, 2018]. URL: https://web.archive.org/web/20101125060340/http://www.designnews.com/article/511392-Automakers_Opting_for_Model_Based_Design.php.

[30] Vinod Reddy. "Accelerating development with model-based design". In: *embedded.com* (August 21, 2015). URL: https://www.embedded.com/electronics-blogs/say-what-/4440209/Accelerating-development-with-model-based-design.

[31] Wind River. *Wind River Diab Compiler 5.9.6 Evaluation*. [Online; accessed February, 2018]. URL: https://www.windriver.com/evaluations/diab_compiler/.

[32] Lars Rosqvist, Roger Aarenstrup, and MathWorks Inc. Kristian Lindqvist. *Processor-In-the-Loop Simulation on Embedded Linux Boards*. [Online; accessed May, 2018]. URL: https://www.mathworks.com/company/newsletters/articles/processor-in-the-loop-simulation-on-embedded-linux-boards.html.

[33] NXP Semiconductors. *AUTOSAR-4: AUTOSAR 4.0.x (Classic Platform) Software*. [Online; accessed January, 2018]. URL: https://www.nxp.com/autosar.

[34] NXP Semiconductors. *Introduction to NXP AUTOSAR MCAL Software*. [Online; accessed March, 2018]. URL: https://www.nxp.com/video/introduction-to-nxp-autosar-mcal-software:AUTOSAR-OnDemand-Training-Video.

[35] NXP Semiconductors. *MPC5748G-LCEVB: Development Board for MPC5748G*. [Online; accessed February, 2018]. URL: https://www.nxp.com/MPC5748G-LCEVB.

[36] NXP Semiconductors. *MPC574xB-C-G: Ultra-Reliable MCUs for Automotive & Industrial Control and Gateway*. [Online; accessed February, 2018]. URL: https://www.nxp.com/mpc5748g.

[37] NXP Semiconductors. *S32DS-PA: S32 Design Studio IDE for Power Architecture based MCUs*. [Online; accessed January, 2018]. URL: https://www.nxp.com/s32ds.

[38] Simon Tatham. *Putty - SSH and Telnet client*. [Online; accessed February, 2018]. URL: https://www.putty.org.

[39] Vector Informatik GmbH. *AUTOSAR configuration Process*. [Online; accessed February, 2018]. URL: https://vector.com/portal/medien/cmc/events/Webinars/2013/Vector_Webinar_AUTOSAR_Configuration_Process_20130419_EN.pdf.

[40] Vector Informatik GmbH. *MICROSAR - product information*. [Online; accessed May, 2018]. URL: https://vector.com/portal/medien/cmc/info/MICROSAR_ProductInformation_EN.pdf.

[41] Vector. *Vector E-Learning - virtual VectorAcademy*. [Online; accessed January, 2018]. URL: https://elearning.vector.com/.

List of Figures

2.1 Autosar OS Layers Structure 8

3.1 System Architecture 11

3.2 Autosar implementation provided by NXP [36] 14

3.3 Simulink generates only user application (model's code) 15

3.4 Internal structure of the user application 17

4.1 Photo of the evaluation board taken from [35] 21

5.1 Directory tree of the project 25

5.2 Shorten directory tree of the Simulink folder 29

5.3 Application inner structure 33

5.4 STM0 clock generation (taken from reference manual at [36]) 35

6.1 The block is point up by the red rectangle 41

7.1 Demonstration model of all implemented IO blocks 43

7.2 Demonstration model of control loops with different freqencies 44

7.3 Freqincies of the model shown in figure 7.2 44

7.4 Model used to show correctness of timing mechanism 45

7.5 Signal recorded with osciloscope for model in figure 7.4 45

7.6 Model of the Echo server 46

8.1 PIL execution workflow taken from [32] 59

8.2 Warning in the library browser shown when the library is added 61

8.3 The dialog for fixing missing repository information 62

8.4 Screenshot of FXOSC configration in EB Tresos Studio (see 5.3 and 4.2.4) 63

8.5 Screenshot of STM0 configration in EB Tresos Studio (see 5.3 and 4.2.4) 63

8.6 Screenshot of counter configration in EB Tresos Studio (see 5.3 and 4.2.4) 64

8.7 Screenshot of alarm configration in EB Tresos Studio (see 5.3 and 4.2.4) 64

Acronyms

Autosar Automotive Open System Architecture. 8, 14

BSW Basic Software Modules. 8, 13, 14, 23

GDB GNU Debugger. 22, 25, 26, 28

MBD Model-Based Design. 5, 7

MCAL/MAL Microcontroller abstraction layer. 9, 15, 23, 40

MEX Matlab executable. 18

OEM Original equipment manufacturer. 8, 14, 15

PIL Processor-in-the-loop. 1, 5, 7, 10, 12, 19, 39, 47

RTE Run-time Environment. 1, 14, 15

RTW Real-Time workshop (Matlab Embedded coder). 28, 29

SWC Software Component. 14

VFB Virtual Functional Bus. 14, 15

CD content

```
./
├── Project ................................ Developed Project
├── DP_2018_Andrey_Albershteyn.pdf ........... This document
└── External-Mode .... Collected materials about External Mode
```

Appendix A

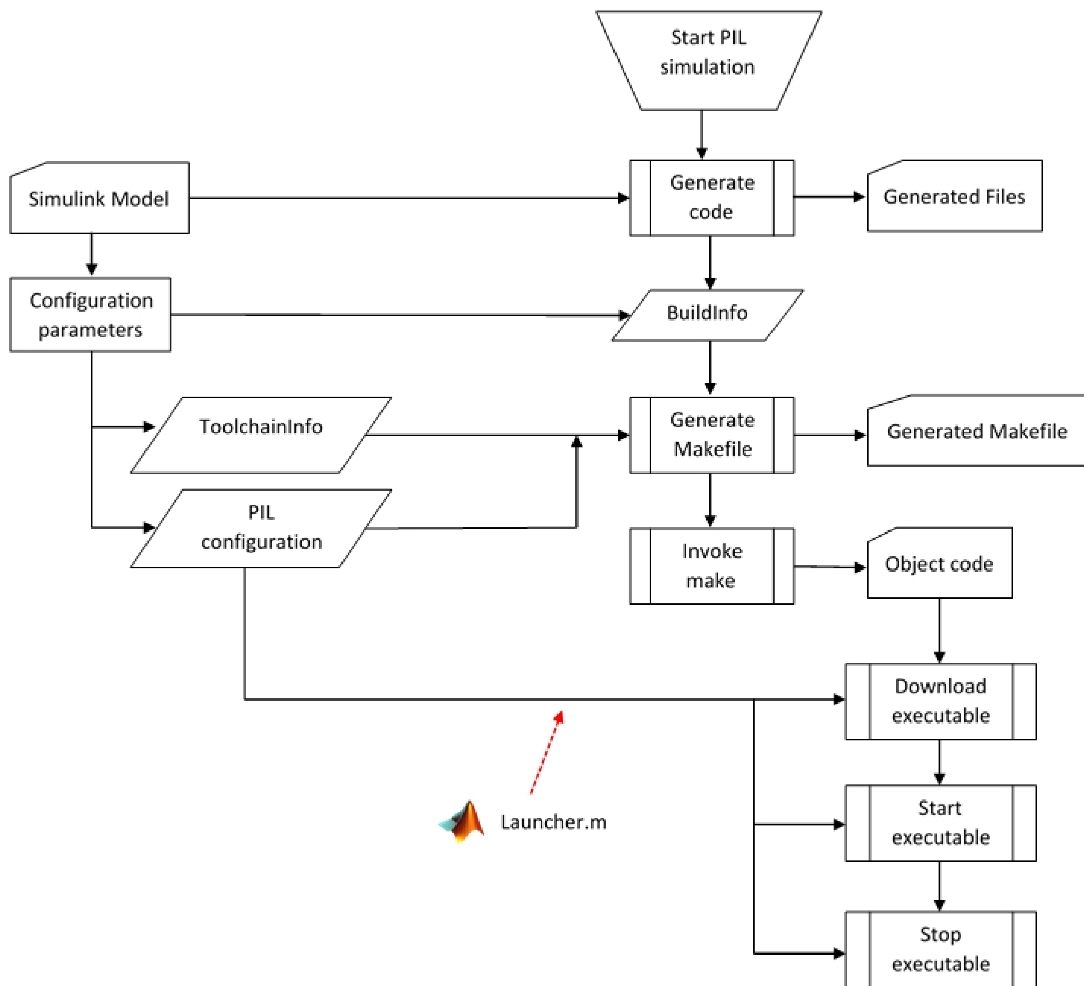


Figure 8.1: PIL execution workflow taken from [32]

Appendix B

This section contains advice and troubleshooting information which can be useful in further development. During the development of the project we spent some time on solving these issues which could be easily avoided.

- Official NXP web-pages contains Autosar + MCAL example for MPC5748G microcontroller. But actually it is for MPC5748C which is a slightly different processor. *Winter, 2018. Forum Discussion [Link]*
- Due to differences in slash usage for *Cygwin* utilities and Windows's paths there could occure mistakes. Make sure your paths are correct.
- Linker Error during Simulink Library blocks compilation. One of the possible reason for this error is that blocks are locked by Matlab/Simulink. Try to close Matlab and Simulink, then, delete all .mexw64 files in *simulink/blocks* directory.
- License Error during compilation process. If you get get fatal error with following message *License error: Unknown LMAPI error* one of the possible reasons is that your environment variables for the compiler are incorrect. Try to regenerate **env.bat** file (the process is described in the Wind River Diab compiler documentation).
- When you assemble the library and add it to the Simulink for the first time you can get warning in the library browser (shown in figure 8.2). The library will not be visible in the list.

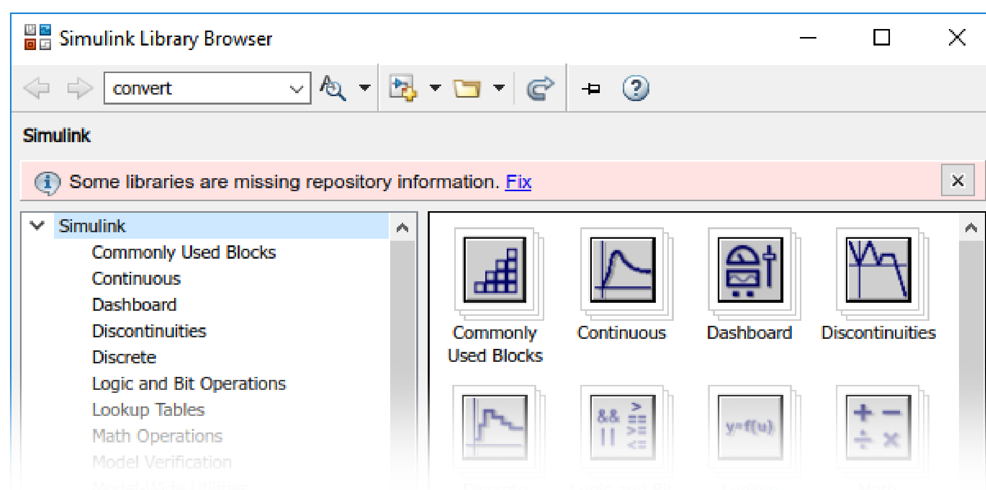


Figure 8.2: Warning in the library browser shown when the library is added

To fix it press **Fix** link. The new dialog will open, shown in figure 8.3.

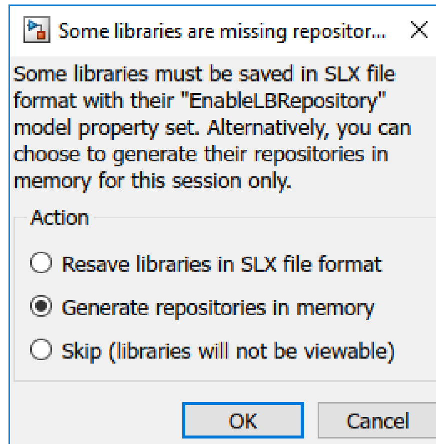


Figure 8.3: The dialog for fixing missing repository information

Choose the second option "**Generate repositories in memory**" and press **OK**. After that the library browser should update information about libraries and the custom library should appear in the list. The name of the library is **Autosar PIL - MPC5748G**.

Appendix C

This Appendix contains screenshots from EB Tresos Studio with configuration of the key components - FXOSC clock, STM0, Counter, and Alarm. These are the key components are used to activate main application task.

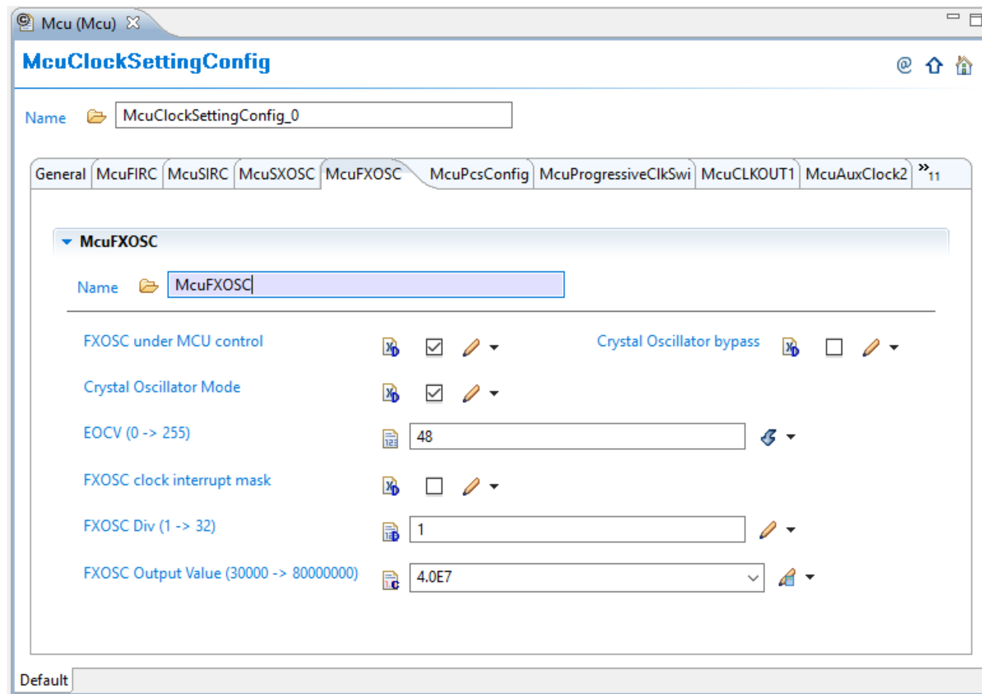


Figure 8.4: Screenshot of FXOSC configration in EB Tresos Studio (see 5.3 and 4.2.4)

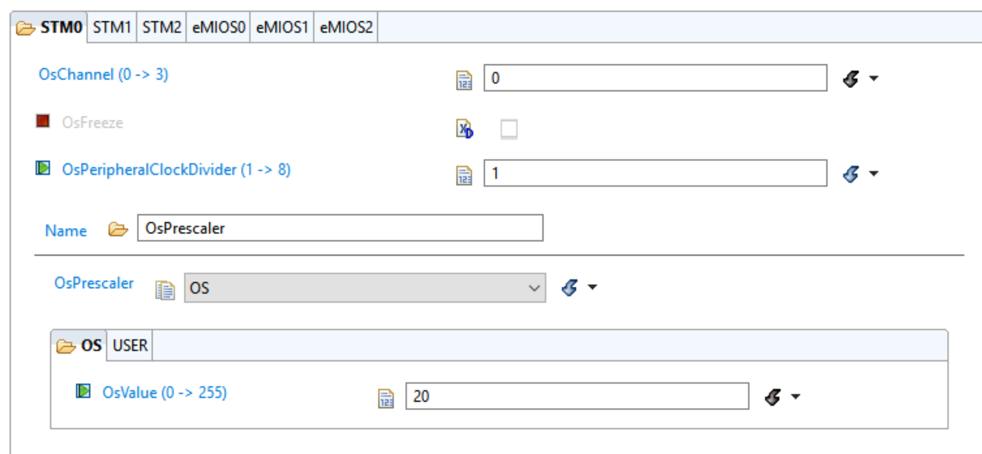


Figure 8.5: Screenshot of STM0 configration in EB Tresos Studio (see 5.3 and 4.2.4)

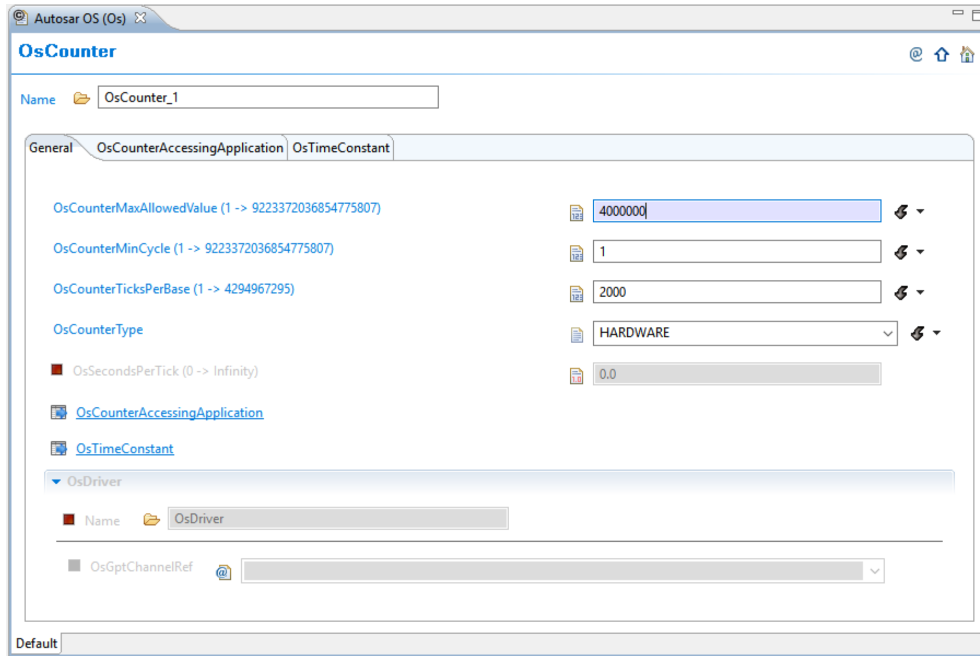


Figure 8.6: Screenshot of counter configration in EB Tresos Studio (see 5.3 and 4.2.4)

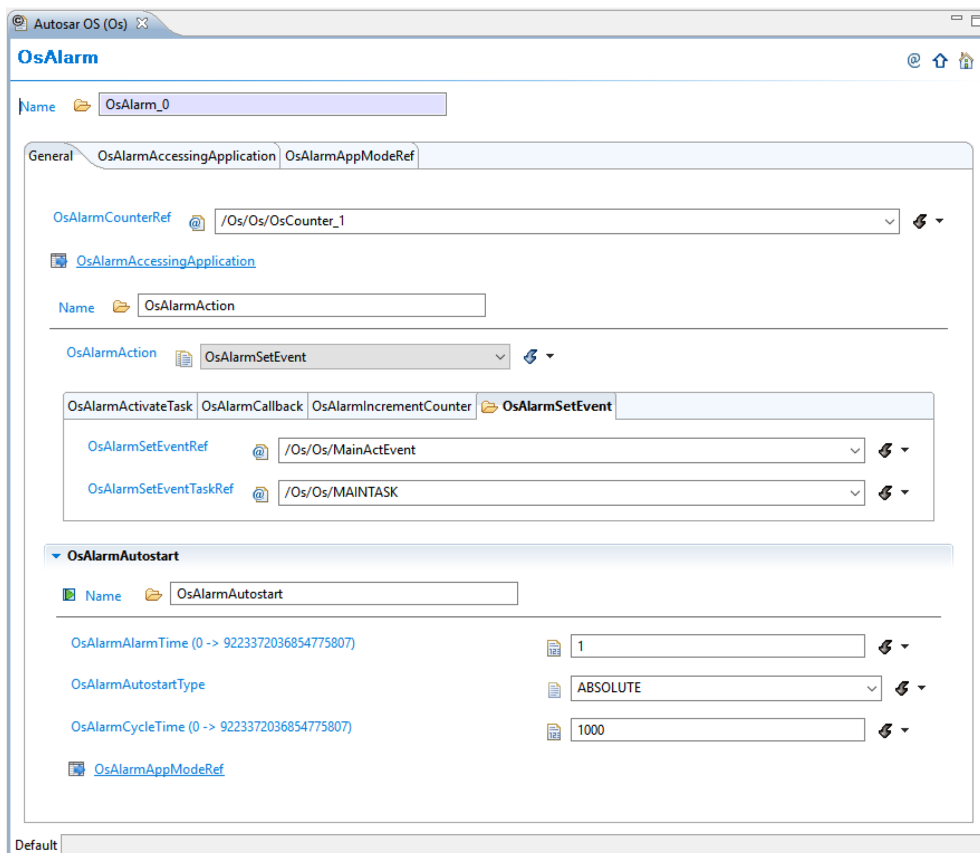


Figure 8.7: Screenshot of alarm configration in EB Tresos Studio (see 5.3 and 4.2.4)

Appendix D

```
>> cd C:\Users\Andre\Project\simulink\
>> start
Start library generation process
Compiling blocks...
"C:\Program Files\MATLAB\R2016b"\bin\mex sfunction_adc.c
"C:\Program Files\MATLAB\R2016b"\bin\mex sfunction_din.c
"C:\Program Files\MATLAB\R2016b"\bin\mex sfunction_lout.c
"C:\Program Files\MATLAB\R2016b"\bin\mex sfunction_or.c
"C:\Program Files\MATLAB\R2016b"\bin\mex sfunction_pwm.c
"C:\Program Files\MATLAB\R2016b"\bin\mex sfunction_scir.c
"C:\Program Files\MATLAB\R2016b"\bin\mex sfunction_scis.c
Removing old blocks:
    pil_mpc5748g/ADC
    pil_mpc5748g/Digital Input
    pil_mpc5748g/Digital Output
    pil_mpc5748g/PWM
    pil_mpc5748g/SCI Receive
    pil_mpc5748g/SCI Send
Adding new blocks:
    from: din.slx:
        din/Digital Input as pil_mpc5748g/Digital Input
    from: dout.slx:
        dout/Digital Output as pil_mpc5748g/Digital Output
    from: adc.slx:
        adc/ADC as pil_mpc5748g/ADC
    from: pwm.slx:
        pwm/PWM as pil_mpc5748g/PWM
    from: scis.slx:
        scis/SCI Send as pil_mpc5748g/SCI Send
    from: scir.slx:
        scir/SCI Receive as pil_mpc5748g/SCI Receive
Closing and saving file pil_mpc5748g.slx
Target setup is complete!
    Current configuration:
        CompilerRoot : C:/Apps/Diab/compilers/diab-5.9.6.2/WIN32
        TargetRoot   : C:/Users/Andre/Project/simulink
        Hardware     : mpc5748g
Load library
>>
```

Listing 8.1: Sample output of the **start.m** script