Multi Chip Simulation of Battery Management System with Synopsys Virtualizer Studio

Frank Poppen, Ralph Görgen, Manfred Thanner
NXP Semiconductor Germany GmbH
Hamburg and Munich, Germany
https://www.nxp.com

ABSTRACT

This work describes the virtual integration and usage of a complete multi-chip battery management system (BMS) in an extensible Synopsys Virtualizer Studio Development Kit (VSDK). The full system simulation (FSS) consists of a microcontroller unit (MCU) S32K3xx with serial peripheral interface (SPI), gateway (GTW) MC33664 to transport protocol link (TPL), daisy chain of battery cell controllers (BCC) MC33774 as parts of a cell monitoring unit (CMU) and simplified battery cell models. Such a FSS is an enabler for early development and evaluation of BMS complex device drivers (CDD) in a safety critical automotive open system architecture (AUTOSAR) environment. Especially the aspects of safety require the completeness of the model in all main functions. The work denotes the virtual integration process, obstacles that needed to be overcome and use of the FSS.

Table of Contents

Table of Figures

1. Introduction

The development of today's automotive microelectronic systems is a demanding challenge. On one hand, there is the high complexity of the systems and the high degree of interaction inside and between them. On the other hand, there are strong requirements for instance on functional safety, reliability, bill of material (BOM[1]), and time to market. Our work here is about a multi-chip automotive battery management system (BMS) consisting of a microcontroller unit (MCU), a gateway (GTW) and a chain of battery cell controllers (BCC) assembled in a cell monitoring unit (CMU). The BMS supervises the voltage and temperature of lithium-ion cells in automotive batteries through highly accurate measurements in a high-voltage environment including requirements such as functional safety, power consumption, and reliability. The possibility of catching fire in the battery of an electric vehicle is rated as an automotive safety integrity level (ASIL) D hazard i.e., the BMS needs to support ASIL D requirements in an ASIL D environment. Furthermore, the system's BCCs are directly mounted to the battery modules and supplied by the battery, meaning that they are permanently powered. Consequently, the ICs power consumption needs to be very low to avoid discharge of the cells. Additionally, this means that the IC cannot be powered off and restarted in case of a failure. In [1] we describe our method to close the gap between the architecture description in SysML and a BCC virtual prototype. As stated there, this allows us to use the architecture definition not only for the virtual prototype but also for a starting point for RTL design and integration as a common source of truth. At the time, the flow concentrated on structure only. In the meantime, we continued our approach by adding functional behavior towards a more complete functional BCC model. But such single component model, even though quite complex in its own internal structure, is just a small part of the system that we need to make saver and more secure. For pre silicon evaluation we had to bring together several stakeholders of different business lines for a complete virtualized BMS in one full system simulation (FSS).

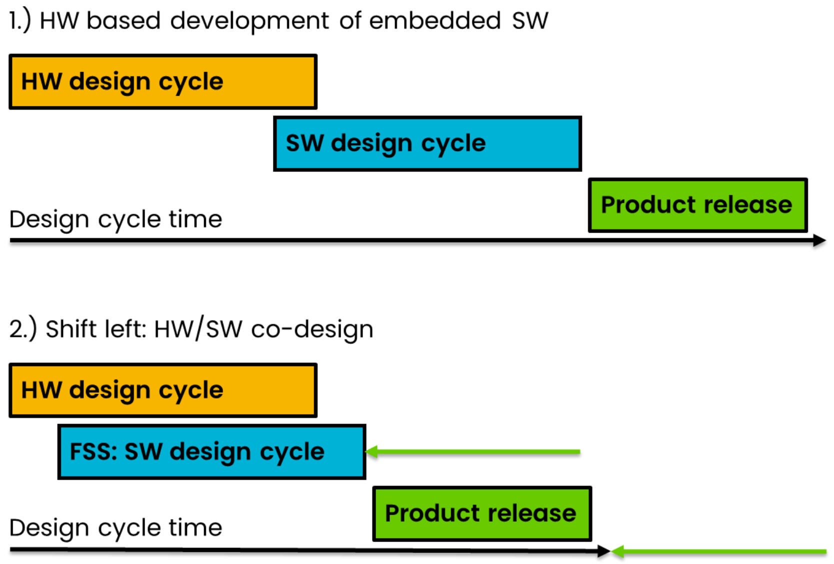


Figure 1. Shift Left with Hardware/Software Co-Design.

The FSS is part of our shift left approach (compare with Figure 1) of moving software development, testing, quality, and safety evaluation early in the development process before we expect first silicon to be available. As with any embedded software, the traditional development and evaluation of the complete software stack of a BMS is originally hardware based and requires the availability and setup

[1] Please refer to Chapter 7. which holds an extensive Glossary.

of several physical components which are usually not available until very late in the design cycle of a new product. As soon as they are available, it becomes a time-consuming task to bring up a yet untested software on complex hardware equipment which generally does not come with visible and fully controllable behavior. Also, with physical components it becomes rather tedious to test multiple configurations as permutations of HW setups with changing number and type of components. Such setups' costs can go up very high for each system. A FSS setup on the other hand can be changed by a simple adaptation of parameters and a restart. RTL simulation is no solution for early software development known as hardware/software co-design. Its execution performance is not sufficient for live software debugging sessions. So, our objective was to create a showcase which proofs that FSS enables the required shift left in early software development.

In the following Chapter 2. an example of a high-voltage battery management system (HVBMS) is examined from a bird's eye view. The knowledge transfer on the topic in this paper is not that deep, but sufficient to be able to understand what we had to accomplish for our FSS setup documented in Chapter 3. In the process we came across an anomaly that involved the simulated SPI communication. It prevented us from communicating SPI data streams of arbitrary bit length. Chapter 4. provides the story on how NXP and Synopsys worked together to understand and fix the issue. Once the FSS was complete we were able to run a full software stack. Chapter 5. depicts what that looks like and what significant use can be drawn from it. The paper closes with our conclusions and future work in Chapter 6. followed by a glossary in Chapter 7. and references in Chapter 8.

2. High Voltage Battery Management System

When we talk about electric vehicles (EV), the first thing that comes to mind is an electric powered car with a battery. In general, however, scooters, motorcycles, buses, trucks, trains and even boats or airplanes can also be designed as EVs. For some applications as in trains, but also buses and trucks, it is an option to draw power from an extravehicular source like overhead contact wires. While this offers a nearly unlimited traveling range it at the same time confines the freedom of mobility severely to limited available infrastructure of electrified tracks and roads. The circumstances are inverted when we have a look at the situation where the EV carries its own source of limited power with it. We can think of fuel that is converted into electric energy by fuel cells or combustion engines as in hybrid cars. Or, as it is in the scope of this paper, we put our emphasis on in-vehicle batteries that come in as many configurations as there are use case scenarios imaginable. In other words, there is no one size fits all BMS solution. The chemistry today is typically good for battery cells at around 3 to 5 V each that are interconnected in parallel for higher currents and in series for higher voltage. This way battery packs are created as low as 14V or high voltages of 800V and more. Unfortunately, the chemistry reacts very sensitively to leaving the range of specified parameters.

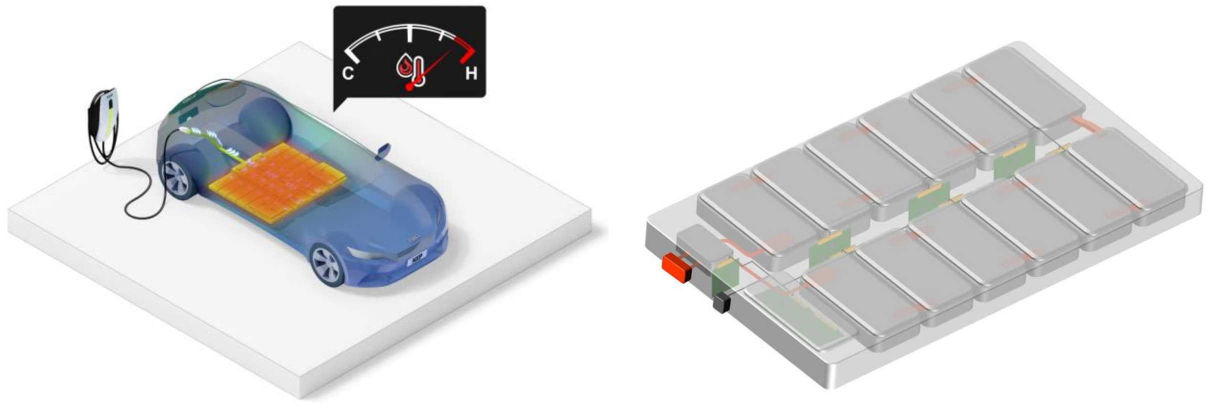


Figure 2. Integrated into an electric car: battery pack of several modules.

NXP offers full system solutions for HVBMS with complete chip sets. They are fully compliant to ASIL D safety requirements and the technical features include outstanding flexibility. Please refer to the following Figure 3 as one of many possible BMS setups.

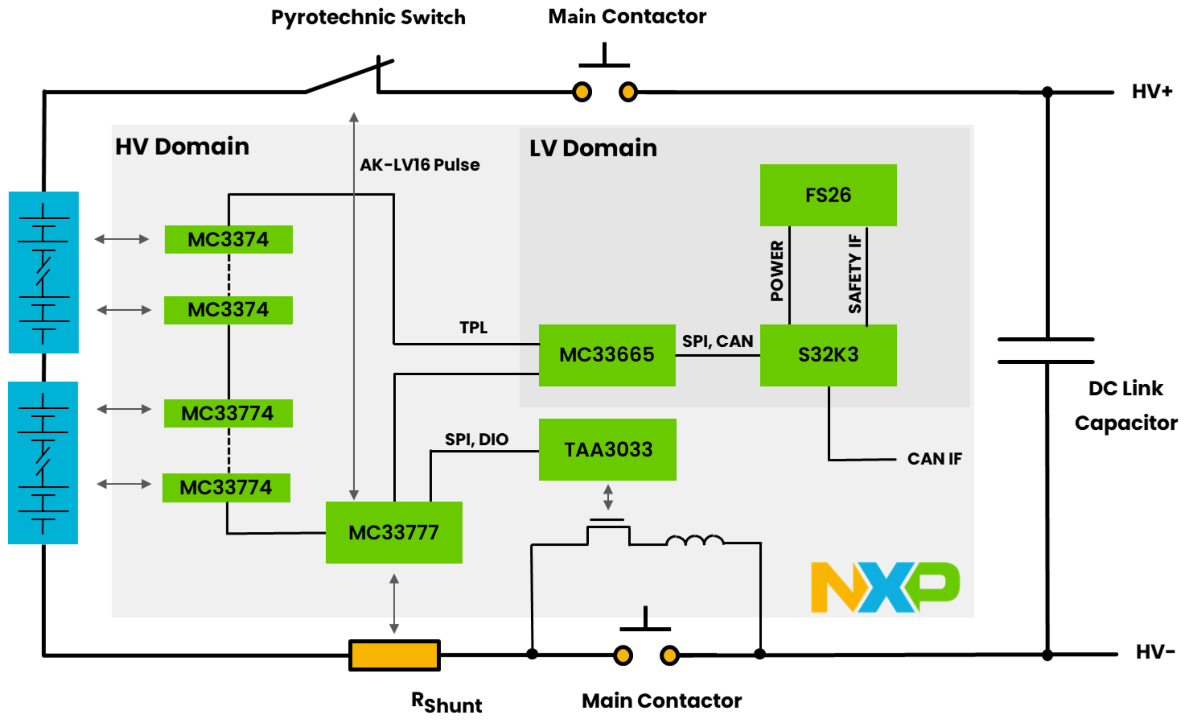


Figure 3. System Solution of a High Voltage Battery Management System

A HVBMS typically consists of three different modules. The first is a battery management unit (BMU) and its MCU which is the brain of the BMS. It makes measurement decisions, implements mechanisms of protection, and communicates with the main vehicle unit. It calculates the State of Charge (SoC) as the battery's current capacity. It is an important factor to know, how many more kilometers an EV can go before it needs to reach a charging point. Another key performance indicator (KPI) of a battery is the State of Health (SoH). It offers feedback on a battery's general health and life expectancy in comparison of the present capacity and its initial capacity after production. The second module is the cell monitoring unit (CMU) which hosts a chain of BCCs that measure cells' voltages and temperatures. It balances the cells to guarantee maximum capacity on charging the pack as otherwise the cell that first reaches over voltage (OV) would prevent further charging of other cells, while under voltage (UV) of a cell would prevent further discharge of the battery pack (refer to Figure 4).

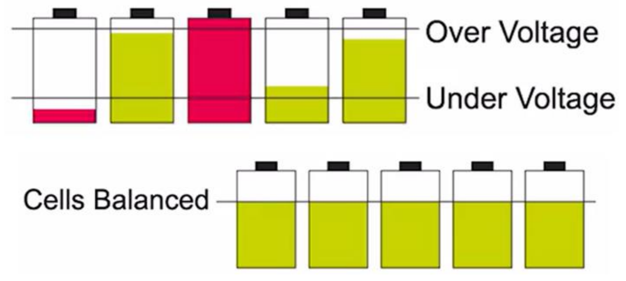


Figure 4. Balancing Cells for Optimal Charging and Discharging of a Battery Pack.

The battery junction box (BJB) is the third module and measures the battery pack's current, the pyrotechnic switch's and contactors' high voltages and the isolation between the battery and the vehicle's chassis. If the current from the battery becomes too large due to a dramatic event, the BJB fires the pyrotechnic switch that uses a small explosive very like an airbag or seatbelt tensioner to physically cut of power supply as last resort safety measurement. A safety critical feature that should not engage unintentionally. During power on, a precharge mechanism protects the main contactor from welding together due to too high inrush currents by initially charging the car's battery attached capacitors at a limited current. BJB and CMU operate in the high voltage (HV) domain which must be isolated from the low voltage (LV) domain where the BMU resides. TPL stands for NXP's Transport Protocol Link to establish communication between high voltage BMS components. It is a communication bus that offers electrical isolation with speeds of typically 2.0 Mbps. A single BCC can handle up to 18 cells with 5 Volts in series which sums up to 90 Volts [4]. Electric cars typically rely on batteries with more cells in series going up to 400 or even 800 Volts. A daisy chain of connected BCCs that monitors such cell configurations needs to be electrically decoupled through TPL. The physical connection of communication interfaces like SPI, UART, CAN or others are not an option for such a use-case without destroying the devices. The BMU makes use of a GTW to translate between the MCU's SPI and the TPL. The MCU executes the BMU's software to control everything with real time drivers (RTD) for these communication peripherals and complex device drivers (CDD) to configure and operate the BCCs of the CMU. Figure 5 illustrates the complexity of the software stack, which is based on the AUTOSAR standard.

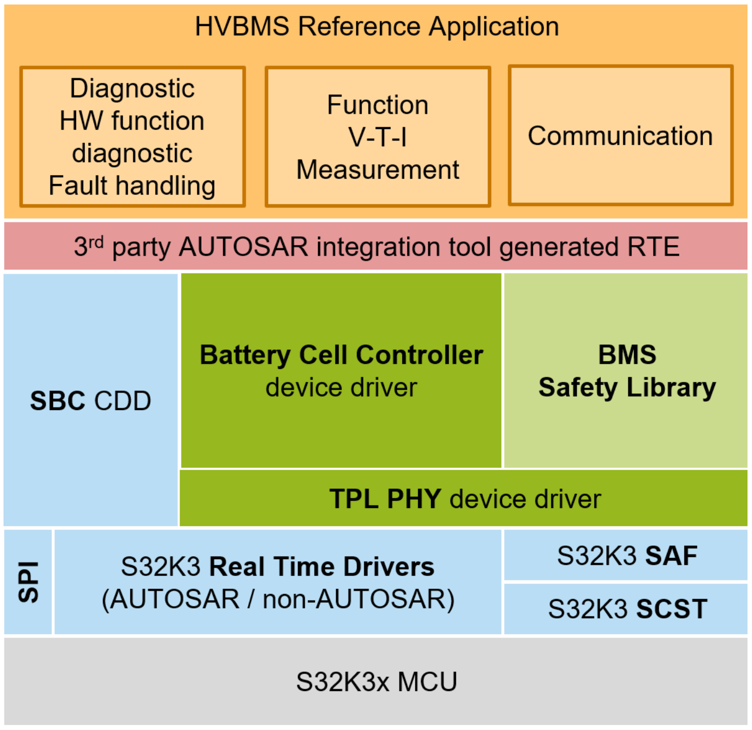

HVBMS Reference Application
Diagnostic HW function diagnostic Fault handling
Function V-T-I Measurement
Communication
3rd party AUTOSAR integration tool generated RTE
SBC CDD
Battery Cell Controller device driver
BMS Safety Library
TPL PHY device driver
SPI
S32K3 Real Time Drivers (AUTOSAR / non-AUTOSAR)
S32K3 SAF
S32K3 SCST
S32K3x MCU

Figure 5. Complete Software Stack of BMS Reference Application

In the following Chapter 3. we describe how we created an FSS of the described BMS but leaving out the BJB for the moment.

3. Full System Simulation Setup

We started with two main parts. A Synopsys VSDK of NXP's S32K3xx MCU and a BCC architectural model as we describe in [1]. The first can execute a compiled binary of any software that would also run on the real processor. Unlike a simple instruction set simulator (ISS) it includes a complete hardware perspective with registers and peripherals (GPT, UART, CAN, SPI). It is only the MCU that executes BMS software. The BCC itself is software-free but requires its configuration registers initialized by the CDD running on the MCU. Also, the MCU must read and process measuring values from the BCC's result registers. Our SysML generated SystemC architectural model of the BCC already implements all registers together with read and write accesses. Being SystemC, we made use of Synopsys TLM creator tool to set up a BCC TLM component. Please refer to Figure 6 to understand the full system setup which is still missing the BJB as introduced in Chapter 2. , an enhancement for future work.

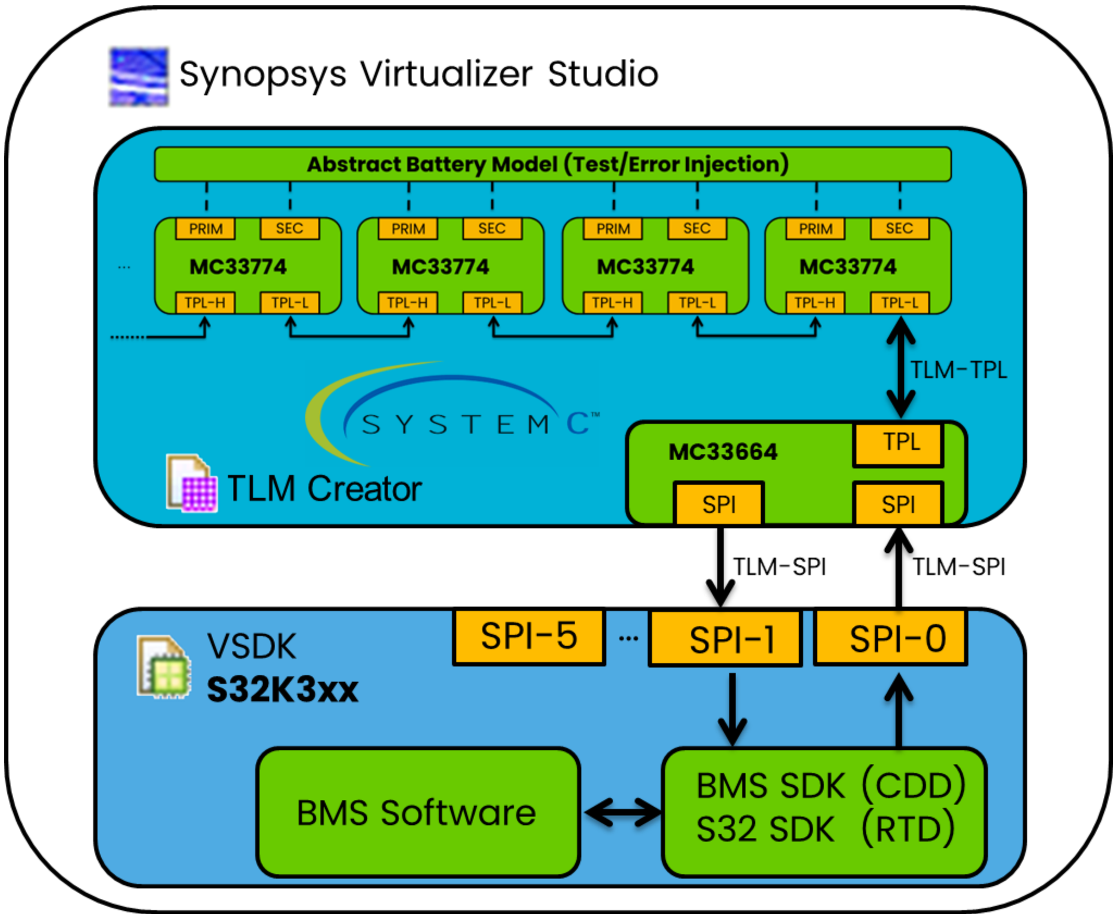


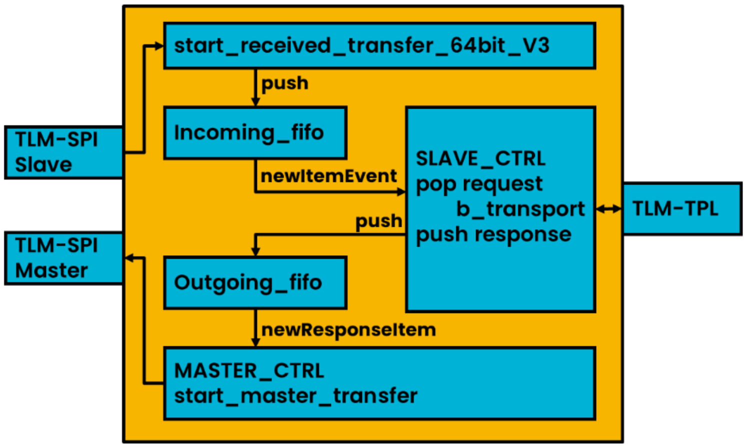
Figure 6. Full System Simulation Setup with Model of S32K3xx, MC33664 and MC33774.

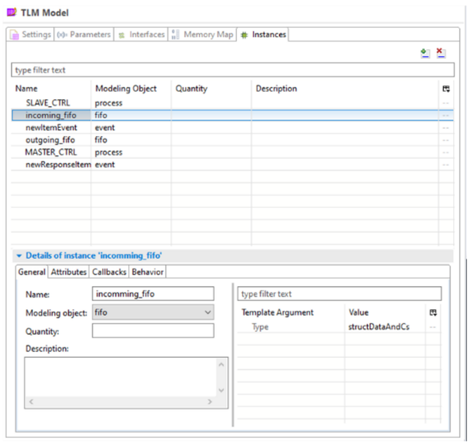
Our original architectural model of the BCC [1] comes with an interface to a python script API. The execution of test python scripts triggers the call and creation of TLM transport methods with proper payloads for our TPL communication abstraction (TLM-TPL). This enables us to configure register values of the BCC and read out measured values from a BCC standalone SystemC test-environment executing specialized python-testcases.

Just like the physical device, the simulated S32k3xx of the VSDK comes with several SPI interfaces implemented through Synopsys proprietary TLM-SPI socket[2]. We required a gateway from Synopsys TLM-SPI to NXP TLM-TPL and connect the two models. This required communication gateway is not just an artefact of simulation, but a needed functionality in the physical world, too. For this purpose, NXP offers a physical GTW device named "MC33664: Isolated Network High-Speed Transceiver" [5] or "MC33665: General Purpose BMS Communication TPL Transceiver and CAN FD Gateway" [6]. which needs to become part of the FSS to make it complete. Synopsys offers a helpful tutorial lab on how to design a SPI Controller Model [2]. This tutorial was the perfect implementation start for a

[2] tlm_ft_spi_device_socket

behavioral model of the required gateway as depicted in Figure 7. While the left part of the figure shows the architecture of the MC33664 model, the right part visualizes a screenshot of the TLM creator tool.





Name	Modeling Object	Quantity	Description
SLAVE_CTRL	process		
incoming_fifo	fifo		
newItemEvent	event		
outgoing_fifo	fifo		
MASTER_CTRL	process		
newResponseItem	event		

Name: incomming_fifo
Modeling object: fifo
Quantity:
Description:

Template Argument	Value
Type	structDataAndCs

Figure 7. TLM Model of SPI to TPL Gateway MC33664

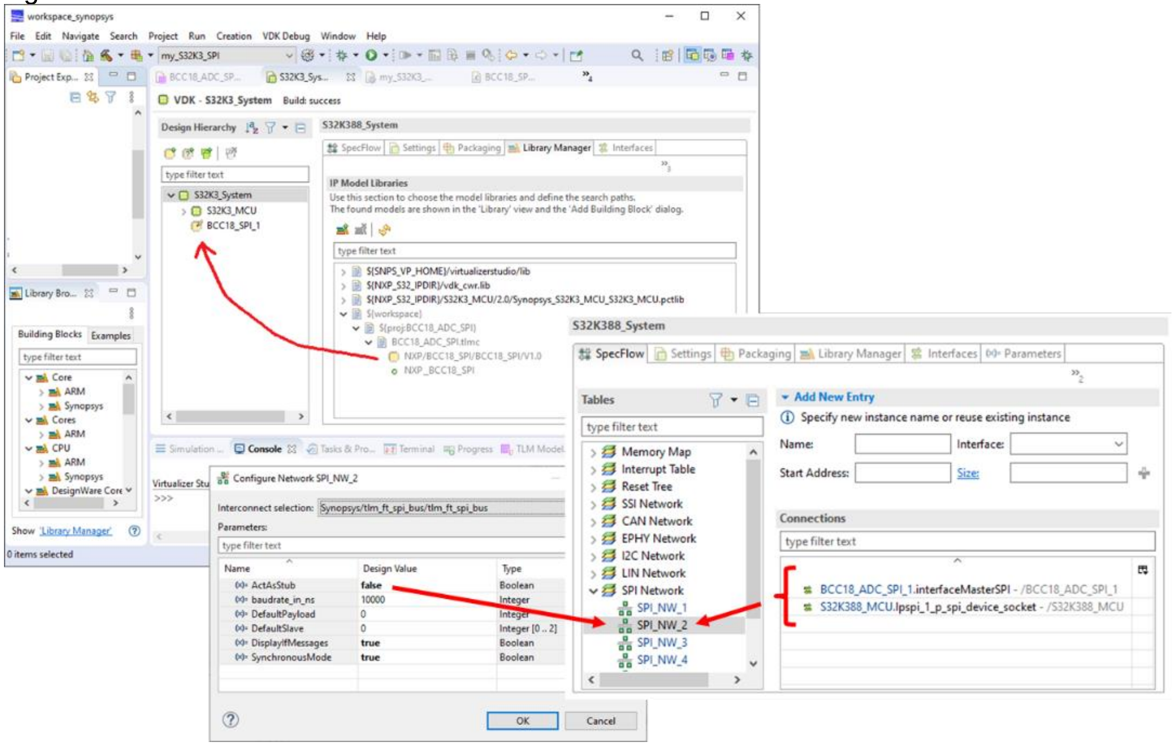
The gateway is equipped with one SPI Slave interface for request commands and one SPI Master interface for respective responses. In the TLM creator graphical usage interface (GUI) both interfaces are created in the tab “Interfaces”. Other instances are added to the model in the tab “Instances”. The “Details of instance …” tab leads to callbacks for SystemC code sections to fill in the behavior of e.g. the processes “SLAVE_CTRL” and “MASTER_CTRL”. Referring to [2] should make it quite clear how this works in all detail.

TLM creator also comes with the option of importing existing SystemC TLM Models. Being an TLM based model, it seemed obvious to use this feature to pull the architectural model of the BCC into the FSS. As stated in [1], the model is autogenerated from SysML leading to a highly structured SystemC code with many .cpp- and .h-files to import. We did not expect it to be promising to draw all of this in as one enormous chunk. Instead, we decided to achieve this in a controlled semi-manual step by step approach making use of the Synopsys Visualizer Studio python API. Whenever we were satisfied with what part of the BCC we imported last, we continued to extend the python script as outlined below for the next section of code to add. Defining approximately 300 path- and filenames by hand in the TLM creator GUI would otherwise have been a very tedious endeavor. Especially knowing that we would want to start from scratch whenever we did not like the outcome of the previous import. Outline of Synopsys Virtualizer Studio python script to setup/modify an existing TLM project:

```
# Path to BCC_REPOSITORY
bcc_repo = '<*path*>'

# Finding the correct project with name
projs = tlmcreator.get_tlm_projects()
for i in projs:
  if (i.get_eclipse_project_name() == "<*proj*>"):
      proj = i
bs = proj.get_build_specification()
bc = bs.get_configurations()[0]

# Adding compiler flags to the project
cxxFlags.append('<*flag_1*>')
cxxFlags.append('<*flag_2*>')
bc.set_cxx_flags(cxxFlags)

# Adding include information to the project
```

```
includePath.append(bcc_repo + '<*/path/SLMODEL/inc1*>')
…
includePath.append(bcc_repo + '<*/path/SLMODEL/inc90*>')
bc.set_include_path(includePath)

# Adding source information to the project
sources = []
sources.append(bcc_repo + '<*/path/SLMODEL/src/source1.cpp*>')
…
sources.append(bcc_repo + '<*/path/SLMODEL/src/source180.cpp*>')
bc.set_sources(sources)

proj.save()
```

After import and compile of our GTW plus BCC chain model, the TLM component is accessible via the Synopsys Virtualizer Studio Library Manager. The new component can be dragged and dropped into the VSDK S32K3_System as seen in Figure 8 top left. To connect the SPI sockets, one selects the SpecFlow tab and finds the proper SPI Networks (middle right). With originally only the MCU model attached to the network, it is configured as "stub" to cope with the missing communication partner. To enable transmission of TLM messages the "*ActAsStub*" flag needs to be set to false (bottom left). The FSS we created is a scalable model of a complete CMU with a configurable number of BCC instances attached as one TPL daisy chain (compare with Figure 6). TLM creator offers the concept of TLM model parameters which we used to make SPI IDs, use case selection and number of BCC instances configurable.



IP Model Libraries
Use this section to choose the model libraries and define the search paths.
The found models are shown in the 'Library' view and the 'Add Building Block' dialog.

- $(SNPS_VP_HOME)/virtualizerstudio/lib
- $(NXP_S32_IPDIR)/vdk_cwr.lib
- $(NXP_S32_IPDIR)/S32K3_MCU/2.0/Synopsys_S32K3_MCU_S32K3_MCU.pctlib
- $(workspace)
  - $(proj:BCC18_ADC_SPI)
    - BCC18_ADC_SPI.tlmc
      - NXP/BCC18_SPI/BCC18_SPI/V1.0
      - NXP_BCC18_SPI

Specify new instance name or reuse existing instance

Connections
- BCC18_ADC_SPI_1.interfaceMasterSPI - /BCC18_ADC_SPI_1
- S32K388_MCU.lpspi_1_p_spi_device_socket - /S32K388_MCU

Configure Network SPI_NW_2
Interconnect selection: Synopsys/tlm_ft_spi_bus/tlm_ft_spi_bus
Parameters:

Name	Design Value	Type
ActAsStub	false	Boolean
baudrate_in_ns	10000	Integer
DefaultPayload	0	Integer
DefaultSlave	0	Integer [0 .. 2]
DisplayIfMessages	true	Boolean
SynchronousMode	true	Boolean

Figure 8. Instantiating BCC in VSDK and Connecting SPI-TLM Network.

4. Identifying and Fixing an Issue with TLM-SPI

In the process of setting up the FSS as documented in Chapter 3. we came across an issue that involved the behavior of TLM-SPI communication. In the real world and the simulation, the MCU is coupled to the CMU via a GTW which serves as gateway between SPI and TPL. The bit length of a message frame is encoded inside the data itself and can have a length of 64, 80, 96 or 112 bits (refer to Figure 9).

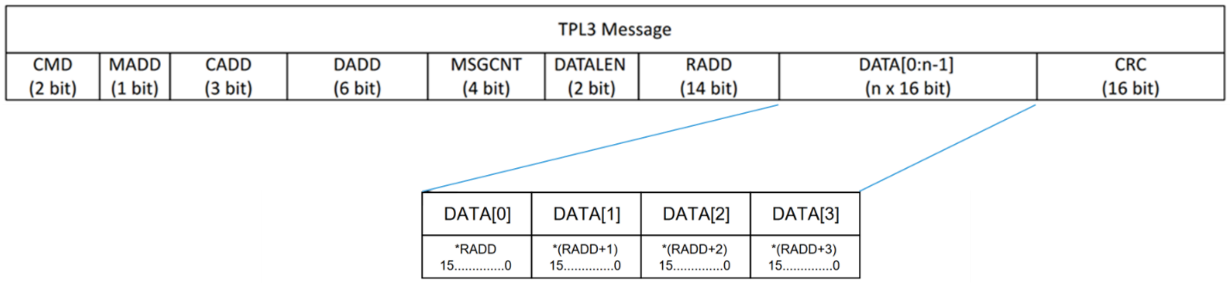


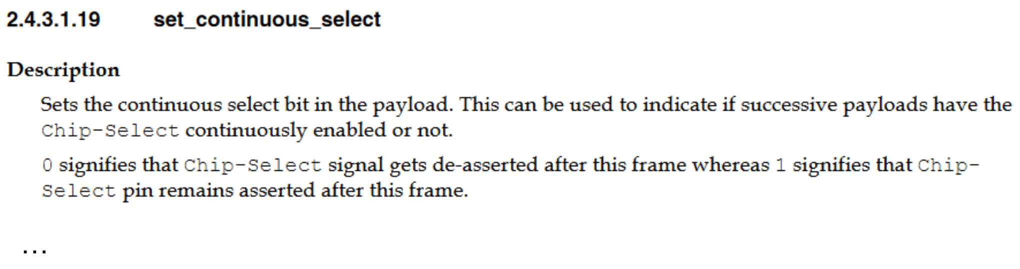
Figure 9. TPL Data Field Alignment: 64-, 80-, 96- or 112-bit Frame for One or up to Four Data Values transmitted.

The transmitted bits through the GTW are a one-by-one copy of a frame from SPI to TPL and vice versa. When we think of how this would be abstracted with TLM-SPI one might expect that the MCU's simulated SPI interface would create a single TLM-SPI message containing the complete frame in its payload. But a TLM-SPI message can carry only a max of 64 bit, one unsigned long long. The reader is refered to the documentation [3] for all details: "*bits - Number of valid data bits to be transmitted in the SPI payload. It can set from 0 to 64.*". The following shows the method's receive signature.

```
received_transfer_64bit_V3(
   unsigned long long data,
   unsigned bit_length,
   bool continuous_select
)
```

Instead, the simulated MCU sends several SPI-TLM for one message frame. The software real time device driver (RTD) handling the SPI interface is configurable in such a way, that the "*SpiDataWidth*" can be set between 1 to 64 bits. Meaningful values are 8, 16, 32 or 64 bits only. We decided to configure a value of 16-bit lengths. The simulated MCU of the VSDK is modeled to such accuracy that it then creates one TLM message for every 16 bits of data. This makes good sense as the BCC's registers are 16-bit word based and we can interpret the length of a message with respect to that as 4, 5, 6, or 7 words of 16 bit. But how can the GTW model know if the 5th word is still part of the current message frame received or already a new one? We could have implemented it in the way that we interpret the "*DATALEN*" information inside the TLM messages received so far (compare with Figure 9). But that would be inappropriate for two reasons. Firstly, semantic bitwise interpretation of the data in the payload would break the level of abstraction. The reason to use TLM is exactly to abstract away such detail of communication. Secondly, this would add functionality to the GTW model that is not there in the physical device. The GTW does not interpret the data it is sending from SPI to TPL and vice versa. Consequently, the model should not do this either.

The way to go here is to make use of the flag "*continuous select*" as to be seen from the method signature above. The Synopsys Virtuallizer Studio documentation [3] gives the following information (Figure 10).



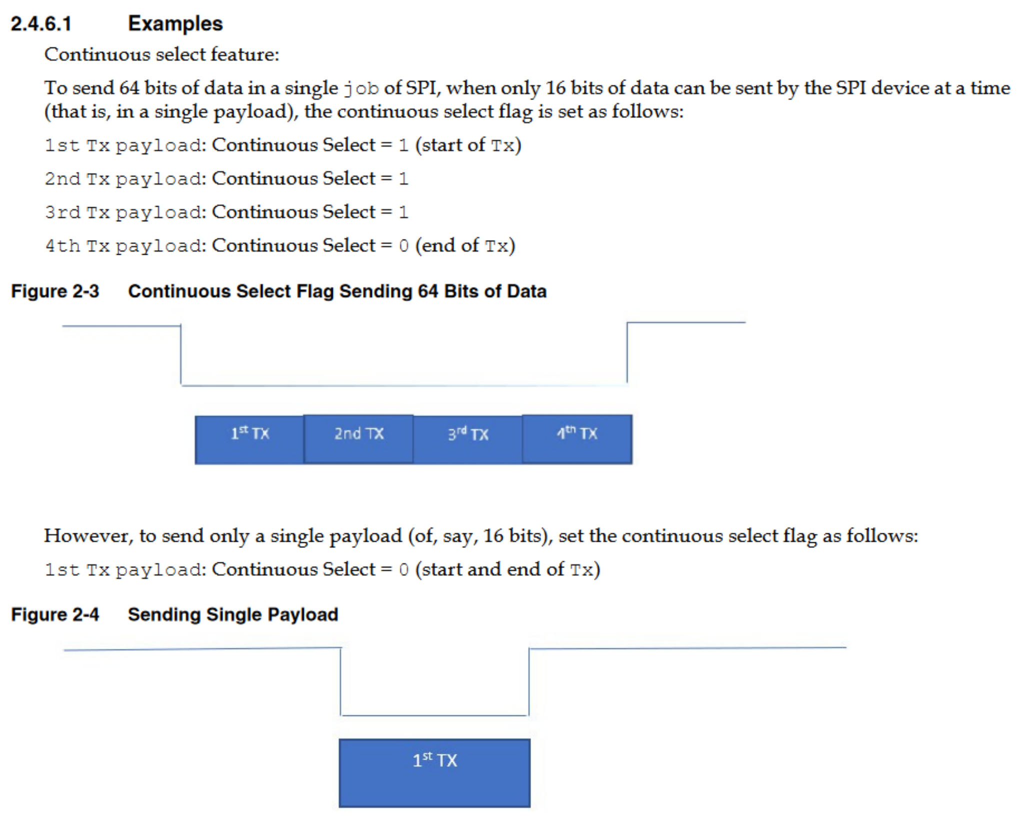
2.4.3.1.19 set_continuous_select

Description

Sets the continuous select bit in the payload. This can be used to indicate if successive payloads have the `Chip-Select` continuously enabled or not.

`0` signifies that `Chip-Select` signal gets de-asserted after this frame whereas `1` signifies that `Chip-Select` pin remains asserted after this frame.

…



2.4.6.1 **Examples**

Continuous select feature:

To send 64 bits of data in a single `job` of SPI, when only 16 bits of data can be sent by the SPI device at a time (that is, in a single payload), the continuous select flag is set as follows:

`1st Tx payload`: Continuous Select = `1` (start of `Tx`)

`2nd Tx payload`: Continuous Select = `1`

`3rd Tx payload`: Continuous Select = `1`

`4th Tx payload`: Continuous Select = `0` (end of `Tx`)

Figure 2-3 Continuous Select Flag Sending 64 Bits of Data

However, to send only a single payload (of, say, 16 bits), set the continuous select flag as follows:

`1st Tx payload`: Continuous Select = `0` (start and end of `Tx`)

Figure 2-4 Sending Single Payload

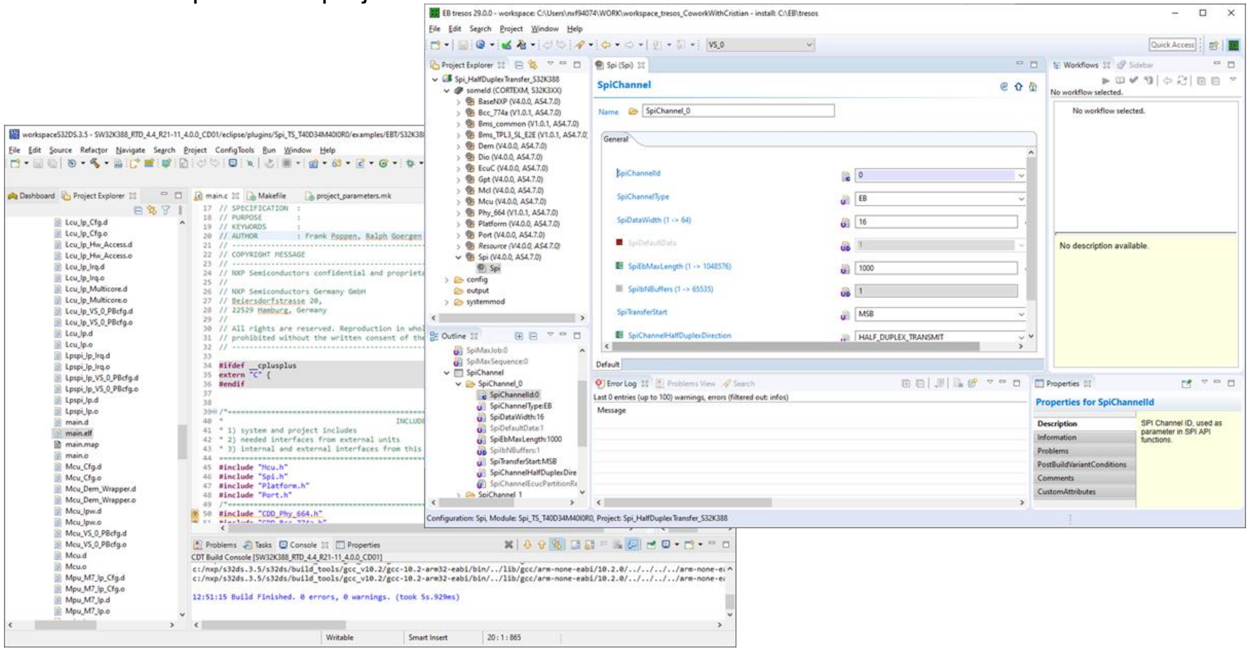
Figure 10. Description of the Continuous Select Flag.

We implemented the GTW model to collect SPI-TLM messages to the point when the continuous select flag would become ‘0’ before creating one single TPL-TLM payload and calling the b_transport of the BCC model. The GTW model would issue an error if less than four or more than seven SPI-TLM messages are received. Unfortunately, this approach was not successful right away. We would see cases where the continuous select flag is faulty. The cause of this erroneous behavior could have been the RTD software running on the simulated MCU, our newly created model of the GTW or the simulated SPI of the VSDK. We started an extensive analysis to find the cause for this. Our special thanks go to the Synopsys team who took the effort to help us track down the issue to the MCU’s SPI interface implementation. A fix for the S32K3xx VSDK was delivered by Synopsys within three working days and the FSS setup became operational.

5. Observing Complete SW Stack Execution on FSS

Looking back at Figure 5 gives some impression on the complexity of BMS software and that its development and safety validation testing is a tedious task that takes a significant amount of effort and time to complete. We are happy to report that the FSS we created in the previous chapters is accurate to the point where it can execute the complete software stack and enables a significant shift left in our product development cycle. In the following we want to document the look and feel of a simulation run and pre silicon software debugging session.

Simulation and real device operate on the same *.elf* file as executable. Meaning, that the same compiler and compiler setup is used for both cases. Our setup is based on Elektrobit (EB) Tresos as microcontroller abstraction layer (MCAL) for an AUTOSAR workflow. It generates the RTD and CDD source code configuration files of which the BMS application makes use of. The design and compilation environment for that is S32 Design Studio (S32DS) [7]. Figure 11 shows the GUIs of the two tools. The screenshot of EB Tresos symbolizes the configuration of the SPI RTD with 16 bit data width (mentioned in previous chapter 4.), while the one of S32DS depicts the generated *.elf* file after successful compile of the project.



```
// SPECIFICATION  :
// PURPOSE        :
// KEYWORDS       :
// AUTHOR         : Frank Poppen, Ralph Goergen
// ----------------------------------------------
// COPYRIGHT MESSAGE
// ----------------------------------------------
// NXP Semiconductors confidential and proprieta
//
// NXP Semiconductors Germany GmbH
// Beiersdorfstrasse 20,
// 22529 Hamburg, Germany
//
// All rights are reserved. Reproduction in whol
// prohibited without the written consent of the
// ----------------------------------------------

#ifdef __cplusplus
extern "C" {
#endif


/*==============================================
*                                        INCLUDE
* 1) system and project includes
* 2) needed interfaces from external units
* 3) internal and external interfaces from this
================================================
#include "Mcu.h"
#include "Spi.h"
#include "Platform.h"
#include "Port.h"
/*==============================================
#include "CDD_Phy_664.h"
```

```
CDT Build Console [SW32K388_RTD_4.4_R21-11_4.0.0_CD01]
c:/nxp/s32ds.3.5/s32ds/build_tools/gcc_v10.2/gcc-10.2-arm32-eabi/bin/../lib/gcc/arm-none-eabi/10.2.0/../../../../arm-none-e
c:/nxp/s32ds.3.5/s32ds/build_tools/gcc_v10.2/gcc-10.2-arm32-eabi/bin/../lib/gcc/arm-none-eabi/10.2.0/../../../../arm-none-e

12:51:15 Build Finished. 0 errors, 0 warnings. (took 5s.929ms)
```

SpiChannel

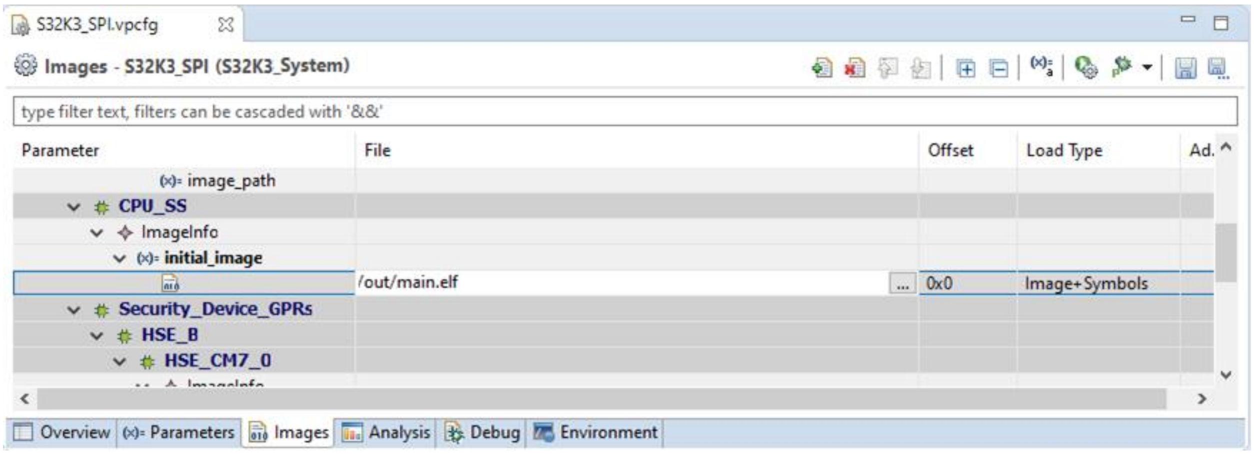
Name	SpiChannel_0
SpiChannelId	0
SpiChannelType	EB
SpiDataWidth (1 -> 64)	16
SpiDefaultData	1
SpiEbMaxLength (1 -> 1048576)	1000
SpiIbNBuffers (1 -> 65535)	1
SpiTransferStart	MSB
SpiChannelHalfDuplexDirection	HALF_DUPLEX_TRANSMIT

Configuration: Spi, Module: Spi_TS_T40D34M40I0R0, Project: Spi_HalfDuplexTransfer_S32K388

main.c
Makefile
project_parameters.mk
main.elf
Problems
Tasks
Console
Properties
Writable
Smart Insert
20 : 1 : 865
Spi (Spi)
General
Default
Error Log
Last 0 entries (up to 100) warnings, errors (filtered out: infos)
Message
Properties for SpiChannelId
SPI Channel ID, used as parameter in SPI API functions.
No workflow selected.
No description available.

Figure 11. Configuring RTD and CDD Software with EB tresos and Compiling with S32 Design Studio

Synopsys Virtualizer Studio makes use of a VP configuration concept that comes in the form of vpcfg-files. Selecting the “*Images*” tab gives the option to go for the above compiled executable for a simulation run.



S32K3_SPI.vpcfg

Images - S32K3_SPI (S32K3_System)

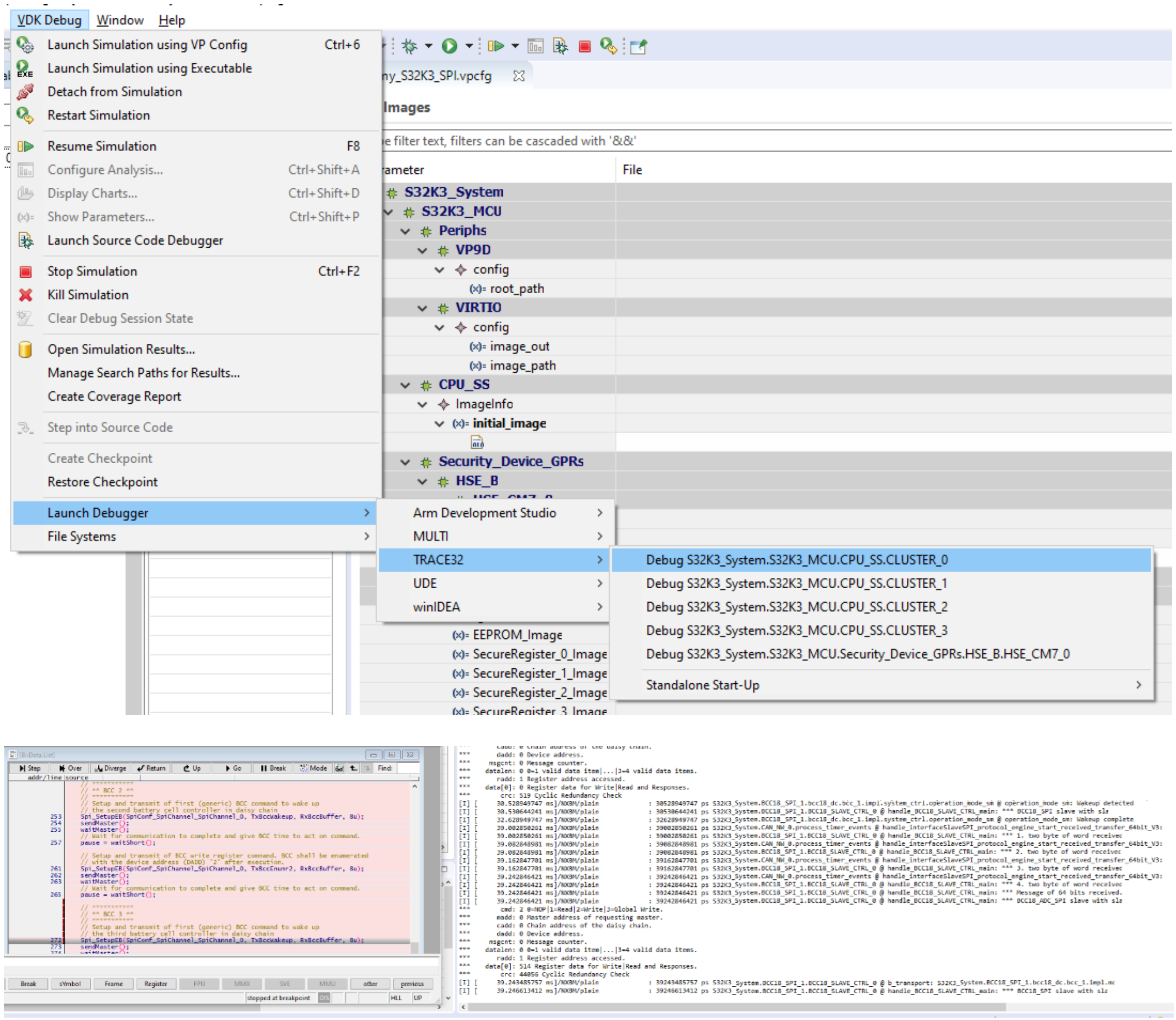
type filter text, filters can be cascaded with '&&'

Parameter	File	Offset	Load Type	Ad.
(x)= image_path				
CPU_SS				
ImageInfo				
(x)= initial_image				
	/out/main.elf	0x0	Image+Symbols	
Security_Device_GPRs				
HSE_B				
HSE_CM7_0				

Figure 12. Loading Executable .elf File on Simulated MCU. Images Ttab of VP Config (vpcfg).

The simulation starts with a click on the play button and stops right before the first processor command executes. This halt is different from that one of a physical processor. A CPU can be stopped by a debugger, but the environment cannot. In a real hardware setup, the debugger would halt but GTW, BCC and all the world’s physics around it would still be running. When the simulation stops, the simulated time itself stops. The concept of time itself halts.

A debugger can attach to the simulated MCU, or rather to any of the four cores that it provides. In our case we connect the Lauterbach TRACE32 debugger to core 0 of the simulated K3xx processor as shown in Figure 13 top part. Here we can apply break points or do anything else that debugging allows us to do, step through the code, add watchpoints, etc. (bottom part of Figure 13).



```
// ***********
// ** BCC 2 **
// ***********
// Setup and transmit of first (generic) BCC command to wake up
// the second battery cell controller in daisy chain
Spi_SetupEB(SpiConf_SpiChannel_SpiChannel_0, TxBccWakeup, RxBccBuffer, 8u);
sendMaster();
waitMaster();
// Wait for communication to complete and give BCC time to act on command.
pause = waitShort();

// Setup and transmit of BCC write register command. BCC shall be enumerated
// with the device address (DADD) '2' after execution.
Spi_SetupEB(SpiConf_SpiChannel_SpiChannel_0, TxBccEnumr2, RxBccBuffer, 8u);
sendMaster();
waitMaster();
// Wait for communication to complete and give BCC time to act on command.
pause = waitShort();

// ***********
// ** BCC 3 **
// ***********
// Setup and transmit of first (generic) BCC command to wake up
// the third battery cell controller in daisy chain
Spi_SetupEB(SpiConf_SpiChannel_SpiChannel_0, TxBccWakeup, RxBccBuffer, 8u);
sendMaster();
```

```
***     dadd: 0 Device address.
***   msgcnt: 0 Message counter.
***  datalen: 0 0=1 valid data item|...|3=4 valid data items.
***     radd: 1 Register address accessed.
***  data[0]: 0 Register data for Write|Read and Responses.
***      crc: 519 Cyclic Redundancy Check
[I] [    30.528949747 ms]/NXBM/plain            : 30528949747 ps S32K3_System.BCC18_SPI_1.bcc18_dc.bcc_1.impl.system_ctrl.operation_mode_sm @ operation_mode_sm: Wakeup detected
[I] [    30.530644241 ms]/NXBM/plain            : 30530644241 ps S32K3_System.BCC18_SPI_1.BCC18_SLAVE_CTRL_0 @ handle_BCC18_SLAVE_CTRL_main: *** BCC18_SPI slave with sla
[I] [    32.628949747 ms]/NXBM/plain            : 32628949747 ps S32K3_System.BCC18_SPI_1.bcc18_dc.bcc_1.impl.system_ctrl.operation_mode_sm @ operation_mode_sm: Wakeup complete
[I] [    39.002850261 ms]/NXBM/plain            : 39002850261 ps S32K3_System.CAN_NW_0.process_timer_events @ handle_interfaceSlaveSPI_protocol_engine_start_received_transfer_64bit_V3:
[I] [    39.002850261 ms]/NXBM/plain            : 39002850261 ps S32K3_System.BCC18_SPI_1.BCC18_SLAVE_CTRL_0 @ handle_BCC18_SLAVE_CTRL_main: *** 1. two byte of word receivec
[I] [    39.082848981 ms]/NXBM/plain            : 39082848981 ps S32K3_System.CAN_NW_0.process_timer_events @ handle_interfaceSlaveSPI_protocol_engine_start_received_transfer_64bit_V3:
[I] [    39.082848981 ms]/NXBM/plain            : 39082848981 ps S32K3_System.BCC18_SPI_1.BCC18_SLAVE_CTRL_0 @ handle_BCC18_SLAVE_CTRL_main: *** 2. two byte of word receivec
[I] [    39.162847701 ms]/NXBM/plain            : 39162847701 ps S32K3_System.CAN_NW_0.process_timer_events @ handle_interfaceSlaveSPI_protocol_engine_start_received_transfer_64bit_V3:
[I] [    39.162847701 ms]/NXBM/plain            : 39162847701 ps S32K3_System.BCC18_SPI_1.BCC18_SLAVE_CTRL_0 @ handle_BCC18_SLAVE_CTRL_main: *** 3. two byte of word receivec
[I] [    39.242846421 ms]/NXBM/plain            : 39242846421 ps S32K3_System.CAN_NW_0.process_timer_events @ handle_interfaceSlaveSPI_protocol_engine_start_received_transfer_64bit_V3:
[I] [    39.242846421 ms]/NXBM/plain            : 39242846421 ps S32K3_System.BCC18_SPI_1.BCC18_SLAVE_CTRL_0 @ handle_BCC18_SLAVE_CTRL_main: *** 4. two byte of word receivec
[I] [    39.242846421 ms]/NXBM/plain            : 39242846421 ps S32K3_System.BCC18_SPI_1.BCC18_SLAVE_CTRL_0 @ handle_BCC18_SLAVE_CTRL_main: *** Message of 64 bits received.
[I] [    39.242846421 ms]/NXBM/plain            : 39242846421 ps S32K3_System.BCC18_SPI_1.BCC18_SLAVE_CTRL_0 @ handle_BCC18_SLAVE_CTRL_main: *** BCC18_ADC_SPI slave with sla
***      cmd: 2 0=NOP|1=Read|2=Write|3=Global Write.
***     madd: 0 Master address of requesting master.
***     cadd: 0 Chain address of the daisy chain.
***     dadd: 0 Device address.
***   msgcnt: 0 Message counter.
***  datalen: 0 0=1 valid data item|...|3=4 valid data items.
***     radd: 1 Register address accessed.
***  data[0]: 514 Register data for Write|Read and Responses.
***      crc: 44056 Cyclic Redundancy Check
[I] [    39.243485757 ms]/NXBM/plain            : 39243485757 ps S32K3_System.BCC18_SPI_1.BCC18_SLAVE_CTRL_0 @ b_transport: S32K3_System.BCC18_SPI_1.bcc18_dc.bcc_1.impl.mc
[I] [    39.246613412 ms]/NXBM/plain            : 39246613412 ps S32K3_System.BCC18_SPI_1.BCC18_SLAVE_CTRL_0 @ handle_BCC18_SLAVE_CTRL_main: *** BCC18_SPI slave with sla
```

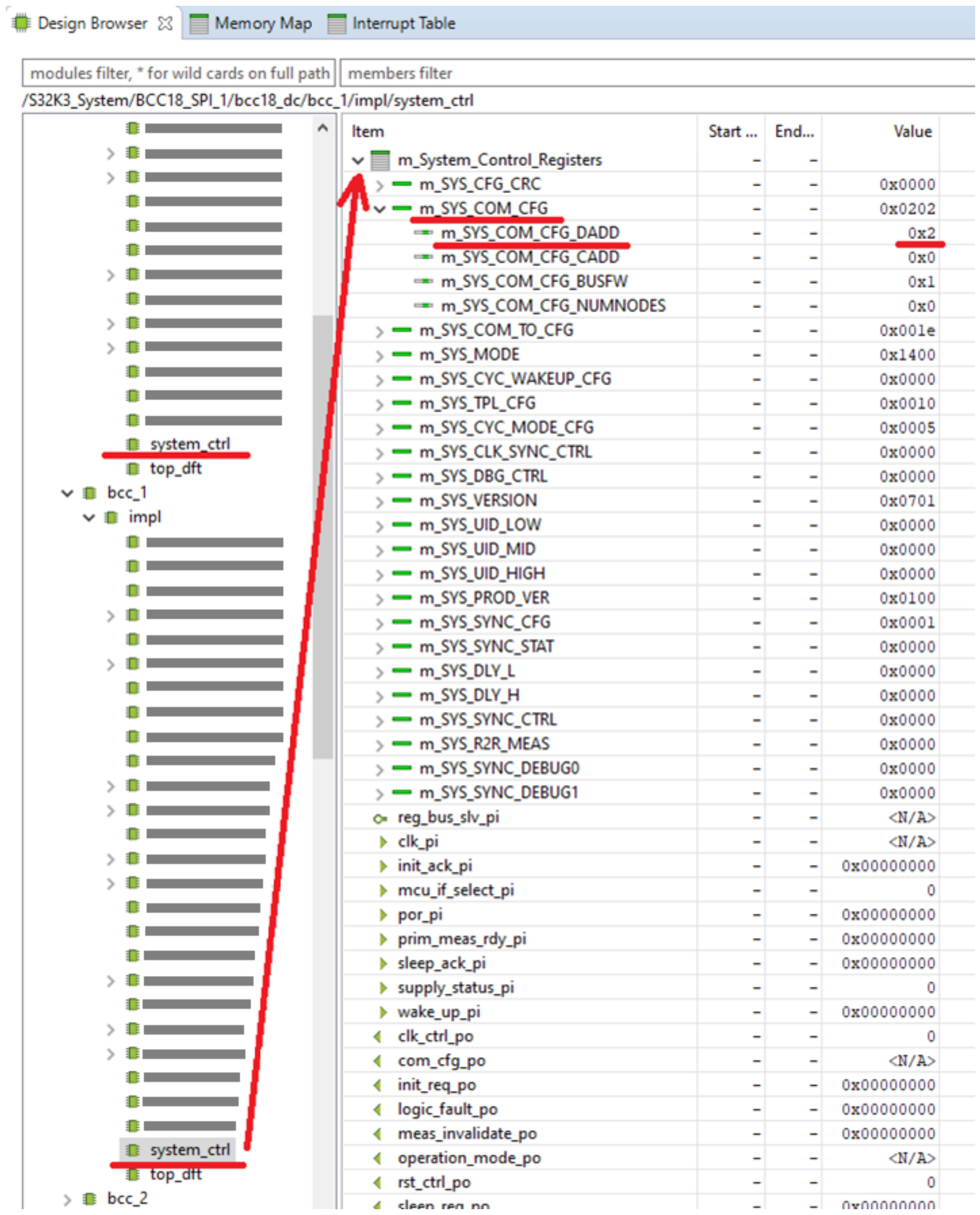
Figure 13. Attaching Lauterbach TRACE32 Debugger to Simulated Core 0 of Simulated MCU.

Additionally, a virtual platform gives full visibility to the state of the simulated HW. Figure 14 and Figure 15 depict the introspection potential that gives us full observability and traceability of all BCC's configuration registers content. The use of Synopsys proprietary SystemC Modeling Library (SCML) in the model is required to have this feature available in Synopsys Virtualizer Studio. Our model generation flow of [1] automatically makes use of the library to instantiate all model's registers and bitfields. SCML source code is available and SCML based models can run standalone without Synopsys tools. Still, we consider this proprietary dependency a drawback. SCML is not an open standard, and we currently are evaluating other options that help us to keep single vendor dependencies at a minimum.

The first step in the startup of a BMS with several BCC devices in a TPL daisy chain (CMU) is enumeration. After power on, all BCC's system communication configuration register's bitfield device address (DADD) contains the value zero. Such an unconfigured device acts on any message that it receives, independently of the DADD in a TPL message. Firstly, the device must be awakened, though. Any activity on the TPL bus does this e.g., a NOP. A second message during initialization of the BMS software stack is a TPL write command to the SYS_COM register address (RADD) with the proper enumeration value. After enumeration it will only respond to such messages that match with its configured DADD. Otherwise, the message is sent further down the chain. The following pseudo code gives the idea off this.

```
for devNum=1 to 5 do (
  nopTPL(DADD=0);
  writeTPL(DADD=0, RADD=0x0001, val=devNum);
  devNumCheck=readTPL(DADD=devNum, RADD=0x0001);
  if devNumCheck == devNum -> OK
  else NOK

)
```

The proper initialization is directly observable in Synopsys Virtualizer Studio as shown in Figure 14. Just to point out one of the many features, the tool's Analysis enables a timely tracing of values. In Figure 15 one can nicely see how enumeration ripples down the TPL chain.



/S32K3_System/BCC18_SPI_1/bcc18_dc/bcc_1/impl/system_ctrl

```
[redacted]
system_ctrl
top_dft
bcc_1
  impl
    [redacted]
    system_ctrl
    top_dft
bcc_2
```

Item	Start ...	End...	Value
m_System_Control_Registers	-	-	
m_SYS_CFG_CRC	-	-	0x0000
m_SYS_COM_CFG	-	-	0x0202
m_SYS_COM_CFG_DADD	-	-	0x2
m_SYS_COM_CFG_CADD	-	-	0x0
m_SYS_COM_CFG_BUSFW	-	-	0x1
m_SYS_COM_CFG_NUMNODES	-	-	0x0
m_SYS_COM_TO_CFG	-	-	0x001e
m_SYS_MODE	-	-	0x1400
m_SYS_CYC_WAKEUP_CFG	-	-	0x0000
m_SYS_TPL_CFG	-	-	0x0010
m_SYS_CYC_MODE_CFG	-	-	0x0005
m_SYS_CLK_SYNC_CTRL	-	-	0x0000
m_SYS_DBG_CTRL	-	-	0x0000
m_SYS_VERSION	-	-	0x0701
m_SYS_UID_LOW	-	-	0x0000
m_SYS_UID_MID	-	-	0x0000
m_SYS_UID_HIGH	-	-	0x0000
m_SYS_PROD_VER	-	-	0x0100
m_SYS_SYNC_CFG	-	-	0x0001
m_SYS_SYNC_STAT	-	-	0x0000
m_SYS_DLY_L	-	-	0x0000
m_SYS_DLY_H	-	-	0x0000
m_SYS_SYNC_CTRL	-	-	0x0000
m_SYS_R2R_MEAS	-	-	0x0000
m_SYS_SYNC_DEBUG0	-	-	0x0000
m_SYS_SYNC_DEBUG1	-	-	0x0000
reg_bus_slv_pi	-	-	<N/A>
clk_pi	-	-	<N/A>
init_ack_pi	-	-	0x00000000
mcu_if_select_pi	-	-	0
por_pi	-	-	0x00000000
prim_meas_rdy_pi	-	-	0x00000000
sleep_ack_pi	-	-	0x00000000
supply_status_pi	-	-	0
wake_up_pi	-	-	0x00000000
clk_ctrl_po	-	-	0
com_cfg_po	-	-	<N/A>
init_req_po	-	-	0x00000000
logic_fault_po	-	-	0x00000000
meas_invalidate_po	-	-	0x00000000
operation_mode_po	-	-	<N/A>
rst_ctrl_po	-	-	0
sleep_req_po	-	-	0x00000000

Figure 14. Browsing BCC's Software Configured Hardware Registers.

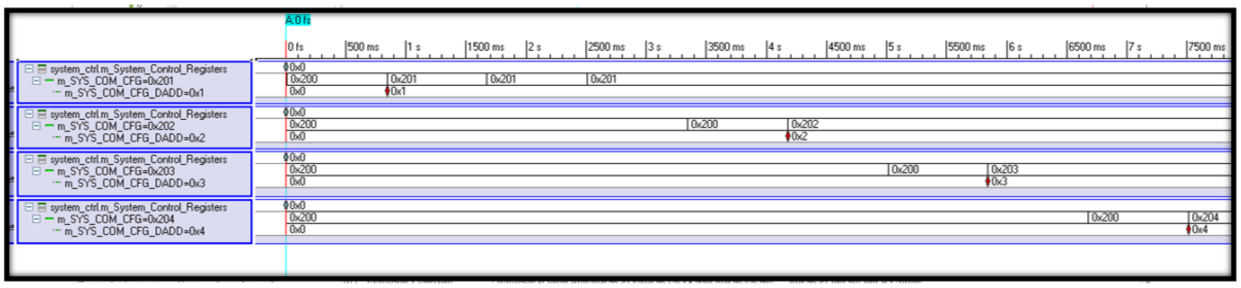
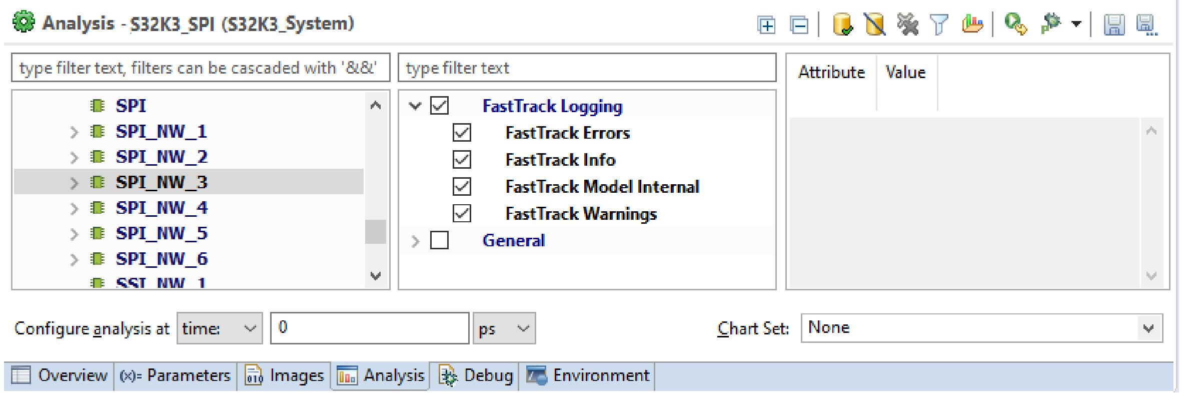


Figure 15.Vpcfg Analyses Tracing Device Enumeration Value Changes in CMU over Time.

Analysis can also be used to configure the verbosity of messages. During the setup of the FSS we e.g. had the issue that SPI communication was unsuccessful. First issue was that the SPI network was falsely configured as stub (compare with Figure 8 of Chapter 3.) Another issue was that SPI slave device ids were configured incorrectly. In a physical setup one would probably have set up a logic analyzer to have a look at the SPI bus signals and the communicated values. With the FSS available, enabling FastTrack Logging (Figure 16) for the SPI network was sufficient to understand the issue: a faultily configured SPI slave id 0.



- [✓] FastTrack Logging
  - [✓] FastTrack Errors
  - [✓] FastTrack Info
  - [✓] FastTrack Model Internal
  - [✓] FastTrack Warnings
- [ ] General

Figure 16. Enabling FastTrack Logging on SPI Network for Versatily Messages on Interface Status.

```
SPI_NW_3Model Internal[Legacy Logging]Trying to transmit... slave id
is  0
SPI_NW_3Warning[Generic Warning]Trying to transmit from master - No
such slave with id : 0 exists...
```

6. Conclusions and Future Work

The usage experience and the received results from our setup FSS is very promising to the point of convincing. While we were working together with RTD and CDD SW developers to get everything up and running, their involvement triggered questions like “when will I be able to make use of this in my customer projects?” and statements like “this would so help me to clean out our HW laboratory space.” Albeit to get there, some further work must be done. As stated previously, we automated the generation of the complete register set of devices from SysML models. The functionality behind registers in the model must be coded in SystemC manually and is not yet complete for all parts of the MC33774. It is yet missing completely for other variants of BCC, BJB and GTW. The practical results achieved in this work justify the investment of further resources in the topic to achieve complete FSS for which we expect a lot of reuse potential between the required SystemC models.

Another potential topic to add would be models of battery cells of different accuracy and potentially their chemistry. In this work we made use of a very simple model that delivers constant voltage values per cell as input for measurement. The model does not respond to or act on inputs it receives. It is an option to connect simulation environments like those from Mathworks Matlab/Simulink with the FSS. Sophisticated battery models are available for these or can be created that can respond to external parameters like load, current, etc. This would make the interaction and effect of SoC and SoH algorithms observable during research and development.

An idea to mention for future work is the setup of our models as models in the loop (MIL) in an emulation sort of scenario. The SystemC executes in a small box with sufficient SW compute power to run it and a HW TPL interface to connect it to physical BMS. We executed first experiments with Raspberry Pis 4 and 5 that encouraged us to proceed this path with further work.

SystemC and TLM are open standards unlike Synopsys SCML. As stated before, our BCC model can run standalone without Synopsys tooling but requires the proprietary SCML. We are in the process of evaluating other options like VCML to name just one. Ideally, Synopsys would be willing to open up SCML towards an open-source standard. Maybe this work can convince decision makers to do so.

7. Glossary

ASIL	Automotive Safety Integration Level
BCC	Battery Cell Controller
BJB	Battery Junction Box
BMS	Battery Management System
BMU	Battery Management Unit
BOM	Bill of Material
CADD	Chain Address
CAN	Controller Area Network
CDD	Complex Device Driver
CMD	Command
CMU	Cell Monitoring Unit
CRC	Cyclic Redundancy Check
DADD	Device Address
ELF	Executable and Linkable Format
EV	Electric Vehicle
FSS	Full System Simulation
GPT	General Purpose Timer
GTW	Gateway
GUI	Graphical Usage Interface
HV	High Voltage
HVBMS	High Voltage Battery Management System
HW	Hardware
KPI	Key Performance Indicator
LV	Low Voltage
MADD	Master Address
MCAL	Microcontroller Abstraction Layer
MCU	Microcontroller Unit
MIL	Model in the Loop
MSGCNT	Message Count
OV	Over Voltage
RADD	Register Address
RTD	Real Time Device Driver

RTL	Register Transfer Level
SCML	SystemC Modeling Library
SoC	State of Charge
SoH	State of Health
SPI	Serial Peripheral Interface
SW	Software
SysML	Systems Modeling Language
TLM	Transaction Level Model
TPL	Transport Protocol Link
UART	Universal Asynchronous Receiver/Transmitter
UV	Under Voltage
VCML	Virtual Components Modeling Library
VSDK	Virtualizer Studio Development Kit

8. References

[1] R. Görgen, E. de Kock, “SysML based Architecture Definition and Platform Generation Flow”, Design and Verification Conference and Exhibition Europe 2019.

[2] “TLM Creation Quickstart Training, Module 2: Building a SPI Controller Model”, Synopsys Solvnet, TLMCreation_Quickstart_Training_Q2020.06-2

[3] “tlm_ft_spi_bus”, Synopsys Solvnet, U-2023.03-2

[4] “MC33774: 18 Channel Li-Ion Battery Cell Controller IC ASIL D”, www.nxp.com

[5] “MC33664: Isolated Network High-Speed Transceiver”, www.nxp.com

[6] “MC33665: General Purpose BMS Communication TPL Transceiver and CAN FD Gateway”, www.nxp.com

[7] “S32 Design Studio IDE”, www.nxp.com