Thomas Hämmerle, BSc

Hardware Abstraction Layer (HAL) for an automotive real-time multicore operating system

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Electrical Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Bernd Eichberger

Institute of Electronic Sensor Systems

Dipl.-Ing. Dr.techn. Eduard Unger, AVL List GmbH

Graz, March 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

date	(signature)

Acknowledgments

I would like to thank my advisor Ass.Prof. Dipl.-Ing. Dr.techn. Bernd Eichberger for his support throughout the course of my thesis.

Furthermore, I would like to thank Dipl.-Ing. Dr.techn. Eduard Unger, for his excellent guidance during my work at AVL List GmbH and the outstanding support during the work on my thesis. I also really appreciated the help of my colleagues and the pleasant working environment.

I would like to thank my family and friends for their encouragement. I am especially grateful to my parents, who not only gave me the chance of doing my Masters degree but most of all motivated me not to take the easy route and to get the best out of myself.

Thomas Hämmerle
Graz, March 2017

Abstract

One of the most important tools in the area of engine research and development is a performant and high flexible electronic control unit for internal combustion engines. The increasing complexity of tasks, placed to electronic control units creates new requirements and demands more real-time performance.
AVL Rapid Prototyping Engine Management System (EMS) represents such a generic engine controller. It is provided to customers of AVL as a highly flexible and configurable engine management system for the operation of all kind of combustion engines during performance and emission development.
To meet the demands of the future, the performance of the next EMS generation is further improved by replacing the current singlecore microcontroller by a state-of-the-art multicore microcontroller. Through increased capabilities floating point format calculations are supported, as well as the distribution of complex processing tasks on different cores.

In the course of this master's thesis, an application specific Hardware Abstraction Layer (HAL) for a real-time operating system was established, forming the basis for the future EMS software.

This work presents the implementation of the Hardware Abstraction Layer including the Low Level Drivers for a multicore microcontroller. Used concepts and functionalities of real-time operating systems are discussed on the basis of the automotive operating system ETAS RTA-OS5.

Kurzfassung

Eines der wichtigsten Werkzeuge im Bereich der Motorenforschung und -entwicklung ist eine leistungsfähige und hochflexible elektronische Steuereinheit für Verbrennungsmotoren. Die zunehmende Komplexität der Aufgaben, die an die elektronischen Steuergeräte gestellt werden, schafft neue Anforderungen und verlangt mehr Echtzeit-Performance.
AVL Rapid Prototyping Engine Management System (EMS) stellt eine solche generische Motorsteuerung dar. Sie wird Kunden von AVL als hochflexibles und konfigurierbares Motormanagementsystem für den Betrieb verschiedener Verbrennungsmotoren bei Leistungs- und Emissionsentwicklung zur Verfügung gestellt.
Um den zukünftigen Anforderungen gerecht zu werden, wird die Performance der nächsten EMS-Generation weiter gesteigert, indem der aktuelle Singlecore-Mikrocontroller durch einen modernen Multicore-Mikrocontroller ersetzt wird. Durch die gesteigerte Leistungsfähigkeit werden Fließkomma-Format-Berechnungen sowie die Verteilung komplexer Bearbeitungsaufgaben auf verschiedene Kerne unterstützt.

Im Zuge dieser Masterarbeit wurde eine anwendungsspezifische Hardware Abstraktionsschicht (HAL) für ein Echtzeitbetriebssystem entwickelt, welche die Basis für die zukünftige EMS-Software bildet.

Diese Arbeit stellt die Implementierung der Hardware Abstraktionsschicht einschließlich der Low Level Treiber für einen Multicore Mikrocontroller vor. Verwendete Konzepte und Funktionalitäten von Echtzeitbetriebssystemen werden anhand des Automotive-Betriebssystems ETAS RTA-OS5 diskutiert.

Contents

List of Abbreviations

ADC Analog-to-Digital-Converter

ARU Advanced Routing Unit

ASC Asynchronous/Synchronous Communication

ASW Application Software

ATOM ARU-connected Timer Output Module

AUTOSAR AUTomotive Open System ARchitecture

AVL Anstalt für Verbrennungskraftmaschinen List GmbH

BSW Basic Software

CAN Controller Area Network

CFGU Configurable Clock Generation Subunit

CMU Clock Management Unit

CPU Central Processing Unit

ECU Engine Control Unit

EGU External Clock Generation Subunit

ELF Executable and Linking Format

EMS Engine Management System

EPM Engine Position Management

FIFO Firt In First Out

FXU Fixed Clock Generation

GTM Generic Timer Module

HAL Hardware Abstraction Layer

IC Integrated Circuit

IDE Integrated Development Environment

IFX Infineon Technologies AG

iLLD	Infineon Low Level Driver
IP	Intellectual Property
ISR	Interrupt Service Routine
JTAG	Joint Test Group Array
LSB	Least Significant Bit
MSB	Most Significant Bit
MSC	Microsecond Channel
MUTEX	mutual exclusion
OS	Operating System
PCB	Printed Circuit Board
PCP	Peripheral Control Processor
RTOS	Real-time operating system
SFR	Special Function Register
SPI	Serial Peripheral Interface
TIM	Timer Input Module
TOM	Timer Output Module
VADC	Versatile Analog-to-Digital Converter

List of Figures

1. Introduction

1.1. AVL (Rapid Prototyping) Engine Management System (EMS)

AVL (Rapid Prototyping) Engine Management System (EMS) is a generic engine controller and is provided to customers of AVL as a highly flexible and configurable engine management system for the operation of all kind of combustion engines during performance and emission development. While there is one basic hardware setup of the EMS, there exists variants for the management of different engines (like direct injection turbo charges gasoline engines, common rail turbo charged diesel engines, etc.) that differ in the type and number of dedicated sensor and actuator interfaces. Figure 1.1 gives an overview of the structure of the EMS. Sensors signals are connected via different interfaces to the core of the engine management system (EMS). Two of them are unidirectional (from sensor to core) and the other interfaces are bidirectional (Controller Area Network (CAN) connected intelligent sensors, etc.). Based on the signals from the different interfaces, the engine is controlled by different actuators, illustrated on the right hand side of figure 1.1.

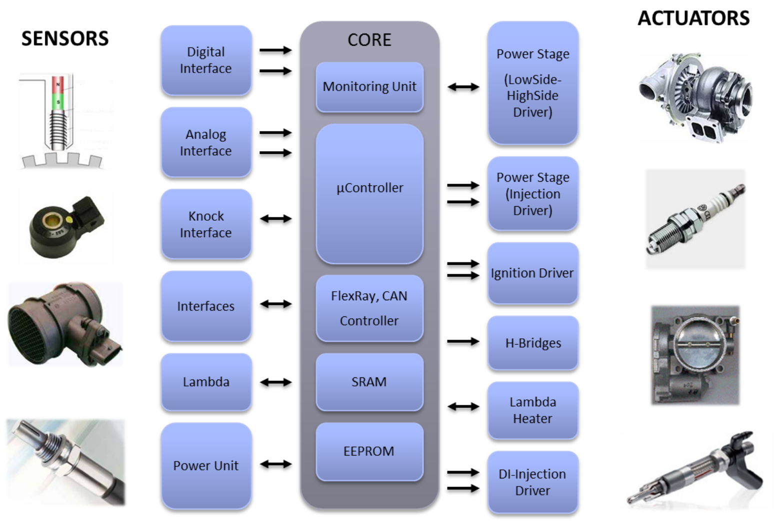


Figure 1.1.: Overview of AVL EMS[1]

[1]`www.avl.com`: *RPEMS - Rapid Prototyping Engine Management System* 2017

1.1.1. Hardware architecture

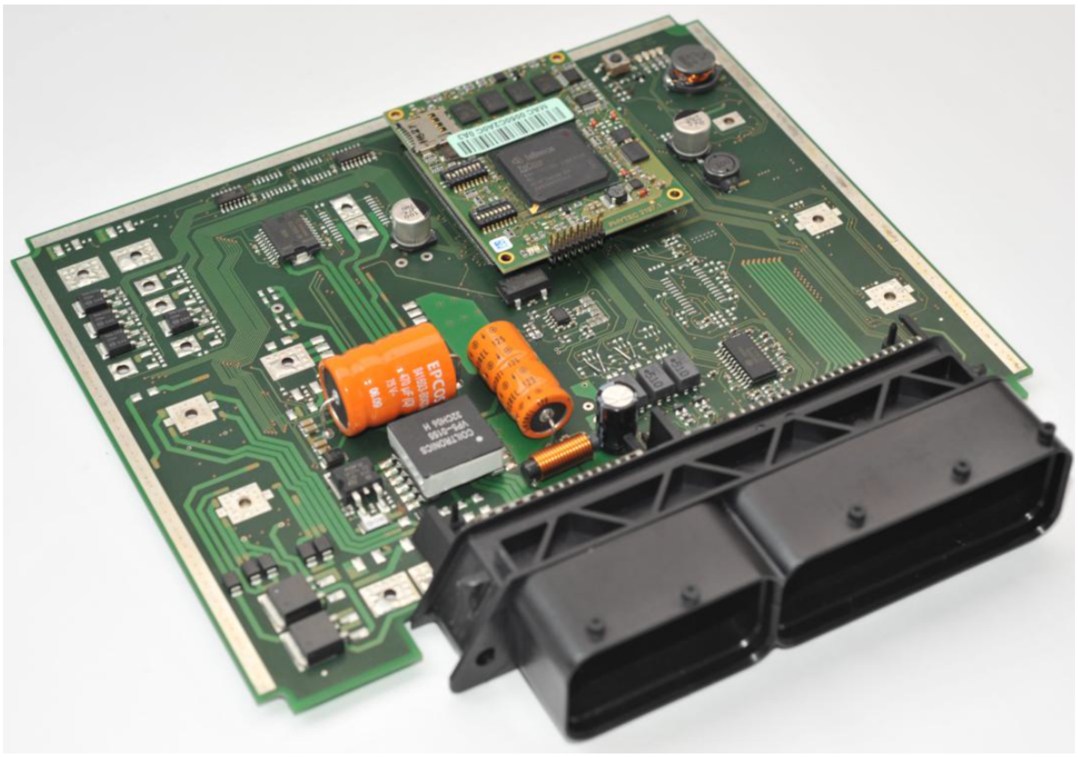


Figure 1.2.: AVL EMS main board with processor board[2]

The current version of all AVL EMS derivates is operated by a processor board equipped with a single core microcontroller. This processor board is connected to the main board providing power supply, signal conditioning for sensor inputs, power drivers for actuators and all necessary bus transceivers, necessary for power train management. Figure 1.2 shows a partially populated AVL EMS main board with a connected processor board.
The central element of the processor board is the single core processor TC1793 by Infineon.

1.1.2. Software architecture

The EMS software architecture is characterized by its layered design supported by real-time operating system. The software is based on an application layer and a hardware abstraction layer (HAL). While the application layer integrates all main functionalities of the EMS, the HAL provides access to the hardware resources. This fundamental layer is divided into HAL-layer 1 and HAL-layer 2. Layer 1 handles the access to peripheral hardware, using dedicated on-chip interfaces. Layer 2 abstracts the data and functions provided by or rather passed to

[2]Eichberger and Unger, 2012, page 4

layer 1 by means of the conversion of raw input/output data into physical unit values and vice versa. Figure 1.3 shows an overview of the EMS software layer concept.

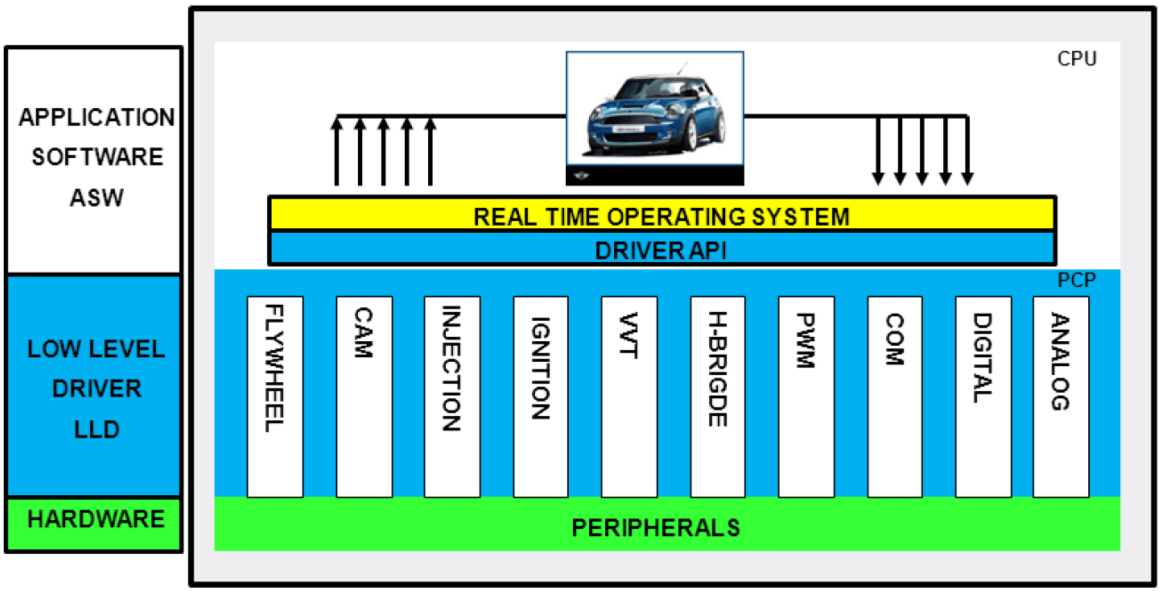


Figure 1.3.: Overview of AVL EMS software concept[3]

For unloading the central processing unit (CPU) of the TC1793, time critical tasks are managed by the peripheral control processor (PCP) of the TC1793.
This lean and modular software design is a prerequisite for a rapid prototyping engine management system. In 4.2 the implementation of the software is explained in detail.

[3]Eichberger and Unger, 2012, page 7

1.2. Technical goal

The basis for the usage of a multicore microcontroller in AVL EMS shall be provided.
This includes following aspects:

Development and implementation of an application specific Hardware Abstraction Layer (HAL) for IFX AURIX TC277T. This HAL is a fundamental part of the EMS BSW (Basic SW) and is used in combination with the EMS ASW (Application SW) to support rapid prototyping algorithm development for automotive powertrain control.
Configuration and adaptation of BSW is mainly a task related to new or changed HW components on the PCB - during ASW development it's not intended to do HAL or BSW modifications. A seamless usability as well as flexibility of the functions provided by the HAL and integrated device drivers has to be guaranteed. The reuse of already existing and validated low level drivers from IFX TRICORE TC1793 and AURIX iLLD is mandatory for this project.
Interface drivers to support SPI and MSC communication to different ASICS on the PCB as well as a CAN driver are main bricks of the AURIX HAL.

2. Utilized hardware

2.1. TriBoard TC2X7

During development of the device drivers and the hardware abstraction layer, an IFX TriBoard TC2X7 populated with a TC277T in BGA-292 package was used. The discussion of the evaluation board summarizes the description in the manual (Infineon, 2013).
The TriBoard includes the necessary external wiring of the microcontroller as well as variety of peripheral devices for testing and debugging.

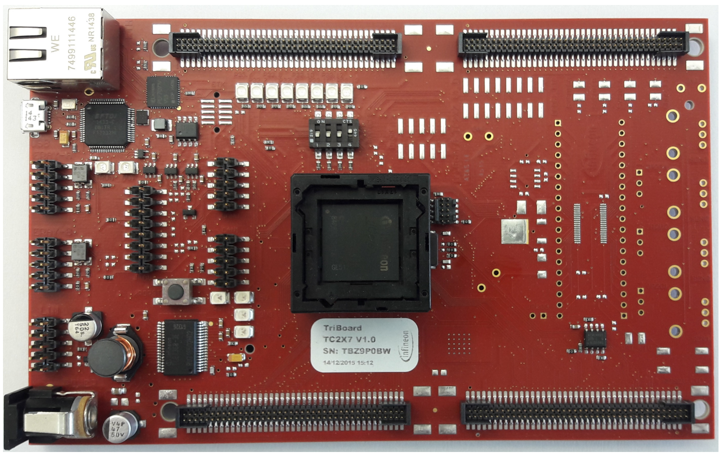


Figure 2.1.: Infineon TriBoard TC2X7 with TC277T (BGA-292)

2.1.1. Power Supply

The board can either be supplied via a Micro USB connector or via a DC supply connector. When the board is supplied via the DC supply connector, a voltage in the range from $5V$ to $50V$ can be applied. This input voltage is connected to the multifunctional power supply circuit IFX TLE7368-3E by Infineon. The TLE7368-3E converts the input to a $5.5V$ voltage provided to the microcontroller. The microcontroller generates the necessary voltage levels of $3.3V$ and $1.3V$ via separate parallel Embedded Voltage Regulators (EVR33 and EVR13). By default the TC277T is configured to use a $5V$ voltage level for its ports, including all communication interfaces and reference voltage for the ADCs.

However, AVL EMS uses $3.3V$ as voltage level. Therefore the configuration was adjusted by changing the wiring of the HWCFG[0:2] (Hardware Configuration) pins of the microcontroller. The HWCFG[0:2] corresponds to the port pins P14_2 (HWCFG2), P14_5 (HWCFG1) and P14_6 (HWCFG0). By default a 680Ω resistor is connected to each hardware configuration pin. To set the voltage levels of TC277T to $3.3V$, the resistor R524 at P14_5 (HWCFG1) needs to be removed.

2.1.2. Startup configuration

TC277T basically provides three possible startup configurations[1]:

- **Internal Start**: the first instruction is fetched from address `0xA000 0020` of the internal flash
- **Bootloader Mode**: code/data is downloaded into the Instruction Scratchpad Memory SPRAM by using:
  - ASC bootloader (Asynchronous/Synchronous Communication)
  - CAN bootloader (Controller Area Network)
  - Generic bootloader - based on the first received byte at specified port pins one of the bootloaders above is selected
- **Alternate Boot Modes**: program code is started from a user-defined address

One of the configurations is selected by the wiring of the HWCFG[3:6] (Hardware Configuration) pins. On the TriBoard a DIP-Switch is provided to change the configuration. During the development the configuration to start from internal flash was selected (HWCFG[6...3] = 1110) as shown in figure 2.2.

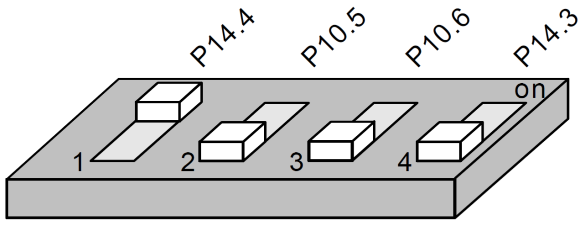
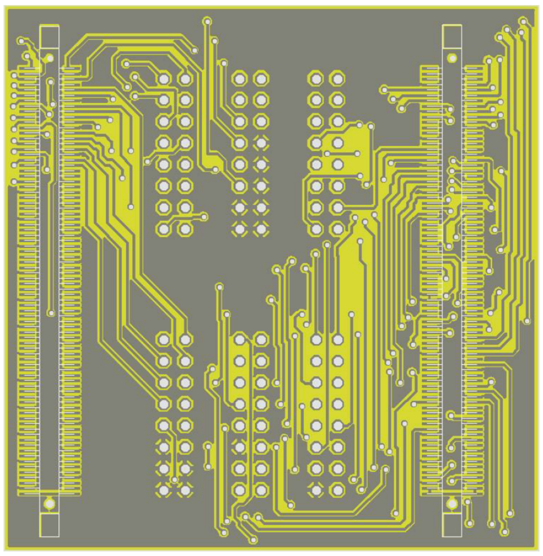


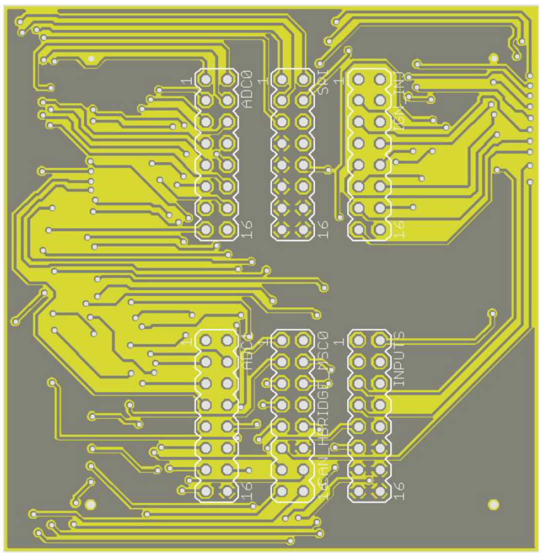
Figure 2.2.: TriBoard Hardware configuration DIP-Switch[2]

[1]The Boot Options are defined in Infineon, 2014a, page 240
[2]Infineon, 2013, page 26

2.2. Connector PCB

In order to connect the EMS main board with the TriBoard TC2X7 during the development and testing, a connector Printed Circuit Board (PCB) was designed. The board can be connected to the target EMS main board instead of the processor board and provides access to several signals. Figure 2.3 shows the bottom and top side of the bare connector PCB.


(a) Bottom side



(b) Top side

Figure 2.3.: Bare connector PCB

2.3. Lauterbach LA-7704

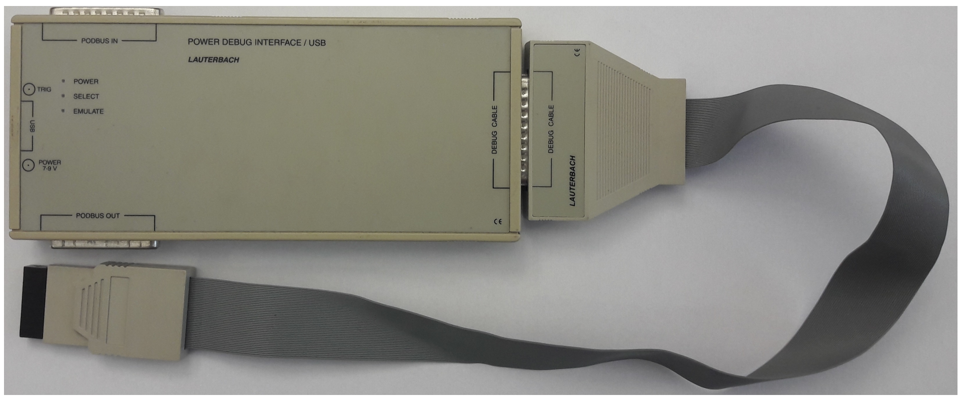


Figure 2.4.: Lauterbach LA-7704 with connected Debug Cable

During development the generated code[3] was programmed into the internal flash of TC277T with the JTAG Debugger (Joint Test Group Array) LA-7704, shown in figure 2.4. The Debugger can be connected to a host PC via USB. The Software interface on the host provides the following possibilities[4]:

- high-level and assembler debugging
- Display of internal and external peripherals at a logical level
- Flash programming
- Hardware breakpoints and trigger
- Software trace

[3]After the build process an file of type ELF (Executable and Linking Format) is generated.
[4]Referring to Lauterbach, 2010

3. Utilized software

The following chapter is intended to introduce the utilized software in the development process as well as the build process of the software for the EMS in ETAS ASCET. Further the key concepts of the real-time operating system in the AVL EMS software are pointed out and Infineons software framework used for the development of the HAL is depicted.

3.1. ETAS ASCET

The embedded software for the EMS is developed with the ASCET product family by ETAS. It consists of a number of products, supporting development process for embedded automotive control software. The tool of this family which supports the interface for a developer to design code in different languages (for example C or visual programming) is ASCET-MD. With this tool ASCET models are generated, which can be passed to ASCET Software Engineering (ASCET-SE).
The ASCET-SE tool provides functionalities to generate target specific C code, integration of the code into a target operating system and invoke a target-specific compiler and linker for generation of an executable application file. Furthermore it offers the possibility to generate an ASAM-MCD-2MC file for measurement and calibration from an existing executable file.
The main functionality of ASCET-SE is the conversion of an ASCET model (usually a project in ASCET for a chosen target), into C code. For each component of the ASCET model, C source code files are generated. These components include the ASCET project itself, each software module in the project, each class and each operating system task body. The code generator uses the target configuration files to optimize code generation or customize the code where necessary. For example ASCET-SE can generate compiler-specific pragmas to place code or data at specific memory sections.
ASCET-SE also generates the configuration file for the target operating system (OS). This file defines all OS objects required by the ASCET configuration. In the build process for the EMS software the OS generator tool RTA-OSEK (see 3.2) is used to generate the data structure required by the operating system.
The combination of ASCET and OS code includes all variable and data definitions required to make the ASCET system work.

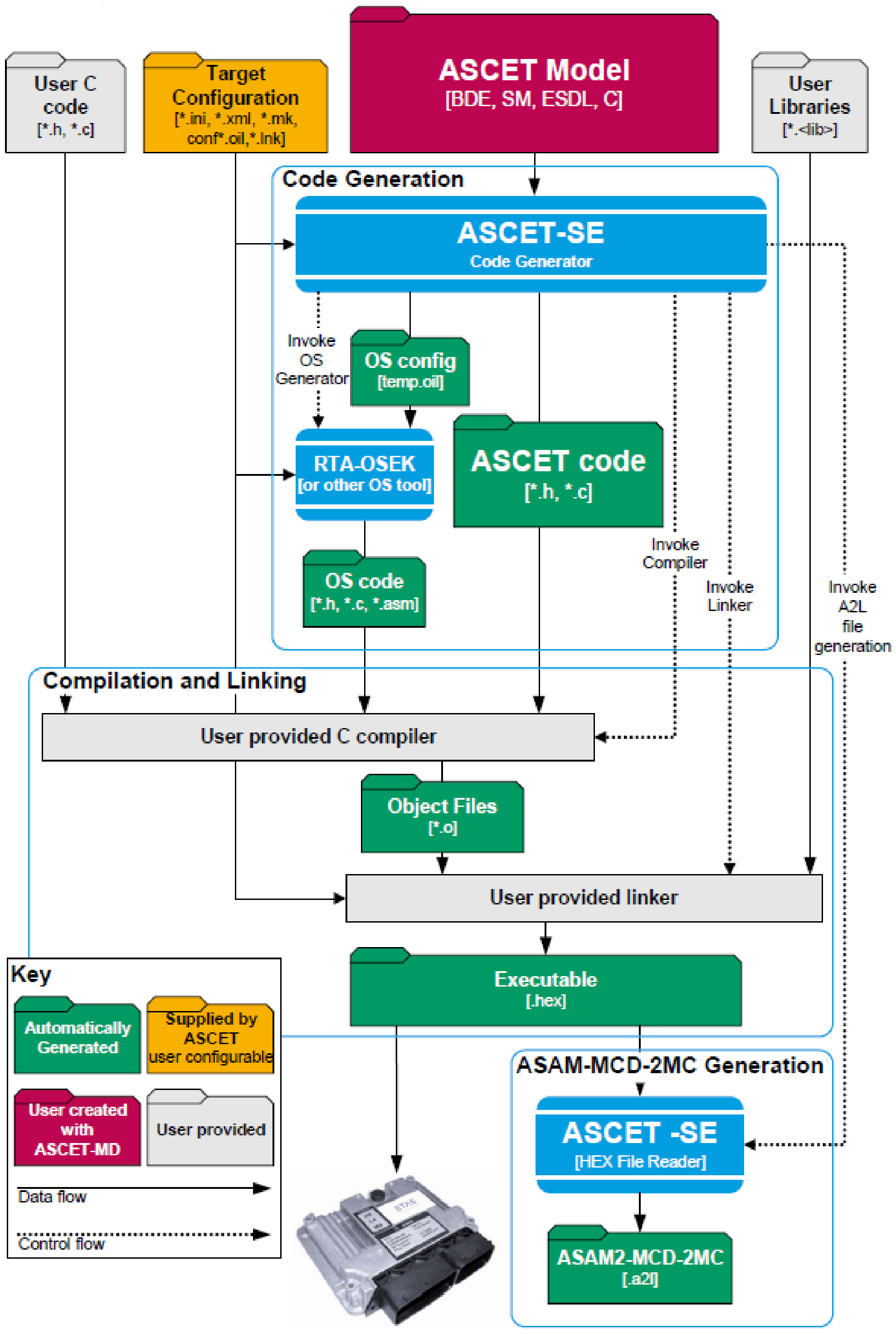


Figure 3.1.: Stages of ASCET-SE code generation[1]

[1]ETAS, 2011, page 22

The ASCET and OS code, as well as external included C code is passed to the user specific compiler and linker. The toolset, which is used in the build process for the EMS is the TASKING TriCore Software Development Toolset (Altium, 2012). Based on the target configuration files the linker generates the executable file for the target system from the object files. The object files are generated by the compiler and can additional be provided by external included libraries.
These stages of the ASCET-SE code generation are illustrated in figure 3.1.

3.2. ETAS RTA-OS5

In this section the characteristics of a real-time operating system are described, subsequently the main concepts implemented in ETAS RTA-OS5 are explained. The discussed concepts of real-time operating systems refers to Noergaard, 2012, page 383 - 441.

3.2.1. Real-time operating systems

A real-time operating system (RTOS) is characterized by using time as a key parameter for the execution of tasks. Therefore the term *real-time* refers to the algorithm used for the scheduling of task. For further explanation, it is necessary to take a closer look on the concept of a task[2].

Tasks

An operating system differentiates between a program as a passive, static sequence of instructions and the executing program, which is an active, dynamic event. Processes are generated by an operating system to encapsulate all the information needed for an executing program (like the stack, program counter, source code and data, etc.). A program is therefore only a part of a task.

Scheduling

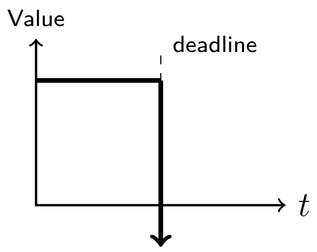
In multitasking operating systems, multiple tasks can exist at a given time. All these tasks share the same resources including the CPU (central processing unit), that can't be allocated by the tasks simultaneously. To overcome this problem the tasks are multiplexed to the CPU in time. This generates the illusion of executing the tasks simultaneously. The order and the

[2]Tasks are also often referred to processes in literature. To be consistent with ETAS RTA-OS the term task is used in this work.

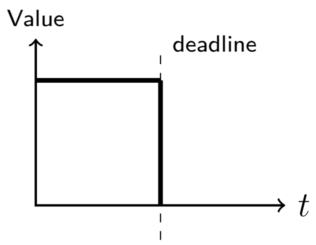
duration of the tasks to run on the CPU is determined by the scheduler, a mechanism provided by the OS. The scheduler queues the tasks based on an algorithm and the CPU is allocated to the next task, what is called dispatching. In some operating systems the dispatching is done by the scheduler, while others have a dedicated mechanism, called dispatcher. Various scheduling algorithms are implemented in operating systems and every design has its strengths and tradeoffs.
In general all different algorithms can be grouped into two categories:

- **non-preemtive scheduling**: tasks are given control of the master CPU until they have finished execution, regardless of the length of time or the importance of the other tasks that are waiting. This algorithms can be riskier to support, since a task could run into an infinite loop.
- **preemtive scheduling**: the OS forces a context-switch on a task, whether or not a running task has completed executing or is cooperating with the context switch

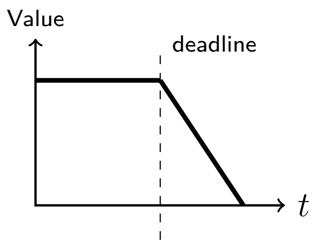
For a RTOS, preemtive scheduling is the most efficient scheduling policy. The algorithms used for scheduling in RTOSs usually determine the next task to run on the CPU based on different parameters including the priority (importance of the execution of a task), frequency (number of times a task runs), deadline (when a task has to finish its execution) and the duration (time it takes to finish execution of a task).



(a) Hard deadline



(b) Firm deadline



(c) Soft deadline

Figure 3.2.: Real-time requirements for operating systems[3]

In a RTOS tasks always have to meet their deadlines. However, it can be distinguished between three types of real-time requirements as illustrated in figure 3.2.
hard real-time requirements: If the deadline of a task is missed, the costs of missing the deadline is extremely high and can include human lives. The value of the computed results heads for negative infinity.
firm real-time requirements: A missed deadline does not result in a catastrophe, but the computed results are useless.
soft real-time requirements: Dismissed deadlines are tolerated in a certain range and deadlines are met in a statistical distribution. The value of the computed results decreases in proportion to the delay, depending on the application.

[3]based on Noergaard, 2012, page 409

Inter task communication

An operating system has to provide mechanisms enabling tasks to communicate and synchronize their behaviour, in order to prevent concurrent accesses to the same shared hardware or software resources. Usually these mechanisms consist of one or a combination of the following three models:

Shared data model The tasks access the same data at a shared memory space. The main issue of this model are so called *race conditions*, which occur if one task is preempted while writing to the shared memory space and therefore the integrity of the data is not guaranteed. To prevent race conditions, parts of the tasks accessing the shared memory can be labeled for mutual exclusion (MUTEX). One mutual exclusion technique is the implementation of *Semaphores*, which are used to lock a shared memory for access by other tasks.
Another technique are *Processor assisted locks*. If a task is scheduled in a way that no other task can preempt it, an occurring interrupt is still able to force a context switch. The interrupt service routine could then access the same shared memory, as the preemted task and a race condition occurs. By disabling interrupts while executing critical section (earmarked for MUTEX) this can be prevented. Another option to prevent such scenarios is to set a flag in a register, for tasks that must not be interrupted. Any other task has to test this flag (*test-and-set-instruction*).

Message passing Tasks communicate via messages that are queued into *message queues* between the tasks. The implementation of this communication scheme varies from one operating system to another.

Signaling Signals are used to advise tasks of an asynchronous event. Whenever a task receives a signal, it suspends its execution and a context switch to the signal handler is enforced. Signals are usually used for interrupts, where the signal handler becomes the corresponding interrupt service routine.

3.2.2. Main concepts of ETAS RTA-OS5

The information about the concepts of RTA-OS5 can be found in ETAS, 2012, where ETAS defines RTA-OS5 as follows:

> *"RTA-OS is a statically configurable, preemptive, real-time operating system (RTOS) for use in high-performance, resource-constrained applications. RTAOS is a full implementation of the open-standard AUTOSAR R3.x and AUTOSAR R4.0 OS (including multicore) specifications and is also fully compliant to Version 2.2.3 of the OSEK/VDX OS Standard."*
> ETAS (2012, p.16)

AUTOSAR (AUTomotive Open System ARchitecture) is a development partnership of automobile manufacturers, suppliers and tool developers worldwide, with the primary goal of the standardization of automotive software architectures. The open-standard provides, among other specifications, a specification for operating systems intended for automotive applications. This specification subsumes features from the earlier OSEK[4] OS standard. OSEK is on the other hand a European automotive industry standard, with the goals of supporting portability and reusability of software components across a number of projects.
Neither AUTOSAR nor OSEK are relevant in the development of the software for AVL EMS and therefore they will not be explained in detail during this work, however the concepts of RTA-OS5 implements these standards.

Tasks in RTA-OS5

Tasks in RTA-OS5 correspond to the definition in 3.2.1. However two types of tasks are differed:

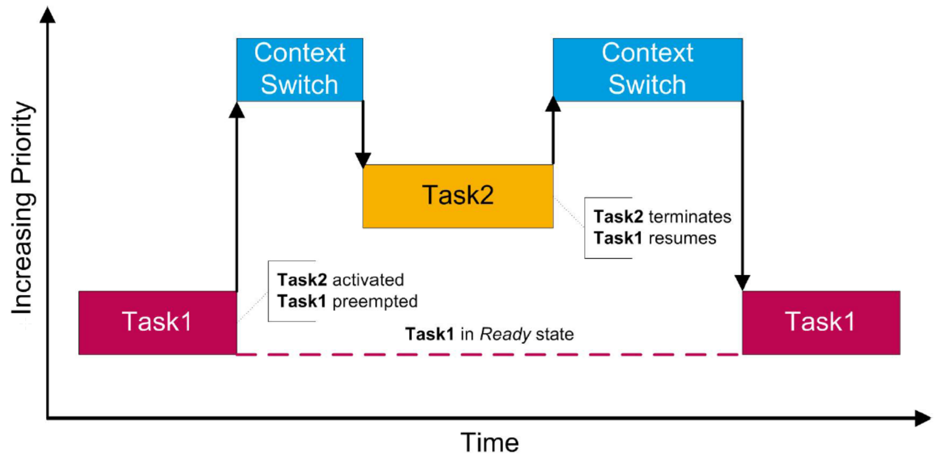
- **Basic tasks** are intended to start, execute and terminate. The execution of a basic task is performed until it is preempted by a task with a higher priority. Therefore only three different states can be assigned to a basic task: suspend, ready or running.
- **Extended tasks** can have one more state in addition to the ones of basic tasks. The additional state is called *waiting state*. Extended tasks can stop their execution and wait for events to occur (for example a user interaction).

Every task has a state, which defines its current behavior. These states consist of suspended (the default state of all tasks), ready (a task is scheduled for execution but not dispatched i.e. the task is activated) and running (the task is dispatched to the CPU) tasks. Extended tasks can change their state to an additional one: waiting (the task voluntarily suspends and waits for an event).

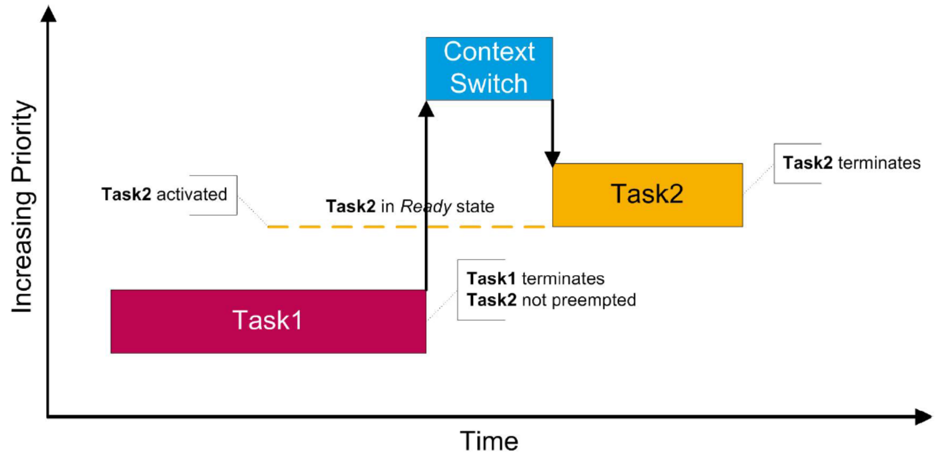
Scheduling

Preemptive as well as non-preemptive scheduling is supported by ETAS RTA-OS5. It is possible to define a task as preemtive or non-preemtive. Preemtive tasks can be preemted by other task where non-preemtive tasks finish their execution even if a task with a higher priority gets activated. The priority of a task defines the order when a task is scheduled and reflects the relative urgency of tasks.

[4]OSEK is a German acronym for *"Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen"*. English: *"Open Systems and their Interfaces for the Electronics in Motor Vehicles"*



(a) Preemptive tasks



(b) Non-Preemptive tasks

Figure 3.3.: Scheduling of tasks[5]

Figure 3.3 illustrates the two scheduling methods for tasks, where in 3.3(a) the preemtive Task1 is preempted by the higher priority Task2. In 3.3(b) Task1 is a non-preemptive task. Therefore Task2 is not executed until Task1 terminates, although Task2 has a higher priority. Whether a task is preemtive or not, it can be preempted by an interrupt at any time.

Cooperative scheduling, where a task suspends voluntarily at a given time is not provided by RTA-OS5, however this can be implemented by making a `Schedule()` API call at certain points of execution, what is for example implemented by ASCET-SE.

[5]ETAS, 2012, page 53f

Interrupts

If the interrupt is generated by the hardware, the processor loads the corresponding interrupt vector, which contains the address of the interrupt handler. The OS provides this interrupt handler and first saves the current context. Then the interrupt service routine (ISR) is executed. When the ISR has finished, the context is restored and the OS resumes normal execution.

Mutual exclusion

Occurring interrupts can lead to race conditions, where data consistency is not guaranteed. Therefore when executing critical sections in tasks that must not be interrupted, RTA-OS5 provides processor assisted locks like the disabling of interrupts.
Further binary semaphores are provided to prevent any task or interrupt to enter the same critical section at a time. However binary semaphores can introduce priority inversion, where a task with low priority prevents a task with higher priority from execution. In extreme cases even deadlocks can occur, where all tasks are waiting to enter a critical section that is currently used by another task.
RTA-OS5 overcomes this problem by priority ceiling protocol, where the priority of a task is increased for the time it is in a critical section.

Counters

RTA-OS5 implements counters to count ticks, where ticks can be defined by the user of the OS. Usually ticks are defined as a specific time interval (for example $1ms$) but also other definitions can be used (Rotations, button presses, errors, etc.). If a tick occurs, an ISR is used to increment the value of the counter.

Alarms

Alarms are used to activate a task if a counter has reached a specific value. Each counter can trigger multiple alarms. Alarms can be specified to expire on a periodic basis, what is called a cyclic alarm. In the AVL EMS software cyclic alarms are used to call tasks periodically. ASCET-SE provides tasks of type alarm and generates the corresponding C code for the OS (see 3.3).

3.2.3. Multicore support of RTA-OS5

RTA-OS5 provides the following concepts concerning multicore applications as described in RTA-OS5 User Guide ETAS, 2012:

- *The OS objects that belong to an OS Application run exclusively on the core that they are allocated to – this includes tasks and ISRs.*
- *The scheduling of tasks and ISRs is independent between cores. A low priority task on one core does not get affected by higher priority tasks or ISRs on a different core. Disabling interrupts on a core does not affect any other cores. Similarly resources are restricted to a single core, so locking a resource cannot affect tasks on a different core.*
- *Tasks can be activated from any core. If the task is configured to run on a different core then the OS will ensure that this happens correctly.*
- *Spinlocks are introduced to protect read/write access to shared memory by different cores. As has been explained, one can not rely on task priority, interrupt locks or resources to help safely share data between tasks on different cores. Spinlocks are used to ensure that only one core at a time can access shared data. A core must 'get' the spinlock before entering the critical section of code and must release it afterwards. If one core has already got the lock, then any other core that tries to get the lock will become blocked (busy-wait) until the lock gets released. Higher priority tasks or ISRs can still preempt tasks that are blocked waiting for the spinlock.*

The first statement in the list above means that every task and interrupt service routine (ISR) is executed only on the core it is assigned to. The OS editor integrated to asket provides the possibility to specify the core the tasks and ISRs are executed on.
Scheduling of tasks and ISRs as well as the locking of resources is treated separately on each core.
The activation of a task[6] can be done on a different core the task is executed.
Since the locking of resources does not affect tasks on different cores, a spinlock[7] mechanism is used.

3.3. Development with ASCET

ASCET tools provide a multi-paradigm modelling framework, where modules can be programmed textually or visually. These modelling languages abstract from the low-level details and separate the concerns of what the system software must do from how it is realized in the final code. However, this is only useful for the development of the application software. The

[6]A task is activated by the scheduler. Activating a task means that a task is scheduled for execution.

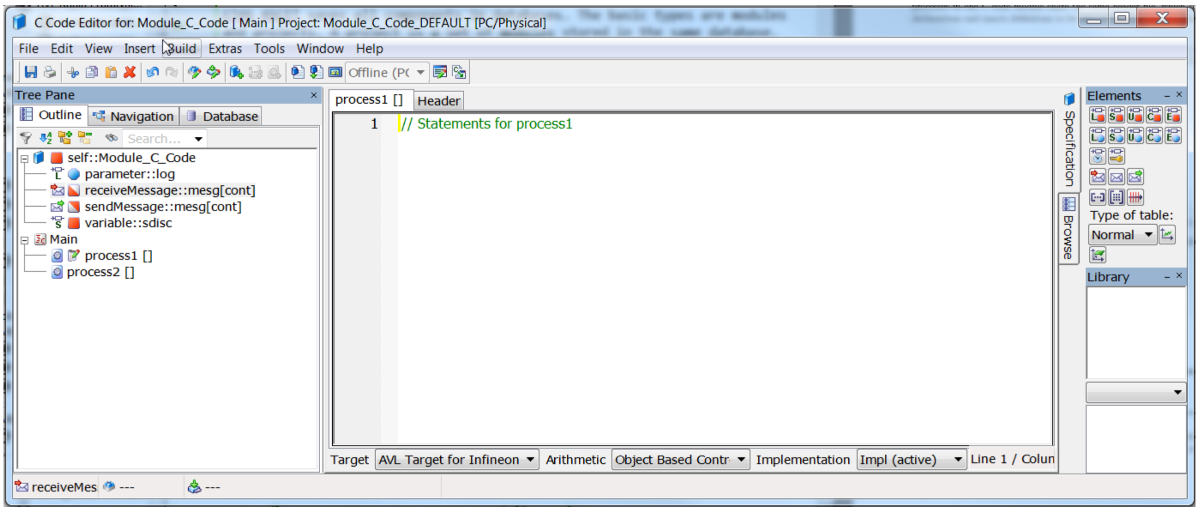
[7]If a task tries to get a lock for a resource in a loop, this is called spinlock. For setting a lock for a resource a lock variable is used. (From Mandl, 2014, page 162)

low-level functions (like the device drivers) need to be programmed in a low-level programming language and therefore ASCET also provides C code development. In the following, only the programming interface for C code is addressed.

ETAS ASCET saves all components in databases. The basic types are modules and projects. A project is a set of modules stored in the same database. Multiple projects can access the same module, what means that one module can be reused in different projects.

Each module consists of at least one process [1], that clusters sequential statements. Processes are generated by ASCET-SE as `void`/`void` functions ETAS, 2011, meaning that they have neither arguments nor a return value. Further a module can contain send and receive messages for data exchange with other modules to ensure data consistency. ASCET-SE generates copies of messages in all required cases, explained by an example below.

Messages can also be used for data exchange between the processes of one module. Data that belongs to the module itself is stored in variables and parameters.
For programming of the statements of one process a C code editor is available. All processes in one C code module share the same header file, which contains C function declarations and macro definitions to be shared between the processes.


C Code Editor for: Module_C_Code [Main] Project: Module_C_Code_DEFAULT [PC/Physical]
File Edit View Insert Build Extras Tools Window Help
Offline (PC
Tree Pane
Outline Navigation Database
Search...
self::Module_C_Code
parameter::log
receiveMessage::mesg[cont]
sendMessage::mesg[cont]
variable::sdisc
Main
process1 []
process2 []
process1 [] Header
Specification
Browse
Elements
Type of table:
Normal
Library
Target AVL Target for Infineon Arithmetic Object Based Contr Implementation Impl (active) Line 1 / Colun
receiveMes --- ---

```
// Statements for process1
```

Figure 3.4.: ASCET code editor for example module

[1]Here, the term process is not referred to the usual concept of processes or rather tasks in operating systems, as described in 3.2.1.

Figure 3.4 shows the module editor for one example C code module, which consists of the processes *process1* and *process2*, the messages *receiveMessage* and *sendMessage*, a parameter and a variable. In addition to the code editor for each process and the shared header file, ASCET provides an external C code editor intended to provide functions, which can be called in the code for every process.

The OS editor in ASCET provides the possibility to assign all processes of the modules in a project to tasks. Tasks can be added to a project with different types, which define the event on that the task is set to ready-state. The activation of a task does not imply its immediate execution, but it is scheduled by the operating system. The four different types provided by ASCET are:

- **Init**: The task is activated at the start of the OS. As indicated by the name, these type is used for the task that holds the processes to initialize the system.
- **Interrupt**: A task of type *Interrupt* is activated if a defined hardware interrupt occurs and contains usually the process that serves as the corresponding interrupt service routine.
- **Alarm**: Tasks are activated when a particular counter value is reached. The OS editor of ASCET provides a numerical input in nanoseconds to define the tick duration of a counter (refer to counters in 3.2). The value of this counter is compared to a period defined for a task of type alarm. If the counter reaches this value the alarm is generated and the corresponding task is activated. Therefore alarm is usually used for the periodical activation of tasks.
- **Software**: A software task is activated by calling it in a statement in a process of another task.

If a task is activated and dispatched, the processes that are assigned to it are executed. An example of the configuration of the operating system in ASCET is given in 4.2.

The scheduling method for a task can also be selected in the OS editor. The options are *FULL* for preemptive scheduling, *NON* for non-preemptive scheduling and *COOPERATIVE* for cooperative scheduling that ASCET SE implements at code generation, since this method is not directly provided by RTA-OS5.
As already mentioned preemptive tasks can always be preempted by tasks with a higher priority or an interrupt, where on the other hand non-preemptive tasks can only be preempted by interrupts.
Cooperative tasks can be preempted by non-preemptive tasks, preemptive tasks and interrupts. However a cooperative task can only be preempted by another one with a higher priority, between the process boundaries. This is achieved by dividing the priority space of the OS into two parts - one for cooperative tasks and one for other tasks. The part that is used for the cooperative tasks is the lower part of the priority space of the OS in the range of 0 to a defined value $Coop.Levels - 1$, that can be defined in the OS editor. The second

part of the priority space starts at *Coop.Levels* and all non-cooperative tasks are assigned a priority from this part. Figure 3.5 illustrates the partition of the priority space.

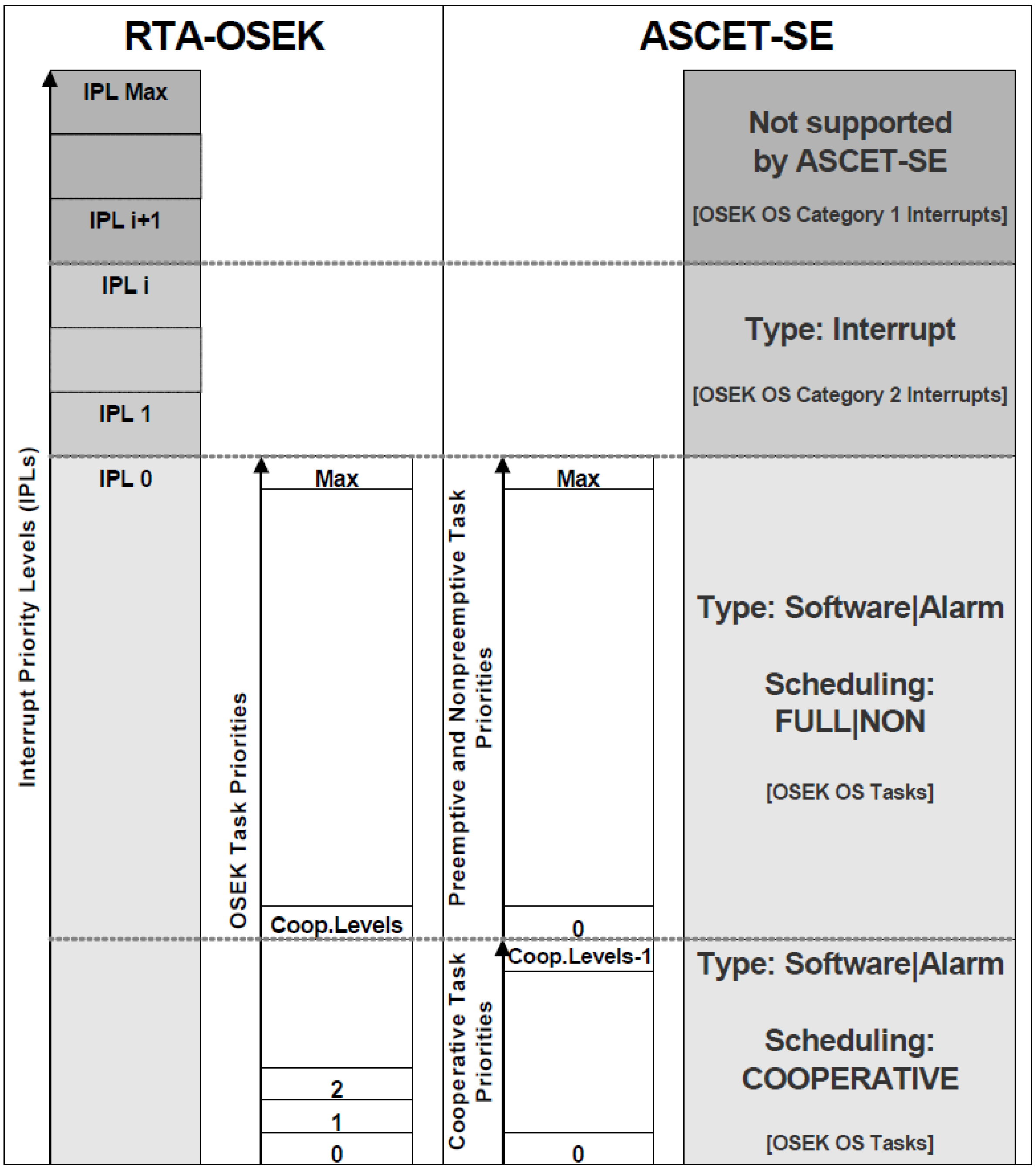


Figure 3.5.: Partition of operating system's priority space by ASCET-SE[8]

[8]ETAS, 2011, page 83

In AUTOSAR operating systems interrupts are categorized into two priority categories. Category 1 includes all interrupts, which do not interact with the OS. These interrupts are assigned the highest priorities. A developer has to configure the hardware correctly, write the handler and the return from the interrupt. Interrupts interacting with the OS are assigned Category 2 and are supported by ASCET-SE, as can be seen in figure 3.5.

Implementation of messages by ASCET SE As already mentioned, modules can communicate via messages. The concept and how messages are implemented by ASCET-SE is explained in the following example:
If a message `mAB` is used to send data from module `moduleA` to module `moduleB`, it has to be added to module `moduleA` as send message, while in module `moduleB` it has to be added as receive message. Let us further assume the process `p1_A` of module `moduleA` writes to the (send) message and is assigned to the task `task1`. On the other hand the process `p1_B` of module `moduleB` reads the (receive) messages and is assigned to task `task2`. Figure 3.6 gives an overview of this example.

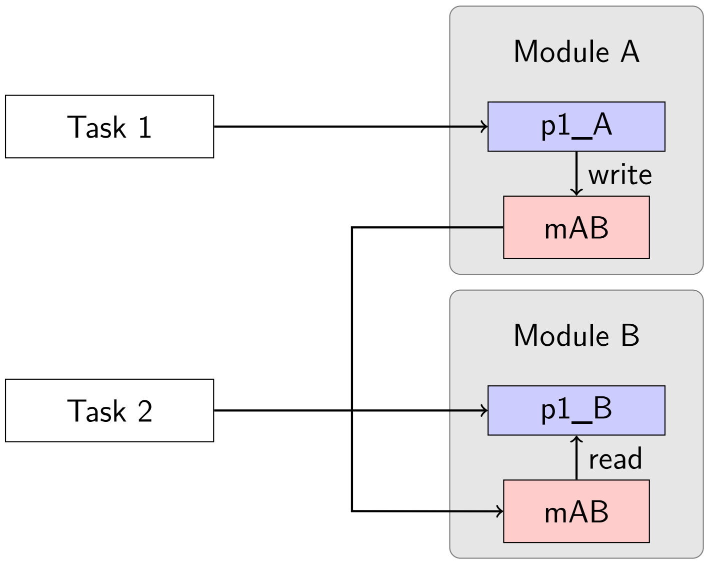


Figure 3.6.: Illustration of a example for messages in ASCET

In order to ensure data consistency, ASCET-SE generates two global variables for each message in the output C code. In this example its `mAB` and `mAB__42__` (the number 42 in the name represents the priority level).
The code for the processes `p1_A` and `p1_B` is given in the listing 3.1 and listing 3.2. As it can be seen, in function `MODULEA_IMPL_p1_A`, which corresponds to the process `p1_A`, the copy of the message is used, where the function `MODULEB_IMPL_p1_B` reads from the message itself. The definition of the tasks `task1` and `task2` is given in listing 3.3. `task1` contains the function call of `MODULEA_IMPL_p1_A`. The function writes to the copy of the

message, therefore interrupts may occur during the write process. The second function that is called in `task1` is `task1_msgsend`. This function disables the interrupts during the copy of the data of `mAB__42__` to `mAB`, whereby data consistency is guaranteed.

```c
...
/* messages used by this process */
#define mAB_MODULEA_IMPL_p1_A mAB__42__

void MODULEA_IMPL_p1_A (void)
{
  mAB_MODULEA_IMPL_p1_A = x;
}
...
```

Listing 3.1: Extract of moduleA.c

```c
...
/* messages used by this process */
#define mAB_MODULEB_IMPL_p1_B mAB

void MODULEB_IMPL_p1_B (void)
{
  x = mAB_MODULEB_IMPL_p1_B;
}
...
```

Listing 3.2: Extract of moduleB.c

```c
...
static inline void task1_msgsend (void)
{
  DisableAllInterrupts();
  mAB = mAB__42__;
  EnableAllInterrupts();
}
...
TASK(task1) {
  ...
  MODULEA_IMPL_p1_A();
  task1_msgsend();
  ...
  TerminateTask();
}
...
TASK(task2) {
  ...
  MODULEB_IMPL_p1_B();
  ...
  TerminateTask();
}
...
```

Listing 3.3: Task definitions in conf.c

3.4. Infineon AURIX Software Framework

During the development of the software components Infineon AURIX Software Framework was used. The framework includes an IDE (integrated development environment), which is based on *Eclipse* (open-source, developed by Eclipse Foundation). The build process is supported by the Software Framework Tools, which generate the specific make files for the used compiler toolchain[9].
The advantage of using Infineon AURIX Software Framework was the available example project including all header files with register address definitions, which is a good basis to start. Infineon provides also a set of examples for the hardware modules of TC277T.

3.5. TASKING VX-toolset for TriCore

To build AVL EMS software the TASKING TriCore Software Development Toolset is used. It contains dedicated C/C++ compilers and assemblers for the complete TriCore family, and a multi-core linker/locator[10].

[9]Referring to Infineon, 2015b
[10]Altium, 2012

4. Background

4.1. Definition of hardware abstraction layer

A uniform explicit definition of the term *hardware abstraction layer* (HAL) does not exist, since its implementation varies from one operating system to another or designers define its implementation for their specific hardware / software architecture.
However generally speaking the term HAL can be defined by the description of its properties and the functions it should provide as follows:

- it is a software layer between the hardware platform and the operating system
- it hides implementation details of the hardware platforms from the software
- The purpose of its usage is on one hand the shortening of the time for new development of higher layer software. On the other hand the purpose is to increase portability of the higher layer software.

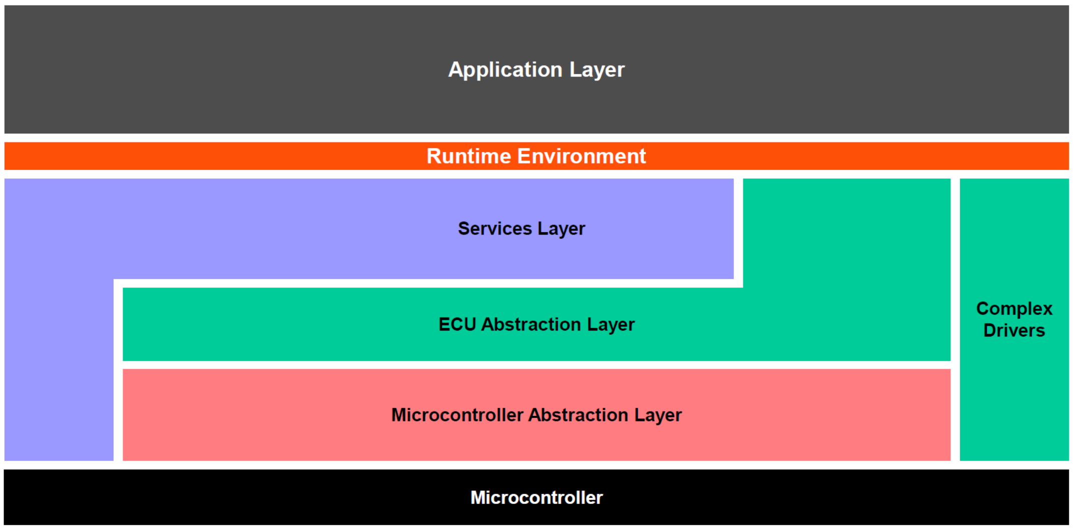


Figure 4.1.: Overview of Software Layers defined by AUTOSAR[1]

[1]AUTOSAR, 2016, page 12

As an example AUTOSAR does not define a HAL, but a software layer *Basic Software*, according to AUTOSAR, 2016. This layer consists of four further layers as illustrated in figure 4.1.
The lowest (closest to microcontroller) layer of the basic software is the *Microcontroller Abstraction Layer*, which provides a standardized interface to software modules with direct access to the microcontroller and its internal peripherals. On top of this layer, the *ECU abstraction layer* is placed. It offers an API for access to peripheral devices, whether microcontroller internal or external. On the right hand of the overview in 4.1, the *Complex Drivers Layer* is illustrated. It integrates special purpose functionality, which includes device drivers with very high timing constraints. The highest layer of the *Basic software* is the service layer. AUTOSAR defines this layer to offer operating system functionalities, vehicle network services, memory services, diagnostic services, ECU mode management and program flow monitoring.

Since the AVL EMS software does not implement an architecture as defined by AUTOSAR, because this has turned out to be not appropriate for the rapid prototyping development of the software, the hardware abstraction layer also differs from the *Basic Software layer*, defined in AUTOSAR, 2016.
In this work the term HAL is used for a software layer that

- provides a hardware independent interface to the application layer,
- integrates all device drivers (microcontroller internal and external) and network services,
- is independent of the operating system.

4.2. Current AVL EMS software architecture

As described in 1.1.2, the hardware abstraction layer (HAL) of the current EMS software consists of the layers 1 and 2. The layer 1 of the HAL is implemented by different ASCET modules, each corresponding to a peripheral device of the microcontroller TC1793. The modules assigned to layer 1 provide processes for the initialization of the devices as well as for interaction with them. On the other hand layer 2 of the HAL converts the raw input/output data to/from layer 1 into physical unit values, which can be used by the application layer.
The software design of AVL EMS is explained below through the example of the input chain of an analog sensor (figure 4.2). The analog signal from the sensor is first converted into digital domain by the Analog-to-digital-converter (ADC) of the microcontroller, triggered by a process of the ADC-module in layer 1, which is running in a specific task that is executed periodically by the operating system. The result of the conversion is stored in a special function register (SFR) of the ADC. This SFR is read by another process of the ADC-Module. This process is part of the corresponding interrupt task, which is triggered by the ADC after the conversion has finished. When the process is executed, it reads the conversion value

and stores it in a message. The message is read by the module ADC-remap, which converts the raw value into a physical unit value (for example in degree Celsius). The example shows the basic idea behind the layered design of the EMS software and makes clear that in contrast to other operating systems, where the HAL is part of the kernel and functions provided by the HAL are called via system-calls, the HAL of the EMS software is not part of the underlying OS. The HAL is rather implemented in different processes (refer to 3.3) beside the application processes and also integrated in tasks, organized by the operating system.

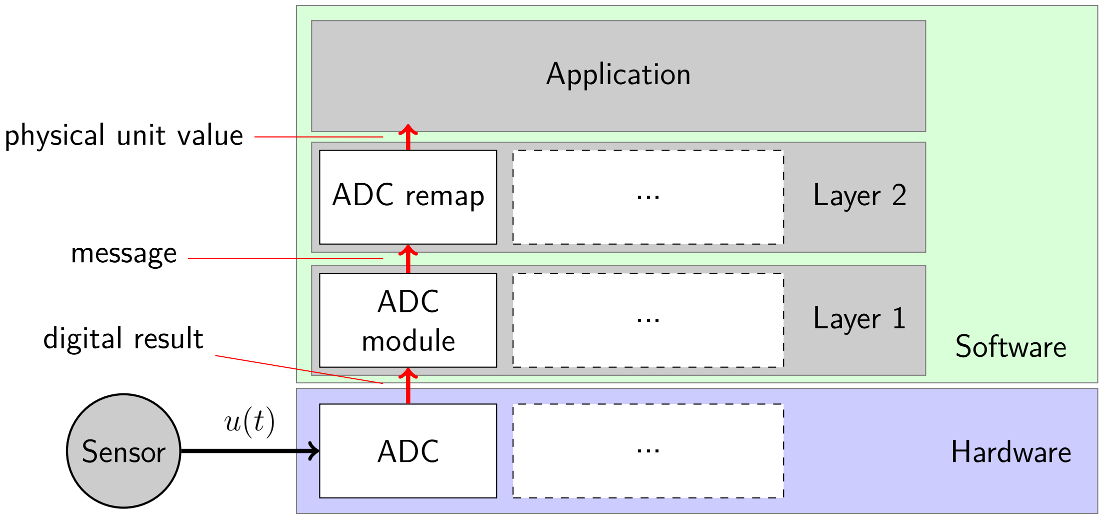


Figure 4.2.: Structure of the layered conversion of a analog input signal

The implementation of the two layers of the example for the ADC in ASCET is as follows:

HAL layer 1 The layer 1 for the ADC of the microcontroller is implemented in the module *ADC*, which provides three processes:

1. A process for the initialization of the ADC of the microcontroller by setting the specific values of the special function registers (SFR). This process also sets the SFRs to generate an interrupt, when the conversion has finished.

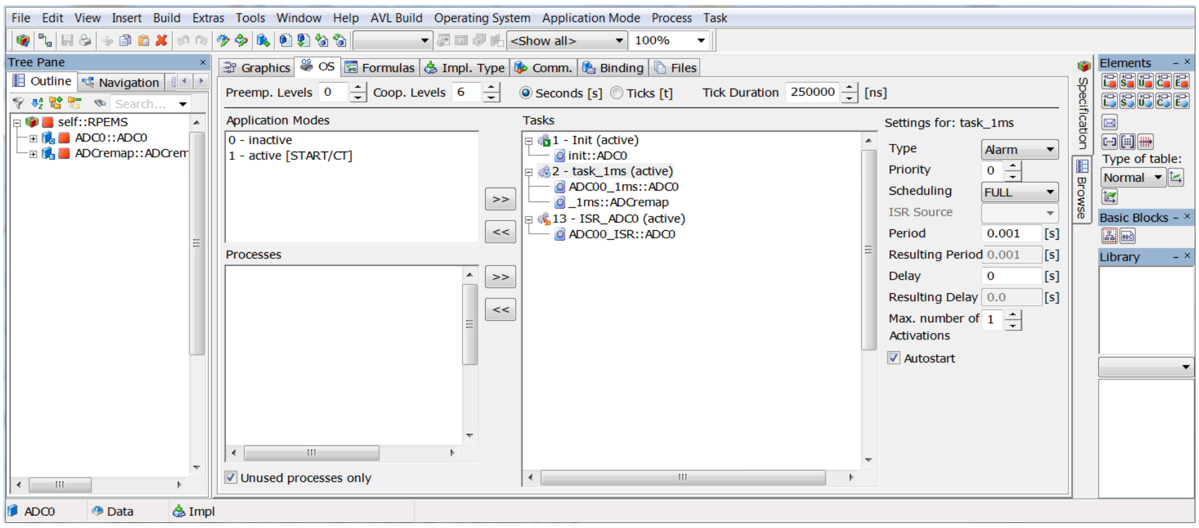
2. A process to trigger the conversion of the ADC.

3. A process to read the conversion results from the corresponding registers. This process is intended as interrupt service routine for the interrupt generated when the conversion has finished.

Furthermore the module contains multiple send-messages - one for each digital raw conversion result of an analog input.

HAL layer 2 As layer 2 is intended to convert raw values into physical unit values and vice versa, the module for the ADC is called *ADC_remap*. The module consists of receive-messages (the counterparts for each send-message of the layer 1 module) and one process that converts the values into physical unit values.

Operating system The operating system provides the possibility to define tasks that are executed periodically with various periods and also interrupt-tasks that are executed if a specific interrupt occurs. The processes of the different modules can be assigned to these tasks.
For the example of the implementation of layer 1 and 2, figure 4.3 illustrates the interface of the OS editor in ASCET.



```
self::RPEMS
  ADC0::ADC0
  ADCremap::ADCrem

Preemp. Levels 0    Coop. Levels 6    (●) Seconds [s]  ( ) Ticks [t]    Tick Duration 250000 [ns]

Application Modes
0 - inactive
1 - active [START/CT]

Processes
[✓] Unused processes only

Tasks
1 - Init (active)
    init::ADC0
2 - task_1ms (active)
    ADC00_1ms::ADC0
    _1ms::ADCremap
13 - ISR_ADC0 (active)
    ADC00_ISR::ADC0

Settings for: task_1ms
Type               Alarm
Priority           0
Scheduling         FULL
ISR Source
Period             0.001  [s]
Resulting Period   0.001  [s]
Delay              0      [s]
Resulting Delay    0.0    [s]
Max. number of Activations  1
[✓] Autostart
```

Figure 4.3.: Interface of the OS editor for an example project in ASCET

The ADC example project consists of the modules *ADC0* and *ADCremap*, where the layer 1 module *ADC0* contains the processes *init*, *ADC00_1ms* and *ADC00_ISR*. The layer 2 module *ADCremap* contains only one process *_1ms*.
Three different tasks were added to the operating system:

- **Init**: The first task to execute only once
- **task_1ms**: A periodically generated task with period of $1ms$ of type *Alarm*
- **ISR_ADC0**: A task with high priority that is executed if the corresponding interrupt for the completion of the conversion of ADC0 is generated.

The processes of the modules are assigned to the corresponding task as can be seen in the tree view of the tasks in figure 4.3.

First the process *init* of the module *ADC0*, which is assigned to the *Init* task is executed and the ADC of the microcontroller is initialized.
With a period of $1ms$ the task *task_1ms* is executed. It contains the process *ADC00_1ms*, which starts the conversion of the ADC0 of the microcontroller. Further the task *task_1ms* contains the process *_1ms* of the module *ADCremap* that receives the raw values of the conversion via messages. However in the first iteration these messages contain their default values, since the conversion has not finished and the interrupt is not generated.
If the conversion, started every $1ms$, has finished, an interrupt is generated and the task *ISR_ADC0* is executed. It contains the process *ADC00_ISR*, which reads the conversion results in the SFRs of the ADC0 and sends them via messages to the layer 2 module *ADCremap*.

This ADC example project shows the design of the AVL EMS HAL and gives a basic knowledge of a smart HAL architecture in ASCET.

5. Implementation of the HAL

In the following the implementation of the modules are discussed. Each implemented software module is explained and a detailed overview of the corresponding hardware module of TC277T is given. The introduction of the hardware modules is based on the document Infineon, 2014a. Further code examples are given for accessing the functionalities of the modules.

5.1. Architecture of the software

The current EMS software implements hardware access as part of layer 1. This layer consists of processes, which are integrated into tasks of the operating system. The processes include explicit read and write statements to the special-function-registers (SFR) of the microcontroller. Therefore the statements are difficult to read and hardly understandable without previous knowledge about the hardware and the corresponding SFRs[1]. If a property of a hardware-module needs to be adapted during the rapid-prototyping development process, an application developer needs knowledge about the bits in a SFR which are responsible for the setting of this property. Moreover most of the code in layer 1 includes statements, which is changed rarely or even not at all.
To overcome this, statements with similar functionality are encapsulated in functions. With this encapsulation of the statements into functions and parameters describing the demands of an application programmer, the access to the hardware is abstracted from the view of an application programmer.
All the resulting functions regarding hardware accesses shall be provided in form of a compiled library and be included during the build process (an overview of the code generation is given in figure 3.1). Figure 5.1 illustrates this concept, where the on-chip hardware as well as the on-board hardware is accessed only by the device drivers. The HAL consists of device drivers, layer 1 and layer 2. The operating system provides different tasks including the processes of the modules in the application layer and in the HAL. Layer 1 processes call the corresponding abstracted functions, provided by the device drivers.

[1]This holds especially for the statements in the initialize-processes of the hardware modules, but also for other processes placed in layer 1

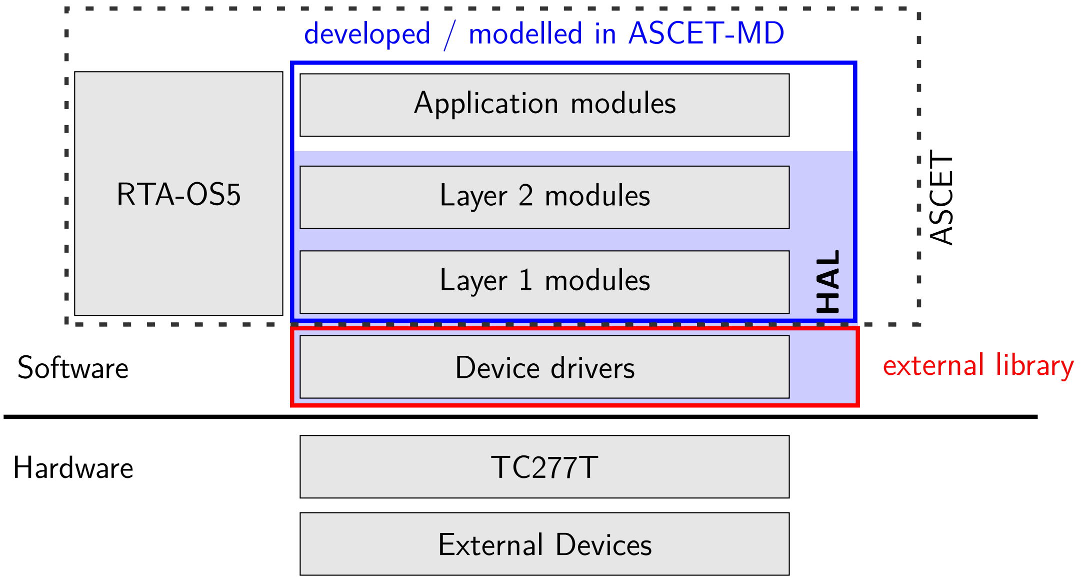


Figure 5.1.: Overview of the software architecture

5.2. Multicore considerations

To prevent simultaneous access to hardware resources, by tasks running on one of the three cores of the TC277T, only one core is responsible for hardware access. This core uses the interface to the externally provided library, while the other two cores run the application software. Therefore the HAL is executed only on one core, which converts data and provides physical unit values to the other cores. On the other hand it should convert physical unit data from the other cores into raw data, needed for the peripheral devices.
The communication between the cores is done by message passing. However messages only disable the interrupts during writing on the specific core the task is running on[2]. Therefore it is necessary to use spin locks for inter task communication.

As this work is the basis for the future use of the multicore microcontroller, where one core is responsible for the access to the hardware, the HAL was implemented on core 0 of TC277T.

[2]Messages are explained in 3.3 - *Implementation of messages by ASCET SE*.
Multicore support of ETAS RTA-OS5 is discussed in 3.2.3.

5.3. Digital in- and outputs

5.3.1. TC277T General Purpose I/O Ports

In order to provide logical signals to peripheral devices on the main-board of the EMS, multiple port pins of TC277T are configured as general-purpose outputs. The state of external logical signals can be read by configuring port pins as general-purpose inputs. Figure 5.2 gives an overview of the general structure of a Port Pin. The behavior of each Port Pin is configured by setting the SFRs (Special Function Registers).

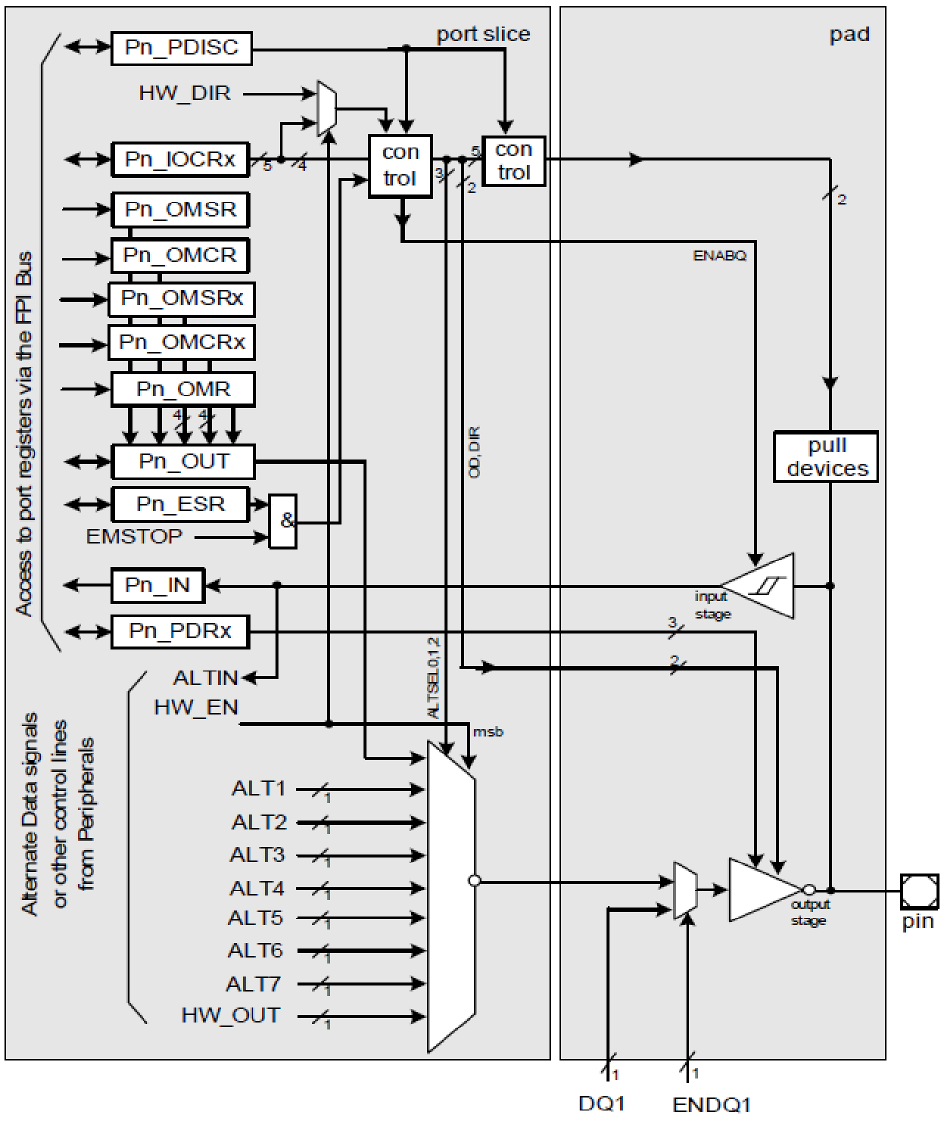


Figure 5.2.: General structure of a Port Pin in TC277T[3]

If the Port Pin is configured for input mode the output driver is switched off (high impedance). The input voltage is translated to a logical signal by a Schmitt-Trigger and can be read via the register PN_IN. Furthermore the logical input signal is passed to various inputs of peripheral units. The Schmitt-Trigger is always enabled, whether the Port Pin is configured for in- or output mode, so the signal can also be read via the PN_IN in output mode. Additionally a weak pull-up or a pull-down device can be connected.

In output mode the output driver is turned on and is fed by the signal from the output multiplexer, which selects the signal source. If the Port Pin is used as general-purpose output, the multiplexer is switched to the register PN_OUT. The inputs of the multiplexer ALT1 - ALT7 are connected to different peripheral units that uses the Port Pin as output.

5.3.2. DIGIO software module

All functions for reading/writing the voltage level from/to external wires are included in the *DIG_IO* module, which consists of the files `DIG_IO.c` and `DIG_IO.h`.
In the file `DIG_IO.c`, two fixed-size arrays of structure type `DigIO` are declared in the file scope - one for the output and one for the input signals. The structure `DigIO` contains the information about the corresponding port, the pin, and the mode the port pin should be initialized to.
The interface to module is listed in table 5.1.

Table 5.1.: Interface to the DIGIO software module

Function	**Parameters**	**Return**	**Description**
DIG_IO_init	void	void	Initializes the arrays for input and output and configures the SFRs of the Port Pins.
DIG_OUT_set	uint16 number bool state	void	Sets the Port Pin defined by number from DigitalOut[] array to either high or low.
DIG_OUT_get	uint16 number	bool	Reads the logic value of the Port Pin defined by number from DigitalOut[] array and returns high or low.
DIG_IN_get	uint16 number	bool	Reads the logic value of the Port Pin defined by number from DigitalIn[] array and returns high or low.

[3]Infineon, 2014a, page 1067

5.4. Analog to Digital Conversion

Several sensors connected to the EMS deliver their output signal (like temperature, pressure, mass air flow, etc.) in form of an analog signal. The continuous analog signal, representing the measured environment information, is converted into discrete digital domain by a Analog to Digital Converter (ADC) for further processing.

5.4.1. TC277T Versatile Analog-to-Digital Converter (VADC)

The TC277T provides eight ADC Kernels for analog to digital conversion, working on the principle of successive approximation. Assigned to every kernel is a group. Each group comprises a converter unit, a request source and a result register. The converter unit includes a multiplexer, which selects an analog input channel as input to the AD converter. For each group eight analog input channels are available. The AD converter is able to convert signals with a resolution of up to 12 bits. Each channel can either be configured for 12-bit, 10-bit or 8-bit conversion. In addition a 10-bit fast compare mode is available, providing an indication flag if the voltage level of an analog input is over or under a user specified value[4].
Furthermore, external multiplexers are supported for extending the input channels of one group. Figure 5.3 depicts the ADC Kernel block diagram.

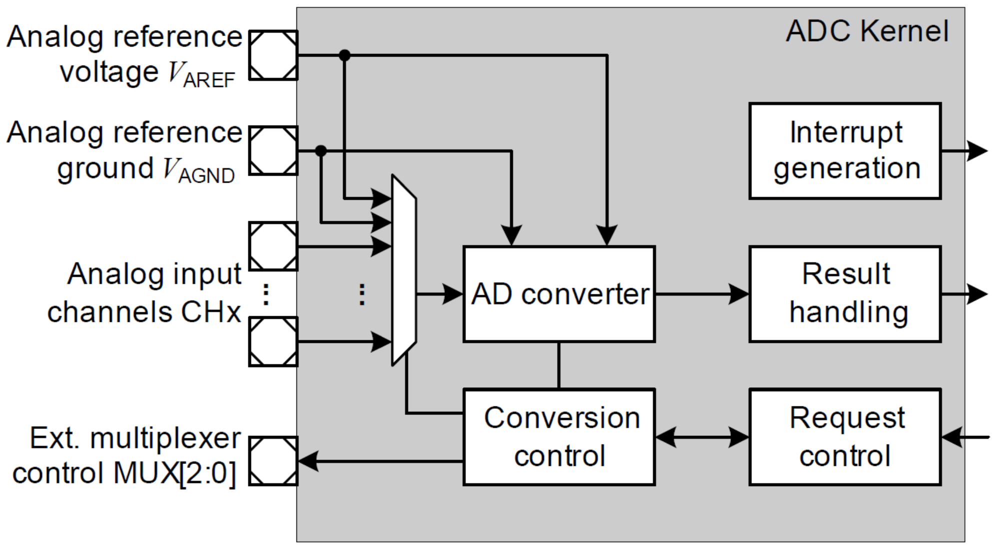

Analog reference voltage V_{AREF}
Analog reference ground V_{AGND}
Analog input channels CHx
Ext. multiplexer control MUX[2:0]
ADC Kernel
Interrupt generation
AD converter
Result handling
Conversion control
Request control

Figure 5.3.: ADC Kernel block diagram[5]

[4]Hemlin; 2015, page 21
[5]Infineon, 2014a, page 3878

A conversion can be triggered by external events, continuously or by software events. Upon a trigger event a request source requests the conversion of the specified sequence of analog input channels. Three request sources are available for each channel:

- Queued Request Source:
  A queued request source stores a programmed sequence of up to 8 channels to convert into a FIFO (First In First Out) based queue buffer. The channels are converted in the order defined by the sequence. Any channel combination is possible.
- Group Channel Scan Request Source:
  The group channel scan request can request the conversion of up to eight channels of a group.
- Background Channel Scan Request Source:
  This scan source can request all channels of all groups.

Because all request sources can be enabled at the same time, an arbiter resolves concurrent conversion requests from different sources.
The conversion results of each analog input channel can be stored in one of 16 result registers (for each group) or in a global result register accessible from any group. An available Wait for read mode blocks new writes in a result register before the content has been read to avoid loss of data.

The VADC is able to generate service requests for three types of events:

- Request source event: channel converted in queue source or sequence finished in a scan source
- Channel event: indicates a channel conversion has finished or in case of limit checker, indicates the corresponding event
- Result event: a new result is available

For indicating the completion of conversion of all defined channels of a specific group the Channel Event is used to issue a service request to the CPU by EMS software. The service request is used to generate an interrupt.

Conversion timing

The converters of the VADC are all supplied with two clock signals to ensure a deterministic behavior of converters that shall operate in parallel. The clock signal for the arbiters and f_{ADCI} for the converters are derived from the peripheral system clock by a divider - one for each clock signal.

The conversion timing depends on several user-definable factors[6]:

- The ADC conversion clock frequency, where[7] $f_{ADCI} = \frac{f_{VADC}}{(DIVA+1)}$
- The selected sample time[8] $t_S = (2 + STC)t_{ADCI}$
- The selected operating mode (normal conversion / fast compare mode)
- The post-calibration time PC (0 for uncalibrated conversion)
- The result width N (8/10/12 bits) for normal conversions

For standard conversions the minimum conversion time can be calculated by the following formula:

$$t_{CN} = (2 + STC + N + PC)t_{ADCI} + 2t_{VADC}$$

As an example the following configuration is assumed:

- ADC frequency (equal to System frequency): $f_{VADC} = 100MHz$, $t_{VADC} = 10ns$
- Prescaler factor: $DIVA = 5$
- Result width $N = 12$
- Uncalibrated conversion: $PC = 0$

$$\begin{aligned} t_{VADCI} &= \frac{DIVA+1}{f_{VADC}} = \frac{6}{100MHz} = 60ns \\ t_{Cmin} &= (2+12)60ns + 2 * 10ns = 860ns \end{aligned}$$

[6]From Infineon, 2014a, page 3967

[7]f_{VADC} equals the system frequency, $DIVA$ is the prescaler factor and f_{ADCI} is the frequency provided to the AD converters.

[8]The sample time is used to adapt to sensors. It is configured by a user defined number STC of additional clock cycles to be added to the minimum sample phase of 2 analog clock cycles.

5.4.2. VADC software module

The software module VADC provides functionalities for initializing the VADC hardware module of the TC277T, initializing the groups of the module and for reading the results of the conversion.
As the conversion of the analog inputs into the digital domain takes a certain amount of time, each group is configured to generate an interrupt after a conversion has finished. Thereby it is signaled that results can be read in the corresponding registers. Figure 5.4 depicts the conversion sequence. First a group of the VADC is triggered for conversion. During conversion the CPU is not involved and can execute other tasks. After a conversion has finished an interrupt is generated and the results, stored in the conversion registers, are returned by the read-function.

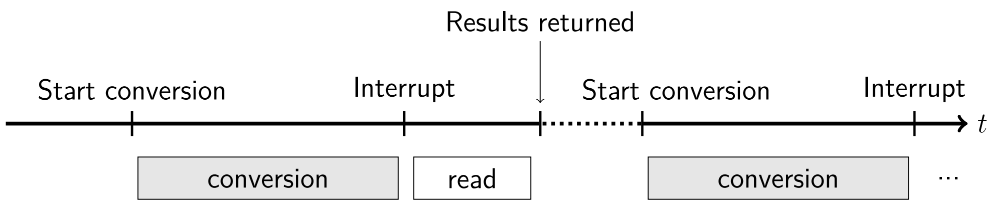


Figure 5.4.: Timing of the conversion process

The addresses of the registers corresponding to one group and the ones corresponding to the channels of the group are encapsulated in the structure `VADC_Group`. All groups that are used in the EMS software are defined in the file `VADC_regs.c`, which provides the structures `VADC_Group` for each group in the array `VADC_Groups` to the software module. The size of the array is defined in the symbol `VADC_GROUPSIZE`.
For each element in the array (and therefore for each group that is used), an element is added to the enumeration `VADC_GroupNumber` and is assigned the value of the index in `VADC_Groups`. For example if Group0 and Group2 shall be used, the addresses of the corresponding registers have to be added to the array `VADC_Groups` of type `VADC_Group` in the file `VADC_regs.c` as shown in listing 5.1.

```c
#include "VADC_regs.h"
#include "IfxSrc_reg.h"

VADC_Group VADC_Groups[VADC_GROUPSIZE] = {
  //Group0
  {
    .ARBCFG = (uint32 *)&VADC_G0ARBCFG,
    .ARBPR = (uint32 *)&VADC_G0_ARBPR,
    ...
  }
  //Group2
  {
    .ARBCFG = (uint32 *)&VADC_G2ARBCFG,
    .ARBPR = (uint32 *)&VADC_G2_ARBPR,
    ...
  }
};
```

Listing 5.1: Encapsulation of the SFRs of used VADC groups

The enumeration `VADC_GroupNumber`, defined in `VADC.h` has to provide 2 constant values `VADC_GROUP0 = 0` and `VADC_GROUP2 = 1` (since the addresses of the registers of Group2 are stored in the second element of `VADC_Groups`) as shown in listing 5.2.

```
...
typedef enum{
  VADC_GROUP0 = 0,
  VADC_GROUP2 = 1,
}VADC_GroupNumber;
...
```

Listing 5.2: Definition of the enumeration for accessing the SFR array

The initialization function enables the VADC module and sets the frequency f_{ADC}. Subsequently the groups are initialized to start conversion after a pending bit is set. Furthermore the groups are configured to generate an interrupt after all channels of the group are converted and the results in the corresponding conversion registers are valid.
To trigger the conversion of a group the function `VADC_startScan` sets the pending bit of the specified group. The function for reading the conversion results `VADC_ReadGroup` copies the values of the result registers to a array. The base address of the array is passed as parameter to the function. Table 5.2 provides an overview of the interface to the VADC software module.

Table 5.2.: Interface to the VADC software module[9]

Function	**Parameters**	**Description**
init	void	Initializes the VADC module and the used groups to generate an interrupt after conversion has finished.
startScan	GroupNumber group	Sets the pending bit of the specified group to trigger the conversion.
ReadGroup	GroupNumber group uint16 *results	Copies the values of the result registers of the specified group to the array at the passed base address.

[9]For a compact illustration the prefix *VADC_* in the names of the functions and data types is omitted. All functions return void.

5.5. Pulse width modulated signals

Several pulse width modulated signals with different frequencies and duty cycles are necessary for the operation of an EMS. This includes the signals for ignition and injection as well as signals for high pressure pumps and knock sensors that are not dependent on time but on the engine position. Therefore, these angle-synchronous signals are generated in the EPM (Engine Position Management) module, which measures the signals from the camshaft and crankshaft sensors and generates the ignition and injection signals.
All PWM signals for time based control of actuators are initialized and controlled in the PWM software module.

5.5.1. TC277T Generic Timer Module

In the following section the Generic Timer Module (GTM) of Infineon's TC277T, based on the Intellectual Property (IP) by Robert Bosch GmbH, is introduced according to Bosch, 2013 and Infineon, 2014a.

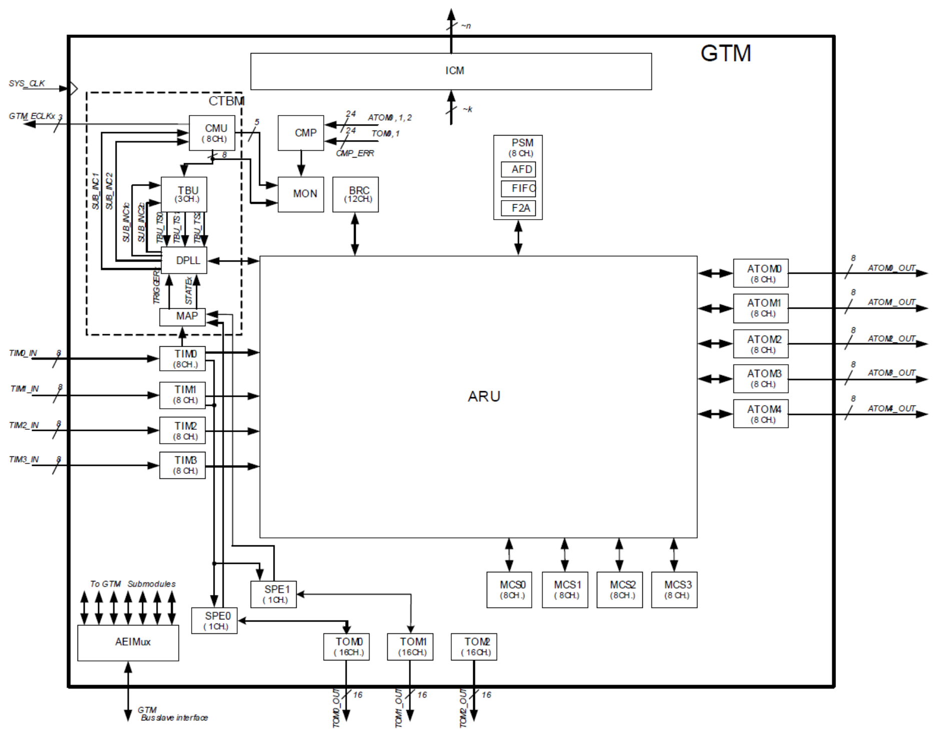


Figure 5.5.: GTM Architecture Block Diagram[10]

[10]Infineon, 2014a, page 2731

The GTM-IP by Robert Bosch GmbH forms a generic timer platform for different applications. Its modular design allows hardware vendors (like IFX) to choose a configuration of the submodules to fit their needs. The main advantage of the GTM is the possibility to offload work from the CPU.
The implementation of the GTM-IP by Infineon in TC277T is illustrated in the architecture block diagram in figure 5.5.
The central component is the Advanced Routing Unit (ARU). The ARU is able to route data from a connected source submodule to a connected destination sub module. The routing is done in a deterministic manner with a round-robin scheduling scheme.
The modules of the GTM used to generate PWM signals, are discussed in detail below.

Clock Management Unit

The Clock Management Unit (CMU) generates the clock signals, mapped to different sub-modules of the GTM. It is divided into three subunits as illustrated in figure 5.6.

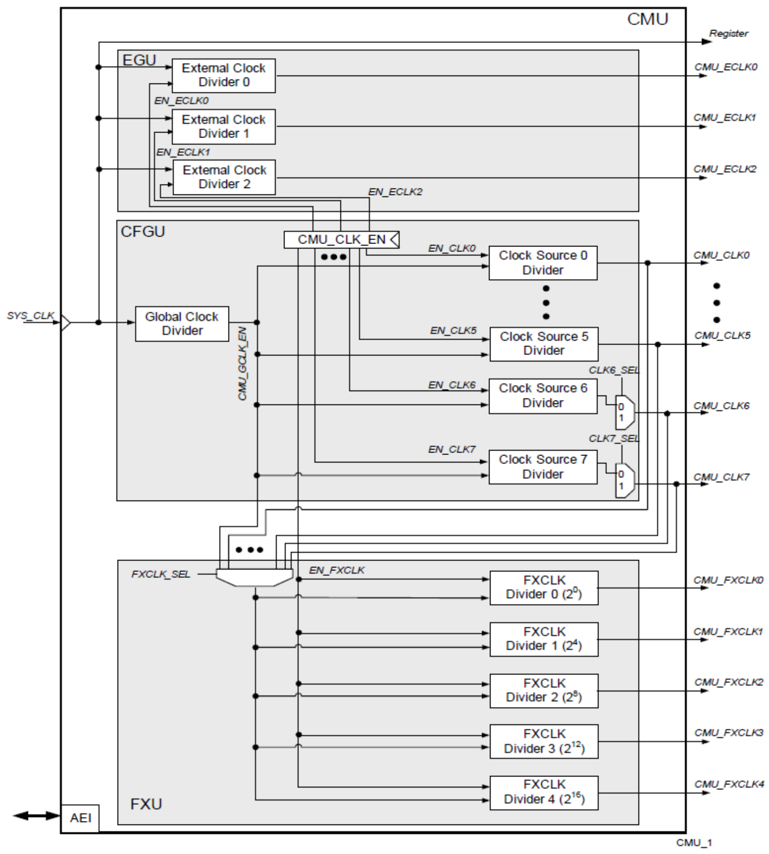


Figure 5.6.: Block Diagram of Clock Management Unit (CMU)[11]

The subunits EGU (External Clock Generation), CFGU (Configurable Clock Generation Subunit) and FXU (Fixed Clock Generation) provide other submodules of the GTM with different clock signals. The signals are derived by dividing the input clock signal SYS_CLK.
The External Clock Generation (EGU) generates up to three configurable clock output signals CMU_ECLK0 - CMU_ECLK2, which are derived from the corresponding External Clock Divider.
The Configurable Clock Generation Subunit (CMU) provides a pre-divided clock signal (output of the Global Clock Divider) CMU_GCLK_EN to eight configurable clock divider blocks. The output signals of these blocks CMU_CLK0 - CMU_CLK7 are additionally mapped to the FXU.
The input signal for the Fixed Clock Generation Subunit (FXU) is selected by a multiplexer. The selectable signals are CMU_GCLK_EN and the output signals of the CMU. The selected clock signal is mapped to five fixed clock divider blocks.

Timer Output Module

The Timer Output Modules (TOM) can be used to generate simple PWM signals. Each TOM consists of two Global Channel Control (TGC0 and TGC1) submodules and 16 independent channels. An overview of the internal structure of the timer output module is given in the block diagram in figure 5.7.
Both Global Channel Control modules TGC0 and TGC1 drive 8 channels synchronously by internal or external events. The channels are controlled by four different signaling mechanisms for each channel:

- Global enable/disable mechanism
- Global output enable mechanism
- Global force update mechanism: the compare registers CM0 and CM1 are updated to the values of the corresponding shadow registers if the channel trigger signal is raised
- Update enable: the compare registers CM0 and CM1 are updated to the values of the corresponding shadow registers on counter reset of CN0

The first three mechanisms can be triggered by three different sources:

- direct write access by the CPU to the registers
- time stamped signals from the Time Base Unit (TBU_TS0, TBU_TS1 and TBU_TS2)
- internal trigger signals TRIG_[x], generated by the channels

Each channel comprises two Counter Compare Units (CCU0 and CCU1) and a Signal Output Generation Unit (SOU). The counter CN0 is clocked by one of the five selectable signals CMU_FXCLK from the Clock Management Unit. In CCU0 the value counter register CN0 is compared with the value of the compare register CM0. In CCU1 the counter register CN0

[11] Infineon, 2014a, page 2829

is compared with the compare register CM1. The width of the registers CN0, CM0 and CM1 is 16 bit. If the value of CN0 is greater or equal to the compare registers, the corresponding Counter Compare Unit (CCU0 or CCU1) triggers the Signal Output Generation Unit.

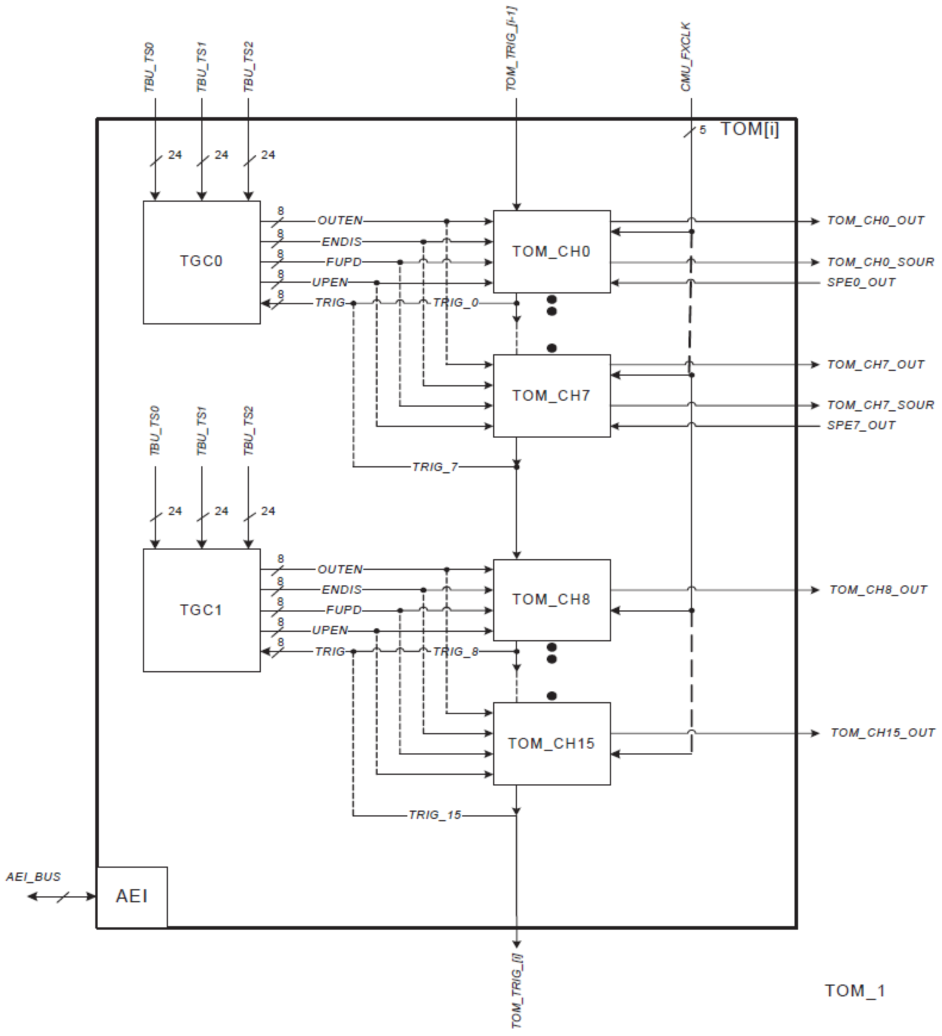


Figure 5.7.: Block Diagram of the Timer Output Module (TOM)[12]

The values of CM0 and CM1 are updated if the Force Enable Update (FUPD) signal from the corresponding Global Channel Control is raised or Update Enable signal is set and the RESET signal is raised. The RESET signal is either raised by the signal TRIG from the previous channel or by the trigger signal TRIG_CCU0 of CCU0. In the first channel the signal TRIG is '0' hard coded. If TRIG from the previous channel is selected for update and UPEN is set, multiple sequenced channels can be updated synchronously. The selection

[12]Infineon, 2014a, page 2918

of one of the five clock signals CMU_FXCLK is selected by the Clock Selection Register CLK_SRC. The register is updated to the value of the shadow register CLK_SRC_SR at the same time as the compare registers of CCU0 and CCU1 are updated. Depending on the value of SL (Signal Level Bit), the output of the channel is set to high or low signal level from CCU0 or CCU1 in SOU. In figure 5.8 the architecture of a channel 0 - 7 is depicted.

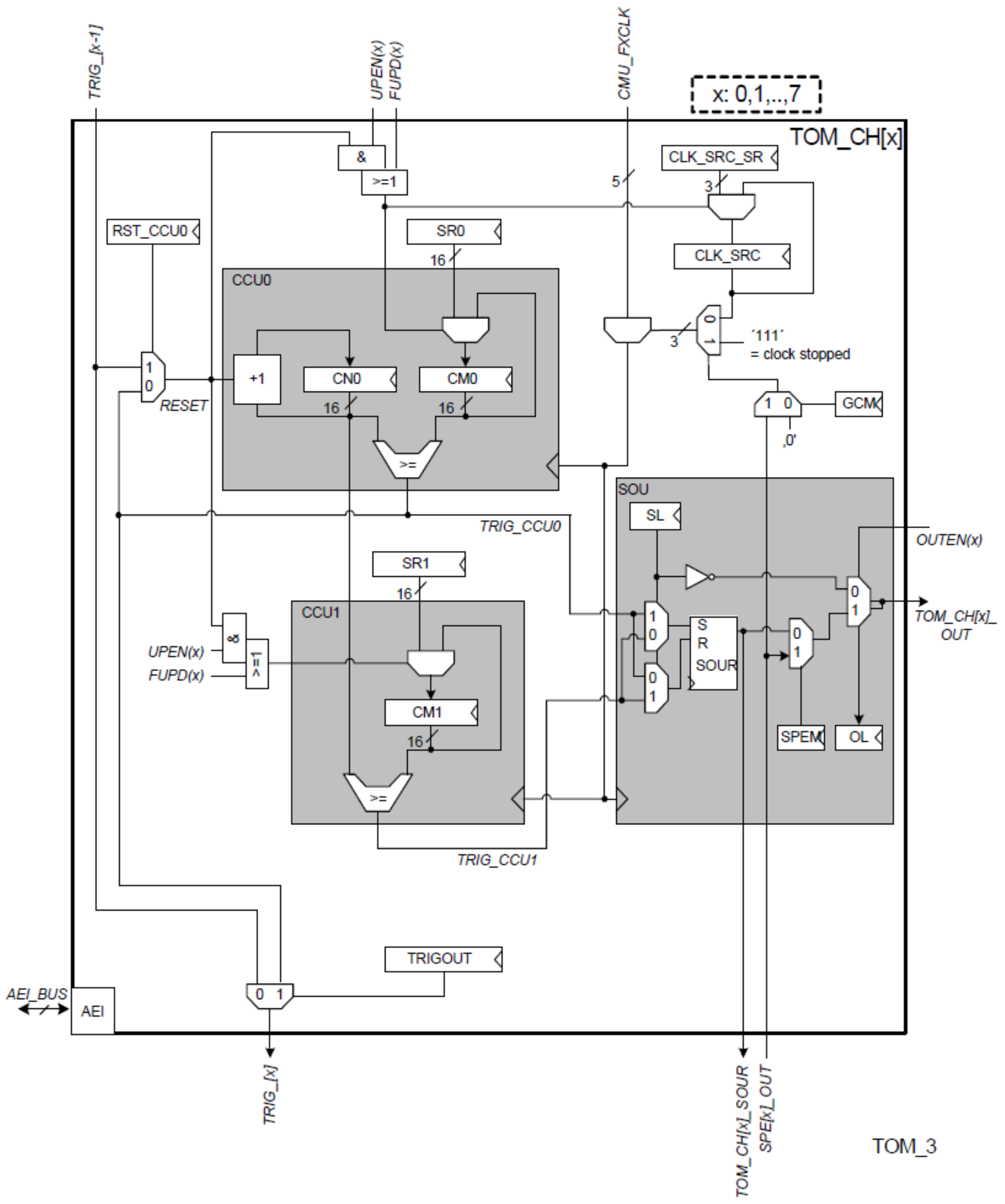
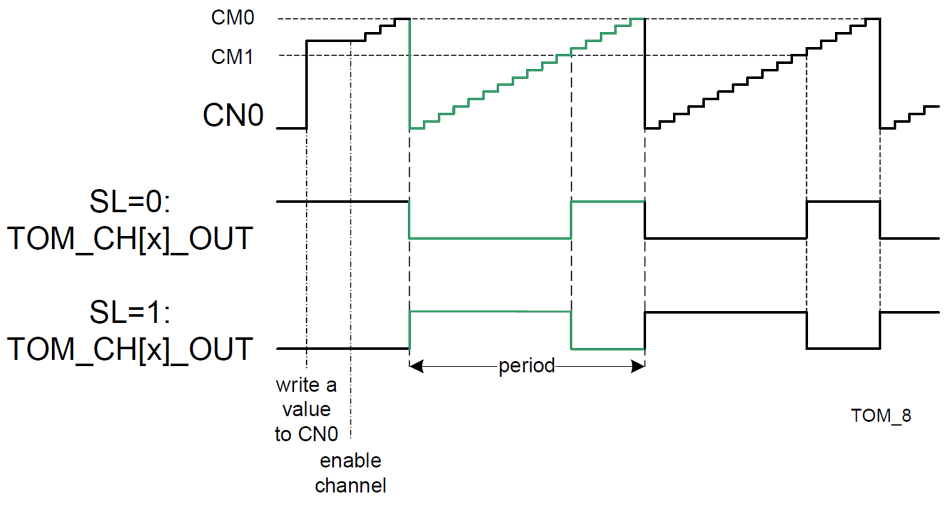


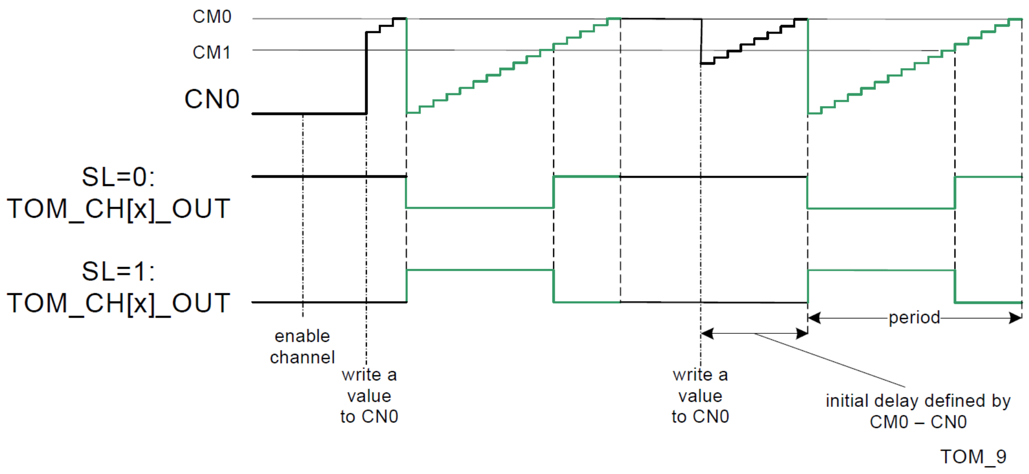
Figure 5.8.: TOM channel 0 - 7 architecture[13]

[13]Infineon, 2014a, page 2923

The TOM channels can either be configured in continuous or one-shot mode. In continuous mode the channel runs independently after it is enabled and an initial value is written to the counter register CN0. In one-shot mode CN0 is only incremented until the counter reaches the value of CM0. In order to start the counter again CN0 has to be re-loaded with a value. The signal level of the output signal TOM_CH[x]_OUT depends on SL as can be seen in the SOU structure. The output of a TOM channel in continuous mode is illustrated in figure 5.9(a) for both configurations of SL. Figure 5.9(b) illustrates the output in One-Shot mode.



(a) Continuous mode



(b) One-Shot mode

Figure 5.9.: PWM Output of a TOM channel[14]

[14]Infineon, 2014a, pp. 2930, 2931

ARU-connected timer output module

With the ARU-connected timer output module (ATOM) complex output signals can be generated without CPU interaction. The Advanced Routing Unit (ARU) provides the possibility to connect the ATOM with other connected submodules. The structure of the ATOM is similar to that of the TOM, although in addition to the ARU connection there are some other differences. In contrast to the TOM, ATOM integrates only 8 channels and therefore only one ATOM Global Control subunit (AGC) is necessary. The AGC's structure is the same as that of the TGC. Furthermore the ATOM is connected to the configurable clock signals CMU_CLK of the CMU submodule. The width of the counter and compare registers of the channels is 24 bit. As illustrated in figure 5.10 the ATOM channel has an additional subunit - the ARU Communication Interface (ACI). The communication between ARU connected modules is based on transmitting and receiving 53 bit ARU words. An ARU word comprises 5 control bits and 48 data bits. Received ARU words are split into three parts and passed to the ATOM channel registers. An overview of the architecture of the ACI is depicted in figure 5.11. Each ATOM channel can be configured to one of four possible modes:

- **Signal Output Mode Immediate (SOMI)**: The output signal ATOM_CHx_OUT is set immediately after receiving an ARU word according to the encoded signal level in bit 48 (first bit of the 5 control bits). The control bits are copied into the 5 bit wide ACBI bit field. ACBI0 is connected to the output stage of SOU (Signal Output Unit). The output signal also depends on the signal level bit SL. the remaining 48 bits of the ARU word has no meaning in this mode. If ARU access is disabled the output signal depends on bit ACB0 and SL of the channel control register.

- **Signal Output Mode Compare (SOMC):** The level of the output signal depends on the comparison of CM0 and/or CM1 with the time based values TBU_TS0, TBU_TS1 or TBU_TS2 from the Timer Based Unit (TBU). If ARU access is enabled, the behavior of the channel is controlled by the 5 control bits of the ARU word, which are passed to the ACBI bit field. The upper 24 data bits of the ARU word are passed to the shadow register SR1 of CM1 and the lower 24 bits are passed to SR0.

- **Signal Output Mode PWM (SOMP):** The channel is operated in the same way as the TOM channels and it compares its registers with the counter register. However they are 24 bits wide and the values for the compare registers can either be written by the CPU or by the ARU. If ARU access is enabled the clock source can be configured by the bits 52 - 50 of the ARU word. The data fields are passed to the shadow registers in the same way as in SOMC mode.

- **Signal Output Mode Serial (SOMS):** The content in CM1 in CCU1 is shifted to the Signal Output Unit (SOU). The direction of the shifting is controlled either by the channel control register or if ARU access is enabled by bit 48 of the ARU word (copied to bit 0 of ACBI bit field). The content of CM1 is written by the upper 24 data bits of the ARU word if ARU access is enabled.

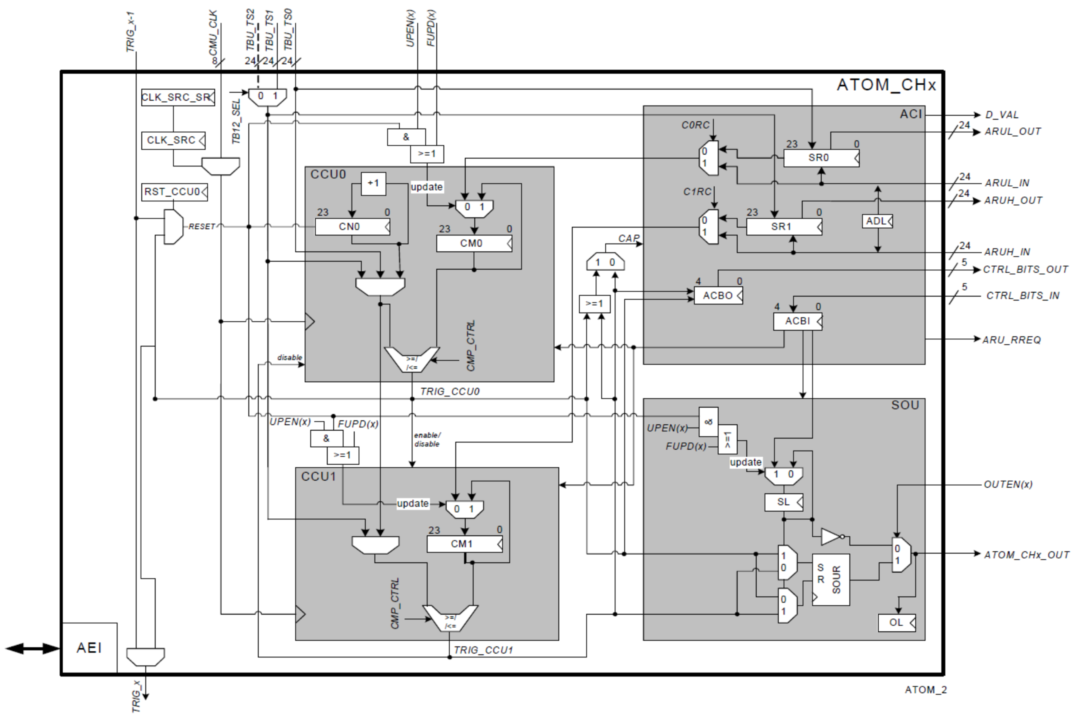


Figure 5.10.: ATOM channel architecture[15]

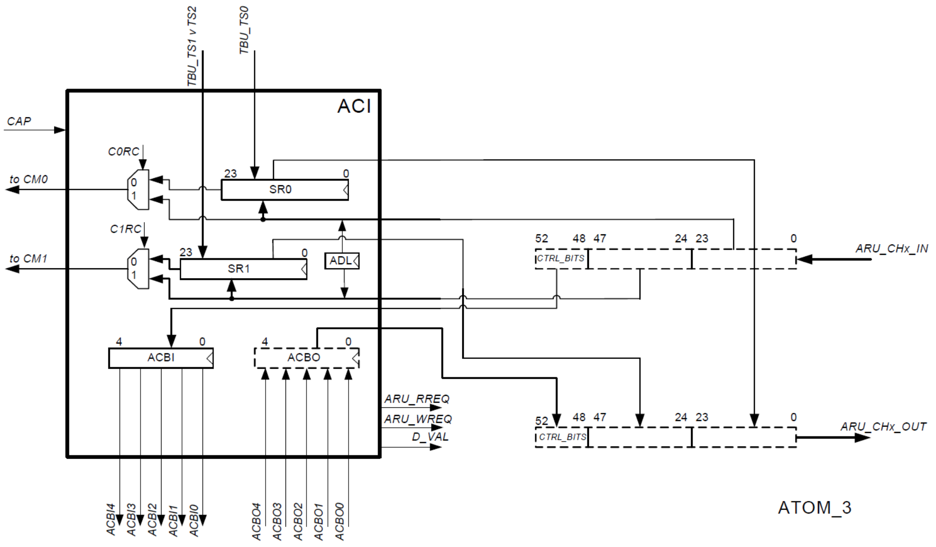


Figure 5.11.: ARU Communication Interface architecture[16]

[15]Infineon, 2014a, page 2974
[16]Infineon, 2014a, page 2976

Timer input module

The Timer Input Module (TIM) provides functionalities for filtering and capturing of input signals. The TIM is also connected to the Advanced Routing Unit (ARU). It is possible to measure time stamp values of detected input rising or falling edges (and the current signal level) of an input signal. The number of edges received since channel enable together with the actual time stamp or the period and duty cycle of pulse width modulated input signals can be measured. Each of the eight input signals of a TIM can be connected to 2 of the eight filter submodules by input multiplexers. The input signals are synchronized to the system clock SYS_CLK, resulting in a delay of 2 periods of SYS_CLK.
After the filtering of the selected input signal[17] the signal is routed to the corresponding TIM channel. The measured values can either be read directly by the CPU or routed to the ARU, providing it to other ARU connected submodules. The output signal for the detection of a rising edge and the one for the detection of a falling edge of the input signal from FLTn subunit are also routed to the Timeout Detection Unit (TDU). The TDU provides the functionality for detecting if a edge (rising or falling) is not followed by a subsequent edge for a specified duration.

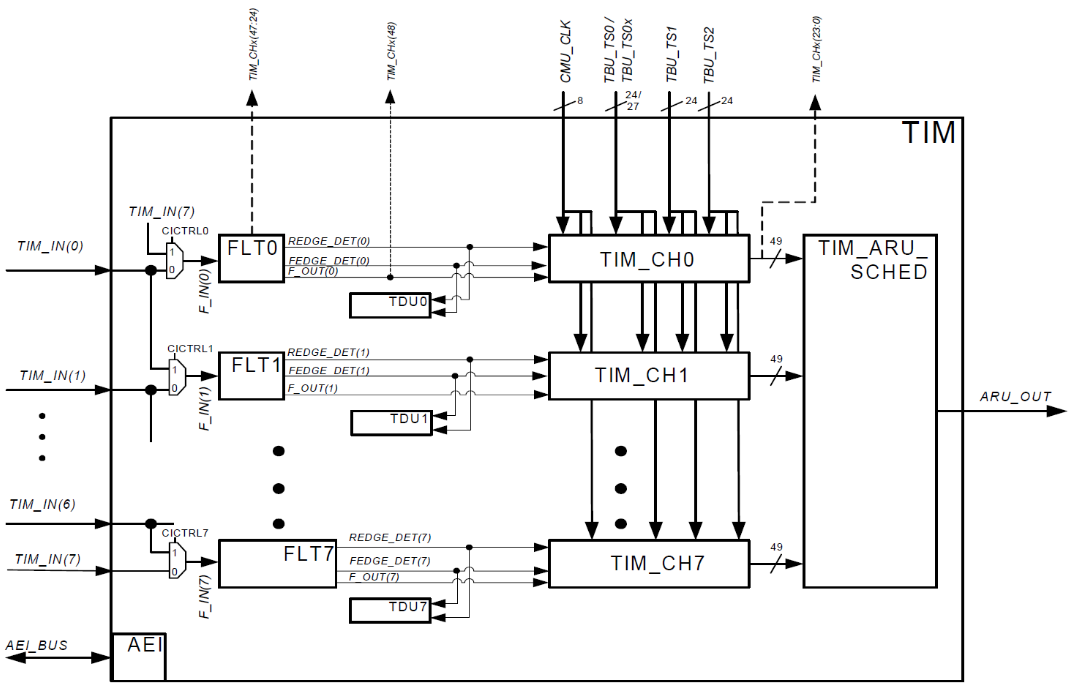


Figure 5.12.: Architecture of the Timer Input Module (TIM)[18]

[17]For Filter block FLTn, the input signal TIM_IN(n) or TIM_IN(n-1) can be selected. For FLT0 TIM_IN(0) or TIM_IN(7) can be selected.
[18]Infineon, 2014a, page 2853

The Architecture of the TIM is depicted in figure 5.12. The three dashed lines represent the signal outputs from TIM0 only, to the TIM0 Input Mapping Module (MAP) of the GTM. Also for TIM0 only the extended time base TBU_TS0 from the Time Base Unit (TBU) is connected to the eight channels. The time base signals TBU_TS1 and TBU_TS2 are connected to all channels of all TIMs.
The TIM Filter Functionality (FLT) provides configurable filter mechanisms for each input signal and provides the subsequent channel with the filtered version of the signal, a signal for the detection of a rising edge in the input signal and one for the detection of a falling edge. Three different modes can be configured and applied individually to the falling and rising edges of the input signal[19]:

- Immediate edge propagation mode
- Individual de-glitch time mode (up/down counter)
- Individual de-glitch time mode (hold counter)

The outputs of FLT are connected to the corresponding TIM channel. Figure 5.13 illustrates the architecture of a TIM channel, where the signals from the FLT are represented by FEDGE_DETx, REDGE_DETx and F_OUTx (the filtered, delayed version of the input signal). The number of falling edges are represented in bits 7 - 1 of the counter ECNT. Together with the signal F_OUT connected to bit 0 of ECNT, it is possible to count every filtered edge, where even counter values of ECNT refer to detected falling edges and odd counter values refer to detected rising edges.
The core part of the channel is the Signal Measurement Unit (SMU). Its clock can be selected by CLK_SEL in the channels control register.
A TIM channel provides 6 different modes:

- TIM PWM Measurement Mode (TPWM)
- TIM Pulse Integration Mode (TPIM)
- TIM Input Event Mode (TIEM)
- TIM Input Prescaler Mode (TIPM)
- TIM Bit Compression Mode (TBCM)
- TIM Gated Periodic Sampling Mode (TGPS)

The TPWM mode is used to measure duty cycle and period of an incoming PWM signal. This mode is used in the AVL EMS software and is explained in the following:
The start of the measurement (polarity of the PWM signal) can be configured. If the configure signal (FEDGE_DETx for falling edge or REDGE_DETx for rising edge) is received, counting with CNT register is started until the opposite edge signal is received. This initiates the copy of the counter value in CNT to the shadow register CNTS (CNTS_SEL = 0).

[19]The modes are not discussed further in the following. Detailed information is given in Infineon, 2014a, page 2856 - 2864.

If the selected edge signal is received, the value in CNT is copied to GPR1[20] and CNT is cleared. The value temporarily stored in CNTS is copied to GPR0. By this, GPR0 contains the duty cycle (positive or negative, depending on the configuration) and GPR1 contains the period of the incoming signal. Data consistency of the registers can be checked by using the bits 7 - 1 of ECNT.
If one PWM period was measured an interrupt can be raised and the data in GPR0 and GPR1 is marked as valid for reading by the ARU.
If the values in GPR0 and GPR1 are not read by either the CPU or ARU when a new period was measured a notification bit is set and the values are overwritten.

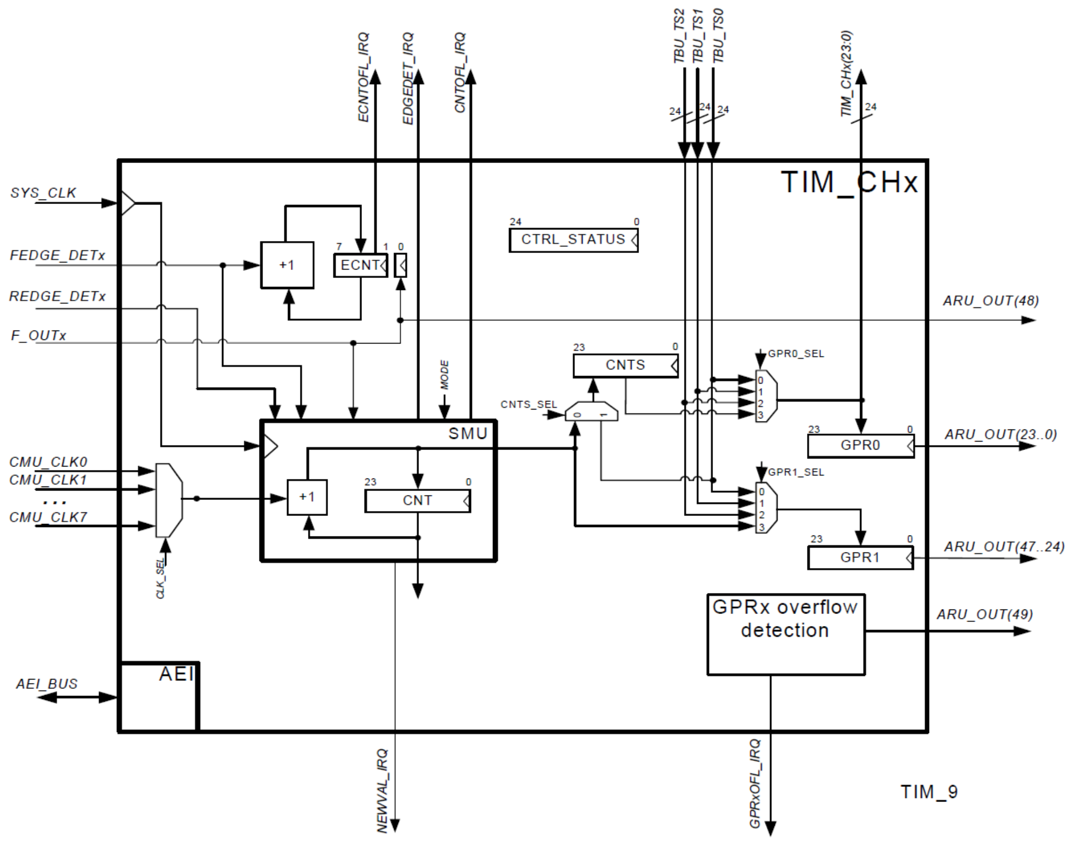


Figure 5.13.: Architecture of TIM channel[21]

[20]The selectors GPR0_SEL and GPR1_SEL has to be set to 3
[21]Infineon, 2014a, page 2868

MSC Connections

The output of one TOM channel can be mapped to SET1, SET2 or SET3 by the configuration of the GTM to MSC Control Registers. Each SET can map 16 outputs of the channels of TOMi (i = 0-2) or ATOMn (n = 0-4) to its 16 outputs. The inputs of the MSC0 and MSC1 are mapped to the outputs of a SET by multiplexers. However MSC0 ALTINL[x] can only be mapped to the output x of one SET (x = 0-16). These multiplexers are configured by MSC0 Input Low Control Register, MSC0 Input High Control Register, MSC1 Input Low Control Register and MSC1 Input High Control Register. Figure 5.14 shows the connections from the Generic Timer Module to the Micro Second Channel.

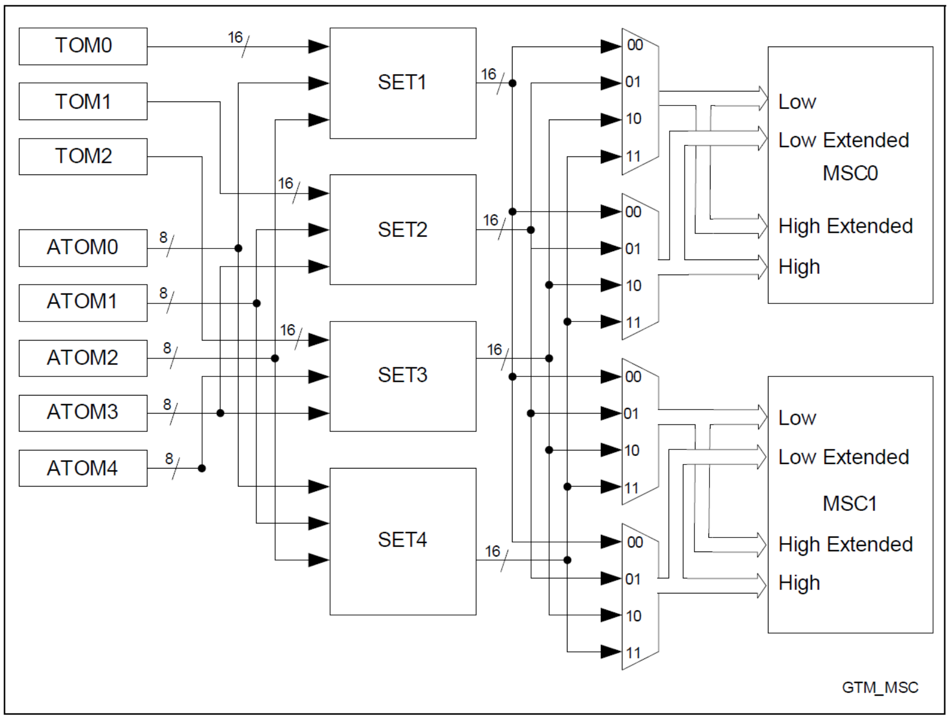


Figure 5.14.: GTM to MSC connections[22]

Port pin connections

Multiple port pins are connected to the GTM as inputs or outputs. The paths are denoted as TINn for input signals to the GTM and TOUTn for output signals from the GTM (n = 9-150). The connection of a port pin to the corresponding path is set in the Input/Output Control Register (IOCR) of the port. However, the inputs and outputs of the GTM are not directly connected to the paths.

[22]Infineon, 2014a, page 3544

The outputs of ATOM and TOM channels are connected to multiple 4-to-1 multiplexers. The output of each multiplexer is connected to one TOUT path. The multiplexers are configured by the Timer Output Select Registers (TOUTSEL).
The inputs of TIM channels are connected to multiple 1-to-2 multiplexers. The input of each multiplexer is connected to one TIN path. The multiplexers are configured by the TIM Input Select Register (TIMnINSEL (n = 0-3)).
For example the connections of port P00.0 of TC277T (BGA-292) are shown in table 5.3. The meaning of this example is that P00.0 can either be configured as input or output. If configured as input the input signal can be connected to the path TIN9. The path can be multiplexed to channel 0 of TIM2 (TIM2_0) or to channel 0 of TIM3 (TIM3_0).
If configured as output the channel 8 of TOM0, channel 0 of TOM1, channel 0 of ATOM0 or channel 0 of ATOM1 can be mapped to P00.0 via the path TOUT9.

Table 5.3.: Connections of port P00.0 to GTM[23]

Port	Input	Output	Input		Output			
			A	B	A	B	C	D
P00.0	TIN9	TOUT9	TIM 2_0	TIM 3_0	TOM 0_8	TOM 1_0	ATOM 0_0	ATOM 1_0
P00.1	...							
...	...							

5.5.2. PWMIO software module

The PWMIO software module consists of the PWMIN, PWMOUT modules and a function for initialization of the Clock Management Unit (CMU). The CMU provides the clock signals for the submodules TIM, TOM and ATOM. The initialization function `PWM_init` sets the global clock signal CMU_GCLK_EN to the system clock by setting the Global Clock Divider accordingly (refer to figure 5.6) and sets FXCLK_SEL to select CMU_GCLK_EN as input for all fixed clock dividers in the Fixed Clock Generation Subunit(FXU). The configurable clock source dividers in the Configurable Clock Generation Subunit (CFGU) are set to divider values, stored in the Array `uint32 dividerCLK[CMU_CLKS]`[24]. The channels of the TOM submodules are clocked by one of the five Fixed Clock signals CMU_FXCLKn. The channels of the ATOM and TIM submodules are connected to one of the eight configurable clock signals CMU_CLKn. After the initialization of the dividers the clock signals are enabled.

[23]Based on table 25-68 in Infineon, 2014a, page 3490
[24]This array is defined as extern to provide the information of the input frequencies to the PWM_IN and PWM_out software modules. Based on this information it is possible to calculate the corresponding time of the duty cycle and period of an input or output PWM signal.

PWMOUT

PWMOUT software module provides functions to set the period and duty cycle for multiple PWM output signals. The signals are mapped to either a Port Pin or to the MSC (see 5.6). The module stores all informations concerning a PWM signal in a structure `PWM_config` defined in the file `PWM_OUT.h`. This structure encapsulates the information about the type of the corresponding output module (ATOM/TOM), the memory addresses of the corresponding registers (thereby the assigned module and the channel are defined) and the values for the registers to generate a PWM with the desired duty cycle and period.
For each PWM output signal that should be generated, a element exists in the module-internal array `PWMs` of type `PWM_config`. An example of the statement for the initialization of PWM0 is given in listing 5.3.

```
// Init PWM0 with 50% duty cycle, 1ms period for TOM2 channel 7
PWMs[0] = (PWM_config){{1000,500}, &(TOM2_GO.channels[7]), TOM};
```

Listing 5.3: Initialization of PWM 0

The values for the registers defining the period and the duty cycle are calculated with the internal function `PWM_setRegisterValues`. Based on the period in μs and the duty cyle in ‰ it calculates the necessary register values. Further the function needs the information of the defined clock dividers for the output module (TOM or ATOM) in form of a array.
The two functions building the interface to the PWMOUT module, given in table 5.4, take both a pointer to the address of a variable of type `PWM_setup`. This structure encapsulates the period in μs and the duty cycle in ‰of a PWM signal.

The pointer passed to `PWM_OUT_init` needs to be the address of the first element of an array with the same size of the internal array or in other words the number of PWM signals. The size is defined in `PWM_OUT.h` by the symbol `NUMB_OF_PWMs`.
On the other hand the pointer passed to `PWM_OUT_update` has to be the address of only one element of type `PWM_setup`.

Listing 5.4 provides an example of an application that uses the PWMOUT module and initializes all PWM signals with a duty cycle of 50% and a period of $100\mu s$. After the initialization the period and the duty cycle of PWM4 is changed.

Table 5.4.: Interface to the PWMOUT software module

Function	**Parameters**	**Return**	**Description**
PWM_OUT_init	PWM_setup *p_setup	void	Initalizes the internal array, the Port Pin Connections, the MSC connections and the SFRs of the output mod-ules(TOM/ATOM).
PWM_OUT_update	uint8 pwmNo, PWM_setup *p_setup	void	Sets the SFRs of the channel of the corresponding PWM to the calculated values for the desired duty cycle and period.

```
#include "PWM.h"
...
int i;

PWM_init();          //init CMU
PWM_setup PWMSetup[NUMB_OF_PWMs];

for(i=0; i<NUMB_OF_PWMs; i++)
{
  PWMSetup[i] = (PWM_setup){.period_us = 100, .dutyCyle_10 = 500};
}
PWM_OUT_init(&PWMSetup[0]);
PWMSetup[4].period_us = 200;
PWMSetup[4].dutyCyle_10 = 700;
PWM_OUT_updatePWM(4, &PWMSetup[4]);
..
```

Listing 5.4: Example of use of the PWMOUT module

PWMIN

PWMIN software module provides the possibility to read the period and the dutycycle of a PWM signal connected to a specified Port Pin by using the Timer Input Modules (TIMs). The structure of PWMIN is similar to PWMOUT. A function for the initialization of the channels and the connections to the port pins is provided. This function takes no parameters, since the channels are initialized in a static manner.
The function `PWM_IN_getVal` is used to read the period in μs and the dutycyle in ‰of

a specified PWM. Inside the module the values of all input PWMs are stored in an array `PWM_inputs` of structure type `PWM_info`. This type encapsulates the duty cycle as unsigned 16 bit value and the period as unsigned 32 bit value. `PWM_IN_getVal` takes the number of the PWM input signal to be read as parameter and reads the GPR0 and GPR1 registers of the corresponding TIM channel. Then it calculates the respective time value for the period and the resulting duty cycle in ‰and updates the values in the `PWM_inputs`. The function returns the address of the corresponding element in the array. The two functions provided for use in ASCET are summarized in table 5.5.
An example of reading the period and the duty cycle of a signal assigned to PWMIN[0] is given in listing 5.5.

```
#include "PWM.h"
...
PWM_init();     //init CMU
PWM_IN_init();
PWM_info *pwmIn0 = PWM_IN_getVal(0);

uint32 period0 = pwmIn0->period;
uint16 duty0 = pwmIn0->duty;
..
```

Listing 5.5: Example of use of the PWMIN module

Table 5.5.: Interface to the PWMIN software module

Function	**Parameters**	**Return**	**Description**
PWM_IN_init	void	void	Initalizes the Port Pin Connections,and the SFRs of the TIMs.
PWM_IN_getVal	uint8 pwmNo	PWM_info *p_info	Reads the values in the registers of the corresponding TIM channel, converts it and stores the converted values in the internal array. The address of the element in the array is returned.

5.6. Microsecond Channel

The Microsecond Channel (MSC), often also referred to as microsecond Bus (usBus or μBus) is an asymmetric serial interface, designed for short distance communications between a master and multiple slaves.[25] MSC is especially designed to connect external power devices to a microcontroller. The main advantage of the MSC is the reduction of required Port Pins to connect power devices to a microcontroller.
In our application the MSC is used to transmit PWM signals to on board 18 Channel Smart Lowside Switches[26]. Therefore only the Downstream Channel of the MSC is used to distribute PWM signals from (A)TOM channels to the Lowside Switches.

5.6.1. TC277T Micro Second Channel Interface

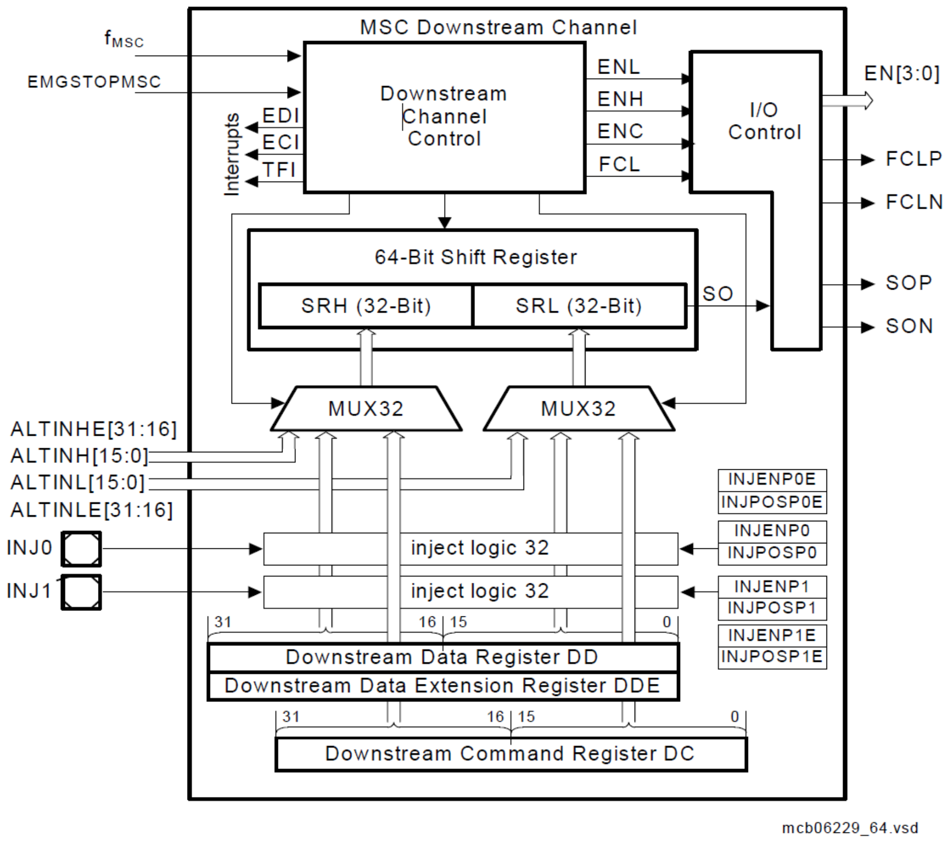


Figure 5.15.: Blockdiagram of the MSC Downstream Channel[27]

[25]Wikipedia, 2015
[26]IFX TLE6244 Infineon, 2003
[27]Infineon, 2014a, page 2051

A downstream channel frame consists of a selection bit followed by the bits shifted out from the shift register and a passive phase where at least two clock cycles have to be provided. The basic timing of a downstream channel frame is illustrated in figure 5.16. Since AVL EMS uses the Downstream functionality of the MSC Interface this functionality is discussed in the following.
Figure 5.15 gives an overview of the MSC Downstream Channel. It includes a 64 bit shift register that is divided into two 32 bit parts - SRL and SRH. The source of the data in the shift register is either the downstream data register (DD), the downstream command register (DC) or the two 16-bit wide input signal buses ALTINL and ALTINH. The MSC module can operate in two modes: standard (up to 32 data bits) or extended (up to 64 data bits).
ENL, ENH and ENC are enable signals from the Downstream Channel Control to the I/O Control. The I/O control combines these signals to four enable/select outputs EN[3:0]. Slaves with different clock polarity can be connected to one of the two serial clock outputs FCLP (positive polarity) FCLN (negative polarity). Different interrupts are provided by the Downstream Channel Control. The Injection Logic (shown in figure 5.15) provides the possibility to inject two input signals from the port pins INJ0 and INJ1 at two user specified data bit positions in a data frame.

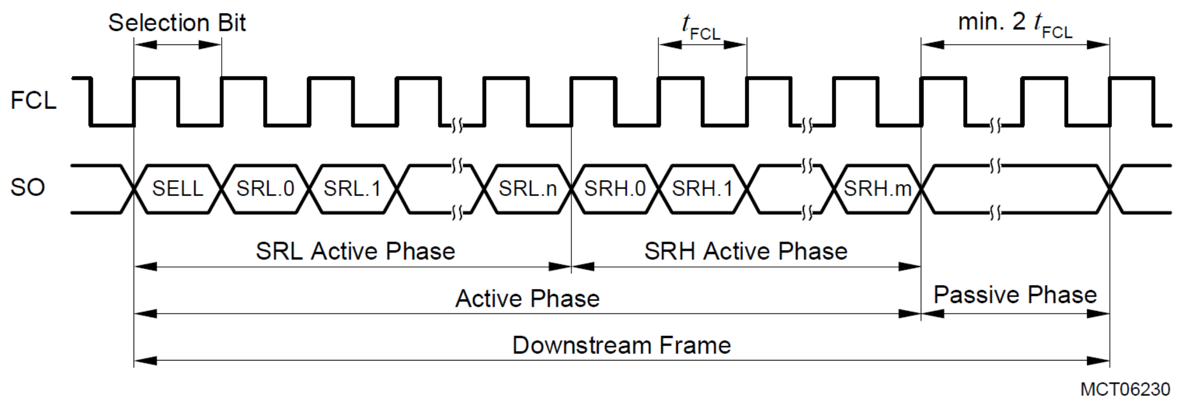


Figure 5.16.: Downstram Channel Frame[28]

The upstream channel of MSC is usually used for sending diagnostic information from the peripheral devices to the microcontroller. One out of eight input lines are used as serial data input (SDI) signal for the upstream channel. The protocol is based on the standard based asynchronous data transfer protocol. However AVL EMS uses the additional serial peripheral interface for diagnostic purposes.

[28]Infineon, 2014a, page 2052

5.6.2. MSC software module

Since the Micro Second Channel Interface of TC277T only needs to be configured once and then passes all connected signals to the bus, the software module consists of only one function, namely `MSC_init`. The function parameter/return value is `void`/`void`. It configures the interface for selection of ALTINH and ALTINL as source for the shift register and a standard transmission mode (32 bits), sets the frequency divider and configures the Port Pins used for the transmission. An example for the MSC transmission is given in 6.2.

Configuration of MSC in AVL EMS

The MSC interface's downstream channel needs to be configured as follows:

- clock signal is set to $1MHz$
- enable signals (ENL and ENH) are low active
- 16 bits are transmitted at both active phases (SRL active phase and SRH active phase)
- both phases start with a low level selection bit
- 3 clock cycles passive phase added after SRH active phase

To illustrate the configuration a data frame of the downstream channel was measured, where the bits SRL.6/SRH.6 are set to '1' and all other bits are set to '0'. The measured signals are depicted in figure 5.17.
It can be seen that the period of the clock signal $t_{FCL} = 1\mu s$. The SRL active phase starts with a low level selection bit followed by the 16 bits of SRL. During the transmission of SRL the corresponding enable signal ENL is active (low).
At the transition from SRL active phase to SRH active phase, ENH becomes active and ENL becomes inactive. In SRH active phase the low level selection bit is followed by the 16 bits from SRH, where at the position of bit SRH.6 the SO (Serial out) signal is high.
At the end of the active phase the frame is completed by the passive phase - both enable signals are inactive for 3 clock cycles. After the passive phase the next frame begins.

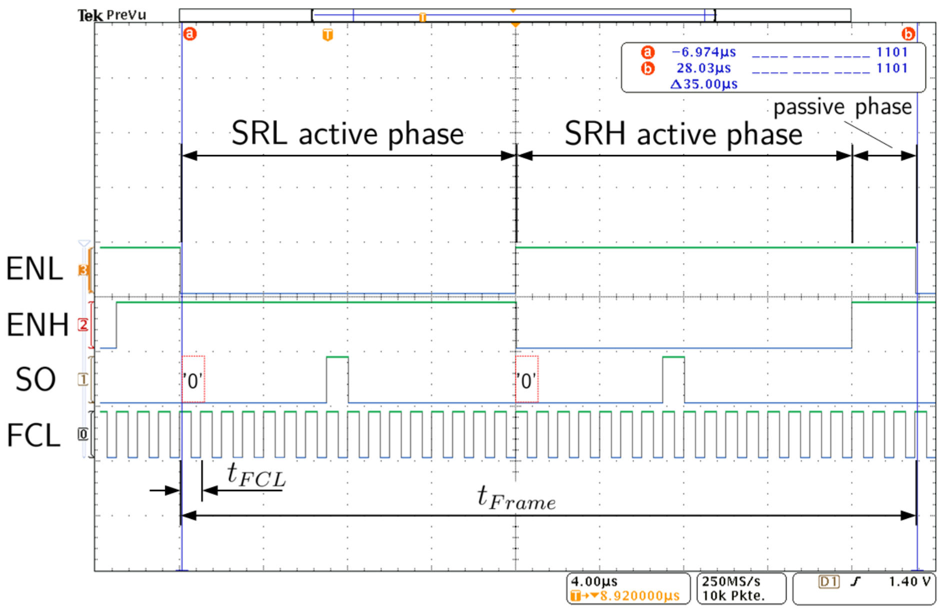


Figure 5.17.: MSC downstream data frame

The length of a frame is defined by the number of selection bits, the number of bits to transmit and the number of clock cycles for the passive phase. In the example above the length of one frame is $t_{Frame} = 35\ t_{FCL} = 35\mu s$.
This means that the minimum time interval for updating one specific bit in a peripheral device is $35\mu s$.

5.7. Controller Area Network

The Controller Area Network (CAN)[29] is an asynchronous, multi-master serial data communication protocol with a maximum bus speed of 1 Mbit/s. A CAN bus consists of at least two participants, which are called nodes. Although CAN is referred to as an asynchronous protocol, because no clock signal is used, the nodes synchronize themselves by analyzing the transitions from recessive to dominant state and compare the transition events with a multiple of the nominal bit time. The Non-Return-To-Zero method and the method of bit stuffing are applied for coding the bit streams in a message.
Compared to the layered OSI model[30] the relevant layers for CAN are the physical, the data link and the application layer. On the physical layer exist three main variants, where the *High speed CAN* (defined in ISO 11898-2) is the most important one. The nodes are connected via two twisted pair wires (CANH and CANL), which are terminated at each end with 120Ω resistors in order to prevent reflections. The logic of the bus corresponds to a wired-AND mechanism, where recessive bits representing logic 1 are overwritten by dominant bits (logic 0). The signaling is differential, what makes CAN robust against noise. In the recessive state both wires are passively biased to $2.5V$. In the dominant state CANH is set to $3.5V$, while CANL is set to $1.5V$ and a differential signal of $2V$ is generated. This is illustrated in figure 5.18.

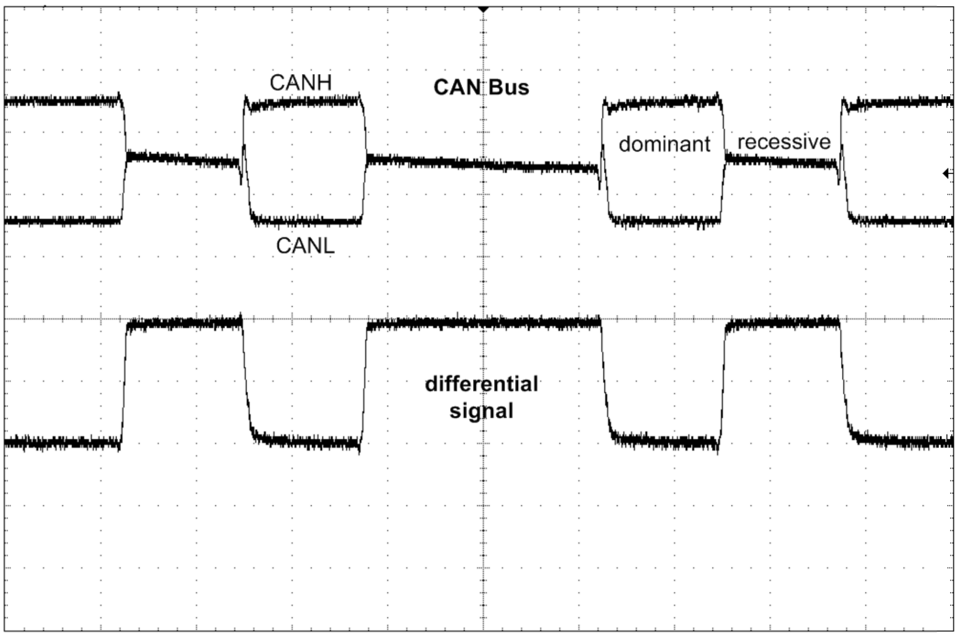


Figure 5.18.: CAN Dominant and Recessive Bus States

[29]The section about CAN follows the description in Bosch, 1991, Corrigan, 2008 and Infineon, 2014a

[30]The Open Systems Interconnection model (OSI model) is a conceptual model that characterizes and standardizes the communication functions of a telecommunication or computing system ISO/IEC, 1994.

CAN is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority, what means that a node is only allowed to start sending a message after a given time slice since the last message was sent over the bus. If two nodes concurrently start to send a message, after the *start of frame field* the *arbitration field* (the format of a data-frame is shown in figure 5.19) that contains the unique identifier. The identifier is intended for the identification of every message. Since the connection to the bus corresponds to a wired-AND mechanism, a low-order identifier "wins" against a higher one.
A CAN data frame consists of seven different bit fields:

- **Start of frame:** A single dominant bit to mark the beginning of a data frame.

- **Arbitration field:** Consists of the identifier and a dominant RTR bit (Remote Transmission Request Bit). The identifier length is 11 bits for standard identifiers or 29 bits for extended identifiers.
  If the message is an extended message, the RTR bit is replaced by a recessive substitute remote request bit (SRR), followed by an recessive identifier extension bit (IDE). The IDE bit indicates the extended identifier. After this bit the remaining 18 bits of the extended identifier follow.

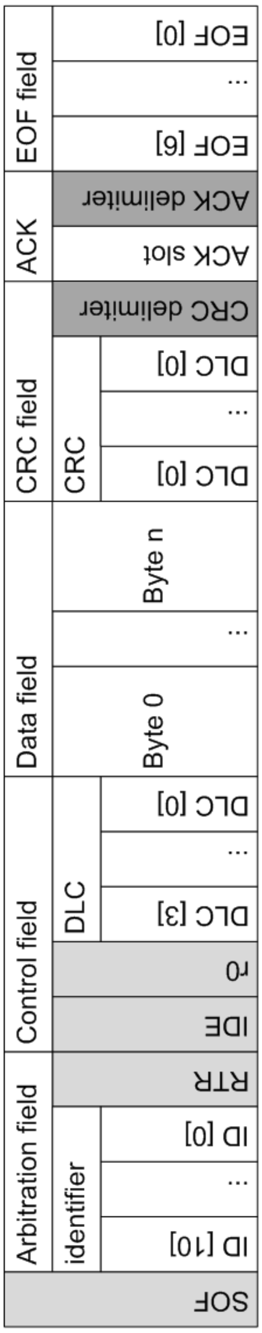
- **Control field:** Contains the identifier extension bit (IDE), 1 reserved dominant bit and the 4 bit wide Data Length Code (DLC) . The IDE bit is dominant if a standard identifier is used, otherwise it is recessive.
  In case of an extended identifier message the IDE belongs to the arbitration field and is replaced by an additional dominant reserved bit.
  The DLC indicates the number of bytes in the data field.

- **Data field:** This field consists of the transferred data within a data frame. The size is from 0 - 8 bytes (MSB first).

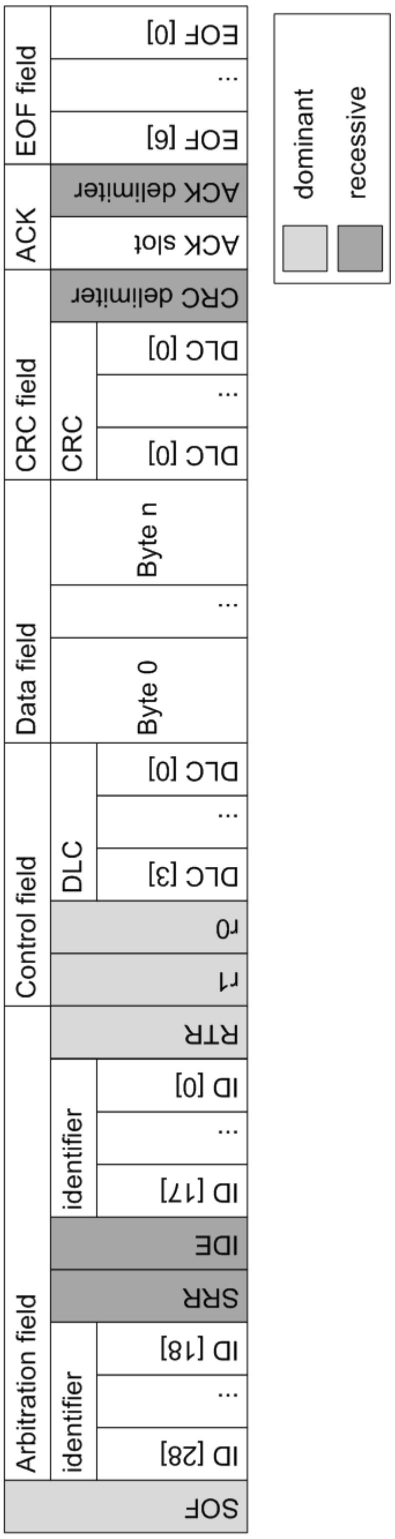
- **CRC field:** Contains a 15 bit wide cyclic redundancy code followed by a recessive delimiter bit.

- **ACK field:** This field is composed of two bits - the ACK slot and the recessive ACK delimiter bit. The transmitting node sends two recessive bits. A receiver node, which received the message correctly, indicates this by sending a dominant bit during the ACK slot.

- **End of frame:** Delimits a data frame by a flag sequence of seven recessive bits.



(a) Standard identifier



(b) Extended identifier

Figure 5.19.: CAN data frames

5.7.1. TC277T Controller Area Network Controller

As described in Infineon, 2014a, the MultiCAN+ module in Infineons AURIX products implements 4 CAN nodes, representing 4 serial communication interfaces. The nodes share a set of 256 hardware message objects, where message objects are assigned to one specific node. For this assignment every node has its own list of message objects.
The nominal bit timings for the CAN nodes are derived from the module [Baud Rate Clock Block]. Figure 5.20 gives an overview of the MultiCAN+ module.

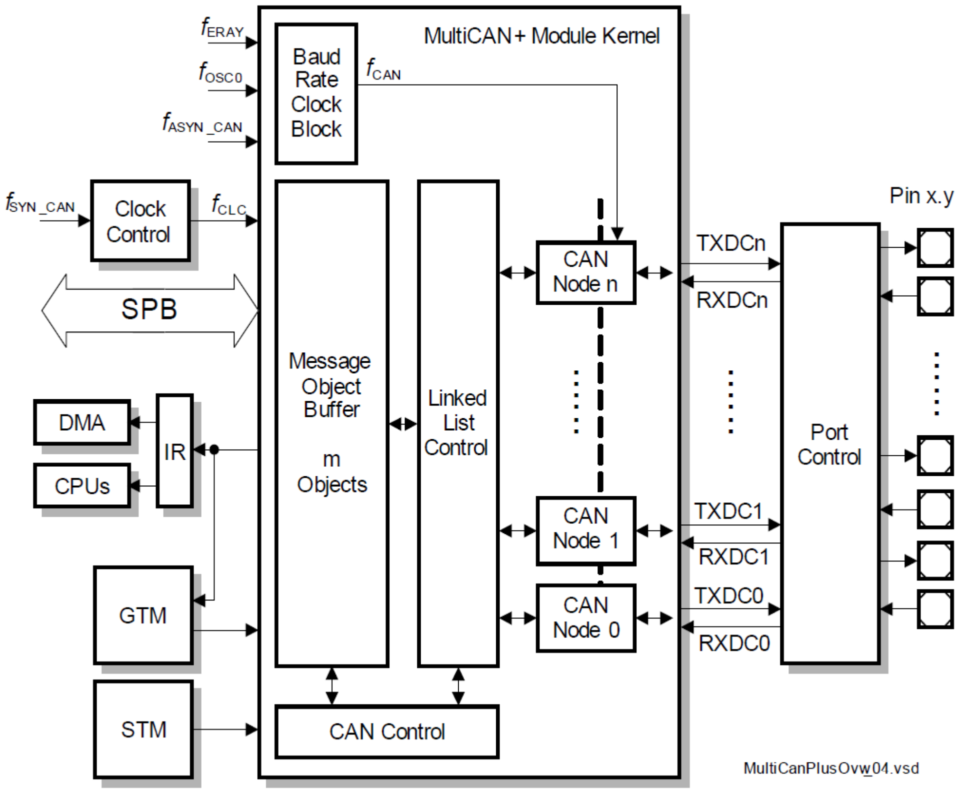


Figure 5.20.: Overview of the MultiCAN+ Module[31]

The hardware message objects represent the messages to be sent or received. Their properties are configured by setting the corresponding special function registers (SFRs). Each hardware message is configured by multiple SFRs. Data to be transmitted or received data are stored in two 32 bit wide registers.
A message is allocated to a CAN Node by the configuration of the linked list. The configuration of the linked list is done by writing specific commands to the Panel Control Register. In order to allocate a message to a CAN node the command *Static Allocate* with the message

[31]Infineon, 2014a, page 2171

object number and the number of the node as argument has to be written to the Panel Control Register.

5.7.2. MultiCan software module

The nodes on the CAN bus differ with respect to the application. Therefore the MultiCan software module needs to provide the possibility to adapt its behavior to different bus configurations. This means that there has to be a possibility to change the baud rate of each CAN node and the properties of the message objects.
The MultiCan software module consists of the two files `MultiCan.c` and `MultiCan.h`. For the basic initialization of the MultiCAN+ module and its nodes the function `CAN_Init` is provided. The initialization of a hardware message object is supported by the function `CAN_AddMO`. This function takes a pointer to a structure `CAN_MO`. The structure encapsulates all necessary information for the configuration of a hardware message object. The members of the structure are given in table 5.6.

Table 5.6.: Members of structure CAN_MO

Name	**Type**	**Description**
moNumber	uint8	number of the corresponding hardware message object
direction	enumeration CAN_Direction	either *transmit* or *receive*
identifier	uint32	holds the object-identifier of the message
dlc	uint8	Data Length Code - defines the number of bytes in the CAN-frame
acceptenceMask	uint32	holds the acceptance mask of the message
data[]	uint8	byte array that holds the data to send or the received data
node	enumeration CAN_Node	defines on which CAN node of TC277T the message is sent/received

The structure `CAN_MO` reflects all the properties required to send or receive data to/from a CAN node. Based on this information, passed as parameter to the function `CAN_AddMO`, the registers of the specific hardware message object are initialized. Listing 5.6 shows an example of the initialization of a message object from the application software. A message object is declared and initialized to transmit data with `0x1D7` as extended identifier. It is assigned to the hardware message object 8 and allocated to node 0 of MultiCAN+ module. The DLC is set to 8, which means that the message consists of 8 data bytes. By passing the base address of the structure to the function `CAN_AddMO` the corresponding SFRs of hardware message object 8 are initialized accordingly. The message object is allocated to node 0 by writing the *Static Allocate* command to Panel Control Register.

```
CAN_MO mo1;
mo1.direction = transmit;
mo1.identifier = 0x1D17;
mo1.moNumber = 8;
mo1.node = CAN_Node0;
mo1.acceptenceMask = 0x3FFFFFFF;
mo1.dlc = 8;
CAN_AddMO(&mo1);
```

Listing 5.6: Adding a message object to CAN

For sending an initialized message object, the array `data` of the structure `CAN_MO` has to be set. The values are written to the registers of the hardware message with the function `CAN_SetMOData`. The message is actually sent by calling the function `CAN_Send`.
A receive message is initialized in the same way (direction is set to `receive`). The reception of a message can be determined by calling the function `CAN_GetMOData`. It returns true if a message was received and in this case sets the array `data` in the passed software message object to the received data. A detailed example can be found in 6.1.
Table 5.7 lists the functions provided by the CAN software module.

Table 5.7.: Interface to the CAN software module

Function	**Parameters**	**Return**	**Description**
CAN_Init	uint32 BaudNode0 uint32 BaudNode1 uint32 BaudNode2 uint32 BaudNode3	void	Initalizes the MultiCAN+ module, the nodes to the baudrates defined by the parameters and the Port Pin connections.
CAN_AddMO	CAN_MO *p_MO	void	Initializes the corresponding hardware message object according to the properties defined in the software message object CAN_MO at the address p_MO.

Table 5.7.: (continued) CAN SW module

Function	**Parameters**	**Return**	**Description**
CAN_Send	CAN_MO *p_MO	void	Sends the message objects that corresponds to the software message object CAN_MO at the address p_MO.
CAN_SetMOData	CAN_MO *p_MO	void	Writes the data stored in the array data of the software message object CAN_MO at the address p_MO to the corresponding hardware message object.
CAN_GetMOData	CAN_MO *p_MO	bool	Checks if a message with the defined identifier is received. If this is the case the data is copied from the hardware message object to CAN_MO at the address p_MO and the functions returns true.

5.8. Serial peripheral interface

For the operation of the EMS communication with several on-board-chips is necessary. The peripheral devices can be accessed by the serial peripheral interface.

The Serial Peripheral Interface (SPI) is a synchronous interface defined by Motorola. It allows the interconnection of microcontrollers and peripherals. In general four wires are required for data, clock and the selection of the devices. The naming of the SPI signals varies depending on the manufacturer. Table 5.8 shows the corresponding names, used by Infineon and Motorola.

Table 5.8.: SPI naming conventions

Motorola	**Infineon**
MISO (Master In Slave Out)	MRST (Master Receive Slave Transmit)
MOSI (Master Out Slave In)	MTSR (Master Transmit Slave Receive)
SCLK (Serial Clock)	SCLK (Serial Clock)
SS (Slave Select)	SLSO (Slave Select Output line)

Signal Descriptions

The four basic signals are discussed in the following, using the naming convention of Infineon.

- **MRST** In the master device this line is configured as input while in a slave device it is configured as output. Data is therefore transferred over this line from a slave to the master. If a slave device is not selected it has to place this line to high-impedance state.

- **MTSR** The master devices uses this line as output and slave devices as input. Data is sent from the master device to a slave device over this line.

- **SCLK** This signal is provided by the master (output) to the one or more slaves (input) and is used to synchronize data movement over MRST and MTSR.
  Four timing relationships are possible by changing the polarity or the phase (with respect to data signals) of this line.

- **SLSO** To select a slave device, the master device sets this line to low or high (depending on the slave device) for the duration of the transaction.

Functional description

At the beginning of a data transmission the master device selects one slave device by the SLSO line and generates the clock signal, connected to a shift register in the master device and to the SCLK line. In the slave device this clock signal is also provided to a shift register. Data from the master device' shift register is shifted into the slave device' shift register via MTSR line. The data from the slave device is shifted to the master device via MRST line. Figure 5.21 shows this principle of a duplex transmission.

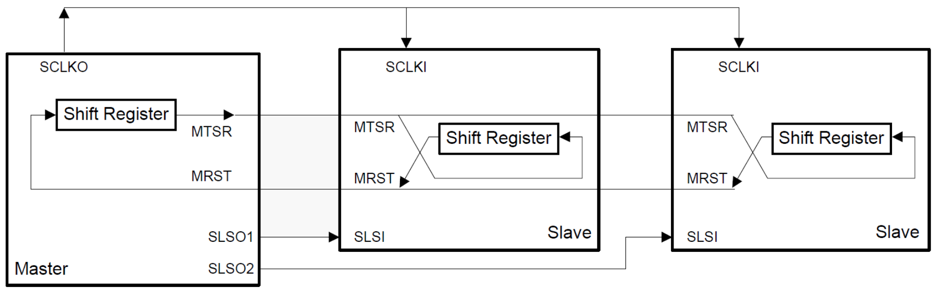


Figure 5.21.: Duplex connection of SPI[32]

5.8.1. TC277T Queued Synchronous Peripheral Interface

Infineons TC277-C offers four Queued Synchronous Peripheral Interface (QSPI) modules for communication with external SPI devices. Each QSPI can operate in either master or slave mode. In master mode several slave select signals are available for connecting multiple devices to one interface. The term "Queue" describes the functionality implemented for switching the timing of the communication with the selected slave device. The 32 bit configuration for the communication and the 32 bit data are queued in a 32 bit TxFIFO buffer (First In First Out). The 32 bit configuration and the 8 bit configuration extension register (ECON) of a QSPI module define together the full configuration of the module. Each of the eight ECON registers are used for the configuration of the communication with devices connected to two out of 16 slave selects (ECON0 for SLSO0 and SLSO8, ECON1 for SLSO1 and SLSO9, etc.). A configuration entry in the TxFIFO is distributed to the basic configuration register (BACON) and is connected to one ECON. Data entries in the queue are copied to the shift register. Based on the configuration the next data entry in the queue is shifted to MTSR with the selected configuration (timing configuration, data width and SLSO activation). At the end of the shifting process, received data is copied from the shift register to the RxFIFO. Figure 5.22 depicts the architecture of the QSPI.

[32]Infineon, 2014a, page 1772

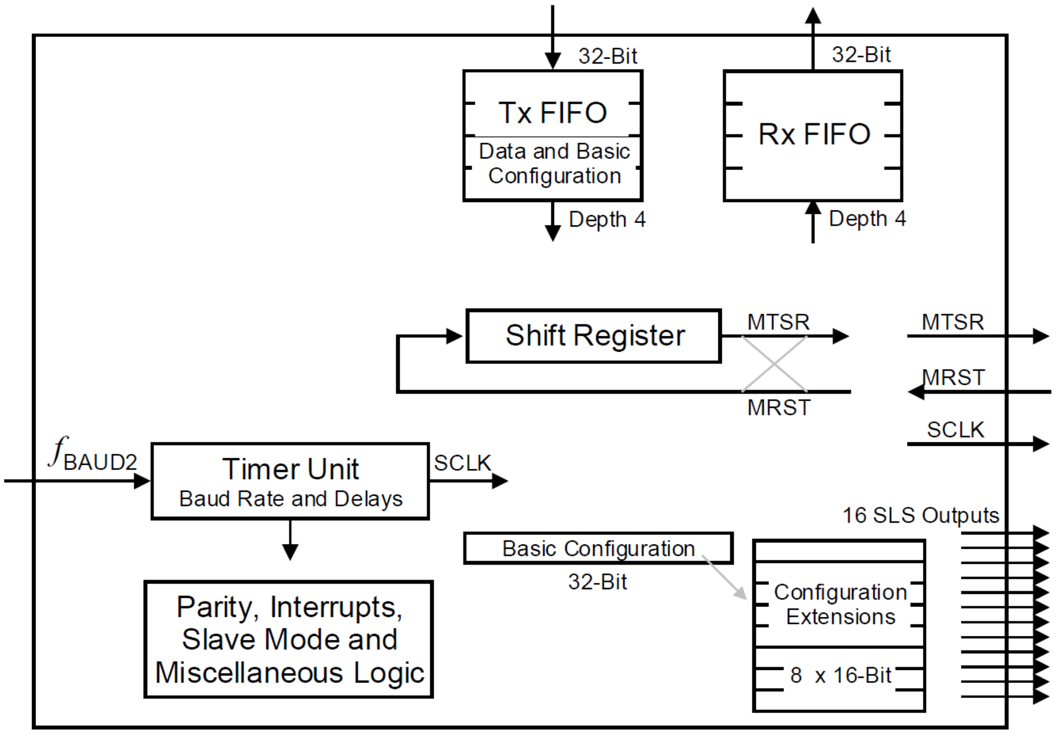


Figure 5.22.: Architecture of the QSPI[33]

5.8.2. SPI software module

The software module SPI takes use of Infineons Low Level Drivers (iLLDs). It is divided into two sublayers A and B. In sublayer A the QSPI module takes place, which handles all interactions with the peripheral devices connected to one of the interfaces of TC277T. The purpose of this module is to abstract the functions provided by the iLLDs (Infineon Low Level Drivers), and provide appropriate functions to sub-layer B. Figure 5.23 illustrates the connections of the software module QSPI in the HAL and interface to the file `IfxQspi_SpiMaster.c` of the iLLDs.
The functions of the iLLD, used by QSPI module are summarized in table 5.9. For basic initialization of a QSPI hardware module the function `IfxQspi_SpiMaster_initModule()` is used. The second argument `*config` is provided by the file `QSPI_cfg.c`. The file provides the configurations of the QSPI modules and channels. Based on the passed base address of the configuration, the QSPI module is initialized. The first argument represents a connection to the QSPI module and is stored in an element of the internal structure `QSPIn` (n = 0 - 4) of the QSPI software module.

[33]Infineon, 2014a, page 1774

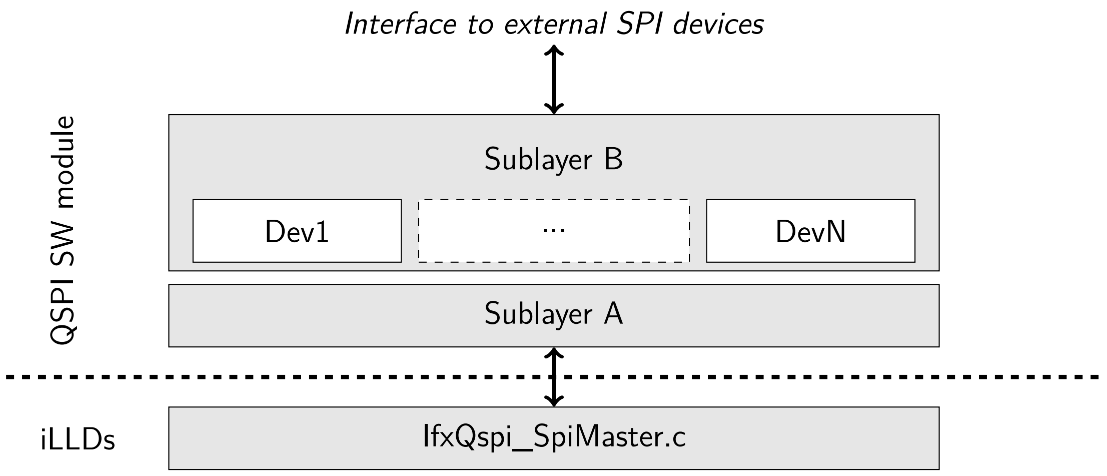


Figure 5.23.: Architecture of the SPI software module

The channels represent the external devices and hold the device specific information (the QSPI hardware module the device is connected to, the slave select port pin, etc.). A channel is initialized by the function `IfxQspi_SpiMaster_initChannel()`. Like the initialization function of the hardware module it takes the configuration variable as parameter and initializes the channel accordingly. The connection to the channel is stored at the base address of `*chHandle` in the structure `QSPIn` (n = 0 - 4) of the QSPI software module.

Table 5.9.: Functions of iLLD used by QSPI module[34]

Name	**Parameters**	**Return**
SpiMaster_initModule	SpiMaster *handle SpiMaster_Config *config	void
SpiMaster_initChannel	SpiMaster_Channel *chHandle SpiMaster_ChannelConfig *chConfig	SpiIf_Status
SpiMaster_exchange	SpiMaster_Channel *chHandle void *src void *dest sint16 count	SpiIf_Status

For communication via SPI the iLLD provides the function `IfxQspi_SpiMaster_exchange`. The function takes a pointer to a variable of type `IfxQspi_SpiMaster_Channel` as argument. Thereby the device for communication is defined. The base address of an array is passed to the function, holding the data to be sent. A second base address of the same type defines where received data shall be stored. The last argument defines the number of

[34]For a compact illustration the prefix *IfxQspi* in the names of the functions and data types is omitted.

elements in the array.
To provide the possibility to send data to SPI devices, while the interface is busy, the QSPI software module is implemented by means of a queue. Further data to be sent can be marked with two priority levels. For this purpose, all the information necessary for communication with an external device[35] are encapsulated in the structure `QSPIMessageObject`.
To send and receive data, a message object is stored in a queue by calling the function `QSPI_QueueMO()`, which takes the address of a message object and a priority (*high* or *low*) as parameters. The function stores the address of the message object and the priority into the next free element of `QSPI_nodelist`. `QSPI_nodelist[]` is a array of structure type `QSPINode`, with the three members `message` (address of the relevant message object), `priority` (*high* or *low*) and `next` (address of the next node).
For each of the four interfaces of the hardware QSPI module a corresponding global variable `QSPI_Queue0 - QSPI_Queue3` exists. They are of type `QSPIQueue` and hold the address of the first element in `QSPI_nodelist` to be sent, an array of size 2 with the addresses of the last element of priority *high* and *low* and the current size of the queue. The flowchart given in figure 5.24 of the function `QSPI_QueueMO()` illustrates how a `QSPIMessageObject` is queued. The function takes the address of the message object and the priority (*high* or *low*) as arguments. These parameters are encapsulated in a variable of type `QSPINode`. The pointer `next` of the node (which is intended to point to the next node to send) is first set to zero. By comparing the counter for the size of the nodelist with zero, it is detected if this node is the first node. In this case the pointer `head` of the queue is set to the address of the node in the node list. Since it is the only node the corresponding tail pointer (one for high priority and one for low) is also set to the address of this node.
If the queue is not empty, the priority of the node at the address stored in `head` of the queue is compared with the priority of the current node, to detect if this node is the first one with high priority. If this is the case, the pointer `next` is set to the same address `head` of the queue is pointing to and head is set to the new node. This means that the new node is inserted as first node to send. If the new node is not the first node with high priority it is checked if there is already a node with the same priority in the queue. If this is not the case, the new node has to be inserted after the last node of high priority. Otherwise it is checked if the existing node is marked with high priority and if this holds true, the new node is inserted as last node of high priority. In both cases `next` of the new node is set to same address `next` of the existing node was pointing to.
At the end of the function the tail pointer (corresponding to the priority of the new node) of the queue is set to the new node and the size of the queue is incremented.

[35]This includes the corresponding channel, the memory address of data to transmit, the memory address of data to receive and the status of a message

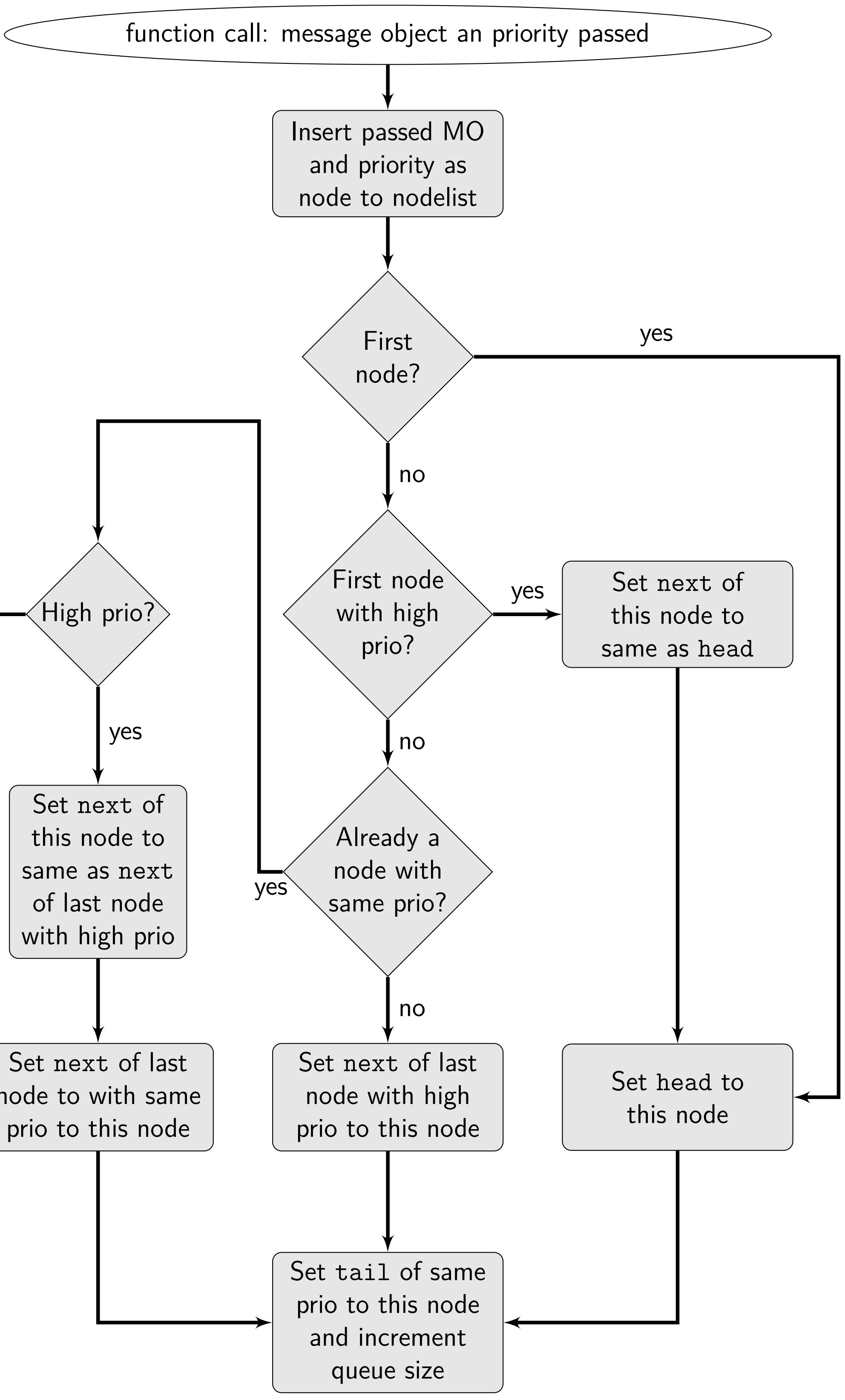

function call: message object an priority passed
Insert passed MO and priority as node to nodelist
First node?
yes
no
First node with high prio?
yes
Set next of this node to same as head
no
Already a node with same prio?
yes
High prio?
no
yes
Set next of this node to same as next of last node with high prio
no
Set next of last node to with same prio to this node
Set next of last node with high prio to this node
Set head to this node
Set tail of same prio to this node and increment queue size

Figure 5.24.: Flowchart of the queue function

Listing 5.7 gives an example of queuing message objects with different priorities. First two message objects (MO) with low priority are added, followed by three MOs with high priority and one with low. This results in a configuration shown in figure 5.25.

```
// Enqueue message objects;
QSPI_QueueMO(&mo1, low_priority);
QSPI_QueueMO(&mo2, low_priority);
QSPI_QueueMO(&mo3, high_priority);
QSPI_QueueMO(&mo4, high_priority);
QSPI_QueueMO(&mo5, high_priority);
QSPI_QueueMO(&mo6, low_priority);
```

Listing 5.7: Example of queueing message objects with different priorities

The head pointer of the queue is pointing to the first node, queued with high priority. The next pointer of this node points to the next node with high priority. Since `mo5` is the last MO inserted with high priority the tail pointer for high priority points to the corresponding node. The tail pointer for low priorities points to the node corresponding to `mo6`.

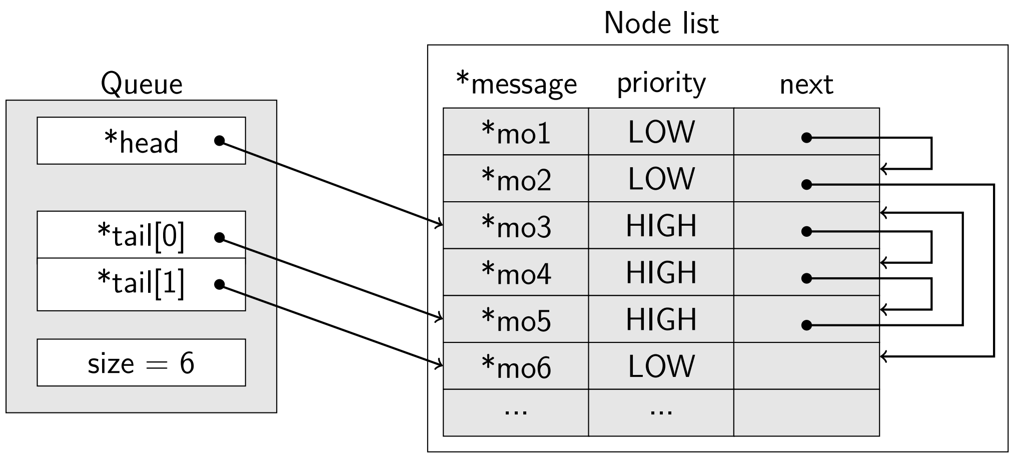


Figure 5.25.: Enqueued message objects

To the first node in the queue the function `QSPI_TransmitBuffer` is provided. It calls the function from the iLLD to exchange data with and passes the data of the message object in the first node to it. The function sets the DMA (Direct Memory Access) to copy the data to be sent to the correspondent QSPI module. After reception of the data from the device DMA is used to copy the data to the receive array of the message object and an interrupt is generated. The head and tail pointers of the queue are set accordingly and the node is removed from the queue. In the ISR of the DMA the functtion `QSPI_TransmitBuffer` is called again. If the queue is not empty the next node is sent. Figure 5.26 illustrates the communication procedure.

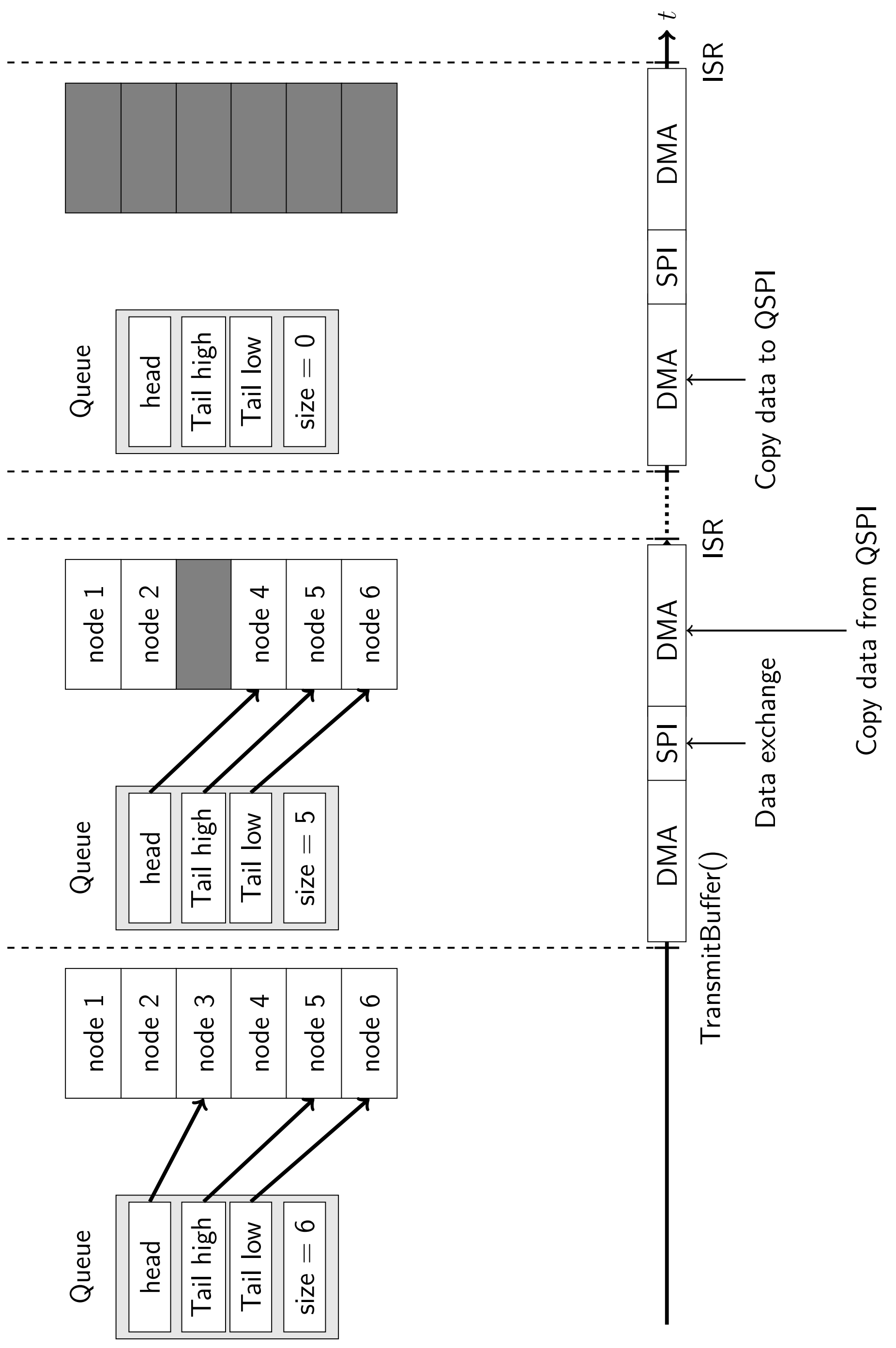


Figure 5.26.: QSPI communication procedure[36]

[36]Refer to example in figure 5.25

Every on-board-chip connected to one of the four interfaces is abstracted by a module placed in sublayer B. The purpose of this modules is to provide abstracted functions for the communication with the peripheral devices to the processes in ASCET. Therefore modules implement the different protocols of the devices. However these functions only queue messages into the corresponding queue, where the sending is triggered in a periodic task in ASCET. An example for such a module is given in 6.2.
The only function of the SPI software module directly accessed from ASCET is the initialization function for the QSPI hardware modules. All other functions are provided by the device modules in sublayer B, which take use of the QSPI module in sublayer A.
Table 5.10 summarizes the functions provided by the SPI and QSPI[37] software module. The return type of all functions is `void`.

Table 5.10.: Interface to the SPI and QSPI software module

Function	**Parameters**	**Description**
SPI_Init	void	Provides access to the initialization function of the QSPI hardware modules and calls QSPI_init.
QSPI_Init	void	Initializes the QSPI hardware modules, port pin connections and a channel for each connected SPI device.
QSPI_TransmitBuffer	void	Transmits the first message object in the queue and updates the queue accordingly.
QSPI_QueueMO	QSPIMessageObject *p_mo QSPIMoPriority priority	Adds a message object in form of a node to the queue.

[37]The QSPI software module is placed in sublayer A and accessed by sublayer B.

6. Evaluation

The functionalities of all implemented software modules have been verified and tested for use with AVL EMS.
In the following the module test of the MultiCAN module is described. It is a representative example of the verification of the implemented modules.
Further the interaction of the MSC, PWMIO and SPI modules was tested. An analysis of the results is included as well.

6.1. MultiCAN module

To evaluate the function of the MultiCan module a project in ASCET was generated. The source files of the module was provided during the build process by adapting the configuration file for the project.
The example project contains an ASCET module providing the processes `init`, `_receive` and `_transmit`. The process `init` is assigned to the initialization task and contains the function call for the initialization of the CAN hardware module of TC277T, provided by the MultiCan module. Further it initializes two message objects of type `CAN_MO` declared in the header of the ASCET module. In the example project Node0 of the microcontroller is initalized to 1Mbaud.
The first message object `mo1` is initialized for transmission from CAN node 0 as the hardware message object 8 of the microcontroller with the extended identifier `0x1D17` and a data length code of 8 bytes.
The second massage object `mo2` is initialized as receive message at CAN node 0, hardware message object 9 and the identifier `0xD12`. The data length code is also set to 8 bytes.
The acceptance mask is set to `0x3FFFFFFF` (all identifiers) for both messages. The function `CAN_AddMO` sets the defined properties to the selected hardware message objects. Listing 6.1 shows the statements of process `init`.

```
CAN_vMultiInit(kbaud_1000, kbaud_500, kbaud_500, kbaud_500);

mo1.direction = transmit;
mo1.identifier = 0x1D17;
mo1.moNumber = 8;
mo1.node = CAN_Node0;
mo1.acceptenceMask = 0x3FFFFFFF;
mo1.dlc = 8;
CAN_AddMO(&mo1);
```

```

mo2.direction = receive;
mo2.identifier = 0x1D2;
mo2.moNumber = 9;
mo2.node = CAN_Node0;
mo2.acceptenceMask = 0x3FFFFFFF;
mo2.dlc = 8;
CAN_AddMO(&mo2);
```

Listing 6.1: CAN Init-process in ASCET

The process `_receive` (the code is shown in 6.2) checks the reception of a message with the identifier of `mo2` by using the function `CAN_GetMOData`. To be able to analyze the received data, while debugging, a statement concerning the data was added to the if-clause.

```
if(CAN_GetMOData(&mo2))
{
  mo2.data[0]++;
}
```

Listing 6.2: CAN receive process in ASCET

Transmitting the message object `mo1` is done with the process `_transmit`. The data in the software message object is set to some values in a for-loop. Then the data is written for the hardware message object by calling the function `CAN_SetMOData` and is actually sent by the function call of `CAN_Send`.

```
for(int i=0; i<8; i++)
{
  mo1.data[i] = 0xA0 + i;
}
CAN_SetMOData(&mo1);
CAN_Send(&mo1);
```

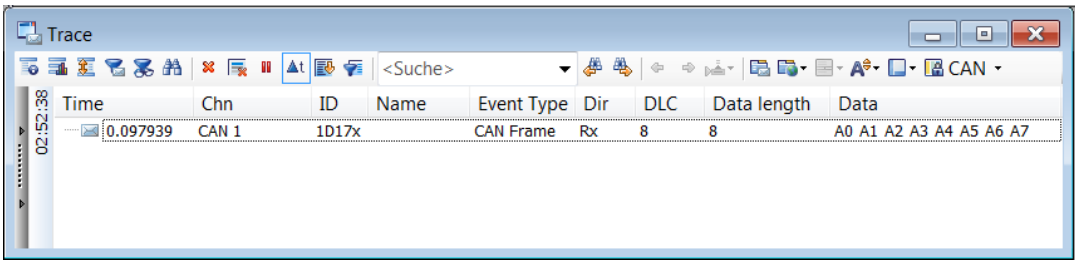
Listing 6.3: CAN transmit process in ASCET

Both processes `_receive` and `_transmit` are assigned to an alarm task executed with a period of $100ms$.

The software was tested on the TriBoard TC2X7 (see 2.1), equipped with the High Speed CAN-Transceiver TLE6250[1]. Channel 1 of the *CANcaseXL* by *Vector* was connected to the CAN0 interface of the Triboard and the software *CANalyzer* by *Vector* was used.

The test of the example project showed that the message with the extended identifier `0x1D17` with the correct data was successfully sent over the CAN bus. The measured trace is illustrated in figure 6.1.

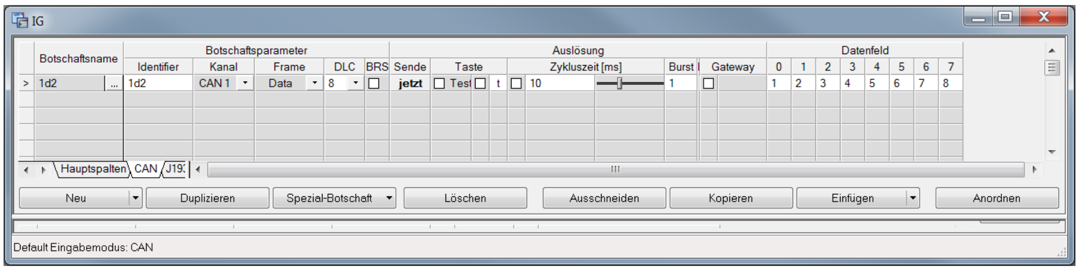
[1]Infineon, 2008b



Time	Chn	ID	Name	Event Type	Dir	DLC	Data length	Data
0.097939	CAN 1	1D17x		CAN Frame	Rx	8	8	A0 A1 A2 A3 A4 A5 A6 A7

Figure 6.1.: Trace of CAN Bus

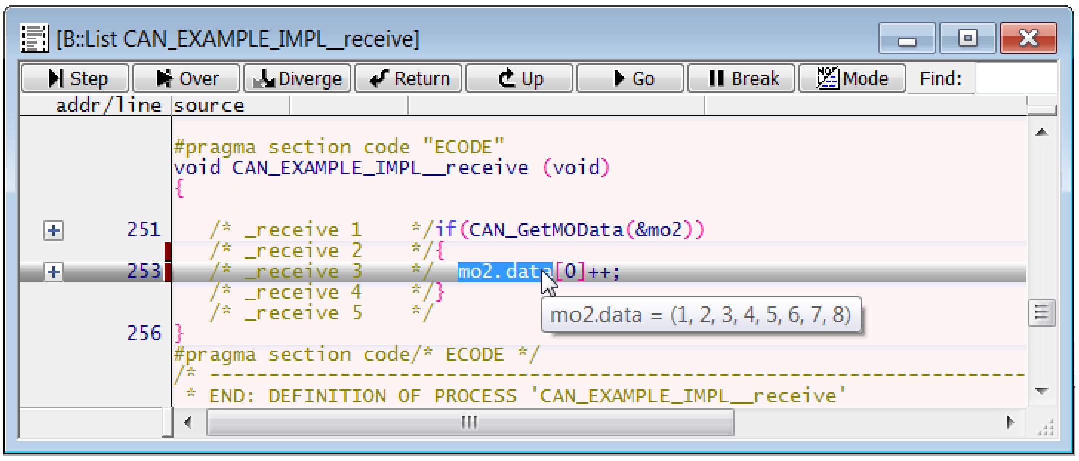
By setting a breakpoint in the debugging software *Trace32* by *Lauterbach* and sending a message with the identifier `0x1D2` and some defined data to the CAN bus with *CANalyzer*, the reception of the correct data could also be confirmed as illustrated in figure 6.2 and figure 6.3.



Botschaftsname	Botschaftsparameter					Auslösung					Datenfeld							
	Identifier	Kanal	Frame	DLC	BRS	Sende	Taste	Zykluszeit [ms]	Burst	Gateway	0	1	2	3	4	5	6	7
1d2	1d2	CAN 1	Data	8	☐	jetzt	☐ Test ☐ t	☐ 10	1	☐	1	2	3	4	5	6	7	8

Default Eingabemodus: CAN

Figure 6.2.: Transmission of CAN message with *CANalyzer*



```
[B::List CAN_EXAMPLE_IMPL__receive]
addr/line source

#pragma section code "ECODE"
void CAN_EXAMPLE_IMPL__receive (void)
{
   /* _receive 1    */if(CAN_GetMOData(&mo2))
   /* _receive 2    */{
   /* _receive 3    */  mo2.data[0]++;
   /* _receive 4    */}
   /* _receive 5    */
}
#pragma section code/* ECODE */
/* ---------------------------------------------------------------------
 * END: DEFINITION OF PROCESS 'CAN_EXAMPLE_IMPL__receive'
```

mo2.data = (1, 2, 3, 4, 5, 6, 7, 8)

Figure 6.3.: Reception of a CAN message in the software of TC277T

6.2. MSC, PWMIO and SPI module

To test the functionalities of the MSC, PWMIO and SPI module a project, which makes use of EMS on-board Low Side Switch IFX TLE6244, was created. The device is connected to one of the EMSs SPI buses for configuration and diagnostic purposes and to one MSC bus (connected to ENH). The data in high phase of the downstream data frame defines the signal level of the output pins of TLE6244.
The IFX TriBoard TC2X7 was connected to the EMS via the pin headers of the Connector PCB, plugged to the EMS main board. Figure 6.4 shows the test setup.

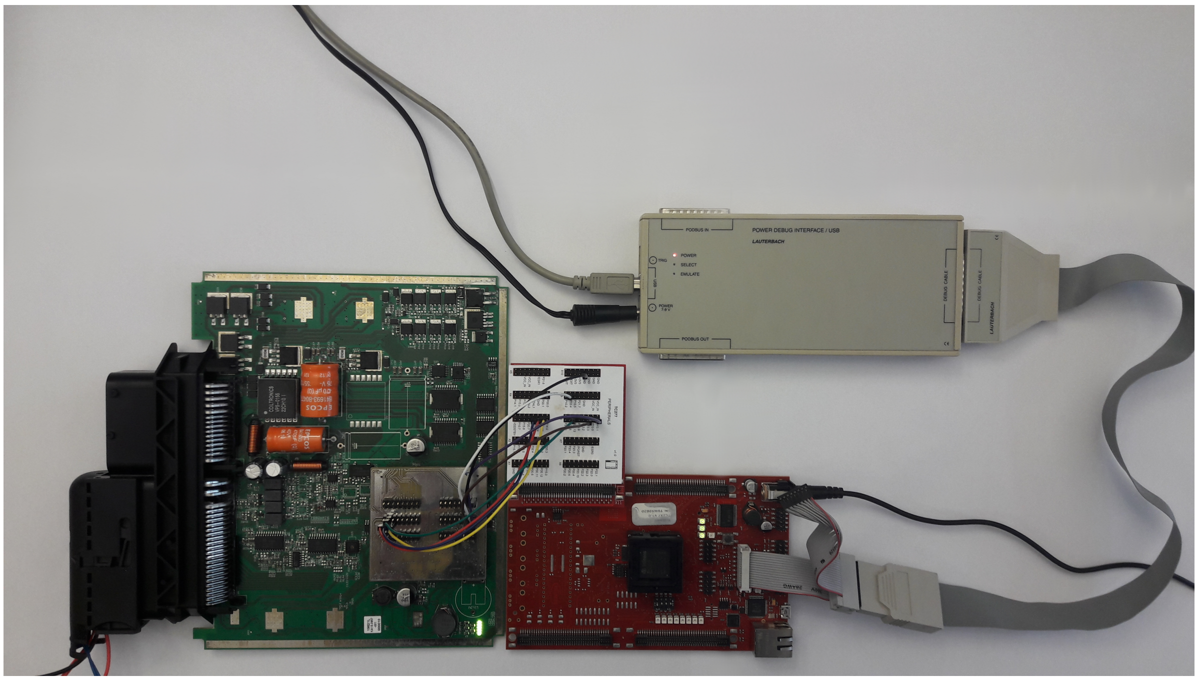


Figure 6.4.: Test setup with TC2X7 connected to AVL EMS main board

The software first initializes the QSPI module and IFX TLE6244 by calling the initialization function from the module TLE6244, placed in sublayer B of the SPI module (refer to figure 5.23). The Low Side Switch is initialized by receiving three 16 bit blocks (`0x2E00`, `0x24FF` and `0x27FF`)[2] via SPI.
The Micro Second Channel (MSC) is initialized according to *Configuration of MSC in AVL EMS* (5.6.2), by calling the function `MSC_init`.
After the initialization of the MSC the GTM module is initialized. All PWM output signals' duty cycles are set to `0` except the one of PWM33.

[2]The initialization process and configuration of TLE6244 are not discussed any further.

PWM33 is generated by TOM1, channel 7. The output signal of TOM1 channel 7 is connected to ALTINH7. This signal level is distributed to bit 7 of SRH register (Shift Register High) of the MSC downstream channel.
TLE6244 maps the input from the SRH active phase of MSC downstream data frame to its outputs. Therefore the signal from PWM33 is mapped to pin OUT7 of the Low Side Switch. The corresponding code can be found in appendix A.1.

6.2.1. Measurement

The initialization of the Low Side Switch TLE6244 via SPI is illustrated in figure 6.5. The signals of the serial peripheral interface was measured and interpreted (illustrated in cyan). The reception of write instructions sent on MTSR to the Low Side Switch are acknowledged with `0x1500` on the MRST line.

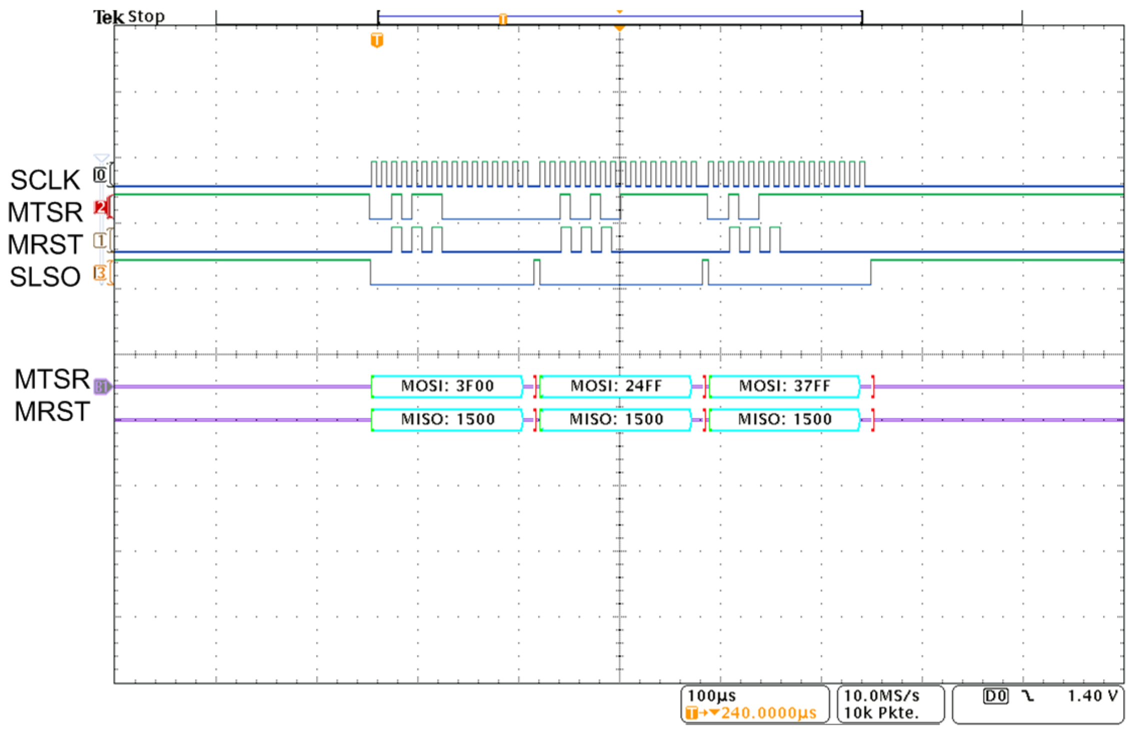


Figure 6.5.: Initialization of TLE6244 via SPI

The period of PWM33 was set to $5ms$ $(200Hz)$ with a duty cycle of 50%. This period is common in most applications. Figure 6.6 shows the signals of the MSC and output signal at the corresponding pin of TLE6244.

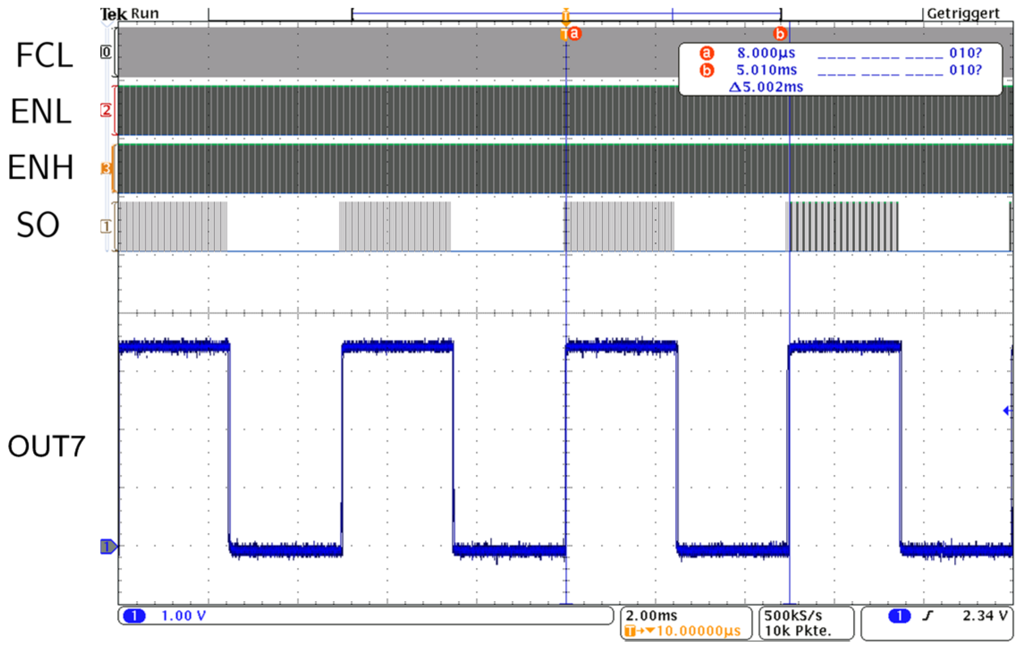


Figure 6.6.: MSC signals and output of TLE6244

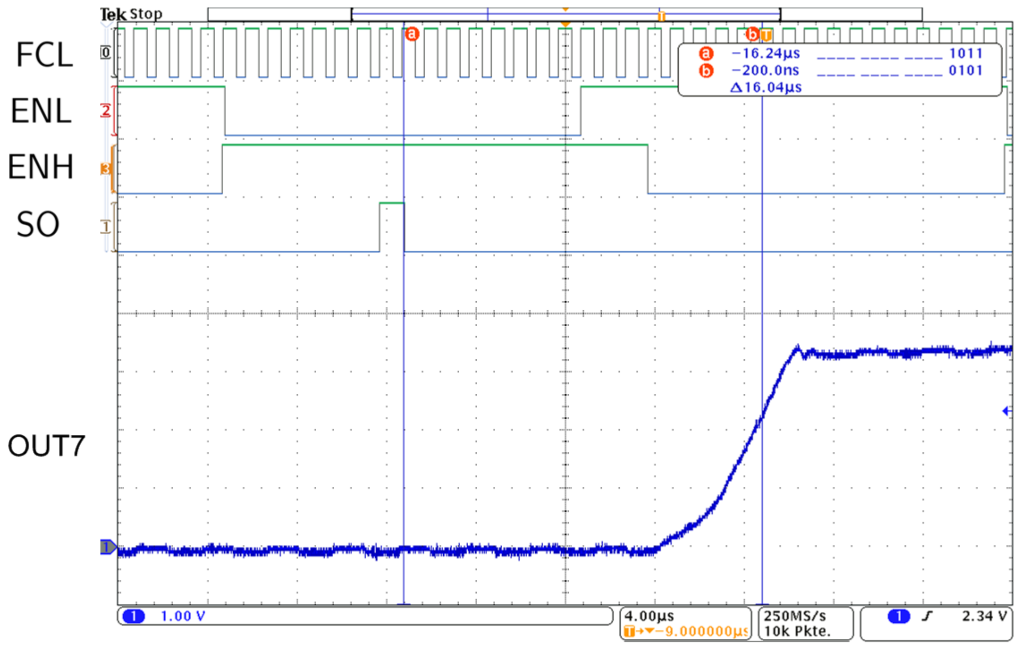


Figure 6.7.: MSC signals and output of TLE6244 at transition from low to high-state

The output PWM signal shows the configured period and duty cycle. However, the timing of the transitions is not constant, resulting in a small deviation of period and duty cycle. A measurement of the standard deviation of the period and duty cycle was done. The result values were $12.99\mu s$ for the period and 0.34% ($16.8\mu s$) for the duty cycle. This is not relevant in most applications.
The delay time of the reception of a bit that corresponds to an output signal on the MSC and the actual transition from low to high-state was measured with a constant value of $\sim 16\mu s$ as illustrated in figure 6.7. The delay time of the transition from high to low-state could also be measured with a constant value of $\sim 11.8\mu s$.

A second measurement with a period of $1ms$ with a duty cycle of 50% was done. Figure 6.8 depicts the output signal at the corresponding pin of TLE6244.

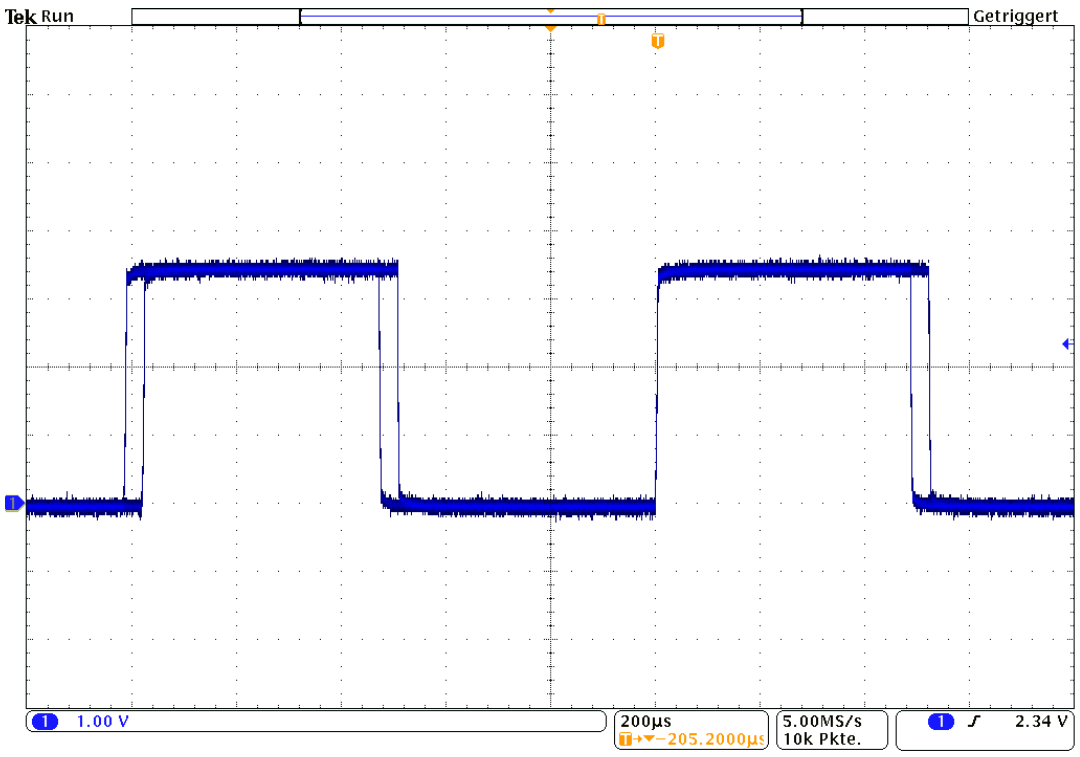


Figure 6.8.: Output of Low Side Switch (period is set to $1ms$ and duty cycle to 50%)

The output signals period and duty cycle vary. The jitter of the output signal is caused by the frame length of the MSC. In 5.6.2 (*Configuration of MSC in AVL EMS*) the duration of $t_{FRAME} = 35\mu s$ was calculated. Therefore, one specific output of TLE6244 is updated every $35\mu s$ and transitions of the output signals can only take place at multiples of this time. Depending on the output signal level of TOM1 channel 7 at the time the corresponding bit in SRH of MSC downstream channel is shifted out, the PWM signals high phase is

$t_{HIGHmin} = 14\ t_{FRAME} = 490\mu s$ or $t_{HIGHmax} = 15\ t_{FRAME} = 525\mu s$.
The same holds for the period of the signal: $t_{Pmin} = 27\ t_{FRAME} = 980\mu s$ and $t_{Pmax} = 28\ t_{FRAME} = 1015\mu s$.
To analyze the jitter of the output, the output signal and the signals of the MSC (ENL, ENH, SO and FCLK) was measured with a period of $200\mu s$ and a duty cycle of 20% $(40\mu s)$.
Figure 6.9 depicts the MSC signals, the output of TOM1 channel 7 (red) and the resulting signal at output pin OUT7 of TLE6244 (green).

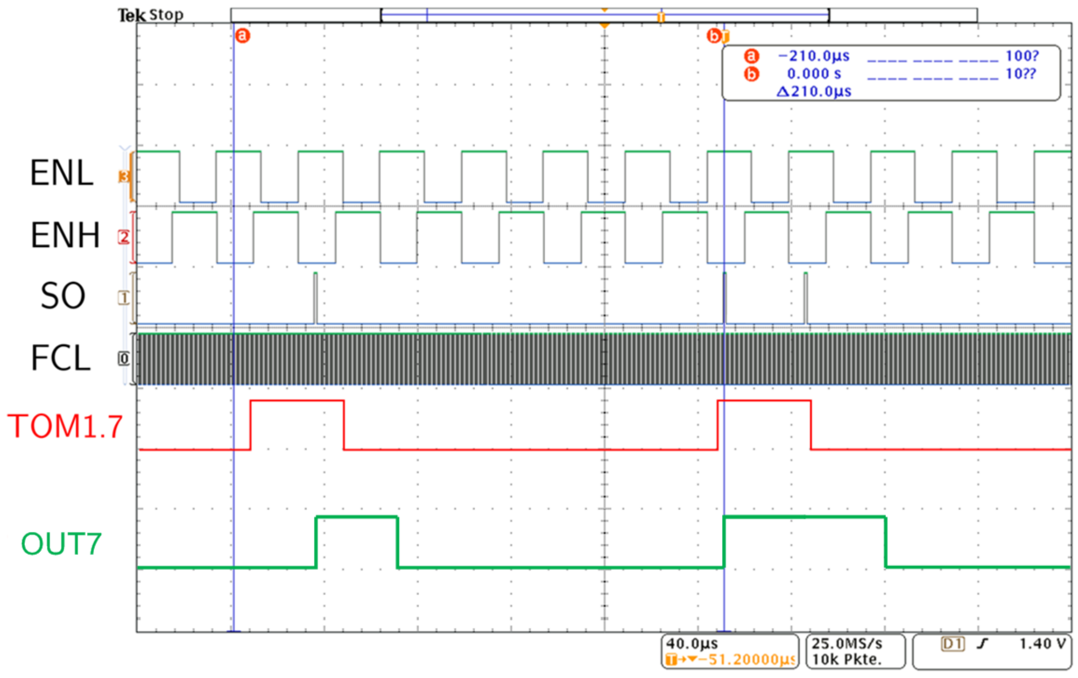


Figure 6.9.: MSC signals, output of TOM1 channel 7 and output of TLE6244

It can be seen that the output of TOM1 channel7 is "sampled" with a period of $35\mu s$.
The first high phase of the channels output shows the correct width of $40\mu s$. However it is sampled once, resulting in a high-level state of OUT7 for $35\mu s$ until the next state of the channels output is received (low).
The second high phase $(40\mu s)$ is sampled twice. Therefore the output OUT7 shows a high phase of $70\mu s$. This means that in this case, the high phase of consecutive periods always alternate between $35\mu s$ and $70\mu s$.
The resulting jitter output signal is illustrated in figure 6.10.

In the case of high frequencies and duty cycles, which lead to a high phase shorter than the frame duration t_{FRAME}, the resulting output signal can show 0 duty cycle for some periods,

since the TOM channels output is not "sampled" at high state.
However, usually the periods of the PWM signals transmitted via the MSC bus of the EMS, are lower than $1ms$ ($1kHz$). This leads to a minimal duty cycle and a maximal jitter of 3.5%, this is not relevant in most applications.

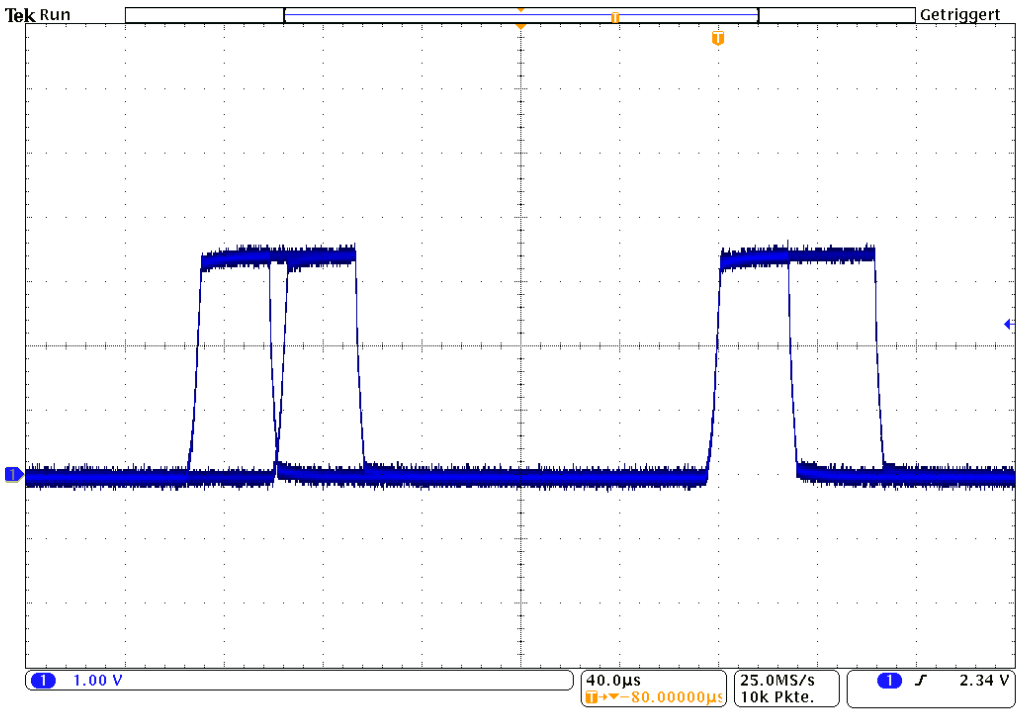


Figure 6.10.: Output of Low Side Switch (period is set to $200\mu s$ and duty cycle to 20%)

7. Conclusion and Outlook

In the course of this work, the basis for the usage of a multicore microcontroller in AVL EMS (Engine Management System) was developed.
This basis consists of a Hardware Abstraction Layer (HAL), meeting the requirements of rapid prototyping software development. It offers the possibility to port the current ASW (Application SW) to the next generation of the EMS, powered by a state-of-the-art multicore microcontroller, to cover the increasing complexity of tasks a power train management system has to handle.

The tools for the development of rapid prototyping software, including an automotive real-time operating system, were discussed and the concept of the software was outlined. Based on the smart and modular design of the rapid prototyping software, a concept for the integration of the HAL, abstracting hardware access to an ASW developer was introduced. It is aligned to the ASW development and provides flexible access to the HAL.
The developed HAL comprises device drivers for peripherals and interfaces of the multicore microcontroller as well as drivers for intelligent on board devices. Each of the software modules provides a consistent and abstracted access to peripherals. The functionalities of the resulting software modules could be verified and analyzed by different tests.

During development of the device drivers it was necessary to decide which port pins of the microcontroller are used for peripheral devices. With this definitions the work can be used as guidance for the future design of a multicore processor board. For further integration of the HAL into the rapid prototyping software a multicore processor board is mandatory and will be the next step in the development of the next generation of AVL EMS.

A. Code

A.1. MSC, PWMIO and SPI module test project

```
#include "PWM.h"
#include "MSC.h"
#include "SPI.h"

PWM_setup PWMSetup[NUMB_OF_PWMs];

SPI_init();        // Init QSPI modules
TLE6244_init();        // Init Low Side Switch
MSC_init();        // Init Micro Second Channel
PWM_init();        // Init the Clock Management Unit of GTM

for(i=0;i<NUMB_OF_PWMs;i++)    // Init period and duty cycle of PWMs
{
  PWMSetup[i].period_us = 1000;
  PWMSetup[i].dutyCyle_10 = 0;
}
PWMSetup[33].period_us = 1000;
PWMSetup[33].dutyCyle_10 = 500;
PWM_OUT_init(&PWMSetup[0]);    // Init PWM signals
```

Listing A.1: Initialization

```
PWMSetup[33].period_us = 200;
PWMSetup[33].dutyCyle_10 = 200;
PWM_OUT_updatePWM(33, &PWMSetup[33]);
```

Listing A.2: Update

Listings

Bibliography

Alan Holt, Chi-Yu Huang (2014). *Embedded Operating Systems*. Springer.

Altium (2012). *TASKING VX-toolset for TriCore User Guide*. MA160-800 (v4.0). Altium Limited (cit. on pp. 11, 23).

Andrew S. Tanenbaum, Herbert Bos (2015). *Modern Operating Systems*. Pearson Education, Inc.

AUTOSAR (2016). *Layered Software Architecture*. 4.3.0. AUTOSAR (cit. on pp. 25, 26).

Bosch (1991). *CAN Specification*. Version 2.0. Robert Bosch GmbH. D-70442 Stuttgart (cit. on p. 60).

Bosch (2013). *GTM-IP Specification*. 1.5.5.1. Robert Bosch GmbH (cit. on p. 40).

Corrigan, Steve (2008). *Introduction to the Controller Area Network (CAN)*. Tech. rep. SLOA101A. Dallas, Texas: Texas Instruments (cit. on p. 60).

Eichberger, B. and E.; Unger (2012). "Design of a Versatile Rapid Prototyping Engine Management System". In: *Proceedings of the FISITA 2012 World Automotive Congress* (cit. on pp. 2, 3).

ETAS (2008). *ASCET V6.1 Getting Started*. 10565-UG-5.1.1 EN-12-2012. ETAS GmbH.

ETAS (2011). *ASCET-SE V6.1 User's Guide*. EC014201 R6.1.3 EN. ETAS GmbH (cit. on pp. 10, 18, 20).

ETAS (2012). *RTA-OS User Guide*. 10565-UG-5.1.1. ETAS GmbH. Stuttgart (cit. on pp. 13, 15, 17).

Gijesel, Ino de (2010). *CAN und EOBD in der Fahrzeugtechnik*. Ed. by Kurt Diedrich. Aachen: Elektor.

Hemlin; Dan Larsson; Jonas (2015). "Exploring the Generic Timer Module's Feasibility for Truck Powertrain Control". MA thesis. Chalmers University of Technology (cit. on p. 35).

Infineon (2003). *TLE6244 - 18 Channel Smart Lowside Switch*. 2003-08-29. Infineon Technologies AG. D-81541 München, Germany (cit. on p. 56).

Infineon (2005). *AP32013 Connecting TLE6244X to TC1796 via usBus*. V 1.0. Infineon Technologies AG. 81726 Munich, Germany81726 Munich, Germany.

Infineon (2008a). *AP16115 Programming the on-chip Flash using the SSC Bootstrap Loader*. V1.01. Infineon Technologies AG. 81726 Munich, Germany81726 Munich, Germany.

Infineon (2008b). *TLE6250 High Speed CAN-Transceiver*. Rev. 4.0. Infineon Technologies AG. 81726 Munich, Germany (cit. on p. 78).

Infineon (2012). *TC1798 32-Bit Single-Chip Microcontroller Users Manual*. V1.2. Infineon Technologies AG. 81726 Munich, Germany.

Infineon (2013). *TriBoard TC2X7 Hardware Manual*. V 1.3. Infineon Technologies AG. 81726 Munich, Germany (cit. on pp. 5, 6).

Infineon (2014a). *AURIX^{TM} TC27x C-Step*. Edition 2014-12. Infineon Technologies AG. 81726 Munich, Germany (cit. on pp. 6, 31, 34, 35, 37, 40, 42–45, 47–52, 56, 57, 60, 63, 68, 69).

Infineon (2014b). *TC275 / TC277 Data Sheet*. V 1.0. Infineon Technologies AG. 81726 Munich, Germany.

Infineon (2015a). *TC277 / TC275 / T270 Addendum*. v1.3. Infineon Technologies AG. 81726 Munich, Germany.

Infineon (2015b). *User Manual - Software Framework Tools*. V3.2. Infineon Technologies AG. 81726 Munich, Germany (cit. on p. 23).

ISO/IEC (1994). *Information Technology - Open SSystem Interconnection - Basic Reference Model: The Basic Model*. Genève, Switzerland: ISO/IEC (cit. on p. 60).

Lauterbach (2010). *JTAG Debugger - Technical Information*. Lauterbach GmbH. D-85649 Hofolding (cit. on p. 8).

Mandl, P. (2014). *Grundkurs Betriebssysteme: Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung*. SpringerLink : Bücher. Springer Fachmedien Wiesbaden. URL: `https://books.google.at/books?id=T5SLBAAAQBAJ` (cit. on p. 17).

Noergaard, T. (2012). *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Embedded technology series. Elsevier Science. URL: `https://books.google.at/books?id=96jSXetmlzYC` (cit. on pp. 11, 12).

NXP (2016). *Programmable solenoid controller PT2000*. Rev. 7.0. MC33PT2000. NXP Semiconductors.

Qing Li, Caroline Yao (2003). *Real-Time Concepts for Embedded Systems*. CMP Books.

Robert Oshana, Mark Kraeling (2013). *Software Engineering for Embedded Systems*. Elsevier.

RPEMS - Rapid Prototyping Engine Management System (2017). Accessed: 2017-01-30. URL: `www.avl.com` (cit. on p. 1).

Schaffer, Horst (2008). "Entwicklung eines HAL für den 32-bit Mikrocontroller TC1775 der Fa. Infineon". MA thesis. Technische Universität Graz.

Vector (2015). *Handbuch - CANcaseXL, CANcaseXL log*. Version 5.2. Vector Informatik GmbH. Ingersheimer Straße 24, D-70499 Stuttgart.

Wikipedia (2015). *Microsecond Bus — Wikipedia, The Free Encyclopedia*. [Online; accessed 25-March-2017]. URL: `https://en.wikipedia.org/w/index.php?title=Microsecond_Bus&oldid=649198410` (cit. on p. 56).