DIPLOMA THESIS

Comparison of Verification and Productive Firmware for a Wireless Low-power System-on-Chip

Submitted at the
Faculty of Electrical Engineering and Information Technology,
TU Wien
in partial fulfilment of the requirements for the degree of
Diplom-Ingenieur

under supervision of

Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Institute number: 384
Institute of Computer Technology

and

Dipl.-Ing. Michael Rathmair
Institute number: 384
Institute of Computer Technology

by

Christian Tauber, BSc
0928877
Fasangartengasse 64/9
1130 Wien

23 May 2017

_____

Kurzfassung

Steigende System-on-Chip (SoC) Komplexität und damit verbundener Aufwand für funktionale Verifikation begründen die Nachfrage für innovative Ideen um den Ressourceneinsatz und Time-to-Market zu verringern. An einem aktuellen Pre-Silicon Entwicklungsprojekt eines ARM-basierten drahtlosen Sensor-SoC wurde die Kombination eines Hardware Abstraction Software Designs für System-level Verifikation und einer produktiven Programmierbibliothek untersucht. Die vorgeschlagene Ansatz ist ein drei schichtiger Hardware Abstraction Layer (HAL), unter Einhaltung des ARM Cortex Microcontroller Software Interface Standard (CMSIS), wobei die unteren zwei Schichten gemeinsam genutzt, und die oberste die individuelle Implementierung des Anwendungsfalls beinhaltet. Um das vorgestellte Konzept zu untersuchen wurden exemplarisch die Implementierungen zweier Hardware Module mit einem vorgeschlagenen Set an passenden Metriken für Bare-Metal Software untersucht. Basierend auf batteriebetriebenem Einsatz des Chips und den damit verbundenen Anforderungen an geringen Stromverbrauch, wurde der Teil des HALs für die Power Management Unit untersucht. Als zweites Modul wurde die Schnittstelle zum dem auf dem Chip integriertem Transceiver gewählt, aufgrund der Schlüsselfunktion drahtloser Kommunikation. Die umfassende Auswertung zeigte, dass ungefähr 80% des HAL Quellcodes von der Pre-Silicon Verifikation ohne Veränderung für die darauffolgende produktive Programmierbibliothek weiterverwendet werden können. Jedoch müssen die genauen Schnittstellendefinitionen im Vorfeld festgelegt werden und bedürfen der Achtsamkeit des Trade-Offs hinsichtlich Performance und Speicherbelegung. Abschließend an die Anwendbarkeit der kombinierten Entwicklung, wird ein detaillierter Ausblick auf weitere zukünftige Forschungsfragen zu diesem Thema gegeben.

Abstract

Rising system-on-chip complexity and therewith functional verification effort drives the demand for innovative ideas to decrease required resource usage and shorten time-to-market. On a current pre-silicon development project of an ARM-based wireless sensor system-on-chip, the combination of hardware abstraction software design for system-level verification and a productive programming library was evaluated. The suggested approach is a three layer Hardware Abstraction Layer (HAL), in accordance to ARM Cortex Microcontroller Software Interface Standard (CMSIS), where the bottom two layers are used in common, and the upper one is providing the particular use-case implementation. To survey the proposed concept, the implementations on two example hardware modules were evaluated with a proposed set of appropriate metrics for bare-metal software. Based on the battery-driven chip operation and therewith important low power requirements, the part of HAL regarding the custom developed power management unit was evaluated. As second module, the interface to the on-chip transceiver was selected, by the chip key functionality of wireless communication. The comprehensive evaluation showed that reuse of about 80% of HAL source code from pre-silicon verification could be achieved without any modification necessary, for the followed productive programming library development. However, the straight layer interface definitions have to be defined a-priori and need awareness for trade-offs regarding performance and memory utilization. Concluded with the applicability of combined development, a detailed outlook on reasonable extended future research on this topic is given.

Acknowledgements

After an intensive but interesting period of 8 months of working on this diploma thesis and the achieved personal growth & development, I would like to thank everyone who has supported me.

First I have to thank my colleagues from Infineon Technologies AG for the enjoyable, instructive collaboration. Christian Hambeck and Johannes Schweighofer continuously and dedicatedly helped me during the challenging tasks of the development project. At TU Wien, I would like to say thank you to my supervisor Michael Rathmair and Prof. Jantsch for their guidance to finish my diploma thesis.

I would especially like to thank my parents and my sister supporting me throughout my studies and give me the confidence to get where I am now. Finally, thanks to all my friends that accompanied me during the enjoyable time at university and the great time in Vienna.

Table of contents

Abbreviations

AES	Advanced encryption standard
AHB	Advanced high-performance bus
AMBA	Advanced microcontroller bus architecture
ANSI	American national standards institute
APB	Advanced peripheral bus
API	Application programming interface
CAN	Controller area network
CLK	Clock
CMOS	Complementary metal-oxide-semiconductor
CMSIS	Cortex microcontroller software interface standard
CPU	Central processing unit
DAP	Debug access port
DMA	Direct memory access
DSP	Digital signal processing
FCLK	Free-running processor clock
FPGA	Field programmable gate array
GND	Ground
GPIO	General-purpose input / output
GUI	Graphical user-interface
HAL	Hardware abstraction layer
HCLK	High-speed clock
I2C	Inter-Integrated circuit
IoT	Internet of things
IP	Intellectual property
ISO	International organization for standardization
ISR	Interrupt service routine
JPEG	JPEG file interchange format, ISO/IEC 10918-1
JTAG	Joint test action group
LED	Light-emitting diode
LOC	Lines of code
McCabeCC	MyCabe cyclomatic complexity
MCI	Memory card interface
MCU	Microcontroller unit
MISRA	Motor industry software reliability association
MOSFET	Metal-oxide-semiconductor field-effect transistor
NAND	Not-and

NMOS	N-type metal-oxide-semiconductor
NVIC	Nested vector interrupt controller
OOP	Object-oriented programing
PCLK	Peripheral clock
PDT	Processor-driven verification
PLL	Phase-locked loop
PMOS	P-type metal-oxide-semiconductor
PMU	Power management unit
RAM	Random-access memory
RFID	Radio-frequency identification
ROM	Read-only memory
RTL	Register-transfer level
SAI	Serial audio interface
SCLK	System clock
SEL	Select
SFR	Special function register
SLOC	Source lines of code
SoC	System-on-chip
SPI	Serial peripheral interface
TLOC	Total lines of code
TRX-IF	Transceiver interface
U(S)ART	Universal (synchronous/) asynchronous receiver/transmitter
UHF	Ultra-high frequency
UPF	Unified power format
USB	Universal serial bus
V_{dd}	Positive power supply
WB	Wishbone bus
WIC	Wakeup interrupt controller
XML	Extensible markup language

1. Introduction

Next generation wireless sensor networks put the requirements on secure and flexible configuration together with high-performance while satisfy the demand on low power consumption for longest possible application lifetime. A system-on-chip (SoC) development project by Infineon on a sub-GHz transceiver chip with integrated ARM Cortex-M0 processor and a flash memory combined with efficient power management strategies shall meet these requirements. Figure 1.1 shows this wireless sensor SoC, by the extension of microcontroller architecture with an application specific sensor and a transceiver to communicate with other devices. The de-facto standard of ARM architecture benefits compatibility, code reuse and integration of well-tested IPs. With the focus on performance and flexibility, the request for intelligent Internet of Things (IoT) network nodes can be satisfied. The development of an efficient power management, the incorporation of versatile wireless interfaces and the goal of minimal chip size integration are thus major challenges in the current pre-silicon development stage.

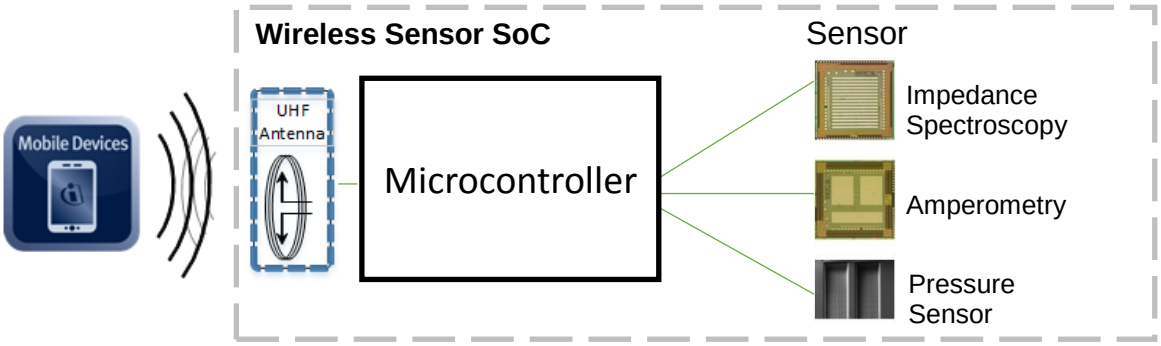

Figure 1.1: Wireless Sensor System-on-Chip

Target firmware applications will be able to access the functionality of the modular hardware structure by simple, slight interfaces, provided by a Hardware Abstraction Layer (HAL). Within this context, this thesis presents a concept for combining verification and productive purpose in the HAL design, motivated by potential reduction of the development efforts, a challenging field in research according to the raising system-on-chip complexity and consequently verification efforts. With the research on state-of-the-art software metrics and the treatise on the applicability on embedded bare-metal software an evaluation on two example hardware module implementations will allow an in-depth conclusion about the proposed HAL and the code reuse over different software design goals.

1.1 Motivation

In a current SoC development project by Infineon the upwards trend in project effort was identified to be surveyed in the pre-silicon stage. A potential strategy for reduction in firmware development and hardware verification is treated by this diploma thesis by examining overlapping tasks in software design, and providing a contribution to a more efficient development process. In combination with the contemporary requirements of IoT wireless nodes, driving node intelligence and longest lifetime, this survey subject is positioned on current topics of research and development.

The design flow of a SoC starts from the specification, goes over to the co-development of hardware and software, where the latter is quite important to verify the functional correctness of the chip, before it is sent to the fab for a first prototype production. This so-called "tape-out" and the associated mask design for the production process is a quite expensive task in SoC development, justifying an excessive effort in ensuring functional correctness. Post hoc design changes result in single or up to total redesign of all masks and maybe in a project fail by financial unfeasibility. An industrial study from 2014 [Fos15] claimed that in SoC development projects 57% of the total project time is spent on verification, with an ongoing trend upwards. In addition, 61% of these projects are behind schedule and thus, are not able to finish the project within the proposed time plan. Approaches and contributions in efficient verification can therefore decrease the development effort and as a consequence time and money resources.

1.2 Problem Statement

The overall rising complexity in systems design and the associated verification effort forces the creation of new innovative strategies and approaches in functional verification. On state-of-the-art SoC development projects this task consists of assertion-based verification in hardware description languages (HDL including VHDL, Verilog, etc.) and processor-driven verification, where a C or Assembler test facilitates the processor and bus system to check the functionality of a module-under-test on system level. The development of firmware therefore is done for verification purpose and in addition to realise a productive firmware, generally in form of a HAL. This software is then provided for 3rd party application engineering when using this SoC chip for custom application development. Obviously, in pre-silicon development at the semiconductor manufacturer there is an overlapping between verification and productive firmware development that could contain potential for improvements in development effort reduction, by identifying redundant programming tasks and the resulting parts of code.

Since power management is a major topic to enable longest lifetime with the hard limited amount of energy from battery or energy harvesting, and the connection of the power management unit (PMU) to almost all other components in the system to drive for example sleep or power down states, the verification task is a formidable challenge.

1.3 Task Setting

The proposed SoC development project is divided into several development domains, where the delimitation to the thesis survey topic is quite important. The hardware development state includes digital and analog design. While the digital components are controlled and in direct interaction with the central processing unit (CPU) subsystem and the executed software instructions, the analog domain, including clock oscillators, power supply, or input/output pads, is connected to software via digital interfaces. The digital hardware design is closely linked to the domain of software development, with the application of verification firmware to ensure the functional correctness on system level besides HDL module assertion tests. Therefore it is mandatory that the digital domain is explained in detail to understand the chip architecture. However, the digital design itself and detailed questions about it are not part of this thesis, but important to describe the actual task setting.
Already mentioned, to present the software task intersection of verification and productive development purpose, the combination through a particular HAL design was examined. To make both variants comparable, suitable metrics qualify the software attributes.
With the important role of the power management unit as all-connecting to the other modules and the risk to bring the system into critical states, the verification of this component is devoted a major focus in this thesis. Through evaluation of the two firmware variants with appropriate, state-of-the-art metrics for embedded software, allowed a comprehensive conclusion about the proposed analysis, with a proof-of-concept presented on the example of the power management module and the transceiver interface.

Shortly summarized, the tasks of this thesis were:

- State-of-the-art survey on combined verification and productive firmware, plus as key subject on power management tests
- Survey on software metrics and examination of applicability in the context of bare-metal software and HAL benchmarking
- Design and implementation of the combined HAL in C
- Functional verification of the hardware components with the developed HAL and deployment of the productive variant
- Evaluation of the proposed concept on the example of two hardware modules with the selected set of appropriate metrics
- Discussion of results and contribution on development and verification effort

1.4 Methodology

The de-facto standard of ARM in microcontroller design includes the hardware environment and the Cortex Microcontroller Software Interface Standard (CMSIS), a HAL design guideline with provided core-drivers and templates which was used as basis for the proposed software design.
The verification of hardware modules can be done on different abstraction levels and various methods. Single digital components often tested by assertion-based HDL tests which can get complex on bigger design. Hardware Verification Languages (HVLs) therefore are designed to support the verification task. However, on microcontroller system level the interaction of a component with the other blocks, especially the CPU, via a common bus interface is a proper use-case. As a consequence functional verification is in addition done by C-test, facilitating the microcontroller (MCU) system. In the current development project, a combination of HDL-based tests and system-level tests was applied, where the latter one is further discussed in this thesis. To realise the surveyed comparison of the verification firmware variant to the productive application library, software metrics were used to measure the quality of several meaningful attributes. A broad selection from the review on state-of-the-art metrics was sorted out by the discussion about applicability for each metric and resulted in a reasonable set for hardware abstraction layer measurements. These metrics evaluate characteristics (i) of the source code itself only, to examine internal attributes and (ii) software within the environment, in particular by executing the software on the target MCU system. In this context, the execution was in the current pre-silicon development stage done in register-transfer level (RTL) simulation, where the software binaries are loaded into memory behaviourals. As a consequence unrestricted white-box testing of software and hardware was possible in the RTL simulation environment. For the source code and binaries analysis, several tools were applied to calculate metric results.

2. State of the Art and Related Work

Verification, the answer to the question "Does it work?" was historically side-by-side in an evolutional process to the development tools and design process models. In SoC development both, hardware and software are nowadays co-developed and have to be constantly checked against. With raising system complexity far beyond single and manageable HDL digital designs towards processor-based microcontroller system architectures, assertion-based hardware verification methods were extended by system-level tests, to test convincing use cases in interaction with the system environment itself. To clarify the context of this thesis, the term of verification in general and processor-based verification are explained, with power management verification in detail. Subsequently, the popular ARM architecture is due to its market and technological dominance discussed, by explaining the modular structure and interaction of components. On the subject area of software, to fit to the developed hardware and allow abstraction for simplified access of hardware functionality, the common hardware-abstraction layered design approach benefits flexibility and code reusability. ARM provides a HAL design guideline, including core drivers and templates. To adjust this model for a proper customized design, experiences from literature research shall allow a better understanding for particular design decisions. Next, the focus to mobile low-power applications can best be discussed by looking at the technical causes for power dissipation, and common minimization techniques. To compare the developed HAL variants for verification and productive purpose a broadly based summary on state-of-the-art software metrics is presented. Finally, related work is discussed to point out the positioning of the thesis contribution.

2.1 Functional Hardware Verification

IEEE 1012-2012 Standard for System and Software Verification and Validation [1] defines verification as "*the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase*".
Validation in contrary is explained by the IEEE Standard 15288-2008 [2] as "*confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled*". To put it simply, verification answers the question "Does it work" and validation "Do we build the right thing?"

Hardware Verification in particular refers to the tools and techniques, used to ensure that a system does not have critical hardware bugs and performs correctly in response to outside stimuli like executing software. Setting up a system, run it and check if it fails is called testing, and is differentiated to verification by being less comprehensive [And05]. While in in the early stages independent hardware and software development were quite conventional, rising complexity and therewith higher development cost substantiated new approaches. Hardware/Software Co-development parallels the

sequential development process by starting software implementation while still evolving and verifying the corresponding hardware components. Hardware/Software Co-Verification is the process of verifying embedded system software runs correctly on the hardware design before the design is committed for fabrication [And05]. This is resulting in shorter time-to-market and an additional stimulus in addition to HDL test benches for the hardware developers.
A few years ago the term *Processor-driven Verification/Test* (PDT) also known as *Software-driven verification* [Bai14] [Goe14] was introduced and addresses the problem that test-benches for single IP modules do not cover the interaction in systems with an embedded processor. In PDT the tests are stored in memory, and the results evaluated after execution [Lus16]. A major advantage is the reusability of tests during the overall project lifecycle, since the C or assembly test programs could run on all stages of development from RTL- and Gate-level-simulations to FPGA implementation or up to the fabricated device.

Kenney proposed a detailed PDT methodology [Ken06] in 2004, with these advantages but described PDT as just simple block tests written in C and not firmware per se. A similar approach was shown in [Kom06] connecting several IPs around an *Advanced Microcontroller Bus Architecture bus* (AMBA), and demonstrating that functional verification could be mainly achieved by software tests in C. In [Hun03] the authors proposed a three-layer-architecture (HAL - API – Operating System) to use high level software test programs and showed the success in hardware abstraction, but conclude a big overhead in debugging and simulation as a major disadvantage. A "Hardware Abstraction Layer Generator" as part of STMicroelectronics "Spirit Assisted Verification Environment (SAVE)" was presented in [Lin10] to address the problem that test case coding is a critical phase during functional verification and the challenge is to write bugless lowest-level C or assembly test cases. In contrast to [Hun03] the architecture was reduced to two layers, to decrease the overhead. A major idea was to automate the process of HAL generation to write and read the registers, and make C test case coding much easier. A time reduction in the verification process of 33% is proposed, and obviously an increase in test execution time. However, there is no information available on the ongoing status of this project. Moreover, the idea of HAL generation could be easily found on the web from several digital design global players, but without specific conjunction with the topic of functional verification.

Block-level verification is nowadays a widely solved problem, whereas the challenges at SoC level are very different [Goe14]. The difficult task is the integration of various blocks and to make sure that they all work properly together. Consequently PDT is a potential field of research by examining methodology to balance high coverage and verification quality with delimitation of development costs and time-to-market.

Power Management Verification

Wireless sensors have the requirement of low-energy to ration the limited amount of energy by a battery or harvesting system to accomplish the goal of longest lifetime. Formats like the Unified Power Format (UPF) [12] enable the modelling of power supply and control intent and thus extend pre-silicon simulation and described in [Mba12] and [Kar13].
The central component is a power management unit that controls with a mix of power consumption reduction techniques the other system modules, which are grouped together to power domains. A

system-on-chip consists of several domains, for example the CPU-subsystem, the peripherals and the PMU itself. Whenever the CPU-subsystem and peripherals are switched to a power saving mode, the PMU domain is still supplied to wake-up the domains when for example an external event is detected on a preconfigured general-purpose input/output (GPIO) pin.
[Pin08] defines power management verification as the functional verification at pre-silicon stage and involves functional correctness testing in a simulation environment. The challenge of power management verification is based on its very global functions, involving interactions with a lot of other blocks. Thus tests should be system-like; the functional verification of the PMU component itself is claimed as only around 30% of effort.
[Win12] divides the debug and verification of a power management system into hardware and software section, while the latter is divided further into two layers: (i) firmware layer regarding hardware functionality to manage power states without application interaction, such as shut down of components after a certain idle time, (ii) application layer that contains four states with functionality of configuration, operation, sleep and hibernation. With these two layers a fertile approach for a pre-stage of abstracting power management tests from the particular hardware is presented.
One of the latest publications on this topic by [Mac16] shows a verification approach by performing (i) syntactic checks, more precisely language syntax checks regarding to power management, (ii) run-time checks that could not be revealed by a SystemC compiler, (iii) static analysis, to validate semantic conditions like the number of power domains used, (iv) equivalency checking with UPF and (v) assertion-based verification to check lower-level control sequences.
The literature on the broad topic of power management verification shows that common understanding is not yet established. Therefore, rising system-complexity and the requirement to low-power application design will make further research on this topic necessary. Innovative verification techniques in general combined with a proper understanding and focus on power management verification will be essential to meet the requirements of fast evolving trends in the wireless, low power IoT domain.

2.2 ARM Architecture

The ARM microprocessor architecture, developed by ARM Limited [3] is a reduced instruction set computer (RISC) architecture and nowadays very popular due to its industrial leadership and was first introduced in 1985. The designs are developed by ARM and can be licensed by semiconductor vendors to implement them into own custom products. ARM launched the very successful Cortex-M3 processor in 2004. While in 2007 the microcontroller market was shared among 40 vendors in 50 different architectures and a per vendor share of maximum 5% [Pow10], in 2009, based on "Gartner - Market Share Analysis - Preliminary Total Semiconductor Revenue, Worldwide 2008", ARM already had included 6 of the top 10 worldwide semiconductor companies as licensees [Spe09]. In early 2015 ARM already reached a total market share of 24% in microcontrollers and smartcards, and 70% in 32 bit MCUs [York15].
Figure 2.1 shows a typical ARM-based SoC. The ARM architecture not just includes the processor core but also a great number of commonly integrated SoC components and appropriate interconnection buses. The ARM processor core is connected to the 32+ bit wide *Advanced high-performance*

bus (AHB), to access memory modules such as *read-only memory* (ROM), *random-access memory* (RAM), Flash, additionally a graphics processing unit or other components that need high data rate connections. When a *direct memory access* (DMA) controller is used, the bus is commonly implemented as matrix, to available parallel access by paying this with higher gate count due to necessary scheduling functionality. A DMA controller can then process data transfers between memory and/or peripheral units, to relieve the core processing unit. Slower peripherals are connected via the *Advanced Peripheral Bus* (APB), such as a *universal asynchronous receiver/transmitter* (UART), timers or the *Serial Peripheral Interface bus* (SPI), and accessible via a transparent-operating bridge between the two bus segments. This segmentation over different bus systems is not visible to the CPU, all component access-operations are done in the same way, realized by a particular address range of the AHB mapped to the APB bus domain. Both AHB and APB are part of the *Advanced Microcontroller Bus Architecture* (AMBA). With the success of the ARM architecture, AMBA became a de-facto standard for on-chip communication among functional blocks.
A debug controller in addition provides access to the microcontroller via JTAG or Serial Wire interfaces.

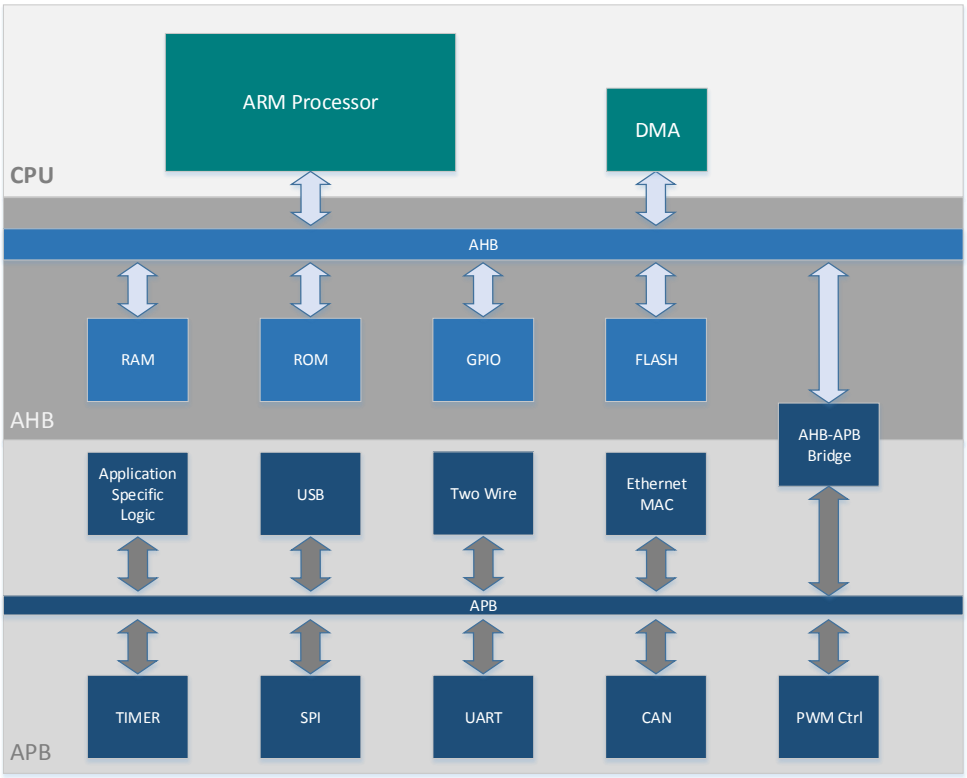

Figure 2.1: ARM System-on-Chip block diagram [3]

The ARM processor family provides processor systems for all kind of application requirements, from highest performance to serve rich operating systems (Cortex-A Series), over fast response requirements for hard real-time applications (Cortex-R), to smallest/lowest power optimized microcontrollers (Cortex-M) or resistant security applications (SecurCore). In every domain again multiple processor cores are available; distinguish from another again by performance, efficiency, power

and area consumption. In the category of smallest cores, the Cortex-M family and particularly the aforementioned first in 2004 released Cortex-M3 raised a significant market position in the embedded ecosystem. Five years later, in 2009, the by now smallest family member Cortex-M0 was introduced, aiming lowest gate count and power consumption. In 40nm technology process, a total floorplan area of 0.007 mm2 is achieved but by providing a full 32-bit processor core [4].

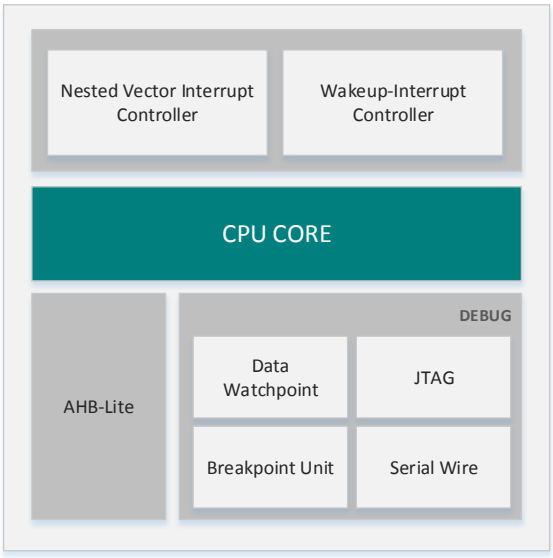

Figure 2.2: Main parts of a Cortex-M0 processor [4]

Figure 2.2 shows schematically the main components of the Cortex-M0 processor. The CPU is equipped with the *Nested Vectored Interrupt Controller* (NVIC) that receives events from peripherals and triggers on input an interruption of the program execution by calling and executing the assigned *interrupt service routine* (ISR). Cortex-M0 therefore supports up to 32 Interrupts. The *Wakeup Interrupt Controller* (WIC) handles this functionality in combination with a *Power Management Unit* (PMU) when the core is in power saving deep sleep mode, to buffer the requests while waking-up the core and then forwarding the interrupts. The WIC is a lightweight implementation with only combinatorial logic, to provide the minimal necessary functionality for wake-up, to be itself as power-efficient as possible. The AHB-Lite interface connects the processor to the other components on chip, such as the memories and the peripherals.
For efficient application development, a *Data Watchpoint* is provided to monitor variables or expressions. In addition with a Breakpoint Unit and the JTAG/Serial Wire access ports (*Debug Access Port* – DAP) debug functionality of the Cortex-M0 is served. The JTAG/Serial Wire access allows a complete access to the CPU core registers and the bus system, thus all system components. It can also be used to flash the system, besides popular implementation facilitating the simple UART or SPI interface.

2.3 Hardware Abstraction Layer

The increase of SoC design and therewith software complexity drives the demand of code reusability to hold the development time of a new application in an acceptable range. An answer is the concept of decoupling the interfaces between software and addressed hardware, by dividing the software in a hardware-dependent and hardware-independent part to challenge the requirement of portability [Jer05] [Pos03]. However, the term of hardware abstraction layer is a widespread and abstract treated topic of study in literature. By the ongoing evolving field of SoC development, and lately dominance of ARM in a former field of many share proprietary hardware architectures (see Chapter 2.2), the subject literature still has to find concrete consolidations in many aspects to avoid ambiguity.
[Sun03] describes the hardware abstraction layer as all the software that is directly dependent on the underlying hardware, including boot code, context switch code and providing configuration and access to hardware resources. Thus, parts of this software have to be changed, whenever the hardware architecture is changed. The question of one standard in hardware abstraction for SoCs is clearly negated in this publication, reasoned by the application-specific hardware architecture design. However the possibility of a generic set of specific application programming interfaces (APIs) combined with a common HAL is considered or a possible focus on a suited HAL for specific application domains.

A more detailed approach on HAL definition is presented by [Han05] in a specific tree-layer hardware abstraction architecture design for wireless sensor networks, including a *Hardware Presentation Layer* (HPL) that interacts directly with the Hardware, a *Hardware Abstraction Layer* (HAL), the core component that abstracts specific devices into domain models (like alarm or analog-digital converter) and a *Hardware Interface Layer* (HIL) that provides the hardware-independent "typical" hardware service interfaces to applications.

In a now standard reference book on this topic, "Hardware-dependent software" [Eck09] a HAL is defined as a software layer that provides an abstract interface to access hardware resources, and typically divided into access, register, and functional shielding. Furthermore the code is divided into processor specific software code, e.g. enable interrupt vectors, and device drivers to access peripherals or power management. The proposed services are ANSI C integration to provide standard-C functions such as *printf*() or *fopen*(), the device drivers, a consistent interface to this services for the application layer, as well as system and device initialization. The overall advantage of using hardware abstraction to increase reusability and flexibility is shown on a proof-of-concept of a JPEG application, ported to different processors by using hardware abstraction.
This definition of the term hardware definition, as software stack that abstracts the hardware access from the application layer, is compliant to the following interpretation in the subchapter of standards and classification for HAL, as well as to the nowadays dominant ARM architecture, which will be topic of subchapter 2.3.2.

Standards and Classifications

[Bha13] presented an approach on separation of hardware abstraction in the following categories: (i) industry-standard, (ii) vendor-definied or (iii) user-definied. Industry-standards are rare, but for ex-

ample exist in the field of test and measurement equipment as part of the *Interchangeable Virtual Instruments* (IVI) standard, driven by an industry consortium called *IVI Foundation* [5]. IEEE Standards Association has an active project (P2415) on low power design and verification standard in development that "addresses energy proportionality through tight interplay between energy-oriented hardware and energy-aware software." [6] The project authorization request was approved in 2014 and expires in 2018. The overall goal is to provide a standard for well-connected energy oriented design flow.

Vendor-defined HALs are developed by hardware vendors, for reusing the application software on different projects, for example the Infineon *Automotive Open System Architecture* (AUTOSAR) which provides a standard for all layers of software in the automotive area, including the input/output hardware abstraction [7]. Another category of vendor-defined HALs, from the perspective of software, is operating system dependent ones. Software companies create their own abstraction designs, to use the operating system on all variations of hardware architectures, such as *Windows Embedded Compact* (former Windows CE) [12].

User-defined HALs follow in principle no specific design rules, but a consideration on various design aspects might seem expedient. The intended goal is a thin software component, that just encapsulates all functionality that is hardware-dependent, and an appropriate interface to provide it to the upper layers. In [Hel10] the authors describe a HAL design in two separated layers, the lower level device specific code and above an application specific layer, which provides functions for the application interface. According to [Ben15], the focus in development should be on core feature identification, avoiding an all-encompassing HAL, and be aware of an iteration process in development.

Cortex Microcontroller Software Interface Standard (CMSIS)

To abstract the complexity of the ARM architecture and make developing applications much easier, ARM provides the C-implemented hardware abstraction library CMSIS, vendor-independent and expanded broadly in related areas such as debugger interfaces, a real-time system operating API or a DSP library. The goal is better industry collaboration as well as accelerating software development projects that make use of ARM microcontrollers [9]. It is checked against MISRA C, a set of programming guidelines for critical systems, published by the *Motor Industry Software Reliability Association* (MISRA) [10].

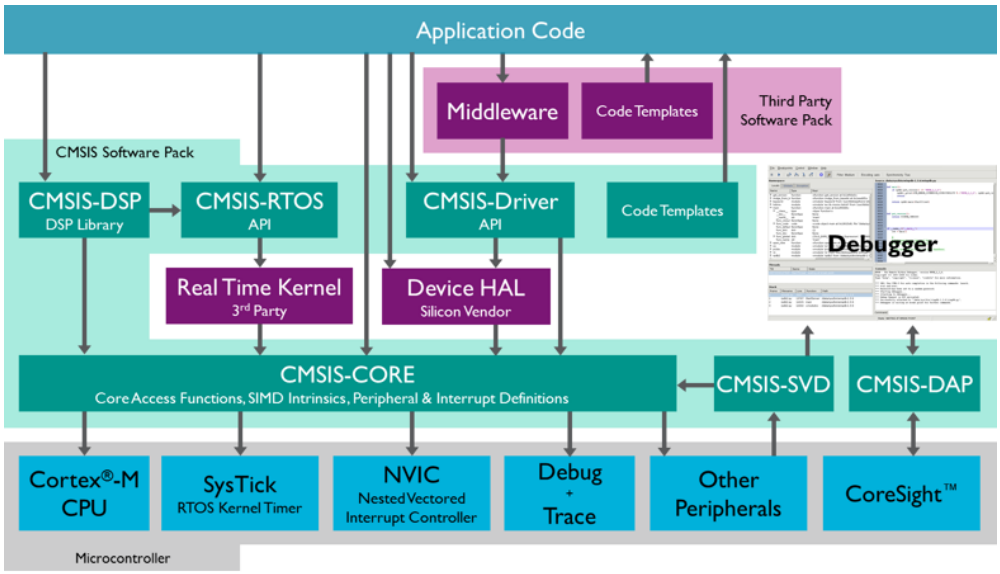

Figure 2.3: ARM CMSIS [9]

In figure 2.3 the components of the ARM CMSIS, as well as the interfaces and interconnection between them is visualized. On the bottom, the CMSIS-CORE block allows the hardware access by abstracting from writing bits to register addresses to name mappings. On the same level right-hand connected is the CMSIS-SVD (System View Description) block, which allows abstract description of peripheral devices in XML format for integration into the CMSIS Device family packs [11]. CMSIS-DAP (Debug Access Port) provides a standard communication between Debugger and ARM Cortex device via a USB to JTAG/SerialWire debug unit, which runs the DAP firmware for CoreSight Debug Access Port. Bottom-up arranged is the device HAL implemented by the silicon vendor, and optionally a (usually 3rd party) real time kernel when a real time operating system (RTOS) is in use. The CMSIS software pack in addition optionally serves a digital signal processing (DSP) library, containing basic signal processing functions, such as filters, matrix functionality or complex math functions. The CMSIS-RTOS API provides easy access to real-time features and makes middleware easier to adapt to the actual real-time kernel. The CMSIS-Driver API specifies standardized peripheral driver interfaces for applications or middleware, to promote reusability and abstract the actual peripheral implementations used, by categorizing in currently 11 different devices. Figure 2.4 simply illustrates the abstraction from physical components on the microcontroller up to middleware access and the logical abstracted units such as USB Device or Networking.

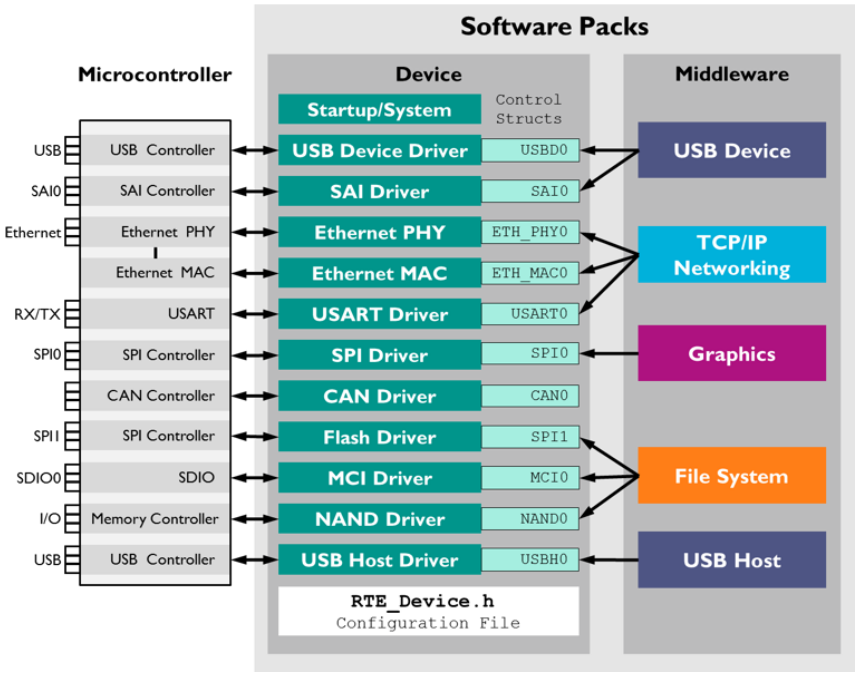

Figure 2.4: Peripheral driver Interfaces [11]

At last, beyond in Figure 2.3 complex systems may use additionally 3rd party middleware and code templates. Based on this hierarchical architecture, the application layer in the top is able to access the devices, independently from the actual physical hardware blocks implemented. In conclusion, this HAL design and features are consequently in line with the definition of hardware abstraction in the initial part of this chapter.

2.4 CMOS Power Consumption and Management Strategies

For wireless applications, power consumption is a major topic when for example a coin cell or energy harvesting determines low power requirements on the system-on-chip design to allow a longstanding lifetime. Apart from wireless applications, high integration density such as in modern multicore processors makes awareness to power consumption inevitable too. Dependent on the application, the behavior in uptime and sleep modes, as well as on technology reasons the strategy for power-saving can differ. With greater integration density, and decreasing sizes the dominant parts in dissipation changes. This chapter outlines the two fundamental parts of power dissipation in complementary metal-oxide-semiconductor (CMOS) technologies and the determining components, followed by an overview of common techniques to minimize power loss in low power design.

There are basically two parts that determine CMOS power dissipation. A static component, which does not exist in the ideal CMOS but has an impact in reality, describes the constant leakage when the CMOS component is not switched. Secondly, a dynamic dissipation appears when CMOS is switching states. The total dissipation is expressed in equation 1. Often read in literature, the dynam-

ic dissipation is further broken down in the short circuit current between V_{dd} and Ground and the loss of power in the dynamic load of the capacitance, see Figure 2.5.

$$P_{total} = P_{static} + P_{dynamic} = P_{static} + (P_{short} + P_{switch}) \quad (1)$$

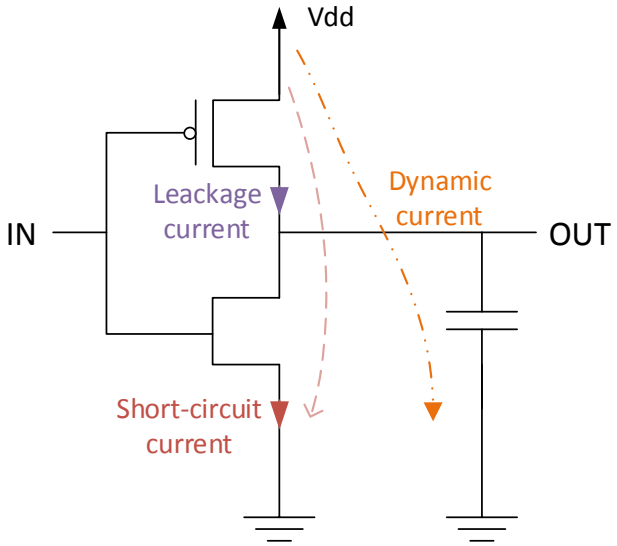

Figure 2.5: The three main components in CMOS power dissipation [Pan10]

CMOS Static Power Dissipation

In theory CMOS has the major advantage over other technologies that power dissipation does not occur when the component is not switched, due to the complementary design with a P-type MOS (PMOS) and a N-type MOS (NMOS) transistor. This ideal behavior is shattered in reality by multiple parasitic leakage effects that cause current flow and therewith power dissipation. Figure 2.6 shows schematically the dominating effects that occur between the partly overlapping, different doped junction areas on an NMOS transistor.

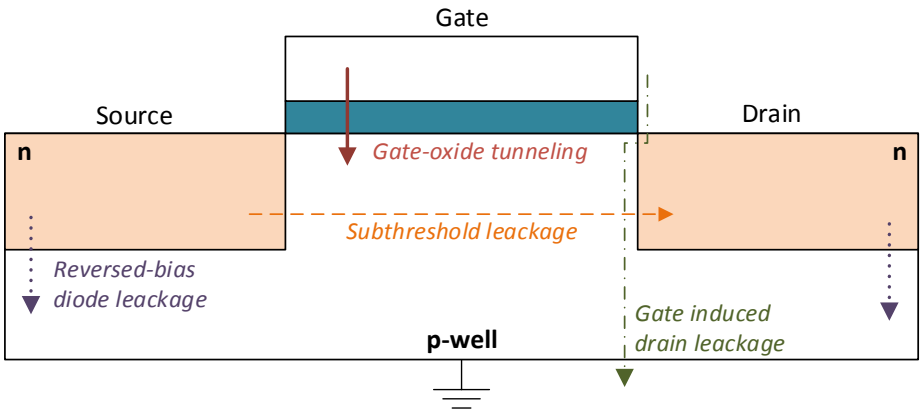

Figure 2.6: NMOS Static Leakage: (i) Gate-oxide tunnelling (ii) Subthreshold leakage between source and drain; (iii) Reverse-biased diode leakage across parasitic diodes; (iv) Gate induced drain leakage between drain and substrate [Pan10]

- Reversed Bias Diode Leakage
  The diodes formed between the diffusion regions and the substrate float a parasitic current based on the reverse bias of p-n junctions. The effect is shown graphically in Figure 2.7, represented by the offset current between 0V and the breakdown voltage. Overall the contribution to leakage is small compared to the other components, thus often neglected in power calculations.

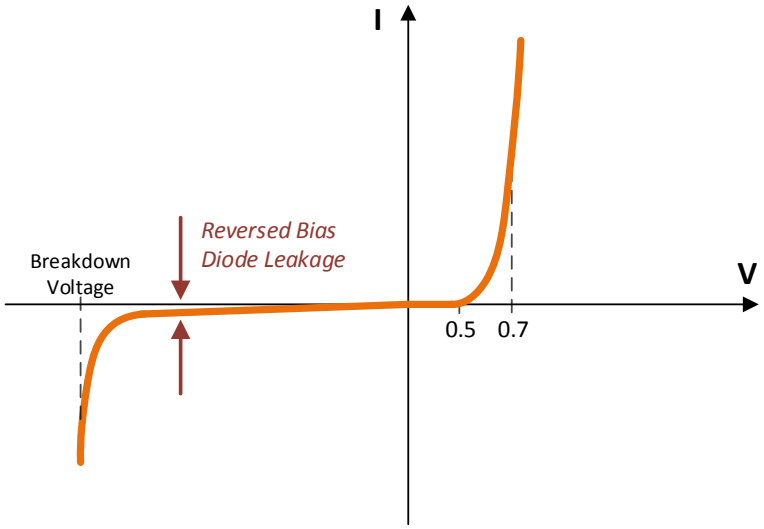

Figure 2.7: Current-voltage characteristic of a p-n junction diode. The difference from the ideal zero current in reverse mode is the reversed bias diode leakage

- Subthreshold Leakage
  The weak-inversion effect describes the flow of current when the gate voltage is below the threshold voltage and therefore the transistor operates not in saturation mode. As a result free load carriers can move between drain and source. This effect is caused by diffusion, and becoming most significant with higher gate voltage, near to the threshold voltage. In addition there are several effects that are directly influencing the subthreshold voltage. *Drain-Induced Barrier Lowering* (DIBL) is a parasitic effect that explains the dependency and decrease of the threshold voltage on high drain voltage, while *direct punch-through* describes the approaching and "touching" of the depletion regions, thus is a subsurface version of DIBL.
- Gate Oxide Tunneling
  The downsizing of MOSFET and as a consequence smaller gate-oxide layers increases the electron tunneling rate through the physical energy barrier. With further ongoing downsizing this component is effecting in a higher contribution to the total share of static power dissipation.
- Gate Induced Drain Leakage
  In the overlap region of gate and drain region a current is generated caused by band-to-band tunneling. To limit this effect, restrictions are set on the oxide thickness and power supply voltage.

CMOS Dynamic Power Dissipation

This component of CMOS power loss occurs during switching activity, and described by the following two main causes:

- Switching Power
  Energy is dissipated by loading and unloading parasitic capacitances of the MOS transistors where a major part dissipates in heat. The derivation of switching power loss is as follows. Since

$$dQ = C \cdot dU \tag{2}$$

  and the relation between shifting of a load between two points with voltage difference U and the energy is

$$dE = U \cdot dQ \tag{3}$$

  Inserting equation (2) in (3) and integrating over the supply (load) voltage leads to the energy supplied by the power source for charging:

$$E_{Load} = C \cdot V_{dd} \cdot \int_0^{V_{dd}} dV = C \cdot {V_{dd}}^2 \tag{4}$$

  The energy stored in the capacitance is the integration of the instantaneous charge during the load process, thus

$$E_{Capcity} = C \cdot \int_0^{V_{dd}} V \cdot dV = \frac{C \cdot {V_{dd}}^2}{2} \tag{5}$$

  Comparing equation (4) and (5) shows a difference of $E_{dissipation} = \frac{C \cdot {V_{dd}}^2}{2}$ that was dissipated in heat during the charge process.
  To obtain the dissipation power the energy term is multiplied by the clock frequency f. Due to the fact that not all logic gates in a digital circuit switch in each clock period, an activity factor α in the range from 0 to 1 is added to express the probability of switching the state.

$$P_{dissipation} = \frac{C \cdot {V_{dd}}^2}{2} \cdot f \cdot \alpha \tag{6}$$

- Short-circuit current
  Another dynamic dissipation effect is based on the finite rise and fall times of switching behavior in the used nMOS and pMOS transistor. As a consequence, for a short period both are ON at the same time, establishing a direct current path connecting the positive supply voltage (V_{dd}) and Ground (GND).

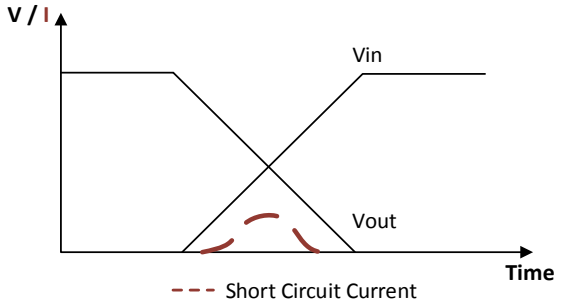

Figure 2.8: Short-circuit current in CMOS caused by the finite switching time behaviour [Pan10]

Techniques to reduce Power Consumption

After working out the causes of power dissipation in CMOS technology, the question on how to minimize the effects is coming up. Several, nowadays standard techniques evolved which are consecutively explained.

- Power Gating
  The dissipation effects described in the previous chapter showed altogether a relation to the supply by a power source. To address the simple solution of "where is no power, there is no loss", power gating explains the deactivation of CMOS cells or in bigger scope of components in an MCU system, to fully disable power consumption in inactive mode, thus effectively get rid of all dissipation effects. When the component is needed again for operation the supply is being restored. Splitting up a SoC to shut down components is regularly realized by forming power domains, thus an always-powered-on domain and one or multiple domains for temporarily shut down.
- Clock Gating
  As another approach to reduce dynamic power dissipation, clock gating addresses the problem that flip-flops are connected to the CLK and thus dissipate energy on every clock cycle. The simplest way to overcome this loss when the input data is not changed is by grouping flip-flops to functional units and transform the connection of the flip flop as seen in Figure 2.9 by AND connecting the SEL and CLK wire, thus only on activation the grouped flip-flops dissipate power. A reduction of about 5-10% is possible [Pan10].

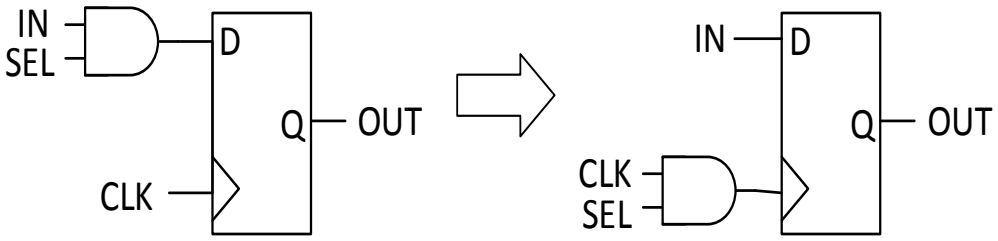

Figure 2.9: CLK passing to logic block only when selected [Pan10]

By adding more combinatorial logic to the design a more complex but higher reduction potential could be achieved. In a so called pipelined design an input change of the first stage determines the activation of the clock signal for the following flip-flops, see Figure 2.10.

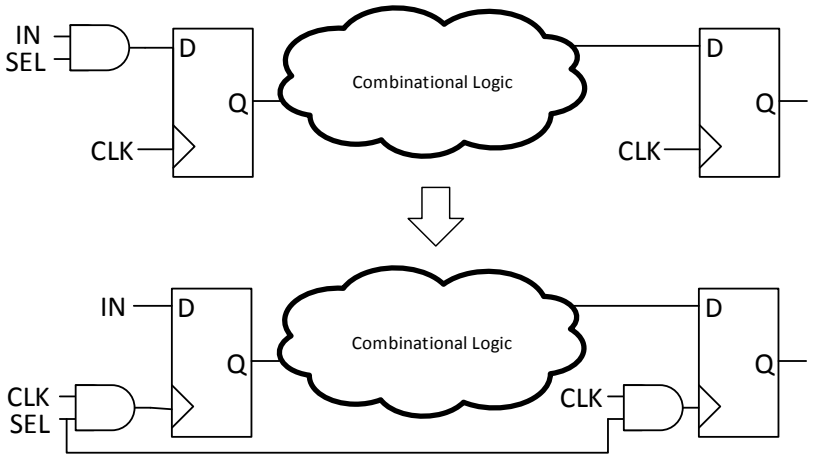

Figure 2.10: Higher effectiveness of clock gating on pipelined designs [Pan10]

- Voltage and Frequency Scaling

  The dynamic switching power is squared proportional to the power supply voltage, thus a reduction in voltage can make significant decrease of power consumption possible. Frequency is direct proportional, but also has a direct proportion to voltage, thus both can be lowered together resulting in a cubic proportional power reduction. However the circuit delay will increase and must be considered to meet the system constraints. A common approach is to settle at design time, by adjusting voltage and frequency just to meet the necessary constraints and fixate for application. A regulative approach but less sophisticated is dynamic frequency scaling, see Figure 2.11, which is quite popular in embedded systems design, allowing for setting the CLK speed during execution, and adjusting just the frequency to the current work load and needed response time of the system. For example a system that just waits for an signal edge on an input pin and the processing is not time critical, a lower frequency could still be satisfying while lowering the overall power consumption.

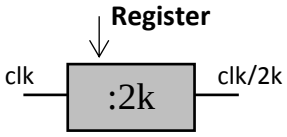

Figure 2.11: A 2k clock divider to scale down frequency, configurable via a register

Dynamic Voltage and Frequency Scaling (DVFS) expresses the dynamic scaling during runtime to achieve lowest power consumption, while concurrently analyzing and meeting the systems performance constraints, for example by adaptive control, where previous configurations and the current system measurements are determining the next frequency and voltage configuration. DVFS for microcontrollers is mostly not suitable because of PLL instabilities and where peripherals use the (constant) processor clock in operation [Yiu15]. However, multiple supply voltage domains can still meet these requirements, but then additional level-shifters are needed to convert the signals between the different voltage levels.

ARM architectural Low-power Management

The ARM Cortex-M0 is designed for ultra-low power consumption, to be suitable for small and wireless applications. Besides the low gate count, ARM provides two built-in power modes to disable parts of the system, but leave the particular implementation details in the design of the particular system-on-chip. The predefined standard modes are (i) SLEEP, to stop the processor clock (HCLK) and (ii) DEEPSLEEP which stops the HCLK as well as turning off PLL and flash memory. The *deepsleep* bit in the system control register defines whether SLEEP or DEEPSLEEP mode is triggered when the associated sleep instructions WFI (wait for interrupt) or WFE (wait for event) are called. When the system is in sleep mode, the WIC takes over the basic functionality served by the disabled NVIC to power up the system again on allowed interrupt input. The WIC has only combinational logic therewith does not allow additional configuration. The interrupt mask is therefore copied to the WIC before the CPU system switches into a sleep mode.
For applications that just react on time distributed events, and shall then go back to sleep as fast as possible, the *System Control Register* provides the flag *sleep-on-exit*. After completing the handling of interrupts, the processor enters immediately sleep mode again.

2.5 Software Metrics

A detailed background research on software measurements will found the knowledge for firmware comparability that is treated as a key topic within this thesis. Therefore the definition and delimitation of the terminology is quite important: (i) software testing addresses activities to find potential errors, defects bugs and failures in the software, (ii) source code analysis collects and examines information about possible future errors that are mostly classified as warnings but can result in hazards, and (iii) software metrics addresses the information collection regarding software characteristics, with measurements such as *Total Lines of Code* (TLoC) or complexity estimations [Zou10]. While the first two are related and were also part of the implementation tasks of this thesis, the benchmark and results is dependent on the survey of appropriate software attributes. The following literature review and a detailed insight on software attributes will allow a differentiated understanding and the ability to determine the necessary subset of metrics for embedded software.

Measuring all aspects of software and its interaction to the environment is still a challenging broad subject area. While single terms are differently used under the hypernym of software metrics, a commonly accepted classification and applicability yet accrued. From the comprehensive perspective of software project development [Vie14] [Gwa06] metrics can be divided into three categories:

- Product metrics measure the software quality by performance, complexity or size of the program. They can be further divided into internal (static) metrics that examine attributes of the product itself (code/binary) and external (dynamic) ones, including the interaction and execution within the hardware environment and user.
- Process metrics address testing, detection and fix of defects and can be used to improve software quality.
- Project metrics pertain to the quality of the project including the quantification of cost, productivity and the project schedule.

Product metrics can be divided into internal metrics, which only depend on the software itself and not on the execution, and external metrics that can only be measured with respect of the environment, in particular the machine environment and the user. While the first ones are usually easier and objectively measurable, the latter ones depend on subjective user perception and on sometimes difficult possibilities of accomplish measurements within the environment. In addition to mention, internal metrics are often used in early development stages, while external are evaluated (normally) on an almost completed product [Fen14]. ISO/IEC 9126 [14] addresses the development of a common understanding of project objectives by qualifying four parts: (i) a quality model with six abstract objectives (functionality, reliability, usability, efficiency, maintainability and portability), connected to (ii) internal metrics, (iii) external metrics and (iv) quality-in-use metrics for the final product deployment. The relation of this metrics is that in ideal the internal determine external metrics, and external determine the quality-in-use metrics.

Internal, static product attributes can be divided into five categories [Oli08]:

(i) Coupling counts the links of a software module to other elements, is positive (min. 0 connections), does not increase on multiple module relations, and is additive: merging two modules results in the sum of the single coupling values. Metrics describing this attribute allow conclusions on encapsulation, reuse and maintainability, in particular by examining function calls and the number of instances to measure the relationship between components. *Afferent couplings* (Ca) counts number classes in other packages that depend on classes in the current package, and *Efferent Couplings* (Ce) number of classes on which classes from the current package depend, and can be used in modular programming, first proposed by [Mar02]. In this context, packages are groups of classes in object oriented programming languages. In addition, *Instability* describes the ratio of efferent coupling to the sum of couplings (Ca + Ce), where zero represents total stability of a package.

(ii) Cohesion, determines the degree of interaction within one module. It is commonly a normalized value between 0 to 1, where 0 means that all contents are not linked, adding links is positive monotone, and the result of merging two modules cannot be greater than the maximum of the single module values. Metrics can give information weather modules (e.g. classes) should be separated in subclasses. For example, *Lack of Cohesion of Methods* (LCOM) was first introduced by Chidamber and Kemerer by forming pairs of methods and shared data [Chi94]. A high number indicates a poor design, and motivating the question of splitting a module. This metric can be found in multiple variations.

(iii) Extendibility and reuse measurements qualify the degree of extension feasibility and the ease of using parts of the software in other projects. It includes *Abstractness* which is the ratio of the number of abstract classes to total number of classes in a package. Another metric is *Distance from the Main Sequence* [Mar02], describing the balance of abstractness to stability of a package by calculating

$$D = |A + I - 1| \tag{7}$$

with *Abstractness* A and *Instability* I and a favored result against zero, indicating that the package is exactly on the main sequence.

(iv) Population (size) metrics include the popular *Total Line Of Code* metric, *Number of Attributes, Classes, Interfaces, Methods, Packages, Parameters, Static Attributes* or *Static Methods* which strongly determine performance and memory footprint.

A substantial attribute of code is the size of the program code, which can be commonly counted in *Lines of Code* (LOC). This metric has to be differentiated, in particular of what to do with blank and comment lines, with headers and multiple statements in one line. The commonly accepted approach on comments and blank lines is to neglect them, and counting the *Source Lines of Code* (SLOC). This seems reasonable to decide how much space a program allocates in memory, but in terms of measuring effort for example, commented lines are also carrying programming efforts [Fen14]. The count of commented lines is the *Comment Lines of Code* (CLOC) and is related to the other numbers by

$$Total\ Lines\ Of\ Code\ (TLOC) = SLOC + CLOC \tag{8}$$

Furthermore, the SLOC without headers and footers (incl. *define* and *import* statements, brackets) is known as *None-Commented Lines of Code* (NCLOC) or *Effective Lines of Code* (ELOC). *Logical SLOC* (LLOC) is tied to specific programming languages and counts the number of executable statements, to solve the problem how to proceed with multiple statement lines.

Another benchmark to determine software size is *Function Point Analysis* (FPA), by considering amount of functionality as indicator. It quantifies functionality of the software with the focus on the value for the users, by extracting functions, classifying and assigning weights.

(v) Complexity, quantifies the degree of internal code interactions. The popular *McCabe Cyclomatic Complexity* algorithm [McC83], calculates the flow through code by incrementing a count variable each time a branch occurs. It computes a control flow graph of a software code, where the nodes represent program statements and the directed edges the relation that the second statement is executed after the first one. Control structures, such as IF, ELSE, FOR etc. create multiple independent paths through the program, which can be counted as complexity M

$$Complexity\ M = E - N + 2P \tag{9}$$

with number of edges E, the number of Nodes N and the number of connected components P.

The *Halsted Complexity Metrics* [Hal77] proceed from the assumption that code can be interpreted as a sequence of operators and operands, and includes eight metric calculations. First the *Number of Operators* (N1), *Number of Operands* (N2), *Number of unique Operators* (n1) and *Number of unique Operands* (n2) are counted. Operators are all operators defined in the C-standard (!, !=, +, -, etc.), storage and type qualifiers and reserved keywords. Control structures (if, else, switch, for, etc.) are treated differently,

by describing the alternative program flows. Operands are identifiers, type names, type specifiers (bool, int, etc.) and constants.
The *Vocabulary Size* (n) is the sum of distinct operators and distinct operands.

$$Vocabulary\ size\ (n) = n1 + n2 \tag{10}$$

The *Program Length* (N) is the total sum of operators and operands.

$$Program\ length\ (N) = N1 + N2 \tag{11}$$

The *Program Volume* (V) is the program length two times the logarithm of vocabulary size and represents the implementation content of the program. Per function this metric should result in a value between 20 and 1000, a greater number means the function ought to be divided up; the volume of a parameter-less empty function is 20.

$$\text{Program volume (V)} = \text{N} * \log2\ (\text{n}) \tag{12}$$

The *Difficulty Level* (D) or error proneness is driven by the new operators and repeated operands.

$$\text{Difficulty level (D)} = \left(\frac{\text{n1}}{2}\right) * \left(\frac{\text{N2}}{\text{n2}}\right) \tag{13}$$

The *Program Level* (L) is the inverse of the *Difficulty Level*. A more difficult (error prone) program is equal to a low programming level.

$$\text{Program level (L)} = \frac{1}{\text{D}} \tag{14}$$

The *Implementation Effort* (E) is the proportion of difficulty and program volume.

$$\text{Implementation Effort (E)} = \text{V} * \text{D} \tag{15}$$

The *Implementation Time* can be determined by dividing the implementation effort with a constant of 18. Halstead proposed this approximation for time in seconds.

$$\text{Implementation Time (T)} = \frac{\text{E}}{18} \tag{16}$$

The *Effort* of a program directly affects the number of bugs in software.

$$\text{Number of delivered bugs (B)} = \frac{\text{E}^{\frac{2}{3}}}{3000} \tag{17}$$

Additional complexity metrics are *Nested Block Depth* and *Weighted Methods per Class* to indicate alternative execution flows, element granularity and nested execution. Fenton et.al propose the importance of understanding that a single number cannot identify the complexity of a whole system, but the combination of multiple metrics can together accomplish in an encompassing understanding [Fen14].

External, dynamic product attributes describe the quality of software and are related to the environment, in particular the machine environment and the user. As a consequence, this embeddedness and interaction makes measurements much more complex as on the internal attributes which only describe the software itself. The common external attributes of software are

(i) Functionality is defined by ISO/IEC 9126 as "*A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that sat-*

isfy stated or implied needs." The proper functionality is related to the defects that are included in code, and often seen as the one and only major topic by developers in the hypernym of software quality. Therefore *Defect Density* is the ratio of detected defect count to total size of software

$$\text{Defect densit} = \frac{\text{Number of detected defects}}{\text{Software size}} \tag{18}$$

This information can be used to decide if software is ready for deployment, or on a long-term release to assess post hoc the quality level of the release.

(ii) Efficiency addresses the consumption of resources and in term of software can be measured by the execution performance on the target hardware environment.
ISO/IEC 25010 2011 defines: *"Efficiency is resources expended in relation to the accuracy and completeness with which users achieve goals"*.
In this context where software is executed on a hardware system, physical metrics determine the resource consumption. Therefore, the count of program and data memory determines the memory footprint of the software during runtime. Obviously there is a relation to the static code size population metrics, examined in [Oli08]. For embedded applications the *Energy Consumption* and the *Cycles for Execution* are additional meaningful numbers to evaluate software efficiency.

(iii) Usability describes the interaction of the software with the user and defined in ISO/IEC 25010 2011:*"Usability is the degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specific context of use."*
A simple metric to determine the effectiveness of software can be realized by setting a set of tasks to multiple users, and calculating

$$\text{Effectiveness} = \frac{\text{Number of tasks sucessfully completed}}{\text{Total number of tasks}} \cdot 100\ \% \tag{19}$$

Efficiency can be calculated by measuring the time users take to complete given tasks and relate this result to the time experts take. Satisfaction, as subject sensation is commonly identified by questionnaires.

(iv) Maintainability addresses the change of software after deployment, and defined in ISO/IEC 25010 2011: "*Maintainability is the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers".*
One measurement is to address ratio of implementation attributes to code, number of module or methods implemented. To indicate the changes by comparing a priori and post hoc status, internal metrics are applied on the code modifications. In this context, size and complexity are major features where McCabe suggested the guideline, that for proper maintainability the cyclomatic number of a module is not allowed to be greater than 10 [McC83]. The *Maintainability Index* (MI) represents, by including LOC, McCabe CC and the *Halstead Metric*, the ease of maintaining the software code.

(v) Reliability is the property of a system that indicates how safe the providing of functionality is for a time period. ISO/IEC 25010 2011 defines: *"Reliability is the degree to*

which a system, product or component performs specified functions under specified conditions for a specified period of time".
It is a well-studied quality attribute, by the common important interest of software quality in high reliability by often ignoring other attributes. Based on probability theory and prediction models, various measures can be determined. The *Rate of Occurrence of Failures* (ROCOFs) is the number of failures in a given time interval. To describe how long a system runs successfully between the occurrences of failures, *Mean Time between Failures* (MTBF) is

$$\mathrm{MTBF} = \mathrm{MTTF} + \mathrm{MTTR} \tag{20}$$

with *Mean Time to Failures* (MTTF), the expected value of the failure distribution function, and *Mean Time to Repair* (MTTR), an estimated or empirically determined value how long it takes to locate the error and repair the system. The *Availability* of a system is then simply calculated by

$$\text{Availability} = \frac{\mathrm{MTTF}}{\mathrm{MTTF+MTTR}} \cdot 100\ \% \tag{21}$$

(vi) Portability qualifies how easy the software can be ported to another hardware or user environment. ISO/IEC 25010 2011 defines: *"Portability degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another"*.
ISO/IEC 9126 [14] describes potential portability compliance metrics as a measure of transfer ability to another environment by assessing applied standards in the software to total available ones.

Summed up, the list of potential metrics for the comparison of the HAL variants is taken together in Table 2-1. In general these metrics can be applied to any type of software. However, the particular application characteristics, the hardware environment, the type of user interaction and of course the objectives of interest will result in a subset of this comprehensive list.

	Name	**Attribute**
Internal Metrics	Lines Of Code (LOC)	Code Size
	Number of Parameters (NOP), Methods (NOM), Classes (NOC), Interfaces (NOI), Packages (NOP), Static Attributes (NOSA), Static Methods (NOSM)	Code Population
	Function Point Analysis	Code Size
	Halstead Metrics	Code Size, Complexity
	McCabe Cyclomatic Complexity (McCabe CC)	Complexity
	Method Lines Of Code (MLOC)	Complexity
	Nested Block Depth	Complexity
	Weighted Methods per Class (WMC)	Complexity
	Afferent Coupling (Ca)	Coupling
	Efferent Coupling (Ce)	Coupling
	Instability (I): Ce / (Ca + Ce)	Coupling
	Lack Of Cohesion Of Methods (LCOM)	Cohesion
	Abstractness	Extendibility / Reuse
	Depth of Inherence Tree (DIT)	Extendibility / Reuse
	Number Of Overridden Methods (NOVM)	Extendibility / Reuse
External Metrics	Defect Density	Functionality
	Maintainability Index	Maintainability
	Effectiveness Metrics	Usability
	Efficiency Metrics	Usability
	Satisfaction Metrics	Usability
	Rate of Occurrence of Failures (ROCOF)	Reliability
	Mean Time Between Failure (MTTF)	Reliability
	Portability Compliance Metrics	Portability
	Program Memory	Efficiency
	Data Memory	Efficiency
	Energy Consumption	Efficiency
	Clock Cycles	Efficiency

Table 2-1: State-of-the-art software metrics

2.6 Related Work

You et.al presented a test method for power management modes in SoC [You11], motivated by the functional key role of the PMU and the potential corruption of state information and data in volatile memories. They described three typical power modes, (i) Standby, by applying clock gating, (ii) Retention, by applying clock gating and power gating to the core domain, but still supply at least a small portion of memory, and (iii) Power-off, where all components are clock- and power-gated. For the tasks to do an "idle state"- driver design was presented, by setting the interrupt mask, saving the processor states to retention memory and after sleep and wake-up trying to resume the processor state to return to the original state. Secondly, they proposed some simple ways to estimate the power consumption and transition delay of the processor to quantify these indicators. However, the shown practices bring accuracy into question, but conclude with the essential importance of power management testing to ensure functional correctness.

A "Hardware Abstraction Layer Generator" was presented in [Lin10] to address the problem that test case coding is a critical phase during functional verification and the challenge is to write bugless lowest-level C or assembly test cases. A time reduction in the verification process of 33% is proposed, and obviously an increase in test execution time. However, there is no information available on the ongoing status of this project.

3. Proposed Concept

Based on a current development project of a system-on-chip for wireless sensor applications at Infineon, a three layer hardware abstraction design in accordance with ARM CMSIS is presented to survey the combined implementation of a verification and productive API variant with appropriate software metrics. This chapter will be structured as follow: first the SoC design and the corresponding hardware modules are annotated. Subsequently, the implemented power management strategies, which are targeting the low power requirements, are explained. Based on the hardware architectural exploration, the within this thesis developed HAL design is presented in detail, with the realization of the two variants with different requirements regarding the level of abstractness from the low level register access and the interface to the application layer. For a profound evaluation of this concept, several software attributes are measured on the implementations by a subset of the presented software metrics from chapter 2.5. The selection process is explained by a review about applicability on the bare-metal HAL. These metrics are exemplary evaluated on two essential, custom modules, the power management unit as a system-critical component and the transceiver-interface which drives the transceiver module for wireless communication.

3.1 ARM-based SoC with a custom Transceiver Module

The proposed system-on-chip is a development project that shall meet the requirements of next generation wireless sensor applications and is currently under development in pre-silicon stage at Infineon Austria Technologies AG. By using state-of-the-art chip architecture, flexible communication interfaces and innovative long-lifetime power management strategies, a broad field of applications could be served. An ARM low-power processor and standard bus systems form the core unit of the chip and are connected to various peripherals, including a sub-GHz transceiver module to communicate with other devices or base stations. Therefor chosen was an on-chip integrated, Infineon TDA 5340 module [15] that was redesigned for the technology of this SoC project. Equipped with a 64kB flash module the system enables even complex firmware implementations or to cache data from the on-chip sensor interface. Simple reprogramming is provided with a SerialWire and a UART interface. In addition, an RFID interface allows near-field communication that can be used for example to configure or readout the wireless sensor application with a smartphone. Equipped with a custom developed power management system, a high coverage on clock and power gating could be achieved, allowing applications to drive only parts of the chip that are actually needed, to realize longest possible lifetime.

Architecture

The system-on-chip architecture has to be flexible for serving a broad field of applications. The dominance of ARM in the microcontroller sector enables code-reuse, a well-tested system and a quasi-standard of the processor core and bus interfaces. This fact allows silicon vendors the easy reuse of hardware IP-blocks among various development projects without any additional adaptions, from small low-power to high performance microcontrollers. In the proposed project, presented in Figure 2.1, due to the requirement of lowest power consumption for battery-driven operation, the smallest available ARM processor, Cortex-M0 is used. Besides the core unit itself it is equipped with an interrupt controller and a debug access port to enable the debugging and access via Serial Wire for post-silicon application development.

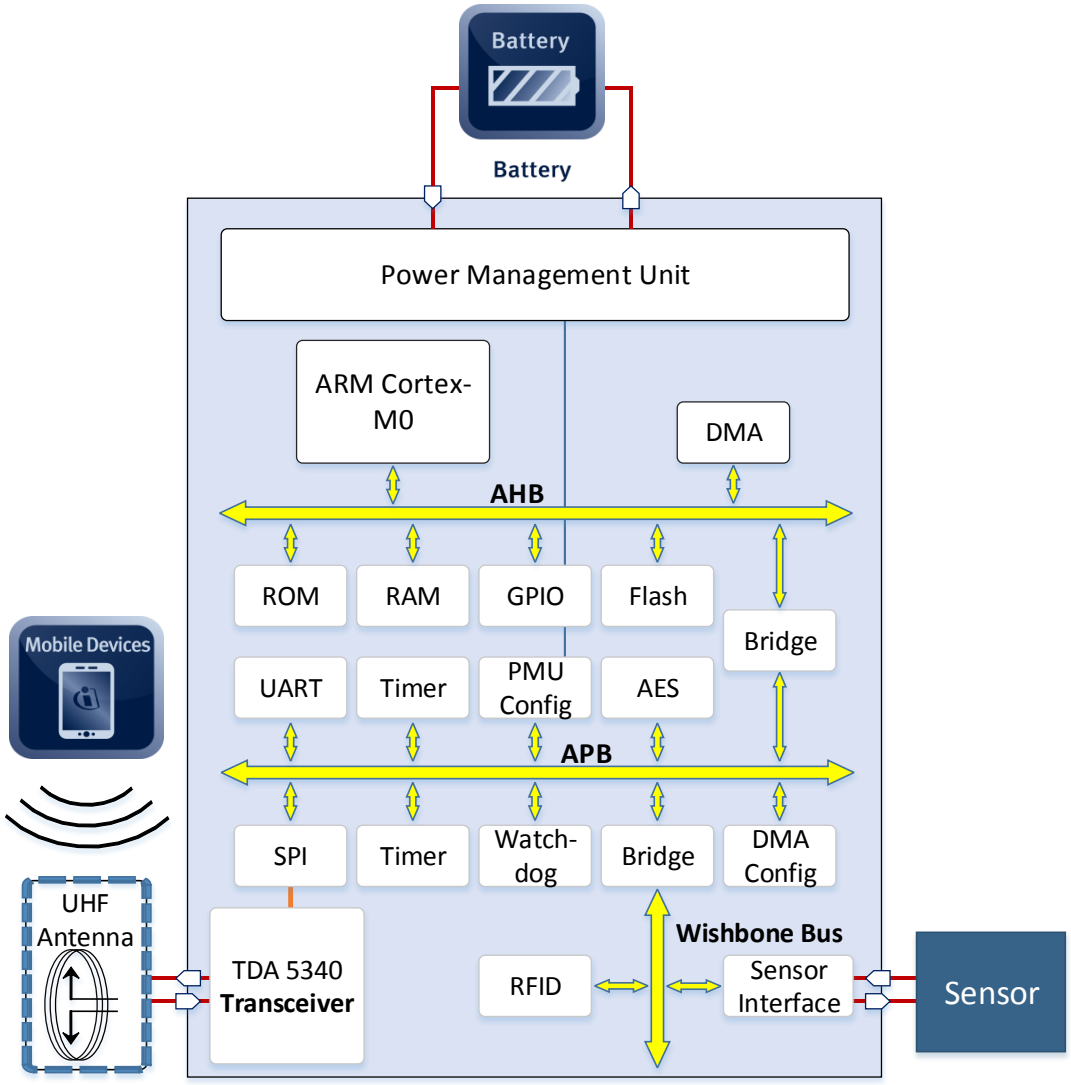

Figure 3.1: System-on-chip architecture

The Cortex-M0 processor unit is connected via the AHB to the high speed components, including three memories:

(1) An 8kB ROM is equipped with a small bootloader, to configure core components on start-up and to check whether the control shall be given to the flash firmware or to enter the UART/SerialWire programming mode. This choice can be made by setting a predefined boot-pin at the GPIO before powering up the device.
(2) A 4kB RAM contains the volatile data during program execution, and is also used in programming-mode. Therefore, first flash tool software is loaded into and executed from RAM and providing the load procedure of the actual firmware data from UART to the flash memory.
(3) A 64kB flash module for non-volatile storage of firmware, configurations and data from the sensor interface.

To serve the flexibility of an external connection interface for simple communication protocols a 16-pin GPIO module is accessible via the AHB bus system. The DMA can copy data between peripherals and memory while the CPU can handle other tasks. Even event-driven activation is possible, by a handshake and request/acknowledge interface to the peripherals. For example a preconfigured DMA task can handle the periodical requests of the UART when a packet is in the receive buffer ready to be copied to RAM. The AHB bus-matrix therefore allows parallel connections of the CPU and DMA to other components, and applies a round-robin scheduling when both want to access the same resource. At last module to be mentioned at the AHB domain a bridge connects to the slower APB.

On the APB bus domain, a UART module provides a simple serial communication to external devices, and can be used to flash firmware, copy configurations or is also popular for console debug output by redirecting the appropriate C-commands *printf* or *puts*. Timers are used to interrupt the system one-time or periodically to enable counting timespans. To give a simple example, it can be used to control the brightness of an LED by creating pulses at the output of a GPIO pin. The power management configuration registers are a custom module including control to energize and isolation states of modules or put the system into one of the predefined power management sleep modes. By the comprehensive functionality of this custom developed module where wrong configuration settings or malfunction can bring the system into critical states, by for example disabling all wakeup detection modes and putting the system into a deep power state, the functional correctness is very important. A hardware implementation of the *Advanced Encryption Standard* (AES) enables en-/decryption with high data rates and therewith lowest latency without facilitating the systems processor unit. The security requirements for wireless transmission can therefore for example be realised as follow: A block of data, ready for transmission is stored in the RAM. The CPU configures DMA tasks to copy the data subsequently to the AES and transfer them when finished to the transceiver module. Since all of these tasks are not software implemented and outsourced to the peripherals and DMA, the CPU can meanwhile execute other tasks, or wait for completion in an energy-saving sleep mode. The aforementioned transceiver module is a redesigned TDA 5340 module that is integrated as macro on the SoC. It is equipped with a SPI interface which is on-chip connected to a SPI master module that provides the interface to the APB bus. The multiband, sub 1 GHz transceiver is designed for lowest system power consumption and up to +14dBm output power [15].

Bus	Component	Description
-	Power Management Unit	Manages the power states of the system (e.g. *SLEEP*, *DEEPSLEEP*, *POWERDOWN*) configurable via the APB configuration interface
AHB	Cortex-M0	Central processing unit (CPU)
AHB	DMA (Direct Memory Access)	Controller to transfer data independently from the CPU
AHB	ROM (Read-Only Memory)	Non-volatile memory, contains bootloader (cannot be changed after manufacture)
AHB	RAM (Random-Access Memory)	Volatile memory for data during execution
AHB	GPIO (General Purpose Input/Output)	16 pins for additional external digital control lines
AHB	Flash	Non-volatile reprogrammable storage
AHB/APB	AHB-APB-Bridge	Connection between AHB and APB domain
APB	UART (Universal Asynchronous Receiver/Transmitter)	Serial communication to external devices
APB	Timer	Multiple timers that interrupt system on overflow, once or continuously
APB	Power Management Config Registers	Status and configuration of the power management Unit
APB	AES (Advanced Encryption Standard)	Hardware implementation of AES algorithm
APB	SPI Master / TDA 5340 Transceiver Interface	Interface to transceiver mode, for wireless communication to other devices
APB	Watchdog	Timer to detect and recover from system faults, can reset the system
APB	DMA Config	Status and configuration of the DMA controller
APB/WB	APB-WB-Bridge	Connection between APB and Wishbone domain
WB	RFID (Radio-frequency identification)	RFID interface
WB	Sensor Interface	Connection to application specific sensor

Table 3-1: SoC hardware modules

A watchdog is used to detect and recover from system faults by resetting the system. It is a 32-bit counter, which raises an interrupt when reaching zero. On the next clock edge the timer is reloaded and continues down counting. When until the next time the counter reaches zero a reset of the interrupt was not monitored, the watchdog asserts the reset signal. The DMA has its data transfer interface at the AHB bus, while the configuration is set via an APB interface. 4kB of memory is allocated to set the controller configuration registers such as reading the status, starting the DMA, or setting the address of channel control data structures stored in RAM. At last, a bridge connects the APB to the open source 8-bit *Wishbone* bus system [13], which is integrated because of an ancestor project for reasons of component reuse. Most importantly it connects a RFID module to support ISO 14443-a tags and a sensor interface to connect the application specific sensor module. This can be for ex-

ample a pressor sensor or an amperometric sensor. Table 3-1 again summarizes all components of the presented SoC.

Power Management Strategies

The reasonable application of power management techniques, discussed in chapter 2.4 allows the system to utilize its limited amount of energy from battery for highest efficiency and longest lifetime. To reach this goal, functional domains shall be only powered on and provided with the clock signal when actually used. As a demonstrative example on a typical sensor application, data is not streamed periodically in that way that the whole system is constantly facilitated, the CPU subsystem therefore could go into a sleep mode until a timer or an external interrupt signal indicates the next data processing step. To address the different functional groups including each multiple modules, three power domains are defined:

(i) MCUSYS: contains the Cortex-M0 processor system and all peripherals of the AHB, APB und Wishbone subsystems

(ii) PMU: the power management unit, uncoupled, to control the state of the other domains

(iii) AHB-RAM: forms an own power domain to enable power-gating of the CPU-subsystem while preserving the content of the RAM

To provide an adequate granularity in controlling the states of these domains, 6 different power management modes are defined, listed in Table 3-2 with the associated behaviour. They are linked to possible clock gating of the four clocks used in the ARM system, and power gating of the defined power domains. In addition the disabling of the clock oscillators can save additional power consumption, but results in a longer wakeup time due to the oscillators' settling time.
The first two power management modes are ARM built-ins which can be activated in firmware by calling the WFI (wait for interrupt) or WFE (wait for event) native instructions. The first one targets the wait for an interrupt to wake-up the system, while the latter is to realize semaphores in a multiprocessor system. It can be used in spinlock loops, to avoid busy waiting for exclusive resource access. In combination with the SEV instruction, a CPU can signalize the waiting others that the resource is free for allocation.
By setting in the Cortex-M0 *System Control Register* (SCTLR) the SLEEPDEEP bit defines weather the processor goes into sleep or deep sleep mode when calling one of the standby instructions. By setting the SLEEP-ON-EXIT bit in the same register, the processor goes back to sleep when an interrupt service routine handling was finished. This makes short wake-up times possible to react on an interrupt from a peripheral, process it and send the system immediately back to the sleep mode until the next interrupt is raised. The other four modes are custom defined, and controlled by setting the appropriate SFRs of the power management unit via the APB configuration interface.

Power management mode	**Clock Gating**				**Power Gating**			**Oscillators disabled**
	HCLK	SCLK	FCLK	PCLK	MCUSYS	PMU	AHBRAM	
ARM -SLEEP	X							
ARM-DEEPSLEEP	X	X						
SLEEP	X	X	X	X				
DEEPSLEEP CLK-ON	X	X	X	X	X			
DEEPSLEEP CLK-OFF	X	X	X	X	X			X
POWER-DOWN	X	X	X	X	X	X	X	X

Table 3-2 Power management modes and corresponding applied power saving techniques

The ARM-SLEEP mode gates just the HCLK, which is the appropriate clock for the AHB components and the processor core. Consequently, the CPU subsystem freezes without losing the state information, since the power supply is still available, while the APB peripherals can further operate connected to the PCLK. The NVIC is supplied by the FCLK and thus, still available for peripheral interrupts to wake-up the processor core within one cycle.
The ARM-DEEPSLEEP mode gates in addition the FCLK, and thus deactivates the NVIC for interrupt handling. Therefore the much smaller WIC is then responsible for handling incoming request and wake-up of the system. In the ARM reference implementation this mode also includes power gating, which was redesigned for this SoC and shifted to the more powerful implementation of the custom PMU modes.
The SLEEP mode applies clock gating on all system clocks, the HCLK, FCLK, SCLK and PCLK. Hence the whole microcontroller system is frozen, but can be waked-up by the within the PMU integrated wakeup timer or the event detector for external GPIO input within one clock cycle. The DEEPSLEEP-CLK-ON mode includes power gating, by turning off the MCUSYS domain but keeps the AHBRAM enabled for data retention and the PMU for wake-up. Since the system has to boot when reactivated and restore its state, a longer delay is expected. DEEPSLEEP-CLK-OFF serves the same power techniques, but in addition disabling the chip oscillators. As a consequence the more stable synchronous event detection is disabled too and replaced by the slightly error prone asynchronous event detector, which has to struggle with spikes on input signals. To be mentioned, the reactivation and settling of the oscillators adds an additional wake-up delay. POWERDOWN disables the overall digital system of the chip, thus only analog components on the chip can boot-up the system again. As a result the lowest possible power consumption can be achieved, but results in the highest delay in system reactivation.

3.2 Hardware Abstraction Layer for Processor-driven Verification

The rising system complexity in current microcontroller systems and the demand for flexibility in application code reuse substantiate the common approach of dividing the software into hardware dependent and independent part. To ensure compatibility in the de-facto standard ARM environment among various projects the incorporation of the ARM CMSIS is reasonable. To undertake the survey of combined development of a HAL as verification and productive API a three layer HAL is presented. The basic two layers remain the same for both variants, while the layer on top is custom for the particular use-case. Therewith the different requirements regarding functionality and the interface to the application layer can be realized.

Layer Design

To divide the bare-metal API for hardware access in a hardware dependent and independent part, a common three layer architecture was chosen in accordance to the design guidelines of the ARM CMSIS architecture. While the lowest layer itself only contains definitions in header files, combined with the middle layer basic driver functionality is provided to the top layer. Figure 3.2 shows the schematic design, with the microcontroller and the component blocks at the bottom. To explain the purpose of each of the three layers, the example on a flash module will be given, see Table 3-3.
The lowest layer (Layer 0: Register Definitions) of the HAL is formed by the register definitions, where register addresses are defined relatively to the components base addresses and mapped into symbolic names. In addition masks and bit values are declared with symbolic names and their write- and readability, to avoid for example the write access to read-only status registers. On the same level located is the ARM-specific CMSIS-CORE with the key features of abstracting the processor core registers, the NVIC, debug and trace access configurations. On the flash module example to erase a sector, registers for the sector address and control registers to start the erase operation are be defined here.
On the next layer above (Layer 1: Basicdrivers) the register level access is abstracted into basic routines, which are still dependent on the hardware used. The multiple register assignments are encapsulated into basic routines, for example *set_address(addr)* to calculate and set the sector address register, and an *erase()* routine to set the control register and check the correct completion.
The layer on top (Layer 2: Drivers) uncouples the application layer interface from the specific hardware used, by using standardized routine templates for classes of devices. The CMSIS driver specification includes 11 definitions: CAN, Ethernet, I2C, MCI, NAND, Flash, SAI, SPI, Storage, USART and USB[1]. Additional components, used in the presented system-on-chip architecture are custom defined to enable a consistent interface, and can be reused for similar ARM-based SoC developments. For the flash example, ARM CMSIS driver specification for flash devices includes the function *int32_t ARM_Flash_EraseSector (uint32_t addr)* to provide a generic API.

[1] CAN… Controller Area Network, I2C … Inter-Integrated Circuit, MCI … Memory Card Interface, NAND … Not-And Flash, SAI … Serial Audio Interface, SPI … Serial Peripheral Interface, USART … Universal Synchronous and Asynchronous Receiver/Transmitter, USB … Universal Serial Bus

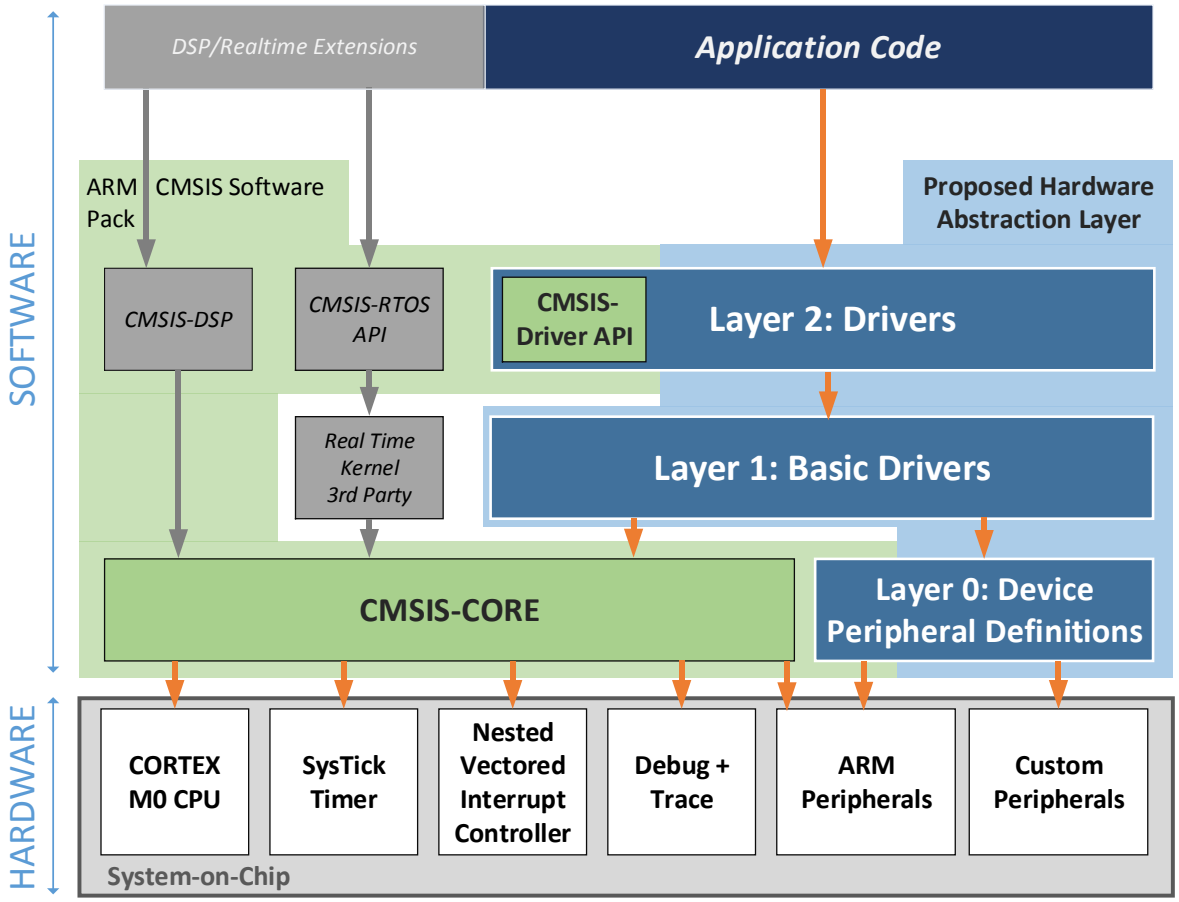

Figure 3.2: Proposed HAL design in accordance with ARM CMSIS

Layer	Example: Flash module
Layer 2: Driver (Hardware/Vendor-independent)	ARM_Flash_Initialize (…) ARM_Flash_ReadData (…)
Layer 1: Basic Drivers (Hardware/Vendor-dependent)	Startup_flash_memory(…) Remap_sector(…)
Layer 0: Device Peripheral Definitions (Hardware/Vendor-dependent)	#define AHB_FLASH_STATUS #define AHB_FLASH_CTRL #define AHB_FLASH_MODE

Table 3-3: Flash module example of HAL layer competencies

Use-Case Variants for Verification and Productive API

The proposed HAL design should be both used in productive deployment as well as in the purpose of hardware verification testing to abstract the access from register level to an appropriate application interface. Both variants aim therefore different objectives, listed in Table 3-4, including the granularity of low level register access. The coverage and quality of hardware verification is obviously dependent on a fine granularity level to monitor all various status registers after every possible register manipulation. Hence, the high functional coverage of the underlying hardware, by facilitating and testing each configuration setting and is reflected by an extensive API. The productive use-case will provide a slightly defined interface, by abstracting most of the complexity of register configurations, apply default settings and provide only necessary configuration and status access for the intended use of the module within the SoC.

Objective	**Verification**	**Productive**
Granularity	Fine	Coarse
Level of Abstraction	Low, to enable profound configuration and status flag access	Complexity obscured by simple access members
Interface to Application Layer	Broad	Slight
Functional Coverage	High	Medium to Low

Table 3-4: Objectives of the two HAL variants

The major challenge is to identify the requirements for both use-cases in the a-priori software concept stage, extract functional intersections to achieve an efficient trade-off for both application scenarios. In the previous subchapter, the three-layer architecture with a different implementation of layer 2 was already presented, to meet the requirements for two use-cases. Figure 3.3 outlines the interaction of the layers for the verification variant to provide accesses and to an example hardware module, which provides several functional registers to set configurations, read status bits, data and control the module's processes. Layer 0 defines the access to these registers by their addresses and the meaning of the values by textual mask identifiers. Obviously, for the verification purpose all configuration registers as well as status bits are facilitated to obtain the highest possible test coverage. Above, layer 1 composes the combination of multiple register access in a collection of parameterized functions. These hardware dependent methods are providing still the broad functional coverage, but simplify the access by encapsulation to a smaller set of functions. A reasonable division therefor, and the number of parameters allowed per function will be discussed in detail on the evaluation results in chapter 5. It is connected to considerations regarding performance and memory footprint. On top, layer 2 provides a hardware independent, standardized interface from test applications to the hardware specific layer 1 functionality. The challenge is to define an interface that allows the reuse of the tests but connecting them to the underlying basic drivers without losing functional coverage.

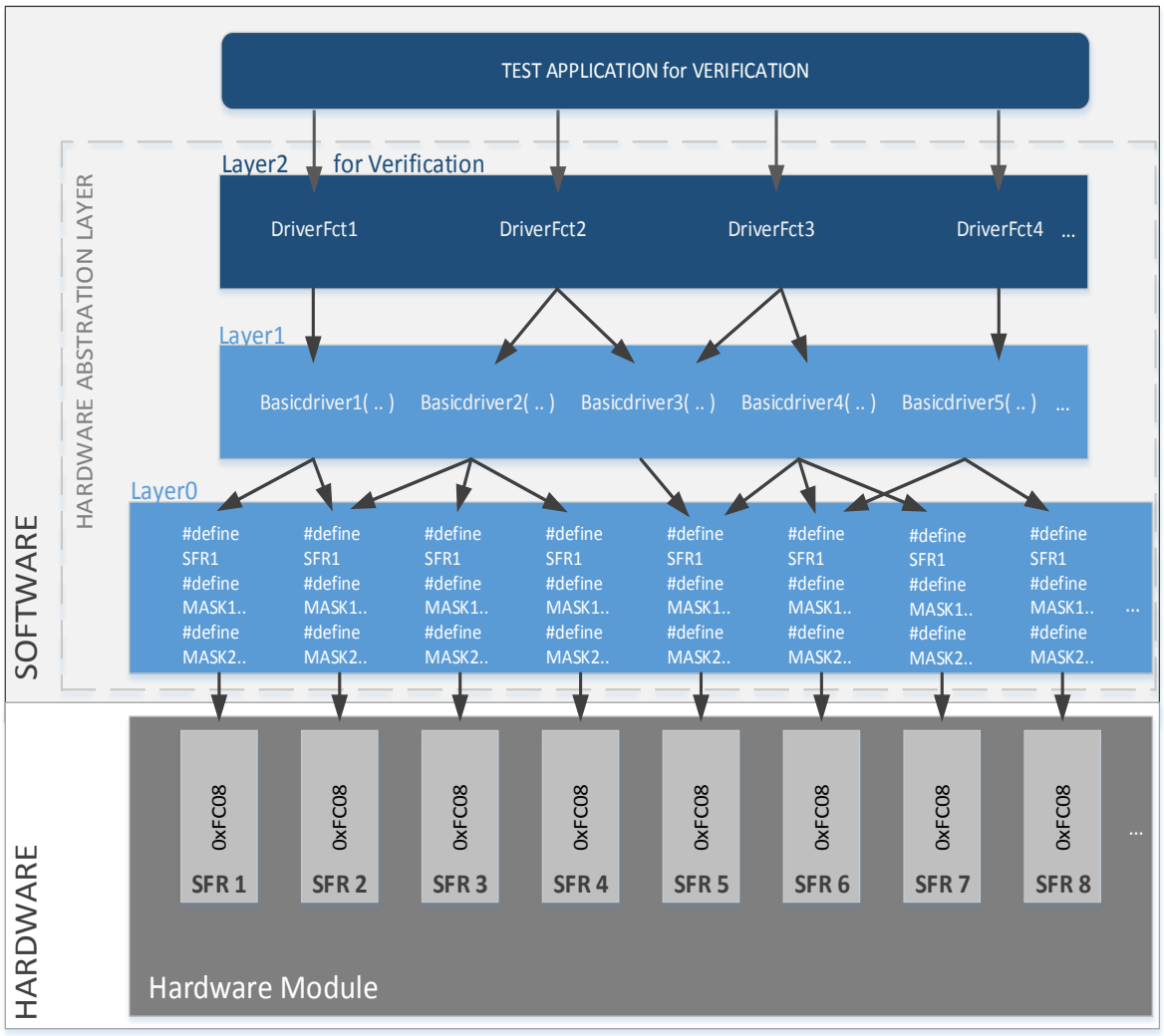

Figure 3.3: Interaction of HAL layers for the use-case verification API

Figure 3.4 outlines the layer interaction for the productive library use-case, by reusing exactly the same implementation for Layer 0 and Layer 1 to abstract from hardware functional registers up to the basic driver functionality. Nevertheless, the productive version does not facilitate all registers definitions and basic driver functions (grayed out in Figure 3.4), for example certain status registers or special functional modes used for debugging or in other SoC applications. Anyhow, the full source code is added and used during programming, but just the necessary parts of code is then added by the linker in the binaries build process. It resolves the dependencies from application code downwards, unreferenced files and functions are neglected. Therewith, the additional code is not affecting the binary size. This is the key aspect that the total reuse of code can be accomplished and likewise meets the slight interface requirements for a productive API deployment. On top of the design, layer 2 provides the abstracted slight interface to the application, by connecting driver functions to the fixed subset of configurations and actions of the basic drivers.

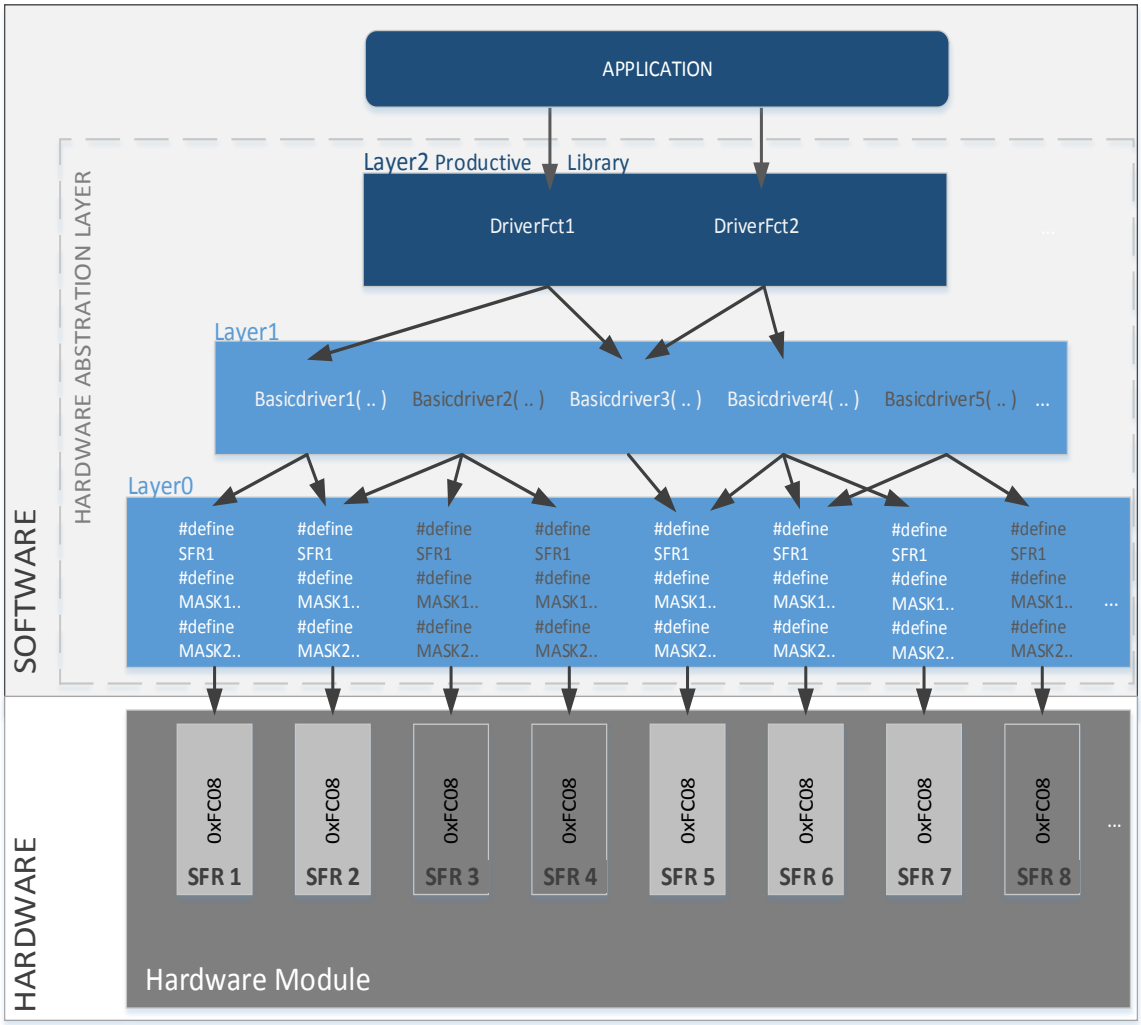

Figure 3.4: Interaction of HAL layers for the use-case productive API

3.3 Evaluation of Verification and Productive HAL Variants

In the previous chapter the three layer hardware abstraction design with the proposed advantage of code reusability, by replacing the highest layer for the particular use-case variant, were presented. To evaluate the implementations with valuable quality assertions, software metrics are used and the results discussed to justify the measured attributes as well as the meaningfulness of the proposed concept. The evaluation is described on two examples, the power management unit, which allows comprehensive power control operations, including power energize and isolation on the peripherals. The multitude of configurations shall be encapsulated in a defined set of power management modes for application development but all possible combinations of settings applied on verification. The second example is the transceiver interface (TRX-IF) to provide communication and configuration to the TDA 5340 transceiver module, which has around 260 hardware registers for settings and debug access. However the interface itself is quite simpler, but must create the wave forms expected by

the transceiver to be able to interpret the data. Both two examples serve a key functionality on the chip, and chosen as example for a meaningful examination with the different requirements of full functional coverage for verification and slight simplified interface for a productive library.

Suitable Metrics for Embedded Software

Measurements on software are applied to obtain information about attributes to meet product requirements or to make it comparable to other software. In chapter 2.5 a comprehensive review on state-of-the-art metrics was given, divided into internal (static) metrics, that evaluate internal attributes that are only related to the software itself, and external (dynamic) metrics by examining software within the hardware environment and user interaction. In the context of embedded systems, the hardware is delimited and the necessity and type of user interaction application dependent. It can range from no user interface, via LEDs and buttons to complex interfaces using touch screens. The heterogeneity of the hardware environment, by an indeed de-facto standard of ARM architecture but mostly in combination with custom functionality and various contrary requirements (e.g. battery-driven milliwatt vs. power electronics applications) make the objective evaluation of external attributes quite challenging.

"*Embedded software is defined as a special-purpose software system built into a larger system. The end user usually doesn't recognize embedded software as software in the traditional way; instead, he or she perceives it as a set of functions that the system provides.*" [Ebe09]

This definition of embedded software explains the unawareness of the user to the functionality and sometimes either existence. Another aspect of embedded software is that it runs usually not as task of an operating system, but directly on hardware as so called bare-metal software. The appropriate source code is mostly written in C and single routines in Assembler, simple but powerful procedural languages that however not support high-level concepts such as object-orientation. As a consequence, the applicability of several metrics is restricted or not even possible. With these elaborated limitations of the embedded system environment and the potential metrics for software analysis summarized in Table 2-1 the following part of this chapter will present the selection process to get a set of meaningful metrics that can be applied on embedded software in principle, and further for the task of comparison of the two HAL variants.

For this elaboration, product metrics will be examined in detail to measure and compare the software by internal and external attributes. Process metrics, which cover the testing process, detection of bugs and improvement of quality will be neglected, as well as project metrics, including attributes of project scheduling or development productivity. In the current scope the design of the HAL shall be evaluated, where aspects of project and quality management are subsidiary, but however not unnecessary in the whole SoC development project and firmware development stage.
In the category of **internal metrics**, five categories were listed:

(i) Coupling represents the linkage of one module to others. Thereby the meaning of the term module has to be declared first on C language. In object-oriented programming, provided for example in C++ or JAVA, this term is mostly equated with classes that represent a template for object creation. However, the imperative C and Assembler language do not support object-orientation, but are designed for structured programming.

Thereby, the C standard provides a multiple file concept where header files provide the public interfaces of modules. By including a header file with function declarations from another module, the functionality can then be called. Nevertheless, the loose concept of module inclusion by the hierarchical file structure and especially the code include instead of instantiation concept make the attribute of coupling not expressive in this context. Strict interfaces in an object-orientated design are required for applicability.

(ii) Cohesion, as the contrary to coupling refers to the inner interaction of a module and is also strongly related in the context of software engineering to object-oriented design. Thereby classes with low cohesion indicate multiple non-related tasks that should be outsourced into separate objects. However, it could not be applied on C and Assembler source code.

(iii) Extendibility and reuse measurements qualify the effort of adding post-hoc functionality to the software and reusing parts of code in related projects or reengineering tasks. The metric *Abstractness* which sums the number of abstract classes cannot be applied on C or Assembler Code. Abstraction as an OOP Concept has no direct equivalent in procedural programming. However by the custom design specification for a hierarchical structure of the HAL, where each layer is built up on the lower one, therefore a reuse of code can be described by the encapsulation of layer functionality. Compared to the OOP concept, where each layer would form mainly one object, the objects are here described virtually by a source code file according to one layer. By calculating the size of each layer $SLOC_0$, $SLOC_1$ and for the Layer2 variants $SLOC_{2,V}$ for verification purpose and $SLOC_{2,P}$ for the productive library the reuse of code from verification during the software and hardware co-development to the stage of software development for a productive library is proposed to be

$$\text{SLOC reuse in productive HAL} = \frac{SLOC_0 + SLOC_1}{SLOC_0 + SLOC_1 + SLOC_{2,P}} \cdot 100\ \% \tag{22}$$

The code reuse is the sum of SLOC of layer 0 and 1 that is reused in the productive library divided by the total lines of code. Analogous to this definition we can also evaluate how much code is being reused from the verification implementation by simply calculating

$$\text{SLOC reuse of verification HAL} = \frac{SLOC_0 + SLOC_1}{SLOC_0 + SLOC_1 + SLOC_{2,V}} \cdot 100\ \% \tag{23}$$

(vi) Population (size) metrics, includes the popular *Total Line Of Code* metric, *Number of Attributes, Classes, Interfaces, Methods, Packages, Parameters, static attributes or static methods* which strongly determine performance and memory footprint. The reuse estimation proposed in (iii) already used SLOC as size measurement. To recap an important statement from Chapter 2.5: LOC is not the only and often not the most suitable size measurement, as it does not give any information on the functional content of these lines. As an alternative *function point analysis* was discussed calculating a dimensionless number by counting scores of code items with the value for the user. In the intro-

duction to this chapter, the often missing awareness of the user about the software presence in embedded systems was listed. Since this SoC as wireless sensor node has no user interface at all this metric is not applicable. For the evaluation of HAL the LOC metric should be satisfying anyway, provided with explicit conditions and discussions about the expressiveness in combination with other metrics. *Number of methods* and *parameters* is in line with the language features of C code and will be used in extension to LOC to examine the internal software attributes. *Number of classes, attributes, interfaces, static attributes,* and *packages* are addressing OOP concepts and therefore neglected.

(vii) Complexity is measured to determine the degree of internal code interactions. *McCabe Cyclomatic Complexity* algorithm, calculates the flow through code by incrementing each time a branch is counted. It can be used in procedural languages like C and allows conclusions about the proper design of the HAL, where greater numbers as 10 indicate that splitting a function up would be useful. In this way, the single layers can be examined, and the question about the reasonable number of functions answered. The eight simple measurements of the Halsted Complexity Metrics will allow additional information about static code attributes, and are appropriate applicable on C-Code.

External metrics were surveyed in chapter 2.5 on six different attributes:

(i) Functionality includes the set of functions that shall be satisfied and also the estimation of defects that obstruct this goal. The overall interest in comparing the HAL versions is to get information about the concept of combined HAL development, where the *Defect Density* is not identified as major objective. Nevertheless, in software development these metrics can provide an important contribution to software verification.

(ii) Efficiency addresses the consumption of resources during execution, which is an interesting attribute on embedded systems. Especially the straight limitation of energy supply by a small battery, reasons major interest in efficient system design. Therefore, the hardware design and power management is considered to enable maximum battery lifetime. Battery-driven sensor SoCs normally spend most of the time in low power states, and just reactivated for a short period of time to react on input changes or to periodically process tasks. Obviously, the execution time measured in *Clock Cycles* determines the up-time and hence power consumption. A common term of *CPU Facilitation* is not applicable here, where bare-metal software is executed directly on the system, and no scheduling is applied. Furthermore in embedded systems, less memory is commonly available to reduce chip size and dynamic power consumption. This limitation is a straight delimiter for software design, and concerns therefore in particular the HAL. Since it is no full-valued software, but instead providing functionality for the actual application firmware code, the memory footprint should be as small as possible. Memory resource measurements can be applied on program and data memory. First one can be evaluated by analyzing the compiled binaries, depends on the instruction set and optimization techniques that are available on most standard compilers. The latter one is measured during execution, as minimum, maximum or average values, by examining directly the content stored in RAM and monitoring the access and addresses via the bus interface.

(iii) Usability describes the interaction of software with the user. The consequence of the slight or non-existent user interface was discussed in the introduction to this chapter and on other metrics yet. The presented SoC and the HAL itself have no user interface anyway; hence this attribute is not present at all. Usability in terms of how easy the API can be integrated and accessed in application development is not treated in this context.

(iv) Maintainability measures the ease of software modification after deployment. In the current pre-silicon development and thesis topic this attribute is not of superior interest. However, the maintainability index is an interesting metric that associates the results of various other metrics, and can cover therefor in a single number multiple attribute quantifications.

(v) Reliability estimates the system uptime and failure probability. The strong dependency on the reliability of the hardware system and the determination through a prediction model is not a component of this thesis and as a consequence, appropriate reliability metrics neglected. The creation of an appropriate model to describe the system reliability by the interaction of software with the hardware environment is quite challenging and includes a future comprehensive analysis of the chip in post-silicon project steps in prototype to product development.

(vi) Portability measurements qualify the transferability of software to other hardware systems. Since embedded software is due to the custom hardware strongly dependent on the hardware environment used, portability is a difficult attribute to measure. However by the layered structure of the HAL, where Layer 2 is independent from the particular hardware implementation used, and written in standard-C code, at least the portability of layer 2 is with compliance of the CMSIS driver classes fully given.

	Metric Name	**Description**
Static Code Metrics	Lines of Code (LOC)	Lines of code without blank and comment lines.
	Number of Methods (NOM)	Total number of methods defined.
	Number of Parameters (NOP)	Total number of parameters
	Code Reuse	Custom measurement to express the code reuse of Layer 0 and Layer 1 among the HAL variants
	McCabe Cyclomatic Complexity (McCabeCC)	Each time a branch occurs this metric is incremented. High values imply a high complexity and number of alternative program flows
	Halsted Complexity Metrics	Analysis of sequence of operators and operands and calculation of eight metrics
Dynamic Metrics	Program Memory	Amount of memory needed by the firmware
	Data Memory	Amount of memory needed by the data
	Clock Cycles	Number of clock cycles for execution

Table 3-5: Selected Metrics for Hardware Abstraction Layer evaluation and comparison

With the limitation in embedded systems and bare-metal software the large number of potential software metrics was analysed and reduced to a proposed set of suitable metrics, see Table 3-5. They are used to evaluate the characterizing attributes of the two HAL variants on the example of two modules, which are described in the following subchapter.

Evaluation on the Example of two Hardware Modules

To evaluate the proposed HAL concept and apply the two implementation variants, two example modules of the presented SoC are used exemplary for applying the proposed set of metrics. This proof-of-concept shall allow a profound discussion about the HAL design in general, and the applied use-case specific layer 2 implementations. To choose two hardware modules, the selection was oriented on custom developments instead of out-of-the-box ARM components and such with system-critical or key functions of the microcontroller. In this way the following two modules were selected:

The PMU is a custom module to control the power states of the multiple power domains, defined in chapter 3.1, including clock and power gating of the CPU subsystem, peripherals, RAM and the PMU itself. The switch of clocks, the power-up and shutdown of components and the control of clock gating are major tasks. In addition related tasks include the declaration of wake-up sources (GPIO pins, built-in wakeup timer) and state freeze of GPIO outputs to retain the values even when the system is in sleep mode to provide it to external devices. With the complex cross-linkage to the majority of other components on the MCU and the partly system-critical power configuration, the functional verification of this module is of particular importance. Sending for example the CPU-subsystem into a power saving sleep mode, but the wakeup detection is not properly working, the system will never return into a safe working state, without an external power reset. To give another example, wrong clock settings can massively affect the responsiveness and power consumption. For the productive application development library the multitude of settings were composed in the fixed set of power modes and the access abstracted by the HAL. Hence, a slight interface to switch the power states, switch clock and configure the wakeup sources is provided. For the verification, all possible combinations of settings must be applicable, even when a critical system state can be entered. This can occasionally be requested by a verification engineer, to monitor the system behaviour in RTL simulation regarding worst case scenario estimations. To avoid the risk of the system to go into an unwanted state, the knowledge about the states can be quite helpful in accomplishing this task. To check the state and have traceability when configurations are changed within the PMU, a multitude of status flags allow the transparency at system-level to check whether the configured settings are taken on, and the expected behaviour is entered.

The TRX-IF provides the communication to the TDA 5340 transceiver module by an internal SPI connection. The transceiver itself has around 260 hardware registers to meet all different kind of software and hardware requirements. However the tasks of the TRX-IF are just to generate the correct wave forms on the communication interface to realize a proper configuration and data connection to the transceiver. By the integration of the TDA5340 module as hard macro into the SoC design, the functional correctness is assumed, since it has reached a productive state in the development cycle, was often implemented and well-tested. The communication and configuration to the module and the interaction within in the SoC is still challenging. The TDA5340 therefore offers multiple access modes to write and read its registers, with a strict specification on the signal wave

forms. A major issue in the digital design wiring the transceiver or the SPI interfaces would result in a useless component on chip. Since the wireless communication is a key element of the wireless sensor SoC, a defect results in an unusable device. This justifies the extensive verification efforts on this module. To apply an extensive verification, all configurations, even those the transceiver would not expect, are driven on the SPI to expose a potential design error. For the productive application API a single default configuration can be sufficient, reducing the application interface to a minimal of send and receive requests. Hence the interesting aspect on this module is this extreme difference in granularity. As a consequence, significant differences in results are expected.

4. Implementation

4.1 HAL Implementation in C

The proposed hardware abstraction layer is implemented in C and checked against the MISRA-C standard to follow best practices with a safer language subset to enhance code security, safety and reliability. As described in chapter 3 the HAL consists of three sublayers, each with the appropriate functionality and abstractness. The tasks of the lowest layer (Layer 0: Register Definitions) is to define the addresses of the registers and group related ones that are placed on the same basis address with ascending offset, to structures. Hence, a first simplification for reuse and portability is achieved.

```
typedef struct
{
  __IO   uint32_t  CTRL;          /*!< Offset: 0x000 Control Register (R/W) */
  __IO   uint32_t  VALUE;         /*!< Offset: 0x004 Current Value Register (R/W) */
} Module_TypeDef;

#define MyModule               ((Module_TypeDef *) ((uint32_t)0x40000000))
```

Furthermore in this layer, masks and values are provided as string identifiers which can be used in the next layer, by using again the *define* compiler macro.

```
#define MASK_MYMODULE_CTRL_ENABLE_CONST_1   0x1
```

On the next layer (Layer1: Basic Drivers), the register-based access is abstracted by simple functions, which contain one or multiple register assignments and can be parametrized to enhance flexible configuration. Since the functions are customized to the particular hardware, this layer is still not-hardware-independent.

```
void _basicdriver_initMyModule(uint32_t initValue)
{
        /* Enable Module */
        MyModule->CTRL = MASK_MYMODULE_CTRL_ENABLE_CONST_1;
        /* Set Initial Value */
        MyModule->VALUE = initValue;
}
```

The top layer (Layer 2: Drivers) provides the application interface; hardware independency is achieved by using a set of standardized functions, which are independent from the accessed registers and the underlying hardware. For that reason the ARM CMSIS driver pack was used, including 11

groups (CAN, Ethernet, I2C, MCI, NAND, Flash, SAI, SPI, Storage, USART, USB) with standardized peripheral driver interfaces including definitions for control and data structures. For the devices not included in this specification (AES, Transceiver, PMU, Sensor Interface, RFID) the ARM driver template was used to define equivalent standardized definitions for internal future reuse.

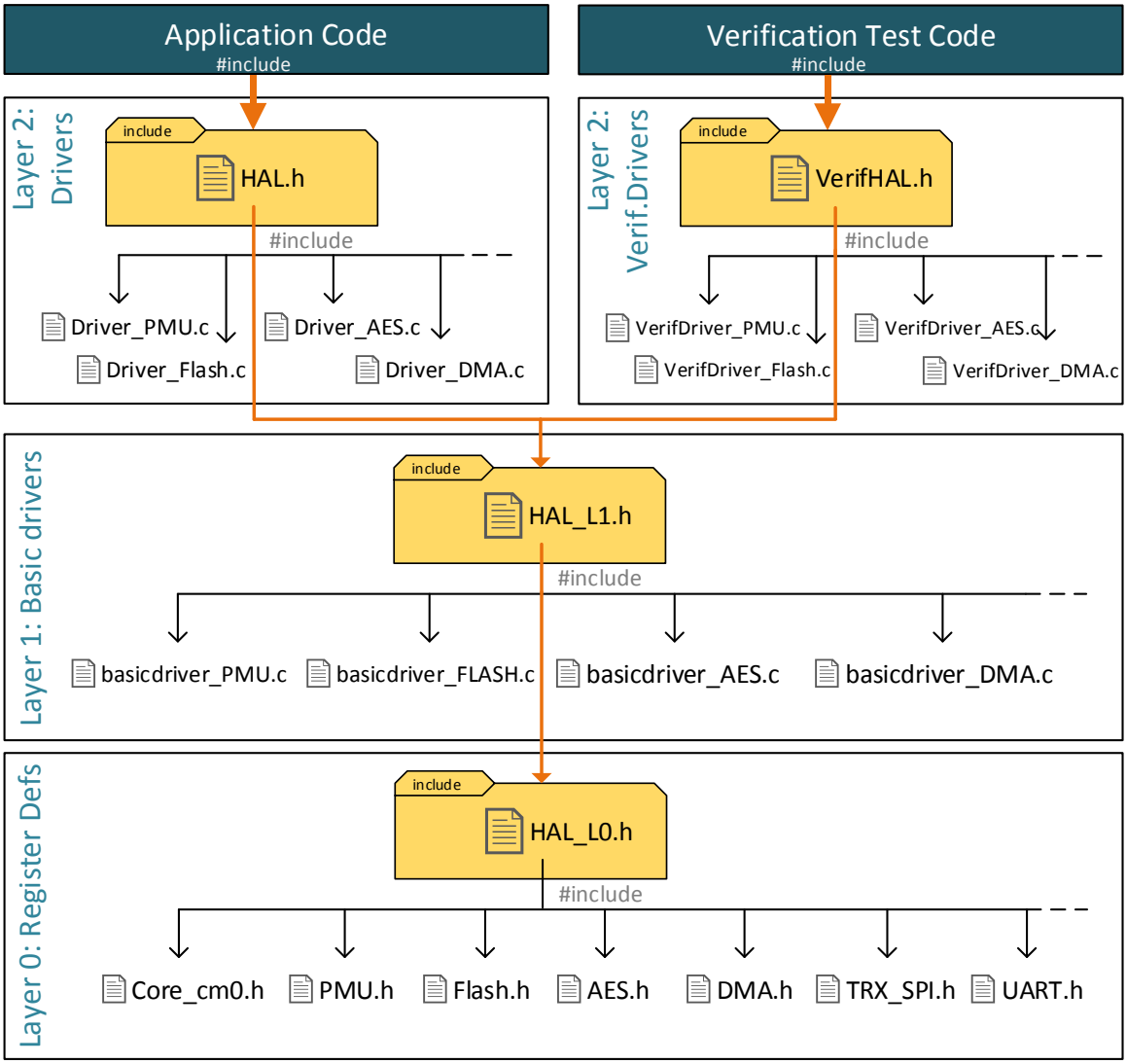

Figure 4.1: File tree of the implemented HAL

In Figure 4.1 the tree structure of the files is presented, showing one header include file per layer, that itself includes all appropriate header and code files. On top, both layer 2 implementations form a variant of the hardware abstraction layer, providing the appropriate interfaces for application and verification test code.

4.2 Development and Simulation Environment

The SoC development project is currently in pre-silicon stage, where the digital modules are implemented by a mix of SystemVerilog, Verilog and VHDL. Primarily they run together with hardware macros and the software within memory behavioural models in RTL simulation. When the pre-silicon development and simulation stage is almost finished, near to the tape-out gate level simulation is also applied, to check timing limitations and again functional correctness. By the expended simulation time against RTL, this level of detail is inappropriate for test/debug simulation during development. As a consequence, RTL simulation was used within software development.

The C and Assembler implemented code was built using the *GNU ARM Embedded Toolchain*, an open source toolkit including tools for compiling, linking, operating of binaries and a lightweight C standard library for embedded systems. To use in addition the comfort of a GUI based development environment, the *ARM DS-5 Development Studio* was used in the free Community Edition by integrating this toolchain. It is also available in licensed professional versions with the ARM compiler toolchain included. The *GNU ARM embedded toolchain* as well as the *ARM toolchain* come with examples for start-up code, including the vector table definition and initialization routines, and linker scripts to position the code on prerequisite addresses. For example the vector table is positioned on hardware address 0x0 where Cortex-M0 expects it to be.

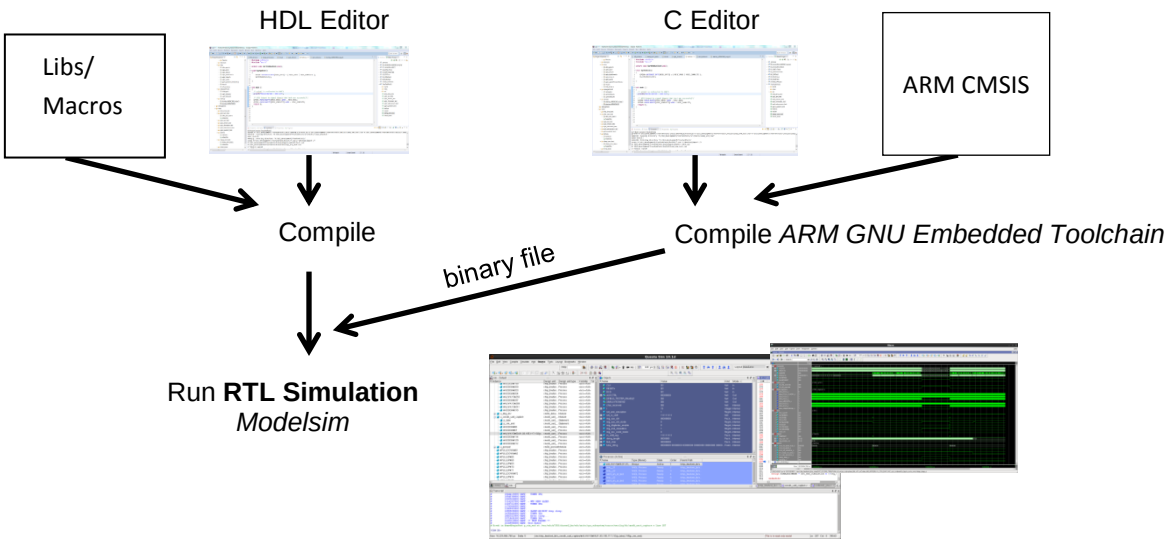

Figure 4.2: Development environment, software binaries are included within the RTL simulation

The built binaries where included by the HDL memory behavioural models, see Figure 4.2; for example Verilog provides these feature with the built-in commands *$readmemh* and *$readmemb* (hex and bin representation) to store the content of a file to a memory array on simulations start-up. To debug the hardware and software system, a simple test-bench was used, capturing the UART output and write to the simulator console using the Verilog $display command. In addition the GPIO pins were stimulated and monitored to model in/outputs.

4.3 Tools for Metric Evaluation

In the field of static code analysis a multitude of tools are available, from simple applications to evaluate a particular metric, up to powerful code analysis toolkits. For the evaluation of the HAL and the proposed metrics, several tools were used:

- CCCC (C and C++ Code Counter) [17] is a free command line tool to measure different metrics on C, C++ and JAVA source code, including *Lines of Code* related measurements and *McCabe's Cyclomatic Complexity*. This program was in the first stage used to calculate the SLOC and the McCabeCC.
- SourceMonitor 3.5 [18] is freeware to analyse C++, C, C+, VB.Net, Java, Delphi, Visual-Basic or HTML code. It was used in addition to CCCC to check against on SLOC and McCabeCC.
- HalsteadMetrics [19] is a simple metric analyser for the same-named metrics. It calculates *Number of Operators* and *Operands*, *Number of Unique Operators* and *Operands*, as well as the actual metrics: *Program Vocabulary, Length, Volume, Effort, Difficulty* and *Time*. An export generation in HTML or PDF format is possible.
- GNU Binary Utilities - size [20]: displays the section sizes of an object or archive file. It is part of the *GNU Binutils*, including the GNU linker and assembler, and miscellaneous helper tools. This program was used to determine the program memory of the HAL implementations.
- ModelSim is a HDL simulation environment by Mentor Graphics [21] and was used to perform the pre-silicon RTL and Gate-level simulations of the digital SoC design. In chapter 4.2 the simulation and development environment was explained: The software is executed within the RTL simulation, which allowed a profound evaluation of the dynamic metrics. Data memory was determined by examining the content of the RAM and the clock cycles for execution in the wave prints by measuring the rising clock edges between two curser values when the particular analysed function was called, executed and called-back.

5. Benchmarking and Results

The proposed HAL for the developed SoC was designed in combination for both of the two use cases of functional hardware verification and as application library. In the previous chapters the approach of reusing the lower two layers - register definitions and basic driver functions - for both scenarios and a custom layer 2 implementation for each was presented. To evaluate the two resulting software variations, a profound review on state-of-the-art software measurements and the check for applicability on embedded software, in particular hardware abstraction firmware resulted in a set of proposed conclusive metrics. The following benchmarking was just applied on a part of the entire HAL software, to discuss the results in detail. Two example hardware modules that are custom developments, and therefore have a higher demand for functional verification, were chosen. The first one is the power management unit of the chip and is a critical component of the system, by providing functional access to power down the core components of the system , thus including the risk of crossing a point of no return where the system can be locked until a manual reset. The second component is the transceiver interface, serving the communication to transmit and receive functionality of the wireless SoC. As a key module, functional correctness had major priority in the project verification plan.

The intro of this chapter is straight followed with the detailed result of the several metrics, analysed with different 3rd party tools on the internal and external software attributes. The setup and limitations are important to review, to understand the expressiveness and coverage of the measurements. Then a discussion on the meaning of these values, with comparing to limits and common results will give a detailed analysis of the software characteristics. In the second part of this chapter these results are jointly considered to discuss the proposed HAL concept in general, the advantages and disadvantages, as well as possible improvements. In addition yet untreated aspects on the HAL will be introduced, such as the question on security aspects, by reusing low-access software und might open security holes for system threats. When a sensor application is embedded in a safety critical system, the harm of resources or human life is not acceptable. The disquisition on security and safety considerations is finally followed by the discussion on extensibility of the proposed design and growth of knowledge for HAL designs in combination with CMSIS.

Table 5-1: Evaluation results of the proposed metrics on the two example modules

	Power Management Unit (PMU)		Transceiver Interface (TRX-IF)	
Metric	**Verification**	**Productive**	**Verification**	**Productive**
Total Source Lines of Code (TSLOC)	1190	1075	533	460
▪ SLOC - Layer 2	220	105	98	25
▪ SLOC - Layer 1	371		335	
▪ SLOC - Layer 0	599		100	
Number of Methods (NOM) [2]	11	5	7	3
Number of Parameters (NOP) [2]	14	6	17	6
SLOC Reuse	81,5%	90,2%	81,6%	94,6%
McCabe Cyclomatic Complexity (McCabeCC) [2]	Max.12, Avg. 4,64	Max. 6, Avg. 3,00	Max. 8, Avg. 3,71	Max. 4, Avg. 2,00
Halstead Complexity Metrics [2]				
▪ Number of Operators (N1)	456	134	186	47
▪ Number of Operands (N2)	890	340	507	133
▪ Number of Unique Operators (n1)	14	11	16	12
▪ Number of Unique Operands (n2)	111	52	43	24
▪ Vocabulary Size (n)	125	63	59	36
▪ Program Length (N)	1346	474	693	180
▪ Program Volume (V)	9375,95	2833,23	4076,67	930,59
▪ Difficulty Level (D)	56,13	35,96	94,33	33,25
▪ Program Level (L)	0,018	0,028	0,011	0,030
▪ Implementation Effort (E)	526 235,51	101 887,33	384 534,42	30 942,00
▪ Implementation Time (T)	29 235,31	5660,41	21 363,02	1719,00
▪ Number of Delivered Bugs (B)	0,32	0,11	0,26	0,05

[2] *Evaluated on HAL Layer 2 source code only.*

Program Memory [Bytes]	2556	1708	1192	334
Use-Cases to evaluate Clock Cycles and Data Memory				
a) Set WakeupTimer, goto Sleep				
▪ Clock Cycles	1260	829	*n/a*	*n/a*
▪ Data Memory [Bytes]	56	48	*n/a*	*n/a*
b) Clock Switch				
▪ Clock Cycles	510	360	*n/a*	*n/a*
▪ Data Memory [Bytes]	48	44	*n/a*	*n/a*
c) Write 10 bytes of data				
▪ Clock Cycles	*n/a*	*n/a*	243	60
▪ Data Memory [Bytes]	*n/a*	*n/a*	68	48
d) Read 10 bytes of data				
▪ Clock Cycles	*n/a*	*n/a*	289	72
▪ Data Memory [Bytes]	*n/a*	*n/a*	64	48

Metric Evaluation Results

To finally answer the questions on the hardware abstraction design about the meaningfulness of the approach on verification and productive library combination, the quantification of software attributes shall found the base for the following discussion. This examination on the two variants per example was conducted with the composed suitable metrics, listed in Table 3-5. This set persists of measurements for internal attributes of software (source code/binaries) and external attributes, including the interaction with the environment in software execution. In Table 5-1 the final results of the HAL evaluation are presented.

- *Total Source Lines of Code (TSLOC)* measures the size of code by counting all lines except for commented and blank ones. In the results it is divided up into the single SLOC values for each layer, to gain information about the amount of code distribution. Layer 0 consists of all register definitions and masks, layer 1 of basic functionality to compose related register assignments to functions. These two are the same implementation for both use cases, whereas layer 2 is adapted for the particular application layer. However, the presented SLOC values of all layers in this table just represent the functionality for the specific hardware module, and not the HAL in total. A look on the different values between PMU and TRX-IF is justified by the overall higher functionality served by the PMU. The connection to all memories and peripheral domains, the exhaustive control of system states and access to distributed state information make this component to a functional core block, but also in terms of risk. Therefore on the functional verification tasks, the highest possible coverage is favoured, by

a priori, in-between and post-hoc checking of flags, to monitor state transitions. This huge number of flags and control options reflects in the high SLOC value of layer 0. Layer 1 combines register access to build-up a slighter interface to Layer 2. The difference on the top layer, between verification and productive variant is around 48% because of the neglected flag and state checks for productive operation. The TRX-IF is less versatile compared to the PMU, however providing a multitude of configurations and status registers. The higher SLOC of layer 1 compared to layer 0 is justified by the combination possibility of the huge number of configurations to customize the output signal. In the productive variant, there is only one configuration used, whereas the verification firmware tries to provide this extensive wideness for testing. This results at the layer 2 variants in the relation of SLOC by a factor of 4.

- *Number of Methods (NOM)* counts the number of methods in a module, here applied on layer 2 only, to further characterize the interface to the application layer. Based on the statements given to lines of code, the results of NOM can be explained in analogy. The difference between the verification and productive variants are justifiable by the granularity of the application interface. The verification implementation tries to obtain the wideness of configurability for highest coverage on functional verification testing, whereas the productive library shall provide a slight, simple and default configured interface. This measurement exclusively does not allow a comprehensive analysis. It is strongly dependent on the implementation, and is best combined with *Number of Parameters* and complexity measurements to fully examine the functional division and internal structure of methods.
- *Number of Paramters (NOP)* counts the module's parameters of a module. Obviously, again the verification implementations have a greater number. In combination with the *Number of Methods (NOM)* a derivation on the additional configurability against productive implementations can be made. For the PMU and TRX-IF the proportion is about 2. However, if in C code for example structures in combination with *typedef* are used to encapsulate related data into one variable, the expressiveness of this measurement has to be verified. For the proposed examples, nested data encapsulations are mostly not selected; standard datatypes or enumeration parameters were used to keep the interfaces simple.
- *SLOC Reuse* is a custom, within this thesis proposed number to reflect the reuse of layer 0 and 1 source code for both use case variants. In chapter 3.3 these numbers were defined by formula 22 and 23 to calculate the ratio of the sum of layer 0 and layer 1 SLOC to total SLOC. Taking a look at the results in Table 5-1, the numbers obviously create the impression of a very high scale, especially at the maximum result of 94.6% at the TRX-IF HAL. Important at this point is to consider the outcome from SLOC review in chapter 2.5: *Lines of Code* is just one measurement for software size, but it does not cover all aspects of size. There is no information about the content and functionality of the lines, a highly complex calculation is counted the same as a series of no-operation instructions. However, this calculation shows that a huge amount of source code can be shared between the two use cases and the common intersection of programming tasks obviously exists. The linker will nonetheless not add parts of the lower layers to the productive layer particularly, if they are not used from the higher layer. In the second part of this chapter, this numbers will be taken up again, as indicator for the connections between verification and productive purpose.

- *McCabe Cyclomatic Complexity (McCabeCC)* measures the alternatives in the program flow by counting a variable for each branch that occurs in the code including if-else, switch or loops. The higher complexity values for the verification use-case are based on mainly two reasons.
  (i) The processing of a higher number of parameters, with reference to the results of NOP. Not every (e.g. data value) but the configuration parameters (e.g. choice of one clock out of four) create an additional branch in the program flow, and therewith increase the cyclomatic number.
  (ii) Checking the a-priori / in-between / post-hoc status flags in the verification purpose, creating if/else branches and again the cyclomatic number

  McCabeCC is popular metric to gain knowledge about the structure of code, in comparison to just size and population metrics. The limit proposed by McCabe is a value of 10 [McC83] to indicate that splitting up the function would be useful. Applied to the results the functional division in the verification purpose source code of PMU layer 2 is slightly across this line and should be revised. The other values are in line with the limitation of McCabe.

- *Halstead Complexity Metrics* are a set of eight calculations, based on the assumption that code can be seen as a series of operators and operands. The detailed definitions according to C were given in chapter 2.5 and the calculations by formula 10 to formula 17. They are all based on the count of operators and operands in total and distinct.
  Number of Operators (N1) and Operands (N2) are another expression of the code size, and therewith obviously correlate in sum as *Program Length (N)* with the SLOC results.

  The values of *Number of Unique Operators (n1) and Unique Operands (n2)* describe the diversity of code. The results do not differ that much between the two examples and use-cases, and can be explained by the similar composition of code. On layer 1 the register definitions of layer 0 are assigned ("=") with defined masks and values. Therefore, masking can be applied by AND-conjunction ("&"), multiple bit assignments with OR-conjunctions ("|") accompanied by negations ("~") and shift operators ("<<", ">>"). On layer 2 and layer 1 Boolean statements ("= =", ">" "<", etc.) are used to control for example the if/else branches. Consequently, with the difference in functionality per se, the operators are related to the overall task of serving HAL functionality. By analogy, the quantification of operands can be explained.

  Program Volume (V) measures the information content of a source code, can be used as size code measurement and is less sensitive against code structure than SLOC. The proposed limits for a file are at least 100 and most 8000, where a greater number indicates that maybe the file ought to be split up. Hence, in accordance to McCabe complexity results, not just the functions, but also the file functional division should be reconsidered on the PMU verification implementation. The other results are within the limits and obviously correlating to the previous results that yet confirmed the higher code size of the verification versions.

  The *Difficulty Level (D)* is proportional to *Number of Unique Operators* and to the ratio of *Number of Operands* to *Number of Unique Operands*. Using more distinct operators or reusing same operands multiply in code increases the *Difficulty Level*, whereas new operands

decrease it. The *Program Level (L)* is the inverse of it, where a lower level indicates the propensity for error occurrence. Very interesting in these two numbers compared to the previous results is the fact, that the use case PMU/Verification with the highest cyclomatic complexity (max.12/avg. 4.64) and SLOC/Layer 2 (220), is not the implementation with highest difficulty level with a result of 56.13 (and therewith expected lowest program level). TRX-IF/Verification has a much greater result in the difficulty level of 94.33 by a cyclomatic complexity of max.8/avg.3.71 and SLOC/Layer2 of 98. To find the explanation to this behaviour, a review on formula (13), definition of the *Difficulty Level* is necessary.
For the first term, the result of Number of Unique Operators is almost the same for both examples (PMU: 14, TRX-IF: 16), already justified above in the explanations to this metric. The second term is the proportion of total operands to unique operands, and results for the PMU in 8.02 and for TRX-IF in 11.79 and is dominating the result. This difference can be explained by the broad configurability of the PMU (great number of unique operands) in comparison to the less options but huge combination possibility of the TRX-IF (multiple reuse of operands).

The *Implementation Effort (E)* is the direct proportion of *Volume* to *Difficulty Level* of software, and in the results again dominated by PMU/Verification. While the difficulty is much greater for the TRX-IF/Verification implementation, the dominance of the high value in program volume at PMU/Verification results in a total higher scale of implementation effort. After the revealing results on the *Difficulty Level*, the comparison to TRX-IF/Verification by examining the ratio of SLOC to *Implementation Effort* can be quite interesting. The PMU implementation resulted in a SLOC/Layer 2 of 220 to implementation effort of 526235.5, which is now set into proportion resulting in 2391.98 calculated by formula (24). It shall reveal the average effort needed for implementation per source code line by simply calculating

$$\text{Effort per Source Line of Code} = \frac{\text{Implementation Effort}}{\text{SLOC}} \tag{24}$$

TRX-IF/Verification with a SLOC/Layer2 of 98 and the *Implementation Effort* of 384534.42 results in an effort per line value of 3923.82, which is a 64% greater value compared to the PMU software for verification. Hence, it makes sense to relate the number of *Implementation Effort* to a size measurement to examine the meaningfulness of the result.

The *Implementation Time (T)* approximation in seconds is the division of the *Implementation Effort* with the constant value 18. The high result values for the Layer 2 are converted for easier understanding: 8h7min for PMU/Verification, 1h34min for PMU/Productive, 5h56min for TRX-IF/Verification and 29min for TRX/Productive. The difference within the implementations for each example module is considerable, with an about 5 times higher time amount for the verification compared to productive variant of the PMU in analogy a factor of 12 for the TRX-IF example. These results can be explained similar to the reflections on *Implementation Effort*, by creating a relation to software size. While these numbers only present estimation for the layer 2 implementations, presumably the overall total implementation effort is primarily in the a-priori HAL design concept, and the implementations of layer 0 and layer1.

At last, *Number of Delivered Bugs (B)* provides estimation for the number of errors in the software. The recommended threshold is 2 for a file, which is far away from the small numbers calculated for the layer 2 implementations. However the results values shall be interpreted as a lower bound for expected errors in code, the actual error rate is of course dependent on various factors, such as the experience of the programmer.

- *Program Memory*, in bytes, is the permanent memory footprint for the program in ROM. It consists of the instructions and read-only data, including addresses to registers or the vector table that contains the jump addresses to the interrupt handlers. Obviously, the results are again greater for the verification implementations, but do not have the strong correlation to SLOC that might be expected at first view. When the source code files are compiled and linked together to a program, only actually referenced code is added. Even though, layer 0 and 1 are shared between verification and productive use-case, all parts of code will not be used by the productive library. The evaluation of re-linked code resources is much more difficult than the source code analyzation, by the resulting binary file of the linking process which then contains the application layer code as well, and is cleaned up from comments. Therefore the mapping to the origin source code is tricky and prone to error. This kind of analysis was neglected in this thesis, but might be an interesting additional source for examination. The particular limitations on *Program Memory* are dependent on the resources available at the particular SoC; respectively for the presented microcontroller, with an 8kB ROM for the bootloader application and a 64kB flash memory for the actual application the values are within the limitations.

- *Clock Cycles* are counted to measure the execution duration of software on a specific target system. Dividing the result with the clock frequency actually used (in the tests 6MHz), results in execution time. *Clock Cycles* makes comparability on similar hardware systems much easier on a common level, by subtracting out the actual frequency used of the time measurements. *Data Memory*, in bytes quantifies the volatile memory allocation that contains mainly variables and register stacking on function calls.
  During the realization of this thesis, a substantial problem of evaluating both of these metrics on the HAL variants was revealed: since verification and productive use-cases provides an intersecting but also different set of functionality to the application layer, and a HAL is not stand-alone software at all, a direct evaluation of the execution is hardly possible. To quantify these interesting measurement anyway, significant example functionality was chosen that are supported by both HAL variants, but implemented in a different way:

  a) PMU – set WakeupTimer, goto Sleep: this example describes one of the key tasks of the SoC's power management, by sending the system into a power-saving sleep mode. First, to make sure that a later return from this state is possible, a wake-up-timer is configured within the PMU. Subsequently, the target power modus *DeepSleep* is selected and the request for execution set.
     For the productive variant, this procedure is straight forward implemented as described, whereas the verification implementation is checking the status flags of the timer a priori and post hoc to the configuration. Same is done for the power mode selection, before the system actually is sent to the low power configuration. This re-

sults in an increase of 51% of clock cycles for this overhead checks. The data memory is almost at the same level, by the main dominance of the register stacking by function calls. Since both variants have the same software structure together in principle and variable use is rare, the results are nearby. The results on the static Halstead Metrics included a discussion about the similarities of the used operators and operands. This can be directly associated to these results, where same operators are resulting in a similar data memory behavior. A note on the remarkable high values of clock cycles is necessary here: the high amount of branches inside the functions, and the function calls between the layers are dominating parts.

b) PMU – Clock Switch: The presented SoC is equipped with two oscillators, providing a 24 MHz and a 64 kHz Clock. In addition the 13MHz Clock of the RFID controller can be used as system clock as well or an external input connected to a GPIO pin. To enhance flexibility, from these clocks can be further subdivided with a configurable clock divider. To switch from one clock to another, a control and status register in the PMU allow the enable of the new clock source, the configuration of the clock selection and the disable of the beforehand used clock. Again the procedure is implemented in these steps in the productive use case, whereas on verification the status registers are subsequently monitored. The justification on data memory consumption is in analogy to a).

c) TRX-IF – Write 10 bytes of data: The transceiver to send and receive data to/from other devices or base-stations is connected via an SPI interface to the core AHB/APB bus system. The proper generation of the signal forms expected by the transceiver is quite important for correct data transmission. Therefore the write of 10 bytes of data is started with the initialization including setting the baud-rate, duplex mode, packet size, polarization, edge for writing etc. and followed by writing the values subsequently into the send buffer. On verification, after setting the configuration each setting is checked against the status flags and the subsequently write operations are checked for errors. The accessed register values at the transceiver can be checked in the tests by reading them for comparing against the written values. The significant difference of the clock cycles for execution can be in addition explained by the high number of branches in the verification layer 2 implementation. All different settings are possible, whereas the production firmware can be used preconfigured for the transceiver expectation data output and interface configuration.

d) TRX-IF – Read 10 bytes of data: Respectively to the procedure illustration in c) for transmission of packets to the transceiver, the read initialization and receive is realized in analogy. After the settings configuration, subsequently the packets are read from the receive buffer. On the verification use-case the procedure is extended by extensive status checks. Again the high number of branches in the layer 2 functions justifies the great number of clock cycles needed for executing the verification implementation.

Within this comprehensive analysis on the example of PMU and TRX-IF, each implemented in the two use-case variants for verification and application API, the obviously expected larger implementation for verification was confirmed. However some unexpected results were examined, but could be justified by the particular program structures. With these single and partly already connected metric results, a comprehensive further analysis is profound.

HAL Design Analysis and Security Aspects

Started from a comprehensive literature research on software metrics in general, a set of suitable metrics for HAL analysis was proposed in chapter 3.3. It consists of metrics for the internal attributes regarding the software itself, and external attributes that are related to the hardware environment. Metrics on the internal attributes obviously show certain correlation, by evaluating similar and related objectives of code. However, not a single metric actually allows the thorough analysis; the combination of several is the key for substantiated assertions. For example the results in lines of code revealed that size estimation can be applied; however there is now information about a reasonable division of functions between and within the HAL sublayers.

Hardware abstraction code is bare-metal software, in other words run directly on the microprocessor resources, without an operating system to manage the utilization or provide any abstracted functionalities. Actually a HAL is part of operating systems to provide uncoupling from the particular hardware used. This aspect made *CPU Utilization* not applicable, and was replaced by number of *Clock Cycles* to measure the execution time by dividing it with the clock frequency. One problem in this context that occurred during concept and evaluation was the evolving of the awareness on the different functionality served by the HAL use-cases. Consequently, a direct total comparison is not feasible at all, furthered by the fact that a HAL is not stand-alone software and needs in addition application software to be called. Besides *Clock Cycles*, the determination of *Program Memory* that is obviously dependent on the particular program flow executed is affected too. Nevertheless, to allow a comparison for each example two significant application scenarios were chosen and evaluated.

Besides the details about implementation, other aspects such as safety and security exist in the context of a wireless sensor application. A particular question that occurred during analysis was: By the reuse of a very low-level access, broad configurable code, that was obviously developed for verification; is there any security considerations, using them in application code for 3rd party developers? To elaborate the answer, first the fact of bare-metal software has again to be mentioned. There is operating system or any other software below, that provides an access mechanism, and thus in principle every register can be read or written. Of course particular hardware construct can be used to lock various registers, but none of such mechanisms is realized in the presented SoC. The only way of obscuring registers is by not providing the programmers model, i.e. not defining them and the appropriate masks in layer 0. However, 3rd party developer engineers would still have a change by ARM debug and traces to scan the bus memory allocation and find obscured registers. At this point the question of the reasonableness of such a lock-out is appropriate; there is no interest of locking them out anyway. The security to upper laying, additional software has to be provided by an operating system, if necessary and can thus be neglected in this context on the bare-metal stage of HAL.

Code Reuse between Verification and Productive Use-Case

Based on the metric results and the general discussion about the proposed HAL design, a discussion of the proposed approach on combining the interests of developing firmware for the pre-silicon verification, as well as defined reusing parts of code for the application API can be now carried out. The approach compromised two lower layers as common basis for both use-cases and layer 2 implementations custom to meet the designated requirements. This particular code-reuse of the basis source code is subsequently discussed in this chapter to allow a profound conclusion in the last chapter of this thesis.

The common layer 0 and layer 1 implementation was approached by the identification of uniform procedures that are needed for both scenarios. The abstraction from bit assignments in a multitude of registers is achieved by functional encapsulation in a set of parametrized C-functions. Thereby the fundamental task is how to divide the set of register assignments to a reasonable set of functions. The degrees of freedom are the number of methods and the number of parameters and the boundaries set by program-, data memory and execution time. Two possible extreme configurations are possible:

(i) Implement for every register bit setting a single function without parameters. This results roughly in the number of methods equal to the number of registers times the width of the registers (in bit) times 2 (value 0/1). This solution would allocate an enormous amount of program memory, but would be extremely fast by the absence of branches in code (switch/if/else). Anyhow, there is no added value for abstraction, and the additional code would result in a lower programming level, and higher expectable amount of errors.

(ii) Implement just one function, and pass all possible configurations as parameters. This solution results in the minimal program memory solution, but the extensive branching for all parameters would result in the worst-case number of execution cycles. Furthermore the parameters have to be stored on stack and thus, increase the RAM footprint.

Apart from the not-applicability of both of the extremes described here, the best solution is lying in a compromise in-between. The optimum is a trade-of the determining variables and the delimiters that are obviously bound to the particular SoC implementation specifications. The decisions designed in this stage, can influence the total system behavior, for example reaction time on an external interrupt or the memory utilization, which are important practical delimiters (hopefully) known from a SoC requirements concept. Anyhow the task on layer 1 design is getting much more difficult when it is extended with a second set of requirements, to fulfill the proposed combination of verification and application API development. In Table 5-1 the metrics Number of Methods and Number of Parameters are listed for the four implementations. However they include no information about the reasonableness of this implementation decision. McCabeCC presents complexity of functions, and therefore shows within an upper and lower limit if the division was useful. A lack in the design can be seen in the high branched functions in PMU/Verification, and should be revised again. On the TRX-IF, the different implementation interests in terms of “best solution” for the division problem can be discussed best. The verification API has to provide the setting of all combinations of configurations, while the productive API uses as the simplest implementation, just one set of them. Therefore on the

extremum examples, (i) would fit best, whereas on the verification use case, (ii) is probably more expedient. To overcome a satisfying trade-off, the awareness of this optimization problem hast to be in mind in the earliest HAL design stages, by including all delimiters, verify the implemented HAL with the stated metrics and update the HAL division if inefficiency is identified. A prioritization of the productive implementation seems reasonable, where some limitations are having greater priority (performance/power consumption). If there is just a small set of applied configurations, the implementation with multiple, less parametrized functions is beneficial. When both use-cases need the comprehensive configuration settings, a trade-off with multiple parameters per function and not exorbitant number of methods will both hold program memory allocation and execution time in favorable limits.

Another aspect on the combined development is the usefulness to reuse the broad abstracted register access for the application library development as well, when only a subset of the actual provided functionality is used anyhow. If the additional code would create an enormous overhead, a waste of resources might make the SoC operation inefficient. The economically worst-case might be a favoritism of competitive products because of this major performance lack. Nevertheless, this worry is arbitrary, because the compiler and linker are only adding code to the binaries that is actually used. With additional code optimization techniques the sub code included into the application deployment can be potentially further shrunk.

Power Management Verification

A central element of the SoC chip design is the low level requirement for running the microcontroller on battery or energy harvesting for wireless applications. Longest lifetime must be ensured by a high coverage of clock and power gating techniques. To realize this, the PMU is connected all across the microcontroller system to control the energy behavior. Hence, the importance of the functional correctness was stated, and the decision for the example evaluation justified. If the functional correctness of the hardware is of this importance, is there any loss and risk in using a high-level abstraction to implement verification tests? To answer this importance question, already discussed metrics results are examined in combination. Halstead's metrics showed that with a particular amount of written code a lower boundary for expected bugs within the code can be given. Consequently, higher amount of code results in a higher probability of adding errors and the risk of bringing the system in a state-of-no-return, where a power down mode is entered without an appropriate wake-up source to reactivate it again. On the other hand, working on lowest level with register definitions and assignments reduces the code readability. In addition Halstead's difficulty metric would probably reveal the problem of multiply reusing same registers and their assignments on different position in code, which can be interpreted in a higher error prone of the source code too. Obviously, this aspect gives a great opportunity for additional examinations to find the perfect trade-off in abstraction and lowest error probabilities.

6. Conclusion

The ongoing rising system complexity and therewith verification effort motivated the feasibility exploration of combining a hardware abstraction API development for the purpose of pre-silicon verification and productive deployment. The goal of a contribution to shorten development cycles was started with a comprehensive literature research on state-of-the-art functional verification, ARM architecture, HAL-design, power-management and software metrics. Thereupon the concept of a three-layer HAL was presented, with the lower two layers in common use and the highest including the interface to the application layer as custom implementation to meet the requirements of the particular use-case. To evaluate the software design and the code reuse a set of proposed suitable metrics was examined on the HAL implementations of two example hardware modules. With these results, the feasibility of connecting the related software implementation tasks could be clearly shown. On static code analysis, a reuse of about 80% of source code from pre-silicon verification can be reused in the productive library development, including the layer 0 register definitions and layer 1 basic driver functions. However, the advantages are accompanied by a profound extended a-priori software design, to ensure a justifiable trade-off between generalization to meet all requirements and related performance losses. This disadvantage could get heavy without proper awareness during HAL design, and can potentially result in the worst-case scenario of be wide apart of the performance provided by other competitive microcontroller vendors.
By the topics discussed, advanced studies would be reasonable to answer several open questions including

- Total degree of potential reduction of development effort, by examining the before mentioned increase in a-priori design concept but the proposed decrease of implementation time to deploy a productive library for 3rd party developers
- Examine portability, to quantify reuse of high-level tests and parts of the HAL over different ARM-based SoC development projects
- Error susceptibility of tests using low-level register access versus a high-level hardware abstraction library. Register access is prone to error because of the multiple reuses of same assignments whereas the size of HAL code includes risk of software errors too.

Nevertheless, within this thesis an attempt was provided to make a research foray exploring the potential in identifying, describing and evaluating intersecting development tasks. Mentioned at the outset, the ongoing increase in system complexity and verification effort will be definitely a major driver for future research on this and related promising approaches to improve SoC development.

Literature

[And05] Andrews J.R.: Embedded System Verification: An Introduction, In Co-verification of Hardware and Software for ARM SoC Design. Burlington: Newnes, 2005.

[Bai14] Bailey B.: Software-Driven Verification. Semiconductor Engineering., 2014. http://semiengineering.com/software-driven-verification/, accessed 07.11.2016.

[Ben15] Beningo J.: 10 Tips for designing a Hardware Abstraction Layer (HAL). EDN., 2015. http://www.edn.com/electronics-blogs/embedded-basics/4439613/10-Tips-for-designing-a-HAL, accessed 10.12.2016.

[Bha13] Bhatt R.: The Use of Hardware Abstraction Layers in Automated Calibration Software., NCSL International Workshop and Symposium, Nashville, Tennessee, NCSLI, 2013.

[Chi94] Chidamber S.R. und Kemerer C.F.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20 (6). S. 476-493, 1994.

[Dom09] Domer R.; Gerstlauer A. und Müller W.: Introduction to Hardware-dependent Software design., Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific, Yokohama, Japan, IEEE, pp 290-292, 2009.

[Ebe09] Ebert C. und Salecker J.: Guest Editors' Introduction: Embedded Software Technologies and Trends. IEEE Software, 26, 3, pp 14-18, May-June 2009.

[Eck09] Ecker W.; Müller W. und Dömer R.: Hardware-dependent Software. 1st. Aufl. Springer Netherlands, 2009.

[Fen14] Fenton N. und Bieman J.: Software metrics: a rigorous and practical approach. Third Edition. CRC Press, 2014.

[Fos15] Foster H.D.: Trends in functional verification: A 2014 industry study., 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, pp 1-6, 2015.

[Goe14] Goering R.: Q&A: Moving Towards Use Case and Software-Driven Verification. Cadence., 2014. https://community.cadence.com/cadence_blogs_8/b/ii/archive/2014/11/18/q-amp-a-moving-towards-use-case-and-software-driven-verification, accessed 07.11.2016.

[Gwa06] Gwak T. und Jang Y.: An Empirical Study on SW Metrics for Embedded System. In Software Process Change: International Software Process Workshop and International Workshop on Software Process Simulation and Modeling, SPW/ProSim 2006, Shanghai, China, May 20-21, 2006. Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg. S. 302-313, 2006.

[Hal77] Halstead H.M.: Elements of Software Science. New York: Elsevier Science Inc., 1977.

[Han05] Handziski V.; Polastre J.; Hauer J. et al.: Flexible hardware abstraction for wireless sensor networks., Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on, Istanbul, Turkey, IEEE, pp 145-157, 2005.

[Hel10] Helms J. und Tacha N.: Strategy in practice: Taking a hardware abstraction layer from design to deployment., 2010 IEEE AUTOTESTCON, Orlando, FL, USA, IEEE, pp 1-6, 2010.

[Hun03] Hunsinger F.; Francois S. und Jerraya A.A.: Definition of a systematic method for the generation of software test programs allowing the functional verification of System On Chip (SoC)., In Proceedings of the 4th International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions, Austin, TX, USA, 2003.

[Jer05] Jerraya A.A. und Wolf W.: Hardware/software interface codesign for embedded systems. Computer, 38, 2, pp 63-69, February 2005.

[Kar13] Karmann J. und Ecker W.: The semantic of the power intent format UPF: Consistent power modeling from system level to implementation., 2013 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), Karlsruhe, Germany, IEEE, pp 45-50, 2013.

[Kau14] Kaur A.; Kaur K. und Kaushal P.: Software maintainability prediction by data mining of software code metrics., 2014 International Conference on Data Mining and Intelligent Computing (ICDMIC), New Delhi, India, IEEE, pp 1-6, 2014.

[Ken06] Kenney J.: Using a processor driven test bench for functional verification of embedded SoCs. Mentor Graphics., 2006. http://www.embedded.com/design/configurable-systems/4006718/Using-a-processor-driven-test-bench-for-functional-verification-of-embedded-SoCs, accessed 08.11.2016.

[Kom06] Komarnitzky A.; Ben-Ezer N. und Lyubinsky E.: Unique Approach to Verification of Complex SoC Designs. Avnet ASIC Israel Ltd., 2006. http://www.designreuse.com/articles/12496/uniqueapproach-to-verification-of-complex-soc-designs.html, accessed 7.11.2016.

[Kwa08] Cho K.; Kim J.; Jung E. et al.: Reusable platform design methodology for SoC integration and verification., SoC Design Conference, 2008. ISOCC '08. International, Busan, South Korea, IEEE, 2008.

[Lin10] Lins T. und Barros E.: The development of a hardware abstraction layer generator for system-on-chip functional verification., Programmable Logic Conference (SPL), 2010 VI Southern, Ipojuca, Brazil, IEEE, pp 41-46, 2010.

[Lus16] Luszczak P.: Processor Driven Verification—Use it for More Than Just Sign-off. Mentor Graphics.. https://www.mentor.com/products/fv/verificationhorizons/processor-driven-verification, accessed 07.11.2016.

[Mac16] Macko D.; Jelemenská K. und Čičák P.: Early-stage verification of power-management specification in low-power systems design., 2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Kosice, Slovakia, IEEE, pp 1-6, 2016.

[Mar02] Martin R.C.: Agile Software Development, Principles, Patterns, and Practices 1st Edition. Person, 2002.

[Mba12] Mbarek O.; Pegatoquet A. und Auguin M.: Using unified power format standard concepts for power-aware design and verification of systems-onchip at transaction level. IET Circuits, Devices & Systems, 6, 5, pp 287-296, September 2012.

[McC83] McCabe T.: Structured Testing. IEEE Computer Society Press, 1983.

[Oli08] Oliveira M.F.S.; Redin R.M.; Carro L. et al.: Software Quality Metrics and their Impact on Embedded Software., 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software, Budapest, Hungary, IEEE, pp 68-77, 2008.

[Pan10] Panda P.R.; Shrivastava A.; Silpa B.V.N. et al.: Basic Low Power Digital Design. In Power-efficient System Design. Springer US. S. 11-39, 2010.

[Pin08] Pinto R.R.M.: Power Management Verification – An Evolving Discipline., Microprocessor Test and Verification, 2008. MTV '08. Ninth International Workshop on, Austin, TX, USA, IEEE, 2008.

[Pos03] Pospiech F. und Olsen S.: Embedded software in the SoC world. How HdS helps to face the HW and SW design challenge [hardware dependent software]., Proceedings of the IEEE 2003 Custom Integrated Circuits Conference, San Jose, CA, USA, IEEE, pp 653-658, 2003.

[Pow10] Power L. und Robinson S.: The ARM Cortex-M3 and the convergence of the MCU market. Embedded Labs Ltd., 2010. http://www.embedded.com/design/mcus-processors-and-socs/4027587/PRODUCT-HOW-TO-The-ARM-Cortex-M3-and-the-convergence-of-the-MCU-market, accessed 10.12.2016.

[Spe09] Spencer M. und Tringham A.: Media Alert: ARM Cortex-M3 Processor Momentum Grows. ARM., 2009. https://www.arm.com/about/newsroom/24715.php, accessed 10.12.2016.

[Sun03] Sungjoo Y. und Jerraya A.A.: Introduction to hardware abstraction layers for SoC., 2003 Design, Automation and Test in Europe Conference and Exhibition, Munich, Germany, IEEE, pp 336-337, 2003.

[Vie14] Vieira A.; Faustini P. und Cota É.: Using Software Metrics to Estimate the Impact of Maintenance in the Performance of Embedded Software., 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), Victoria, BC, Canada, IEEE, pp 521-525, 2014.

[Win12] Winterholer M.: Co-debug and Co-verification environment for power management system., System, Software, SoC and Silicon Debug Conference (S4D), 2012, Vienna, IEEE, pp 1-6, 2012.

[Yiu15] Yiu J.: The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors. Newnes, 2015.

[York15] York R.: Embedded segment market update. ARM., 2015. https://www.arm.com/zh/files/event/1_2015_ARM_Embedded_Seminar_Richard_York.pdf, accessed 10.12.2016.

[You11] You D.; Hwang Y.-S.; Ahn Y. et al.: A Test Method for Power Management of SoC-based Microprocessors., 2011 12th International Workshop on Microprocessor Test and Verification, Austin, TX, USA, IEEE, pp 28-31, 2011.

[Zou10] Zoubi Q.; Alsmadi I. und Abul-Huda B.: Study the impact of improving source code on software metrics., 2012 International Conference on Computer, Information and Telecommunication Systems (CITS), Amman, Jordan, IEEE, pp 1-5, 2010.

Internet References

[1] https://standards.ieee.org/findstds/standard/1012-2012.html

[2] https://standards.ieee.org/findstds/standard/15288-2008.html

[3] https://www.arm.com/

[4] https://www.arm.com/products/processors/cortex-m/cortex-m0.php

[5] http://www.ivifoundation.org/

[6] https://standards.ieee.org/develop/project/2415.html

[7] http://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-tm-microcontroller/embedded-software-solutions/infineon-autosar-software/channel.html

[8] https://msdn.microsoft.com/en-us/library/ee504813(v=winembedded.70).aspx

[9] https://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php

[10] https://www.misra.org.uk/MISRAHome/MISRAC2012/tabid/196/Default.aspx

[11] http://www.keil.com/pack/doc/CMSIS

[12] https://standards.ieee.org/findstds/standard/1801-2013.html

[13] http://opencores.org/opencores,wishbone

[14] https://www.iso.org/standard/22749.html

[15] http://www.infineon.com/cms/de/product/rf-and-wireless-control/wireless-control/transceiver/smartlewis-trx-tda-5340/channel.html?channel=db3a3043321e4994013224f2ffd85a92

[16] https://blog.sonarsource.com/

[17] http://cccc.sourceforge.net/

[18] http://www.campwoodsw.com/sourcemonitor.html

[19] https://sourceforge.net/projects/halsteadmetricstool/

[20] https://www.gnu.org/software/binutils/

[21] https://www.mentor.com/products/fv/modelsim/

Image References

Figure 1.1	Consists of contents from Infineon Technologies AG
Figure 2.3	Reproduced with permission from ARM Limited. Copyright (c) ARM Limited
Figure 2.4	Reproduced with permission from ARM Limited. Copyright (c) ARM Limited
Figure 3.1	Consists of contents from Infineon Technologies AG

Declaration

Hereby, I declare, this present work has been drawn up without inadmissible aid of third parties and without usage of other than mentioned resources. Further sources or indirectly appropriated data and concepts are identified by stating the source.

This work has not been presented to other examination procedures, neither nationally, nor in foreign countries, in the same or in a similar form.

Vienna, 26 May 2017

Christian Tauber, BSc