Embedded Automotive - Horizon 2020

Future Technology for Electronic Control Units

Abstract

The future for embedded system applications in the automotive business field commonly known as Electronic Control Unit (ECU) can not be simply derived from the past. Although reusability is a great buzzword in every industry, engineers often know better and design new generations of applications from scratch.

1. Introduction

The future for embedded system applications in the automotive business field commonly known as Electronic Control Unit (ECU) should not be simply derived from the past. Although reusability is a great buzzword in every industry, engineers often know better and design new generations of applications from scratch.

What's so bad with reuse? You may ask yourself, sitting in front of a Personal Computer that can execute code that has been compiled 20 years ago? It's clutter. Pieces added left and right to maintain functionality under whatever circumstances asked for at some point in time. Clutter is the enemy of control over complexity. And complexity has become the No 1 challenge for modern software deployment in the automotive industry.

The benefits of integrated development environments are becoming obvious, most notably thanks to the introduction of Eclipse based work environments. Saving time and energy by being able to access a whole range of useful features without having to interrupt the workflow is the key issue. There are also synergy effects as each individual tool is better utilized when it is integrated to fit the developer's workflow.

After a quick outlook far ahead we have a look at the current embedded development methodology. In 2006 ECU software is often hand-written using the low level programming language C to cope with performance and hardware cost constraints typical of mass produced embedded systems.

When embedded software itself was simple, there was hardly a need for state-of-the-art tools. However, with the complexity of modern embedded applications, the straightforward make-compile-download-debug approach has become the bottleneck and advanced software design methodology is needed to increase productivity and product quality. Along this line there is a growing interest towards standardization of Real-Time Operating Systems either de facto, see for example the market penetration of Embedded Linux in this domain, or through standard bodies such as the OSEK committee established by the German automotive industry. Quite small is the market for advanced development tools even though this is the place where much productivity gain can be had.

Already in use in many application areas is the method to capture system specications at higher levels of abstraction and forcing designers to use tools that emphasize mathematical descriptions, making software code an output of the tool rather than an input. One example of such tool is MatLab, which allows designers to develop their concept in a friendly graphical environment where they can assemble their designs quickly and simulate their behavior with the associated SimuLink tool. While this approach is definitely along the right direction, the mathematical models supported by Matlab and more pertinently by SimuLink are somewhat limited and do not cover the full spectrum of embedded system design. The understanding of the mathematical properties of the embedded system functionality will be a major emphasis in the future.

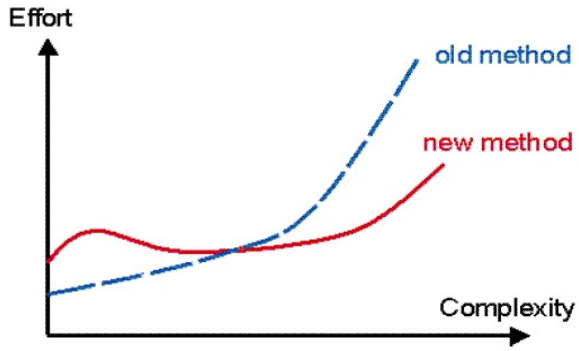


Figure 1 - Development method beats complexity.

Hand in hand with advanced methodology follows an introduction of more secure and efficient ways to generate control software for future ECU generations. In the following pages I will outline the far and the near future of embedded software development.

2. Software Engineering in the 21st Century

An integrated development environment for embedded systems based on Windows-PC was the de-facto standard of the late 20th century in the IT industry. In 2006 the automotive ECU development environment got closer to this standard, thanks to ubiquitous open source tools which became more and more popular during the last 5 years.

Current software generations already are designed with flexibility in mind and present a transition from the fixed systems approach of the last century. But there still is a need to develop a methodology and process for real-time embedded software development that supports management of evolution in a more flexible and dynamic way. Evolution is hereby considered in a broad sense: evolution of requirements, systems and system families, system architectures, individual components, resource constraints (timing & memory requirements) and underlying hardware. To reach this goal, an adaptable component-based architecture and enabling self-verifying infrastructure must be developed, providing support for evolution at both development time as well as run-time. This new architecture and infrastructure will lead to:

- Support for controlled evolution of the system architecture and design.
- Adaptable systems that can deal with dynamic reconfiguration (i.e. at run-time).
- Industrial system development by regenerating existing systems in order to reduce time-to-market [this feature was included for the finance department].
- Managed lifecycle support of product families with different degrees of functionality and/or hardware. System implementation becomes independent of actual hardware.

Each specific application and its environment have different requirements and need different approaches for matching embedded software [domain driven languages and architectures, for example]. It should be possible to co-design all these different approaches in parallel and keep track of the related requirements. Systems might be built flexible enough to adapt themselves automatically and/or allow (remote) updates/upgrades of a part of the embedded software in a consistent and robust manner.

In the near future the Wintel Personal Computer will continue to be the workhorse for software generation, but what's happening "under the hood" has been unheard yet in the world of software for automobiles.

3. Horizon 2020

The World in 2020: The United States of America own parts of other continents: Siberia, Southern Africa, and most of Arabia, as well as Norway and Great Britain now adhere to the star-spangled banner under George P. Bush.

The NASA mission to Mars is still crippled by budgetary limitations. However, the Bill and Melinda Gates Foundation proposes holidays on the Moon where they established the *Windows to the World* Base in 2018.

Europe 2020: European post war baby boomers reach retirement age, causing collapse of all state-funded retirement payment systems. Germany has the most arcane tax system in the world. The majority of older Germans now live outside Germany, more than 40% of people living in Germany have no german ancestry. Berlin became the second largest city of Turkey

after the 2016 earthquake.

European Union includes Israel, Moldovia, Morroco, Tunesia, Turkey, Switzerland, the Ukraine, and what is left of Russia.

Automotive Industry 2020: Toyota maintains biggest market share in US, Germany and Japan. The world's best selling vehicle is the HCC 2020 by the Han Car Company, a chinese/corean cooperation specialized in value-reduced copies of older western and japanese Selfdrive vehicles.

AVC vehicles run safer, faster, and cheaper than human operated Selfdrive vehicles. Public-, and Tele-Transportation are free in most Europe, and the US.

On the road in 2020: Driving in 2020 will become more of taking a ride with an individualized automotive carrier system than the nerve-wrecking experience of self-navigating a car through early 21st century traffic. With many driver assistance devices already available in 2006 (lane departure warning, adaptive cruise control with stop-n-go mode, night vision, automated parking, collision pre-brake system, etc.), the road certification of steer-by-wire opened the way for fully automatic drive functions.

Following the certification of Automated Vehicle Control (AVC) in 2014, in 2020 almost 50 percent of vehicles in USA, parts of Asia and Europe are AVC capable. Research programs from the 1990s made it into reality, there was a historic Bosch roadmap from 2002 featuring growing software complexity as well as growing networking capability leading to automated vehicle control.

The growing interaction of cars with their environment caused enormous growth in software content, which, together with growing availability of computing power, lead to vehicles which can care for themselves! Old dreams of car makers will come true, whether it’s BMW and Benz telling the other cars on the road to get out of their way or drivers who can take their breakfast while being commuted by their cars on the freeway. What once was known under the name of car will become an autonomous transportation service.

What will follow beyond 2020 is difficult to say, once our environment is completly networked the need for transportation may be less of a central point in our lives. May be cars will become a piece in human history, and be simulated in artificial environments as a recreational activity much like horse-riding had been transformed into a leisure activity during the 20th century.

4. The Origins (where we are now)

The overall goal of embedded system design (as well as other economic activity) can be summarized as follows:

Minimize

- production cost,
- development time and cost

under constraints on system performance and functionality.

Recently, Embedded Systems design became the focus of many strategic research and development activity, where before, it was at best hidden in factory basements between soldering iron and oscilloscope. Currently dominating the software development process is the definition and use of platforms.

Embedded Systems design adapts its philosophy to meet performance, quality, safety, cost and time-to-market constraints introduced by the pervasive use of electronics in everyday objects and the inherent desire for economic gains. An essential component newer system design paradigms is the orthogonalization of concerns, i.e., the separation of the various aspects of design to allow more effective exploration of alternative solutions. The design methodology has cristalized into the separation between:

- function (what the system is supposed to do) and architecture (how it does it);
- communication and computation.

The mapping of function to architecture is an essential step from conception to implementation. When mapping the functionality of the system to an integrated circuit, the economics of chip design and manufacturing are essential to determine the quality and the cost of the system. Since the mask set and design cost for Deep Sub-Micron implementations is predicted to be overwhelming, it is important to find common architectures that can support a variety of

applications. The recent shift to FPGA based implementations may be a precursor of the necessary adaptation of design strategy.

To reduce design costs, re-use is the easy way. And since system designers use more and more dominating software to implement their actual products, the substantial re-use of software becomes a necessity. Today this implies that the basic architecture of the implementation is essentially fixed, i.e., the principal components should remain the same within a certain degree of parameterization.

For embedded systems the basic hardware architecture consists of programmable cores, I/O subsystem and memories. A family of such architectures which allow substantial re-use of software is called a hardware platform. However, the concept of hardware platform by itself is not enough to achieve the sought after level of application software re-use. To be useful, the hardware platform has to be abstracted at a level where the application software only sees a high-level interface to the hardware, as if it were another piece of software.

Currently this is a software layer which wraps the different parts of the hardware platform: the programmable cores and the memory subsystem via a Real-Time Operating System (RTOS), the I/O subsystem via the Device Drivers, and the network connection via the network communication subsystem, in more primitive systems there often is only a Hardware Abstraction Layer (HAL) provided.

Following below are defined the system stakeholders in the platform based design approach:

I. Function

A system implements a set of functions. A function is an abstract view of the behavior of the system. It is the input/output characterization of the system with respect to its environment. It has no notion of implementation associated to it. For example, when the engine of a car starts (input), the display of the number of revolutions per minute of the engine (output) is a function, while when the engine starts, the display in digital form of the number of revolutions per minute on the LCD panel is not a function. In this case the display device is an LCD and the format of the data is digital. Similarly, when the driver moves the direction indicator (input), the display of a sign that the direction indicator is used until it is returned in its base position is a function, while when the driver moves the direction indicator, the emission of an intermittent sound until it is returned to its base position is not a function.

The notion of function depends very much on the level of abstraction at which the design is entered. For example, the decision whether to use sound or a visual indication about the direction indicator may not be a free parameter of the design. Consequently, the second description of the example is indeed a function since the specification is in terms of sound. However, even in this case, it is important to realize that there is a higher level of abstraction where the decision about the type of signal is made. This may uncover new designs that were not even considered because of the entry level of the design.

II. Architecture / Hardware Platform

An architecture is a set of components, either abstract or with a physical dimension, that is used to implement a function. For example, an LCD, a physical component of an architecture, can be used to display the number of revolutions per minute of an automotive engine. In this case, the component has a concrete, physical representation. In other cases it may have a more abstract representation. In general, a component is an element with specified interfaces and explicit context dependency. The architecture determines the final hardware implementation and hence it is strictly related to the concept of platform.

The architecture for the majority of embedded designs consists of microprocessors, peripherals, dedicated logic blocks and memories. For some products, the architecture is completely or in part fixed. In the case of automotive electronic control units, the actual placement of the electronic components inside the car and their connections is kept mostly fixed, while single components, i.e., the processors, may vary to a certain extent.

III. Mapping

The essential design step that allows moving down the levels of the design flow is the mapping process, where the functions to be implemented are assigned (mapped) to the components of the architecture. For example, the computations needed to display a set of signals may all be mapped to different components of the architecture, e.g., a microprocessor and a DSP as in a multicore processor or an FPGA, depending on availability of resources. The mapping process determines the performance and the cost of the design. To measure exactly the performance of the design and its cost in terms of used resources, it is often necessary to complete the design, leading to a number of time consuming design cycles. This is a motivation for using a more rigorous design methodology. When the mapping step is carried out, our choice is dictated by estimates of the performance of the implementation of that function (or part of it) onto the architecture component. Estimates can be provided either by the component manufacturers or by system designers. Designers use their experience and analysis to develop estimation models for fast design exploration. Given the importance of this step in any application domain, automated tools and environments should accurately support the mapping of functions to architectures.

The mapping process is best carried out interactively in the design environment. The output of the process is either:

- a mapped architecture iteratively refined towards the final implementation with a set of constraints on each mapped component (derived from the top-level design constraints), or
- a set of diagnostics to the architecture and function selection phase in case the estimation process signals that design constraints may not be met with the present architecture and function set. In this case, if possible, an alternative architecture is selected (low end vs. high end).

Once the mapped architecture has been estimated as capable of meeting the design constraints we have to solve the problem of implementing the components of the architecture. This requires the development of an appropriate hardware block or of the software needed to make the programmable components perform the appropriate computations. This step brings the design to the final implementation stage. The hardware block may be found in an existing library or may need a special purpose implementation. In this case, it may be further decomposed into sub-blocks until either what is needed is found in a library or an implementation by custom design is decided. The software components may exist already in a domain library or may need further decomposition into a set of sub-components.

IV. Software Platform

The concept of hardware platform by itself is not enough to achieve the level of application software re-use the world is looking for. The base software layer wraps the essential parts of the hardware platform:

- the programmable cores and the memory subsystem via a Real-Time Operating System (RTOS),
- the I/O subsystem via the Device Drivers, or Board Support Package, and
- the network connection via the network communication subsystem.

We call this layer software platform.

Here the programming language is the abstraction of the Instruction Set Architecture (ISA), while the API is the abstraction of a multiplicity of computational resources (concurrency model provided by the RTOS) and available peripherals (Device Drivers). In our framework, the API is a unique abstract representation of the hardware platform, similar to the HAL in simple systems. With an API so defined, the application software can be re-used for every platform instance.

V. System Platform

The combination of hardware and software platforms is called system platform. It is the system platform that completely determines the degree of re-usability of the application software and identifies the hardware platform. In fact, the hardware platform may be deeply affected by the software platform as in the case of the PC platform. In fact, the re-usability of the

application software made possible by the PC system platform comes at the expense of a very complex layered structure of the different O.S. families and of the size and complexity of the microprocessor cores supporting the instruction set. In the PC domain, this price is well worth being paid.

In the automotive domain, the use of the OSEK/RTOS specifications, sponsored by the German automotive manufacturers, was conceived as a tool for re-usability and flexibility using the PC platform as a model. However, the peculiarities of embedded systems where cost, reliability and geometric constraints play a fundamental role, make this approach difficult to use. The OSEK initiative intended to set a standard for operating systems to be used in the car so that software produced by different vendors could be easily integrated. OSEK constrains the RTOS and part of the network communication system, but does not constrain any input/output system interface exposing the hardware details of the I/O subsystem to the software. Because in the OSEK specification a device driver API is not defined, this specification is not an architectural constraint strong enough to enable alone the re-use of software components even at source code level, letting each system maker define its own interface to interact with the attached devices. If we add to the OSEK standard for RTOS and Network management, the definition of the I/O sub-systems and the use of a single ISA, even binary software components can be reused.

For a couple of years now the AUTOSAR initiative has taken over for systems interface definition to constitue a set of layers, interfaces and guidelines of interconnection to allow better interoperability of software among OEMs and suppliers. While sharing source code is no black magic, it gets tricky on the binary level, even if the amount of available hardware architectures has been limited to a handful.

5. Embedded Processors for 2020

Embedded engine control in the 21st century will rely on adaptive self-organizing control structures. Goodbye "Kennlinienfelderbezeichnerlabeldaten", binary arithmetic, instruction sets, and the "C" programming language. Hello 1-billion transistor, ubiquitous processors and silicon as cheap as dirt. Welcome to the embedded tomorrowworld! Just remember:

The purpose of embedded technology is to blend into the background, to become an everyday part of our lives.

Extrapolating from 15 years ago suggests that some of 2020's most popular microprocessors haven't been invented yet. The Transmeta chips were a good example of yet to be invented technology, even if not hugely successful.

Future generations of Über-processors will cooperate, sharing tasks among hundreds of processors on the same chip. Interprocessor communication and on-chip networks will take up most of the silicon. The processors themselves will take a technological back seat to the tiny networks that help them communicate.

Nobody programs in assembly language or C any more. Processors will likely adapt their features over time, learning and morphing as they adjust to changing workloads.

At the extreme end of this spectrum will be "soft" virtual processors, with almost no native instructions at all. Instead, they'll run emulation or binary-translation code as a kind of ultra-low-level operating system that enables them to execute software from any legacy processor.

Embedded will be employed everywhere as long as it makes money for someone. With processing power cheaper than mechanical equivalents, even mundane devices get booted up to the next level. Rear-view mirrors with embedded image sensors recognize an impending collision and warn both drivers. Processors will be so cheap and ubiquitous that they'll be disposable. We'll throw away the equivalent of a Cray supercomputer with each week's trash.

GPS receivers and wireless transmitters will accompany every solid object of any value. Cell phones (and wireless data networks) will become nonproducts: uninteresting by themselves, but an added feature on another product, like a camera. Already researchers can transmit sound waves through arm and hand bones; to use the comunicator in your pocket you stick your finger in your ear.

Computers required enormous power supplies. Excess heat from processors is a main problem in embedded systems design. Power from shaking or squeezing a chip will be enough to drive most low-end microprocessors. But our demand for processing performance will rise faster than power consumption will fall.

6. Embedded Systems for 2020

Looking ten..fifteen years ahead is far long, but not long enough for some of the stranger predictions to come into being. Maybe there won't be microprocessors as we know them, at all. Dynamically reconfigurable logic (FPGA) has captured the imagination (and investment capital) of many who see it as an efficient alternative to classic microprocessor architectures. Instead of programming a fixed processor with variable instructions, why not just make the hardware vary over time to fit dynamic requirements? An emerging hardware/software language will become the design and programming tool of choice for new embedded systems, currently there are evolving SystemC and other candidates to be developed into a standard language yet.

Already appreaing advanced drivetrain schemes (multiple engines / motors) mandate distributed control strategies. Intelligent sensors and actuators will form self-organizing networks to implement engine / drivetrain control. In 2015 ample processing power will be available at any place within the car. Energy to these devices may be supplied electrically by wire or induction, mechanically by motion / vibration / noise / or heat. Copper wires as we know them will become the exception in 2015 controller networks. Fibre or wireless connections will shave a few kilos off a car's weight, when hauling along the kilos becomes more and more expansive with rising energy costs.

With sufficient intelligence in sensors and actuators the traditional central fixed programming gives way to adaptive distributed function implementation, i.e. injection unit runs in close loop with corresponding crankshaft acceleration sensors and the whole is supervised by a drive manager ECU, which may cooperate with a security unit as well as with an Human Machine Interface (HMI) manager unit.

Auto calibration replaces todays individual calibration of engine control functions and corresponding overhead. In communication with the available sensors and actuators, the engine control itself figures out how to adapt the most efficiently to any usage profile. Usage profiles may contain any requirement, be that driver temperament, vehicle type, environment or legal restrictions.

Switch times between different usage states supply enough system idle time for reprogramming of components, i.e. software that reprograms hardware to be in optimal condition for the following use. How many processor cycles fit into one gear shift?

Last not least, when original hardware architectures have been shifted out of production, future systems may contain historic control unit functionality (based on MPC 555, TriCore, ..),operated in software emulation mode, as the hardware/software divide can be moved around at will. Already today reconfigurable hardware is frequently used for implementation of obsolete microcontrollers. Vendors offer soft cores based on historic 8051, 68000 and other microprocessors.

In commercial development organizations, increased complexity of products, shortened development cycles, and higher customer expectations of quality have placed a major responsibility on the areas of software debugging, testing, and verification. Today we have exciting improvements in the underlying technology on all three fronts. However, due to the informal nature of software development as a whole, the prevalent practices in the industry are still immature, even in areas where improved technology exists.

In addition, tools that incorporate the more advanced aspects of this technology are not ready for large-scale commercial use. Hence hope for significant improvement is in the air and next generations ECU will make the leap forward since existing software generation was leading into dead end developments and an insufficiently developed software engineering process poses a larger risk to complex car software.

A key ingredient that contributes to a reliable programming systems product is the assurance

that the program will perform satisfactorily in terms of its functional and nonfunctional specifications within the expected deployment environment. In today's typical automotive software development organization, the cost of providing this assurance easily ranges above 50 percent of the total development cost.

May be there will be one common control unit for the world's automotive manufacturers. As there were some 15 different MPC 555 based ECUs used by different OEMs for almost the same purpose, this artificial diversity will vanish due to market pressures. The 2010 electronic control unit may be running at 2 GHz with 500 MByte of memory and the MS Engine Operating System?

7. Code Generation is the Key

Code generation of the past has been about writing programs, and understanding them as we write them. Most large computer programs, and current ECU software qualifies as per its inherent complexity as large computer program, are never completely understood. If they were, they would not go wrong so often and we would be able to describe what they do in a scientific way. A good language should help to improve this state of affairs.

There are many ways of trying to understand programs. In 2006 the automotive industry for a large part relies on one way, which is called "testing and debugging" and consists of running a partly understood program to see if it does what is expected. The whole process is deeply linked to quality methodology which generates enormous overhead effort.

A tempting alternative is to install some means of understanding in the very program code itself.

People used to argue that any overhead in code in order to make code more robust and more secure, runs counterproductive to the competitive situation in the marketplace, i.e. resource-reducing code was valued higher than safer code. This tradeoff worked well in the past with high costs of hardware and little complexity of applications.

During the last ten years of digital engine control software the execution power of hardware grew by a factor of fifty, the number of injections by a factor of five, the number of functions by ten and overall complexity grew more than hundredfold! Today even the most advanced quality assurance schemes run short on actual requirements for software product validation.

When Microsoft sends software updates to millions of PCs, this is regarded mostly as a waste of [customer's life] time. When engine control sends millions of cars into limp home mode, possible accidents cause far more damage to human life and property than can be covered by any insurance scheme.

One of the fundamental features of embedded software is that it interacts with the physical world. The behaviors of an automobile engine are divided into regions of operation, each characterized by appropriate control actions to achieve a desired result. Conventional axiomatic or denotational semantics of sequential programming languages only model initial and final states of terminating programs. Thus, these semantics are inadequate to fully model embedded software. However, much of the code in an embedded application does computation or internal communication, rather than interacting with the physical world. Such code can be adequately modeled using conventional semantics, as long as the model can be integrated with the more detailed semantics necessary for modeling interaction.

Pre-post trace structures are quite similar to conventional semantics. Thus, we can model the non-interactive parts of an embedded application at a high level of abstraction that is simpler and more natural, while also being able to integrate accurate models of interaction, real-time constraints and continuous dynamics.

But apart all the formal mathematical models and use of industrial style software creation, the use of domain design experts must be reintroduced in today's offshored development efforts. These experts may not be cheap, and they should accompany the whole lifecycle of ECU products. In the past, many automotive development projects have experienced the errors which occur when people who have very limited experience with the end product, are used to program electronic control units. While all quality norms may be achieved by some

software production process, only domain experts can assure that the right requirements feed the generation tools to render perfect software.

8. Design for Safety

While in older ECU generations the prevalent factor was efficiency with limited resources in mind, the future will bring resources plentiful [my current phone has more processing power and memory than the PC under my desk 10 years ago], but complexity will grow accordingly and needs to be managed to keep systems safe.

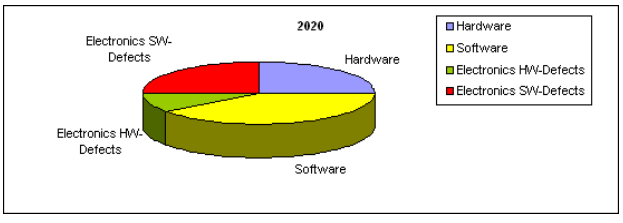
When looking for safer system design for future ECU developments, a possible road to follow is not so difficult to find: Aerospace always had and will continue to have the most stringent security requirements. Looking for generation of safe code?

- Throw out the C-language as intermediate requirements representation alltogether.
- Throw out resource optimization.
- Get redundancy in your system architecture,
- Get a self checking language,
- Move to proof by execution and further advanced schemes where generated code is correct by specification.
- Assure that requirements are sound since implementation will match requirements.

Like the C-language has not been invented with security-related applications in mind, a new language is necessary to cover the requirements of the automotive industry in the future.

APPENDIX

Estimation of electronics costs in automobiles 2020:



ACKNOWLEDGMENT

Acknowledgments (if any) should appear as a separate non-numbered section before the list of references. Use the singular heading even if you have many acknowledgments.

CONTACT

BERNHARD KOCKOTH ENGINEERING

KREUZSTR. 37
D-76133 KARLSRUHE

MOBILE: +49 (0)1577 4080 253
MAIL: BK@EMBEDDEDEXPERT.COM
HTTP://WWW.KOCKOTH-ENGINEERING.COM

BIOGRAPHY

Bernhard Kockoth is experienced in 32-bit embedded systems since the late 80s. At the time he designed an animated simulator for the Motorola 68k family. In the early 90s he joined Bull Engineering in Paris for work on multiprocessor network encryption systems. With his excellent background in digital signal processing he worked for Texas Instruments European Marketing as product specialist. In 1996 he became European Application Engineer with Embedded Support Tools Corp in its european center near Paris. After successful missions at EST office startups in Germany and England he was appointed European Support Manager at the US headquarters in Boston. Due to market pressures back in Europe since 2001 he worked as Embedded Tools Manager for a Robert Bosch Diesel Systems until 2004. Another five years with another Tier-1 supplier, this time Johnson Controls Automotive Electronics in Karlsruhe where Bernhard worked as software project leader and participated in AUTOSAR Workpackage 1.2 for tools and methodology. For the time being Bernhard is an independent consultant on embedded systems implementation and methodology.
Bernhard Kockoth holds an MS in Electrical Engineering and Computer Science from Ruhr University Bochum, Germany.