Universidade de Aveiro
2015

Departamento de Electrónica,
Telecomunicações e Informática

Pedro Manuel
Casal Kulzer

Nova Combinação de Hardware e de Software para Veículos de Desporto Automóvel Baseada no Processamento Directo de Funções Gráficas

Novel Hardware and Software Combination for Automotive Motorsport Vehicles Based on Direct Processing of Graphical Functions

Universidade de Aveiro
2015

Departamento de Electrónica,
Telecomunicações e Informática

Pedro Manuel
Casal Kulzer

Nova Combinação De Hardware e de Software para Veículos de Desporto Automóvel Baseada no Processamento Directo de Funções Gráficas

Novel Hardware and Software Combination for Automotive Motorsport Vehicles Based on Direct Processing of Graphical Functions

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Engenharia Electrónica, realizada sob a orientação científica do Doutor Manuel Bernardo Salvador Cunha, Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Apoio financeiro da FCT e do FSE no
âmbito do III Quadro Comunitário de Apoio.

aos meus Pais,
ao meu Irmão,
e à minha Família

Agradecimentos

O meu maior agradecimento especial vai direitinho para toda a vasta equipa *"ECU2010"*, constituída pelo Nelson "auto-defesa" Bernardino, Filipe "bolachas" Teixeira, Paulo "kamik2000" Martins, Rui "rex" Gomes, Ricardo Marques, Cátia Ferreira (apenas os 4 primeiros chegaram comigo ao fim) e, *last but not least*, pelo meu enorme orientador e mentor, um verdadeiro Engenheiro no melhor sentido do termo, o Prof. Bernardo Cunha.

Agradeço também, especialmente, as valiosas contribuições do Prof. Arnaldo Oliveira e Prof. Luís Almeida, bem como o Prof. Francisco Vaz e Prof. Keith L. Doty por terem influenciado positivamente o meu percurso académico e por me terem convencido a, finalmente, realizar esta tese de Doutoramento.

Agradeço ainda ao Eng. Andreas Becker da Bosch Motorsport, por me ter apoiado sempre e por ter acreditado no projecto *"ECU2010"* desde o início.

Finalmente, não há agradecimento possível para os meus pais e minha família, por me terem dado todas as condições para poder prosseguir os estudos universitários da melhor forma e por me terem aturado durante estes 8 anos de desenvolvimento/escrita desta tese *"ECU2010"*, respectivamente.

palavras-chave

engenharia automóvel, controlo de motor, controlo de chassis, software integrado, programação visual, processamento gráfico, processamento paralelo, arquitectura paralela, periféricos inteligentes

resumo

A principal motivação para o trabalho que conduziu a esta tese residiu na constatação de que os actuais métodos de modelação de centralinas automóveis conduzem a significativos problemas de desenvolvimento e manutenção. Como resultado dessa constatação, o objectivo deste trabalho centrou-se no desenvolvimento de um conceito de arquitectura que rompe radicalmente com os modelos state-of-the-art e que assenta num conjunto de conceitos que vieram a ser designados de *"Macro"* e *"Celular ECU"*. Com este modelo pretendeu-se simultaneamente minimizar a panóplia de software e de hardware necessários à obtenção de uma sistema funcional final.

Inicialmente, esta tese propõem-se fazer um levantamento exaustivo do estado da arte na área específica do software e correspondente hardware das ferramentas e centralinas automóveis. Os problemas decorrentes de tal software serão identificados e, dessa identificação deverá ficar claro, que praticamente todos esses problemas têm origem directa ou indirecta no facto de se continuar a fazer um uso exaustivo de "tool chains" extremamente compridas e complexas. De forma semelhante, no hardware, os problemas têm origem na extrema complexidade e inter-dependência das arquitecturas dos processadores. As consequências distribuem-se por uma extensa lista de *"pitfalls"* que também serão exaustivamente enumeradas, identificadas e caracterizadas.

São ainda propostas soluções para os diversos problemas actuais e correspondentes implementações dessas mesmas soluções. Todo este trabalho final faz parte de um sistema *"proof-of-concept"* designado *"ECU2010"*. O elemento central deste sistema é o já referido conceito de *"Macro"*, que consiste num bloco gráfico que representa uma de muitas operações necessárias num sistema automóvel, como sejam funções aritméticas, lógicas, de filtragem, de integração, de multiplexagem, entre outras. O resultado final do trabalho proposto assenta numa única ferramenta, totalmente integrada que permite o desenvolvimento e gestão de todo o sistema de forma simples numa única interface visual. Parte do resultado apresentado assenta numa plataforma hardware totalmente adaptada ao software, bem como na elevada flexibilidade e escalabilidade, para além de permitir a utilização de exactamente a mesma tecnologia quer para a centralina, como para o datalogger e para os periféricos.

Os sistemas actuais assentam num percurso maioritariamente evolutivo, apenas permitindo a calibração online de parâmetros, mas nunca a alteração online dos próprios algoritmos das funcionalidades automóveis. Pelo contrário, o sistema desenvolvido e descrito nesta tese apresenta a vantagem de seguir um *"clean-slate approach"*, pelo que tudo pode ser globalmente repensado. No final e para além de todas as restantes características, o *"LIVE-PROTOTYPING"* é a funcionalidade mais relevante, ao permitir alterar algoritmos automóveis (ex: injecção, ignição, controlo lambda, etc.) de forma 100% online, mantendo o motor constantemente a trabalhar e sem nunca ter de o parar ou re-arrancar para efectuar tais alterações. Isto elimina consequentemente qualquer *"turnaround delay"* tipicamente presente em qualquer sistema automóvel actual, aumentando de forma significativa a eficiência global do sistema e da sua utilização.

keywords

automotive engineering, engine management, chassis management, integrated software, visual programming, graphical processing, parallel processing, parallel architecture, smart peripherals

abstract

The main motivation for the work presented here began with previously conducted experiments with a programming concept at the time named *"Macro"*. These experiments led to the conviction that it would be possible to build a system of engine control from scratch, which could eliminate many of the current problems of engine management systems in a direct and intrinsic way. It was also hoped that it would minimize the full range of software and hardware needed to make a final and fully functional system.

Initially, this paper proposes to make a comprehensive survey of the state of the art in the specific area of software and corresponding hardware of automotive tools and automotive ECUs. Problems arising from such software will be identified, and it will be clear that practically all of these problems stem directly or indirectly from the fact that we continue to make comprehensive use of extremely long and complex "tool chains". Similarly, in the hardware, it will be argued that the problems stem from the extreme complexity and inter-dependency inside processor architectures. The conclusions are presented through an extensive list of "pitfalls" which will be thoroughly enumerated, identified and characterized.

Solutions will also be proposed for the various current issues and for the implementation of these same solutions. All this final work will be part of a *"proof-of-concept"* system called *"ECU2010"*. The central element of this system is the before mentioned *"Macro"* concept, which is an graphical block representing one of many operations required in a automotive system having arithmetic, logic, filtering, integration, multiplexing functions among others. The end result of the proposed work is a single tool, fully integrated, enabling the development and management of the entire system in one simple visual interface. Part of the presented result relies on a hardware platform fully adapted to the software, as well as enabling high flexibility and scalability in addition to using exactly the same technology for ECU, data logger and peripherals alike.

Current systems rely on a mostly evolutionary path, only allowing online calibration of parameters, but never the online alteration of their own automotive functionality algorithms. By contrast, the system developed and described in this thesis had the advantage of following a *"clean-slate"* approach, whereby everything could be rethought globally. In the end, out of all the system characteristics, *"LIVE-Prototyping"* is the most relevant feature, allowing the adjustment of automotive algorithms (eg. Injection, ignition, lambda control, etc.) 100% online, keeping the engine constantly working, without ever having to stop or reboot to make such changes. This consequently eliminates any "turnaround delay" typically present in current automotive systems, thereby enhancing the efficiency and handling of such systems.

"When a paradigm-shattering discovery is made in science, it goes through three stages before gaining acceptance. First, people don’t believe it; second they claim it is of no interest; and third, they say that they always have known it.
Vilayanur S. Ramachandran

"You can´t overtake someone if you follow in his footsteps."
Francois Truffaut

"A programming language is low level when its programs require attention to the irrelevant."
Alan Perlis

"Two mistakes one can make along the road to truth: not going all the way; not starting."
Buddha

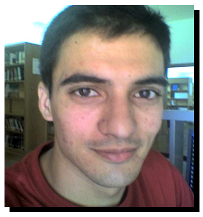
"In order to attain the impossible, one must attempt the absurd."
Miguel de Cervantes

"Most of things worth doing have been declared impossible before they were done."
Louis D. Brandeis

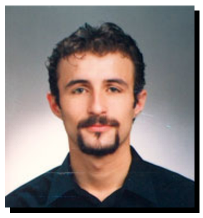
"Hold fast to your dreams, for without them life is a broken winged bird that can't fly."
Langston Hughes

Page intentionally left blank

I would like to thank to all the people involved in this PhD thesis, the ones who helped me achieve it in the range I desired, as well as those who played a decisive role in achieving it after a total of 8 years (2006-2014), with no particular order:

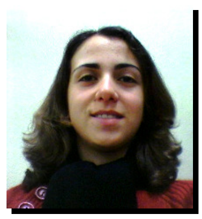

Eng. Nelson "self-defense" Bernardino graduated through the University of Aveiro in 2006 and was a member of the ECU2010 project right-after. I thank him for having implemented most of the software of the *"iEditor"* and most of its graphical components. I thank him for always keeping the necessary calmness, to be able to solve sometimes difficult to implement graphical details of the editor, as well as for never giving up when implementing something completely new without any references. Being a "black-belt" helped keeping everyone calm and controlled at all time… Thanks! ☺

Eng. Filipe "cookies" Teixeira graduated through the University of Aveiro in 2006 and was a member of the ECU2010 project right-after. I thank him for having implemented the entire parallel hardware and some of the graphical editor components. I thank him for being the main integrator of the entire project and for being able to keep the overview of all components at all times. Being a good-withed guy, he always kept the good mood and the hope that everything would work at the end. And it did… His reaction on the very day the engine started: "HEY!!! IT WORKS!!! IT WORKS!!! SPECTACULAR!!!"… Thanks! ☺


Eng. Paulo "kamik2000" Martins graduated through the University of Aveiro in 2006 and was a member of the ECU2010 project after working for 1 year at Grundig. I thank him for having implemented the USB communications, MMC datalogging, wireless datalogger hardware and software, and all PCBs. Being a very inventive guy, very good at “spacey” ideas, he was the problem-solver. Always in search and highly needy for new things, he kept everybody busy at all time by explaining and implementing some of his bright ideas and concepts into this project… Thanks! ☺

Eng. Rui "rex" Gomes graduated through the University of Aveiro in 2006 and was a member of the ECU2010 project right-after. I thank him for having implemented the entire Smart Peripherals part in both the hardware and software components. Being a very calm guy he was able to cope with the enormous piece of work on his hands, conceive all the related ideas and deliver everything working at the end. It was a miracle that all peripherals actually worked fine on the engine, having all those contacts and very low-level crankshaft--related high-speed details. Thanks! ☺

Eng. Ricardo Marques graduated through the University of Aveiro in 2006 and was a member of the ECU2010 project right-after. I thank him for having implemented all the *"Macro-Processing"* infra-structures, including the highly advanced *"Live-Prototyping"* mechanism fromm scratch and without any similar reference. He also implemented the *"Live-Simulation"* features in 2-3 days, which where then later enhanced and completed by Filipe Teixeira. He left the project at July 2008, being all his tasks being mostly delegated to Filipe Teixeira and also some tasks to Rui Gomes. Thanks! ☺

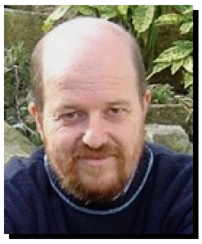

Eng. Cátia Ferreira graduated through the University of Aveiro in 2006 and was a member of the ECU2010 project right-after. I thank her for having implemented all the *“Macros”* and the corresponding ALU and CPU processing units in VHDL. She also did all the optimization and profiling work, especially on the arithmetic *"Macros"*, measuring their execution performance. She left the project as from May 2008 and all her few remaining tasks (implementing the one or other still missing simple *"Macros"*) where delegated onto Filipe Teixeira. Thanks! ☺

Prof. Arnaldo Oliveira, professor at the University of Aveiro, managed and assisted to the last-year work done by Eng. Cátia Ferreira, helping her on XILINX ISE and VHDL detais and techniques. I thank him also for being always ready and helpful when the team needed assistance in detai questions, as well as for having given the team a comprehensive tutorial class at the beginning of this project. He also helped us installing all the needed XILINX ISE software packages. Thanks! ☺

Prof. Luís de Almeida, professor at the University of Aveiro, helped the team out, when it came to communicartions and critical timing details. He was especially usefull when discussing details of the digital-bus for the "Smart Peripherals" and the *"Cellular-ECU"* inter-communications serial ring bus. I also thank him for warning us constantly of potential pitfalls during other technological choices along the entire project. He always was very curious in knowing the progress of this project and to help us out. ☺

Prof. Francisco Vaz, professor at the University of Aveiro and ex-vice-rector of the same University, was the original starter of this project. He was the one who told me right after finishing my course and my Master's thesis, that I should continue to a PhD thesis. I declined and went to Bosch Motorsport in Germany, just to return to Portugal and to this University, by initiating this project under his advisement. After about 2 years of this project, he finally convinced me to make my PhD based on the ideas and results of this project, whereby I thank him most humbly, for having finally convinced me! ☺

Prof. Keith L. Doty, professor emeritus at the University of Florida, USA, was my mentor when I was doing my Master's thesis at the University of Florida. He was then the director of the *"MIL – Machine Intelligence Laboratory"* and taught me many important things, both in the academic and personal fields. He also tried to convince me several times in doing my PhD, telling me that since I did not desire to pursuit the academic path, it would then be my own *"Nobel Prize"*. That is exactly what it ended up to be. Thanks for everything! ☺


Prof. Bernardo Cunha, professor at the University of Aveiro, was the technical, theoretical, practical and even psychological "rock" for everyone to hold to whenever it got rough and difficult. He supplied the modified single-cylinder engine that operated through the *"ECU2010"* system at the very end. He coined the name *"Cellular-ECU"* for the processing modules developed herein. He was always there when needed and his technical wisdom and calmness upon all kinds of problems, helped us all a lot to overcome them and to get to a successful DEMO at the very end. He also made the final *"ECU2010 Presentation Film"* shown at Bosch Motorsport upon the DEMO delivered in Germany. Finally, he also was my mentor during my PhD work, especially during thesis writing and all preparations, readings, corrections, among many other important details. Without him I would never have been able to reach the point of thesis closure and final presentation. I therefore thank him most among all, for all the time and effort he invested into this work. Last but not least, I am very sorry for all the "grey hair" and desperation (pun intended) you had to sustain during revision and correction efforts of this thesis… you did not see it coming at all… A BIG BIG BIG THANK YOU FOR EVERYTHING!!! ☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺!!!

I thank the University of Aveiro for allowing me to earn all my 3 degrees (MSc, Master's, PhD) there and for allowing the *"ECU2010"* team to occupy its facilities during the project.

I thank Robert Bosch GmbH, one of the sponsors and interested parties, stepping into this project through Dr. Ansgar Christ and my brother Dr. André Kulzer, both from CR/AEE2. I thank my brother for believing in me and for the continuous interest in this project.

Bosch Motorsport, inside Bosch Engineering, a Robert Bosch company, main sponsor in this project. Working for this deparment since 2001, I sincerely thank AS/MSD, my ex-colleagues, its director Dr. Klaus Böttcher and especially my ex-boss, Ing. Andreas Becker, for all the support, always encouraging and backing me up, and for giving me the fine chance to be able to work in such an interesting and ever-evolving competitive area.

I thank Fundação para a Ciência e Tecnologia for having granted me the schoolarship *[SFRH_BD_62206_2009]* during the first period of my PhD, from NOV/2009 to MAR/2011.

I thank Agência de Inovação for having recognised value in this work and sponsored part of the initial 6 Engineers' salaries during the period of SEP/2006 to DEC/2008.

Last but not least… I dearly thank my parents for having given me the best conditions possible for pursuing my university studies, and my family: wife and kids, for accepting the risk of letting me dive into this endeavour without knowing the consequences beforehand, in any respect: personally, financially and professionally, besides time consumption and mood changes. In the end, this project turned out just fine, except for its sudden cancellation in the middle if 2009, right before it had already been planned to soon operate inside a real race car… What a great pitty… That would have certainly been an additionally fine and awesome feat for the team!!!

Furthermore and also dearly important to me, I finally achieved a hidden dream of my University learning and fascination times: finally getting my very own personal *"Nobel Prize"*.

To my children, always asking me: "Pappa, when do you finally finish your PhD and have a normal life?!", my answer is loud and clear: "Now it's finished!"… Thanks for asking!   ;-))))

ABBREVIATIONS & ACRONYMS

[Att. xx]	- Attachment "xx" to this thesis
ADC / DAC	- Analog-to-Digital Converter / Digital-to-Analog Converter
ALU	- Arithmetic Logic Unit
ASIC	- Application-Specific Integrated Circuit
BCDP	- Binary Coded Decimal Power numbering scheme
CAPCOM	- Capture-Compare peripheral of a CPU
CCP / SPI	- CAN Communications Protocol / Serial-Peripheral Interface
CPU	- Central Processing Unit of a computer
CRC	- Cyclic Redundancy Check
CRCW / CREW	- Concurrent read concurrent write / Concurrent read exclusive write
CT / HT	- "Cycle-Time" / Hop-Time
DSL	- Domain-Specific Language
ECU	- Electronic Control Unit
ECU2010	- Name given to the project done with Bosch Motorsport
EREW	- Exclusive read exclusive write
“CELLULAR-ECU”	- ECU/Peripherals concept based on ECU2010 modules
FDEF	- (Automotive) Function DEFinition (Bosch naming)
"FDEF-PROCESSOR"	- Complete "Macros-Processor"+ALU+GIMy+LP construct
FIFO	- First-In First-Out queue
FPGA	- Field Programmable Gate-Array
GIMy	- Global Intelligent Memory ("Nodes" values)
HiL / SiL / MiL / XiL	- Hardware/Software/Model/Mixed-in-the-Loop
HW / SW	- Hardware / Software
I/O	- Input / Output
IDE	- Integrated Development Environment
JIT	- Just-in-Time compilation
“LIVE-xxx”	- Online functionality executed during full system operation
LoC	- Lines-of-Code
"LP"	- "Live-Prototyping" block handling RAM+FLASH accesses
“MACRO”	- Basic functional (logic, arithmetic, etc.) operation
"MACROS-SEQUENCE"	- List of Macros; direct input for the Macros-Processor
"MACROS-PROCESSOR"	- CPU+ALU that natively executes "Macros" of an FDEF
MMC	- Multi-Media Card (flash storage card)
MOTRONIC	- Soft-/Hardware controlling a entire automotive vehicle
MUX & DEMUX	- Multiplexer and Demultiplexer
"NODE"	- Variable inside an FDEF ("Macro" inputs and outputs)
OEM	- Original Equipment Manufacturer
OOP	- Object-Oriented Program
OS	- Operating-System
PC	- Program-Counter
PCB	- Printed Circuit-Board
PDA	- Personal Digital Assistant (digital diary)
PLC	- Programmable Logic Controller
PWM	- Pulse-Width Modulation
OS	- Operating System
RF	- Radio-Frequency
RISC / CISC	- Reduced/Complex Instruction-Set Computing
RPM	- Rotations Per Minute
RTC	- Real-Time Clock
RX / TX	- Reception / Transmission
SP	- "Smart Peripheral"
USB	- Universal Serial Bus
VHDL	- VHSIC Hardware Description Language
VM / JVM	- Virtual-Machine / JAVA Virtual-Machine
WYSIWYG	- What You See Is What You Get

Page intentionally left blank

NOTE: *The entire work developed herein does not intend to build a general-purpose programming and development system, nor does it intend to be too generalist in any way. The sole purpose herein is to develop a software and hardware system adapted to the very specific automotive functionality and electronics realm, aimed at demonstrating a "proof-of-concept". It is not commercially optimized in any way, nor was this the initial scope of this project. It could easily be generalized to other control systems such as household apparatus (washing-machines, ovens, etc.), as well as industrial machinery and any other systems requiring control-algorithms, flexible hardware and easy-to-handle software. The work done herein fits into the area of "Embedded Systems", more specifically into the restricted area of "Languages for Coding of Problems in Embedded Systems".*

Page intentionally left blank

CONTENTS

LIST OF FIGURES, TABLES & EQUATIONS

Page intentionally left blank

Page intentionally left blank

Page intentionally left blank

"The most incomprehensible thing about the universe is that it is comprehensible."
– Albert Einstein

CHAPTER 1

Introduction

1.1 Motivation, Location & Time-Frame

[Emphasis: the "whys", "wheres" and "whens" of this thesis]

The main ideas and directions of this work had been acquired during employment at Bosch Motorsport, Markgröningen, Germany, where the author worked for 4 years from 2001-2005 and to whom he still works full-time, now from Portugal. There, the author was able to get acquainted with the many problems and limitations that automotive electronics present, especially in the Motorsport scene. Paying special attention to Series Automotive Technologies, Motorsport Electronics and especially to Motorsport Clients, it was possible to gather a large set of ideas that emerged simply by imagining those same situations but without all those problems and limitations that were observed on every day's work. These ideas were first/originally compiled and presented in 117 slides [563] at Bosch Motorsport back in March 2005 in front of an audience of work colleagues and a board of directors.

But it was only after the author's return to Portugal in 2005 and with the start of the *"ECU2010"* project, that the refinement of these ideas combined with new ideas, actually determined this work to finally start to appear. The location of all developments has always been room 234 at the Department of Electronics, Telecommunications and Informatics of the University of Aveiro, Portugal. All the material part of this work has been done in that room from September 1st, 2006 to August 31st, 2009. Meanwhile, it was on November 28th that the *"ECU2010"* DEMO day at Bosch Motorsport, Markgröningen, Germany, finally took place. There, all the work done was demonstrated with complete success.

After this total success in Germany, parallel to its *"proof-of-concept"*, the project went on up to August 2009 with an also successful attempt to integrate it into Bosch Motorsport's powerful MS5.2 ECU as an “add-on”. After that second demonstration, the project *"ECU2010"* was stopped indefinitely due to the financial world-crisis triggered in 2008. At this point, and taking into consideration the developed work, it was finally time to start thinking about writing down the results of all those technological achievements, which would eventually result in the current thesis.

1.2 Automotive Systems

[automotive control-systems and functionalities]

The automotive systems (Fig. 1.2:1) that are going to be treated throughout the whole thesis, are the specific hardware and software components which manage the chassis an engine of an automobile, be it a series-car or a race-car model. From conception and functional programming, up to the final target-hardware, all software/hardware components and layers in between those end-points will be treated here. Key-discussions throughout this thesis will be related with programming, debugging, testing and prototyping issues that Engineers face when handling this kind of systems, where available handling tools are very often proprietary and historically-grown inside the car-making or car-component companies themselves. Specific infotainment systems in modern cars are going to be intentionally left out of the discussions herein.

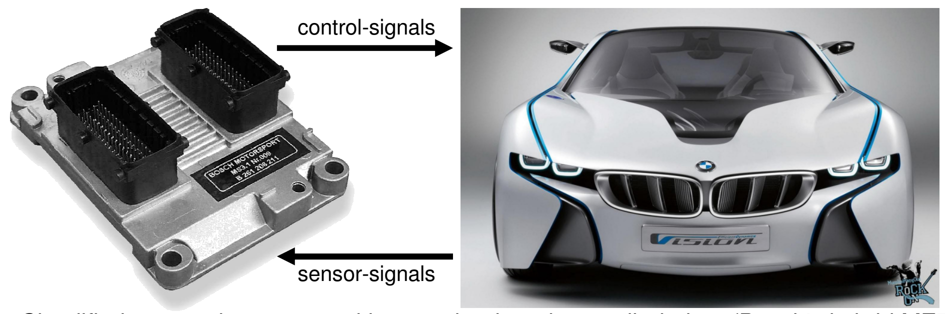


Fig. 1.2:1 – Simplified automotive system with control-unit and controlled-plant (Bosch's hybrid ME7.6 ECU)

To be even more precise, this thesis proposes a new combination of software and hardware to be able to handle those above-mentioned issues in a more efficient and easy manner, than with current state-of-the-art tools and platforms. Not focusing on the automotive functionalities themselves, such as the specific chassis- and engine-management software functions, this thesis discusses the ways and means to get to the implementation of those functionalities. In other words, programming and general handling environments that allow Engineers to handle the related software and hardware structures of those functions is going to be addressed.

One or more *Electronic Control Units (ECU)* control the various chassis and engine functionalities of the car (plant) as listed in Tab. 1.2:2, by inputting sensor values and outputting actuator values as well. Some functionalities even require separate and interconnected ECUs, due to complexity, safety and development reasons. The only functionalities which are mostly concentrated in a single ECU are the engine-management ones, due to the high processing and timing interconnections that exist inside those functionalities. Even those functionalities which reside in separate ECUs, are sometimes interconnected with high-speed communication channels. Tab. 1.2:3 then also pinpoints the most important control and adaptation loop functionalities in automotive software. These software pieces aim at crucial functionalities that guarantee exhaust-gas norms compliance, engine-protection features, components' aging compensation, as well as security features. Most of these control-loops also have learning procedures (adaptation), in that the control-parameters are saved during operation, to allow for future faster control learning and precision.

Typical CHASSIS functionalities	***Typical ENGINE functionalities***
Lights control and diagnosis	Starter control
Steering direction detection	Throttle-pedal and -valve evaluation
Wheels' speed calculations	Engine speed evaluation
Anti-Blocking System (ABS[*])	Engine load calculations
Anti-Slip Control (ASC[†])	Injection calculations and valve control
Motor-Schleppmoment-Regelung[‡]	Ignition calculations and coil control (gasoline-only)
Electronic Stability Program (ESP[§])	Knock detection and control
Clutch detection and control	Turbo-charger calculations and control
Automatic gear-box control	Camshaft control
Active suspension control	Intake manifold control
Tire pressure supervision	Lambda mixture control
Steer-by-wire	Catalyst filling/emptying control
Air-conditioning control	Coolant temperature control
Active Cruise Control (ACC)	Electronic throttle (gasoline-only)
Tire Pressure Control (RDC[**])	General component protection
Electric roof and windows' control	General engine diagnosis
General chassis diagnosis	Etc…
Electronic brake assistance	
Battery management	
Etc…	

Tab. 1.2:2 – Some of the automotive functionalities that must be controlled by the software inside the ECU(s)

Typical CHASSIS control-loops	***Typical ENGINE control-loops***
ABS control and adaptation	Idle-speed control and adaptation
ASC control and adaptation	Knock events control and adaptation
ESP control and adaptation	Turbo-charger control
Electronic brake assistance	Lambda mixture control and adaptation
Etc…	Etc…

Tab. 1.2:3 – The most important "control-loop" functionalities to be tamed by software inside the ECU(s)

Fig. 1.2:4 shows a full-featured BMW "M" V10 gasoline engine. Fig. 1.2:5 then shows a simplified illustration of a vehicle's gasoline engine-management architecture, with the most important sensors and actuators which are readout and controlled by the ECU, respectively. Depending on the engine's fuel type, vehicle type and model, there can be more than 100 sensors and actuators all around. Fig. 1.2:6 explicitly shows a common-rail system and also a lambda mixture and catalyst control example.

[*] *Safety electronic system which processes wheel speed sensors to avoid the wheels to block while braking. This is done by getting brake pressure off of the affected wheels, so that steering and lane-following is still allowed.*
[†] *Safety electronic system which processes wheel speed sensors to counter-act slipping while the vehicle is strongly accelerating. By continuously acting upon the engine's torque, optimal acceleration grip may be obtained. Also called ASR, DTC, TCSS, TRC, TCS, etc.*
[‡] *In English: Engine-Transport Torque-Control. This often not mentioned safety electronic system avoids wheels from slipping due to the driver from releasing the gas pedal or the clutch too quickly, specially on bad grip conditions such as on an icy road. ASC usually contains this control.*
[§] *Safety electronic system invented and developed at Robert Bosch GmbH as an ABS extension which processes acceleration and orientation sensors, in order to guarantee vehicle trajectory stability according to the driver's desire, even in under-steering, over-steering and general squid conditions. This is done by selectively breaking wheels thereby counteracting upon vehicle squid. Also called Electronic Stability Control (ESC), Dynamic Stability Control (DSC), VSA, PSM, MSP, DSTC, etc. Most usually, this system also encompasses the functionalities of ABS, and ASC.*
[**] *"Reifendruckkontrolle"*

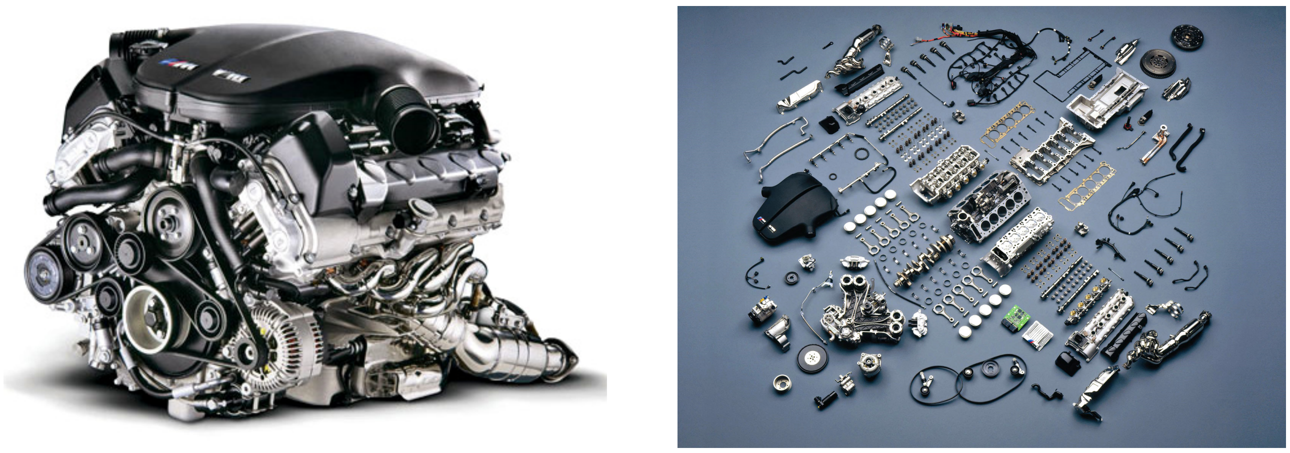


Fig. 1.2:4 – BMW “M” V10 engine with all its components (source: BMW)

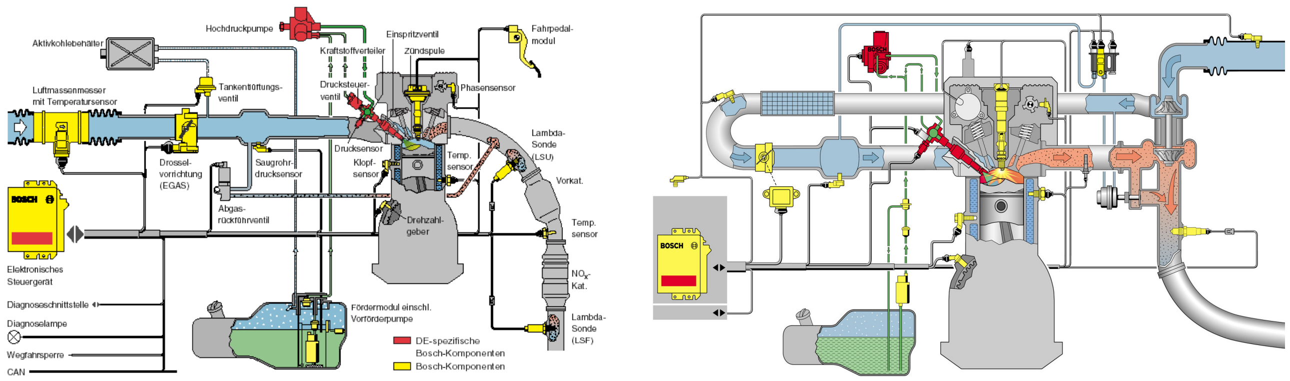


Fig. 1.2:5 – Simplified gasoline engine-management systems with ECU (source: Bosch [93])

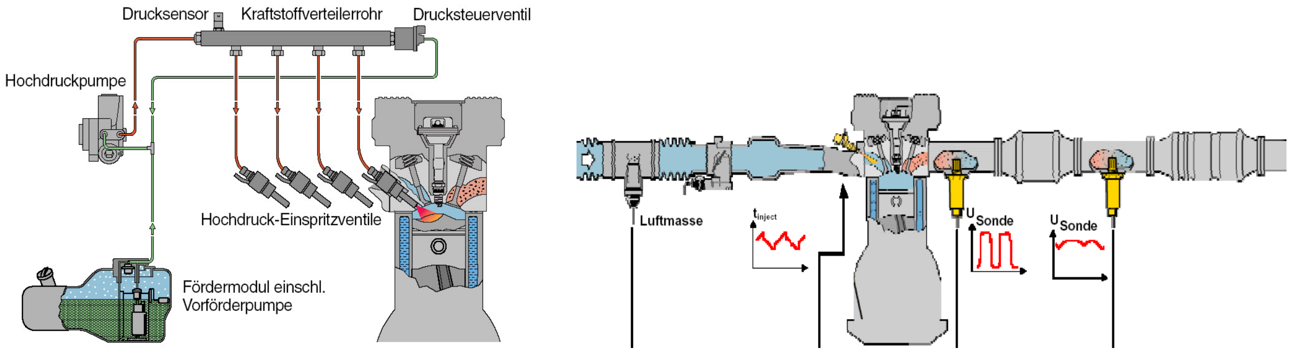


Fig. 1.2:6 – Simplified common-rail system and lambda mixture and catalyst control (source: Bosch [93])

Like in any modern control system, the automotive ECU is comprised by one or more CPUs and by the remaining electronic interfacing circuits for sensors (signal-conditioning) and actuators (power-driving). Normally, central memory units (FLASH and RAM) hold the program and the variable values, respectively. This architecture is depicted in Fig. 1.2:7 and Fig. 1.2:8, where in the middle schematic the sensor inputs and signal-conditioning are depicted on the left half, whereas actuators outputs and power-drivers are depicted on the right half. In the middle of this schematic lies the processing core and corresponding FLASH and RAM memory units. For various external peripherals and programming/debugging operations, several communications interfaces are also encompassed in this ECU hardware.

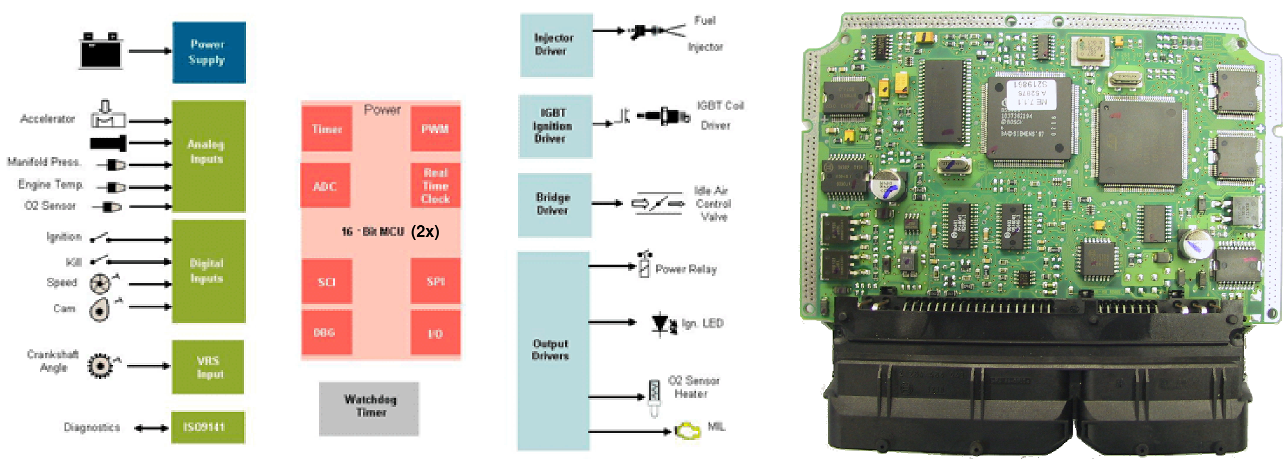


Fig. 1.2:7 – Internal view of a Bosch ME7.1.1 ECU and internal equivalent architecture (source: Bosch [93])

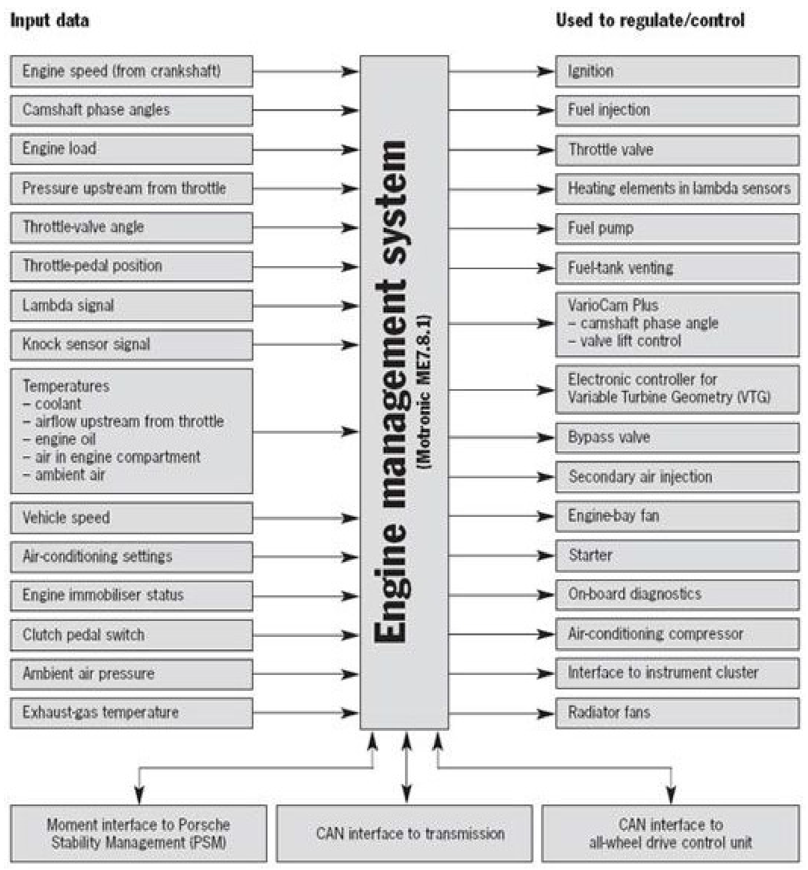

Input data
Used to regulate/control
Engine speed (from crankshaft)
Camshaft phase angles
Engine load
Pressure upstream from throttle
Throttle-valve angle
Throttle-pedal position
Lambda signal
Knock sensor signal
Temperatures – coolant – airflow upstream from throttle – engine oil – air in engine compartment – ambient air
Vehicle speed
Air-conditioning settings
Engine immobiliser status
Clutch pedal switch
Ambient air pressure
Exhaust-gas temperature
Engine management system
(Motronic ME7.8.1)
Ignition
Fuel injection
Throttle valve
Heating elements in lambda sensors
Fuel pump
Fuel-tank venting
VarioCam Plus – camshaft phase angle – valve lift control
Electronic controller for Variable Turbine Geometry (VTG)
Bypass valve
Secondary air injection
Engine-bay fan
Starter
On-board diagnostics
Air-conditioning compressor
Interface to instrument cluster
Radiator fans
Moment interface to Porsche Stability Management (PSM)
CAN interface to transmission
CAN interface to all-wheel drive control unit

Fig. 1.2:8 – More detailed view of Bosch's ME7.8.1 internal equivalent architecture (source: Porsche [142])

This ECUs' hardware is programmed through commercial and proprietary tools which can be text-based imperative programming, as well as graphical high-level programming environments. These environments allow the automotive Engineers to conceive, program, debug and test the functionalities on the corresponding vehicles' ECUs. It is exactly these environments that are going to be thoroughly discussed and targeted by enhancement proposals, in this thesis.

The characteristics of the internal software of these ECU systems can essentially be defined as data-flow in nature but control-flow in implementation. In other words, many sensors' values are inputted to the ECU, processed by a CPU programmed in classical control-flow imperative languages such as "C" or automatic "C"-code generation by usage of visual tools. Strict engine-related functionalities are very heavily data-flow based due to the many sensors which are processed through functionality layers up to the outputs to the few actuators. Chassis-functionalities are also essentially data-flow based but contain more algorithmic code than the previous engine functions. Multimedia and entertainment software is a totally new arena of activity and requires different hardware and software structures. This thesis only addresses the first two areas of activity as a single development paradigm. It will also address the highly data-flow based nature of automotive software, which is not consistently taken from one end (development environment) to the other (hardware platform), as it will be shown later on.

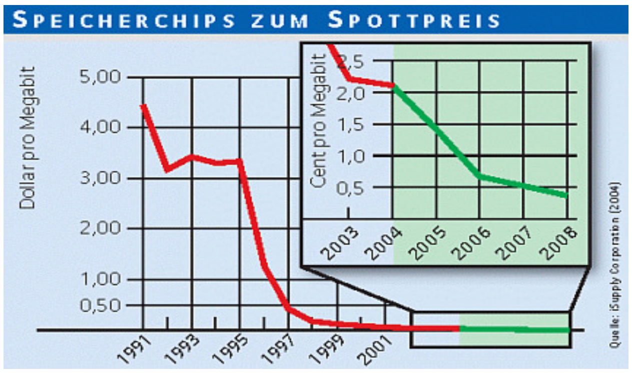
Resulting/needed current automotive control and development systems used in the most renowned car and also in the car-component manufacturers are complex systems which still use classical imperative textual programming techniques and standard handling methods for both development and general control afterwards. The corresponding tools, although being state-of-the-art themselves, centre upon historically layered-grown complex software tool-chains and on correspondingly inflexible hardware platforms. There is a kind of mutual support for the existence of such structures, since software relies on existing hardware and, on the other hand, hardware relies on existing software as well, thereby creating a bi-directional relationship. Very sophisticated visual tools have been emerging, which allow to optimize development efforts [1] [2] at the top-most design tasks of automotive control-functionalities for both engine and chassis. Nevertheless, these systems still rely, at some internal level of their structural assembly, on classical mechanisms such as compilers and assemblers. These lower-layered tools in turn use standard textual imperative languages such as mainly "C" for building machine-code for downloading onto standard and rigid micro-controller based hardware. Although the top-most interfaces are now increasingly visual, even such state-of-the-art tools still generate "C" code for use by lower-level components [3]. In other words, since the compilers and assemblers do not cease to exist by a mere action of will and because they are so mainstreamed, almost possessing the status of "unavoidable", everything new is forced to build on top, underneath or even aside of them. Examples are integrated complex auto-code generators [9] and 3rd-party disruptive debugging tools [134] [135].

1.3 Brief Automotive History

[Emphasis: increase in electronic/software components]

A brief history of evolution of automotive systems is a key-point in the discussion of the pitfalls and limitations of currently used state-of-the-art software development environments and hardware deployment platforms in today's automotive industry. This has to do with the fact that those environments and platforms grew historically in the form of "layers" over already existing ones, creating today's systems that are in reality "stacks of layers" of ever-growing complexity and consequent opacity. The effects and consequences of having those "stacks" are detailed and discussed later on.

The ever-growing need for more safety, more comfort, more durability, less consumption and less pollution, is accelerated by technological innovation, environmental regulations, diagnostics demands and escalating competition among car-makers. This continuously calls for more and more chassis- and engine-management software, both in quantity and quality. While in car childhood no software was employed at all, today there are literally mega-bytes of machine-code inside the cars' ECUs. This is due to the fact that computing systems have become progressively less expensive (both microcontrollers and memory devices) [143], more powerful and more flexible, associated with ideas concerning physical protection of components and safety of passengers (diagnosis algorithms, ABS, ESP, airbags, etc.). Most early mechanical parts of systems such as ABS have been substituted by software, being much more efficient and safe than those purely mechanical counterparts. Statistics show (left side of Fig. 1.3:9) that cost is still going down, opening way to even more software inside cars. Software thus steadily grows to be an increasingly larger part of a car (left of Fig. 1.3:10).



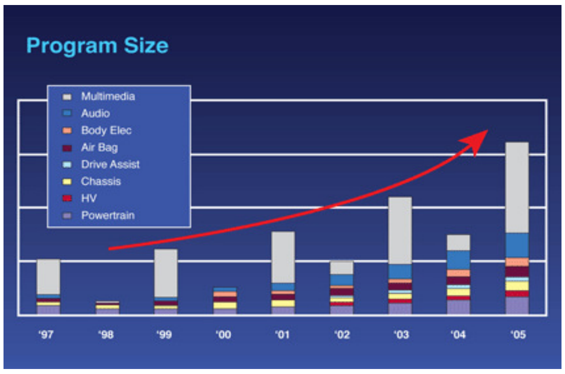
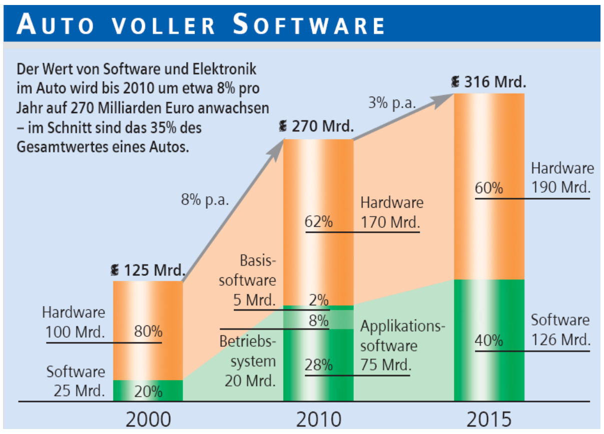


Fig. 1.3:9 – Chronological cost per megabyte and automotive program size (sources: Siemens [143], Toyota)



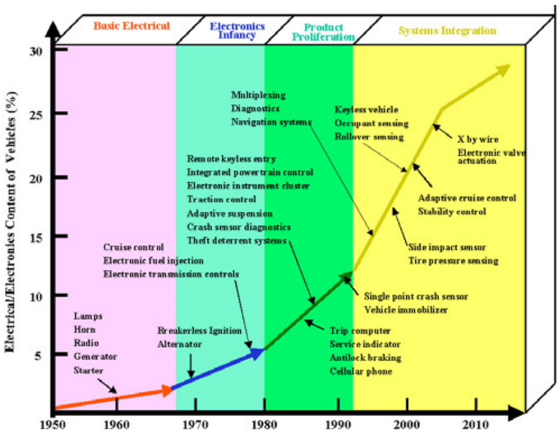


Fig. 1.3:10 – Cost distribution of HW & SW; electronics percent progression (source: Siemens [143] [516])

Fig. 1.3:11 and Fig. 1.3:12 further illustrate this software growth inside critical ECUs in the automobile (not counting with entertainment software). Notice the ESP market penetration increase in a single year (2003-2004). All this combined with challenges related to shorter development times, rising cost pressures and mounting demands for reliable high-quality products, leaves almost no room for the possibility of over-hauling the entire systems. The need and effort to maintain those systems running, consumes all available resources, while calling even more resources on a daily basis. It is also known, as said in [144] [227], that highly innovative novelties in mechanical components are not quite expected, while about 80-90% of innovation in the automotive scene boils down to electronic (10% thereof) and software (90% thereof) novelties. According to the same source, 30 to 50 ECUs are present in middle-sized cars, while premium cars integrate about 80 ECUs in their electronics/software network.

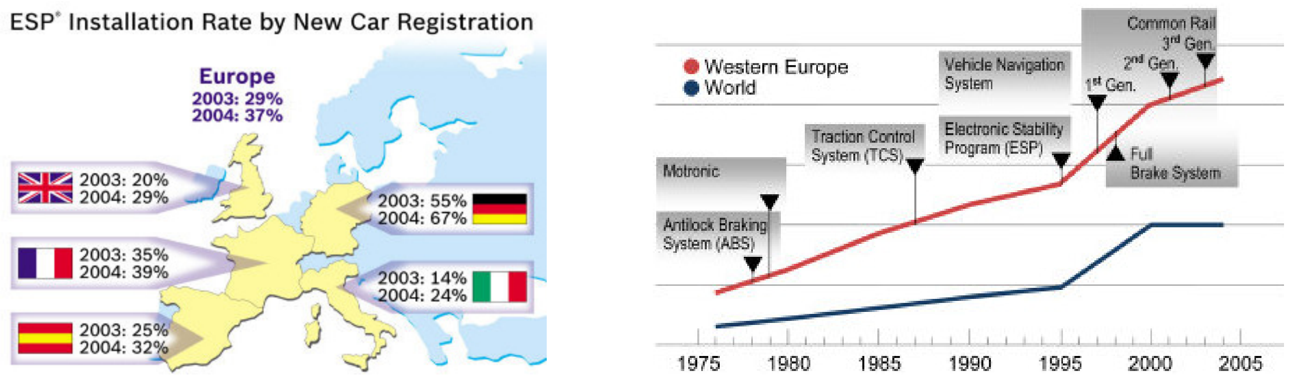


Fig. 1.3:11 – Statistics of software growth inside cars (source: Bosch [93])

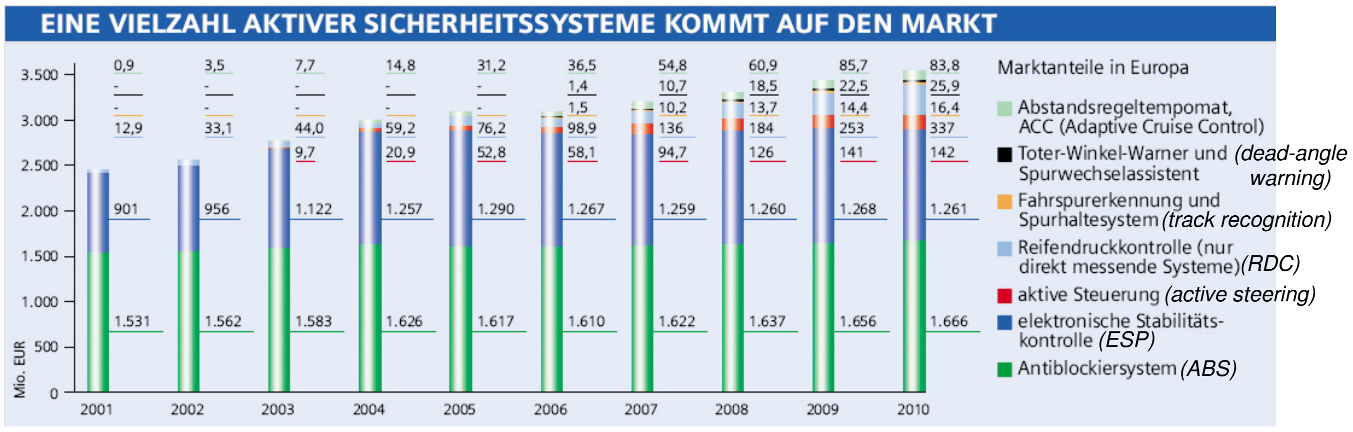


Fig. 1.3:12 – Statistics for chronological market share of electronic systems in cars (source: Siemens [144])

Fig. 1.3:13 shows a simple comparison between the all-mechanical and strictly electrical components of a vintage VW vehicle versus with more recent vehicles with more and more electronic components which potentially contains software. The Volkswagen Phaeton even contains 45 independent ECUs with kilometres of interconnections and interfaces between those ECUs (up to 3860m/64Kg in total length/weight, with 2110 interfaces).

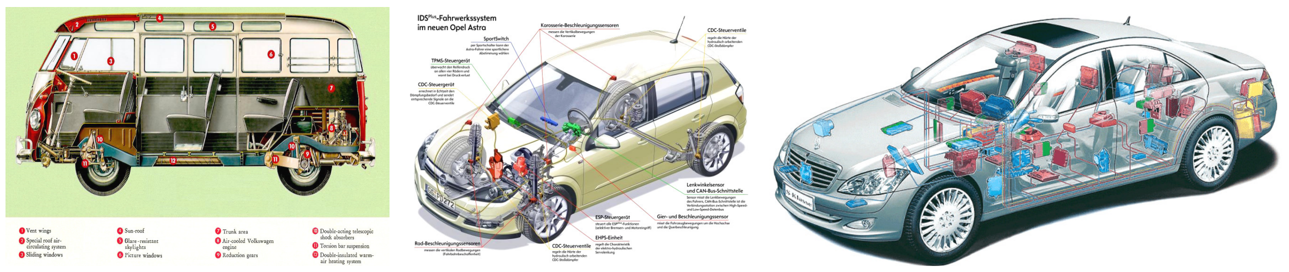


Fig. 1.3:13 – Comparison between vintage VW, mid-sized Opel & Mercedes S (sources: VW/Opel/Mercedes)

A few years ago, it was also predicted that in 2010 about 35% of the value of an average car corresponds to software and its electronics [143] (again Fig. 1.3:10). Long-range predictions [160] even say that in about 2020 up to 50% might be electronics/software in conventional vehicles, while hybrids may even reach 80%. While a regular software application has swollen from about 100.000 lines of code to about 1.000.000 lines in only two years (from 2002 to 2004), predictions for the next decade surely point to explosive numbers in this matter. To have a glimpse of how automotive products' software surpasses even the most advanced and sophisticated military as well as civilian transportations, see Tab. 1.3:14.

Product	***Lines of Code (LoC)***
Power Plant	≈ 100.000
F-22 Raptor Jet Fighter	≈ 1.700.000
F-35 Joint Strike Fighter	≈ 5.700.000
Boeing 787 Dreamliner	≈ 6.500.000
Premium-class automobile	close to 100.000.000

Tab. 1.3:14 – Software content of military transportation products, compared the automobile (source: [160])

An even more impressive comparison: if we exclude the in-flight onboard entertainment system, the new Airbus A380 contains only a few more ECUs than the Mercedes-Benz S-Class. Even low-end cars now have 30 to 50 ECUs embedded in the body, doors, dash, roof, trunk, seats, and just about anywhere else the car's designers can think to put them. In today's premium cars we can find up to 2000-3000 software-related functions, each with many thousands of LoC[*]. While back in 1995 the total amount of embedded software resident in one high-end OEM vehicle passed the level of 1.000.000LoC, in 2005 this already astonishing count increased to an equivalent to 100MByte of binary code [162] [161] that must be downloaded to all the about 70 ECUs inside an high-end OEM car. In 2010 it was even expected that this volume would increase to a whooping 1GByte [162] [161]! This volume is what currently lurks in a PC's memory while working on the applications. Most of the car's innovations relate to software components which produce innovative functionalities, some of them invisible and others visible to the user [162]. Just for the sake of historical comparison, the Motorola M6802 used back in 1981 for all GM's ECUs ran "only" about 50.000LoC in total [163]. For an in-depth chronological historical reference and software growth indicator from 1977 to 1982, see an interesting report in [163]. Also deserving to take a look into is early automotive engine-control systems based on micro-controllers, in [164].

[*] *Lines of Code*

Mounting consumer demand for greater comfort, convenience and connectivity has converted cars into mobile computers. However, this growth in software/electronics quantity and complexity was not always followed by clean and well-structured architectures and not even by software development environments. This causes many problems everywhere due to growing complexity and opacity of these systems. Those problems are most often circumvented but not solved, by using even more software layers which should hide that underlying complexity, or by re-engineering some of those layers. Maintaining quality gets more and more difficult by the day. While this "quality" cannot certainly be defined as the appropriateness of the software-producing environments themselves anymore, it must continue to prevail, at any cost, in the outputted ECU software, since it is this software that will have to guarantee continuous safe vehicle control. All these problems in keeping systems working safely, leads to more frequent electronic systems' failures, which in turn can lead to malfunction of the entire vehicle.

The graphics in Fig. 1.3:15 and Fig. 1.3:16 illustrate these car troubles, where it is easy to recognize that electronic systems produce most trouble (software bugs included), and do seem to increase over time. Although some stagnation is going to occur at some point and although early electrical systems also had their natural failures, this was only to show that the more electronics/software the more failures will add on top of those early ones, instead of lowering them. In 1990, for example, only about 15% of the failures where due to "electronic modules" and 20% to sensor malfunctions [145]. The same source states (quote): *"It* (statistics in [145]*) shows that faults in electronic controllers are not very common. Most of those will be attributable to the mechanical construction rather than to the electronics itself"*. In contrast, software failure is often today a reason for automotive manufacturers to spend millions in recalling thousands of vehicles for firmware updates. Exaggeratingly complex development environments, complex final systems and the urge to continuously innovate and keep up with the competition, invariably lead to occasional more or less damaging failures. This is due to the difficulty of validating and testing everything to a 100% rate [2], since the state-spaces of current automotive software models simply get to large to handle. Very complex testing methods such as those described in [226] must be developed and used. This currently dual complexity brings with it reliability issues. IBM claims [164] that approximately 50% of car warranty costs are now related to electronics and their embedded software.

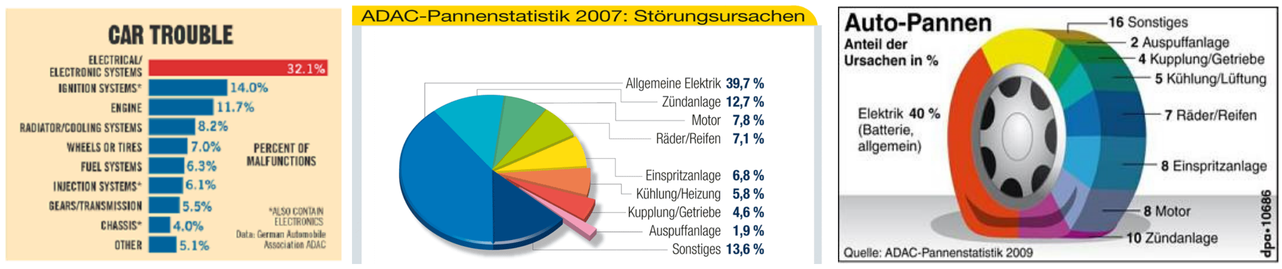
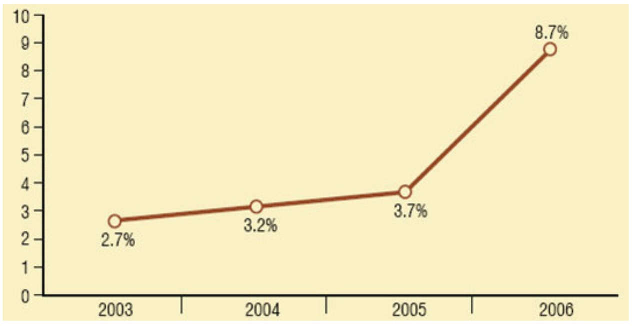


Fig. 1.3:15 – Electronics' car trouble once near 30% now in the years of 2007 and 2009 (source: ADAC [94])



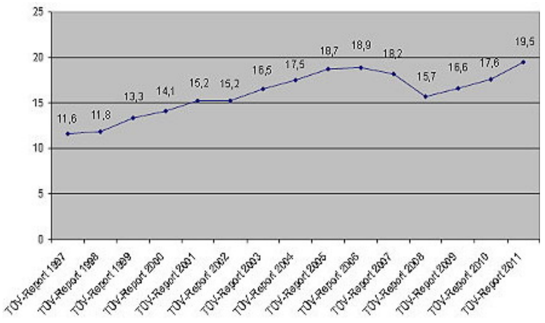


Fig. 1.3:16 – Percentage of recalls due to software failure (source: NHTSA [147], TÜV [403])

Just to give an example of what a massive recall means, in 2005 Toyota voluntarily recalled 160.000 of its 2004 and some early 2005 model year Prius hybrids because of a software problem that caused the car to suddenly stall or shut down. The time needed to repair the software was estimated at about 90 minutes per vehicle, or about 240.000 person-hours. It is even estimated [164] that over 50% of all ECU replacements are unnecessary and only done due to mechanics not knowing how to repair them (the ECUs themselves are technically error-free). Even worse effects coming from software failures are known to be of really disastrous proportions, both in personal and financial losses [261]. One more critical example was given by Toyota [307], where the lack of a very simple software safety feature caused fatal accidents.

In conclusion, current automotive control and development systems used in the most renowned car and car-component manufacturers are systems which mostly use classical imperative programming techniques and standard handling methods, centred on historically layered-grown complex software tool-chains and on correspondingly inflexible hardware platforms. Although more and more of the automotive systems are being developed with help of more sophisticated visual tools which allow optimizing development efforts [1] [4] [5] [6] [7], all of these systems still rely on conventional ways of building and programming functionalities with the help of classical text-based imperative programming paradigms associated to standard micro-controller-based hardware. Thus, in lower-levels, even within highest-level tools [3], standard code will still be generated and used.

1.4 Automotive Systems Nature

[Emphasis: brief/superficial presentation of automotive systems]

A common concept among all of the automotive systems is the closed control-loop comprising the ECUs which contain the control-algorithms (software) and the plant to be controlled (car). Control-signals and sensor-signals represent the data-flow between the ECU and the car components. Since most of the automotive engine/chassis management functionalities/algorithms rely on standard signal-processing and/or data-flow programming, most modern automotive and industry-related software development tools take advantage of those facts and concentrate on a highly visual data-flow programming paradigm [4] [5] [6] [7], avoiding the hassle of more classical low-level textual control-flow programming. Even more standard development tool-chains which make use of classical control-flow programming tools use internal data-flow mechanisms such as time-raster containers with well-defined data-oriented execution sequences, which fully mimic this data-centred automotive functionality processing nature. This data-flow structure is illustrated in Fig. 1.4:17, where a minimum engine control structure is depicted. With most of the feedback paths materialized through the engine and chassis systems themselves, these control structures are themselves mostly feedforward and straightforward regarding natural complexity. Another natural characteristic of automotive systems is that most of the functionalities can be looked at separately, which allows them to be processed in parallel with little or no inter-communication. It is mostly the fact that ECU hardware only contains one CPU that limits this reality to a sequentially processed overall program.

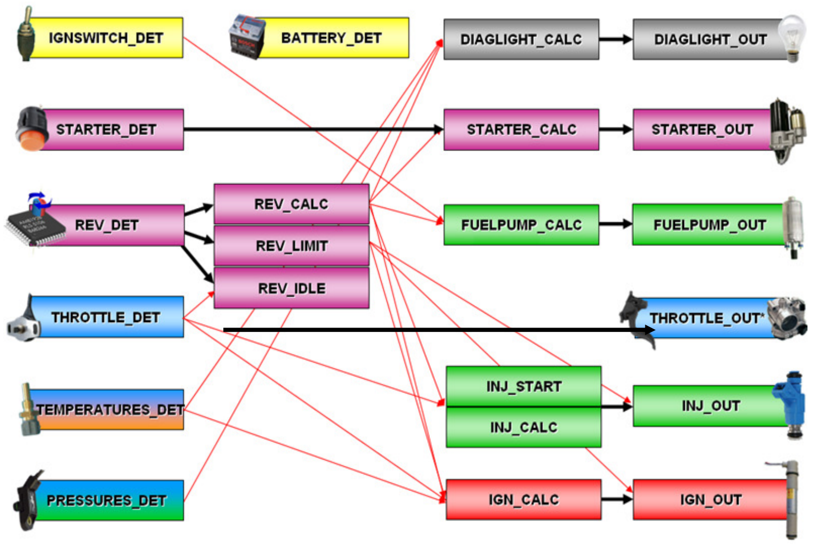


Fig. 1.4:17 – Essential control structure for a gasoline internal combustion engine (source: [Att. 5])

Unfortunately, the hardware solutions and even the internal tool mechanisms did not strictly follow this data-flow nature as closely as the visual software tools did. In fact, they still strongly reflect classical control-flow structures, therefore producing conceptual mismatches between software nature and target hardware reality. Later Fig. 2.1:28 depicts the internal software build structure of an automotive tool, where it can be easily seen that the graphical interface designs are automatically converted into standard "C" code, to finally land into standard hardware processing targets. Fig. 2.1:27 shows that although most ECU software is data-flow oriented, control-flow details are necessary for some algorithmic constructs such as the sequencing of the engine-start procedure. Most of automotive functionalities are data-flow oriented in nature, but are mandatorily converted to imperative "C" style, since existing hardware platforms all work this way.

The result is most often a non-coherent internal tool software building structure and the underlying "driving" or "interfacing" software structure for the application software to be able to run on the hardware target, as both depicted in Fig. 1.4:18. Although one may argue that these herein depicted structures are the way they are due to the need of all their internal components, the complexity arises exactly from the fact that all those components are very heterogeneous among them and therefore add on interfacing complexity on top of the naturally needed complexity level of such automotive systems. Additionally, Fig. 1.4:19 shows a motorsport ECU and its high-level engine map calibration software. MOTEC ECUs even come with an advanced software package that allows to change some algorithms inside the ECU, through a "C-like script language" [419], which does not exist in any other motorsport or even series vehicle development software suite.

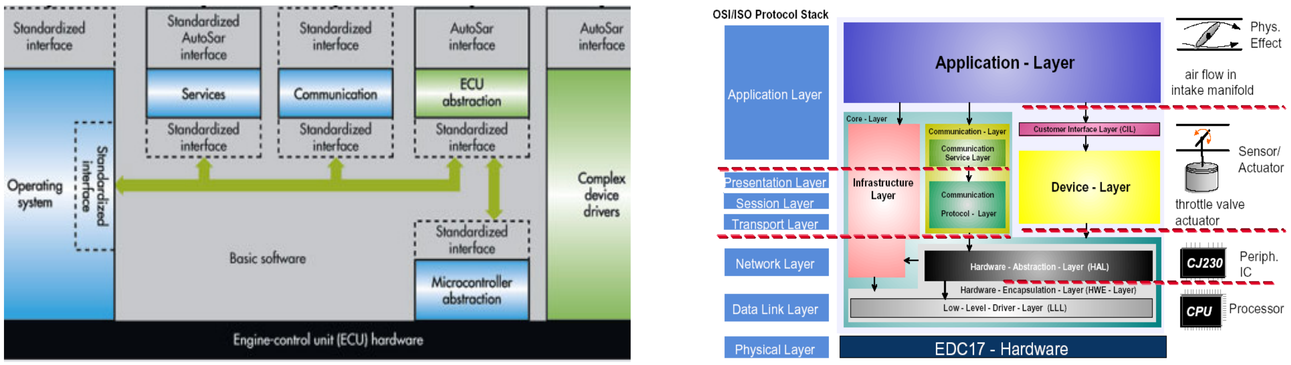


Fig. 1.4:18 – Illustrations of internal Autosar and EDC17 ECU structures (source: Autosar [67], Bosch [93])

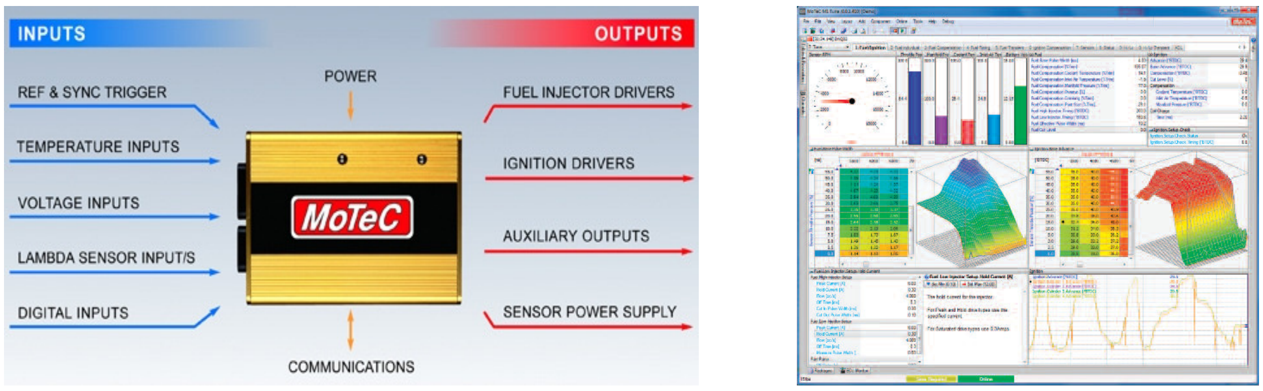


Fig. 1.4:19 – Motorsport ECUs from MOTEC and calibration software illustration (source: Motec [381])

Besides the components that make up an automotive system, be it software or hardware, there is one absolutely unavoidable issue to be taken into account, whatever implementation is done: the typical hard-real-time processing constraints. In modern automotive systems, both injection and ignition, among other things such as knock-control, among others, must be precisely timed or else the engine may be destroyed! Thus, very strict timings must be accomplished, which most often bring highly complex challenges to the developers of the real-time operating system underlying the whole software architecture of an automotive ECU (see Fig. 1.4:20). Automotive systems are in fact one of the most demanding in the issue of guaranteeing real-time constraints, therefore being usually placed in the "hard-real-time" type of real-time systems.

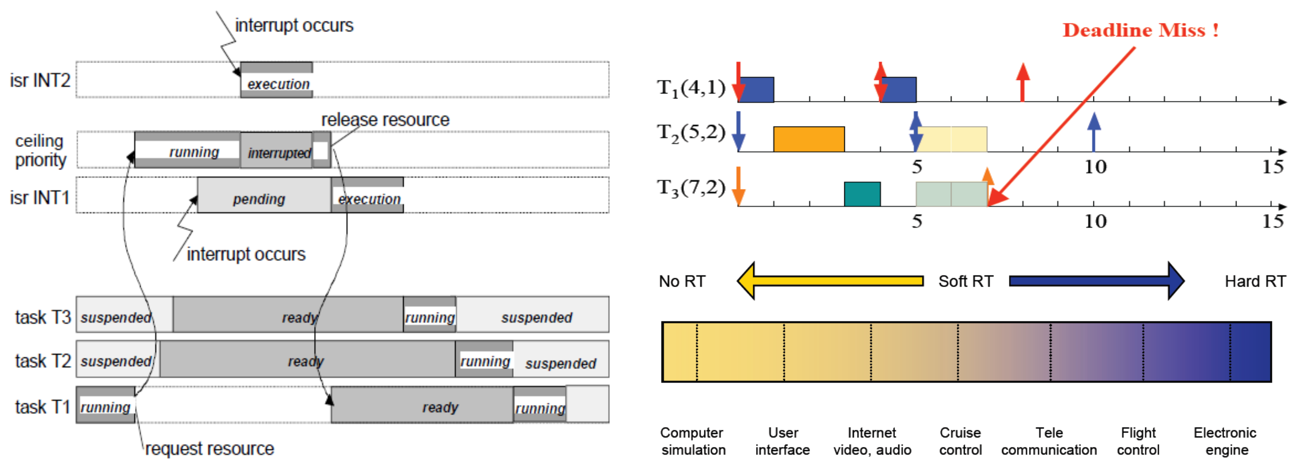


Fig. 1.4:20 – Hard-real-time issues affect automotive engine/chassis control systems (source: [446] [447])

Consequently, real-time processing is the most important factor for an automotive program to do its work, closely followed by safety issues during that processing. All in all, a list of the most important features of an automotive program would look like this, from higher ("C" → critical) to lower ("N" → nice-to-have) priority features:

- **Real-time execution** $^{(C)}$ → It is absolutely mandatory that some automotive functionalities execute in real-time, that is, execution must be in a way that allows the automotive peripherals (sensors and actuators) to be read-out and commanded within the strictly needed time intervals. Else, the entire system would totally malfunction due to engine and/or chassis timing constraints not being complied with. Examples are injection valves that must inject fuel at exact moments and ignition coils that must release sparks at exact moments, or else the engine will either stop, be damaged or not deliver the desired power. Not everything relates to time directly but rather to crankshaft angle: most engine-related "real-time" events have to do with actions that have to be taken at certain crankshaft positions. The faster the engine revs, the faster the ECU must act upon the actuators. In this context, an appropriate "real-time" operation means being able to make all the necessary calculations up until right before having to perform those actions.
- **Safe execution** $^{(C)}$ → Since automotive vehicles carry people and can potentially cause accidents with injuries or even fatalities, the program running inside the ECU must be as secure as possible. The hardware of this ECU must also be reliable enough. All this relates to the ECU's actions upon the actuators, which should not be such that the vehicle behaves badly. Examples are injection and ignition that must not cause the engine to stall or crash in the middle of a highway at 120Km/h or more, ABS and ESP systems that must not cause the wheels to block or to erratically brake and cause the vehicle to spin, and the automatic gearbox system must not turn havoc and self-destroy, the engine or the vehicle altogether. The higher the risk, the more important this feature gets, such as happens in airplanes and military vehicles. Testing is therefore an issue of supreme importance in the automotive software and hardware scene.
- **Functionality range, Comfort and others.** $^{(N)}$ → Automotive functionalities such as injection and ignition are basic needs for an engine, but there are many others that are add-ons to minimal vehicle management. Examples are ABS, ESP, knock-control, lambda-control, traction-control, air-conditioner control, cruise-control, among many others. All these functionalities add to the comfort of the vehicles' driver and remaining occupants but are not strictly necessary, although international regulations now rule for quite a range of obligatory equipment and respective software.
- **Components longevity** $^{(N)}$ → Last but not least, if at all possible, the ECU should contain "component protection" software to protect the life of various components if some critical component-threatening situation occurs. Examples are exhaust path sensor protection against too high temperatures (engine power limitation), engine protection against wear-out when its oil and cooling water are still cold (power limitation) and saving tire wear-out through slippage limitation when accelerating (engine power control).

1.5 List of Pitfalls

[Emphasis: concise list of all identified pitfalls in one place]

Table Tab. 1.4:21 below shows so-called “pitfalls” present in state-of-the-art development systems and hardware platforms. These are going to be addressed throughout this thesis and refer to problems, limitations and general handling difficulties. Most exist since the very beginning of computing history and were never solved or simply engrave through sheer habituation. Developers and users cope with those pitfalls because they were there all the time and because large efforts in both time and money paths would be necessary to eliminate them. Most pitfalls are even regarded as a “necessary evil” for current systems to even exist. In other words, they are considered part of the constraints. The proof for this statement is thinkable easy to find everywhere: the flood of validation, test and aiding tools, which precisely aim their capabilities at individual pitfalls. Instead of creating tools aiming to get these apparently unavoidable pitfalls under control, this thesis tries to go the way around: creating a development system and hardware platform combination that mostly eliminates these pitfalls altogether. This thus means eliminating the need for the current tool-flood that bloats already hard-to-handle systems. This pitfall list is one of the success references for this entire thesis. It will be referenced again at the end, where the success of the developed software/hardware system is discussed.

COMPLEXITY PITFALLS		***OVERHEAD PITFALLS***		***HANDICAP PITFALLS***	
PITFALL C#1	√ Arithmetic Multi-Formats	*PITFALL O#8*	√ Coding Reliability	*PITFALL H#74*	√ Intrusive Handling
PITFALL C#2	√ Endianness	*PITFALL O#10*	√ O Code Coverage	*PITFALL H#36*	√ Fatal-Error Recovery, Traps & Resets
PITFALL C#4	√ Lexics & Syntax	*PITFALL O#24*	√ Branch Prediction	*PITFALL H#47*	√ Floating-Point Discrepancies
PITFALL C#11	√ Translators & their Outcomes	*PITFALL O#25*	√ Superscalar Processing	*PITFALL H#51*	√ Graphical Elements Cluttering
PITFALL C#16	√ Memory Wait-States	*PITFALL O#12*	√ Code Optimization/Compression	*PITFALL H#53*	√ Impossible Reverse-Engineering
PITFALL C#23	√ Pointers & Handles	*PITFALL O#14*	√ Code Tweaking	*PITFALL H#57*	√ Lack of Command-Line
PITFALL C#26	√ Interrupts	*PITFALL O#68*	√ Tool Certification	*PITFALL H#61*	√ Deferred Debugging & Prototyping
PITFALL C#43	√ Time-Raster Dilemma	*PITFALL O#17*	√ Caches	*PITFALL H#69*	√ Tele-Operation
PITFALL C#44	√ Inter-Thread Dependence & Sync	*PITFALL O#19*	√ Stacks	*PITFALL H#71*	√ Peripherals Handling
PITFALL C#30	√ Memory Structure	*PITFALL O#20*	√ Pipelines	*PITFALL H#73*	√ Global Software/Hardware Diversity
PITFALL C#3	√ Word-Alignment	*PITFALL O#21*	√ Flags	*PITFALL H#49*	√ Lengthy Turnarounds
PITFALL C#37	√ Software Aging	*PITFALL O#22*	√ Registers	*PITFALL H#48*	√ Manual/Tedious Graphical Layout
PITFALL C#29	√ Temporary Storage	*PITFALL O#27*	√ Register Renaming	*PITFALL H#50*	√ Tedious Differences-Inspection
PITFALL C#13	√ Source vs. Output Diversity	*PITFALL O#28*	√ Side-Operations	*PITFALL H#60*	√ Costly Live-Feedback/Manipulation
PITFALL C#15	√ Options & Switches	*PITFALL O#38*	√ Operating-System Stealthing	*PITFALL H#62*	√ Execution Profiling
PITFALL C#58	√ Building Error-Messages	*PITFALL O#39*	√ Intermediate Representations	*PITFALL H#63*	√ Mixed Simulations
PITFALL C#59	√ Runtime Error-Messages	*PITFALL O#42*	√ Supervision	*PITFALL H#66*	√ System Testing
PITFALL C#46	√ Timing-Jitter Effects	*PITFALL O#84*	√ Over-Engineering	*PITFALL H#75*	√ Difficult Scalability
PITFALL C#9	√ Bug Searching	*PITFALL O#40*	√ Pre-Processing/Macro-Expansion	*PITFALL H#70*	√ Online human support
PITFALL C#64	√ Side-Effects & Bug Propagation	*PITFALL O#54*	O Multiple Software File Formats	*PITFALL H#34*	√ Too many Coding Alternatives
PITFALL C#45	√ Multitude of Different Timers	*PITFALL O#67*	√ Code Metrics	*PITFALL H#76*	O Environment Installation
PITFALL C#41	√ Hygienic Programming	*PITFALL O#65*	√ Execution Repeatability	*PITFALL H#35*	√ Architecture & Handling Non-Orthogonalities
PITFALL C#80	√ Long Learning-Curves	*PITFALL O#32*	√ Multi-Core Power	*PITFALL H#52*	√ Binary data in files
PITFALL C#5	√ Operation Priorities & Sequencing	*PITFALL O#72*	√ Peripherals Interfacing	*PITFALL H#82*	√ Tool Schizophrenia
PITFALL C#6	√ Semaphores	*PITFALL O#33*	√ O Verification of Correctness	*PITFALL H#77*	√ 3rd-Party Tools & Plug-ins
PITFALL C#81	√ Overwhelming Manpower & Doc. Needs	*PITFALL O#79*	√ Processing Focus	*PITFALL H#78*	√ Development vs. Deployment Modes
PITFALL C#85	√ Complexity Gap	*PITFALL O#83*	√ System Debugging	*PITFALL H#56*	√ Missing/Impossible "upward path"
PITFALL C#7	√ Time-driven vs. Event-driven	*PITFALL O#18*	√ Memory Access Hashing	*PITFALL H#55*	√ Abstraction Losses
PITFALL C#31	√ Multi-Assignment	√ = herein eliminated pitfall			
		O = *herein partially eliminated pitfall*			
		X = *herein not eliminated pitfall*			

Tab. 1.4:21 – Full list of herein identified pitfalls found in most state-of-the-art development systems

Current systems compare to patients with “chronic/genetic diseases” and “bad habits” (the pitfalls), needing constant “medication” (the 3rd-party and general aid-tools) to stay alive (the currently much-appreciated “keep it working” paradigm) under a pseudo-controlled state of operation. Illustration Fig. 1.5:22 below shows this reality in the context of the automotive software development and hardware testing system fronts. In other words, today’s automotive development, deployment and testing tools generally need a whole lot of auxiliary extras around them for them to work without immediately crumbling down.

These may be proprietary or 3rd-party, as well as auxiliary aids developed by the clients (developers and users) themselves. Many companies start up due to the extra need for all these software and hardware aiding systems themselves.

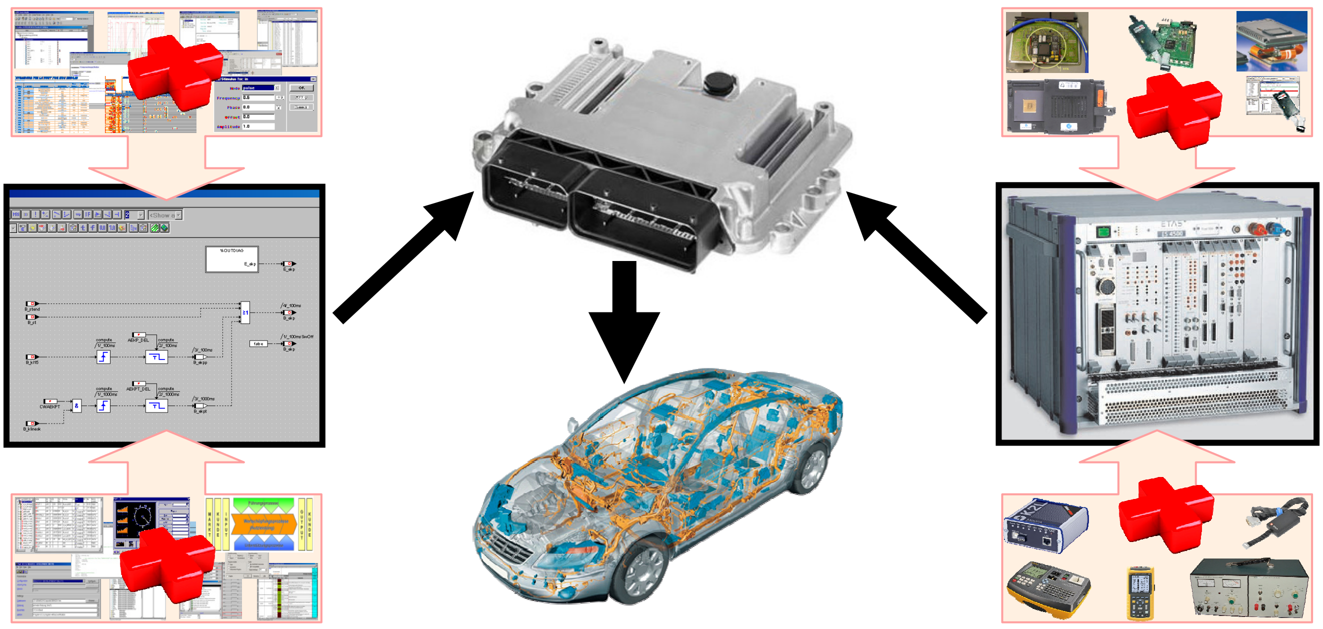


Fig. 1.5:22 – Current "patient-like" medicated automotive system with its software & hardware development fronts

Another detail to be mentioned is that there are at least four ways of dealing with these pitfalls. The first three ways are actively used virtually everywhere today and will continue to be used. The forth way embodies the attempt done in this work developed herein. This is illustrated in the depiction Fig. 1.5:23 below all these four points:

- One can live with the pitfalls by learning to live and to avoid them actively and constantly. This is still the case in just some pitfalls and only for textual tools. The problem here is that one has to learn the highly complex side-task of avoiding pitfalls and to maintain that knowledge active at all times. Not realising this can be quite disastrous.
- One can go completely around those pitfalls. This is the case for 3rd-party tools that circumvent the complete development chain and access hardware directly. The problem here is that those paths are completely separated while synchronization must be manually maintained at all times.
- One can hide the pitfalls under the hood of extra layers and visual interfaces. This is what happens with the vast majority of pitfalls and this is where the majority of current visual tools build onto. The user is fooled into thinking that the path is straight and without barriers. The problem here is that those pitfalls did not just disappear and that lurk all the time under those hoods, just waiting to appear at the most unexpected place and time.
- Last but not least, one can eliminate most pitfalls altogether. This certainly is the least considered possibility. Generally, developers and users had to live with many pitfalls in the past and they learned how to keep them under manual control. Historically, these pitfalls were always regarded as being "problems of the trade". On the other hand, more modern and state-of-the-art development tools successively tried to hide many pitfall through extra cleverly applied layers and hoods over the existing software and hardware. Although this makes systems virtually easier to handle, the pitfalls are never really eliminated and may pose unexpected problems. Eliminating pitfalls the intrinsic way, i.e. through a methodology where they simply do not exist since the conception of the system, may be a third path worthwhile pursuing. This is exactly what this thesis' work is all about. The resulting path is thus straight and without lurking problems.

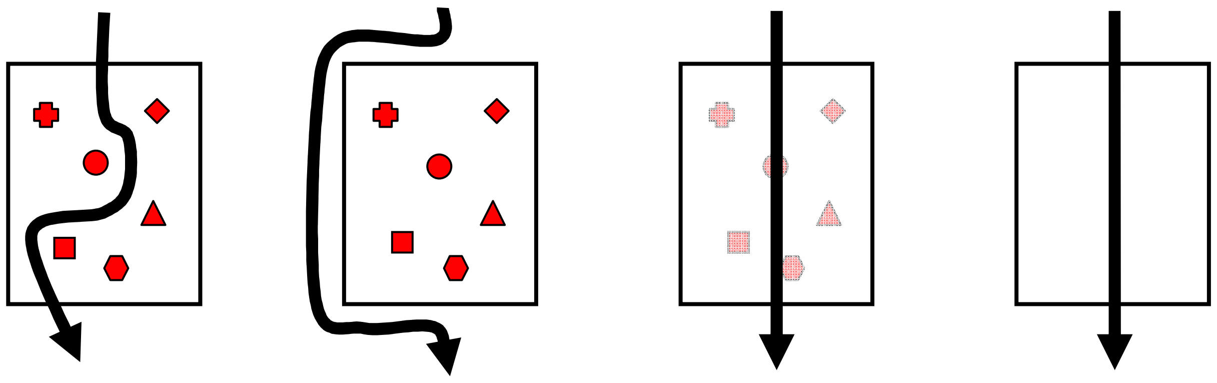


Fig. 1.5:23 – Comparison of historical/current ways of coping with system pitfalls

The importance given to the previously listed pitfalls throughout this thesis is intimately related with their ultimate responsibility for state-if-the-art development systems and hardware platforms being extremely difficult to handle. The usual result is an irritating manifestation of “tool schizophrenia”, where something does not behave as it should and where the developer or user has no idea about what is going on at that very moment. Even though these systems try to hide those pitfalls and present a user-friendly interface, those difficulties are intrinsically present and may manifest themselves at the most inappropriate moments. The only really effective and obvious way to avoid pitfall-related problems is to try eliminating them altogether. That will be the exact purpose of this thesis, after identifying which pitfalls exist in most systems and after devising mechanisms to get rid of them. The most advanced features of the work developed herein, will however be the 100% elimination of some important pitfalls, in a way where they were never there, so that they were never really needed to be eliminated. This “intrinsic” and “never there to start with” elimination will be the exact core of this thesis. To achieve such a result, a completely new idea has to emerge. This idea will be the so-called “core-idea” of this thesis, in the form of the “MACROS” graphical language.

One very important detail that must be made clear here, is that many pitfalls of the previous list will be "intrinsically eliminated" during this thesis' work. In reality, since those pitfalls will never be added to this work, they will never be included here. So, one could say that they were never obfuscated through classical hiding or any automatic mechanism, either. In other words, those specific pitfalls are *"problems not there, by design"*.

1.6 Document Structure

[Emphasis: how this thesis is structured]

Here follows a compact introduction in form of "document thread". This intends to show the main thread of this thesis' work and to show the intended seamlessness of chapter transitions, in which one chapter already introduces the next one, in a path that progressively crystallizes into the final DEMO. A total of 7 chapters are the exact amount needed for this work. The basic flow during this thesis can be synthetically put as follows:

$$\boxed{\textbf{\textit{ANALYSIS}}} \rightarrow \boxed{\textbf{\textit{DEVELOPMENT}}} \rightarrow \boxed{\textbf{\textit{IMPLEMENTATION}}} \rightarrow \boxed{\textbf{\textit{RESULTS}}}$$

1/2. Motivation & extensive research of state-of-the-art solutions ⇒ This is an informative ***analysis*** where the most relevant automotive tools, systems and concepts are presented, briefly explained and compared. Some other non-automotive references are also presented aiming at showing solutions that also contain interesting features that could be of great interest to this thesis' work.

ANALYSIS

3. Problem spots of state-of-the-art solutions ⇒ This is a commented ***analysis*** where existing problems and limitations are pointed out on current systems. While doing that, ***development*** directions are given to serve as guidelines to be followed in this thesis' development work.

DEVELOPMENT

4. Solutions to enhance existing levels of "Easy-Handling" ⇒ This exposes the ***development*** that was necessary to achieve all the desired results, most of which would be the elimination of the problems and limitations identified previously. This development was made along-side with the ***implementation*** of those ideas, so that every single step was always tested "on-the-fly".

IMPLEMENTATION

5/6. *"ECU2010"* project, results and evaluation ⇒ This is the most important milestone of this entire thesis' work, encompassing the ***implementation*** of the theoretical developments as an integrated solution. This solution was then applied to an internal combustion engine and ***results*** were recorded/gathered from the complete and alive system.

RESULTS

7. Conclusions & Future work ⇒ At the end, comments on the ***results*** of the implemented project are made, which also lead to considerations about possible future enhancements that could be done.

Along chapters, attachments are included, containing useful supplementary information about certain topics. These are then referred inside the chapters and their sections, for the reader to jump to, in case more details about those topics are desired. These attachments should thus be optionally read only after reading the corresponding sections.

Page intentionally left blank

Page intentionally left blank

"Problems cannot be solved at the same level of awareness that created them."
Albert Einstein

CHAPTER 2

State-of-the-Art of Automotive Systems

2.1 State-of-the-art software tools

[Emphasis: much has changed, the base is the same]

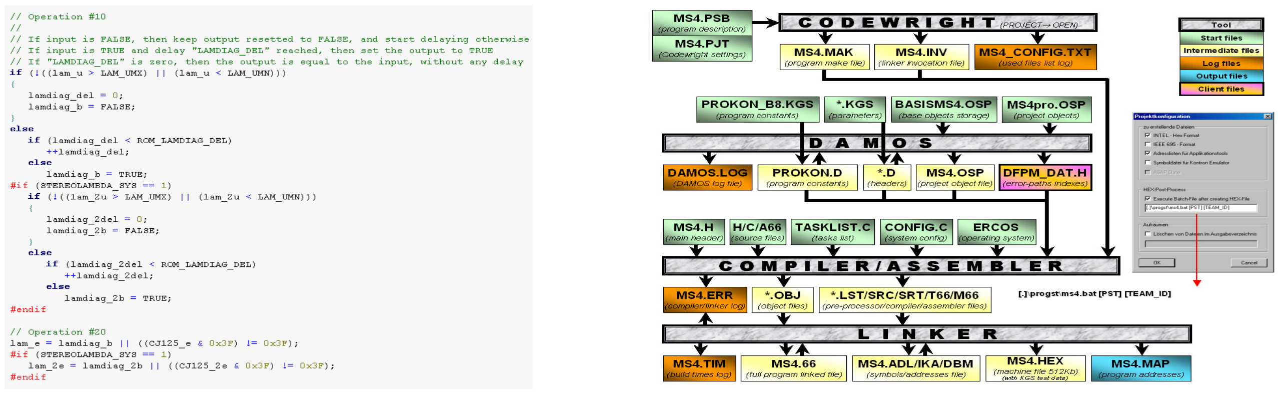
Currently used tools encompass classical ways of programming automotive functionalities into corresponding ECUs, including text-based environments. These are being replaced by more visual and intuitive ones, but at a very slow pace. Hardware platforms also suffer transformations and evolutions, the most important of which is the multitude of testing, validating and debugging/prototyping hardware offers today. While this is the case for testing equipment, the real ECUs built into vehicles suffer much less evolution in term of global architecture, maintaining the classical CPU-centred processing and also requiring massive interfacing/driving layers between control-algorithms and hardware.

Current software development environments and their tool-chains are proprietary and built directly inside car or car-components manufacturers. These often rely on commercial software packages which are adapted to specific needs. While the top-most user-interface level is becoming more visual and better reflecting the intrinsic data-flow nature of automotive functionality development, these tools hide several layers of very high complexity due to the historical growth of new layers on top of the already existing stacks. This phenomenon produces large, very complex and expensive software packages.

2.1.1 Legacy software tools

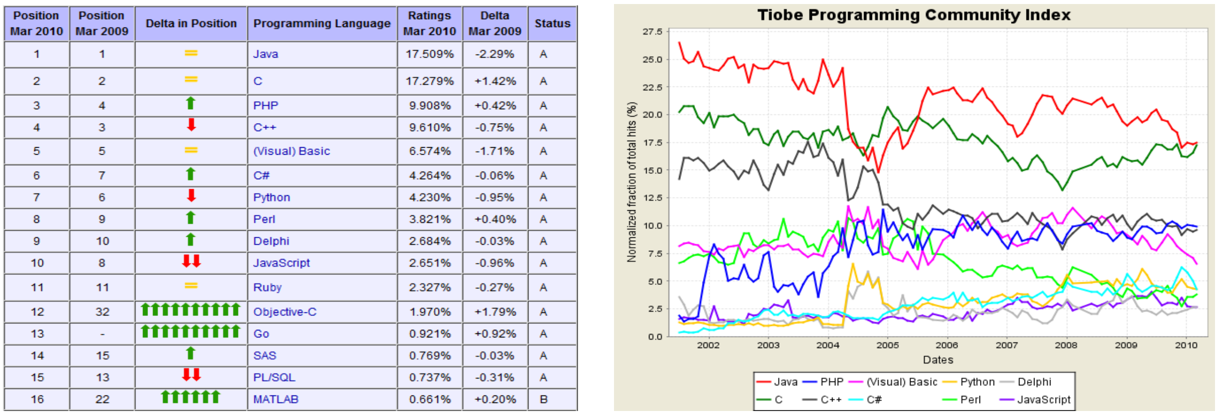
[Emphasis: mainly control-flow programming and processing]

Much of the automotive functionalities programming is still made using standard "C" language and classical compilation developments environments. Fig. 2.1:24 shows an example of such an environment. Besides the text-based source-code and the imperative programming style, there is a complex layered compiling structure behind the code. This structure assures source-code compilation including all the needed libraries and add-ons, until one gets the final machine-code file which is then downloaded into the hardware platform. Although most of the programming is done in "C" language, there are often also components more directly programmed in Assembly language, specially fast-executing and time-critical blocks related to ignition, injection and safety functionalities. Nevertheless, nowadays' faster CPUs are allowing to program almost every detail in "C" language. Although best adapted to the corresponding available ECU hardware platforms, this method is considered "legacy" due to the imperative necessity and to the emergence of visual-oriented tools which try to abstract the user from the underlying complexities of these older tools. The "TIOBE Programming Community Index" [236] rates the "C" language at 2nd place with 17,3% of usage share, increasing in usage and already scratching the JAVA popularity as seen in Fig. 2.1:25. This simply shows that even being considered a too error-prone and even "dangerous" language by some [237], it is clear that it is still one of the preferred ways of programming industrial applications. If it weren't for web- and smartphone-based applications, "C" would be the indisputable rank-winner.



```
// Operation #10
//
// If input is FALSE, then keep output resetted to FALSE, and start delaying otherwise
// If input is TRUE and delay "LAMDIAG_DEL" reached, then set the output to TRUE
// If "LAMDIAG_DEL" is zero, then the output is equal to the input, without any delay
if (!((lam_u > LAM_UMX) || (lam_u < LAM_UMN)))
{
   lamdiag_del = 0;
   lamdiag_b = FALSE;
}
else
   if (lamdiag_del < ROM_LAMDIAG_DEL)
      ++lamdiag_del;
   else
      lamdiag_b = TRUE;
#if (STEREOLAMBDA_SYS == 1)
   if (!((lam_2u > LAM_UMX) || (lam_2u < LAM_UMN)))
   {
      lamdiag_2del = 0;
      lamdiag_2b = FALSE;
   }
   else
      if (lamdiag_2del < ROM_LAMDIAG_DEL)
         ++lamdiag_2del;
      else
         lamdiag_2b = TRUE;
#endif

// Operation #20
lam_e = lamdiag_b || ((CJ125_e & 0x3F) != 0x3F);
#if (STEREOLAMBDA_SYS == 1)
   lam_2e = lamdiag_2b || ((CJ125_2e & 0x3F) != 0x3F);
#endif
```

Fig. 2.1:24 – Example of a standard "C" programming package (source: Bosch [93])



Position Mar 2010	Position Mar 2009	Delta in Position	Programming Language	Ratings Mar 2010	Delta Mar 2009	Status
1	1	=	Java	17.509%	-2.29%	A
2	2	=	C	17.279%	+1.42%	A
3	4	⬆	PHP	9.908%	+0.42%	A
4	3	⬇	C++	9.610%	-0.75%	A
5	5	=	(Visual) Basic	6.574%	-1.71%	A
6	7	⬆	C#	4.264%	-0.06%	A
7	6	⬇	Python	4.230%	-0.95%	A
8	9	⬆	Perl	3.821%	+0.40%	A
9	10	⬆	Delphi	2.684%	-0.03%	A
10	8	⬇⬇	JavaScript	2.651%	-0.96%	A
11	11	=	Ruby	2.327%	-0.27%	A
12	32	⬆⬆⬆⬆⬆⬆⬆⬆⬆⬆	Objective-C	1.970%	+1.79%	A
13	-	⬆⬆⬆⬆⬆⬆⬆⬆⬆⬆	Go	0.921%	+0.92%	A
14	15	⬆	SAS	0.769%	-0.03%	A
15	13	⬇⬇	PL/SQL	0.737%	-0.31%	A
16	22	⬆⬆⬆⬆⬆⬆	MATLAB	0.661%	+0.20%	B

Fig. 2.1:25 – TIOBE rating for the standard "C" programming language (source: TIOBE [236])

2.1.2 State-of-the-art automotive software tools

2.1.2.1 ASCET

[Emphasis: data-flow design, graphical design]

Having been developed at ETAS [92] since the early 90s [26] [298], this "Advanced Simulation and Control Engineering Tool" [7] embodies a graphical editor (Fig. 2.1:26) which allows to visually edit/design automotive functionalities called Function-Definitions (FDEFs). This is done by combining data- and control-flow representations (see also [8] on this matter) by using graphical processing elements and showing them both on the same view. It contains an integrated auto-code generator (see also [9] [27]) and is used for series-cars ECU prototyping, development and mass-production since 2000 with more than 1.700 users, providing code to over 20 million ECUs per year [361]. Furthermore, this tool features SiL (Software-In-the-Loop) capabilities, which allow algorithms and control structures to be tested before deploying them on expensive hardware. Its internal structure and that of compatible ECUs still reflect the classical views of development tool-chains and centralized hardware platforms, by adding the graphical interface on top of existing structures, with the auto-code generator as the interface between the graphical elements and underlying "C"-code specialized classical tools (Fig. 2.1:28) [27].

This internal structure allows to also using "C" blocks as inputs of the whole build process. Starting with ASCET-RS and ASCET-SD versions, ETAS now ships a full range of ASCET tools for every detail of automotive development, being mostly used by Bosch and their clients. Specifically, the latest ASCET-RP package broadens the ASCET scope to rapid-prototyping by integrating the ES1000 prototyping system detailed later. It also allows monitoring simulated variable values on the same modelling interface. Fig. 2.1:27 shows graphical data-flow and control-flow coexistence, in that control-flow is integrated by sequence numbers placed over the operations, allowing setting the exact sequence of processing. This image also shows the use of conditionals (IFs) inside that control-flow.

A visual tool to cope with the need of comparing different versions of the same FDEFs has also been devised with "ASCET-DIFF" [216] shown in Fig. 2.1:29. It is know to work well only in narrow circumstances, which denotes the development difficulty. ASCET is only used for automotive systems so that is not mentioned in the "TIOBE Programming Community Index" [236], falling below 0,05% usage. Last but not least, Fig. 2.1:30 shows the complete development tool-chain using ASCET-SD v4.1 at Robert Bosch.

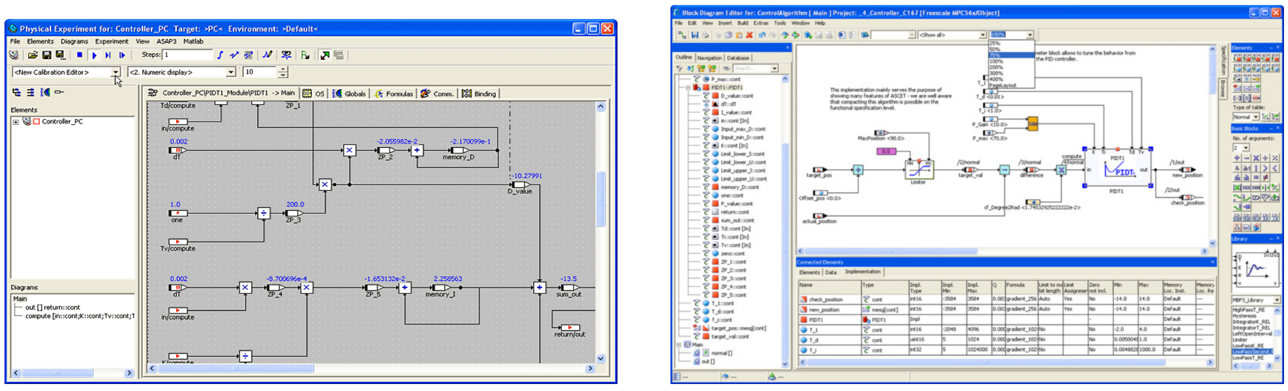

Physical Experiment for: Controller_PC Target: >PC< Environment: >Default<
Block Diagram Editor for: ControlAlgorithm [Main] Project: _4_Controller_C167 [Freescale MPC56x/Object]

Fig. 2.1:26 – ASCET graphical interface programming environment (source: ETAS [7])

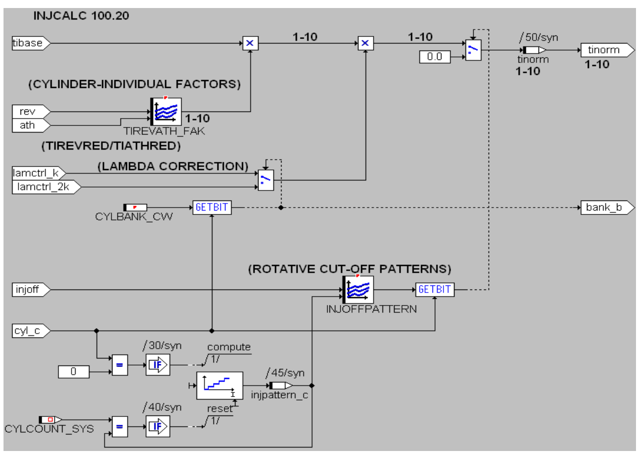


Fig. 2.1:27 – ASCET's data-flow and control-flow graphical coexistence (source: Bosch [150])

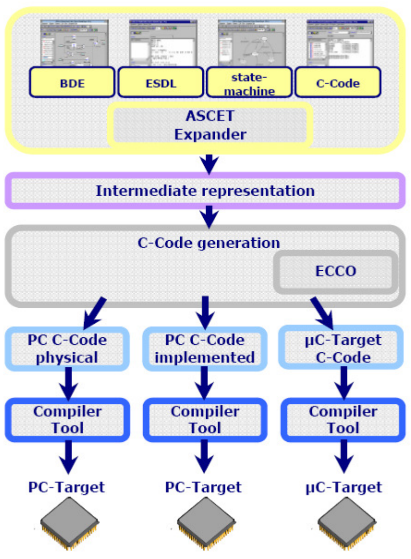


Fig. 2.1:28 – ASCET internal build mechanisms (source: ETAS [27])

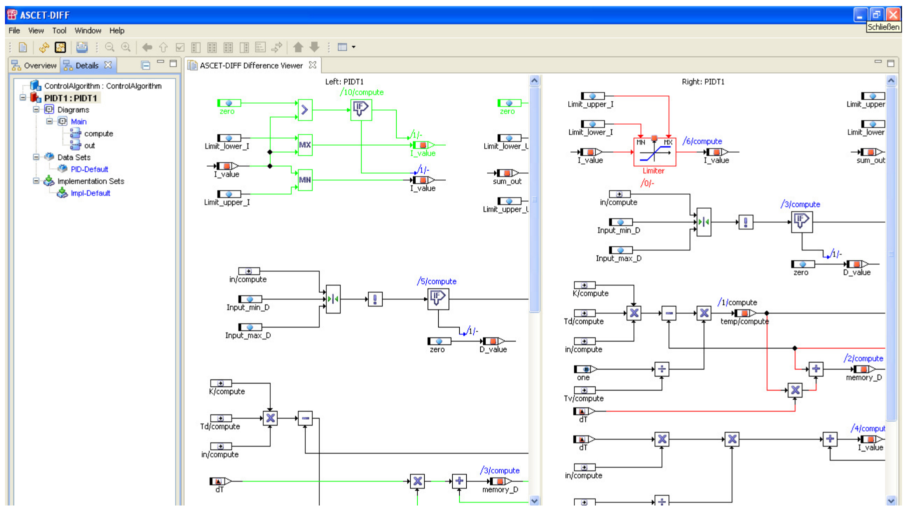


Fig. 2.1:29 – ASCET-DIFF tool for direct graphical FDEF comparisons (source: ETAS [216])

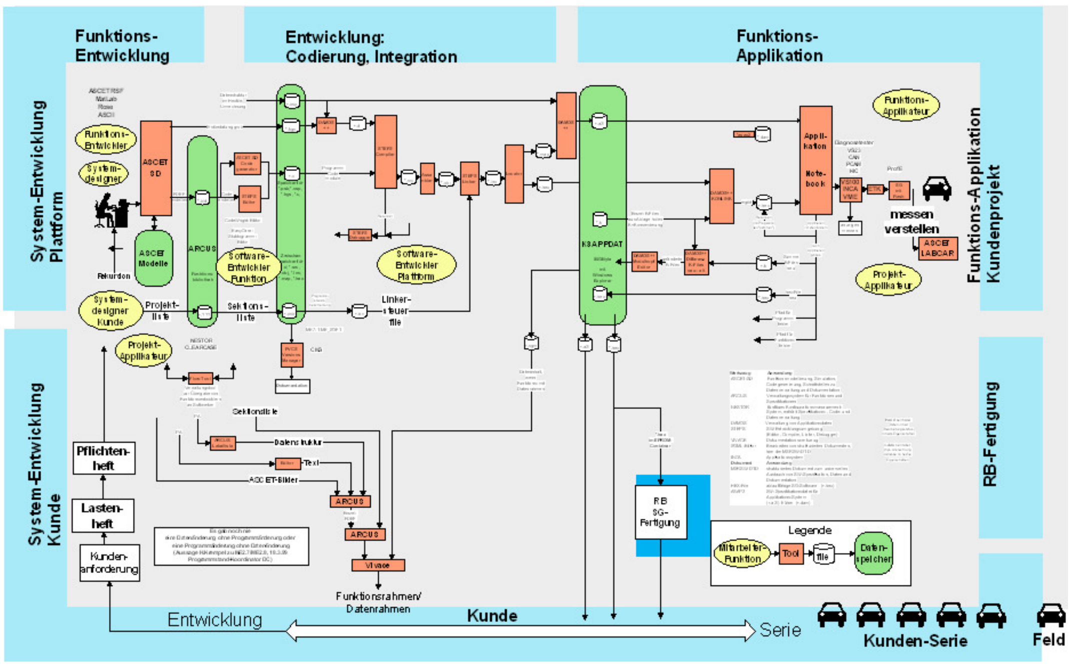


Fig. 2.1:30 – Glimpse of the tool-chain for a complete ASCET-SD v4 based development system (source: Bosch)

2.1.2.2 LIVE-FDEF

[Emphasis: tools integration]

Current research efforts in the automotive scene show some interesting possibilities in respect to raising the handling efficiency ratio. As an example we have the new *"Live-FDEF"* concept (Fig. 2.1:31) from Bosch [30], being developed since 2007 and finally released in 2009 and which tries to get the INCA monitoring/calibrating tool (Fig. 2.1:32) back together with the ASCET engine functionality development tools (Fig. 2.1:26). This tool allows to view the monitored values and even to change parameters directly inside Acrobat-Reader which is used to view Bosch's PDF function-manuals. This greatly reduces previous turn-around delays caused by having to use a separate monitoring and parameter-manipulation tool like INCA (Fig. 2.1:32) where the user has to search the desired variables and parameters indirectly through an alphabetically ordered global list, aside with the usual PDF function-manual reader. Bosch-ETAS plans to integrate this multi-tool into its INCA tool as well, to be used in the *Easy-Hooks* (or *EHOOKS* [223]) hooking development add-on.

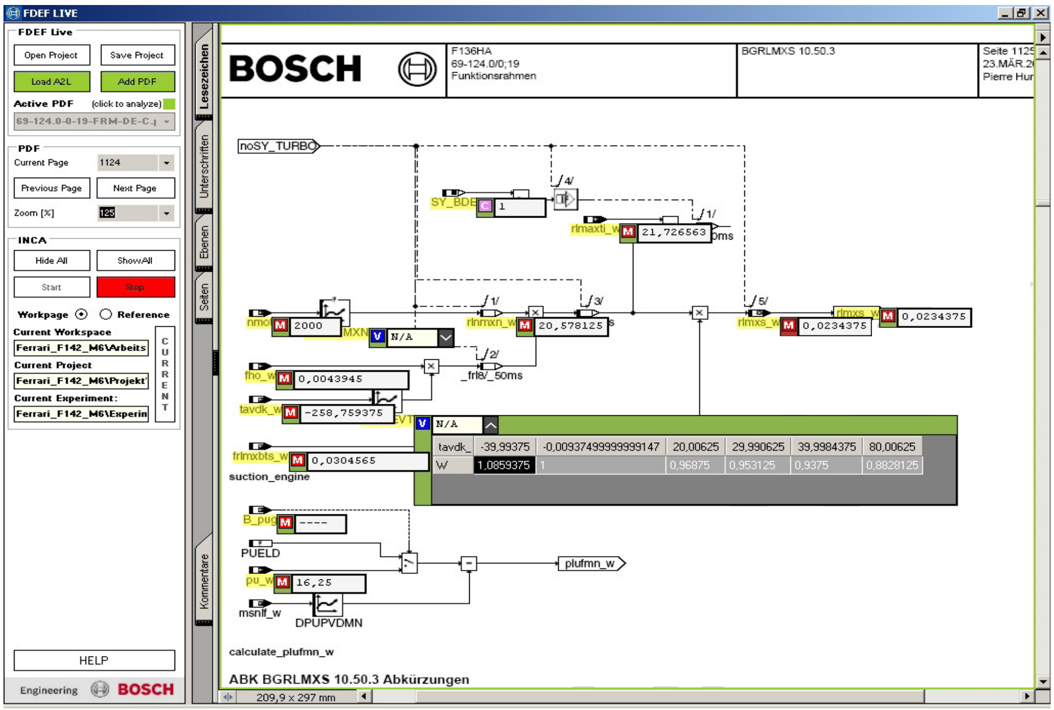


Fig. 2.1:31 – "LIVE-FDEF" tool uses a modified Acrobat Reader as a graphical viewer (source: Bosch [30])

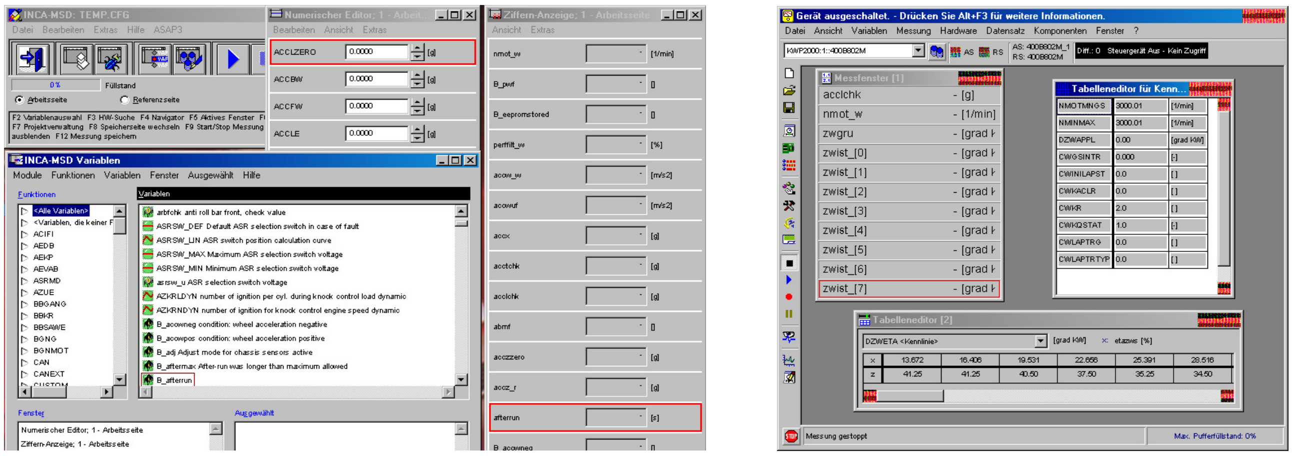
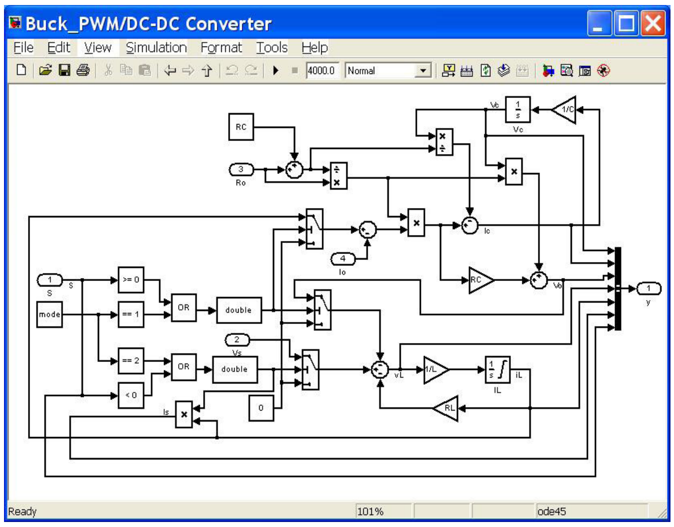


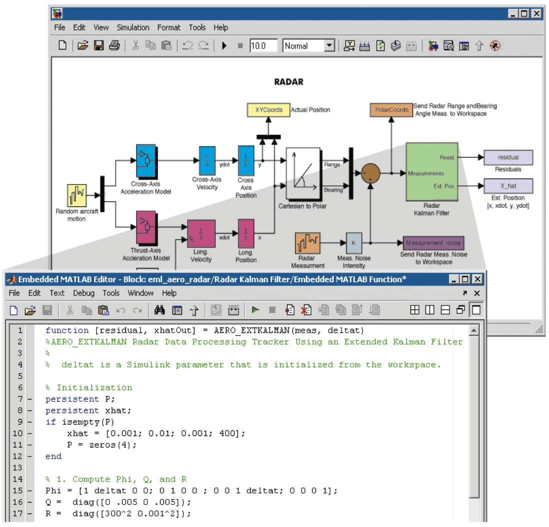
Fig. 2.1:32 – INCA's monitoring and calibration interfaces v1 and v4 (source: Bosch [93])

2.1.2.3 Matlab Simulink

[Emphasis: data-flow design, graphical design]

Having been developed by Mathworks [5] since the late 70s, MATLAB [299] comprises a vast range of software design tool-boxes added on top of a scripting-like language interface. One particular tool-box, the Simulink interface (Fig. 2.1:33) developed since the early 90s allows for graphical automotive functionality design just like the previous ASCET tool. Again and similarly, an auto-code generator connects this graphical top-most interface with the underlying "C"- and script-language classical structure (Fig. 2.1:34). Having this internal structure similar to that of ASCET, it also accepts code blocks written in standard "C" as input of the whole build process. Although not specifically designed for the automotive scene, as the previous ASCET tool is, it is certainly the most used and mentioned tool package in papers, general research and currently used software/hardware combinations in the automotive scene. As with ASCET, it is mainly referred as allowing *Rapid*- or *Fast-Prototyping* because of the relative ease of the visual programming with the associated auto-code generation. Because of this ease, it is even being applied on most recent motorsport ECUs from Bosch [10] and Magneti-Marelli [11] with micro-controller plus FPGA-based hardware. It additionally allows for "test-beds" and "partitioning" for simulation and bypassing of sub-systems. Matlab also has very nice graphical capabilities for creating useful and appealing interfaces, as well as robotic applications being developed (Fig. 2.1:35). Matlab Simulink was even used for years in the Toyota Formula 1 racing team, based on Magneti-Marelli's hardware platform [11]. Matlab Simulink is such a universally used and widespread tool, that it is even used to produce code for systems for which it has not been developed in the first place. [61] [165] [169] [170] are such examples. Similarly to the previous ASCET-DIFF, there are also Simulink Model difference tools commercially available [217] [218] and shown in Fig. 2.1:36. Unlike the previous ASCET tool that needs INCA or "LIVE-FDEF", Mathworks is able to let the user display real-time feedback values directly on the Simulink model's signals and to set parameters as shown in Fig. 2.1:37. This is given by the "external mode" operation possibility [225]. As with all tools of this kind, simultaneous code-generation and values' viewing/setting is not possible, existing a substantial turnaround or delay time between such handling operations. Matlab is currently steadily increasing its notoriety in the "TIOBE Programming Community Index" [236], having already passed 0,66% of usage in the 16th place.




File Edit View Simulation Format Tools Help
10.0 Normal
RADAR
XYCoords Actual Position
PolarCoords Send Radar Range andBearing Angle Meas. to Workspace
Random aircraft motion
Cross-Axis Acceleration Model
Cross-Axis Velocity
Cross Axis Position
ydot
y
Cartesian to Polar
Range
Bearing
Measurements
Est. Pos.
Radar Kalman Filter
residual Residuals
X_hat Est. Position [x, xdot, y, ydot]
Thrust-Axis Accleration Model
Long. Velocity
Long. Position
xdot
x
Radar Measurment
Meas. Noise Intensity
Measurement noise Send Radar Meas. Noise to Workspace
Embedded MATLAB Editor - Block: eml_aero_radar/Radar Kalman Filter/Embedded MATLAB Function*
File Edit Text Debug Tools Window Help

```
function [residual, xhatOut] = AERO_EXTKALMAN(meas, deltat)
%AERO_EXTKALMAN Radar Data Processing Tracker Using an Extended Kalman Filter
%
%  deltat is a Simulink parameter that is initialized from the workspace.

% Initialization
persistent P;
persistent xhat;
if isempty(P)
    xhat = [0.001; 0.01; 0.001; 400];
    P = zeros(4);
end

% 1. Compute Phi, Q, and R
Phi = [1 deltat 0 0; 0 1 0 0 ; 0 0 1 deltat; 0 0 0 1];
Q =  diag([0 .005 0 .005]);
R =  diag([300^2 0.001^2]);
```

Fig. 2.1:33 – Matlab Simulink graphical interface programming package (source: Mathworks [5])

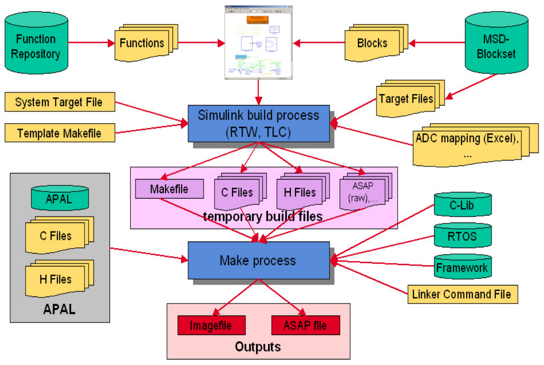
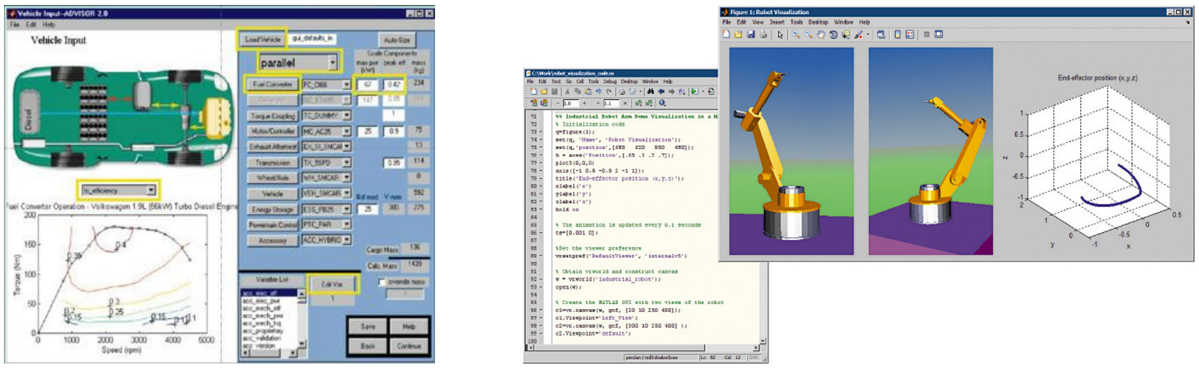


Fig. 2.1:34 – Matlab Simulink internal build mechanisms (source: Bosch Motorsport [10])



```
%% Industrial Robot Arm Demo Visualization in a M
% Initialization code
q=figure(1);
set(q, 'Name', 'Robot Visualization');
set(q,'position',[480   620   950   480]);
h = axes('Position',[.65 .1 .3 .7]);
plot3(0,0,0)
axis([-1 0.5 -0.5 2 -1 1]);
title('End-effector position (x,y,z)');
xlabel('x')
ylabel('y')
zlabel('z')
hold on

% The animation is updated every 0.1 seconds
ts=[0.001 0];

%Set the viewer preference
vrsetpref('DefaultViewer', 'internalv5')

% Obtain vrworld and construct canvas
w = vrworld('industrial_robot');
open(w);

% Create the MATLAB GUI with two views of the robot
c1=vr.canvas(w, gcf, [20 10 250 400]);
c1.Viewpoint='left_View';
c2=vr.canvas(w, gcf, [300 10 250 400] );
c2.Viewpoint='default';
```

Fig. 2.1:35 – Matlab's graphical capabilities and robotics applications (source: Mathworks [5])

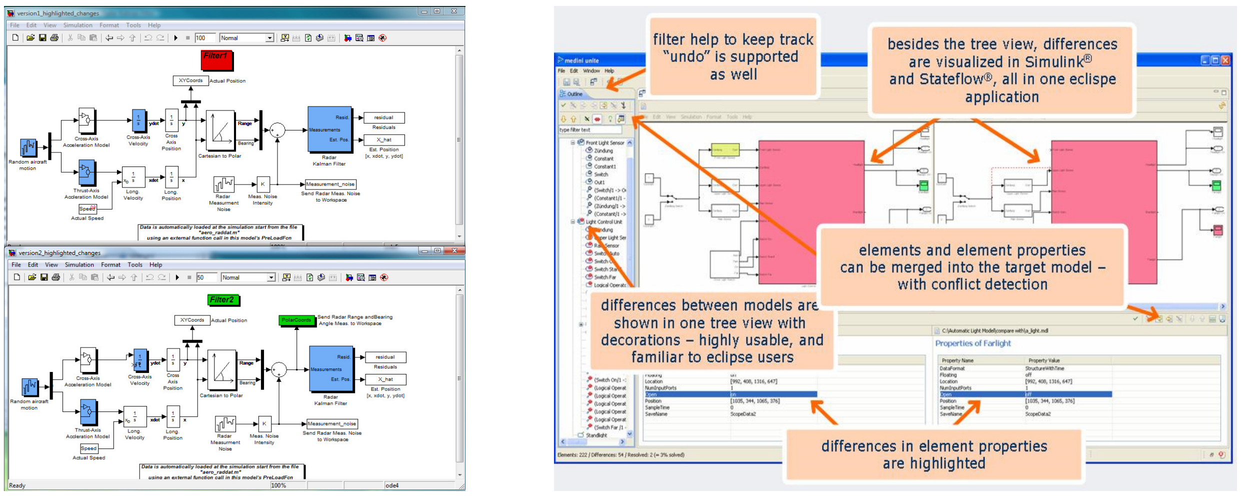

filter help to keep track "undo" is supported as well
besides the tree view, differences are visualized in Simulink® and Stateflow®, all in one eclipse application
elements and element properties can be merged into the target model – with conflict detection
differences between models are shown in one tree view with decorations – highly usable, and familiar to eclipse users
differences in element properties are highlighted
Filter1
Filter2
Data is automatically loaded at the simulation start from the file "aero_raddat.m" using an external function call in this model's PreLoadFcn

Fig. 2.1:36 – Graphical comparison; Left: "SimDiff"; Right: "Medini Unite" (source: Ensoft [218], IKV++ [217])

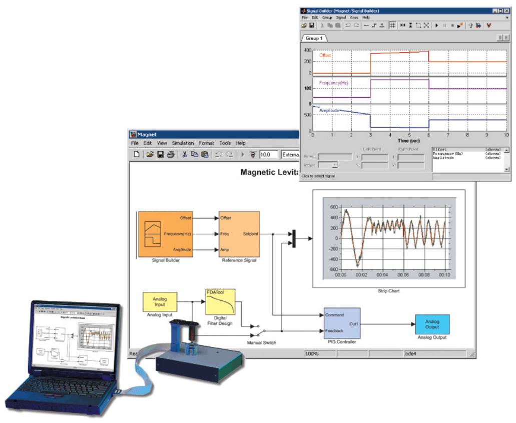
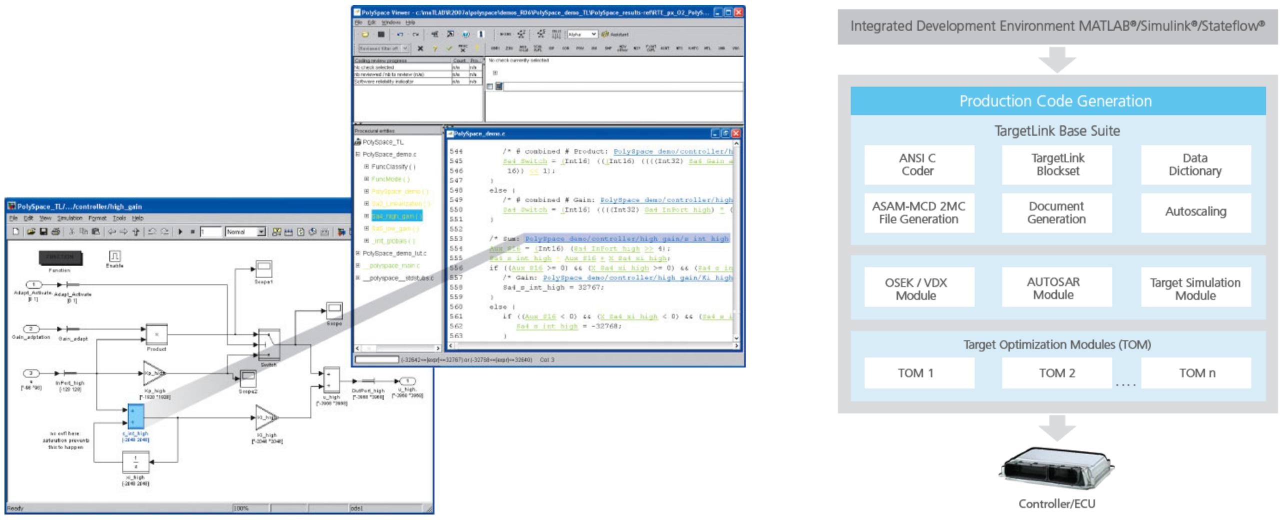


Fig. 2.1:37 – Direct graphical values' display on the model through "external mode" (source: Mathworks [5])

2.1.2.4 TargetLink

[Emphasis: auto-code generation]

dSpace develops the “TargetLink” auto-code generator since 1999 [165]. This software tool generates code out of Mathworks’ Matlab Simulink [5] models. TargetLink is regarded as the established production auto-code generator in the automotive industry. It claims to produce code with hand-coding quality level. Applications range from automotive to aerospace [166] and claims to cut development times by as much as 50%. TargetLink also recently supports AUTOSAR [67] compliant ECUs. Fig. 2.1:38 illustrates this tool. [168] also produce an AUTOSAR-compliant auto-code tool, which claims to even support ASCET and TargetLink in the future. In the meanwhile, dSpace launched the new "MicroAutoBox II" [322], a compact rapid prototyping platform for in-vehicle use, with even more processing power and with even shorter boot-up times, so that in ECU networks the "MicroAutoBox II" can perform immediate boot-up like a real ECU.



```
    /* # combined # Product: PolySpace demo/controller/h
    Sa4 Switch = (Int16) (((Int16) ((((Int32) Sa4 Gain 
     16)) << 1);
  }
  else {
    /* # combined # Gain: PolySpace demo/controller/high
    Sa4 Switch = (Int16) ((((Int32) Sa4 InPort high) * (

  /* Sum: PolySpace demo/controller/high gain/s int high
  Aux S16 = (Int16) (Sa4 InPort high >> 4);
  Sa4 s int high = Aux S16 + X Sa4 xi high;
  if ((Aux S16 >= 0) && (X Sa4 xi high >= 0) && (Sa4 s in
    /* Gain: PolySpace demo/controller/high gain/Ki high
    Sa4_s_int_high = 32767;
  }
  else {
    if ((Aux S16 < 0) && (X Sa4 xi high < 0) && (Sa4 s i
      Sa4 s int high = -32768;
    }
```

Fig. 2.1:38 – TargetLink auto-code generation and development environment (source: dSpace [165] [167])

2.1.2.5 Software Bypasses & Hooks

[Emphasis: legacy easy-handling]

As in OSEK/VDX [444] OS, by changing the source-code, special code snippets may be placed into the main program, to allow him taking control of certain variables, parameters and even code-blocks. This procedure is called by placing software "hooks" into the program or "bypassing" certain regions by replacing them with special testing code.

In this respect, ETAS contributed with an interesting paper [409] that explains how most of these older software bypass methods work with ETAS products such as the ones depicted above, up until 2008 (detailed earlier). This work is further applied in [410]. Generally, bypass methods either use static mechanisms where the desired variables are fixed inside the ECU source-code, or they use more modern mechanisms where the user may choose among a list of known variables. This list is commonly available at the end of the ECU software build process and is normally given to functionality developers. Constant parameters may be temporarily placed inside RAM, to also be changeable at runtime, just as normal variables that reside in RAM anyway. This is what happens in a standard field calibration session, where a running dyno engine is calibrated live through changing its parameters (for maximum power, for example). Generally, a good bypass method allows reading ECU input variables (that would go into an FDEF), making calculations on those values, and writing ECU output variables (that would come from an FDEF). By selectively reading-only (side-calculations only), writing-only (output-forcing only) or even using both (read/write) at the same time (full FDEF bypass), all possible combinations of bypassing may be performed. Fig. 2.1:39 illustrates the typical functionality bypassing mechanism.

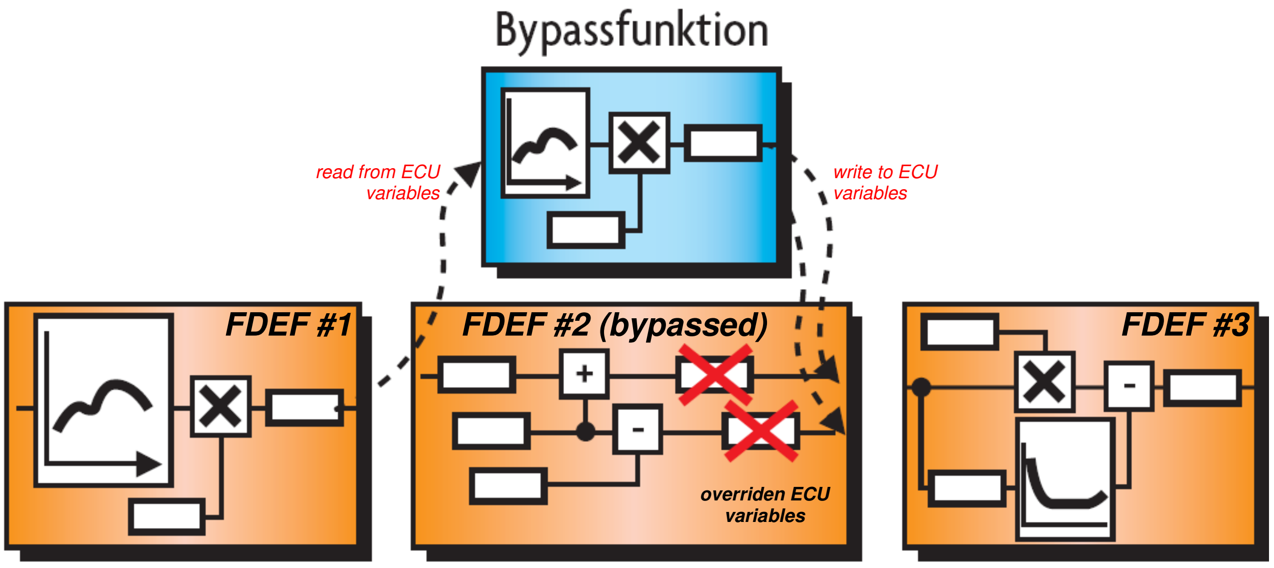


Fig. 2.1:39 – FDEF bypass: read ECU values changed and written back to the ECU (source: ETAS [409])

Service-based bypass – extra calls (services) are placed inside the existing task-lists, whereas those services then contain the bypass code where anything can be done. Although this method has the advantage of not requiring any code changes during the bypassing actions, it requires that those extra service calls be already implemented in that code, also consuming processing time. Furthermore, data inconsistency may occur, for the simple fact that automotive software possesses many processes running inside different task-lists with different priorities (may interrupt one another), where one cannot guarantee that a variable calculated by an FDEF is not read by the next FDEF right before the bypassing function has had the chance to overwrite this variable with its own "bypassed" value. This causes the next FDEF to take the originally calculated value instead of the "bypassed" new one. A buffering technique where memory variables' copies are locally generated at the beginning of the task and written back to memory only at the end of the task, allows to overcome this pitfall, at the expense of much greater code complexity. This method is depicted in Fig. 2.1:40.

Switch-based Bypass - The previous method possesses the data inconsistency pitfall exactly because it allows bypassed or any other FDEFs to potentially overwrite the same variables as the bypass outputs do. This method however, avoids this altogether by statically placing switches inside the code, where the bypassed FDEF cannot write to the output variables any more. When the bypass is disabled, everything works as it always has. When the bypass is activated, then the switch turns and the next FDEF uses the bypassed outputs only, while the bypassed FDEF outputs are simply ignored and lost. This mechanism is totally independent from task-lists and their priority/interruption related pitfalls. Nevertheless, this method also requires previous source-code intervention and the related processing time consumption. This method is also depicted in Fig. 2.1:40.

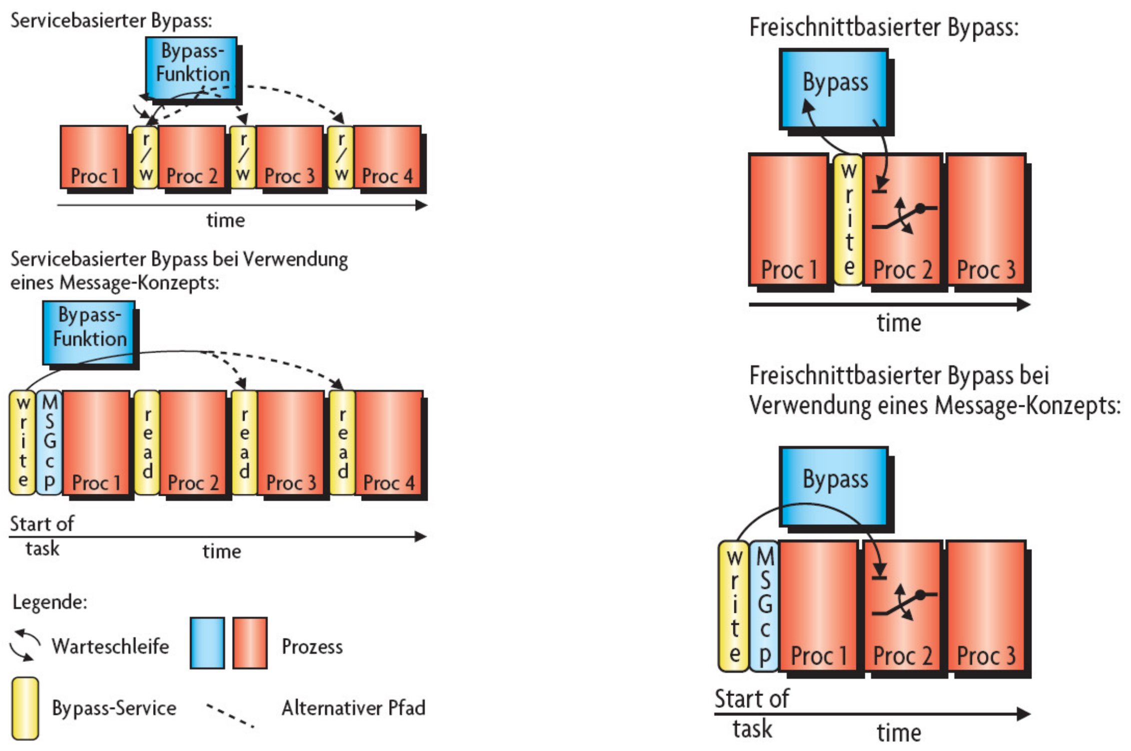


Fig. 2.1:40 – Left: service-based mechanism, right: switch-based mechanism (source: ETAS [409])

2.1.2.6 Easy-Hooks (EHOOKS)

[Emphasis: easy-handling, smart bypasses]

ETAS [7] also developed the so-called *"EHOOKS"* software add-on tool [223] [401] [402], which allows to flexibly set software "hooks" into the automotive program without the former and time-consuming need to place those "hooks" into the source-code for them to be compiled themselves into the output machine-code. This former functionality bypassing method required too much turnaround delays while also obviously disrupting established development paths. These new *"EHOOKS"* allow the user to easily place prototyping and testing code into the main program without having to recompile it. Nevertheless, this solution needs free computing power and free memory space to accommodate prototyping code and values inside an existing ECU, besides causing intrinsic timing and general code-execution intrusion as in any software "hooking" mechanism. The underlying mechanism of this technology needs those special software implants over the existing hardware, to enable all the above mentioned features. Also, for larger-scale prototyping, the usual external bypassing hardware methods must be employed such as the "ES" Series platforms [224] from ETAS.

Fig. 2.1:41 shows a glimpse of this *"EHOOKS"* mechanism, which additionally allows for the standard parameter calibration as well as RAM-variable fixating. This technology nicely fits into the Rapid-Prototyping category. A more detailed view is seen in Fig. 2.1:42 where the "secret" is fully revealed in [401]: the hooks are placed directly inside the "HEX" file, i.e. the downloadable machine-code file, after being prepared by the "EOOKS-PREP" software block. The only needed files are the program "HEX" and the usual "A2L" calibration addresses file. As said before, the advantage of this is that the user does not have to make those hooks inside the source-code or does not have to call the software manufacturer to do that job. In other words, long compilation waits are completely

eliminated, but there is still the need to prepare files and to download them onto the hardware target, thus being an offline method. Fig. 2.1:43 shows illustrations of the internal explicit bypassing mechanisms, where code blocks are fully bypassed. The corresponding input variable values are switched to the substitution bypass code block that calculates the output values passed onto the normal output variables.

In the meanwhile, v2.0 of E-HOOKS supports a variety of software development packages such as ASCET, Simulink, etc., and several ECU manufacturers such as Bosch, Denso and Continental [490].

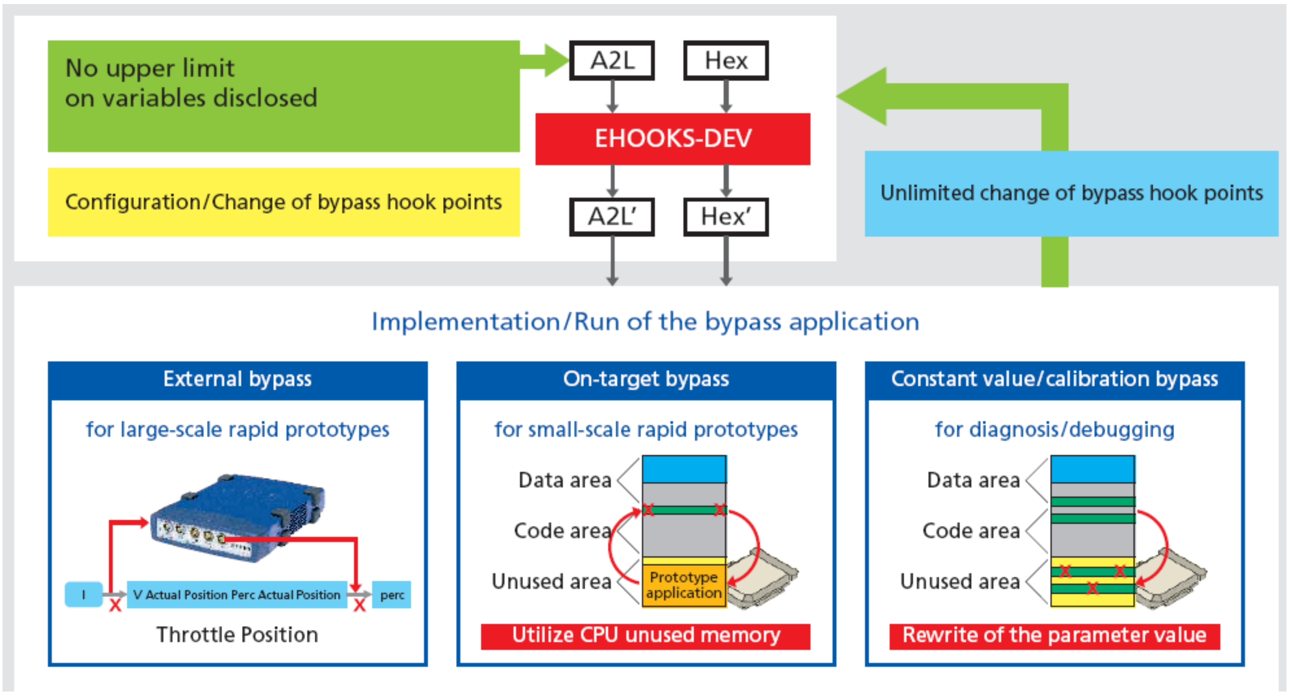

No upper limit
on variables disclosed
Configuration/Change of bypass hook points
A2L
Hex
EHOOKS-DEV
A2L'
Hex'
Unlimited change of bypass hook points
Implementation/Run of the bypass application
External bypass
for large-scale rapid prototypes
I
V Actual Position Perc Actual Position
perc
Throttle Position
On-target bypass
for small-scale rapid prototypes
Data area
Code area
Unused area
Prototype application
Utilize CPU unused memory
Constant value/calibration bypass
for diagnosis/debugging
Data area
Code area
Unused area
Rewrite of the parameter value

Fig. 2.1:41 – EHOOKS software mechanism (source: ETAS [223])

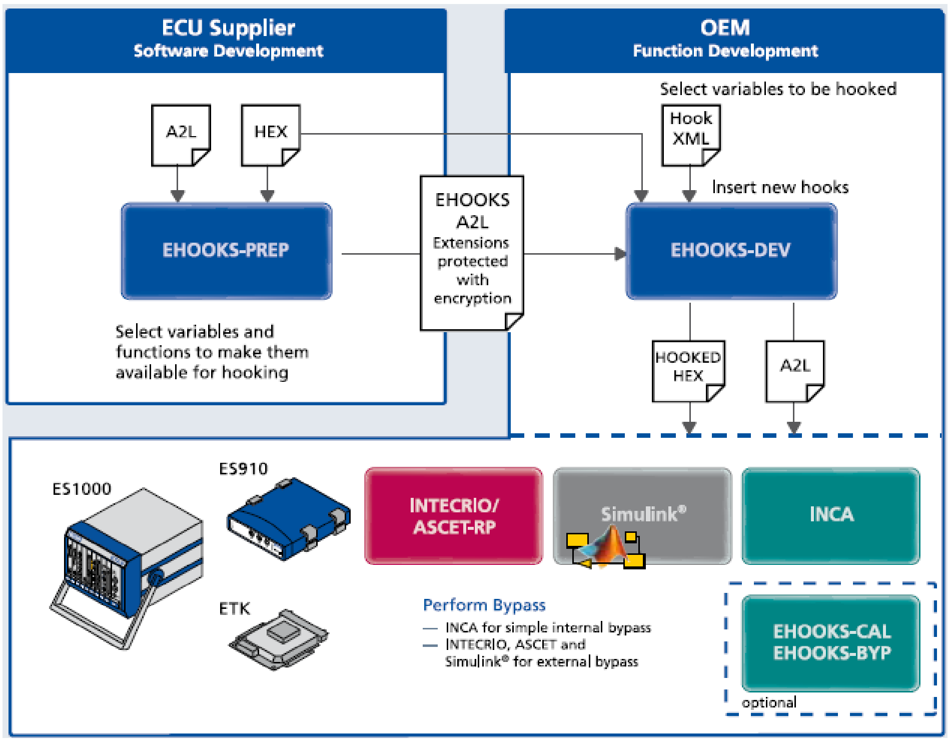

ECU Supplier
Software Development
A2L
HEX
EHOOKS-PREP
Select variables and functions to make them available for hooking
EHOOKS A2L Extensions protected with encryption
OEM
Function Development
Select variables to be hooked
Hook XML
Insert new hooks
EHOOKS-DEV
HOOKED HEX
A2L
ES1000
ES910
ETK
INTECRIO/ ASCET-RP
Simulink®
INCA
Perform Bypass
— INCA for simple internal bypass
— INTECRIO, ASCET and Simulink® for external bypass
EHOOKS-CAL
EHOOKS-BYP
optional

Fig. 2.1:42 – EHOOKS machine-code preparation and development tools in detail (source: ETAS [490])

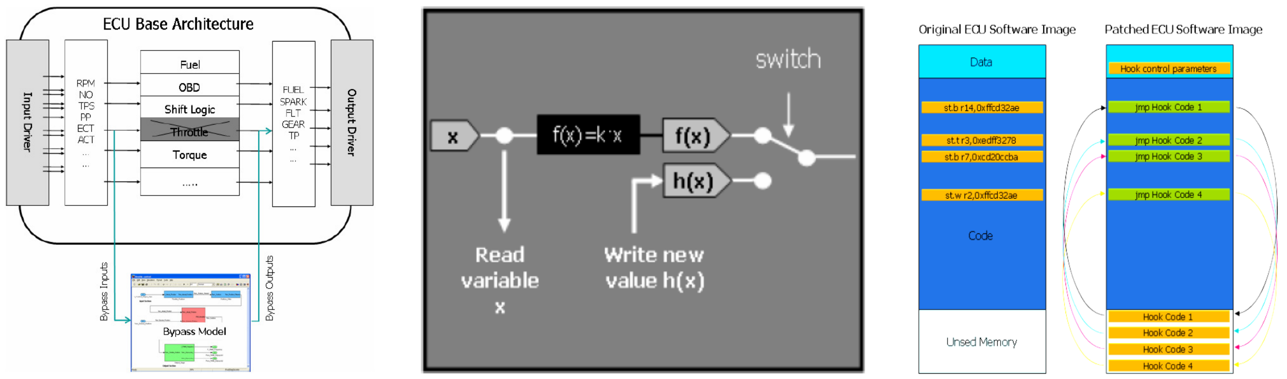


Fig. 2.1:43 – EHOOKS internal explicit bypassing mechanism illustrations (source: ETAS [402])

2.1.2.7 No-Hooks OnTarget

[Emphasis: easy-handling, smart bypasses]

Accurate Technologies (ATI) develops the “No-Hooks OnTarget” software add-on [300], which works in a very similar way to the previous *"EHOOKS"* [223] functionality bypassing method, thereby also modifying the "HEX" file directly. Exactly as in the previous *"EHOOKS"* mechanism, the only needed files are the program "HEX" and the usual "A2L" calibration addresses file. If model bypassing code is also to be used, then the "MAP" file is also needed to hook it into the existing function calls [411]. Therefore, this software tool also allows for Rapid-Prototyping developments without the developer to have to re-compile the source program. The same way, parameter calibration and RAM variable fixation is also part of this packages' capabilities. Code bypass blocks are allowed as well. So, just as in *"EHOOKS"*, *"No-Hooks"* Rapid Prototyping bypasses fixed parameters, variables or function calls and replaces them with alternative values or code [301]. This package comes bundled with the ATI VISION software for easy viewing of variables and such, while also being capable of connecting with Matlab Simulink models for use in its bypasses (see Fig. 2.1:44). As in *"EHOOKS"* and readily seen in Fig. 2.1:45, the underlying mechanism of this technology also relies on a special software layer over the existing hardware, as well as communications with some existing ECU interface, to enable all the above mentioned features, as well. Fig. 2.1:45 shows a powerful memory emulator [301] ATI also offers as an option for enhanced bandwidth for variables, but this imposes hardware intrusion as normal emulators [134] [135] [221] do. Still, if the developer wants to use software-based emulation only, this solution is physically much easier to use.

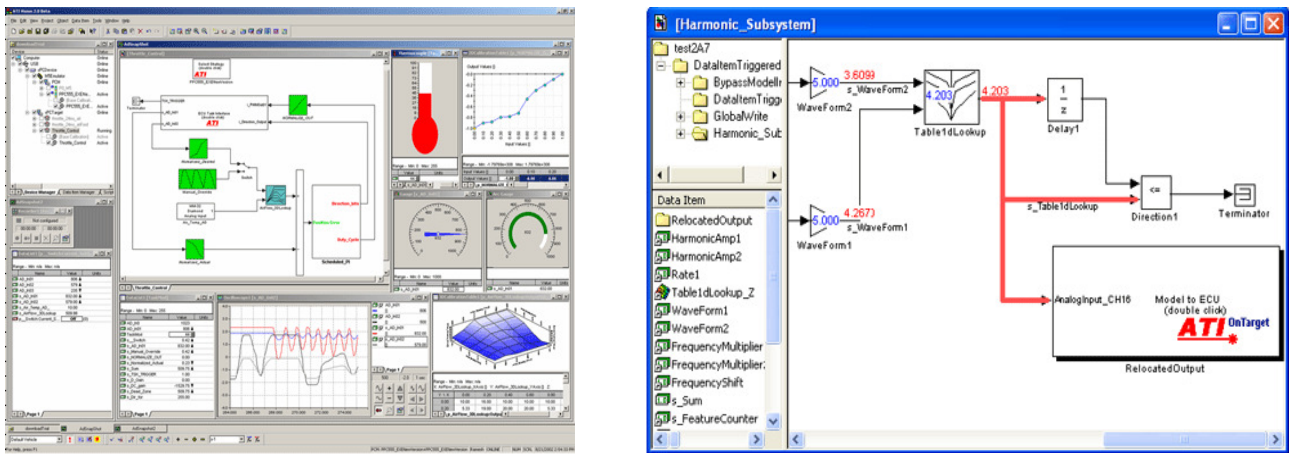


Fig. 2.1:44 – No-Hooks VISION development environment and Simulink model viewer (source: ATI [300])

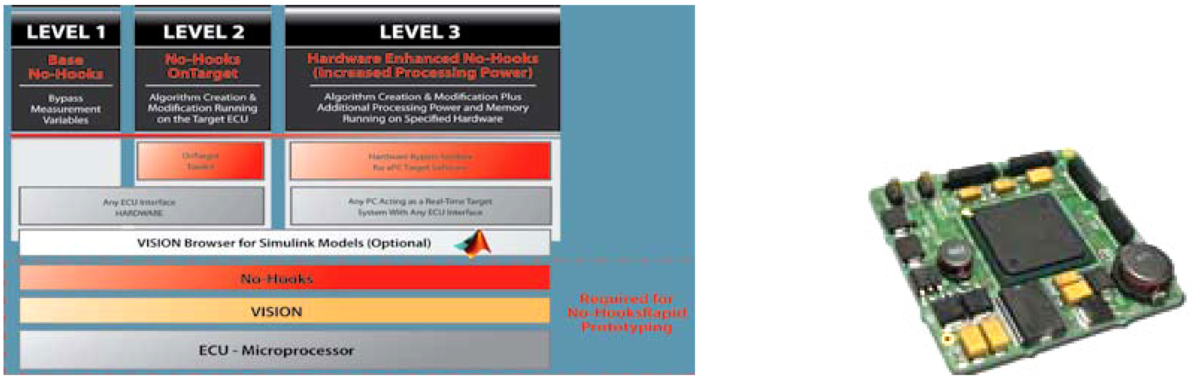

LEVEL 1
Base No-Hooks
Bypass Measurement Variables
LEVEL 2
No-Hooks OnTarget
Algorithm Creation & Modification Running on the Target ECU
LEVEL 3
Hardware Enhanced No-Hooks (Increased Processing Power)
Algorithm Creation & Modification Plus Additional Processing Power and Memory Running on Specified Hardware
Any ECU Interface HARDWARE
Any PC Acting as a Real-Time Target System With Any ECU Interface
VISION Browser for Simulink Models (Optional)
No-Hooks
VISION
ECU - Microprocessor
Required for No-HooksRapid Prototyping

Fig. 2.1:45 – No-Hooks underlying software mechanisms and optional hardware emulator (source: ATI [300])

2.1.2.8 ECU Interface Manager

[Emphasis: easy-handling, smart bypasses]

dSpace [4] made a SW tool [417] similar to the previous *“No-Hooks OnTarget”* [300] and *"EHOOKS"* [223], which also takes the original HEX file and changes it by incorporating the desired "bypasses" for later downloading onto the target (Fig. 2.1:46). The process is thus always the same: memory reads and writes are automatically searched and detected inside the original HEX file and those instructions are then replaced by bypassing code that redirects those reads and writes to special bypassing code blocks, which then allow the user to place its own data for the ECU to work with, thus bypassing the original data.

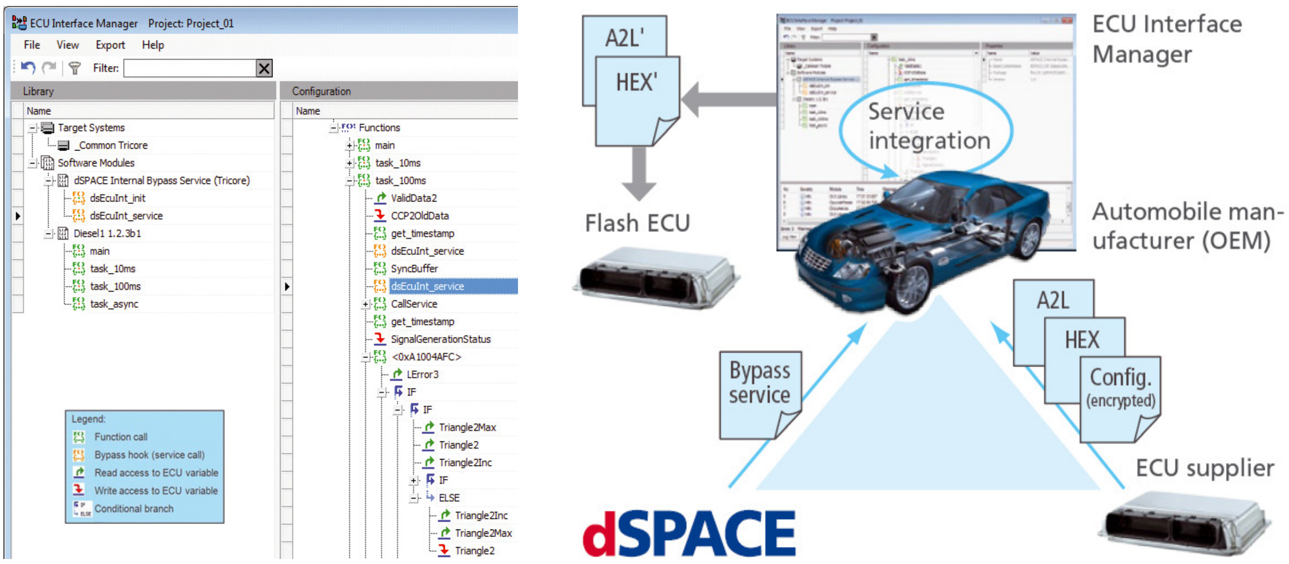


Fig. 2.1:46 – ECU Interface Manager software bypassing package (source: dSpace [417])

2.1.2.9 GAP Graphical Application Programmer

[Emphasis: visual programming since the 1980s]

Woodward Graphical Application Programmer (GAP) [391] works with a proprietary visual language developed by Woodward itself, since mid 1980s. As opposed to other iconic languages, this language bases its blocks on many lines of “C” code that are hidden inside each block. This is a similar approach as in "component-based" software development [159]. Each block thus processes some more complex algorithms than the usual simple arithmetic and logic operations used in other tools. The resulting stream of “C” code is then compiled and deployed into hardware. Fig. 2.1:47 shows examples of this block-like programming. Fig. 2.1:48 then shows an interesting way of monitoring values directly on the graphical program, similar to Live-FDEF [30], in that the values are shown directly on the spots where they really occur. The possibility of adding the respective variables to a tracing window rounds off the whole monitoring concept [392].

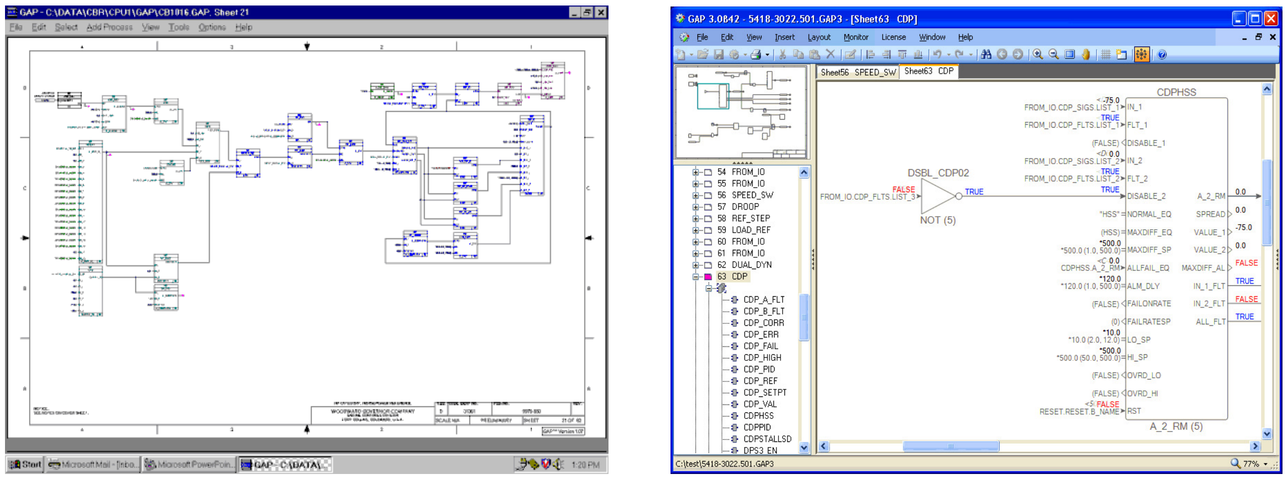


Fig. 2.1:47 – Woodward's visual programming tool for diesel engines (source: Woodward [391] [392])

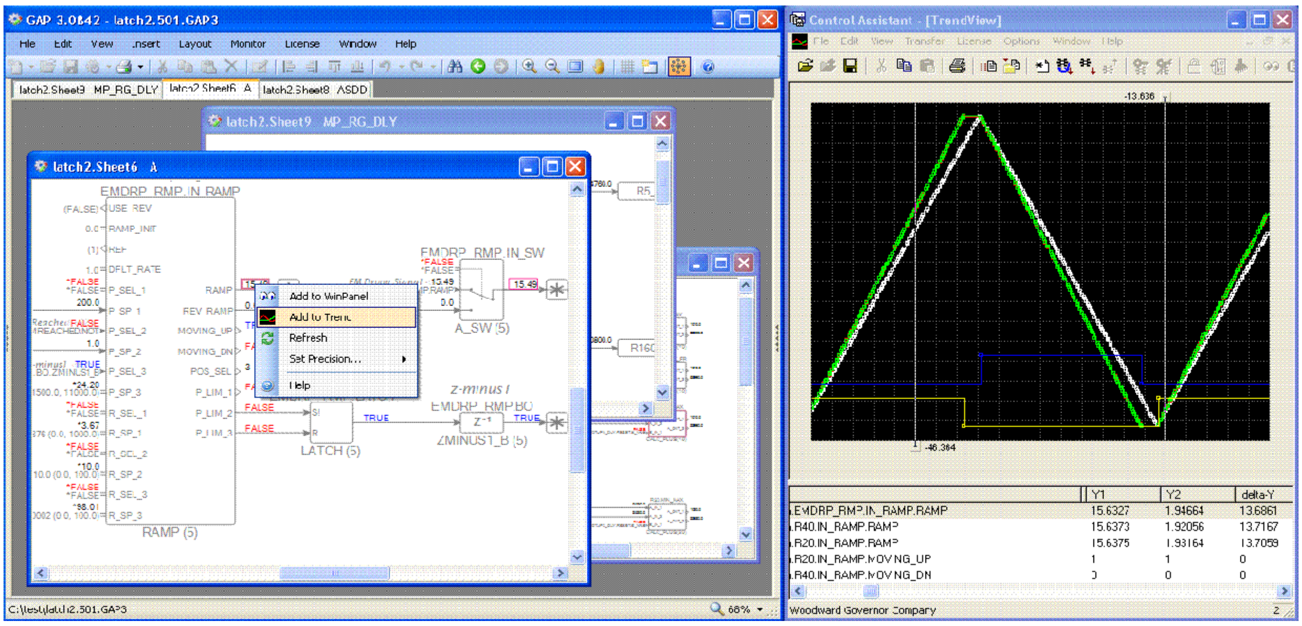
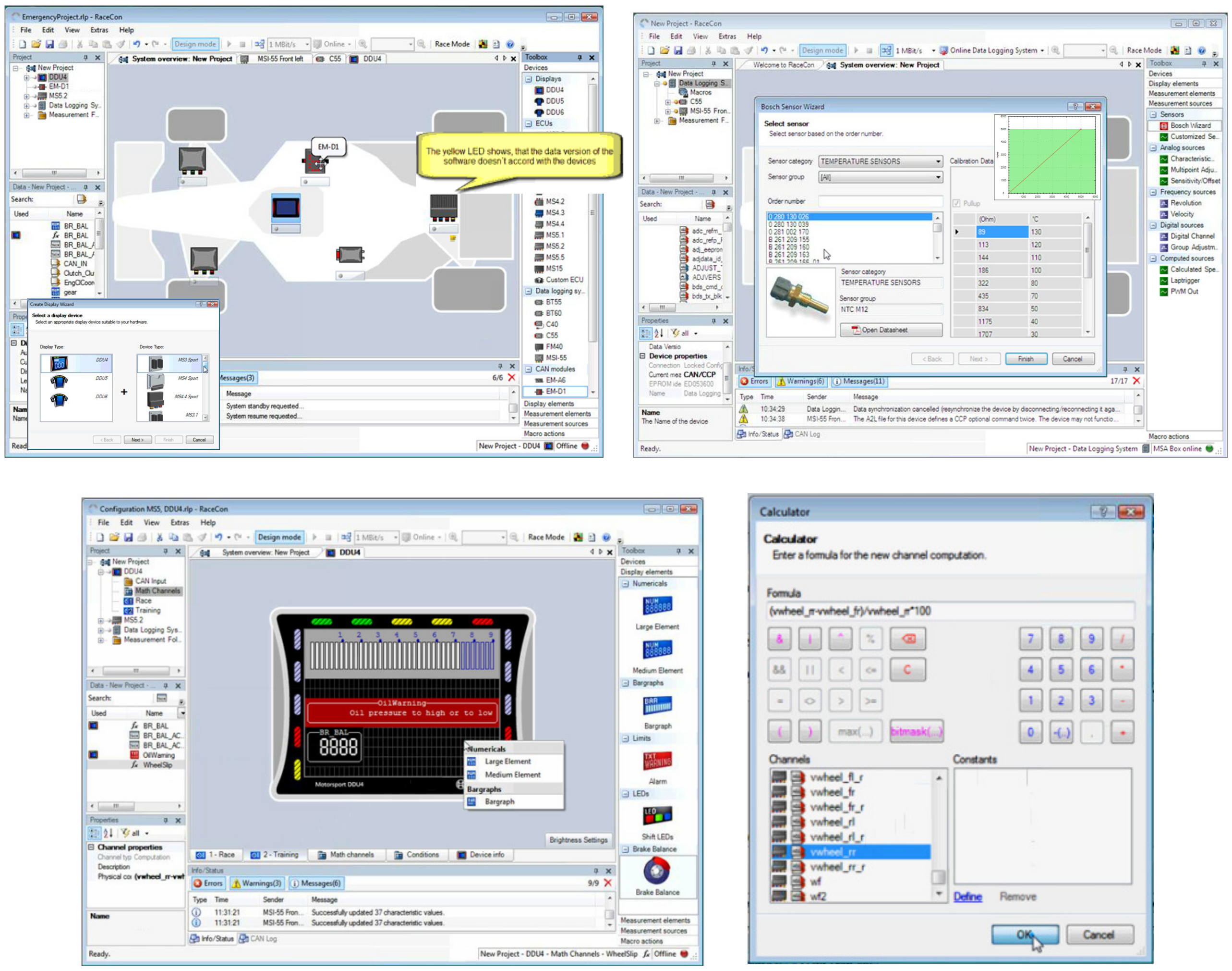


Fig. 2.1:48 – Woodward's visual monitoring tool for diesel engines (source: Woodward [392])

2.1.2.10 RaceCon

[Emphasis: visual configuration for vehicle components]

Bosch Motorsport developed a software tool [484] that serves vehicle component configuration. As can be seen in Fig. 2.1:49, this tool allows for visual configuration of ECUs, dataloggers, driver's displays, sensors, actuators, etc. It not only allows for the basic configuration such as communications setup, but also for advanced sensor parameterization, calculations and so on. Component status feedback such as errors, warnings and other operating conditions are also reported in an online fashion. Contrary to earlier times where everything had to be coded/programmed in a fixed way, customers are now able to quite easily configure their race vehicles and components through this graphical tool and without any programming skills required at all.


The yellow LED shows, that the data version of the software doesn't accord with the devices
Bosch Sensor Wizard
Select sensor
Calculator
Enter a formula for the new channel computation.
Formula
Channels
Constants
OK
Cancel

System standby requested.
System resume requested.
Data synchronization cancelled (resynchronize the device by disconnecting/reconnecting it again...
Successfully updated 37 characteristic values.
Successfully updated 37 characteristic values.

Fig. 2.1:49 – RaceCon screenshots (source: Bosch [484])

2.1.3 Considerations about non-automotive software tools

[Emphasis: tools not used for automotive vehicles but also very interesting]

There are also some highly interesting commercial and research non-automotive software tools, both from the users' and developers' point of views. Some possess features that will strongly relate to the presented work herein. Special attention should be put on tools such as the IEC 61131-3 PLC Standard, LabVIEW, Visual Basic/C#, JAVA and LEGO Mindstorms. These and others have been briefly investigated in the extra [Att. 6], by presenting their most interesting details for the scope of this thesis, and will be taken into account during the work developed herein. Lastly, at the end of the following hardware platforms, some integrative solutions and languages will be presented, to wrap up both software and hardware realities. The relative importance of some of the herein previously presented software and the following hardware realities, is also going to be presented after that as well.

2.2 State-of-the-art hardware platforms

[Emphasis: target platforms]

To be able to execute the previous graphical and non-graphical languages, one needs the corresponding hardware platforms. While it can be imagined that those platforms may not display so much innovation, while being "simply" built with standard processors and standard analog/digital electronics to support them, there are some innovations and state-of-the-art hardware components which are described next, to prepare for the innovations to be implemented in this thesis' work. In many cases, graphical programming languages' software vendors also offer the corresponding hardware targets.

2.2.1 Automotive hardware targets

[Emphasis: target platforms]

In this section, the main actual hardware targets which accept the graphical and non-graphical programming languages exposed previously are presented. This list encompasses hardware actually used in series- and racing-cars, as well as experimental/testing and prototyping targets.

2.2.1.1 Series-cars' ECUs

[Emphasis: "end-of-line/mass-production"]

When it comes down to mass-production, mainly cheap, reliable and small hardware packages are chosen. Therefore, these hardware platforms unavoidably present capabilities that do not go far beyond that what is strictly necessary to keep the vehicles (chassis+engine) running and to be able to diagnose and service them in the auto-shops. These ECUs are therefore most often developed in a completely separated thread than all other prototyping, testing and general development hardware platforms. This later ones are mainly aimed at allowing all possible programming, debugging and prototyping actions, in the least amount of time possible, costing up to thousands of times more than a standard series-car ECU. While internal damage to these standard ECUs often results in simply trashing and replacing them, much more expensive prototyping hardware equipment must be repaired. This shows the drastic difference in prices. Some of the currently existing series-car ECU manufacturers and developers are the following: Bosch, Lucas, Delphi, Saab (Trionic), Siemens, Renault (Sagem), Magneti-Marelli, Delco (GM).

These ECUs are normally still programmed in standard Assembly and "C". At most, they also have hand-tuned ASCET or Matlab Simulink generated code. Strictly auto-generated code is mainly used in prototyping equipment discussed in the next examples of hardware platforms. This has mainly to do with natural "childhood problems" associated with the more recent automatic ways/tools for generating automotive code, so that there is still much manual programming, optimizing and validation work done. Fig. 2.2:50 shows older and newer ECUs, just for technological comparison purposes. The MS3 ECU consists of a ceramic substrate "PCB" where all components lie, including case-less chips directly bonded to that ceramic "PCB". It must be emphasized that these standard ECUs are trimmed to minimize final factory cost and do not allow any changes to the hardware whatsoever, while the software must be adapted and trimmed to that hardware. All these

ECUs and especially state-of-the-art ones rely on processing cores composed by one or more micro-controllers specially developed for the automotive scenario. These specialized micro-controllers contain dedicated I/O peripherals, besides other mechanisms to accelerate basic routines for injection- and ignition-outputting. Micro-controller families such as the C167 [178] and the TriCore [119] were developed by Infineon-Siemens for automotive ECUs used in most Bosch ECUs. Other examples of automotive micro-controllers are the MC and MPC families from Freescale (former Motorola) [120] and chips from Renesas [422]. Intel [423] is interested in getting into this market (on the multimedia side) with its Atom chip. Although these chips are not easy at all to program, debug and maintain, the vast majority of series-car ECUs use this kind of "turbo-charged" micro-controllers, being commercially successful mainly due to the mass-production cost-optimization, by selling the same costly and complex software millions-fold. These ECUs are therefore seen as mere "end-of-line" products that come out from years of prototyping with the equipment that is going to be presented in the next section and which allows for the needed flexibility when developing new systems or evolving existing ones. Fig. 2.2:51 shows the internal hardware and software structures of series-car ECUs, while Fig. 2.2:52 shows a motorsport ECU and its internal view.

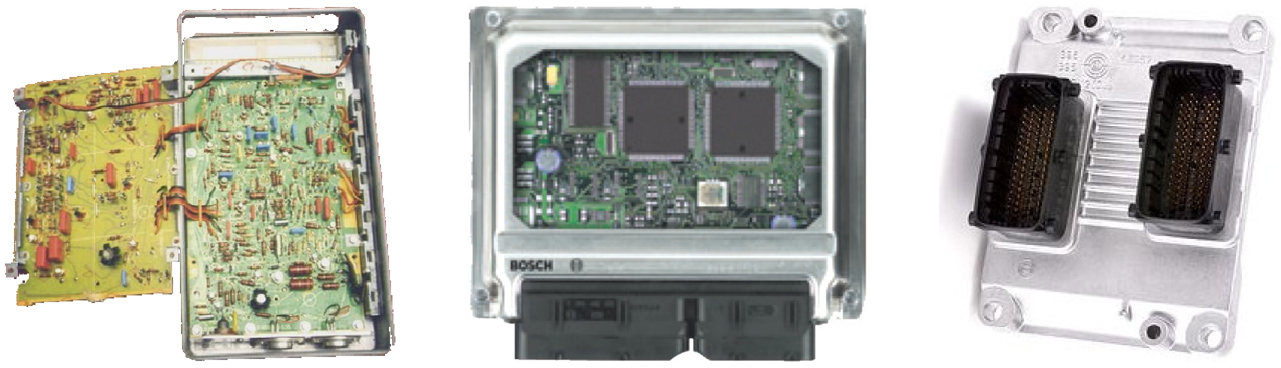


Fig. 2.2:50 – Older "D Jetronic", newer "ME7.1.1" & hybrid-technology "MS3" (source: Lucas & Bosch [93])

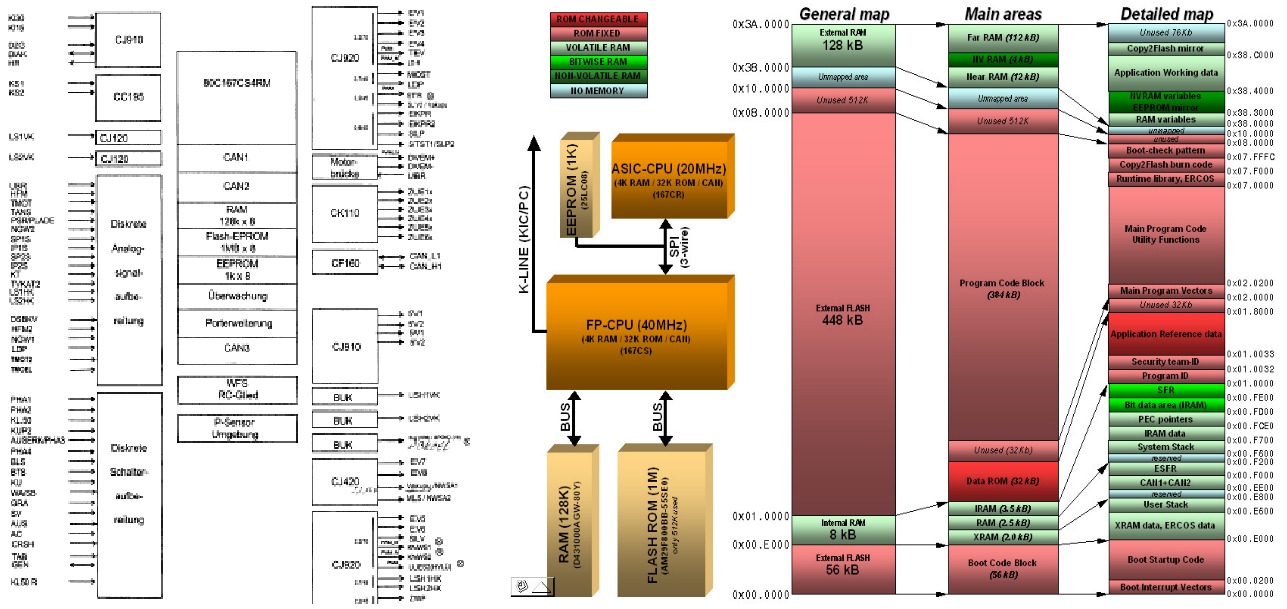


Fig. 2.2:51 – Internal “ME7.1.1” hardware and “ME7.6” software ECU structures (source: Bosch [303])

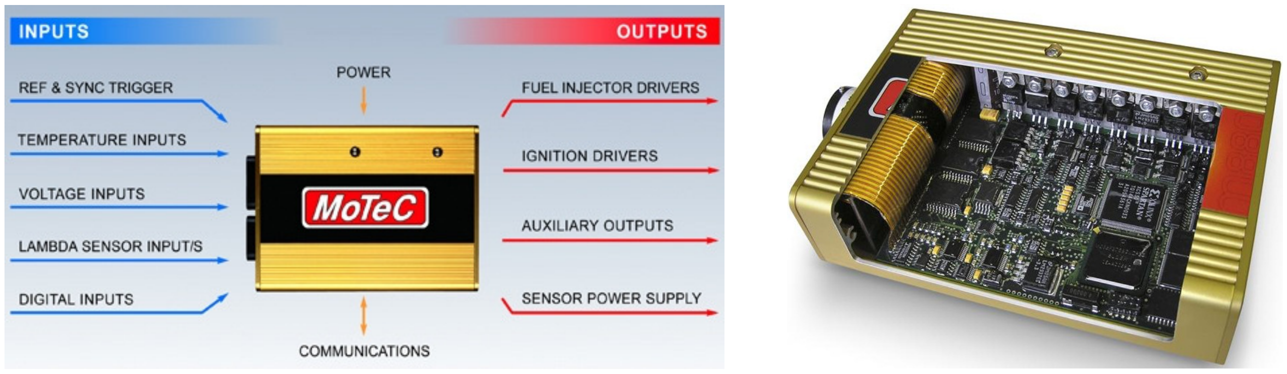


Fig. 2.2:52 – Motorsport MOTEC ECU insides (source: Motec [381])

There are also "multi-core" concepts involved in state-of-the-art series-car and even motorsport ECUs, composed mostly just by a master-CPU and a slave-CPU, where this later one just performs lower-level tasks such as processing inputs and outputs, PWMs, timings, etc. Examples are the previous ME7.1.1 and MS3 ECUs, where the ME7.1.1 image [306] explicitly shows the two CPUs (large flat-pack chips). The earliest TriCore [119] integrates 3 processing cores into a single chip. Now it even integrates one additional complex programmable timing core [415]. Real parallel processing composed by an arbitrary number of separate chips in scalable hardware/software architectures is not yet present in current series-car ECUs.

Fig. 2.2:53 shows the insides of a C167CR [178] automotive microcontroller. Besides the usual digital input and output ports, this type of microcontroller has many specialized hardware sub-components inside, to better suit the automotive functionalities' requirements, such as interrupts/traps, counters/timers, capture-compare units and ADCs. These hardware sub-components are used as follows (example setup/configuration including digital IOs and excluding communications, based on [308]):

- ***Interrupts/Traps:*** engine-speed sensors (crankshaft, camshafts), wheel-speeds, turbo-speeds, time-rasters, ADC conversions, PECs (DMAs), communication RX/TX buffers, non-maskable interrupt (NMI), stack underflow, stack overflow, illegal external bus access, illegal instruction access, illegal word operand access, undefined opcode, watchdog failure, among many other possible sources.
- ***Counters/Timers:*** crankshaft teeth counts, individual crankshaft teeth timing, time-rasters, free-running timers (for use inside capture-compare units), among other possible sources.
- ***Capture-Compare units:*** injection algorithms, ignition algorithms, time-rasters, wheel-speeds, turbo-speeds, knock-window generation.
- ***ADCs:*** temperature sensors (cooling liquid, ambient air, intake air, engine and gearbox oil, exhaust gas, fuel), pressure sensors (cooling liquid, engine and gearbox oil, crankcase, intake air, ambient air, fuel pump, common-rail fuel), lambda sensors (oxygen contents, temperature, before and after the catalyst), acceleration sensors, yaw sensor, steering angle sensor, internal ECU voltage supervision, throttle valve position, knock sensors.
- ***Digital ports:*** main battery relays, ignition-on switch, lambda sensor heating (PWM), injection valves, ignition coils, turbo wastegate, launch switch, fuel pump, malfunction indication light, pitlane switch, lap trigger, resonance flap, knock sensors, electronic throttle (PWM), common-rail valve (PWM), exhaust gas recirculation valve.

It is generally the ignition and injections functionalities that require most of the interrupts and capture-compare units, while sensors require most of the ADCs. ECU electronics also employs some of that specialized hardware for other internal purposes.

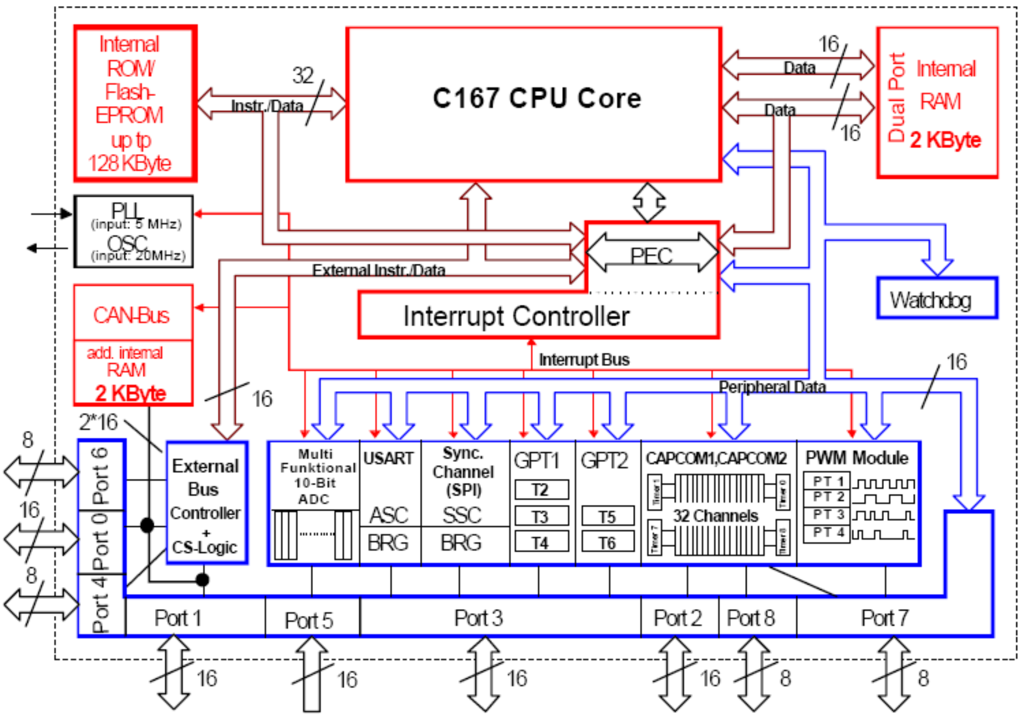
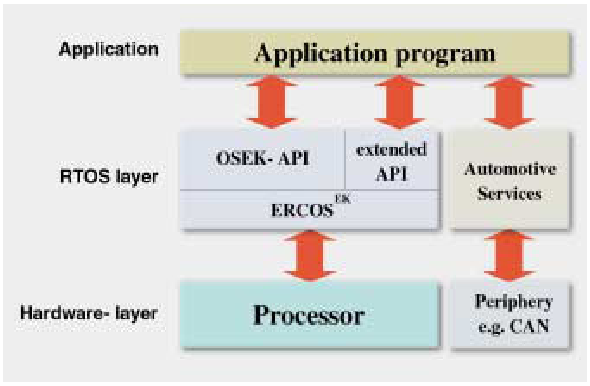


Fig. 2.2:53 – C167CR automotive microcontroller with all its specialized peripherals (source: Siemens [313] [314])

As for the operating systems that are still inside many series-car ECUs, there is the ERCOS^{EK} ("**E**mbedded **R**eal-time **C**ontroller **O**perating-**S**ystem") [424] package/layer (advancement over the older ERCOS [478]), which encapsulates all hardware interfaces and especially abstracts all timing details (left of Fig. 2.2:54). This OS was developed by ETAS [92] and has been used for many years. This OS is being slowly replaced by the more sophisticated and also much more complex AUTOSAR structures (right of Fig. 2.2:54) and way of programming an ECU, shown later on in this chapter. AUTOSAR often uses the more recent OSEK/VDX [444] and is now even developing its own "AUTOSAR--OS" [445] based on the previous OSEK/VDX.



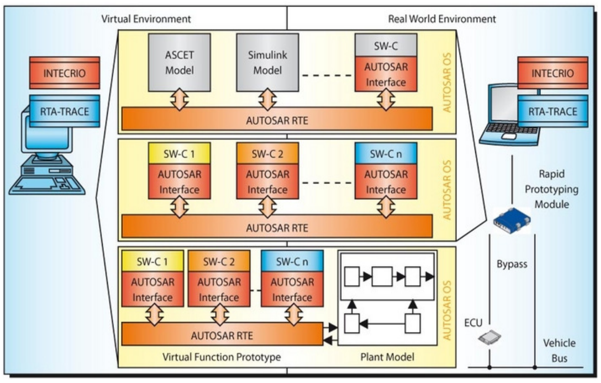


Fig. 2.2:54 – Left: ERCOS^{EK}; Right: AUTOSAR (source: ETAS [424], Autosar [68])

Fig. 2.2:55 shows an example of the inner actions of such OSs on the left. The previous OSEK/VDX also possesses some hook-based debugging strategies, which allow the developer to tap into the running tasks for inspection purposes (right of Fig. 2.2:55). This debugging "hooking" mechanism is further enhanced mostly by 3rd-party products such as *"ECU Interface Manager"* [417] similar to the previous *“No-Hooks OnTarget”* [300] and *"EHOOKS"* [223].

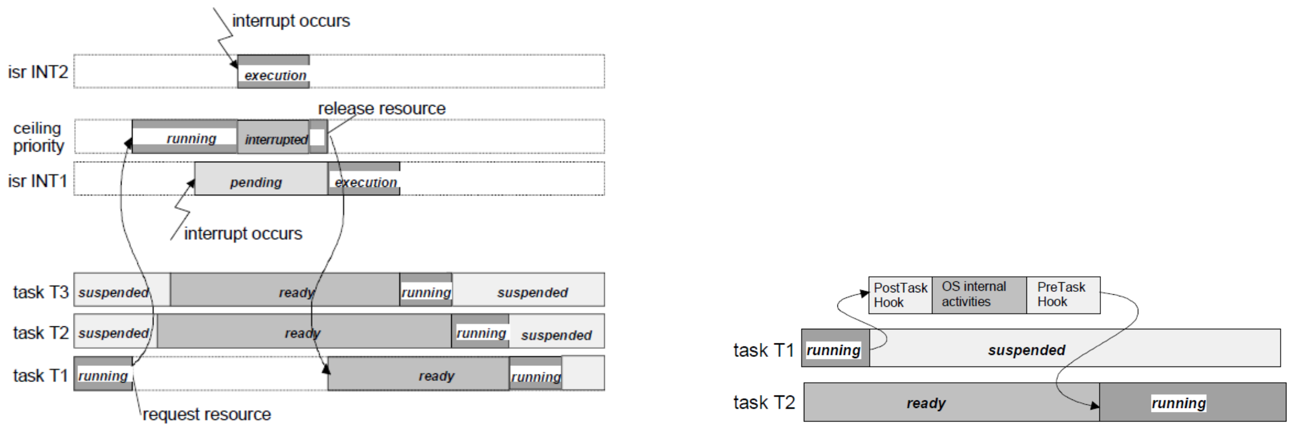


Fig. 2.2:55 – Left: OSEK/VDX in action; Right: OSEK/VDX debugging "hooks" (source: OSEK Group [444])

2.2.1.2 "ES" Series

[Emphasis: "rapid-prototyping"]

ETAS [7] offers the ASCET [7] graphical programming software, which seamlessly integrates in their also offered range of hardware platforms [224] (Fig. 2.2:56). This combination of software and hardware components allows for simulation and rapid-prototyping on the graphical level. The simulation and rapid-prototyping card in Fig. 2.2:57 employs an FPGA, aside of standard micro-controllers, to accelerate the digital processing part of programs. By "Rapid-Prototyping" it is meant that most changes can be made directly and easily on the ASCET graphical interface and then compiled/downloaded onto the hardware target, in a matter of minutes. This hardware platform is oriented toward series-cars prototyping and generally works under the use of "software-hooks" detailed earlier, where the programmer makes the necessary changes to the source-code, building and downloading it afterwards into this hardware.

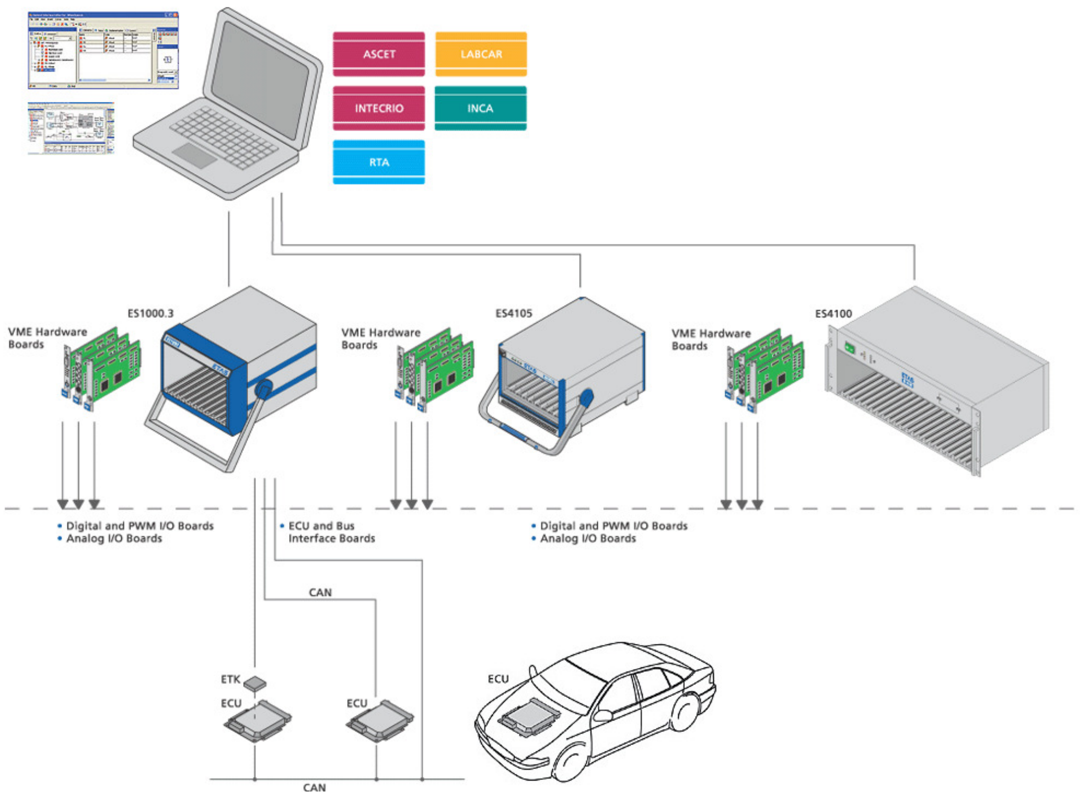


Fig. 2.2:56 – ETAS software development and hardware targeting overview (source: ETAS [224])

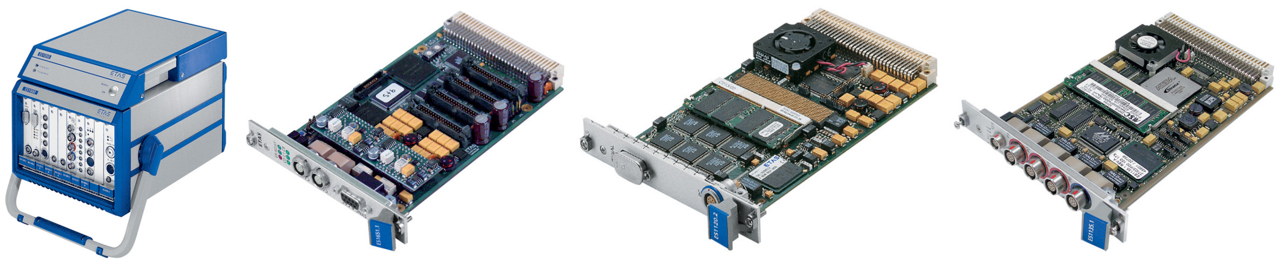


Fig. 2.2:57 – ES1000 housing hardware target system, internal ES1651 support card, ES1120 system control card and ES1135 simulation & rapid-prototyping card with Altera FPGA (source: ETAS [224])

2.2.1.3 MS5

[Emphasis: "rapid-prototyping"]

Bosch Motorsport offers the "MS5" [10] hardware target combination which allows similar functionalities as the previous "ES"-Series and ASCET combination. The main difference is that this combination uses Matlab Simulink [5] as the graphical programming interface. Similarly, by "Fast-Prototyping" it is meant that most changes can be made directly and easily on the Matlab Simulink graphical interface and then compiled/downloaded onto the hardware target, in a matter of minutes. This hardware platform (Fig. 2.2:58 and Fig. 2.2:59) is oriented toward motorsport racing.

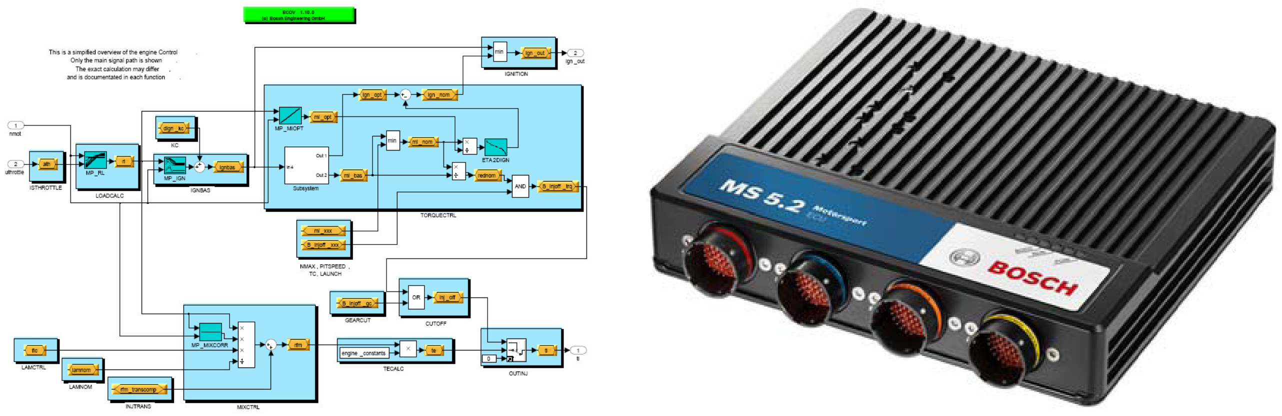

This is a simplified overview of the engine Control
Only the main signal path is shown
The exact calculation may differ
and is documented in each function
IGNITION
ISTHROTTLE
LOADCALC
IGNBAS
TORQUECTRL
GEARCUT
CUTOFF
TECALC
CUTINJ
LAMCTRL
LAMNOM
INJTRANS
MIXCTRL
MS 5.2 Motorsport ECU
BOSCH

Fig. 2.2:58 – "MS5" software and hardware combination (source: Bosch Motorsport [10])

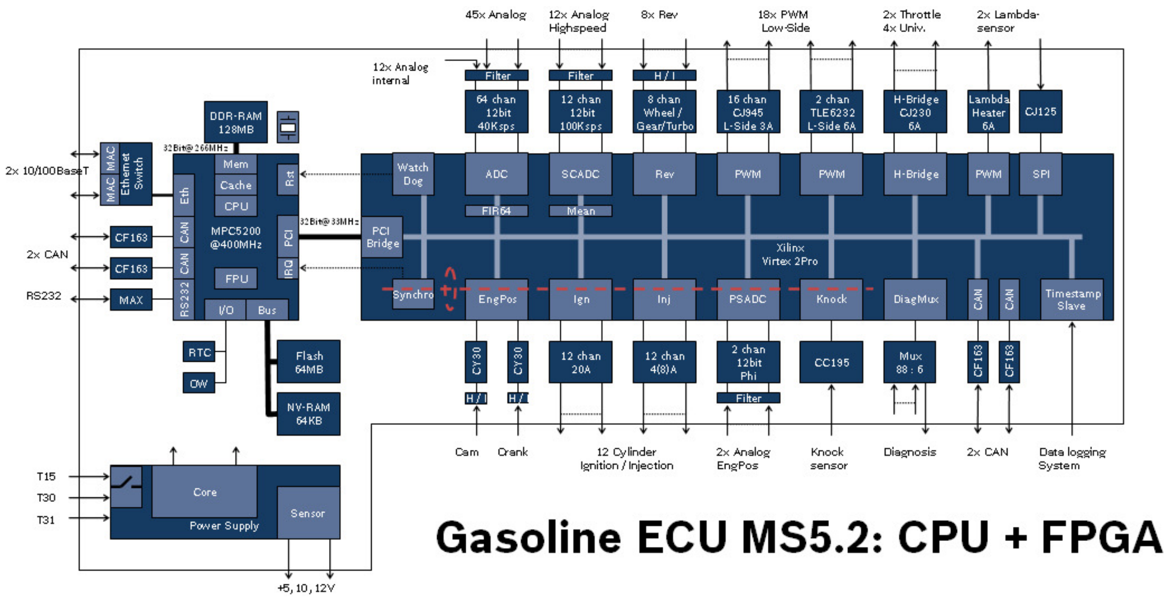
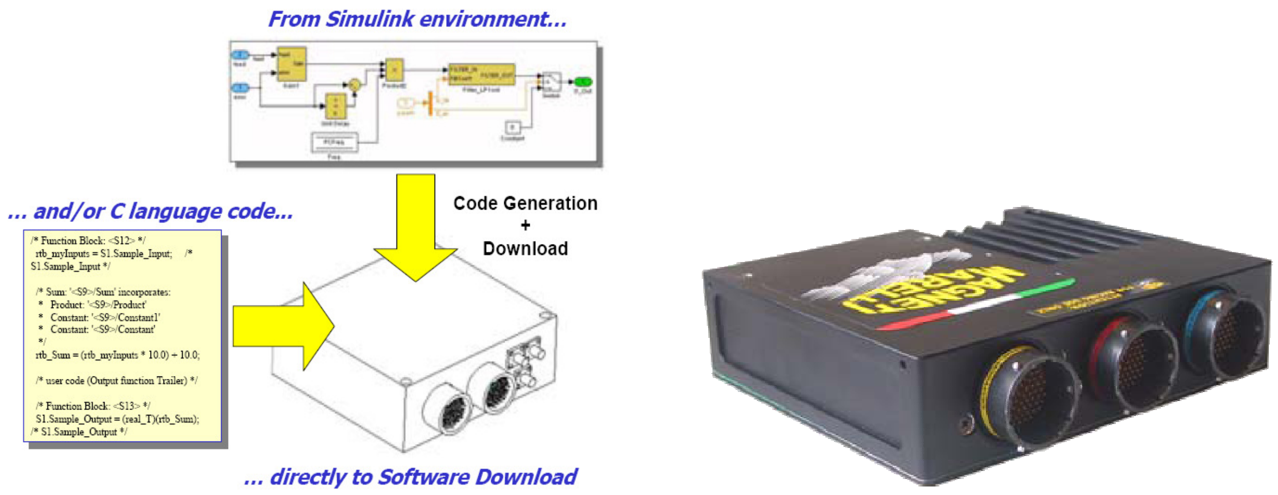


Fig. 2.2:59 – "MS5" internal CPU plus FPGA schematic structure (source: Bosch Motorsport [10])

2.2.1.4 FastPRO

[Emphasis: "rapid-prototyping"]

Magneti-Marelli offers the "FastPRO" [11] hardware target combination which allows similar functionalities as the previous "ES"-Series and ASCET, as well as the "MS5" and Matlab Simulink combinations. As in the previous combinations and mainly because all these solutions have auto-code generators involved in the translation of the graphical programs into hardware-understandable code, "C-"modules may also be included in this translation process. This hardware platform (Fig. 2.2:60) is oriented toward motorsport racing. This combination was even used for years in the Toyota Formula 1 racing team.


From Simulink environment...
... and/or C language code...
Code Generation
+
Download
... directly to Software Download

```
/* Function Block: <S12> */
  rtb_myInputs = S1.Sample_Input;     /*
S1.Sample_Input */

  /* Sum: '<S9>/Sum' incorporates:
   *   Product: '<S9>/Product'
   *   Constant: '<S9>/Constant1'
   *   Constant: '<S9>/Constant'
   */
  rtb_Sum = (rtb_myInputs * 10.0) + 10.0;

  /* user code (Output function Trailer) */

  /* Function Block: <S13> */
  S1.Sample_Output = (real_T)(rtb_Sum);
/* S1.Sample_Output */
```

Fig. 2.2:60 – "FastPRO" software and hardware combination (source: Magneti-Marelli [11])

2.2.1.5 RTI & RapidPro Series

[Emphasis: "rapid-prototyping"]

dSpace [4] is yet another company selling hardware targets for the Matlab Simulink graphical programming environment, by developing complete HIL (Hardware-In-the-Loop) and vehicle simulation software based on that programming tool, associated with the AUTOSAR [67] standards. Its FPGA-based hardware targets such as the DS5203 together with dSpace's Matlab Simulink based "RTI FPGA Programming Blockset" [100], allow to auto-generate FPGA code. This FPGA module may be used together with other processor modules such as the DSDS1005 PPC board. "RapidPRO" [118] are extension modules which allow to rapidly and easily extending the main hardware platform. This hardware platform (Fig. 2.2:61) is oriented toward series-cars prototyping and simulating. The DS1006 board [155] allows connecting to up to 19 other DS1006 boards through proprietary optical-fibre DS911 links (Fig. 2.2:62), thus allowing to create real highly interconnected multi-core systems. Besides this "external" multi-core processing possibility, these DS1006 also feature "internal" multi-core processing due to the use of quad-core Intel processors on the boards.

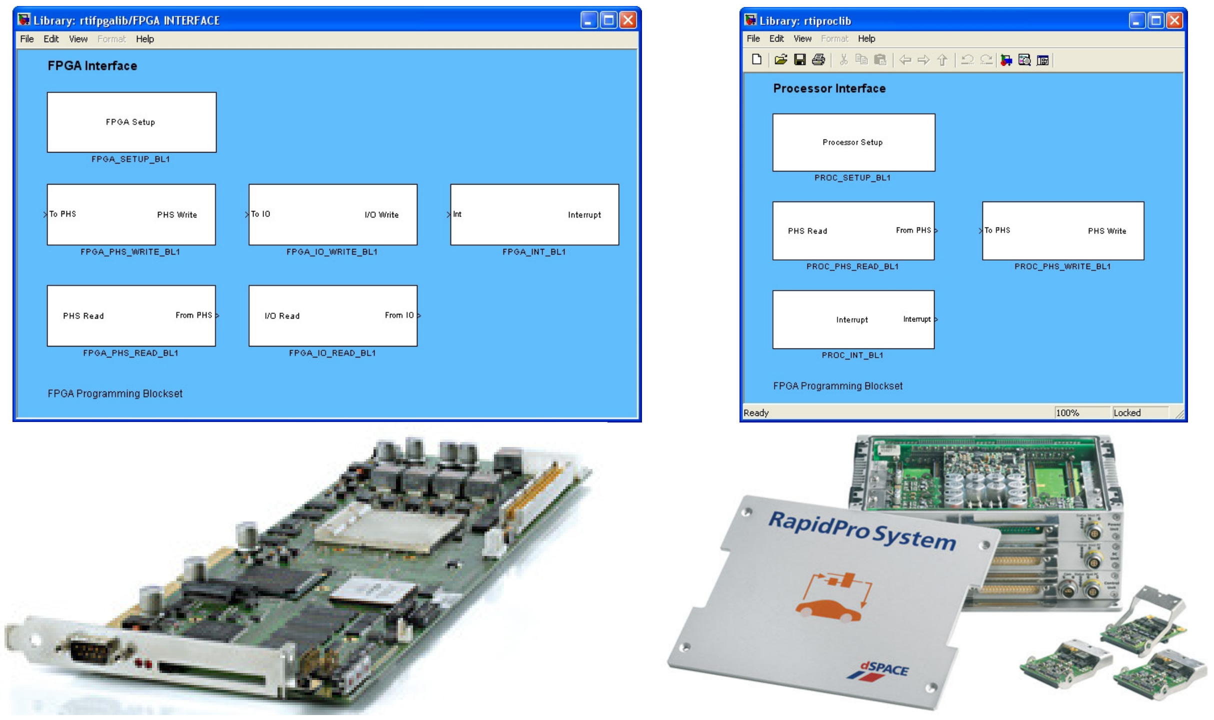


Fig. 2.2:61 – "RTI" software and hardware platform combination (source: dSpace [4])

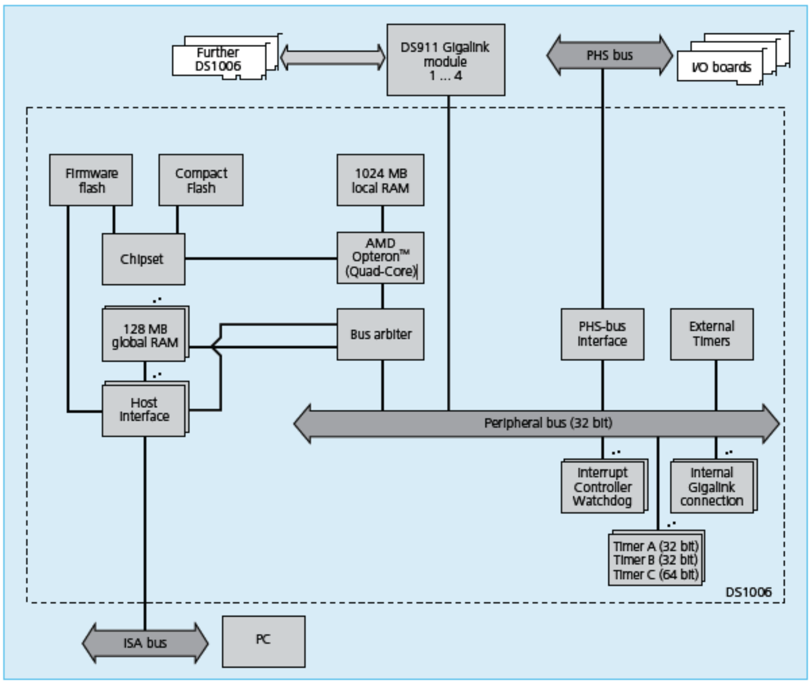


Fig. 2.2:62 – "RTI" software and hardware platform combination (source: dSpace [155])

2.2.1.6 PROtroniC

[Emphasis: "rapid-prototyping"]

AFT [408] manufactures the hardware platform and low-level controller interface development combination called *"PROtroniC"* [408]. As the previous platforms, it also allows the user to develop functionalities on highly flexible hardware. It also uses Matlab Simulink and Stateflow for the design of the functionalities. It uses dSpace's TargetLink or Mathworks' Real-Time Workshop Embedded Coder for auto-code generation. Refer to Fig. 2.2:63 for the internal structure of this solution.

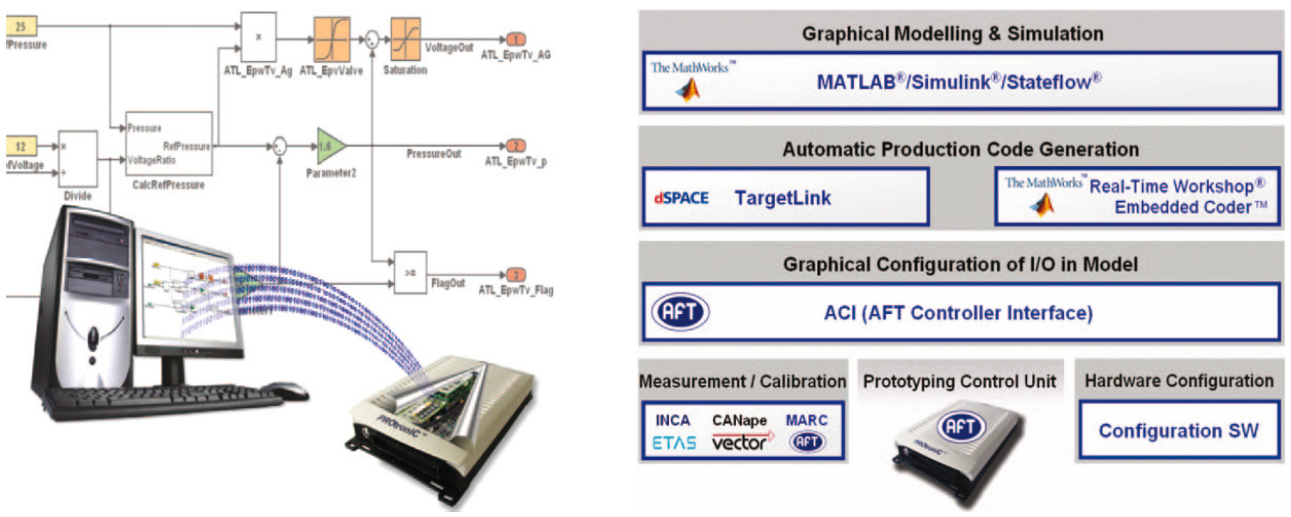
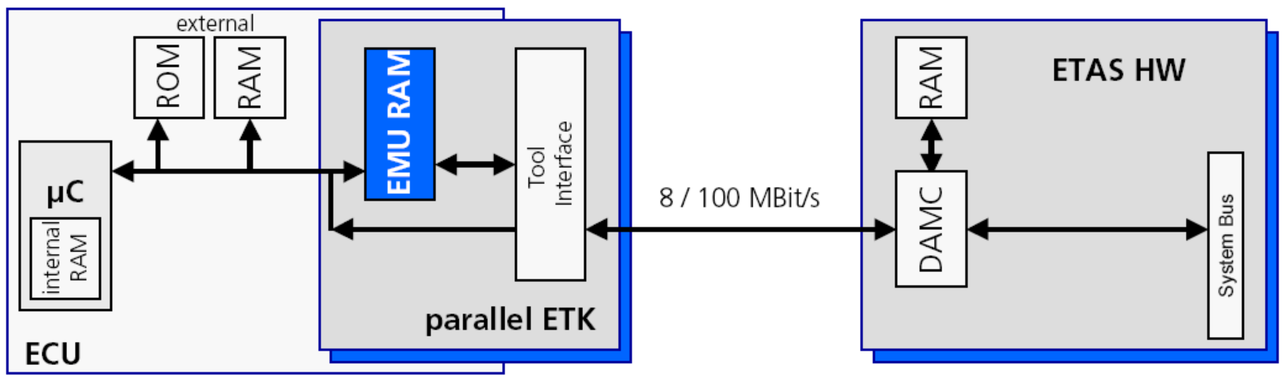

Graphical Modelling & Simulation
The MathWorks
MATLAB®/Simulink®/Stateflow®
Automatic Production Code Generation
dSPACE TargetLink
The MathWorks Real-Time Workshop® Embedded Coder™
Graphical Configuration of I/O in Model
AFT ACI (AFT Controller Interface)
Measurement / Calibration
INCA CANape MARC
ETAS vector AFT
Prototyping Control Unit
Hardware Configuration
Configuration SW
PROtroniC

Fig. 2.2:63 – "PROtroniC" software and hardware platform combination (source: AFT [408])

2.2.1.7 ETK

[Emphasis: "in-ECU hardware bypassing"]

ETAS [7] manufactures this universal (ETK = *Emulator-TastKopf*) automotive ECU hardware interface [134] add-on, which is a memory emulator representing a highly disruptive and physically highly intrusive solution intending to circumvent the complete complex software and hardware structures. The Ethernet-based ETK and XETK interfaces by ETAS provide direct access to the control variables and parameters of an ECU directly via the parallel data and address bus (memory tapping), or via a serial microcontroller testing or debugging interface (JTAG and others). In some cases, it also allows to directly manipulating and inspecting the hardware at micro-controller (registers, stack, ports), as well as allowing to datalog (tracing), step executing and breakpoint setting, among other functionalities such as code changes and bypass setting. Such systems are typical called "in-circuit" emulators or hardware bypassers (Fig. 2.2:64) by replacing the main memory or micro-controller and additionally mimicking the original CPU. A special software user-interface then allows for direct code and memory manipulation. It should be emphasized that this hardware piece is an add-on and, as such, exists parallel to the development environment as a debugging/inspecting facilitator and nothing more. It is very useful and necessary for effective debugging ECUs where the mentioned development environment is so complex that it turns integrated debugging an impossible or too difficult task. This causes the necessity to have to toggle between the development and the debugging environment to accomplish this.



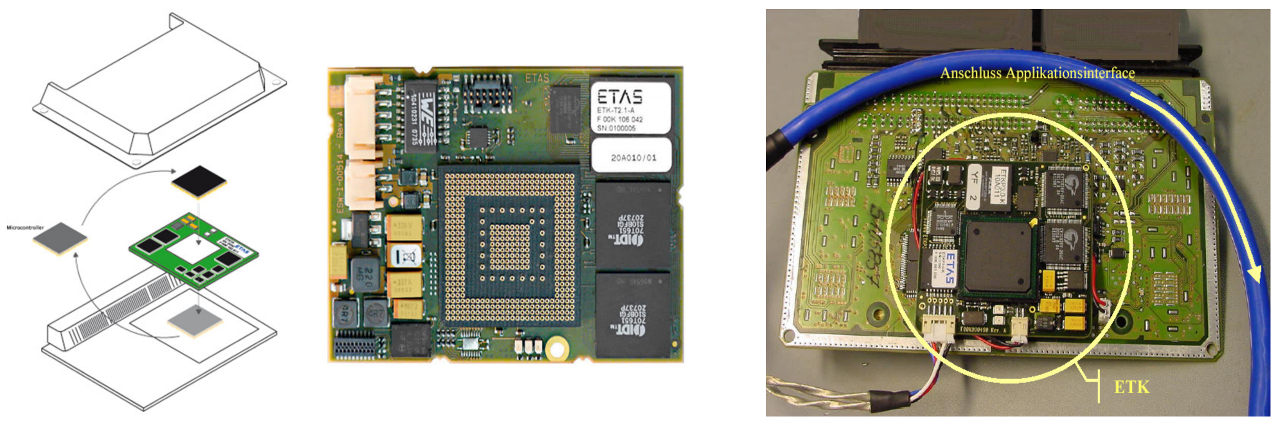


Fig. 2.2:64 – ETK device for Infineon's C167 and Freescale's MPC5500 processors (source: ETAS [134])

2.2.1.8 dProbe

[Emphasis: "in-ECU hardware bypassing"]

HiTEX [135] also develops automotive ECU interfaces (Fig. 2.2:65) which allow essentially the same as the previous ETK interfaces from ETAS. As the previous ETK hardware interface, this dProbe is also just a highly intrusive add-on which must be used in parallel with the original development environment. Basically, it replaces the original processing element with a more advanced counterpart, a so-called "bound-out chip", where internal CPU resources such as registers are bounded out to the dProbe board, for attached software to be able to easily inspect and change the related values.

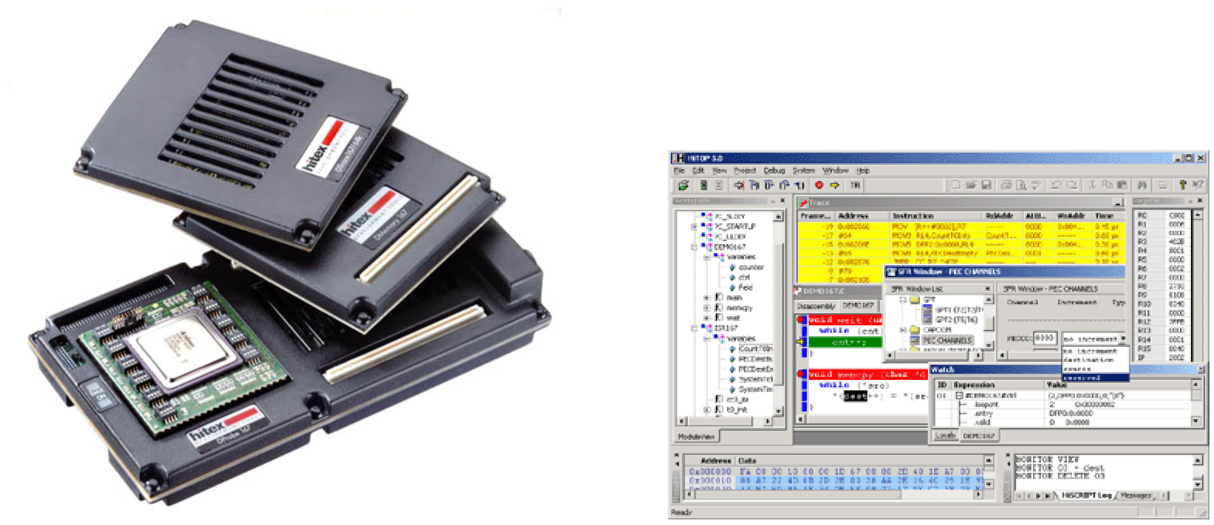


Fig. 2.2:65 – dProbe device with its user-interface (source: HiTEX [135])

2.2.1.9 Lauterbach

[Emphasis: "external/internal hardware debuggers"]

Besides in-circuit emulators, Lauterbach [221] also develops and manufactures highly advanced JTAG debuggers and in-circuit emulators (Fig. 2.2:66) used in automotive systems. Compared to in-circuit emulators, JTAG debuggers are less intrusive and drastic software/hardware tools that allow the user to also communicate, inspect and manipulate the CPU through the well-established JTAG protocol [222]. Although less intrusive, this solution is potentially affected by JTAG problems and crashes. Other than that, it allows similar flexibility and handling like in in-circuit emulators.

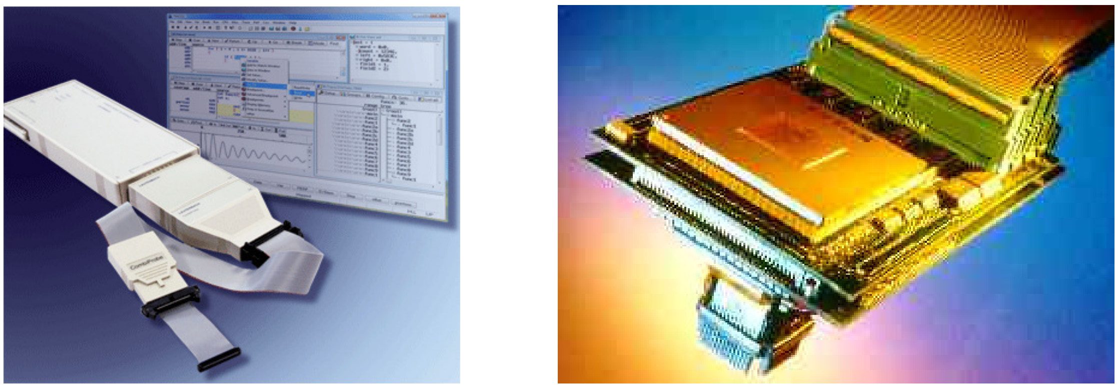


Fig. 2.2:66 – Left: external JTAG debugger; Right: in-circuit emulator (source: Lauterbach [221])

2.2.1.10 Freescale's MC33810

[Emphasis: a truly dedicated chip with dedicated processing operations]

Last but not least in this automotive hardware portfolio, there are intelligent peripherals driving chips with some very advanced and dedicated processing assembly possibilities.

It has been quite a while where normal power driving chips for injection valves and ignition coils, as well as for power management, existed. They have integrated diagnosis, SPI or some other simple communications interfaces, for the main ECU processor to get diagnosis information and program some simple settings. Some Bosch chip examples are the CJ220 (aka Freescale MC33186, H-bridge power driver), CJ420 (power driver), CJ910 (power management, engine-speed detection), CJ920 (power driver), CK110 (ignition coil drivers), MC33810 [464] (left of Fig. 2.2:67), etc.

There is also the Bosch CJ1xx series of oxygen-sensor driving chips such as the older CJ110, CJ120 and CJ125 [465], while the newest CJ136 [466] hit the market in 2011. Both the currently used CJ125 (centre of Fig. 2.2:67) and CJ130 possess analog electronics inside, one part that drives the sensor's heating resistance and the other that evaluates the oxygen contents of the exhaust gas through special oxygen ion pumping currents. Just like in the case of the previous power driving chips, no programmable processing is done here.

The CC195 [467] (right of Fig. 2.2:67) is yet another example of integration of analog processing for complex sensors such as the evaluation of knock sensors is. Same as before: no programmable processing is done whatsoever. Everything is fixed and the programmer can only set some parameters to adapt the behaviour of the inner electronics of the chip to the current engine management algorithms' needs.

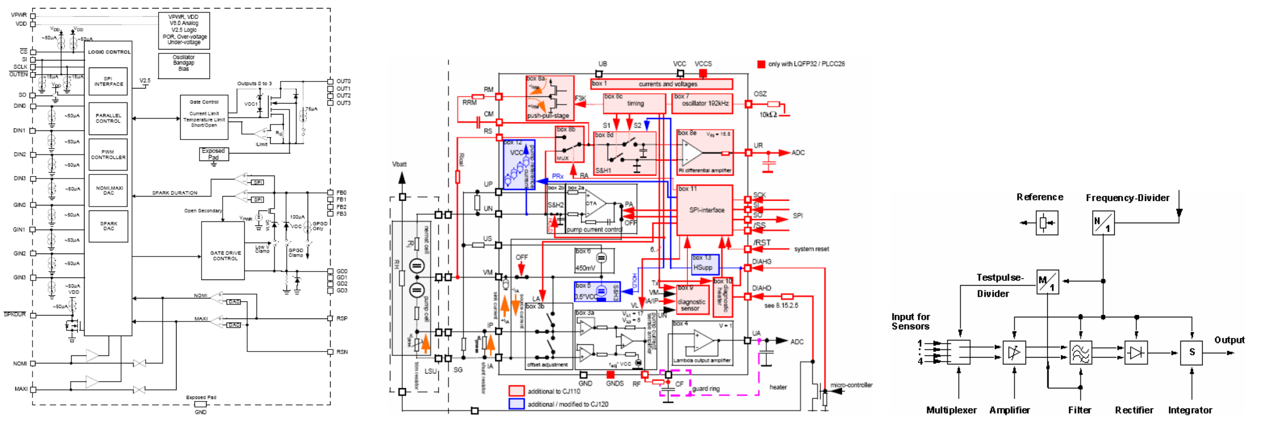


Fig. 2.2:67 – Left: MC33810; Centre: CJ125; Right: CC195 (source: [464] [465] [467])

These chips already have some intelligence and diagnosis capabilities inside them, to unburden the main ECU processor. Simple communication protocols allows the main processor to talk to these chips, set parameters and get diagnosis information. All high-speed related mechanisms are thus outsourced into these chips that take care of everything around the respectively connected actuators and sensors. But there are lots of more advanced possibilities if these chips possess some kind of autonomous programmable processing capabilities inside them. This is the case of the highly advanced and highly complex gasoline direct-injection power driver chips such as Freescale's MC33816 [468] [499], which integrates power driving, autonomous programmable processing, analog/digital circuitry and even a DC-DC converter driver. This chip supersedes the older CY202 that had only settable parameters but no programmable processing whatsoever. Now this MC33816 is a powerhouse on its own. It drives up to 6 high-pressure high-voltage/current injectors, one "quantity control valve" (QCV - the one that regulates pressure on the fuel-rail) and one DC-DC converter. It then consists of 4 sequencers (simple processors), which are capable of processing specific automotive single-cycle instructions with an instruction execution time of down to 166[ns]. These 4 processors thus allow the automotive developer to integrate fast and complex control and signal-shaping algorithms directly inside the power driver itself, getting that burden completely off the main ECU processor. Diagnosis mechanisms are therefore also possible to be freely programmable inside the same chip. Fig. 2.2:68 shows the inner details of this chip, along with the complex current waveforms it must generate, to successfully drive a high-pressure solenoid injector.

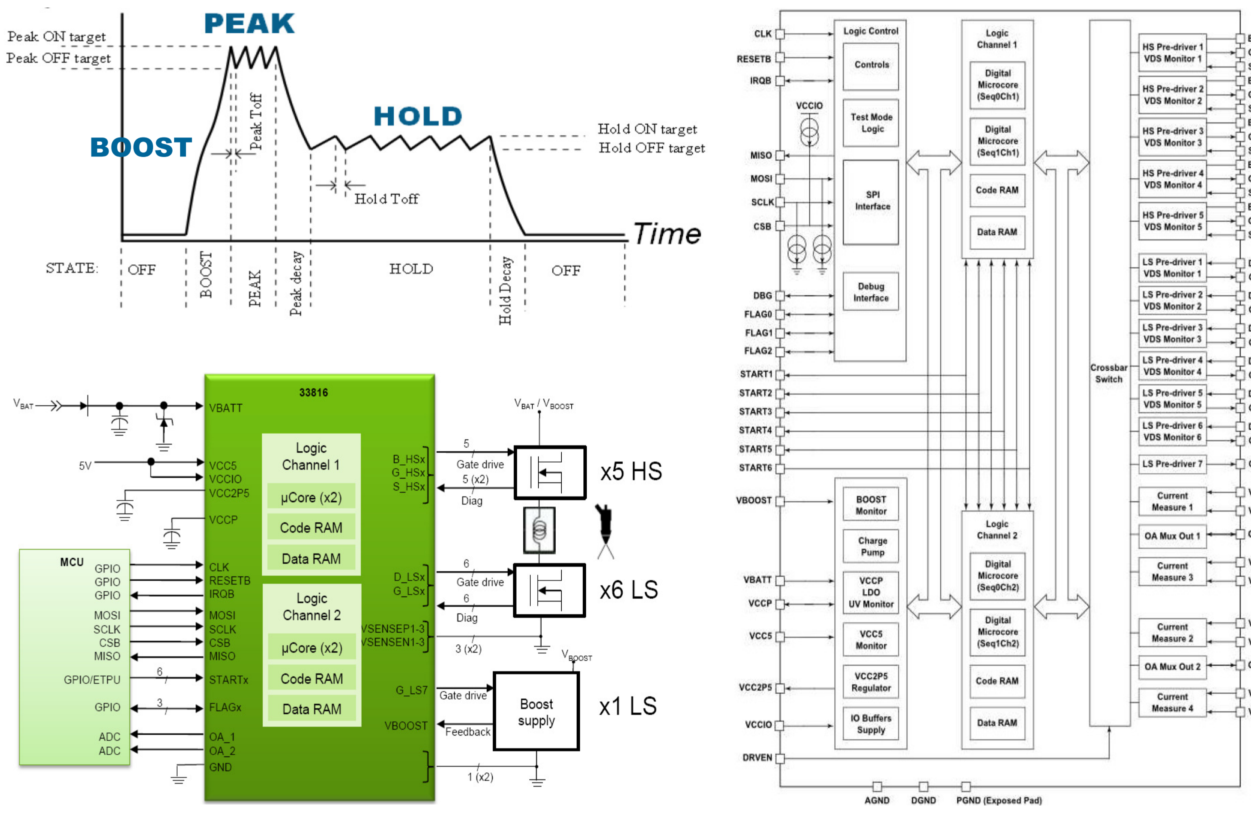


Fig. 2.2:68 – Inner details of the MC33816 programmable injection driver chip (source: Freescale [468])

The most important feature of this chip is without doubt its programmable processors. It contains 4 "sequencers", which execute machine-code encoded programs out of its

2x1KWord code RAM. A data RAM with 64 Words handles variables and parameters. Comprising 2 logic channels, these "sequencers" are capable of executing a very set of special and automotive-specific Assembly language, which of course contains general-purpose instructions (jumps, loads, stores, shifts, logic and arithmetic operations, etc.) as well as the specific injector, QCV and DC-DC converter instructions (ADC and DAC manipulation, multiplexer/demultiplexer settings, amplifier gains, diagnostic thresholds and bias currents configuration, event waiting, general voltage and current configurations, DC-DC automatic hardware regulation, output drivers settings). In other words, instead of doing things as general-purpose micro-controllers do, i.e. with memory-mapped peripherals, those specific instructions operate directly on those peripherals that are only accessible through those specific instructions. Fig. 2.2:69 shows the internal architectural as well as software details/models of each of the 4 sequencers comprising each MC33816. Notice the right image showing the "current feedbacks" and the "voltage comparator feedbacks", which directly combine with the specific automotive Assembly instructions. All analog/digital details of the peripherals driving/sensing the injectors, the QCV and the DC-DC converter, are set and commanded through this direct method by using specific MC33816-Assembly mnemonics especially optimized for the following operations (please note that ach specific mnemonic executes complex internal procedures to respond to the programmer's needs, without any hassle with internal hardware details surrounding them):

- **ADC and DAC manipulation** to read out or to set them, as well as selecting their reference voltages and other operational details. No worries about settling and conversion times, as the specific instructions already include/hide this kind of hardware details. The programmer just has to use those instructions whenever needed, without any other worries.
- **Multiplexer/demultiplexer settings** allow for the ADCs and DACs to receive/send voltage from/to specific sensing resistors and/or analog amplifiers. These multiplexers also control which signals are used from the external pins into which sequencer, as well as which sequencer or internal OpAmp signals reach the external pins, for debugging and measuring.
- Several **OpAmps/Amplifiers** are available for amplifying current sensing resistors as well as comparators for diagnosis thresholds surpassing detection. Their gains can be freely set according to the needs and they can also be automatically calibrated using just the specific instructions available. These then execute internally automatic sequences to accomplish calibration purposes, etc.
- Diagnosis **voltage thresholds** and **biasing currents** can be set through voltage generators and current sources to be used with comparators to detect implausible voltages and currents at the power transistors, such as in the case of short-circuits to battery, short-circuits to ground, short-circuits over the load and open-load.
- An **event waiting table** turns out to be extremely useful during power output current waveform shaping, as up to 5 events may be pre-programmed into this table and then waited upon, stopping the sequencers until one of the selected events occurs. Some examples of these events are voltage threshold surpass detection, timing delays and just plain 2-point current threshold regulation over/under-current detections.
- **Direct hardware support for automatic DC-DC 2-point regulation** allows for high-speed high-precision boost-current regulation, although a sequencer/software oriented "manual" regulation is also possible and used in specific actuation situations. This mechanism is also operated with specific Assembly mnemonics.
- **Output drivers' settings** may be configured in some ways, including the desired slew-rates, shortcut labels for the code itself, as well as other details.

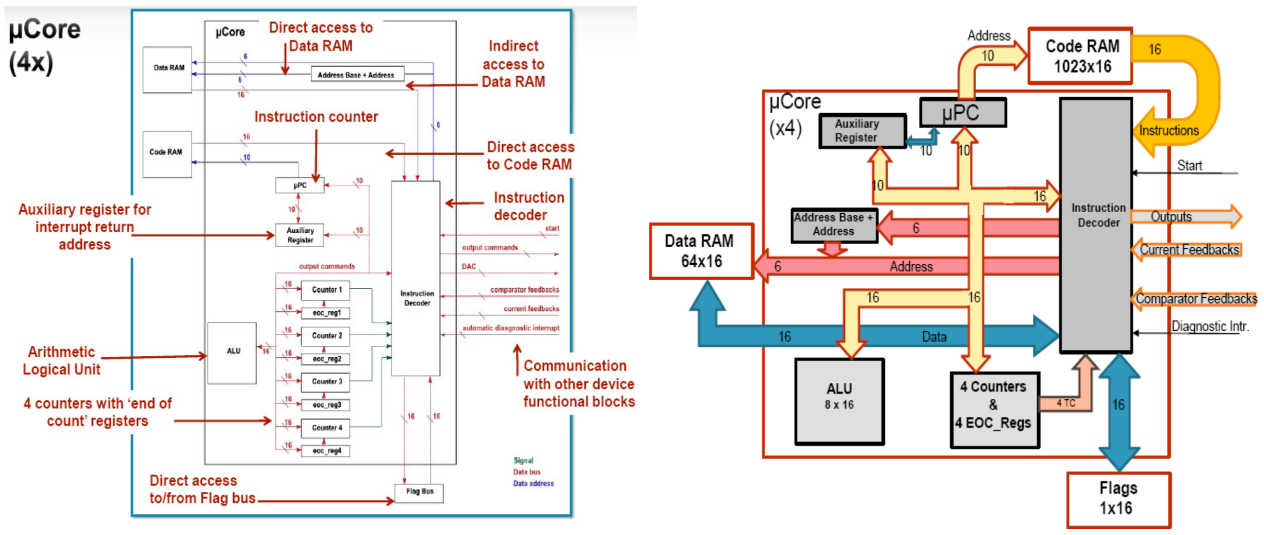


Fig. 2.2:69 – Inner architectural and software details of each of the 4 sequencers (source: Freescale [468])

At the end, the main differentiating feature of this MC33816 chip is its capability of being programmed with a special Assembly language, allowing the developer to place his algorithms inside the very chip. Some of these Assembly instructions are explicitly presented herein below, because of the special fact of them being specifically designed for this chip, allowing for direct manipulation of the various software and hardware components, instead of being just plain general-purpose Assembly. This is thus a very interesting and specific programming language:

- **bias** – enables/disables MOSFET bias current structures
- **chth** – changes threshold on Vds/Vsrc feedback comparators
- **dfcsct** – define shortcut for MOSFET current feedback
- **dfsct** – define shortcut for MOSFET pre-driver output
- **slfbk** – select pre-driver feedback source
- **stdcctl** – set DC/DC converter control mode
- **stdm** – set DAC registers access mode
- **stfw** – set MOSFET freewheeling mode
- **stgn** – set OpAmps gains
- **stoc** – activate automatic OpAmps offset compensation
- **stslew** – set OpAmps slew rate
- **endiag** – enable MOSFET drivers diagnostics
- **cwef** – create wait-table conditional entry (current, timing or diagnosis events)
- **wait** – wait until an event from a wait-table is detected

Besides comprising a highly integrated and intelligent "smart peripheral" and besides being able to autonomously execute fast algorithms, this MC33816 chip is also fully capable of autonomously control a DCDC converter without the usual µCode intervention. This is possible because of included dedicated hardware structures allowing for a direct control, instead of having to execute µCode to get things done. Besides unloading the sequencers, this method also allows for much faster control cycles (about 50[ns]) when compared to the fastest µCode cycle-times (1000[ns]). All in all, this chip goes fully in the direction predicted as necessary in [12], while this special DCDC hardware feature allows this chip to take the "the hardware should do what it is told to do" concepts [12] even a step further. At Bosch, this chip is called "SC900" and is employed inside Bosch Motorsport's MS5 [10] and other ECUs.

2.2.2 Considerations about non-automotive hardware platforms

[Emphasis: platforms not used for automotive vehicles but also very interesting]

The same way as it happens in the software realm, there are also some highly interesting commercial and research non-automotive hardware platforms. Some of their features will also strongly relate to many details of the work developed herein. Again, special attention should be put on platforms such as the Jazelle DBX processor and Transputers, as well as the Sinclair ZX Spectrum domestic/play micro-computer. These and other examples have also been briefly investigated in the extra [Att. 6], by presenting their most interesting details for the scope of this thesis and, similarly to the non-automotive software tools, these details will also be taken into account during the work developed herein.

2.3 Integrative Combinations

[Emphasis: trying to reunite what was once separated]

The previous software tools and hardware platforms have been used by automotive system's manufacturers at such a rapid rate and in such an intermixed fashion, that inevitably many interfaces and standards have appeared, making interfacing between different systems almost impossible to achieve. With the increasing need of re-using and integrating different manufacturers' systems for economical as well as opportunistic reasons, some integrative approaches appeared in the automotive scene.

2.3.1 INTECRIO

[Emphasis: soft integration]

This integration platform [69] [399] from ETAS [7] is capable of gathering automotive software functionalities programmed in both ASCET and Matlab Simulink, as well as classical "C"-code modules. It forms a common interface at these high levels (Fig. 2.3:70 and Fig. 2.3:71) and then communicates with the hardware platforms connected to the AUTOSAR Run-time Environment [67] which runs on standard ETAS hardware components. Bottom-line, it is an integration software interface between existing high-level tools and AUTOSAR, allowing for debugging tools to coexist as well. This is a soft integration attempt, since all tools, interfaces and components continue to exist and to be developed independently from each other. Basically, this integrative approach attempts to fit SiL, HiL and MiL possibilities under the same hood. Notice also the co-simulation possibility illustrated in Fig. 2.3:71, allowing other vendors' (other than Mathworks and ETAS) code and model software to co-exist with all other previous integrated possibilities.

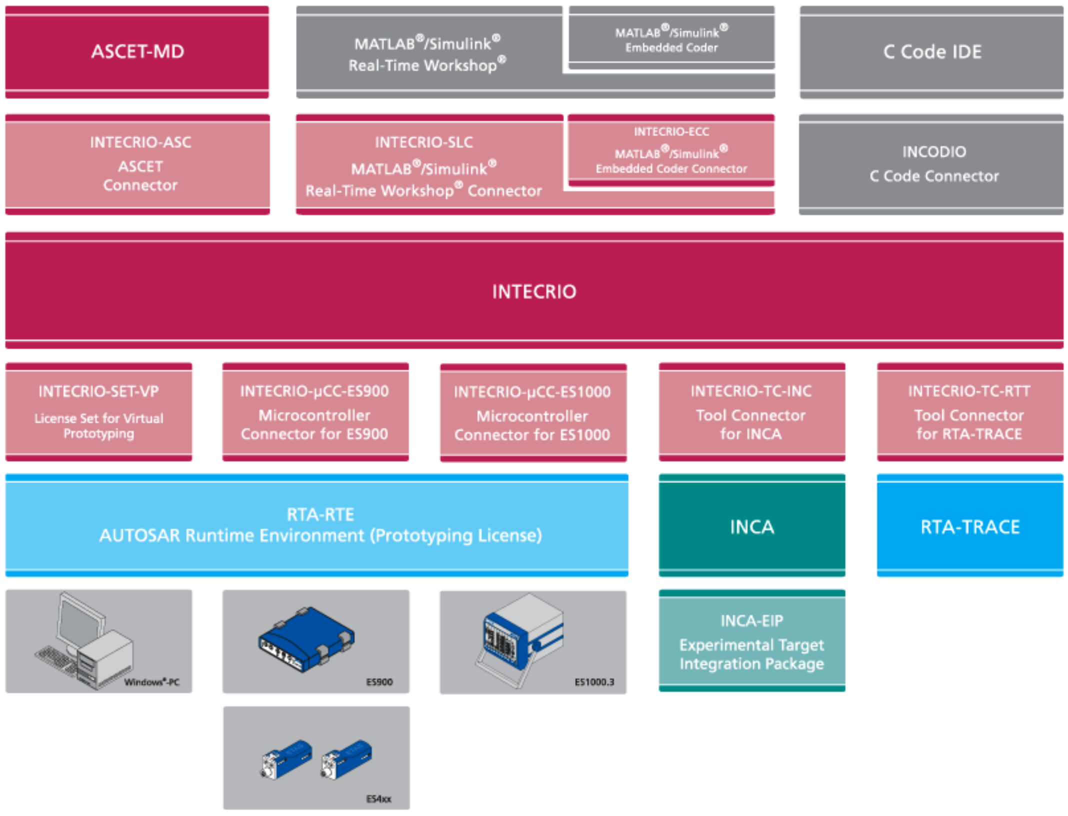

ASCET-MD
MATLAB®/Simulink® Real-Time Workshop®
MATLAB®/Simulink® Embedded Coder
C Code IDE
INTECRIO-ASC ASCET Connector
INTECRIO-SLC MATLAB®/Simulink® Real-Time Workshop® Connector
INTECRIO-ECC MATLAB®/Simulink® Embedded Coder Connector
INCODIO C Code Connector
INTECRIO
INTECRIO-SET-VP License Set for Virtual Prototyping
INTECRIO-µCC-ES900 Microcontroller Connector for ES900
INTECRIO-µCC-ES1000 Microcontroller Connector for ES1000
INTECRIO-TC-INC Tool Connector for INCA
INTECRIO-TC-RTT Tool Connector for RTA-TRACE
RTA-RTE AUTOSAR Runtime Environment (Prototyping License)
INCA
RTA-TRACE
Windows®-PC
ES900
ES1000.3
INCA-EIP Experimental Target Integration Package
ES4xx

Fig. 2.3:70 – INTECRIO common interfacing integration platform (source: ETAS [69])

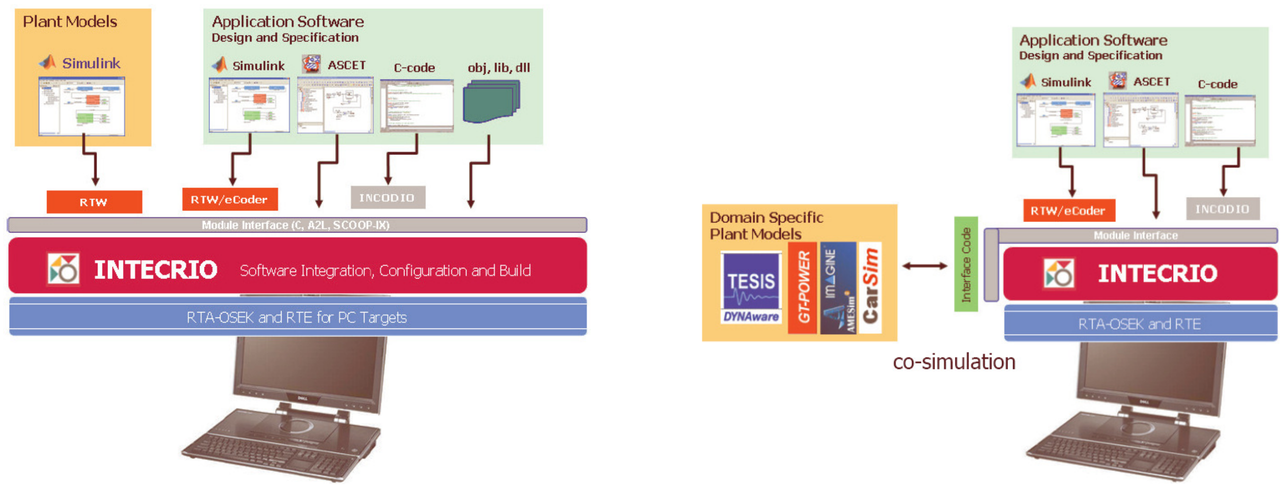


Fig. 2.3:71 – INTECRIO common interfacing integration platform (source: ETAS [69])

2.3.2 AUTOSAR

[Emphasis: many interfaces and components, one standard, soft integration]

Large efforts are being made to standardize the overall existing software and hardware structures, components and interfaces, attempting to reach a common standard among most important car- and tool-manufacturers. The strongest attempt is still being made by the AUTOSAR consortium (**AUT**omotive **O**pen **S**ystem **AR**chitecture) [67], supported by the major automotive manufacturers such as BMW, Bosch, Daimler, PSA/Peugeot/Citroën, Ford, VW, GM/Opel and Toyota. AUTOSAR provides a middleware run-time environment that will abstract hardware dependencies to a common denominator, allowing application software to be the sole differentiator. It aims packaging all OEM vehicle network strategies into a single off-the-shelf solution, specifying the interface details for the non-competitive technologies of vehicle networking software and also specifying a real-time operating system which is also considered to be a non-competitive technology. This standard package should then be mandatorily merged into existing tools to achieve a common multi-compatible, multi-interfacing structure (Fig. 2.3:72), thus enhancing interoperability among different components. The existing multitude of tools, components and interfaces continue to exist.

Like INTECRIO, this is a soft integration attempt, since it does not attempt to modify existent tools but rather attempts to standardize their interfaces and inter-relations. It favours auto-code generating tools and their interoperability. In other words, just like INTECRIO, it is a "wrapping-based" approach [439], in that all available components are wrapped by a single standard. It is further composed by the "Application Layers" which consists of *runnables* or schedulable application entities that the AUTOSAR system integrator then maps into tasks managed by the underlying AUTOSAR-compliant operating system, by AUTOSAR's "Run-Time Environment" (RTE) which provides glue-code/interfaces for signal exchange between modules of the previous layer, and the "Basic Software Layer" which is the AUTOSAR-compliant operating system finally executing the above *runnables*. In 2009 AUTOSAR released version 4.0 of its base-software where it supports multi-core embedded systems for the first time. Fig. 2.3:73 in [296] shows an example of how purely heuristic strategies, based on timing and inter-dependence details, are used in this software to allow for parallel executing *runnables*. Starting with version 4.0 [367], AUTOSAR also enables timing methods to be integrated into its software structure, thus driving standardization even one step further.

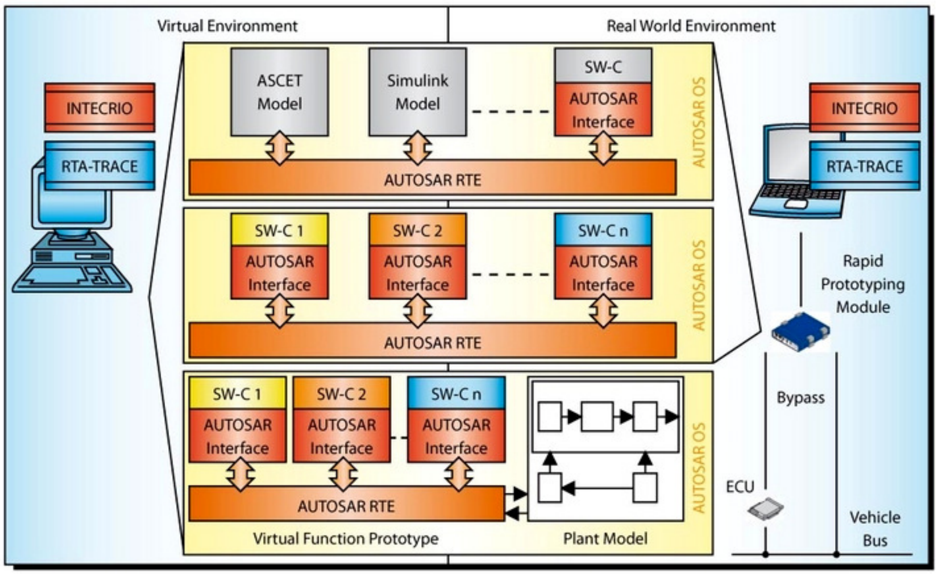


Fig. 2.3:72 – AUTOSAR common interfacing integration platform (source: Elektroniknet [67], ETAS [68])

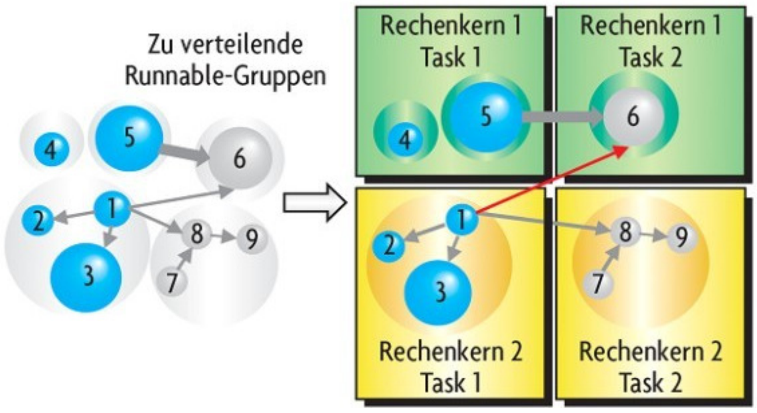


Fig. 2.3:73 – AUTOSAR v4.0 runnables distribution onto multi-core platforms (source: Elektroniknet [296])

2.3.3 UML

[Emphasis: auto-code generation directly out of UML]

Due to the fact that state-of-the-art tools use a great variety of programming languages, both visual and textual, as seen in previous sections, some attempts and combinations [116] [129] [130] to have a unique multi-modelling/development language have been made by using UML [115] and the derivative SysML [131]. These attempts also try to cope and get a grip onto the large increase in software development in the past decades, by giving industry a standardized view of data-structures and program representation. These UML attempts are thus used as a front-end or tying-language on top of the previous visual tools, by representing program entities such as classes, state-machines, sequence-diagrams, etc., as well as use-cases, architecture diagrams, etc. Some attempts even create meta-layers composed by multiple internal abstractions and concretization layers [132] [133], turning these tool-chains into ultra-complex and multi-dimensional software architectures (example on Fig. 2.3:74).

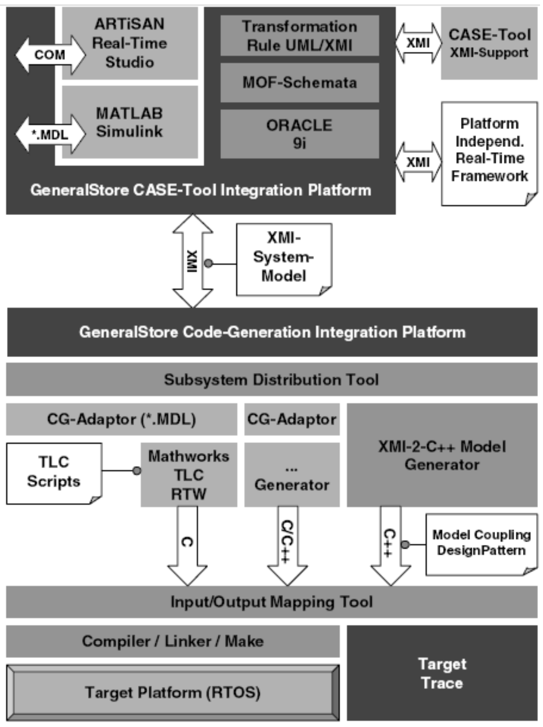
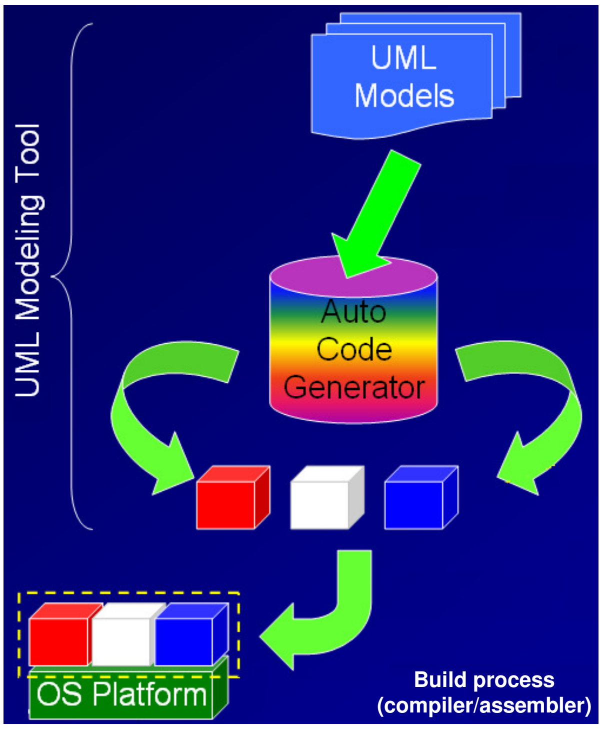

ARTiSAN Real-Time Studio
COM
MATLAB Simulink
*.MDL
Transformation Rule UML/XMI
MOF-Schemata
ORACLE 9i
XMI
CASE-Tool XMI-Support
Platform Independ. Real-Time Framework
GeneralStore CASE-Tool Integration Platform
XMI-System-Model
GeneralStore Code-Generation Integration Platform
Subsystem Distribution Tool
CG-Adaptor (*.MDL)
CG-Adaptor
TLC Scripts
Mathworks TLC RTW
... Generator
XMI-2-C++ Model Generator
C
C/C++
C++
Model Coupling DesignPattern
Input/Output Mapping Tool
Compiler / Linker / Make
Target Platform (RTOS)
Target Trace


Fig. 2.3:74 – Left: GeneralStore CASE integration tool; Right: (source: [132])

All in all, these attempts aim at a global representation that focuses on the functionality and behaviour of a program or system and that is agnostic to the underlying platform, OS or programming language used at the end. Using automatic code-generation to output low-level code from these high-level modelling languages is the biggest leap also attempted here [427]. Much research, fine-tuning and efficiency increase has to be made, before such attempts get really useful, since plain hand-coding is still much more efficient in all ways. Either way, auto-code generation here is not much different from auto-code generation in [5] [7] [165] and any other "C" based code-generation, with all its common advantages and disadvantages.

Generally, as clearly stated in [161], approaches such as this UML and other modelling approaches used in industry are not good enough. They are not based on a proper theory and sufficient formalization, furthermore failing when trying to program an ECU with that approach due to the lack of necessary and efficient language constructs and target support. The obvious result is that tool support is ineffective and the precision of the modelling is insufficient and grossly apart from real development needs. These generalist modelling languages do not even improve the quality of the development processes and of the software intensive products. This is because it does not give precise and uniform views on systems. Matlab Simulink [5], although already being a little better and more appropriate than UML, never quite made it past ASCET-SD [7], which was explicitly developed to best represent automotive applications.

2.4 Endnote

[Emphasis: chosen technological references]

It should be stated that all of the state-of-the-art software tools and hardware platforms are going to be considered throughout the unwinding of this thesis. Of course, not all of them will receive the same attention and level of importance, so that only a few technologies will be considered as top-references during this work. The software and hardware technology state-of-the-art references (detailed above) most considered for this work are the following:

- **"ASCET" (graphical data-flow iconic representation** reference**)**: this software tool is taken most into account when envisaging the automotive development environment software that is going to be done under this thesis. Matlab Simulink and LabView are also strong references but ASCET is graphically the most adapted tool to the automotive scene. These three tools share strong similarities in respect to their data-flow nature of building system models, as this thesis' work will too. *Key-ideas: iconic representation of operations, wires transporting values, wires connecting operations, very homogeneous looks.*
- **"LabVIEW"** (**instrumentation interface** reference [Att. 6]): this software tool is taken into account for its instruments graphics representations. Today's automotive software lacks intuitive representations of values in visual virtual instruments. These are present in very sophisticated software interfaces only. *Key-ideas: visual gadgets/instruments for representing certain automotive values which have direct resemblance with reality.*
- **"EHOOKS/NoHOOKS"** (**easy-handling** reference): these software tools are taken into account for their apparent ease with which the user may interact with complex systems accomplishing apparently simple tasks such as placing bypasses and working with those bypasses. These tools represent a big leap forward in easy-handling techniques for excessively complex systems that would otherwise require mammoth-like efforts to achieve the goals. *Key-ideas: direct manipulation of programs by eliminating the need to use long, slow and complex tool-chains.*
- **"IEC 61131-3 PLC Standard"** (**multi-mode editing** reference [Att. 6]): the software tools which use all of this standard's program representation options show a multitude of ways to edit those programs. This multiple editing is also very useful for the automotive scene, in that different people with different expertise levels may chose the most adequate representation option. *Key-ideas: graphical iconic language for visual intuitive programming, command-line option for experienced user command and program entry.*
- **"LEGO Mindstorms" & "SCRATCH"** (**graphical interfaces** reference [Att. 6]): their graphical representations of the various programming entities is very interesting, especially how loops and conditional entities are represented visually. The fact of having graphical artefacts such as "parameter dockers" is also very useful to limit users' mistakes. *Key-ideas: representation of loops and conditionals, parameter dockers.*
- **"FishFace" & "PCS Visual Logo"** (**macros representation** reference [Att. 6]): beside their graphical representations, the direct translation of graphical operations into lower-lever and well defined "function macros" is interesting from the point-of-view of developing a similar "graphical compiler" for the automotive scene. Since ASCET also works with well-defined iconic operations, it should be possible to develop such a compiler rapidly and efficiently. *Key-ideas: direct graphical compiler.*
- **"Visual Basic/C#"** (**pseudo-live programming and CIL-code representation** references [Att. 6]): Visual Basic was a major breakthrough and eye-catcher back in the 90s with it's almost "live-programming" features. Allowing the user to make small and quick code-changes and get back on track in a matter of seconds and without having to shut down the complete program (although stopping it is still necessary), is highly interesting for the automotive scene in that it could finally allow automotive software developers to debug, test and validate their programming much faster than before. Since turnaround delays are large in the automotive debugging scene, this feature could be an extremely valuable plus. This "Live-Programming" feature is also going to be the most interesting, valuable and central part of all work in this thesis. As with other tools, CIL-code representation is also a considered internal feature. *Key-ideas: live-programming.*

- **"JAVA" & "JAVA Virtual Machine"** (**byte-code representation** reference [Att. 6]): this programming language, as well as others, has the interesting feature of compiling the source-code into JAVA byte-code which is then interpreted on the target machine. Hardware micro-coded implementations show that it is feasible to have fast execution cores for this byte-code programs. This type of low-level representation will also be used in this thesis' work. *Key-ideas: low-level byte-code program representation, possibility of direct processing of this byte-code in hardware.*
- **"Jazelle DBX"** (**direct byte-code execution** reference [Att. 6]): this direct JAVA byte-code execution in hardware without any more translations is what comes nearest to this thesis' software component work. It represents a radical way of benefiting from intermediate byte-code representations without losing execution speed of the programs it represents. The only big difference to this thesis' work lies in the fact that these byte-code are still compiled from a high-level textual language. *Key-ideas: direct/native processing of byte-code in hardware.*
- **"Transputers"** (**parallel processing** reference [Att. 6]): this concurrently executing hardware paradigm is what comes nearest to this thesis' hardware component work. It represents a radical way of benefiting from multiple relatively simple and autonomous hardware components, to form a larger and high-capacity processing platform. The main difference to this thesis' work lies in the fact that there still exist caches, pipelines and other classical hardware artefacts which make its usage more difficult than strictly necessary. *Key-ideas: parallel processing in independent nodes.*
- **"Sinclair BASIC / Spectrum"** (**implicit handling efficiency** reference [Att. 6]): this combination of software and hardware components, along with other similar examples, turned home-computing upside down in the sense of everyone basically being capable of writing some simple programs without having to attending a whole course first. This is one perfect example of people achieve when they are allowed to concentrate on what really matters when building software: simply putting mental algorithms into source-code virtually without having to bother about software and hardware details. *Key-ideas: implicit simplicity, overall handling efficiency, intuitive start-up.*

Note that most are not directly used in automotive systems but are nevertheless highly interesting (please see [Att. 6] for further details). It should be further pointed out that no known language or system implements some sort of direct language-to-hardware relationship, in the sense of the usage of the straightforward execution of byte-codes. There are three main streams which can be identified and separated as follows:

- High- or low-level syntax-rich languages such as "C", Pascal and Basic, which are parsed and compiled to native machine-code of target processor. This code is natively executed on the target processor. This is a declining process today, although still extremely persistent due to the historically grown habituation factor, user community, tool landscape and market share surrounding those languages. It evolved historically in a bottom-up strategy, where layers of programming abstraction were added on top of the existing ones on the bottom.
- Visual/iconic virtually syntax-less languages such as ASCET, Simulink, LabView, etc., which are converted (auto-coded) into "C", which in turn is then parsed and compiled to native machine-code of the target processor. This code is then natively executed on the target processor. This is an evolutionary process which grew on top of the previous process and is state-of-the-art. It evolved historically in a bottom-up strategy, where the visual interfaces are programming abstraction added on top of the already existing structures. In other words, everything started off from the untouched lower components such as the hardware, linker, assembler, compiler, reaching to top visual editors.
- High-level syntactic languages such as JAVA and C#, which are parsed and compiled into general byte-codes. These byte-codes are non-natively executed on interpreters or directly executed on specific hardware processors. This is a state-of-the-art and growing process today. It evolved historically in a top-down strategy, where the top layer of programming abstraction directly translates into code elements (byte-codes) which are then executed anywhere. In other words, everything started off from the desired top interface, passing through the intermediate representation, reaching down to any hardware to execute on.

It does not seem to exist a language which combines both state-of-the-art worlds into a simple source-to-hardware relationship, including direct-execution. Quite the contrary, in fact, it seems that all over the industry the principle is to resort to highly complex software tools and hardware platforms, which, being difficult to understand themselves, then need long and also difficult to grasp tool-chains connecting those both end-point realities. It would be nice to have ASCET translating its graphical elements directly into byte-codes, which would then also directly be executed in dedicated hardware, for example.

The work presented in the next chapters tries to address this by implementing a language which represents neither a top-down nor a bottom-up approach, as all previous standard or even state-of-the-art cases. This language will have novel a "middle-to-the-sides" approach in which the to,p interface and the dedicated hardware target both use exactly the same underlying representation. While Chapter 2 will thoroughly expose and discuss the problems and pitfalls of past and current systems, Chapter 3 will then present and discuss solutions to these problems.

Page intentionally left blank

"Reality is merely an illusion, albeit a very persistent one."
– Albert Einstein

CHAPTER 3

Problems & Limitations of current Systems

3.1 Introduction

[Emphasis: software and hardware trends]

This chapter will focus on identifying the majority of pitfalls lurking inside existing automotive development systems and the corresponding hardware. By analyzing each one of them, they are identified and categorized. As seen in the previous chapter, there are lots of automotive and non-automotive software tools and hardware platforms to accomplish a great variety of development tasks. Just for the sake of illustration, a categorized range of existing tools and platforms is presented in Fig. 3.1:1 and in Fig. 3.1:2, respectively. This is done according to internal complexity versus usability. There turns out to be a clear distinction between classical tools and those which employ visual or any other enhancement at user-level experience (but which grew on top of the classical tools and still exhibit those classical properties). The aimed target lies on the spot where “direct” and “natural” tools emerge as completely new developments[*]. It would be desired that better user-interfaces (nearer to the problem reality space) lie in proportionality in respect to a simpler internal tool structure. What really exists instead is the "invisible effort" that must be undergone by the developers of such tools, to make "reality-close" user-interfaces at the expense of big internal efforts. This “reality-closeness” is very important to produce a good user-interface and encompasses things such as:

- **Intuitiveness** – tries to measure how fast an inexperienced user is capable of finding the tool features he needs to accomplish tasks, without having to spend too much time.
- **Easy-Handling** – tries to measure how effortless a user is capable of accomplishing his development and debugging tasks, without losing time with tool details and general side-effects which have nothing directly to do with the tasks at hand, such as setup, preparation and turnaround delays.
- **Reality Representativity** – tries to measure how much the tool’s user-interface and its handling strategies are close to the real tasks at hand, in respect to things such as biunivocal (“1-to-1”) correspondence, abstraction levels, tangible handling, etc. The discrepancy between reality and program representations strongly relates to the "Complexity Gap" [370], a very important concept explained later on.

[*] By "direct" and "natural' it is meant the opposite to the development methodologies which still employ tools whose working and programming mechanisms, that no longer represent /mimic the real "need vs. solution" dichotomy. In other words, as assertively exposed in [12] for the case of computing machines, today's tools continue to not directly and naturally represent/mimic the problems at hand and that they intend to solve programmatically. Nevertheless, there are already some tools in other areas that try to employ as much directness and naturalism as the respective problem situation allow them to. The more a tool mimics the real situation with its real complexity, the easier to develop and use, it turns out to be at the end, being "3D virtual reality" simulator an extreme of this paradigm.

Much of this last bullet directly relates to the previous two in that intuitiveness and easy-handling may strongly derive from how close the necessary user-actions are from the real tasks at hand. On the other hand, It appears to be clear that improvement on the visible side, high expenses on the hidden side must be undertaken. Thus, although user-interfaces have evolved greatly, most of today's tools cannot be considered "direct" or "natural" in what the complete package is concerned. An immediate illustration of this kind of natural evolution is the Microsoft Visual Studio’s ability of allowing the user to make small code-changes while in the middle of a deep/lengthy debugging session (breakpoint or code-stepping active) and applying/running those changes without having to restart the whole solution or having to wait long re-compile delays, while not even losing any of the currently produced (and often hard-to-reproduce) variables’ values. Everything that leads to delays and the need to take actions unrelated to the reality at hand herein represent overheads related to internal non-direct structures. Again, this strongly relates to the "Complexity Gap" [370] between the highest SW editing levels and the lowest HW execution levels.

Regarding hardware platforms, the most important features to consider are the “reality-adaptiveness” again versus the complexity level to materialize and achieve the high-level user-interface goals. This reality compliance encompasses concepts such as the following, while the work presented in this thesis will address these issues as well:

- **Easy-Interfacing** – tries to measure how fast and effortless a user can connect the system to necessary sensors and actuators, without having to use any non off-the-shelf solutions.
- **Sensing/Actuation Closeness**– tries to measure how close the algorithm-processing hardware core is to sensors and actuators that feed and are fed by it, respectively, not only regarding physical distance but also the location of the necessary analog/power electronics.
- **Structural Flexibility** – tries to measure the capability of the platform adjusting itself to larger changes in reality (e.g. using an ECU on a larger engine or adding sensors/actuators to it), through composing factors such as hardware scalability, processing scalability and interchangeability.

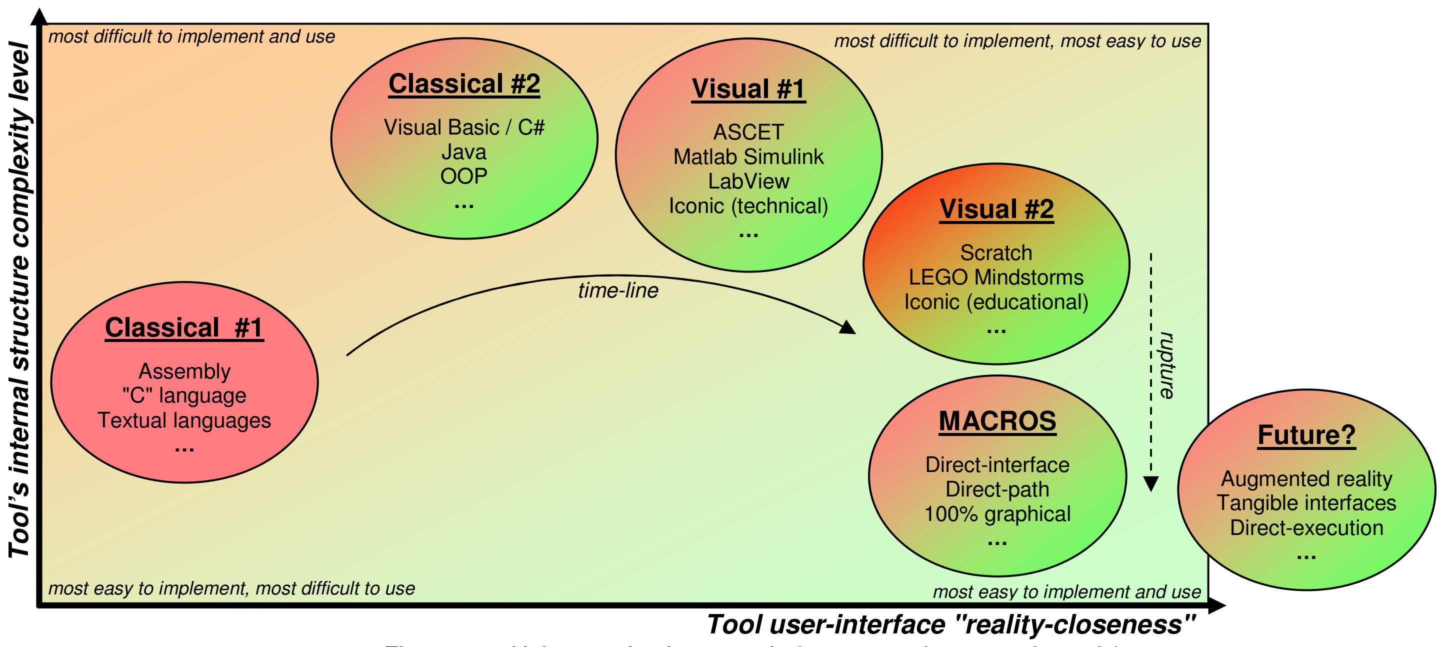
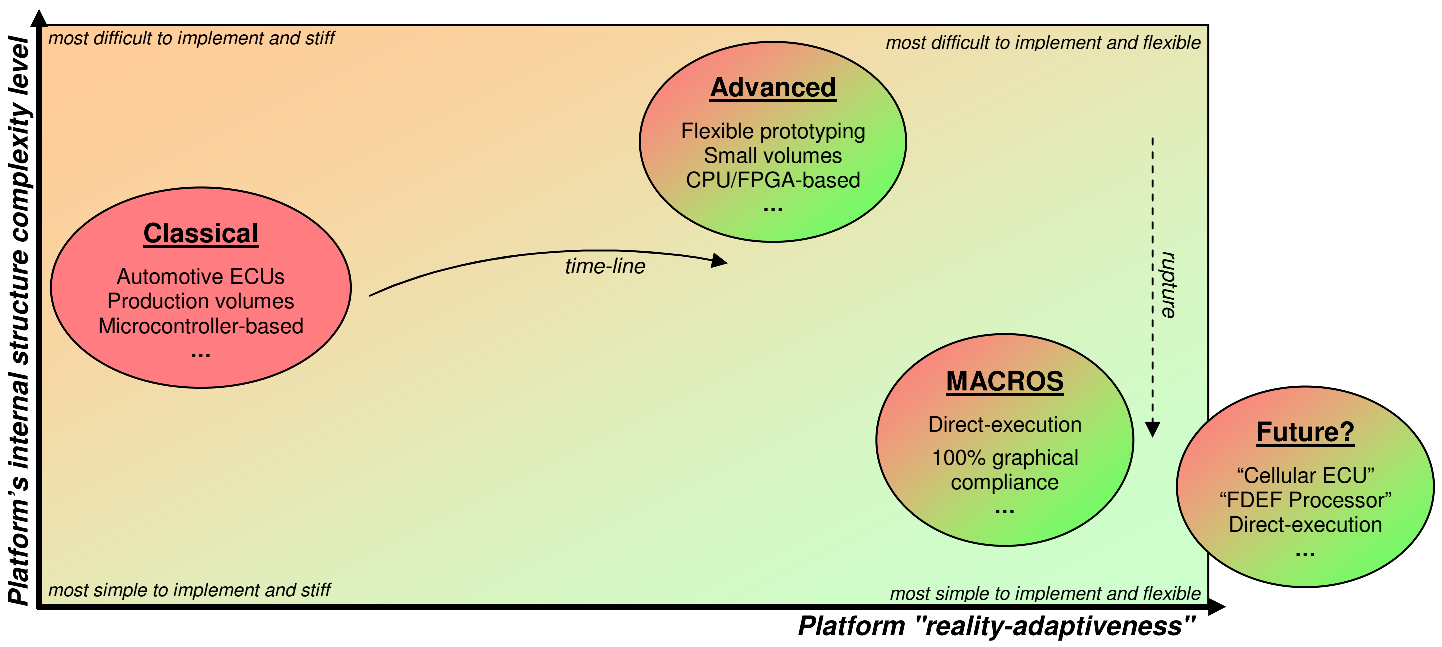


Fig. 3.1:1 – Universe of software tools (representative examples only)



Platform's internal structure complexity level
most difficult to implement and stiff
most difficult to implement and flexible
most simple to implement and stiff
most simple to implement and flexible
Classical
Automotive ECUs
Production volumes
Microcontroller-based
...
time-line
Advanced
Flexible prototyping
Small volumes
CPU/FPGA-based
...
rupture
MACROS
Direct-execution
100% graphical compliance
...
Future?
"Cellular ECU"
"FDEF Processor"
Direct-execution
...
Platform "reality-adaptiveness"

Fig. 3.1:2 – Universe of hardware tools (representative examples only)

It is clear that the automotive industry mostly uses software tool categories "Classical #1" and "Visual #1", plus hardware platform categories "Classical #1" and "Advanced #1". It is also clear that research and commercial solutions indeed try and aim to even better mimic the problem situations at hand and to develop hardware capable of truly expanding the software's possibilities. Furthermore, the "time-line" in both figures represents the trends and progress in both universes, where an "evolutionary" component is clearly stated at the left sides. It can also be recognized that after this evolution/increase in complexity to achieve more reality-closeness and handling, the *"Macros"* paradigm later on in Fig. 4.2:10 and Fig. 4.2:13 tries to drastically reduce complexity while even increasing reality-closeness and handling-ease. This is represented by the dashed "rupture" vertical downward line, where a sudden downward direction was taken concerning internal complexity of the global automotive system.

3.2 Some Automotive ECU Considerations

[Emphasis: software and hardware inappropriateness]

The global and local “inappropriateness” of the system details that this chapter is about to address requires an historical contextualization and a brief discussion on some Automotive ECU considerations to bring into the correct framework the discussion about the actual current challenges and limitations faced by the automotive scenario. This historical contextualization can be found in [Att. 59], providing an overview of some global and common general pitfalls as well as a contextualization of the natural and even understandable industry lethargy. It also discusses general software and hardware issues.

3.2.1 Specific Automotive Functionality

[Emphasis: needed functionality, ways of doing it]

A typical automotive functionality used in both series-car and motorsport areas, also called FDEF (Function DEFinition), is a sequence of arithmetic, logic, conditional, parameters/memory and other operations, which receive inputs from sensors and/or other FDEFs, processes them and outputs one or more values to be used in other FDEFs and or actuators, as illustrated in Fig. 3.2:3. Many of these FDEFs then form the total automotive functionality that essentially manage the engine and the chassis.

Since these FDEFs depend on inputs from back-end sensors, outputs to front-end actuators and on each other's intermediate FDEF outputs as well, a certain hierarchy emerged from their inter-connections, as illustrated in a very basic example in Fig. 3.2:4. Much more complex hierarchies are usually the case for series-car and motorsports ECUs (Fig. 3.2:5), where inter-connections are much more heterogeneous, confusing and numerous. Communications, datalogging and other functionalities usually reside directly in code and do not have visible FDEFs.

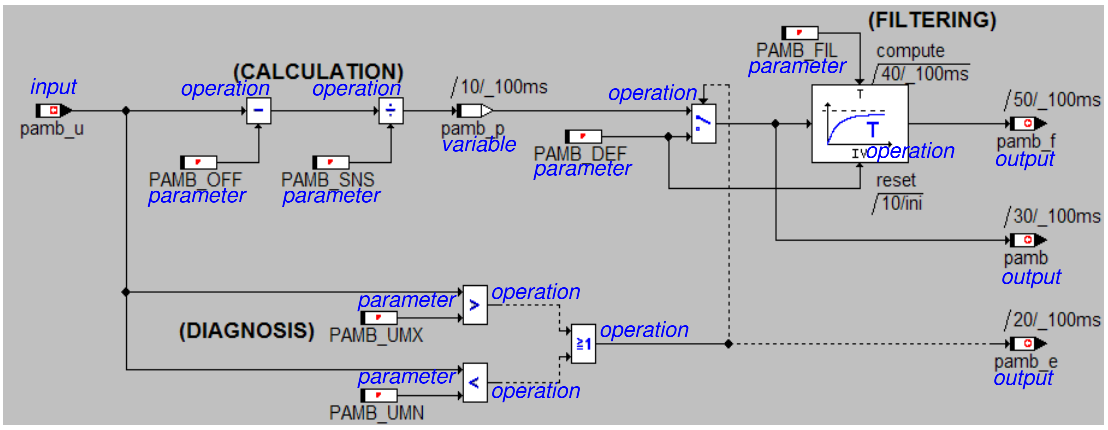


Fig. 3.2:3 – Typical automotive functionality with inputs, outputs, parameters, variables and processing operations

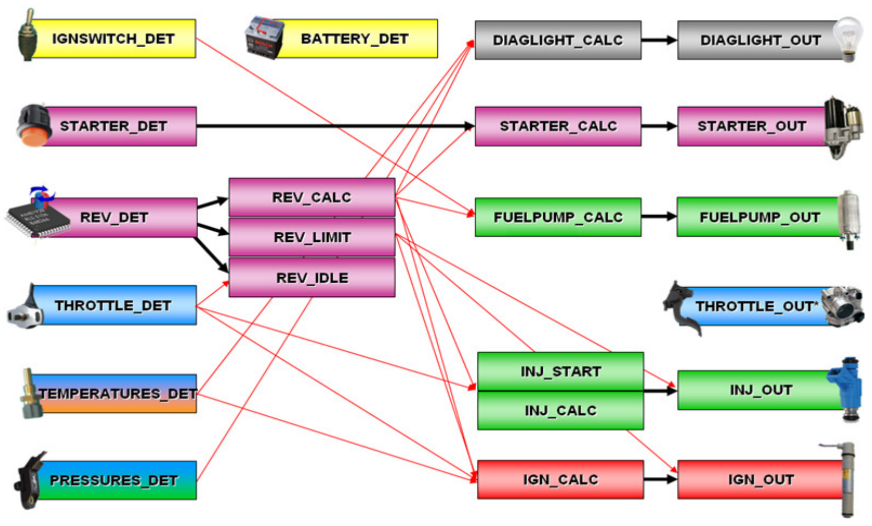


Fig. 3.2:4 – Very basic automotive functionality hierarchy with only the very essential FDEFs to manage an engine

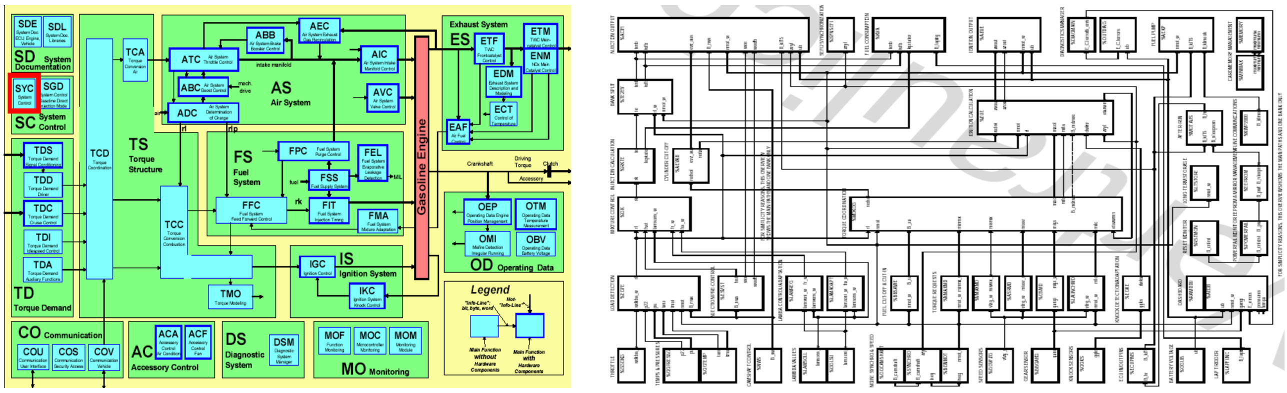


Fig. 3.2:5 – Much more complex automotive functionality hierarchies; Left: series ECU; Right: motorsport ECU

The most basic functionality paths that are absolutely necessary for convenient engine management, are some temperatures, some pressures, injection and ignition. Functionalities such as knock-control, lambda-control, high-pressure fuel-control and others, are needed when a more refined engine management is needed in modern systems. Some examples are shown below in Fig. 3.2:6. A comprehensive but far from exhaustive list of automotive functionalities can be found in [Att. 8].

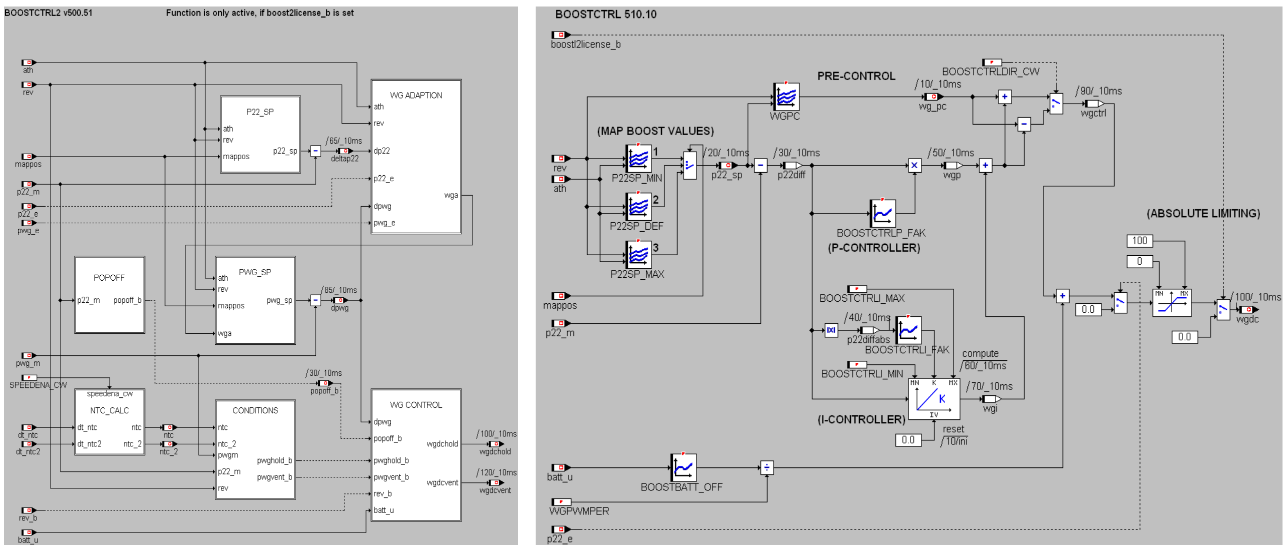


Fig. 3.2:6 – Turbo booster control hierarchy and one of its inner FDEFs (source: [477])

Since automotive functionality range is ever-increasing FDEF count, FDEF complexity, FDEF count, FDEF complexity, inter-FDEF connections, etc., can reach almost out-of-control levels. Here is a basic list of the problems and pitfalls that arise within this realm for developers and even users (scenes most affected by this are marked with "S" for series-cars and "M" for motorsport vehicles:

- **Functionality Count (S)** ⇒ currently, having ca. 2000-3000 FDEFs inside a single ECU is not uncommon, rending impossible to have an overview of an ECU's global functionality hierarchy. This increases the probability of having bugs due to lack of global overview of the system.
- **Mistakable functionality (S)** ⇒ generally, the way most functions operate can get very confusing, since these can be quite large (30+ pages for lambda-detection FDEFs plus 200+ pages for description texts) and have grown in a "band-aid" cumulative way. It is rare for a functionality to be rewritten, most being adapted along with the demands.
- **Functionality Interdependencies (S/M)** ⇒ similarly to above FDEF counts, the inter-connections among those FDEFs also get totally exploded in current systems. This also contributes to bugs related to totally blatant lack of global overview of those inter-connections and their effects upon the FDEFs.
- **Functionality State-Space (S/M)** ⇒ similarly to the above functionality interdependencies, the thereout generated magnitude of the corresponding state-space can get unbearable proportions. The global state-space sometimes reaches hundreds of states and thousands of transitions among those states. It is obviously not possible to have a correct overview of such things and systemic hard-to-find bugs appear naturally. Nevertheless, this diversity is sometimes necessary, as depicted in Fig. 3.2:7.
- **Messy Hierarchy and Heterogeneous Abstractions (S)** ⇒ an ECU software is mostly composed out of many FDEFs belonging to different automotive functionality areas, where not all interfaces between FDEFs and the corresponding hardware or other low-level FDEFs are fully homogeneous. The hierarchical relations between FDEFs and their related abstraction levels are many times confusing and cause easy misunderstandings.
- **Software Integration (S/M)** ⇒ it gets extremely difficult to know how to integrate new functionalities and to do that without disrupting any other existing functionality. It can also be very difficult to know where to make the exact needed cuts and derivations for the new functionality's input value needs and for the new output feeds. It is very easy to implement bugs at this point.
- **Calibration and Testing (S)** ⇒ it gets a highly time/money-consuming task, finishing the complete product in matters of calibrating all the parameters to the desired project application and to test the entire system "on the road". 500.000€ for motorcycle projects/applications and multi-tens-of-million Euros for street-car applications are commonplace.
- **Increasing engine concepts and regulations (S)** ⇒ it gets almost impossible to integrate different engine concepts into the same package. In fact, different engine and regulatory details are most often handled by different teams and the corresponding different ECUs, further increasing the already huge multitude of functionality and lack of system overview.
- **Multitude of SW/HW Components (M)** ⇒ it gets increasingly harder for small engineering teams to hold a grasp over the various different components they have to keep working together. Each component generally comes from a different direction and uses different SW and HW tools and platforms (Fig. 3.2:8). Knowledge/understanding synchronization needs, communications/interfaces havoc, configuration difficultes and wastes of time have led to the development of one common SW/HW platform to derive the different components (ECUs, dataloggers, displays, etc.). The series-car scene beats this with sheer manpower.

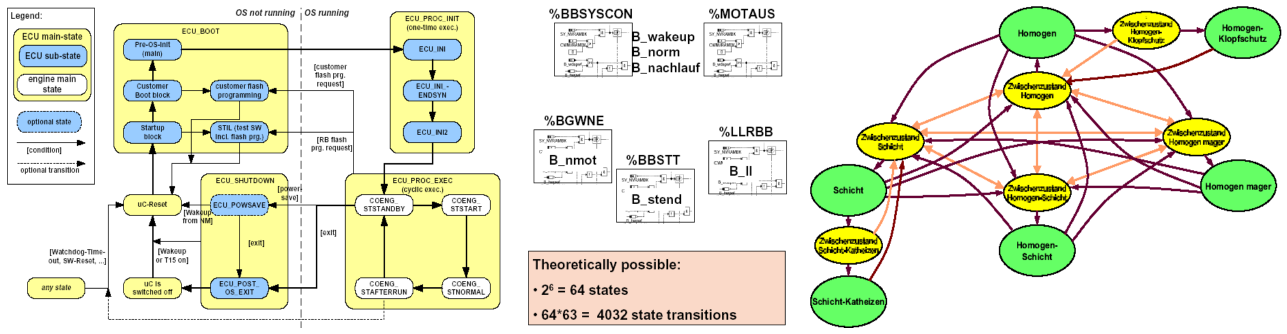


Fig. 3.2:7 – Various state-space generating functionality diversity

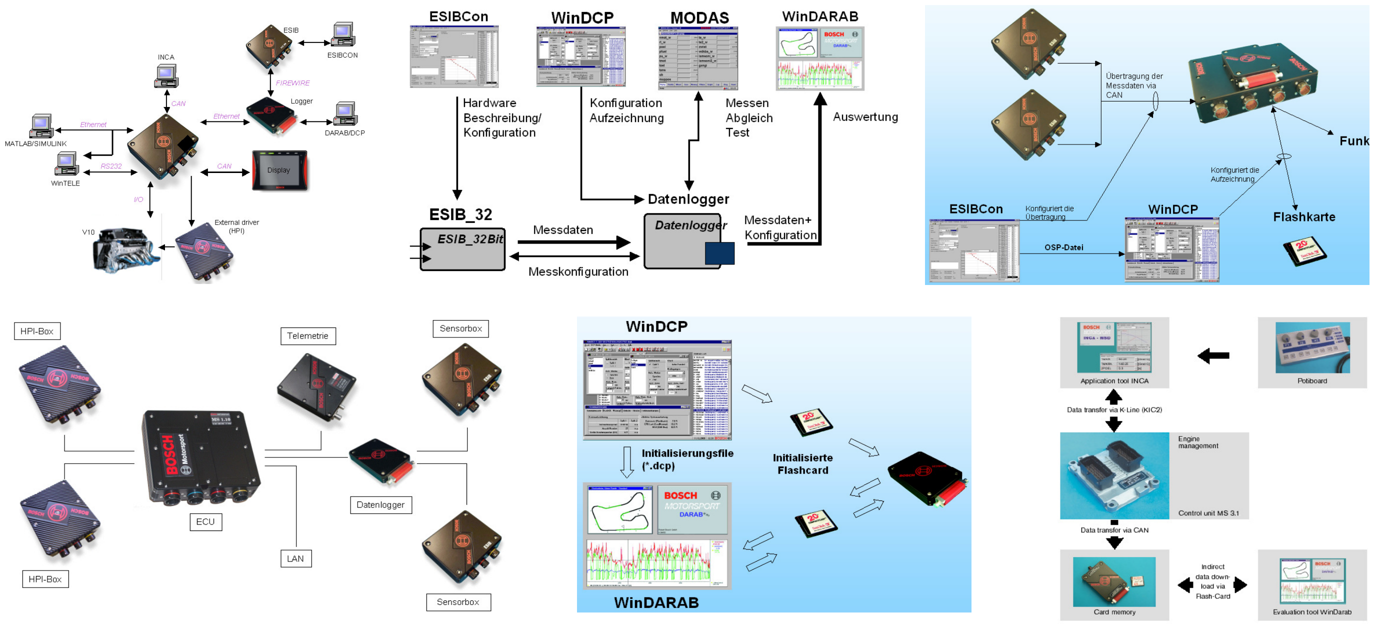


Fig. 3.2:8 – Bosch Motorsport legacy and only partially still ongoing components' and tools' over-diversity

It is very interesting to note in the general way of executing automotive functionalities and in how this historically developed into an almost pure "data-flow" processing, is that this intrinsically makes automotive functionalities good candidates to be processed in parallel. Noting that most FDEFs take their inputs, process them apparently unaware of all others and output their values, leads to the thought that each FDEF could even run on a different processor. This is true, aside from the fact that some functionality still needs some form of event-driven actions, such as for crankshaft engine-speed signal, injection, ignition, knocking and any other that needs asynchronous attention. Getting rid of these critical lower-level functionalities or, at least, displacing them to elsewhere, frees all the higher-level ones for natural parallel processing. This matter will be closely followed and treated later on, because this will be one of the main reasons for giving birth to the *"Smart Peripherals"* and to "Parallel Processing" within this work herein.

Just to finish off this section, here are some of the technical details that must be considered when developing a new automotive FDEF or changing an existing one. These technical details concern the underlying used programming language, numbering scheme details and hardware details. Missing/messing with one of these intricate details, can render an entire system completely useless ("user" is herein defined as the programmer using the programming tools):

- **Programming language syntax** → great care must be taken when programming with the ubiquitous "C" programming language and even more if applying Assembly, because of the diversity of user-related errors that might appear along and disrupt the entire system altogether.
- **Numbering scheme and quantization** → Older ECUs user integer values with different quantizations, to cope with many value ranges. Implementing these can lead to unexpected results, leading to systems misbehaving. An associated problem is that filters and integrators need internal "state" variables with double the output variable precision to avoid remaining stuck below a certain non-representable "state" value. Newer systems use more and more floating-point values, which have another set of problems associated with them, such as the fact of numbers not being 100% correctly represented, due to internal bit quantization limits. Casting operations and underflow/overflow mishaps are another source of user problems.
- **Data endianness and order in communications** → Data word, byte and even bit order are of paramount importance, especially when dealing with communications. User-errors at this level are easy to spot but still the cause for many problems when testing such systems as a whole.
- **Word-alignment in memory operations** → Many micro-controllers go havoc if a word is not correctly aligned to an even word-address. User-errors at this level most often lead to crashes.
- **Interrupts, stacks, registers** → Implementing or changing interruption code is about the most critical task for a developer, since totally unpredictable results may start to appear, as soon as any of the smallest stack, registers or other detail is misused or tampered with. Since interrupts can be issued practically at any time point of global system operation, user-errors herein often turn into complete system failure, occasional crashes, havoc behaviour without any apparent reason and can even signify weeks of bug-searching when intricate timings are involved.
- **Temporary storage for certain operations** → Certain code blocks need temporary values' storage, especially in the case of filters, delays, integrators and other blocks needing values other than the input and output ones. This temporary storage is normally easy to implement, but there are cases where the location and nature of that storage may pose intricate problems.
- **Operations' priorities & sequencing** → Since normally there are multiple time-rasters in an automotive system, their priorities influence the functionality operations' priorities, demanding a careful design from the developer. Similarly, operations' sequencing makes a huge difference in highly inter-locked operations and even in functionalities. User-errors at this level may lead to unexpected results, especially in very closed control-flow type algorithms.
- **Timers and time-rasters** → The usual availability of many timers and time-rasters poses the problem of the most correct choice for the various functionalities' code-blocks, filters, delays, timings, etc. User-errors at this level usually lead to badly behaving control algorithms, wrong time-constants on filters and delays, non-working communications, etc.
- **Differences inspection among versions** → It is very important and useful comparing an older version with a new one, especially after the desired changes have been fully made. This helps a lot in finding user-errors if misbehaviour appears during tests or on the field. The problem is that "C" and Assembly source-code comparison are often not as easy as imagined, due to the verbose and comments extension that source-code may reach in some more complex FDEFs.

These are some of the problems when handling automotive FDEFs, herein called "pitfalls" and listed/detailed later on. From experience, it can be said that at least 90% of the FDEF development time is lost during implementing the FDEF into the conjunct software/hardware structure. This is why these pitfalls will be identified and thoroughly discussed. The work within this thesis will keep a focus over these pitfall details, by trying to eliminate them altogether, to create a software/hardware system that will allow the developer to finally focus on his main job: FDEF development. Implementation will be greatly improved, therefore accelerating FDEF development altogether.

As a final remark, in plain ECU Assembly or "C" programming, the code is written in one place and the data is normally declared somewhere else, thus leading overhead search procedures among the source files. In state-of-the-art visual automotive functionality development systems however, this has been greatly improved by aggregating operations with their respective parameters on the same spot or at least around the same spot. Since in most cases each parameters is used by only one operation and operations are generally independent from each other, this allows for a nice encapsulation, separation and even parallelizable natures, which are going to be further exploited in the work herein. This can again be seen in [349] above. The result is that only the input variables need to be stored "outside" this "pure-ROM" processing flow.

3.2.2 Final remarks & Handling-Efficiency

[Emphasis: keep the hospital operating room neat and clean]

It is very important to note that automotive ECUs currently are very complex and advanced systems that can and must take 100% care of themselves, as long as all sub-systems are working correctly. Just like an animal's organism, they should need only little external intrusion to be able to communicate, react upon sensors, deliver data to actuators and respond to commands to and from the exterior. Unless a heart bypass is needed for the surgeon to be able to perform a more complex surgery on an organism, the least external add-ons needed, the least can go wrong during the procedure. Actually, instead of the still used external add-ons such as [134] [135] [221] and many others of the same kind, more and more internal architectural hardware components are being used for an ECU to be able to instantly perform operations that would otherwise need extensive external gadgetry. Intrusively debugging software embedded inside the ECU's firmware further worsens this already overcrowded picture, turning "handling efficiency" lower as well.

Frequently, most critical debugging operations have to be done externally, while the hardware just lies there, totally obedient and, in virtually all cases, in a "stop-run-stop" condition. The JTAG protocol [222] appeared and virtually solved the intrusive debugging software issue, by adding that debugging add-on in form of extra hardware around the already existing structures. Although not originally intended for complex CPUs, it is now a relatively ubiquitous availability in current modern hardware platforms. Good implementations of this protocol make add-ons such as [134] [135] [221] obsolete. Nevertheless, it still only allows "stop-run-stop" style debugging and CPU operation, with some exceptions. This kind of merging debugging structures with the base-hardware is evolving and the hardware platform developed herein will also closely follow that evolution, with some major advancements in terms of new "online debugging" features.

It is never enough to note that current development and usage tools got to such an advanced and complex level, that the vast majority of automotive users really do not care about any of its details anymore. They just want the system to work as expected and tolerance toward arising problems is lower than ever before. The reason is simple: while in the past almost anyone could still understand the entire system and even correct its problems, current systems have grown almost uncontrollably in all directions. Users and even developers are having daily difficulties keeping up with the state-of-the-art of such systems, no to mention problem-solving that knowingly means lots of lost time and nerves. In this work herein, it will be tried to devise an overall system where all this slack is kept as small as possible. The developers and users generally know what they want to do, it is mostly the systems that are either not sufficiently well-adapted to the application at hand, or are too complex, ambiguous and overcrowded in the first place.

Some people such as in [507] [508] have been presenting some metrics such as "usability" and "task efficiency". These metrics are not going to be discussed in detail here, but to at least in which direction to point the final work presented herein, just some simple assumptions are going to be taken in consideration throughout its development. "Handling Efficiency" shall be the top-notion defining the overall empiric amount of the final system's QUALITY. [508] defines this global system quality as:

Quality of use: *the extent to which a product satisfies stated and implied needs when used under stated conditions.*

Since this definition is far too vague and would require too much technical depth, here we are going to take some of the formulas and ideas of the related work and simply define something similar to that quality of use, the development system's global/unified "Handling Efficiency". This form should intrinsically includes all 3 main handling metrics discussed earlier and deeply studied in [507]: accomplishment <u>effectiveness</u>, task <u>efficiency</u> and user <u>satisfaction</u> into one neat, although simplified, formula. Since effectiveness eventually/naturally reaches 100% [508] in any minimal useful development system, given enough time, only the task efficiency and user satisfaction factors thus remain. Another simplification could be done in that [507] states the three previous handling metrics are essentially orthogonal, i.e. they are more or less independent from each other. Furthermore, user satisfaction seems to be something too ambiguous and should therefore not influence so much a global development system handling efficiency metric, also because it is the user's job to get its planned work done either way. Thus, maybe another possibly related and even less ambiguous metric could be used instead: the user's needed know-how quantity to get a particular job on a particular development system successfully done. This metric partially eliminates the user satisfaction ambiguity, therefore avoiding "outsiders" like incompetent users getting angry at a development system they do not understand in the first place. The final result would then be:

Handling-Efficiency *∝ sub-tasks-to-be-done / (time-needed x system-know-how-needed)*

Or, in a more compact and normalized (relative to task completion) form:

$$HE \propto \frac{1}{(T \cdot devsysknowhow)}$$

The evident aim of the new "iEditor" thus is getting the job done in less time with less development system related know-how. Of course, the user must know the controlled automotive plant well. So, in the end, the new *"iEditor"* should exhibit fast command access, short turnaround delays, and an overall intuitive way of operating to minimize the above denominator of "HE".

3.3 Current Challenges & Limitations

[Emphasis: software and hardware detailed pitfalls]

Modern systems possess hidden, often ultra-complex and voluminous layer-stacks that operate behind the scenes to achieve reality-adaptiveness with rather non-adapted hardware. Due to these hidden, totally visible or even mandatory programmers' knowledge details, automotive systems have grown hugely in overall complexity and size, with pitfalls lurking in virtually every corner.

Programmers and systems engineers are fed up with all kinds of theories and all those different ways of doing the same thing, when all they really want is for the system to behave exactly as designed. In sum, computing systems should do exactly what they were designed and programmed to do [12]. Sadly, nowadays programmers are actually more concerned in keeping things working, even with problems, than using new solutions or changing old solutions. Everything turns out to be everything else other than "WYSIWYG[*]", while basic programming environment understanding is left for gurus.

To better group the potential pitfalls that current systems possess, these are going to be separated into two main categories: low- and high-level pitfalls. The first is related to language constructs/syntax and hardware architecture/core details. The second is related to the abstract and undesired handling effects that result from the system operating as a whole. The herein enumerated challenges and limitations are divided into three categories according to their intrinsic nature:

- **COMPLEXITY (C)** – this type of pitfall refers to strictly necessary mechanisms for the system to undertake its tasks, but that are very difficult to grasp and maintain. Although one might think that complexity can be hidden behind easy-handling interfaces, it may create usability problems when least expected, being therefore undesirable in principle. However, classical systems would not work at all without these mechanisms, thus not allowing their simple elimination.
- **OVERHEAD (O)** – this type of pitfall refers to not strictly necessary and not strictly task-related mechanisms that have to be coped with, doing not much more than keeping certain issues under control (performance, bugs, etc.). Although one might think that overhead can be hidden behind clever automatisms, it may create interfering problems when least expected, being therefore undesirable in principle. Classical systems would be able to work without these mechanisms, although eventually not fast enough, but could allow their simple elimination.
- **HANDICAP (H)** – this type of pitfall is the worse of all three and refers to real limitations posed to a system as a whole, in which this system has no way at all of performing some specifically needed tasks. Although one might think that handicaps can be bridged through costly 3rd-party tools, they may create turnaround and integration problems when least expected, being therefore undesirable in principle. The biggest problem that arises sooner or later, is that these agglomerates are difficult to evolve and change, turning future versions and their enhancements into ever-growing agglomerates of tools, layers and plug-ins.

This grouping is going to be used in pitfall listings for a later recall in Chapters 4/5 where the correspondingly proposed solutions are going to be presented. This is especially useful when discussing "Live-Prototyping", which represents the most overheadless and handicapless way of doing things in a "natural" way without any limitation. This is also the most important feature of the system developed herein, by deriving it from all other system-deep drastic complexity-reductions. Pitfalls are represented in ***PITFALL X#N*** format, where "X" is the trait "C", "O" or "H" and "N" is the count. Only the most evident pitfall trait is shown. Solved pitfalls are going to be referred at the end of the relevant descriptions

[*] *What You See is What You Get*

and points. Those pitfalls completely solved will be referred in a green box, while those only partially solved will be referred to in a yellow box. Those not solved at all or only barely touched will be referred to in a red box. Finally, those pitfalls that are not completely solved in a single text location, but in various texts, will have an additional superscript indicating the current index and total count of partial solutions.

Before going into the whole list of pitfalls, it must be pointed out that those pitfalls are centred on the automotive software and hardware scenario. This thesis' work does not aim to solve any similar problems on the desktop PC and/or Windows OS or the like scenarios.

3.3.1 Low-Level Pitfalls (mostly challenges)

[Emphasis: software and hardware architecture related pitfalls]

Many software features and hardware enhancements have been introduced for the sake of getting flexibility and usability into programming languages, as well as of getting more performance out of already limited and “over-developed” computing devices. Since these features historically grew up to overcome systems' known limitations, they are herein considered as being "challenges", as they require very high care when dealing with them. The problem of many pitfalls listed below, is that they cause data storage that obviously represents context during the execution scope. This context disrupts the herein defended ideal of having “context-less execution” [*]. Many other pitfalls ahead also represent context storage, retrieval and management. All these features are generally undesired and ought to be completely eliminated wherever such thing is at all possible:

- **Arithmetic/Logic Multi-Formats (SW)** – Sets of different arithmetic variable types and sizes is counterparted by the corresponding complexity of the compiler. Processors which possess rich register and access mode sets originate compilers with up to one third of its code dedicated to this “programming flexibility” [12]. Ideally, there should be no need for more than a unique variable type/size for all purposes and it should be intrinsically declared/defined. Additionally, the format should naturally tolerate overflows, "divisions-by-zero" and array "out-of-bounds" in a benign way, producing still "logical" and usable values without the need of extra exception-treatment code. ***PITFALL C#1***
- **Endianness (SW) –** Processors use different storage orders for the bytes in memory [230]. There are big-endian and little-endian processors. Compilers have to take this into account when manipulating variables. This obviously introduces more entropy into compiler-making. Ideally, there should be no need to be careful on this processing detail. ***PITFALL C#2***
- **Word-Alignment (SW) –** A problem that has strictly to do with data positioning is the alignment and subsequent access of data by the CPU. Normally, 16-bit CPUs require bytes and words to be "word-aligned", that is, to be aligned with the beginning of a whole word in memory. In other words, data must reside in even addresses or else the CPU will issue an error or simply misbehave. Hand-manipulations that do not take this into account often result in misbehaving programs. Ideally, there should be no need to be careful on this. ***PITFALL C#3***
- **Lexics & Syntax (SW)** – All text-based languages feature lexical elements and a global syntax, which makes lexical analysers (scanners) [183] and syntax/semantic analysers (parsers) [184] a must. The big multitude of possible combination of lexical elements and the many possible error situations due to syntactic/semantic mishandlings, represent one of the biggest sources of human-errors and compiler-bugs. Ideally, the language should not need any explicit lexical elements or syntactical/semantic structures, as well as no special add-on for highlighting and error-marking. ***PITFALL C#4***

[*] *This paradigm where no context or any other form of “execution-status” retention exists, other than the central inputs/outputs' values holding memory and the program-counter itself, plays a central role throughout this thesis' work, with the aim of achieving “intrusion-less” execution.*

- **Operation Priorities & Sequencing (SW)** – Both arithmetic and logic operation have their pre-defined priorities when it comes to knowing which one should be calculated first and next. These priorities and operator precedence is pseudo-standardized [363] and causes all kinds of misconceptions and errors [362]. Ideally, the language should not need any explicit priotization or sequencing lexical elements and should intrinsically/intuitively show the underlying priorities/sequencings. ***PITFALL C#5***
- **Semaphores & Locks (SW) –** Explicit software semaphores [212] also called locks [548] are known to produce some annoying side-effects when not 100% correctly programmed, such as starvation following dead-locks, resource trashing following lock mishandling, etc. Ideally, there should be no need to have any software semaphores, eliminating the need to be so careful with them. ***PITFALL C#6***
- **Time-driven vs. Event-driven (SW) –** Virtually all embedded systems fall into a merge of these processing categories. Problems arise when the event-driven code disrupts the deterministic operation of the time-driven one. Ideally, there should be no need to have event-driven code and all of it should be time-driven, or even better yet: purely cyclic with no deterministic time-frame at all. ***PITFALL C#7***
- **Coding Reliability (SW)** – When coding, for example in the "C" language, there are so many different potential human coding errors that might be perpetrated at lexical/syntactical level, that it gave rise to the MISRA organization [228] [189]. This includes the human ability to subvert some language constructs to the limit of obfuscation. There are now full-fledged software packages from companies that only dedicate themselves to this kind of code testing [389], but even so code correctness can never be guaranteed. Ideally, the language should be constructed in a way that would leave no margin of misinterpretation whatsoever. ***PITFALL O#8***
- **Bug Searching (SW)** – This procedure is usually related to some undesired system behaviour caused by some low-level bug (any high-level design bugs are excluded here). Since current development tools are highly complex, whenever such a bug search initiates, it may potentially be an arduous and highly time-consuming task. Furthermore, the underlying layers may not even allow for some advanced and highly useful/desired features. Ideally, there should be no need for this kind of low-level bug searching. ***PITFALL C#9***
- **Code Coverage (SW)** – Special software testing tools allow creating conditions and variable values to see the response of software algorithms and their execution paths [235]. When 100% coverage is not possible or practical [2], specialized test-beds are prepared with value ranges that target the most important execution paths, making no guarantees ad the end whatsoever. Code coverage testing can really generate large amounts of data, even in relatively small projects [295]. This pitfall can never be 100% eliminated because functional/logical/algorithmic bugs are always possible with any language. Ideally, the user should only be preoccupied with higher-level bugs such as algorithmic and though-flow errors. This later ones should then still be tested with dedicated testing software, but in a much more integrated and automated way. ***PITFALL O#10***
- **Translators & their Outcomes (SW)** – These software packages are used to convert one language into another and are materialized by pseudo-coders, auto-code generators, compilers, assemblers and, in some way, also the interpreters. Because of the enormous multitude of existing languages, variants of those languages, as well as intermediate representations, there exist literally hundreds of translator packages and combination possibilities. It seems evident that the drawbacks of having translators are those related to potential extra error-production inside those translators, besides the source-code bugs themselves, as well as the overheading need for certification [186] [27]. Ideally, there should be no need for any translators, turning the overall system much leaner, overviewable, maintainable and much easier to deliver truly easy debugging and prototyping functionality. ***PITFALL C#11***
- **Code Optimization/Compression (SW) –** These procedures either aim to reduce code size, to increase code speed or both. Both cases lead to apparently cryptic code in the lower layers of a tool-chain, rendering debugging and profiling efforts almost useless, where the tools themselves will fail altogether. Moreover, optimization/compression conditions may trigger the compiler in a somewhat unpredictable way, causing unpredictable results in some critical timing spots and such. This is similar to changing one spot of the code and having another suddenly malfunctioning due to some critical relationship. Ideally, there should be no need for code optimization or compression. ***PITFALL O#12***

- **Source vs. Output Diversity (SW)** – The most "expectable" thing in past and current systems is that source-code versus output machine-code correspondence never was and eventually never will be biunivocal. It is rather a "many-to-many" relation ("N:M" relationship, in relational database jargon), due to the internal stack of layers, each layer making its own translation and code mixing. Tool certification [282] [186] [27] and tool requirements obedience [267] do not resolve these problems at all. Another problem posed by this reality is that repeatability is not given: exactly the same source-code compiled and deployed later onto a ECU may not give exactly the same results as previously. This poses hard limits when reviving legacy projects or simply restoring some older projects to life. Ideally, there should be some way of guaranteeing that a source-code would always yield the exact same machine-code output. ***PITFALL C#13***
- **Code Tweaking (SW)** – This procedure is related to fine-tuning or trimming some code section to improve its original performance, usually done by hand. Although auto-code generators have evolved a lot, the need for fine-tuning is still an issue on lowest-level code and drivers that are mostly still hand-programmed. It is also well-known that human expertise and knowledge of the whole problem gives hand-coding the necessary advantage to achieve certain almost surprising things where artificial coding fails [256]. Ideally, there should be no need for any tweaking. ***PITFALL O#14***
- **Options & Switches (SW)** – Most tools and tool-chains have too many options, switches, and handling alternatives. This results from current tools having evolved through layer stacking of the various translators and auxiliary tools. Also, compilers and linkers normally present large amounts of possibilities such as optimization, addressing, syntax and other variants, thus producing the need for the developer to setup all the associated options and switches. Usually, these options and switches appear in the so-called "make-files", which contain a script-like commands-list to orient the complete tool-chain through building a program. Ideally, there should be no options and switches at all, while the user-interface should be kept as lean and direct as possible. ***PITFALL C#15***
- **Memory Wait-States (SW)** – This mechanism is used because in the recent past the memory speeds did not catch up with CPU speeds [231]. This causes programs to run faster or slower [361], depending on the need to fetch data from internal or external memory, respectively, thereby causing timing jitter. Ideally, there should be no need for wait-states of any kind at all. ***PITFALL C#16***
- **Caches (HW)** – This intermediate storage for frequently used code and data, leads to “cache incoherence” problems [171] when attempting to parallelize program execution. Special complex mechanisms are needed to warn of outdated cache segments. Ideally, there should be no need for caches. ***PITFALL O#17***
- **Memory Access Hashing (SW)** – Physical memory access hashing is a mechanism found to be greatly useful and even necessary within shared memory parallel processing systems, which tries to minimize memory module access contention due to algorithmic data locality. The memory architecture should be such that it would totally eliminate the need for such overheading mechanisms. ***PITFALL O#18***
- **Stacks (HW)** – This intermediate storage is usually to hold function call arguments, return addresses from sub-routines being called and also interrupt context preservation. Stack-based languages as JAVA even use it as intermediate results’ storage. Since this storage grows/shrinks dynamically to follow program execution needs, overflows and underflows are possible, disrupting the execution [200]. Complex and expensive tools such as depicted in [361] are then developed to keep stack issues under control. Ideally there should be no need for any stacks. ***PITFALL O#19***
- **Pipelines (HW)** – This intermediate storage for code fetching acceleration, also called "instruction-level parallelization" leads to execution sequence hazards when the programmer fails to care on the use of certain critical instructions that need to follow strict sequencing/timing [179]. Problems arise when, for example, an interrupt call got into the pipeline just before the interrupts’ disabling statement - this interrupt will still be executed. Pipelines also allow for execution speed optimization techniques such as super-scalar and out-of-order processing, originating speed variations depending on the sequences executed and interrupts [361]. Tools such as [359] make an attempt in providing means for analyzing these realities, but they suffer from the difficulty of taking this effect into account [364]. Ideally there should be no need for pipelines. ***PITFALL O#20***

- **Flags (HW)** – This intermediate storage is used for storing temporary instruction status results. These results are then used for conditional instructions such as branching, arithmetic instructions such as additions with carry, etc. Problems arise when these are externally changed due to bugs in interrupt sections, therefore potentially trashing an entire block of code which has nothing to do with that external code that trashed the flags. This effect is due to the fact that many conditional, carry-influenced and other instructions are not 100% independent and contained, thereby being influenced by external interference and also causing potential interference to other code blocks. Ideally there should be no need for flags. ***PITFALL O#21***
- **Registers (HW)** – This intermediate storage contains temporary results from instructions, which are generally used in the next instructions. Generally, this storage is used to accelerate program execution in that it resides in internal CPU ultra-fast memory. In most CPUs, instructions are tuned to using these registers very efficiently, accessing external memory only in the extreme cases of reading-in first values and of writing-out end-values [203]. Ideally there should be no need for registers. ***PITFALL O#22***
- **Pointers & Handles (SW)** – Pointers as intermediate "storage/data address" entities have served the purposes of developers for many decades, since they appear both in languages (variable whose value is a memory address) and in processors (indirect addressing registers, etc.), as references to referencing data structures, code segments and other objects of various kinds. At least one type of pointer is almost ubiquitous among CPUs: the *"stack pointer"* The use of pointers is a major source of software bugs throughout languages allowing for that programming construct. Ideally there should be no need for any pointers or handles of any kind. ***PITFALL C#23***
- **Branch Prediction (HW)** – This execution acceleration is used to infer about the most probable branch directions, trying to anticipate pipeline filling with the next instructions, instead of waiting until the branch itself and having to fill it only at that time. If this branch forecast was wrong, then the pipeline is flushed and acceleration is not achieved [202]. This type of hardware influence introduces some non-deterministic timing issues that could pose problems when profiling a program, due to the variance in execution times. Ideally there should be no need for branch speculation. ***PITFALL O#24***
- **Superscalar Processing (HW)** – Typically associated to pipelines, these execution speed optimization techniques fetch several dissociated program instructions in parallel [232]. These are then executed in separate processor sub-cores. The related "out-of-order execution" technique [233] also takes advantage of this type of multi-executing processor cores, with the aggravator of containing special mechanisms and internal memory/state to allow for re-ordering the results of those executions. As with the comparable "branch prediction", this technique also originates timing jitters due to the non-uniform way of executing more or less instructions simultaneously. Ideally, there should be no need for these types of execution-speed optimization techniques. ***PITFALL O#25***
- **Interrupts (HW)** – These code blocks are called in the middle of the execution of main program blocks, or even in the middle of other interrupts (nested interrupts). Potentially random interrupts are certainly the most feared of all programming constructs, due to their intrinsically disruptive and unpredictable intrusive nature, always causing some degree of main program timing jitter [180]. Random interrupts can also make software testing extremely difficult. Badly programmed or ill-behaving interrupts, or bad interrupt inter-relations, are prone to trash an entire program package, which can unpredictably happen anytime and anywhere. Ideally, there should be no need for interrupts at main program level, transferring the strictly needed ones to outer sites or using high-frequency polling instead (better controlling the servicing moments). ***PITFALL C#26***
- **Register Renaming (HW)** – Compilers do the arduous job of producing machine-code including assignment of available registers to the various machine-instructions. This conceptually simple technique then dynamically reassigns register at processor-level, so that initially serialization-forced code can be parallelized without instructions using the same registers disrupting themselves mutually. Processor complexity increases. Ideally, there should be no need for register renaming. ***PITFALL O#27***
- **Side-Operations (SW)** – The vast majority of processors use instruction sets that need certain instructions to do all the input data preparation and resulting output data storage "side-operations". These preparation/post treatment "side-operations" are the first (data transfer instructions) of the three instruction-families present in virtually every processor (the other two

are: arithmetic/logic and flow-control instructions). Processors use temporary registers for intermediate storage, potentially leading to problems related to messing up those registers. Moreover, these side-operations also heavily blur the locations of operations that really matter. Ideally there should be no need for this preparation and post treatment, since the hardware should allow "self-contained" instructions with any extra side-mechanisms needless "by-design". ***PITFALL O#28***

- **Temporary Storage (SW)** – Constructs such as filters, delays, integrators, etc., need temporary storage that is needed to retain their internal states between calls. This leads to more spread-out data to be taken care of. This additionally causes developer errors to happen, since these and other nuances are mistakenly not always taken into account. Ideally, this temporary storage should be somehow automatically taken into account by the development system itself, so that errors are reduced to a minimum. ***PITFALL C#29***
- **Memory Structure (HW)** – Large amounts of ROM/FLASH memory usually contain the code to be executed and, most times, large amounts of RAM memory usually contain the variables upon which the code works on. Most of the times, these memories and other types of storage are mapped onto a contiguous memory map. Since in these cases the processor may access whatever address it needs, simplifying the addressing structures when compared to custom solutions, the danger of the processor somehow trashing this memory is not zero. Ideally, there should be a way of eliminating these dangers. ***PITFALL C#30***
- **Multi-Assignment (SW)** – In virtually all classical imperative language, the same variable may be assigned to multiple times along the program. This potentially causes complex bugs because of the side-effects each assignment has on other program spots also processing that same variable and getting different values from the originally expected ones at those spots. This effect causes debugging to potentially turn into a big problem with some programs, especially those where parallel processing exists in the form of threads, multi-cores, shared-memory, etc. Ideally, this should be sufficient for automotive software. ***PITFALL C#31***
- **Multi-Core (HW)** – Not to be confused with "Scalability" discussed later. This execution acceleration is used where high-level macro-parallelization of programs is possible. Generally, these systems only parallelize independent and separable macro-entities of programs, such as threads, components and, more simply, different programs running on the same hardware. There is the very known problem of "code-distribution" and resulting problem of "load-balancing". Besides synchronization, cache-coherence and extensive communications problems/bottlenecks, there is the more philosophical problem of what rules to take for that distribution when all or most cores are "indistinguishable" from the system's task assignment/location point of view. This problem then also relates to the need of parallelizing compilers, parallel debugging tools and so on. In practice, there should be a more advanced software structure so that it better adapts to the nowadays absolutely necessary multi-core hardware platforms. Thus, the software partitioning should be more or less "seamlessly" related to the corresponding hardware being partitioned in a most similar way, so that the current ultra-complex tool-chains adapting SW to the HW would not be necessary any longer or at least greatly reduced in importance. ***PITFALL O#32***
- **Verification of HW Correctness (HW)** – Functional verification of modern processors is most often more complex than designing them altogether. Just like legacy processors, these are still verified for their correct operation through mostly exhaustive methods. These can take ages to not even fully cover all possibilities a processor might encounter during normal program execution. Ideally, this kind of 100% covered verification of a processor should be somehow not needed or greatly reduced during processor design itself. ***PITFALL O#33***

For a more detailed explanation of these pitfalls listed above, please refer to [Att. 10]. There you may find extensive details and even some examples.

Being able to cope with all of these low-grained pitfalls has become a simple matter of historical habituation. As seen in these previous sections, hardware features either cause extra state/context generation or execution sequence disruption. "Context-content" in processing mechanisms is a consequence of having pitfalls such as flags, registers, pipelines, caches, etc. These cause those mechanisms to be failure-prone just because of having to maintain 100% integrity. Also, when data gets spread-out, it is obviously harder to maintain appropriately. The slightest corruption most probably causes the entire system to fail. They present additional potential places for fatal data corruption [204]. State/context generating features cause the executing core to be influenced in its results and potential low-level bugs may arise from this influence. On the other hand, software features basically and mostly introduce information diversity. This diversity is also the potential cause for high-level bugs to appear (and, thus, high-level pitfalls, detailed below herein). Desired influences at execution-level, interrupts and language diversity, has everything to do with the way programming and executing the programs have been conceptually thought of and then implemented in hardware, along computing evolution.

All of these pitfall features do not directly relate to the main purpose of a program: fetching input values from memory, processing them according to a particular instruction, and finally storing the result(s) back into memory. In other words, none of the above features is really necessary to achieve this conceptual goal. Ideally, the processing part of the hardware could be composed solely by an executive core without any external influence other than the input values themselves.

As an example of enhancement, consider "conditional branching", which is an absolute necessity to achieve Turing Completeness [172]. For it to operate, some sort of decision-variables is needed. But instead of using flags which cause extra context-generation inside the processor, one might just use any other normal variable stored in main memory for this effect. The advantage is that there is no additional context or state that could be potentially disrupted in any way. The executive core would thus just be fed with input values and output the result(s), starting over again for the next input(s). Since it would not have any internal context/state recording, it would execute each instruction without even slightly remembering what happened in the immediately previous execution. This kind of idea contrasts with complex structures such as those depicted in Fig. 3.3:9. Fig. 3.3:10 then compares "normal" diversity-full programming language "C" with its conceptual "bare-foot" or "amnesic" executive core counterpart.

An indispensable "program-counter" keeps track of the current instruction being executed, while an ALU (Arithmetic-Logic-Unit) makes all calculations. Since most automotive operations are calculations related to digital signal-processing, the ALU occupies most of the executor core. All context-related information is 100% represented by the memory that contains all variables used by the program.

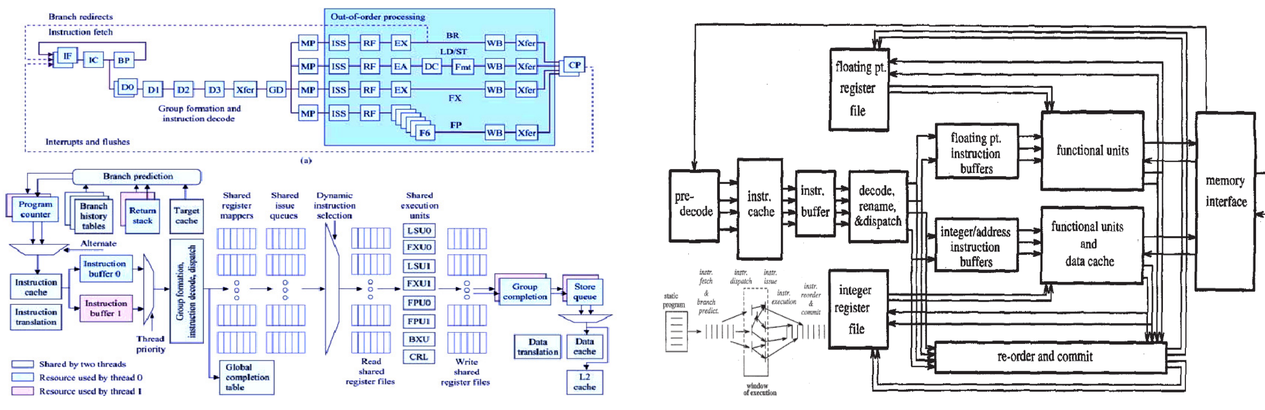
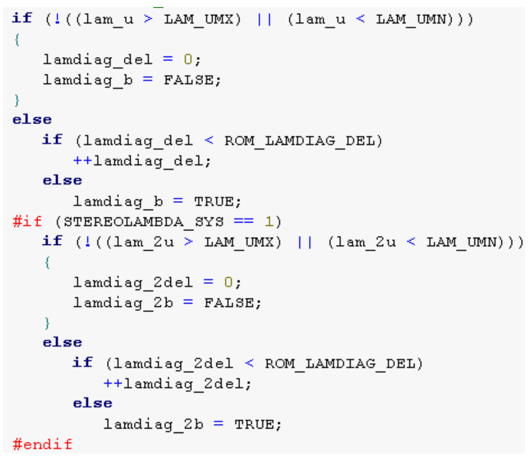
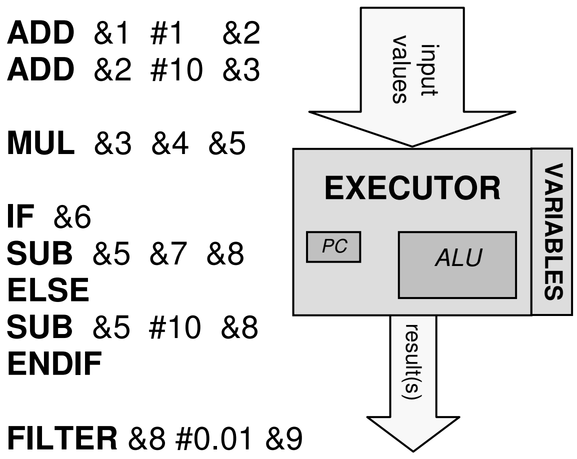


Fig. 3.3:9 – Left: "Power5" processor architecture details (source: IBM [173])
Right: super-scalar processing mechanisms and details (source: [232])



```
if (!((lam_u > LAM_UMX) || (lam_u < LAM_UMN)))
{
   lamdiag_del = 0;
   lamdiag_b = FALSE;
}
else
   if (lamdiag_del < ROM_LAMDIAG_DEL)
      ++lamdiag_del;
   else
      lamdiag_b = TRUE;
#if (STEREOLAMBDA_SYS == 1)
   if (!((lam_2u > LAM_UMX) || (lam_2u < LAM_UMN)))
   {
      lamdiag_2del = 0;
      lamdiag_2b = FALSE;
   }
   else
      if (lamdiag_2del < ROM_LAMDIAG_DEL)
         ++lamdiag_2del;
      else
         lamdiag_2b = TRUE;
#endif
```



```
ADD  &1  #1   &2
ADD  &2  #10  &3

MUL  &3  &4  &5

IF  &6
SUB  &5  &7  &8
ELSE
SUB  &5  #10  &8
ENDIF

FILTER &8 #0.01 &9
```

Fig. 3.3:10 – Left: full-featured "C" language and full-featured "Power5" processor (source: IBM [173])
Right: theoretical "zero-featured" language and "zero-featured" processor concept

As seen previously, “context” and "diversity" are major common denominators among most of the features and permeate the base of all past and current processing systems are based on. Related non-trivial development challenges, problems and even systematic pitfalls are well known and studied [171] [271] [189] [235] [186] [272] [273] [276] [242] and it is also clear that most if not all of them can only be effectively and efficiently eliminated if the related mechanisms are eliminated themselves. Developers may invent the most safe (and necessarily complex) mechanisms to avoid errors and problems derived from those language and processor enhancing features, but since they exist, related problems lie there permanently as potential pitfalls.

As there is no such thing as a “bug-free computing system” on either software or hardware sides or both [260], the best solution may well be to avoid using complex side-features altogether. Even bringing obvious performance increase, the benefits of making certain errors and problems totally impossible to appear "by design", may represent greater gains than costs. This is currently called “correct-by-construction” [262] in the literature, and plays a central role throughout this thesis’ work. But unlike normal implementation of this concept through the "redundant" design methods described in [262], it will be sought after through elimination of the troublemaking mechanisms. In short, this goes into full resonance with the expression used to attack the central problem of computing systems: *"How to make the machines do our bidding"* in [12], with no strings attached. Again, this last excerpt from [12] shows the way that this thesis' work also tries to follow. There is a compelling need for a language and hardware that better comply with needs of automotive systems in engine/chassis control. Note the "parallel processing" tone of the excerpt:

We are our own best model for computer organization. It is our work that we want the machine to do and we have some idea of how we go about it. It is apparent that we can greatly improve the performance of our systems by allowing and controlling unboundedly parallel operation. This system goal implies the invention of a class of artificial languages within which that control will be conveniently expressed. We will be better off if we take natural language concepts as our model than those of conventional programming languages; no matter how machine independent we wished our programming languages to be, they all have been overwhelmingly sequential, arithmetic and random access memory oriented. The obvious attach for programmers and hardware people together is to devise language that reflects what we want to do and how we do it (for instance, in parallel) and machine structures effective in handling that language. Let us call this method "language directed computer design."

Although it can be argued that the above mechanisms greatly accelerate program execution, this thesis' argument goes as to say that making the language and hardware dipole totally adapted in a biunivocal relation, most of those hardware mechanisms can be rendered too power-less to even justify the effort of their implementation. Another argument might be that all these pitfalls are already so well studied, contained and even protected by certification and validation, that it would make sense to keep all of them for the sake of keeping systems working. A whole industry and market gravitate around these, where developers have already mastered circumvention mechanisms to keep everything working smoothly. All this may be true, but one thing is certain: they remain as being important pitfalls, while predictions regarding the industry's lethargic evolution are still valid until today and will stay that way in the near future.

An exception in treatment is made, of course, regarding the parallel processing mechanism of having multi-core hardware, but still in a much different way than envisaged up until now. Nevertheless, all other acceleration and enhancement mechanisms altogether boil down to an obligation to the user, who cannot simply concentrate on the really necessary core of the tasks: the algorithmic problem itself. An experienced embedded software programmer must be multi-talented in that he cannot simply translate a problem into an algorithm, but also has to deal with data types, optimization issues and a whole multitude of hardware details. If he fails doing so, the final system might never work as expected. This already shows the big and growing gap between what a user wants and what he has to do to make the hardware really accomplish that.

Unfortunately, new developments in hardware and software, above all the multi-core paradigm, are being addressed with classic tools and classic ways of doing everything, so that it is clear that all existing pitfalls will remain basically untouched. Later Fig. 3.5:18 and Fig. 3.5:19 show graphical groupings of these pitfalls.

3.3.2 High-Level Pitfalls (mostly limitations)

[Emphasis: software and hardware handling related pitfalls]

These pitfalls are more related to emergent effects of using current computational systems. These emergent effects most often arise due to a combination of the low-level pitfalls detailed above herein. These arise due to the way of doing things, that are well-adapted to current hardware but do not relate too well to the way things ought to be naturally done. Since these features historically grew up as side-effects of the systems' operation, they are herein considered as being "limitations", as they handicap those systems in a controlled way. The following list shows these effects and how they must be coped with, since there is no cheap way of getting rid of them effectively and efficiently. As previously, these are sub-divided into soft- and hard-limitations. Some of the following underlying features are actually necessary to the user, so the whole problem lies in reducing the trouble they still cause:

- **Too many Coding Alternatives (SW)** – Most programming tools allow too many solutions to one and the same problem, leading to a highly heterogeneous and even confusing source-codes. When more than just one developer works on the same project, which is the most usual case, then things get worse because each developer has its very own "modus-operandi". When a multitude of solutions exists inside the same language, then things get even worse, having to chose. The Sinclair BASIC [350] and the Small BASIC [382] leanness is quite the contrary example, where flexibility exists but where only a few solutions are possible. Similarly to "over-engineering", what happens here is a sort of "over-knowledging". Ideally, there should only be one unique way of solving each problem. ***PITFALL H#34***

- **Architecture & Handling Non-Orthogonality (SW/HW)** – In specialized jargon, a microcontroller being called "100% orthogonal" simply means that if the various address source/destination modes and instructions are placed along the three axes of a Cartesian graph, it must be filled with all usage combination possibilities. An extreme example of software development tool orthogonality would be allowing "interactive programming" [292] [293], "almost online" programming [34] or even *"Live-Programming"* (this later is going to be implemented in this thesis' work). In these three scenarios, the developer or user would be allowed to make changes to the algorithms with little or virtually no re-compilation and re-run delay, make limited changes without ever stopping the system, or even make any changes without stopping the system, respectively. Current real software development tools show quite an abyss between this desired property and the operations they can perform in the various situations. This causes long time delays and difficulties in developing and deploying automotive systems due to the large amount of different but strongly inter-related tasks that have to be accomplished in the automotive scene. The basic three system circumstances are: static off-line(X), runtime/online(Y) and tele-operated(Z) development, deployment and troubleshooting. Current tool complexity and evolutionary layering techniques all contribute to the state of things. Ideally, the software development tools should allow developers and users to perform everything that is conceptually possible in any situation, both in the low-level programming of operations and high-level handling, because this property would drastically reduce all still existing turnaround delays. ***PITFALL H#35***
- **Fatal-Error Recovery, Traps & Resets (SW)** – Although most current systems already include sophisticated error recovery mechanisms, the usual thing to happen when even a single critical bit is erratically toggled and uncaught, is for the entire system to "crash", "hang", reset or, in the worst case, perform some erratic behaviour. Multiple redundant software and hardware systems are devised in aerospace (avionics, space-shuttles, satellites and space-probes) for continuing their flight or mission upon failure of some system [205]. A more basic recovery mechanism is the watchdog [334] itself. Automotive software simply resets the entire system if a "deadline overrun" occurs [150] [10] or if the watchdog period expires. Ideally, there should be no need for recovery mechanisms or emergency resets, since ideally fatal-errors would “by design” not be possible at all. ***PITFALL H#36***
- **Software Aging (SW)** – Not to be confused with the above fatal-error recovery, traps and resets pitfall. When a software process starts cleanly, everything is in its place, but as soon as it runs for quite some time, it often succumbs under memory leaks, data/state corruption, dead-locks, numerical error accumulation and other strange effects, which most often render the whole system useless unless fully restarted [329]. Creating systems with reduced time-to-recover delays and keeping them highly available (mainly in internet services) could be achieved through so-called "micro-reboots" [328]. These allow sub-components to reboot selectively in response to an overall system failure or degradation, instead of rebooting the entire system. Ideally, as time goes by, software should keep working just as in the beginning, without deteriorating in any way or even reaching critical/fatal behaviour. ***PITFALL C#37***
- **Operating-System Stealthing (SW)** – Operating-Systems are used in virtually all embedded systems and also in automotive ECUs, in one way or another. They imply "mandatory limitations" to avoid user mistakes, abuses and to protect other programs running on it. They also manage all system's resources in a mostly "stealth-like" manner [207]. Overhead and stealth activities increase greatly. There are implementations not using any OS at all, such as in [211], using alternative and more simple mechanisms to manage task supervision instead. Ideally, there should be no need for heavy OSs in the first place, since the hardware would “by design” eliminate the need for most higher-level management issues. ***PITFALL O#38***
- **Intermediate Representations (SW)** – Although older interpreters such as for the BASIC language [289] are also a type of "managed code" structures, current tools such as Microsoft's ".NET" [70] [71] [75] [78] and Sun's JAVA [79] [80] programming paradigms have driven this concept farther than ever. This concept mixes with former simpler interpretation concepts since it also translates the original source-code into an intermediate language, which is then executed by a common run-time executive on the hardware. LabView [6] also works in a similar way with its "DFIR" and "IL" intermediate representations [368]. This concept has evolved into a massive form of "code obfuscation" and general "management overkill". Ideally, there should be no need for this extra "code management", since all code would run directly and without any more overhead intermediates, turning any form of management redundant “by design”. ***PITFALL O#39***

- **Pre-Processing/Macro-Expansion (SW)** – Most languages allow more or less pre-processing on their constructs. Macro-expansion is just a part of the same mechanism, but also the most error-prone one. Basically, when source-code is pre-processed and expanded, the final code that the compiler will see is not the original code anymore. Ideally, there should be no need for any additional effort when using macros, since everything would be taken care of automatically by the underlying structure "by design”. ***PITFALL O#40***
- **Hygienic Programming (SW)** – Most languages do have quite a wealth of possibilities in doing things in such a way that a program resembles a huge pile of confusing and ungraspable source-code modules. Add the individual programming style of every developer and this picture gets even worse. There are at least two levels of hygiene when programming: the primary level concerns the language itself and its "strength" in keeping things clean, while the secondary hygiene level concerns the developer and the algorithms themselves. Ideally, there should be no need for any additional effort into keeping a program clean, since everything would be taken care of automatically "by design”. In other words, the program entry mode would itself guarantee a high level of primary software hygiene. ***PITFALL C#41***
- **Supervision (SW)** – This mechanism is needed whenever some processing entity needs to get controlled over whether it still operates correctly. When a system gets locked-up, slowed-down or corrupted, then this mechanism causes a warning, alarm or even system reset. Ideally, no supervision should be necessary, let alone highly complex degradation monitoring mechanisms [330], because it just represents overhead to a system. Self-regulated and/or “performance by design” systems would be ideal. ***PITFALL O#42***
- **Time-Raster Dilemma (SW)** – Automotive systems are both event- and time-triggered, mainly composed by a series of hundreds or even thousands of FDEFs with their inner blocks running mostly in pre-defined time-rasters[*]. Usually, special operating-systems provide these time-slicing containers where all engine/chassis control code resides [150] [10] [371]. Besides the fact that these time-rasters are discrete, the user has definitely to choose among them, which sometimes is a very hard task because of trade-offs and other issues. Priorities and pre-emptive details are additional trouble to choose among. Having many time-rasters in a system turns out to be very demanding when debugging or troubleshooting timing problems or bugs that eventually arise, not to mention the interrupts that further introduce confusion into this. Ideally, there should be no need to care about low-level details such as time-rasters, since all code would ideally run inside a single container fast enough to meet any engine/chassis FDEF timing requirement, thus rendering discrete and explicit time-rasters needless “by design”. ***PITFALL C#43***
- **Inter-Thread Dependence & Synchronization (SW)** – Many applications display data/memory and control dependencies between threads and/or require some sort of related synchronization. This raises the problem of having to have a global clock or at least consistent sub-clocks, which feed those separate threads. Furthermore, even the slightest dependence or synchronization related timing bug may disrupt the entire system. This pitfall also strongly related to the fact that changing code in one spot can disrupt the operation of code in another spot, due to some critical relationship. Ideally, there should be no need for such dependences and synchronizations as in current systems. ***PITFALL C#44***
- **Multitude of Different Timers (HW)** – When developing automotive systems, there are many different timing needs throughout the entire system. Typical hardware timer resolution needs varies among different ECUs and functionalities. Changing one timer may disrupt an entire system due to some overseen dependency. Everything gets obviously harder to maintain. Ideally, each timer should serve only one functionality or, similarly to interrupts, exist outside only the main program to globally simplify software development. ***PITFALL C#45***
- **Timing-Jitter Effects (SW)** – Timing jitter was already mentioned a few times in previous pitfalls, being the cause for a hard time testing systems that crash for "unknown" reasons. One jitter at the wrong moment and everything falls apart. Also, profiling a program can be a very complicated job under such varying timing circumstances. Ideally, there should be some processing method that would guarantee lowest possible timing jitters. ***PITFALL C#46***

[*] *Automotive FDEF time-rasters usually encompass all or some of these essential ones: "sync", 500µs, 1ms, 2ms, 5ms, 10ms, 20ms, 50ms, 100ms, 200ms, 500ms, 1s, where the "sync" time-raster represents the frequency of combustions. This "sync" time-raster is obviously used in functions that calculate combustion-related values, while all other rasters are used depending on the critical level of the to them applied FDEFs.*

- **Floating-Point Discrepancies (SW)** – Although not used in ECUs until recently, mainly because of the processing power needed and FDEF-related issues, the IEEE-754 standard [176] has been long used in automotive development systems. Since this is a binary representation, quantization problems limit it slightly when it comes to display such numbers for the human eye. When internal quantization leads to display artefacts[*], the programmer may accept those artefacts, but customers never will. Ideally, there should be an appropriate "human-sensitive" value representation mechanism through which values would not present visible artefacts "by design". ***PITFALL H#47***
- **Manual/Tedious Graphical Layout (SW)** – When using any of the currently available visual programming tools [5] [6] [7], the user invariably has to manually place the graphical elements while building the data-flow diagram. Although connections may be automatically placed, the operation elements themselves are only manually placed. Ideally, there should be an automatic placing mechanism based on data-flow rules, thereby always keeping graphical elements correctly ordered "by design", while also exhibiting some sort of user-guidance to accelerate the entire editing process. ***PITFALL H#48***
- **Lengthy Turnarounds (SW)** – When it comes to set development details such as which variables should be monitored or logged, which operations should be breakpointed or hidden, current tools usually present a large distance from desired ways of doing these settings. Tens and even hundreds of such delays are usual in current systems, summing up quite some "wasted time". Ideally, related options and features should be together and not scattered over menus and dialog-boxes, readily available without delays and they should be well visible to the developer during development or debugging. ***PITFALL H#49***
- **Difficult Differences-Inspection (SW)** – There is always the need to compare code and especially FDEF versions in visual tools. Due to the above mentioned intrinsic need of manual placement of graphical elements, there is a high chance that different programmers will design the same FDEF very differently, rendering graphical comparison attempts rather useless. The required tool should naturally emerge as a language-oriented reality and not as a later add-on requiring high development efforts. ***PITFALL H#50***
- **Graphical Elements Cluttering (SW)** – Whenever an FDEF gets big enough to not fit the screen or to not fit the human short-term memory capacity, it gets difficult to grasp and handle its details. This happens with most of the automotive FDEFs today. Ideally, there should be an appropriate manner of selectively filtering or masking out some secondary segments of an FDEF, to allow the developer to better grasp the parts that really matter at each given instant. ***PITFALL H#51***
- **Binary Data in Files (SW)** – Although a necessity in many cases such as compact image and sound encoding, encoding/saving projects in binary format, turns it unreadable to humans. Encryption and compaction are the main reasons for having binary encodings in the first place. Ideally, there should not exist any binary data in files, but this is mostly a necessity. The most that can be done is to use some sort of combination, just like the XML format, which even also allows direct user editing if necessary. ***PITFALL H#52***
- **Impossible Reverse-Engineering (SW)** – When source-code passes the various stages of parsing, compiling and assembling, resulting in binary machine-code, it looses all human-oriented characteristics such as variable labels, function names, etc. Add auto-code generation from visual diagrams and even compiler optimization into this bowl and things get even worse in terms of human ability to understand what the final machine-code really does, not to mention getting structure information out of it. Reverse-engineering has always been a need in various situations ranging from legacy systems' revivals to emergency recoveries in Motorsport scenarios. There should thus be a much tighter relation between source-code and machine-code but, again, as before mentioned in [12], the relative abyss between hardware and corresponding languages has separated these two worlds beyond mutual recognition. Thus, as also brilliantly remarked in [12], this demands for a cooperative development of a language and hardware package growing alongside one another, thereby allowing reverse-engineering to be a natural feature in it. ***PITFALL H#53***

[*] *Simple experiments with the IEEE-754 standard allow to show quantization-affected results like: [0,70 * 1,05 = 0,73500001430511475]. The correct result would be 0,735 but the internal minimum binary step did not allow this exact value. Although [x = 35,0 / 100,0] results in 0,350000, [100,0 * x = 34,999999] and [x + x + x + … + x (100 times) = 35,000011]…*

- **Multiple Software File Formats (SW) –** It is well known that every development environment produces huge amounts of file contents within different formats. The origin of those different formats is the fact that those systems contain correspondingly complex layered structures, where every layer is a potential producer of yet another format that may or may not be outputted to disc. Ideally, there should be only one unique file format that would include everything needed for a project. Not just a huge file with the same previous different formats inside, but rather a file with a single unified format to represent all program and handling details. This will certainly mean that the tools themselves would have to be overhauled regarding this matter. ***PITFALL O#54***
- **Abstraction Losses (SW)** – Virtually all existing systems abstract their hardware and software in various ways, to be able to reach the programmer or user in a simplified way. In fact, this is quite a natural thing to happen. The real problem arises when this abstraction has to be so deep, buried and multi-layered: besides the top-level having nothing more in common with the underlying system, there are also many system controlling details that get lost along the abstraction path. Also, when problems and bugs arise, it is often useless trying to figure them out at such complex abstraction levels. Ideally, there should be no need for any extensive abstraction needs. ***PITFALL H#55***
- **Missing/Impossible "upward/return path" (SW)** – Virtually all legacy and state-of-the-art development tools and especially their inherent tool-chains are ultimately useful only the "way down". Developer program on a high-level language and user-interface, which gets translated into ever-lower intermediate representation until it reaches the hardware machine-code level. This is an intrinsic characteristic of layered architectures, due to the "one-way" (electronic diode-like) nature of each layer, where the "reverse-current (I_R)" is extremely low or even zero. Another complete software "bottom-up" structure has to be implemented in parallel to the "top-down" one to be able to circumvent the diode-like limitations. There should be the possibility of somehow marrying both "top-down" and "bottom-up" structures, so that the synergetic *"Live-Prototyping"* would be made possible. This would mean that both structures would eventually operate as one single entity, thus implementing both "building" and "debugging" processes into that single structure. ***PITFALL H#56***
- **Lack of Command-Line (SW)** – Advanced users would often wish to enter commands and programming operations directly in text form. This is especially the case when certain tasks are performed rather frequently. There should be the possibility of entering and manipulating programming operations through the graphical interface as well as through a dedicated command-line, by using a "T"-pipe mechanism thus allowing actions to come from two or more distinct sides. ***PITFALL H#57***
- **Building Error-Messages (SW)** – This is a major tumblestone in development systems, due to the large amount of internal constructs and layers. Sometimes the error messages while compiling are so cryptic that the easiest way is to simply write the affected program statements in another way to get rid of that type of message. Ideally, there would not exist any building error-messages with a language that would not allow any such errors, being intrinsically correct "by design". ***PITFALL C#58***
- **Runtime Error-Messages (C)** – Virtually any program written in any language, when run, potentially displays some error-message during that runtime. Being highly disruptive by nature, since being potentially the result of a crashed or partially crashed system, theses messages may not be fatal on desktop applications, but they sure would be in automotive systems. Ideally, there would not exist any runtime error-messages with a language and system that would not allow any such errors, always being capable of self-correcting itself or in any way even being intrinsically correct "by design". ***PITFALL C#59***
- **Costly Live-Feedback & Live-Manipulation (SW)** – Not to be confused with the previous error-messages' feedback. The ideal way of inspecting, debugging and otherwise manipulating a system is to have instantaneous feedback upon values and other execution states. But due to the highly opaque and historically grown layered structure of current automotive tools, this turns out to be a "herculeous" task. There should be a way of including intrinsic/non-intrusive debugging hardware structures inside the system and keeping them closely matched/linked to the external debugging software. The same way, this software should be seamlessly integrated into the main visual user-interface, to create a complete and seamless package, which would allow all editing/debugging the system's program. ***PITFALL H#60***

- **Deferred Debugging & Prototyping (SW)** – When a user debugs and prototypes on a system, he naturally wishes high response speed, total actions' flexibility and seamless integration within the same user-interface. Except for some non-automotive systems such as [70] [71], current automotive development systems do not allow for integrated and seamless prototyping features. The still too long time delays that currently accompany "fast" or "rapid" prototyping should be eliminated, providing the user with a fully responsive system at all time. ***PITFALL H#61***
- **Execution Profiling (SW)** – Profiling a program is of utter importance when trying to optimize algorithms, finding processing bottlenecks or trying to fit code blocks onto critical time-slots. Currently used processors with increasing speed-up hardware add-ons such as caches, pipelines, speculative parallel execution strategies, etc., cause real problems in achieving reliable profiling packages. Moreover, real hardware is a must be used, due to too many unknown variables/effects. Critical real-time systems such as automotive ones, make all this worse. There should be a simple way for assessing profiling data directly from the source-code itself, without having to rely on real hardware measurements. ***PITFALL H#62***
- **Mixed Simulations (SW)** – It is well-known that efforts to add simulation capabilities to existing systems can turn out to be a "herculeous" task. This often has directly to do with the simple fact that languages did not evolve with the defined purpose of allowing easy simulation integration. There should be a simple way of adding simulation capabilities in a completely integrated and seamless manner, in such a way that would even allow to freely and limitedlessly mix simulated and real hardware, as well as simulated and really hardware-processed automotive functionalities. ***PITFALL H#63***
- **Side-effects & Bug Propagation (SW)** – In classical systems, when some software bug, external interference or even sporadic hardware failure at some statement disrupts an internal property such as the stack, registers or flags, it is most sure that subsequent statements if not the whole rest of the program will misbehave and generate wrong outputs and actions. There should be a way of conceiving the instructions in such a form that "architectural-path" bug propagation would be impossible "by design", through error containment at instruction-level instead of object-level as in OOP. ***PITFALL C#64***
- **Execution Repeatability (SW)** – Essentially due to all the accelerations, optimizations, gimmicks and accessory mechanism around classical processing systems, it cannot be guaranteed at all that the exact same instruction at exactly the same point inside a program will always be executing and resulting in the same way. There should be a way of executing each instruction in such a way that would guarantee exactly the same results anywhere they are used and forever. ***PITFALL O#65***
- **System Testing (SW)** – The proof that embedded systems may misbehave at the most improbable spots in the code and that awkward effects appear when least expected, is given by "once-in-a-year" weird gas indicator freeze at ½ of the tank even after complete refill, "once-every-month" RPM gauge havocness with weird peaks even with the engine stopped, "once-in-a-lifetime" engine idle-speed havocness (engine revving largely irregularly) just after engine-start which is only ended after a restart, etc. This calls for extensive system testing and yet some bugs will leave the factory unnoticed. There should be a complete yet seamless and non-intrusive way to test programs inside the development environment itself, instead of having to implement separate intrusive tools for that matter. ***PITFALL H#66***
- **Code Metrics (SW)** – Many different metrics have been developed to trying measure code quality. Although a good thing in principle, the vast range of different programming languages, programming paradigms and programming environments have made those measurements an extremely inaccurate and difficult matter. Ideally, there should be a very reduced set of metrics, reflecting a very lean and straightforward programming language, without all those syntactical and styling possibilities. ***PITFALL O#67***
- **Tool Certification (SW)** – Not to be confused with "Coding Reliability" discussed earlier. This relates to the development tools' components themselves. Current systems whose correctness and reliability are absolutely necessary, such as compilers and auto-code generators used in avionics, military and automotive mission-critical applications, must be certified and validated. Auto-code generators for automotive applications must obey a whole list of complex requirements [267]. These certifications are associated to big effort, time and cost issues. Ideally, there should be no need for certification of development lower-level support components, since these should be mostly "correct by design". ***PITFALL O#68***

- **Tele-Operation (SW)** – Local troubleshooting in presence of the failed automotive system can be immediately begun, by simply physically attaching the appropriate diagnosis equipment to the system. To achieve the same flexibility when it comes to troubleshoot systems that are thousands of miles away, may become a problem. There should be a simple way where to implement this feature, such that no main structures have to be altered, by simply adding a further "T"-pipe to the common user-entry "graphical-interface/command-line" mentioned earlier. This "three-way action entry" mechanism calls for a narrowing-down to a unique "action-entry function" inside the software. ***PITFALL H#69***
- **Human Remote Support (SW)** – Related to Tele-Operation, human interaction with the persons on the remote site is invariably accomplished through ordinary telephone calls. This poses the problem of having to employ extra channels and potentially increasing turnaround delays. There should be a simple way of integrating also this feature to the previous Tele-Operation mechanism. ***PITFALL H#70***
- **Peripherals' Handling (HW)** – Current automotive systems use very few "smart" sensors and actuators, employing mostly passive peripherals. When it comes to using these last ones, corresponding software adaptations, changes or even overhauls have to be necessarily done, since all the driving software resides inside the ECUs. This happens when the same ECU must operate across projects with high levels of diversity. For the same reason, if a new peripheral is to be used and if it requires different electronics (also inside the ECUs), overhaul of the electronics has to be done. Also, the CPU inside the ECU has to have all necessary electronics for all passive sensors. Despite cost-issues, it should be interesting to be able to treat peripherals as separate self-processing and fully self-sufficient software/hardware pieces, which would allow swapping among similar peripherals as well as using them directly "off-the-shelf" without any worries whatsoever about programming and calibrating, etc. Another obvious advantage of this is the fact that all electronics and software would be outside the main ECU and thus do not interfere in the ECU's internal electronics and, more importantly, in its interrupt, flags, stack and registers structure. ***PITFALL H#71***
- **Peripherals Interfacing (SW/HW)** – All kinds of problems arise when connecting CPUs and micro-controllers with needed peripherals (ADCs, DACs, potis, communications, storage, etc.), either internal or external ones. In case of internal peripherals inside so-called "micro-controllers", only the physical connection troubles are reduced, in that everything comes inside the same silicon package and thus everything from timing to processing core cooperation details should be more easy to harness out of the factory. Ideally, there should be no worries about this interfacing. ***PITFALL O#72***
- **Global Software/Hardware Diversity (SW/HW)** – Current systems rely on many components that are manufactured according to different software and hardware plans. This leads to severe cost and time as well as to maintenance consequences. These are getting so aggravated by the multitude of different components in automotive vehicles that companies are really getting to the point of having very complex challenges [227] [278]. There should be a common conceptual technology platform from which at least the most important software and hardware sides of the systems could be derived from, such as the ECU itself, dataloggers, programming environments and even peripherals. ***PITFALL H#73***
- **Intrusive Handling (SW/HW)** – It is well known and even accepted as virtually unavoidable to have some level of intrusion when it comes to inspecting values or interacting with them inside a processing unit. System manipulation leads to the undesired probability of interfering with the behaviour of the system, much like the known Heisenberg's "Uncertainty Theory*". The issue of debugging, monitoring and calibrating mechanisms is still buried deep inside any automotive firmware of current days. The external tools that use such mechanisms, such as INCA [93], MatLab [299], ASCET-SD [7] among others, still have to communicate with the ECU through code components especially developed for this purpose. Normally, these code components exist at the same level as all other automotive functionalities, such that they may cause and in

** Who never witnessed the "printf effect" while debugging a program? The program behaves not as it should, so the developer places one or more "printf" statements to display some intermediate values or variables. After recompilation the program runs smoothly without errors and the "printf" statement displays the correct/expected values. As soon as that "printf" is again eliminated, the program runs berserk again… The explanation for these strange "Heisenberg"-like phenomena is that there are very subtle timing and disturbance properties, which engage under even more specific conditions. It is most probably not the "printf" statement itself that alters the program's behavior, but rather some rare shift or delay that it causes in the compiled output at some other location. This location may not even be the bug location. Needless to say that these effects are quite disturbing and difficult to handle.*

fact really cause interference and performance issues. There should be a parallel path where these entities would grow and where their presence should not be noticed, turning this type of handling into totally intrusion-less mechanisms, while leaving the developers with only the real automotive firmware to program with. ***PITFALL H#74***

- **Difficult Scalability (SW/HW)** – It remains a very hard and complex task to have systems that are easily scalable, especially what hardware is concerned. There should be a relatively easy way of growing hardware, allowing it to scale up to more processing power and capacity in all respects (memory, communications, peripherals), without all the disadvantages and problems caused by parallel structures. ***PITFALL H#75***
- **Complex Environment Installation (SW)** – One pitfall that occurs right at the moment where a computer must receive a development environment, is when the installation process begins. Although current installation packages are generally easy to use, file diversity, components, add-ons, options, etc., causes a big agglomeration of footprints that leave any user at a loss when something goes wrong. Furthermore, current software package rely heavily upon 3rd-party software such as ".NET" and any other support structures. There should be a relatively easy and direct way of installing the necessary software on the host computer as a compact package, without all those sub-components, sub-packages, extra/optional add-ons and dozens of other details. ***PITFALL H#76***
- **3rd-Party Tools & Plug-ins (HW/SW)** – These software "add-ons" have been an easy way of extending tools' capabilities since a long time now. The problem with this mechanism is that these pieces of software are very often supplied by 3rd-party companies. This usually produces a high degree of compatibility problems due to the limited knowledge of the insides of the original tools where those add-ons must reside in an interoperate with. There should be no need for extra external hardware and software tools. The system should be simple enough to get everything working without external help, since this frequently messes things a degree higher. ***PITFALL H#77***
- **Development vs. Deployment Modes (SW)** – Generally, most development tools have a dedicated "development mode" and another dedicated "deployment/management mode". One example are the "design" vs. "runtime" modes within Microsoft's Visual Studio programming tools [70] [71]. These modes have mostly been merged so that there is little difference between the two. There should be exist the possibility of having only one single system mode: one where everything would be readily possible without any more turnaround times, delays or need for user action to toggle between modes. Everything should be intrinsically working and ready at all times. ***PITFALL H#78***
- **Processing Focus (HW)** – Classical systems always present some degree of processing problems to the developer/user. These problems most often result from all the low-level pitfalls explained here so far, such as stacks, registers, flags, and especially complex mechanisms such as pipelines, superscalar processing, register renaming, branch prediction, etc. Generally, the developer spends much of his debugging time to CPU issues and all its surrounding "auxiliary processing mechanisms". Developers should be able to focus on their high-level algorithms and programs only. ***PITFALL O#79***
- **Long Learning-Curves (HW/SW)** – One of the immediate and big problems of getting along with current development tools is the extremely long learning-curve they require to even get the simplest and most needed things done. Nowadays, even the whole installation procedure of a tool can add significant time to this curve. Ideally, learning curves should be display a fast-saturating behaviour, in that all most important and needed features are grasped instantly, while some less used details may be learned afterwards if necessary. ***PITFALL C#80***
- **Overwhelming Manpower & Documentation Needs (HW/SW)** – Since the beginning of computing that manpower needed to harness a processing system has been undoubtedly steadily rising over time. The causes are not limited to the rise of automotive system's complexity in itself (due to rising functionality and software components) but can also be tracked back to the complexity of the hardware itself (ever-growing processors with ever-growing amount of inner components and mechanisms). All this growing complexity also had to be documented along the way, so that today the overwhelming need for manpower and documentation reaches both hardware platforms as well as software. Ideally, the entire system should be simple enough to keep everything working without very large teams, since this frequently messes things a degree higher. ***PITFALL C#81***

- **Tool "Schizophrenia" (SW)** – Not to be confused with "Over-Engineering" explained below but also caused by it. This pitfall directly relates with the user's confusion when using some tool that does not behave like it should despite all attempts being made to understand its bizarre operation at some point. While the user expects the tool to behave in a certain way, lead by experience, tool descriptions/manuals and other cues, the user experiences "though disintegration" in the sense of not having the slightest clue of what is happening. Ideally, a tool should behave exactly as expected, where impossible actions should be hidden or clearly marked as such (through button/menu greying). Additionally, tools should not have hidden features or commands, leaving all possible actions well visible at all times and at the right spots. Software tools need to be a little less confusing and more obvious to work with. ***PITFALL H#82***
- **System Hardware Debugging (HW)** – Classical systems tend to get into a messy situation when one tries to debug them and to search for system hardware level problems. While small micro-controller based embedded systems may still be not that hard to debug at hardware-level, larger embedded (e.g. series-car ECUs), desktop and even aeronautic/space ECU reach levels of individual component complexity, difficult component inter-meshing and architectural details confusion, that debugging a tiny hardware bug/glitch/problem potentially gets highly time- and resource- consuming. There really should exist a way to enable the desired global system behaviours with drastically simpler hardware components and, what's even more important, with less inter-component dependencies and relationships. ***PITFALL O#83***

For detailed explanation of these pitfalls listed above, please refer to [Att. 11]. There you may find extensive details and even some examples.

3.3.3 Over-Engineering

[Emphasis: the centre of all pitfalls, specialists in coping with over-engineering]

This SW/HW high-grained pitfall is what happens when a product is more robust or complicated than necessary for its application. Some bits of over-engineering is desirable when safety or performance are crucial, such as in the automotive scene, but becomes wasteful and even hard to cope with when the application would not require it to that extent. Although not always directly caused by the "programming equivalent" of the very famous "Peter Principle[*]" [459], "Over-Engineering" is often the result of a very inertia-affected manpower-driven circumstance in which complexity increases rapidly until it reaches a level just beyond that with which programmers can comfortably cope. At this point, complexity and our abilities to contain it reach an uneasy equilibrium. The most complex system which can still be made to work is thus built.

Over-engineering is really one of the worst, if not the worst of all pitfalls, because it is able to combine almost all of the low-grained pitfalls into itself, while giving rise to almost all of the high-grained ones, in one way or the other. That is why it receives special attention in this last pitfall point. Complication of processes and systems is maybe the main hurdle when it comes to ease of handling of a system. When things seem to be made for Engineers or specialists only, the average customer rapidly gets into trouble using that system. It is becoming common-sense among programming communities, that coding too many features (including too many unusable and incomprehensible ones) on highly complex hardware demanding too much care, leads to software with more bugs. So, the premise among those communities has also been centered upon trying to reduce programming efforts and care to the necessary minimum. Still, this necessary minimum is still highly bloated due to all the pitfalls already listed herein above.

[*] *The original "Peter Principle" states that "In a Hierarchy Every Employee Tends to Rise to His Level of Incompetence" [459].*

On the other hand, the developers themselves have a hard time when having to deal with development systems that are over-engineered, with too many controls, options, switches, alternatives and even overlapping features. But the worst part is the fact that most development systems allow too many possibilities. Applying the Pareto Principle [247] on the feature diversity present in current tools, probably 20% or less of those features would be sufficient for 80% of their users. And the rest of the users (20%) probably need the rest of the features (80%) just because they are used to doing things the complicated way. For more details, examples, anecdotes and illustrations about over-engineering, please refer to [Att. 12]. Some interesting *Rube Goldberg creative contraptions* [251] can be found there as well, to even better illustrate what is really meant by over-engineered systems executing simple tasks.

In this thesis, "overhead complexity" always refers to some operation that should be represented and executed in only one way, if that operation really only represents only one action, such as adding two numbers arithmetically: just add them! Many formats, quantizations, syntaxes, dependencies, etc., just mess up what the programmer really wanted to do from the very beginning: just adding two values. Moreover, that simple "addition thought" should always produce the same result. That's not quite what happens in practice, where sometimes dozens if framework constraints, programming pitfalls and code dependencies rule otherwise, as illustrated in Fig. 3.3:11, where instead of a plain correct result, we mostly get the result plus several hypotheses of something going wrong (pitfalls) and the result being skewed by some undesired effect/remainder coming from the programming framework and respective tool-chain themselves. In the technical realm this is quite time-consuming, quite unnecessary if the framework would be efficient and source of all kinds of problems arising in the final system where this single addition operation is to be integrated. Article [12] is a great philosophical reference: as implied there, programming actions should unfold into: IDEA → IMPLEMENTATION → RESULT$_{\text{[correct]}}$.

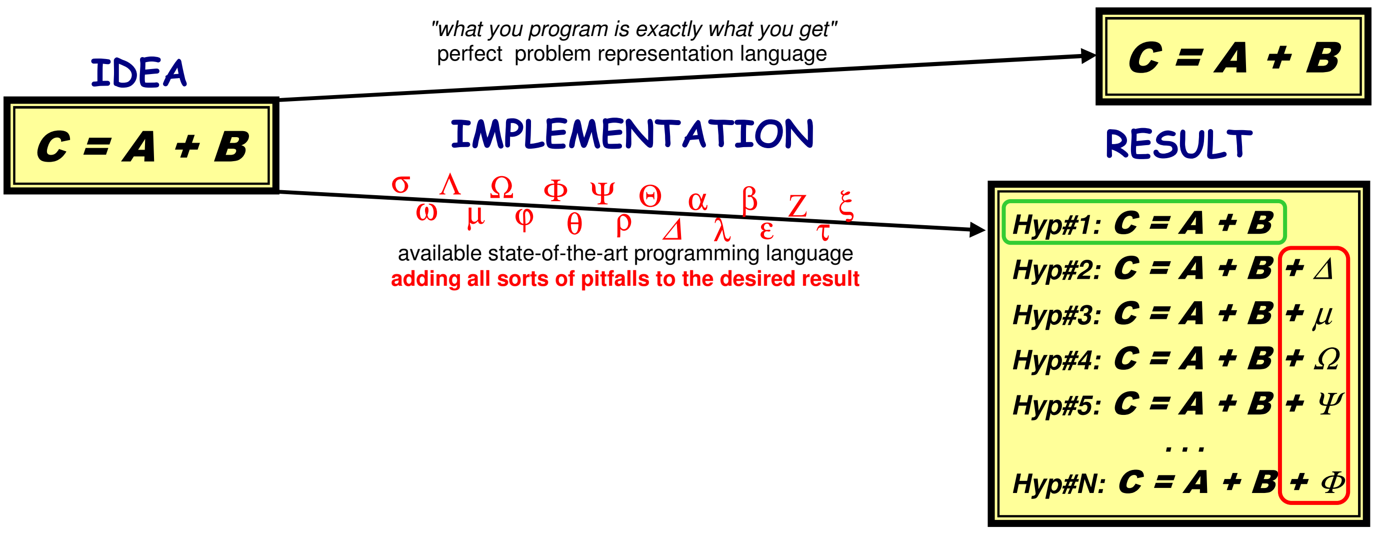

IDEA
C = A + B
"what you program is exactly what you get"
perfect problem representation language
C = A + B
IMPLEMENTATION
RESULT
σ Λ Ω Φ Ψ Θ α β Z ξ ω μ φ θ ρ Δ λ ε τ
available state-of-the-art programming language
adding all sorts of pitfalls to the desired result
Hyp#1: C = A + B
Hyp#2: C = A + B + Δ
Hyp#3: C = A + B + μ
Hyp#4: C = A + B + Ω
Hyp#5: C = A + B + Ψ
. . .
Hyp#N: C = A + B + Φ

Fig. 3.3:11 – Highly potential diversity generation from the basic programming idea/desire to the final result

A fine example of how programmers' ideas may be turned into a mess of possibilities certainly is represented by the "C" language. The multitude, diversity and different ways of doing things can be just overwhelming. Having been developed starting in 1969 [456] [457] and intended to be very near to the underlying Assembly language, and thus to the hardware itself, to allow the programmers a similar level of flexibility and ease the making of the compilers. While being far better to handle than the Assembly it compiles to, in earlier computing history and while it was developed, it was even regarded as a "Portable Assembly Language" or "High-Level Assembly". Although excellent for writing the corresponding compilers for most processors back then, its too high flexibility and especially its extreme diversity, is now regarded more as a problem than as a feature for very complex software systems.

One of the main purposes of this thesis' work is to get back to some of the early ideals of BASIC: *"even a child can learn to program by itself, just by reading the manual and experimenting with the language!".* In-development languages such as [373] are attempts to recover some of the simplicity of basic programming, where one of its supreme premises is very similar to what is directly pursued in this thesis' work: *"Be as simple as possible and adhere to the rule of least surprise in both syntax and implementation"*. Note that this particular implementation of Scribble runs on top of the language Ruby on a Java Virtual Machine, as a language wrapper. Nevertheless, this language goes in the opposite direction of state-of-the-art languages, in the sense that these later ones are just concerned with getting everything bigger, more powerful, more extensible, more flexible, etc. Paradoxically, no-one seems to be really surprised with this path anymore. The same way, it is intended herein to highly structure automotive programming, so that only the exact syntax and flexibility exists, which is strictly needed in that environment. In other words, it is not intended to develop a so-called "general-purpose" language, but a "specific-purpose" one, directed exclusively to control automotive and similar systems.

In other words, this thesis' work will try to get back to some of the very roots of the real purpose of software – making things work at the command of an easy-to-program thought-sequence. Therefore, the language develop herein will be a straightforward sequence of thoughts turned into programming language statements. As already mentioned before, the very basic idea here is: *"Why complicate and translate the desired thoughts into highly complex programs with countless gimmicks and accessory stuff, instead of simply transferring them into being???"*. The quote at the beginning of the next chapter brings current industry tendencies to the very point, while reminding of the opposite direction in this matter. Some other scientists and authors have quite beautifully summed up the meaning of simple things:

"Simplicity is the ultimate sophistication" - Leonardo Da Vinci

"Perfection is attained not when there is nothing left to add, but when there is nothing left to subtract" - Antoine de Saint Exupéry

This "Over-Engineering" pitfall would then be ***PITFALL 0#84***.

3.4 Important System Details

[Emphasis: things that influence everything, things that dictate the course of a complete system]

There are some system details that are usually regarded as not so important but that can, nevertheless, influence the outcome heavily. Although not as regarded as other things such as system architecture itself, these details are further discussed in [Att. 60]. As seen later on, during the explanation of the ideas to be implemented in this work herein, it will be clear why these details can be of major importance after all.

3.5 Combined Software & Hardware considerations

[Emphasis: software and hardware combination importance and their separate paths]

Commercially available software tools had an enormous advance since the advent of the first complete textual high-level programming language FORTRAN in 1956 and its optimizing compiler [280] [281], as an alternative to low-level Assembly language. Visual iconic programming tools such as ASCET, Matlab Simulink and LabVIEW seem to dominate industrial applications' visual programming now, but are still at the beginning of any outlook for massive usage [236]. Visual programming, debugging and validating seems to be an established way of doing things at the high levels of engine/chassis-FDEFs' development.

Commercially available hardware platforms and tools have possibly undergone the least of the evolutions. Although there are some components that have gained intelligence of their own, such as separate ECUs, busses and even some peripherals, the current hardware part of the systems is mostly "centrally-controlled". But due to speed, capacity and safety issues, ABS, ESP, airbag and other critical systems have migrated from the central ECU to peripherals-neighbourhood. Nevertheless, ECUs themselves always display more or less the same mixed internal structure, in that they always have a digital part (where the software resides) and an analog part (where the peripherals' drivers reside). Furthermore, the digital program processing part is never intrinsically adapted to the visual programming interface, in that those used visual languages have always to be translated into "C" code and then compiled and assembled into machine-code language. The higher-level languages describe programs with statements, where each statement if then translated into one or more machine-code instructions. Historically, the higher-level or abstracter the language, the more instructions each of its statements produces at the lowest hardware level, being RISC processors an extreme of this reality. The demise of RISC processing was that it was much easier to build low-level processors with few instructions.

Fig. 3.5:12 illustrates the average complexity investment necessary to process each language statement versus the more or less direct way of processing those statements (smaller or larger overall footprint, respectively). The complexity investment includes instruction counts and all mechanisms involved inside the processor. On the other hand, Fig. 3.5:13 shows an attempt to empirically position main state-of-the-art SW and HW developments into the perspective of this thesis. It can be easily seen that all of them are very interesting and very important, but they all suffer from the very same problem and

limitation: every single SW or HW development is mostly an individual run into a different direction (Fig. 3.5:14), without really integrating a complete system, therefore the "Pseudo-Integration" phrase in the middle, where everything should meet but never really does in a really useful way.

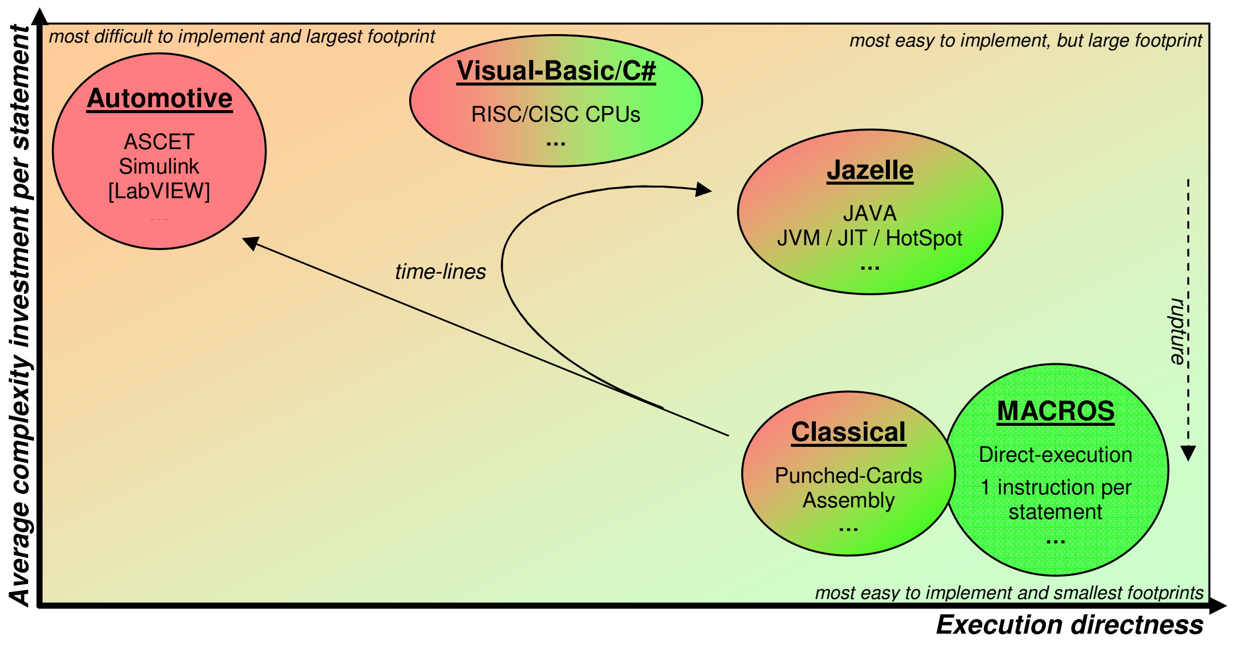
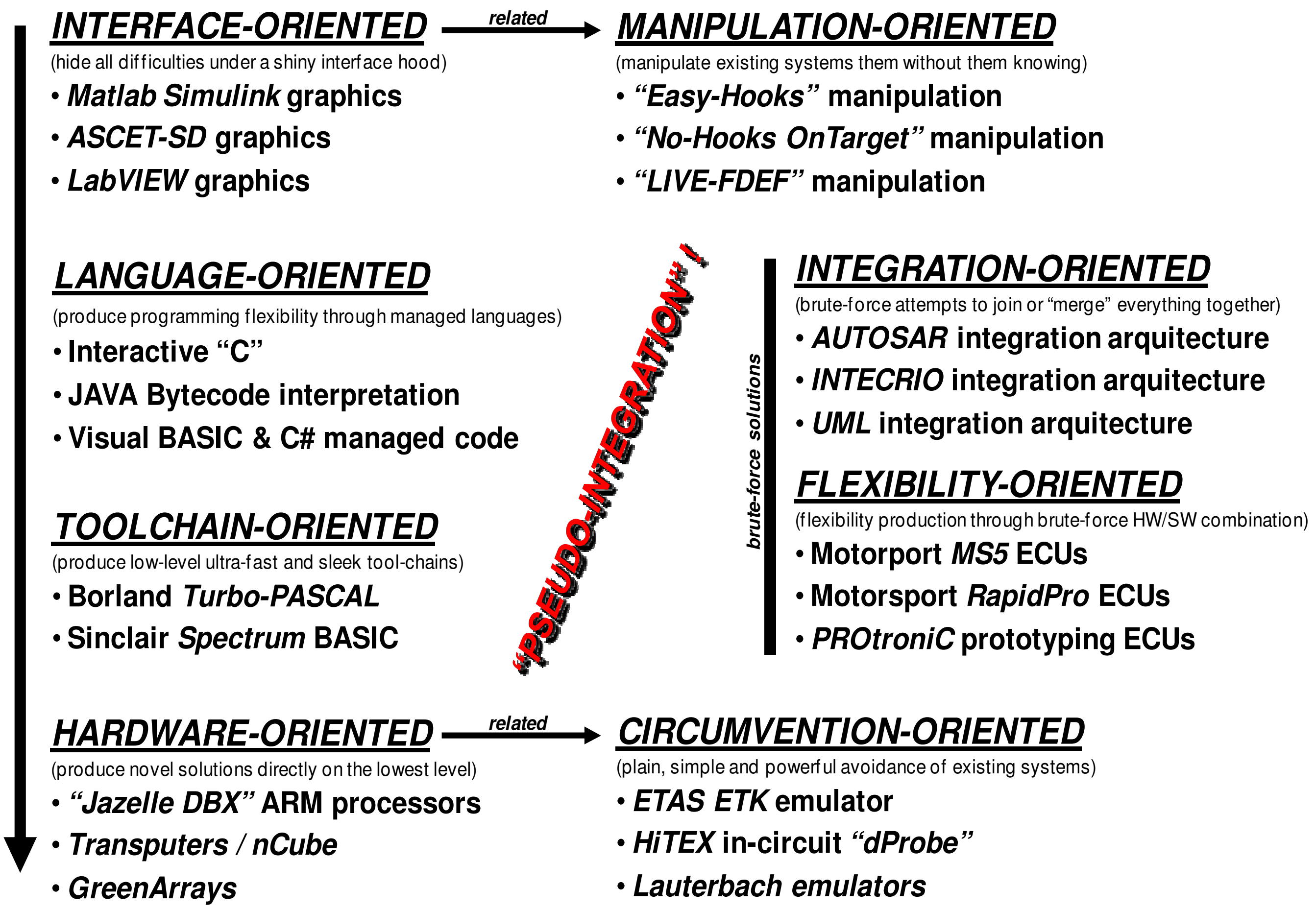

Average complexity investment per statement
most difficult to implement and largest footprint
most easy to implement, but large footprint
most easy to implement and smallest footprints
Execution directness
Automotive
ASCET
Simulink
[LabVIEW]
...
Visual-Basic/C#
RISC/CISC CPUs
...
Jazelle
JAVA
JVM / JIT / HotSpot
...
time-lines
rupture
Classical
Punched-Cards
Assembly
...
MACROS
Direct-execution
1 instruction per statement
...

Fig. 3.5:12 – Comparison of language level with the number of instructions produced for each statement



INTERFACE-ORIENTED *related* → ***MANIPULATION-ORIENTED***

(hide all difficulties under a shiny interface hood)
- ***Matlab Simulink*** **graphics**
- ***ASCET-SD*** **graphics**
- ***LabVIEW*** **graphics**

(manipulate existing systems them without them knowing)
- ***"Easy-Hooks"*** **manipulation**
- ***"No-Hooks OnTarget"*** **manipulation**
- ***"LIVE-FDEF"*** **manipulation**

LANGUAGE-ORIENTED

(produce programming flexibility through managed languages)
- **Interactive "C"**
- **JAVA Bytecode interpretation**
- **Visual BASIC & C# managed code**

TOOLCHAIN-ORIENTED

(produce low-level ultra-fast and sleek tool-chains)
- **Borland *Turbo-PASCAL***
- **Sinclair *Spectrum* BASIC**

"PSEUDO-INTEGRATION" !

brute-force solutions

INTEGRATION-ORIENTED

(brute-force attempts to join or "merge" everything together)
- ***AUTOSAR*** **integration arquitecture**
- ***INTECRIO*** **integration arquitecture**
- ***UML*** **integration arquitecture**

FLEXIBILITY-ORIENTED

(flexibility production through brute-force HW/SW combination)
- **Motorport *MS5* ECUs**
- **Motorsport *RapidPro* ECUs**
- ***PROtroniC*** **prototyping ECUs**

HARDWARE-ORIENTED *related* → ***CIRCUMVENTION-ORIENTED***

(produce novel solutions directly on the lowest level)
- ***"Jazelle DBX"*** **ARM processors**
- ***Transputers / nCube***
- ***GreenArrays***

(plain, simple and powerful avoidance of existing systems)
- ***ETAS ETK*** **emulator**
- ***HiTEX*** **in-circuit *"dProbe"***
- ***Lauterbach emulators***

Fig. 3.5:13 – Empirical positioning of the most interesting state-of-the-art HW/SW in the perspective of this thesis

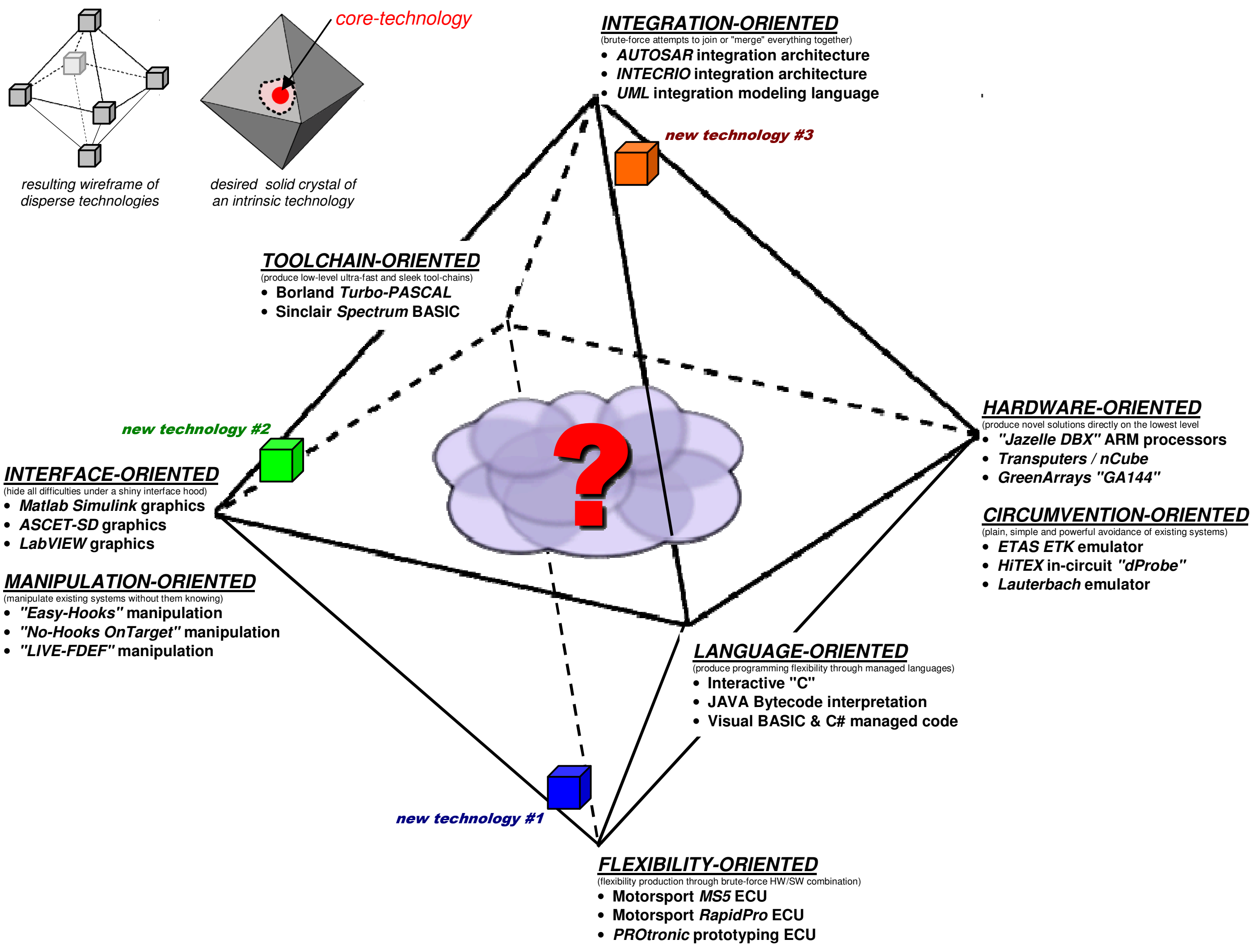

core-technology
resulting wireframe of disperse technologies
desired solid crystal of an intrinsic technology
INTEGRATION-ORIENTED
(brute-force attempts to join or "merge" everything together)
• AUTOSAR integration architecture
• INTECRIO integration architecture
• UML integration modeling language
new technology #3
TOOLCHAIN-ORIENTED
(produce low-level ultra-fast and sleek tool-chains)
• Borland Turbo-PASCAL
• Sinclair Spectrum BASIC
HARDWARE-ORIENTED
(produce novel solutions directly on the lowest level
• "Jazelle DBX" ARM processors
• Transputers / nCube
• GreenArrays "GA144"
CIRCUMVENTION-ORIENTED
(plain, simple and powerful avoidance of existing systems)
• ETAS ETK emulator
• HiTEX in-circuit "dProbe"
• Lauterbach emulator
new technology #2
INTERFACE-ORIENTED
(hide all difficulties under a shiny interface hood)
• Matlab Simulink graphics
• ASCET-SD graphics
• LabVIEW graphics
MANIPULATION-ORIENTED
(manipulate existing systems without them knowing)
• "Easy-Hooks" manipulation
• "No-Hooks OnTarget" manipulation
• "LIVE-FDEF" manipulation
LANGUAGE-ORIENTED
(produce programming flexibility through managed languages)
• Interactive "C"
• JAVA Bytecode interpretation
• Visual BASIC & C# managed code
new technology #1
FLEXIBILITY-ORIENTED
(flexibility production through brute-force HW/SW combination)
• Motorsport MS5 ECU
• Motorsport RapidPro ECU
• PROtronic prototyping ECU

Fig. 3.5:14 – "Technology Polygon" - the same developments as above, but now illustrating the diverse directions

Nevertheless, there are some examples of joining two or more of these peak-technologies into something useful, such as the MS5 ECU [10] combining Matlab Simulink with highly flexible and powerful hardware, but incurring into large manpower, documentation and maintenance costs. In reality, there is no SW/HW combining most of those technologies (centre of Fig. 3.5:14), without all the tradeoffs and costs involved, because it would be too expensive, such as the existing attempts to merge technologies usually are. This further points into the "clean slate" approach going to be taken in the work herein, to attempt to not just develop a costly integration/merge, but to developing something innovative in the sense of intrinsic integrating the features of the various herein presented state-of-the-art technologies.

This intrinsic integration is the main feature of this thesis: a new development crystallizing all those separate features from the centre, instead of combining existing satellite components by brute-force into something they will never be well adapted to. Those satellite components and technologies are already too far apart to be easily integrated into something central. That's the main reason why new developments usually grab themselves a spot on the outside of this "technology polygon" (illustrated by the coloured cubes in Fig. 3.5:14), instead of existing inside that polygon by blending more of the vertice technologies.

This has lead to the already long and glorious age of software development, while hardware only essentially had the merit of getting faster over time. Even though the software part changed the most since the 1950s with the first FORTRAN compilers, it was not until 1986 that the first commercially available and used visual language appeared (LabVIEW [29]). Nevertheless, this language is today below place #50 and below 0,2% usage in the "TIOBE Programming Community Index" [236], while the more recent Matlab Simulink passed 0,66% of usage on the 16th place. This shows how better or "human-sensitive" languages appeared decades after the hardware itself, mainly because hardware was not at all suited (and still isn't!) to be programmed through those languages and thus made developers stick with the painful path of devising growingly complex textual languages. Since the existing hardware mainly only understands "machine-code" that is not very far from the legacy punched-cards concept, everything that mounts on that, must be and cannot be anything more than translators and converters, thus always "pulling down" every effort to artificially "hide" or "bury" that unavoidable reality.

On the other hand, if hardware could be developed that really closes the gap between users and the equipments they want to program and control, but without the known complexity and effort involved in current state-of-the-art systems, there would finally be a much better chance of doing things in a much more efficient way. This efficiency would be immediately noted at the user handling as well as at the developer effort levels. This hardware would then finally allow directly executing visual operations. This directness/translatorless way of doing things, all concentrated on the central-role playing *“Macros-Sequences”* will be further detailed later on. But it can already be revealed that these visual elements will rely on binary patterns (the *“Macros”* themselves), the only entities really understandable by the hardware implemented herein. The only thing that would be still necessary would be some sort of graphical engine that would show those binary patterns in some graphical form for humans to see and easily understand, thus effectively closing the gap between user and equipment in a most simple and efficient way.

This conceptually simple idea would this close the gap in a single leap, virtually eliminating almost all forms of existing complexity. Efforts in a similar direction have been made through the decades [102] [89] but they always turned out to be some sort of "evolutionary adaptation" to the persisting reality of raw machine-code talking hardware. Furthermore, [89] just transferred the software complexity into hardware, with the sole purpose of acceleration, as all posterior hardware mechanisms have attempted. Fig. 3.5:15 illustrates progress over the past decades by comparing three totally different realities and their internal structures.

Past and present structures' hardware architectures differentiate only in the way of processing basically the same "machine-instructions" and in that processing speed. User-interfaces have "raised" in level while the hardware paradigm basically maintained the same basic processing structures on their level, adding more and more extra "speed-enhancing" mechanisms. Inversely, prices keep decreasing. This steadily and increasing speed/price ratio kept this paradigm going for decades. This did lead to six orders of magnitude in performance-price gains over about 30 years [244]. Again, as predicted in [12], computers basically retained their strict low-level raw "evolving calculating/data-manipulating" nature. On the other hand, software has been changing a lot, distancing itself noticeably from the underlying hardware language level. User-interfaces must get lower themselves to a common central denominator, whereas hardware must be able to grasp that denominator by likewise elevating itself, thus making this "single-leap" architecture herein possible. Past, "double"-present and future systems can be grouped in the following manner:

- legacy low-level programming methods with "machine-code"-containing punched-cards, hardware architectures that directly understood them, and a direct path from user to hardware.
- modern high-level indirect programming methods with languages translated and optimized by several layers, hardware architectures that basically still just understand "machine-code", and a relatively wide and opaque path from user to hardware.
- state-of-the-art highest-level indirect visual programming methods with languages translated, optimized and adapted by several layers, hardware architectures that basically still just understand "machine-code", and a bloated and opaque path from user to hardware.
- proposed highest-level programming method with visually-enabled language, directly understood by the hardware architecture, accomplishing every desired feature within a single leap and connecting the user to the hardware through the lean "Macros-Sequence" main stem.

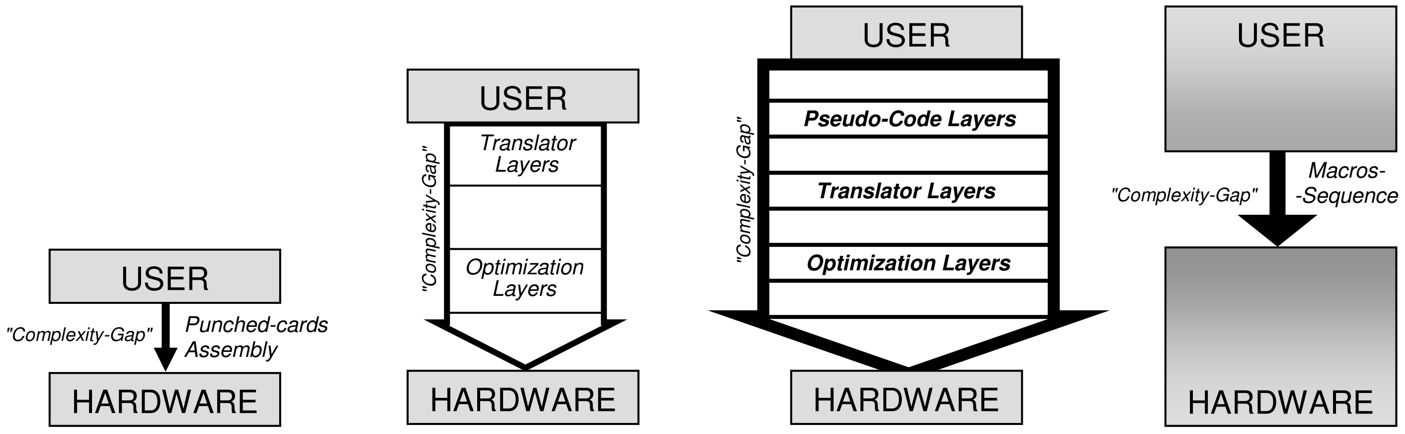


Fig. 3.5:15 – Left: legacy lowest-level DIRECT SINGLE-STEP programming with punched-cards
Middle-left: modern high-level INDIRECT MULTI-STEP programming with translation layers
Middle-right: state-of-the-art highest-level INDIRECT MULTI-STEP programming with translation layers
Right: proposed highest-level SINGLE-STEP DIRECT programming with software/hardware synergy

This gap between the top-most and bottom-most layers of software and hardware, respectively, is called "Complexity Gap" [370]. At the top-most layer there is the so-called "macro-process", "whole program level" or problem domain, while at the bottom-most layer there is the so-called "micro-process", "machine-code level" or solution domain. The "Complexity Gap" lies exactly in-between them both. Furthermore, as stated in [370] and is will turn out to be clear later on, system developers really spend their time inside this "Complexity Gap". Again, this thesis' work will try to close this gap as an unnecessary global pitfall, allowing both developers and users to regain an intimate sense of useful proximity within the software/hardware system. The most important details to retain from this "Complexity Gap" issue clearly pointed out in [370] are:

- The "Complexity Gap" is everything that lies between the top-most "macro-processes" or "problem domain" and the bottom-most "micro-processes" or "solution domain" (see Fig. 3.5:16-A). This gap presents a kind of adaptation space where many layers exist, which transforms problems into solutions. With tool interfaces constantly evolving and distancing themselves from the underlying solution technology, successfully representing the problem has become a relatively easy job, whereas translating it into the solution has become the real hurdle/pitfall in every respect for tool developers. The "Complexity Gap" is where all the hard work is done or, said in another way, it is where everything is done the hard way. Developers spend well over 80% of their time inside the gap, with less than 10% dedicated to the application at hand (problem) and less than 10% to the available hardware platform (solution). Auto-code generators did not solve this problem at all, since they are just another layer that interfaces between visual program design and underlying classical "C" language based tools. Nor did "hardware encapsulations", which also add another layer to the already existing stack.

- There are essentially two kinds of problems: linear problems and chaotic problems. Linear problems involve a single level, with few or no sub-problems, whereas chaotic problems involve many interdependent levels and sub-problems (see Fig. 3.5:16-B). A linear problem example is cake making, in that there are certain ingredients, a certain way of combining them to make an edible cake and very little variation possibilities given the desired outcome and used ingredients. On the other hand, general software programs or automotive programs contain so many sub-components, inter-relations among them and different possibilities of doing things, that they are clearly chaotic. Moreover, the main difference between linear and chaotic problems is the number of interrelated levels, not the size of the project. Another fine example if the difference of the gap between a craftsman and an assembly-line worker: while the first one accomplishes its work at many disparate levels, the assembly-line worker just has to follow very strict rules with a desirably narrow dispersion (see Fig. 3.5:16-C).
- It is argued that the gap can be reduced by lowering the problem domain through more structure and by raising the solution domain through more support (see again Fig. 3.5:16-D). Structure means feeding underlying layers the right way, so that they will process the posed problems in a more efficient and adapted way. Support means having a better interface to hardware through better methodologies. Narrowing the gap could also be accomplished through improving the skills that will ultimately match the problem and solution domains.

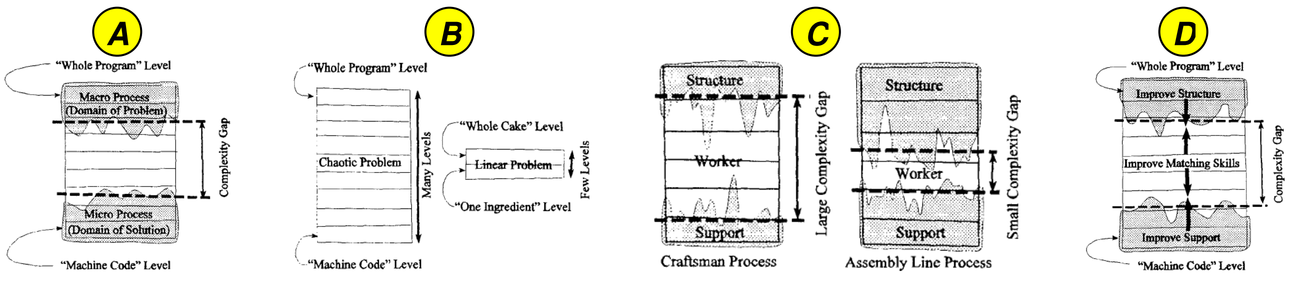


Fig. 3.5:16 – The various ways to look onto the "Complexity Gap" problem (source: [370])

In the last point of Fig. 3.5:16 it was argued within [370] that narrowing the gap could also be accomplished through improving the skills that will ultimately match the problem and solution domains. But unfortunately this clearly does not address the primordial issue that ultimately generates this gap: the fact of having two radically "non-similar" domains where one comes in graphical form and the other in machine-code form. [Att. 14] shows more details and situations concerning this very important “Complexity Gap” concept and other highly related concepts, such as "Language-Oriented Programming" (LOP) [378] [370], the “Knowledge Gap” [397] and “Cross-Layer Design” [121]. As seen there, practically every complexity here boils down to this "Complexity Gap", that is, everything that places between the task/solution and the very execution of the solution program. Thus, it is ***PITFALL C#85*** and arises from the fact that hardware has not changed its basic initial basic "calculator nature" much, as well as its "calculator machine-code" nature, both because it was easier to advance the programming languages that would take care of the adaptation to that hardware through a large pile of ever-growing translator layers and APIs.

The above-mentioned "Cross-Layer Design" approach is one of the radical attempts and needs, in this case in wireless networking [121] to get high-level action more directly into lower-level mechanisms [274] [275]. It represents a very clear signs that software got to a difficult to recover point. The "Gap" is not even empty. It is filled with the most complex part, the translator layers amalgam, therefore being called "Complexity Gap". This "Gap" filling has nothing to do with the reality of the system to program, it only exists because of the distance that grew between HW and corresponding SW. Thus, this "Gap" is something that in theory is 100% overhead that should be attacked and eliminated altogether. That is exactly what this thesis' work is going to aim at, by creating a development system where everything crystallizes from-the-middle-out, instead of "filling-in the gap", between the desired HW and the desired SW.

Hardware manufacturers only had to make it faster and larger. Now that the software part has become an almost out-of-control "intricate web", new hardware developments envisioning to narrow down this "Complexity Gap" are welcome and still in their "baby steps". Their acceptance is still another problem and some technologies never get very far. Maybe a way of getting rid of this recurring problem, is to finally go against the established commercial reality of computer architectures and simply give it a try to systems that would effectively endorse the right-most idea in Fig. 3.5:15. One of such serious attempts has been done within the *"SYMBOL"* language project back in the 60s [136]. This project showed that such architectures are clearly possible and that the historically established commercial computer architectures essentially win by the numbers. Real hardware advancement towards the languages it should execute would also reduce the pitfall herein referred as "Tool Schizophrenia", where even the most advanced user-interface software behaves strangely more than would be desirable, mainly because of their complex and long internal structure to handle the herein mentioned "Complexity Gap". User-interfaces get themselves "sick" because of so much complexity and interference everywhere. User-interfaces could eventually become much simpler if just this "Complexity Gap" could be almost closed or completely eliminated, thereby reducing the length of the path going from the "Solution" to the HW and thus the possible things that can go wrong in-between. The work developed in this thesis will prove this claim.

From here it seems to be progressively clear that this "Complexity Gap" is where most problems of current state-of-the-art software development systems reside. Again, as brilliantly and early recognized in [61], one may ask why computers are not just built to "directly process" programs and to do what the programmer intended to do. This gap kept growing until a point of "virtually no possible return", turning current computing systems into highly "inadequate" processing entities. As all images in Fig. 3.5:16 show, it can be clearly seen that this gap is essentially caused by taking both ends (problem and solution domains) and try to force their link to work out as an approximation or "blending" through the middle. Reducing complexity by trying to narrow down this gap, is the aim of this thesis' work. This may be done by going exactly the way around, that is, by taking the middle part as the essential entity (future *"Macros-Sequence"*) and by "propagating" its presence, effect and power throughout all sides reaching the problem and solution domains. Earlier Fig. 3.5:15 showed the last right-most proposition where a "single-step direct programming" is mentioned. This illustration is not to be confused with lowering the problem domain and with raising the solution domain. Quite on the contrary, that illustration simply means that the whole problem and solution domains are themselves incomparably more "similar" compared to each other.

Last but not least, a last consideration related to the need to maintain current software packages, after they have been first released. The main issue of the process of software maintenance is explained in [460] and boils down to the cost/time needed to accomplish that critical task. As already stated in earlier Fig. [Att. 59]:210, software maintenance is reaching prohibitive levels, which get worse with time passing by a long-lived software application. At the very end where some chaos already governs actions, most of the budget is spent maintaining the software (in more colloquial words: keeping it running). Fig. 3.5:17 illustrates what happens when maintenance bloating consumes almost all available resources (top) and when a better maintainable software allows for cost-reduction or resources allocation to innovation tasks. It is well known and studied [143] [160] [161] [162] that current automotive software follows the top illustration, with huge manpower and allocation of resources to "keep all systems up and running". Just imagine how software systems could evolve and innovate if those 80% of maintenance costs could all be allocated to innovation efforts. The same goes for automotive software, maybe more in a cost-effectiveness direction, although it is also known that those bloated automotive software costs are readily hidden inside the large ECU selling numbers (in the tens of million units).

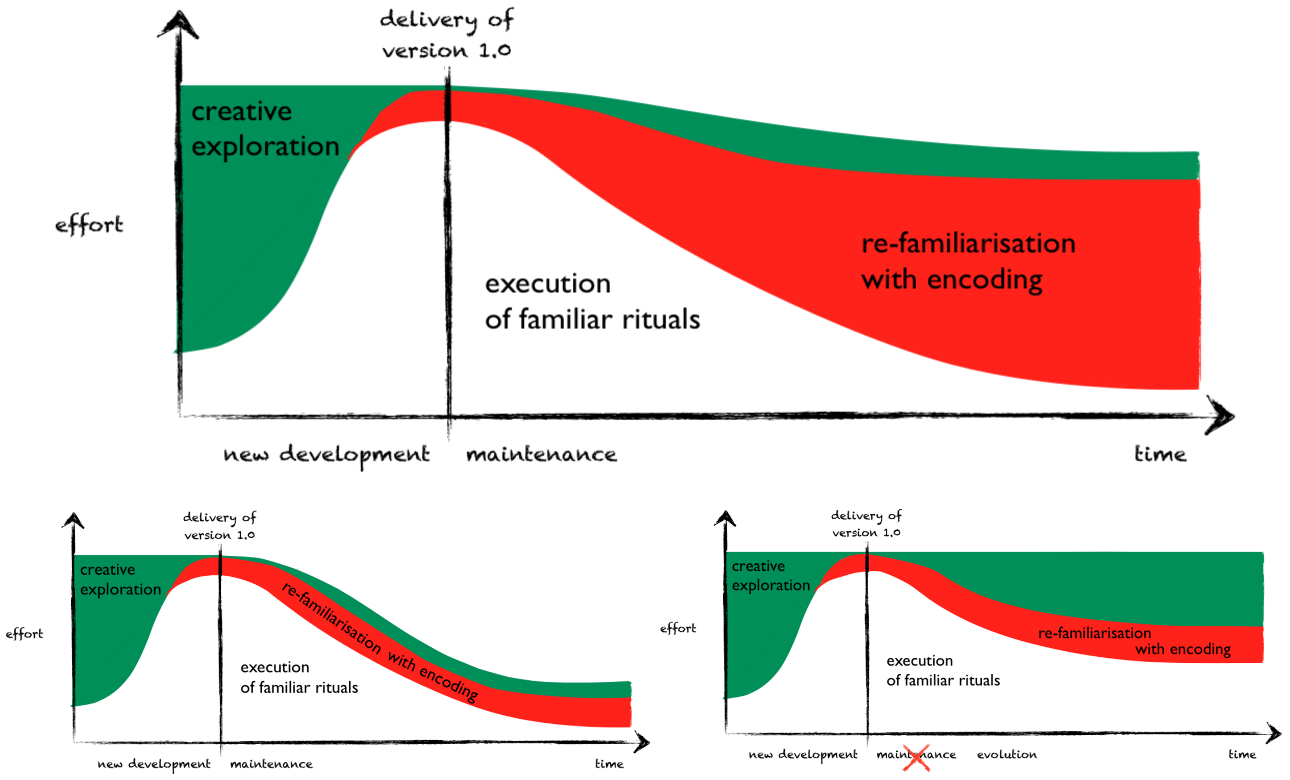


Fig. 3.5:17 – Top: bloating maintenance; Bottom: little maintenance ⇒ cost-reduction & innovation (source: [460]).

3.5.1 Final remarks

[Emphasis: too many pitfalls, limitations and challenges]

There are over 80 potential trouble sources (pitfalls) and handling problems present in most of today's automotive software development systems and hardware deployment platforms. Many of them directly derive from over-evolution. With over-engineering at the top of them, being able to cope with all of these high-grained pitfalls has become a simple matter of historical habituation and also of crucial survival of current systems. The programmer must quite often deal with all these, while drawing his attention away from the really important task. Intricate problems or errors often means days if not weeks of debugging and searching. Programmers even have to build-in "workaround code" to have the product running on the client's premises, while the causing problem is eliminated much later on after very long search efforts. Wrong information also crawls inside large manuals and complex CPUs, causing painful debugging tasks to end with the knowledge that the CPU itself has a hard-bug inside. At the extreme end, errors remain indefinitely inside ECUs, "hidden" by strategically added "workaround code". This further demonstrates that there really is a big gap between the user and the hardware itself. No new and leaner interface will correct that, since this gap is a historically grown artifact that is deeply rooted inside all current development environments.

Unfortunately, new developments in hardware and software, being analysis [290], standardization [67] and simulation [243] tools mostly addressed with classic tools and classic ways of doing everything, reflect the reality that all existing pitfalls will remain basically untouched. As a global and categorized wrap-around, most of the herein identified pitfalls can be graphically viewed distributed among the various macro-tasks in automotive systems in Fig. 3.5:18 and, those same pitfalls that cause context and diversity are depicted in Fig. 3.5:19. From all these pitfalls, "Over-Engineering" emerges as the main conceptual result.

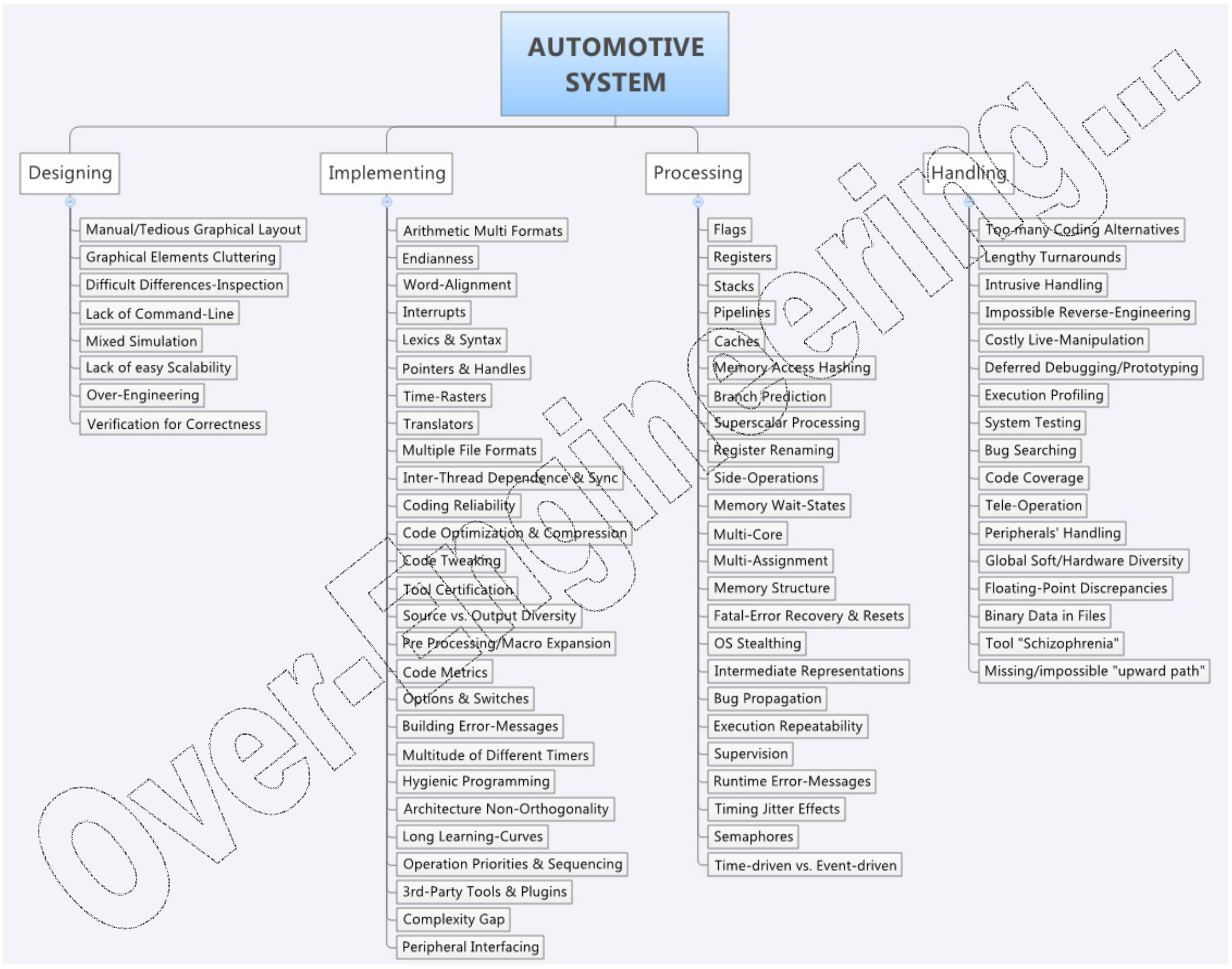


Fig. 3.5:18 – Operational view of all pitfalls identified herein

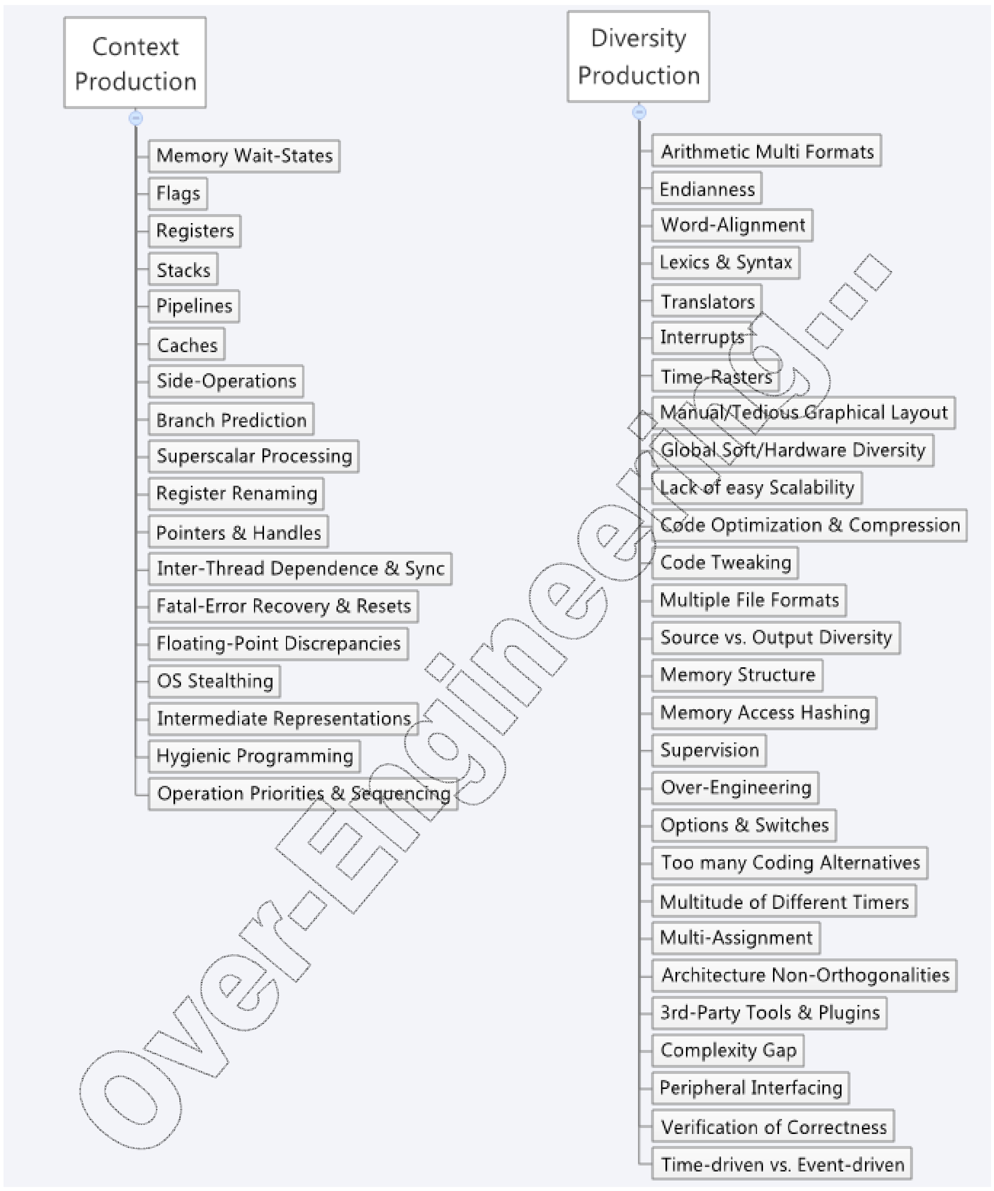


Fig. 3.5:19 – Yet another view of the pitfalls, context and diversity causing ones

Several attempts such as [152] to go round some of the pitfalls, have been made, but none really addressing the deepest pitfalls, such as those present in the very nature of the therein still needed big and complex tool-chains. Furthermore, analysis [290], standardization [67] and simulation [243] tools certainly represent the finest in technology, but these just allow the developer to continue working with systems and programming methods that ought to be revised themselves. Fig. 3.5:20 illustrates the difference in the web of inter-relations between older (top) and contemporary (bottom) systems in non-automotive applications. It is quite clear that some 20-30 years ago it was relatively easy for the developer to understand and keep everything under control. On the other hand, currently it is not the developer that has it under control, but some complex and eventually certified 3rd-party tool that grew over the critical need to keep everything working*. Nevertheless, there are now people [345] starting to appear and to ask

* *This is quite similar to keeping an old and collapse-endangered building intact by holding it from all sides, because rebuilding it would be too expensive and of unsure outcome, while also keeping the danger of neighboring buildings collapsing along with it. Everyone knows that this situation will eventually have an end and that something drastic will eventually have do be done about it, but nobody really wants to take the huge risk of really dealing with the situation.*

themselves how it got to this situation and why one should continue to code over 100 LoC for something that can be done in just 5 LoC. Just because the majority goes down the "over-complication way", it does not mean that everyone has to do it that way. [390] even points out that either people get control of the systems or be controlled by them.

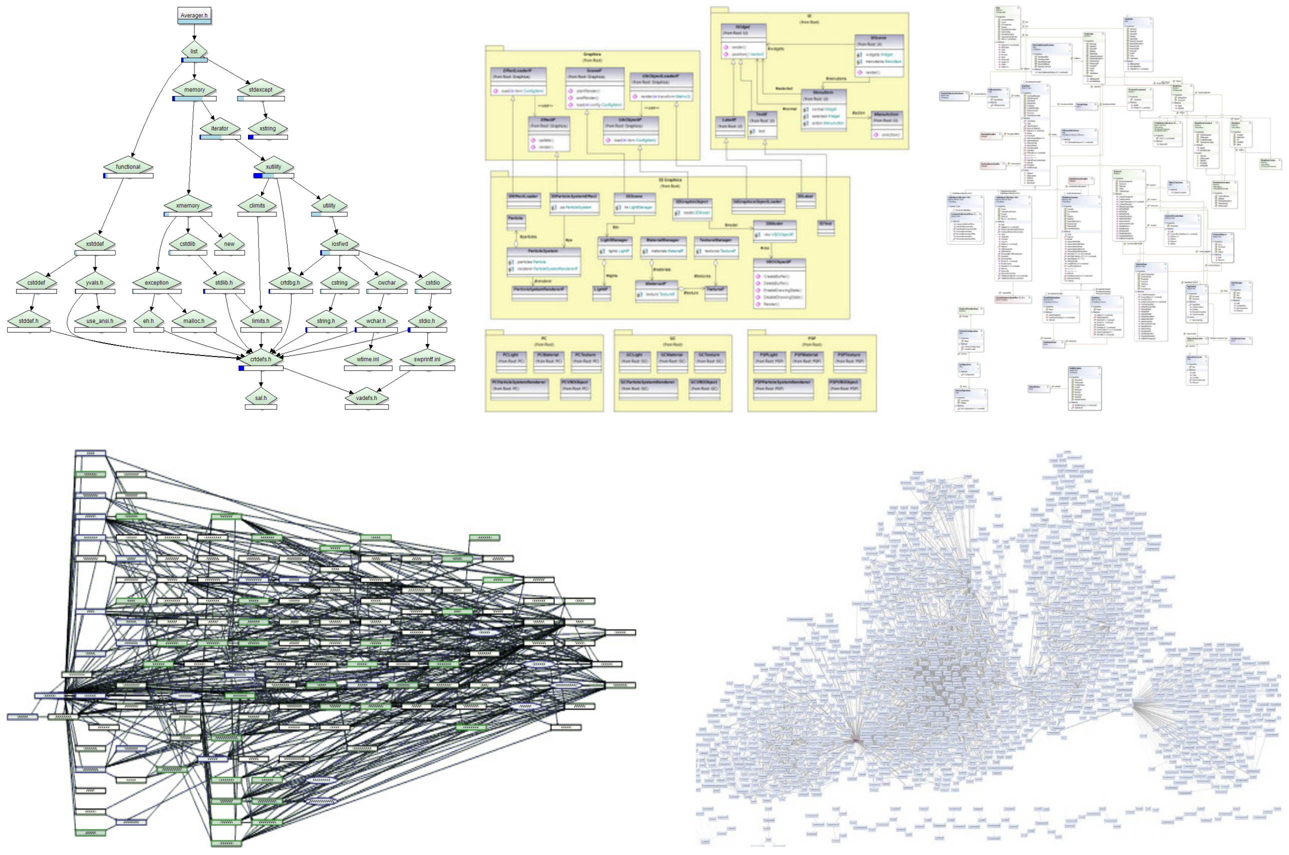


Fig. 3.5:20 – Comparison between some legacy (top) and current (bottom left: 70.000 LoC and 150 files; bottom right: 1.700 classes to replace a legacy system which only had 100 classes) system elements' inter-relations that are one of today's curses in software while waking up all kinds of pitfalls (source: GOOGLE IMAGES)

Automotive systems are getting not much different from the above complexity. While in the 80s a typical automotive system contained only 1-2 ECUs, a "Function Manual" with about 50 functionalities, about 48Kbyte of firmware code and about 50.000 LoC, in 2010 some bloated systems already passed the 70 ECUs, 3000 functionalities, 500MByte of firmware code and 100.000.000 LoC barriers! Not to mention the extensive and complex networking efforts that additionally fall over such systems: over 4Km/60Kg of cabling with a total of over 2.000 interfaces. Evolutionary methods and the additives that keep everything surprisingly still working with the main premises as follows (the sequence is intended), create multi-layered like systems where nobody dares to touch the core of them:

- "Never re-engineer a working system[*]; just tame it and evolve it!" or "if it still does the job, why re-invent it?[†]; just adapt it!". This greatly explains why current systems and even new ECUs still partially contain legacy code no-one understands any more and why some things are still primitively done: no-one risks doing something better because it still works anyway.
- Just add various certified tools and certification/standardization procedures over this, virtually all state-of-the-art and currently used software development and hardware deployment systems will seem to be brand-new. This greatly explains why these tools and procedures are having so much success nowadays: no-one risks going against this industry's "evolutionary" tanker.

[*] *Adapted from the English aphorism "never change a running system".*
[†] *Adapted from the English aphorism "if it ain't broke, don't fix it" or "leave well enough alone".*

- While it is clear that just a few developers eventually cannot hold this entire package under strict control any more, a hundred-fold in manpower certainly overcomes this problem as well, which then represents the outermost "brute-force" shell around these systems. This greatly explains why a thousand people are needed to tame something apparently simple.

It seems clear that evolution drove automotive systems almost as much as it could. The big challenge of really addressing most pitfalls, especially the hard-core ones related to complexity and handicaps, while also maintaining the performance needs of those systems, is that a totally new idea has to emerge. This emerging idea may well be data-flow based graphical processing entities containing everything they need to process, thereby being fully isolated from everything else aside from the inter-connections, including isolation most of the herein listed pitfalls. This strongly resembles the typical neuronal circuitry in the brain, where the synapses are the only inter-connection between neurons and where the blood stream is the nourishment (power). See Fig. 3.5:21 shows the similarity between neural networks and data-flow diagrams. Eliminating all forms and sources of interferences to the outside or from the outside, would be a big step forward, when compared to current systems.

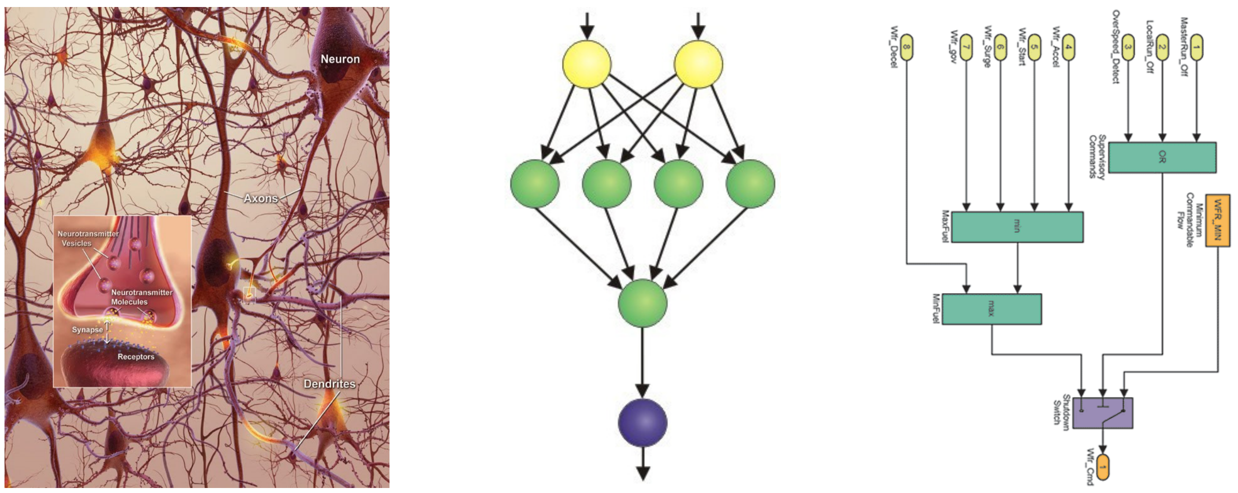


Fig. 3.5:21 – Left: from natural neural networks; Centre: similar data-flow diagrams, Right: Simulink diagram

Finally, one might think that mechanisms such as the ones used inside products such as *"ECU Interface Manager"* [417], *"No-Hooks OnTarget"* [300] and *"EHOOKS"* [223] are very useful, since not requiring source-code changes whatsoever. And in fact they are very useful and easy to use but, again, they hide huge pitfalls under their hood. To operate in the desired way, these packages took lots of time to develop, possess very complex internal processes and work through extremely high intrusion by changing the machine-code itself. All three packages thus represent an extreme form of reverse-engineering since they apply the solution after the complete program build process having finished. This mechanism thus represents "going all the way down just to get back up again" by mixing it with additional information from the source-level (middle of Fig. 3.5:22), just because the direct path is currently impossible (right of Fig. 3.5:22). An even worse alternative would be to request the desired changes and bypasses directly to the manufacturer (left of Fig. 3.5:22), yielding inevitably very long waiting times for the customer. The direct path or something that directly allows instant changes at source-level is what is desired in this work herein.

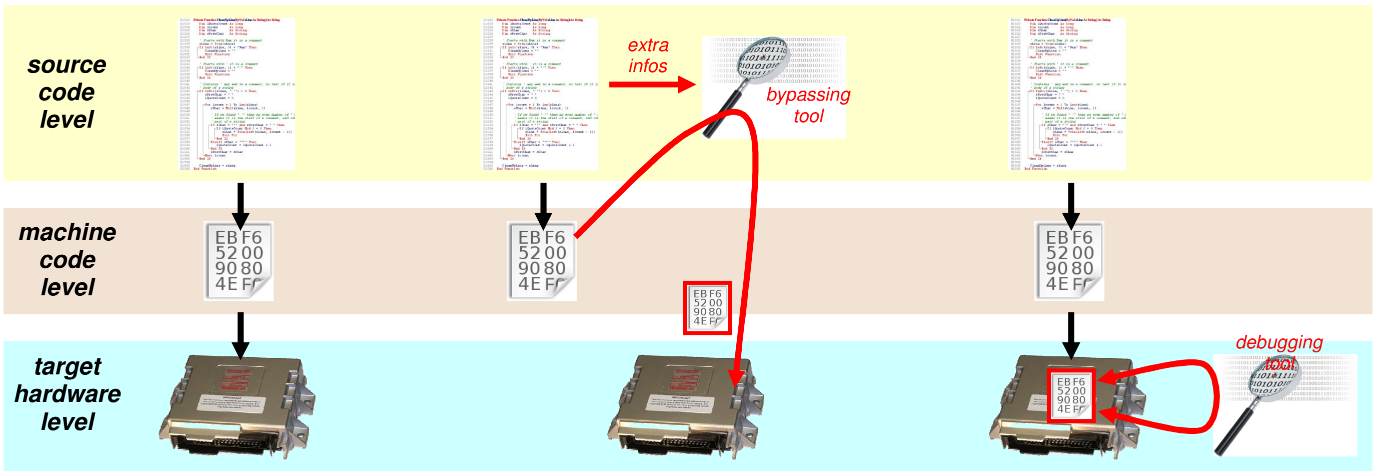


Fig. 3.5:22 – Left: standard classical build path; Middle: existing bypassing path; Right: hypothetical direct path

3.6 Conclusions

[Emphasis: software and hardware must grow together]

All the above considerations mean that the software/hardware relationship is not an intimate one anymore, as it should always be in a technological marriage with over 50 years, but turned itself into one with two entities living in different rooms of the same house. An automobile driver desires to get into a car and expect it to work as such, nothing more and nothing less. If developers are kept trying to hold everything together during an appreciable amount of time, they are kept away from developing the car as such and problems arise. As for the driver, developers should be kept developing and not mostly holding millions of threads so that the whole car does not fall apart. Currently and during the last decades, it seems evident that 99% of time is spent with development of more and more software layers over existing ones, driving user-actions progressively farther away from the hardware that must obey to those actions. While current tools exhibit externally observed reality-near interfaces and features, those intermediate layers just try to maintain the "thin and intricate path" connected between two internally totally separated realities. However, this artificial and hidden "thin and intricate path" has no noticeable meaning whatsoever in the user/hardware relation, so that similarly its elimination and substitution by a more direct path would not even be accounted for. It is as if the meaning of "software" (mouldable component) has itself been driven too far off from reality and where the "hardware" (non-mouldable component) was kept under control to avoid it getting too intelligent or to avoid the danger of it getting in control. Although this seems to be true, the real truth lies in the fact that software has taken over and has taken control of everything, including the way developers have to go to get new things done, namely through a dependence relative to the established and historically grown internal layered mega-structures. The situation is clearly a very complicated one, requiring immediate action to invert it, since the following modified/adapted laws apply well to it:

- "Software is getting slower more rapidly than hardware becomes faster" – *Wirth's Law* [245].
- "One is tended to follow on from a relatively small, elegant, successful system, by designing the successor as an elephantine, feature-laden monstrosity" – *Second-System Effect* [246].
- "Roughly 80% of all benefits come from 20% of all features" – adapted *Pareto Principle* [247].
- "A program tends to expand to match the supplied computing capacity"– *Parkinson's Law* [248].
- "Adding more abstraction layers to an already incomprehensible layered system, turns it into an even more incomprehensible system" – adapted *Brooks' Law* [250].
- "Technological progress increasing the efficiency with which hardware is used, tends to increase (rather than decrease) the consumption of that hardware's increased capacity by even more capacity-hungry software" – adapted *Jevons Paradox* [249].

It so seems appropriate at this time to allow hardware to receive structures that finally allow dismantling somewhat of the enormous software aura around it.

This thesis' work goes exactly in the direction of demonstrating the theoretical and experimental possibility of closing the user/equipment relationship gap in a single leap. It turns out to be clear that an attempt has to be made, whereby hardware and software grow up together as a coherent unit. But to accomplish this successfully, the best way might just be to have a common ground or basis to start from. This work will also demonstrate that it is possible to eliminate most mentioned pitfalls by just keeping in mind that the developed system should mostly be "correct and limitless by design". All this will be further analysed and developed in Chapters 4/5. Chapter 6 shows experimental results in a real industrial environment, where a reduced automotive system is showed working. That is the herein presented main proof of the above concept.

As a final note, this thesis' work aims to return to the original and desired purpose of computational hardware: do exactly what the user wants it to do, through nothing more than precisely the most direct interface possible, while maintaining the user at his level of mental development. This reclaims lost handling efficiency and even lost application efficacy because less potential pitfalls exist in a leaner and more direct system.

Overall historically lost system-purpose could thus be greatly reclaimed. This could be seen under the light of [244] where "accidental complexity" reduces to a minimum, while giving full priority to tool integration and "essentially complex" problem-solving. According to the arguments contained in [244], there really might not be a "silver bullet" for the software complexity problem, but this thesis' work sure hopes to demonstrate that at least a "steel bullet" can really make a huge difference!

Current systems continue their quite unstoppable evolutionary methods. Some quite disturbing but interesting quotes apply here:

"If I asked my customers what they want, they simply would have said ‘a faster horse!’." - Henry Ford

"Science in its ideology sees itself as doing a fearless exploration of the unknown. Most of the time it is a fearful exploration of the almost known" - Rupert Sheldrake

"It is not because things being difficult that we do not dare to try them, but because we do not dare to try them that they are perceived as difficult!" - Seneca

"If at first the idea is not absurd, then there is no hope for it" - Einstein!"

"Take risks. Ask big questions. Don't be afraid to make mistakes; if you don't make mistakes, you're not reaching far enough."
- David Packard

- "The human being is an animal of habits and deals very badly with change. He only changes when the pain produced by those habits is greater than the change itself." - Jamie Oliver

- "Every truth passes through three stages. First, it is ridiculed. Second, it is violently opposed. Third, it is accepted as being self-evident."- Arthur Schopenhauer

- "The least questioned assumptions are often the most questionable." - Paul Broca

Page intentionally left blank

Page intentionally left blank

"Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius and a lot of courage to move in the opposite direction."
– Albert Einstein

CHAPTER 4

Ideas & Solutions to enhance Systems' Handling Efficiency

4.1 Background History

[Emphasis: good reasons for changing a working system]

This chapter will finally present some solutions to pitfalls and general problems identified in the previous chapter. These solutions are the by-product of some years of reasoning about them inside an automotive environment and workplace. This work herein however, clearly attempts to present a global SW and HW combination that addresses most main problems and that constitutes a one-in-all solution. Although this solution is naturally aimed at the automotive motorsport scene, it is not limited to that field of activity. Most details of this solution will be presented as generally applicable to other fields and represent individual proof-of-concepts that will be then integrated in the final global SW/HW solution.

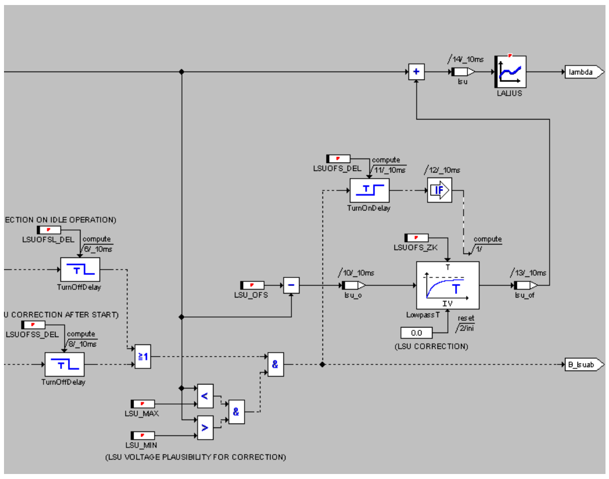
First, the motivational history started around currently used tool-chains at work, where it was perfectly noticeable that many problems exist. Many things and procedures should desirably be made in a completely different way, if much higher levels of *"Handling Efficiency"* would be to be achieved. Also, some "well-designed products" showed that it was still possible to enhance many of those problem-spots, often without any too large efforts to change them. Sometimes, very simple solutions take much time, solely due to the fact that not many people believe they are at all possible.

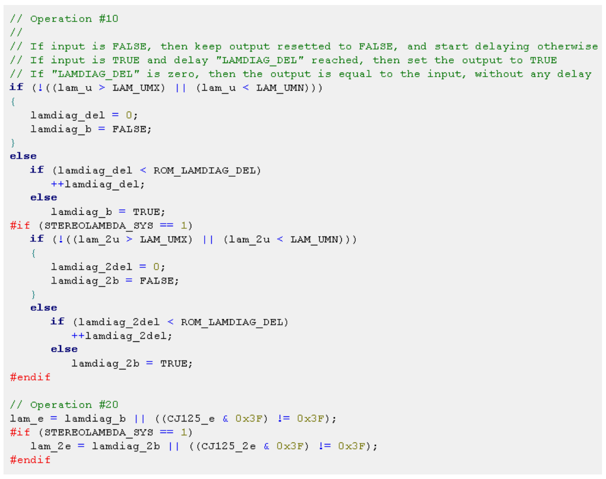
4.1.1 Using currently "Working Systems"

[Emphasis: "never change a working system" taken to the extreme of never changing it]

The author worked at Bosch Motorsport in Markgröningen, Germany, for almost 4 years and still work for them up to today since 2001. Having collected a very large amount of information about how things were done in the global Motorsport scene in general (not only at Bosch but also at the competition), The author realized that there were many points which could be addressed by simply rethinking the way of doing things that are generally done always in the same historically stiff manner. Up to nowadays, systems' limitations and pitfalls are "solved" with great manpower and time resources. The first of all ideas that emerged while working in this complex scene, was to simplify or even to somehow eliminate all that overhead complexity that arises during automotive FDEF designing/programming. This included related problems and limitations that arise when something goes wrong with the usually very complex underlying machine-code producing structures for the automotive ECU.

First of all, there is the task of designing/programming the automotive FDEFs themselves, reducing itself to the battle between text-based and gradually more visual interfaces, where the handling-ease vs. performance trade-off has always been an important issue. While visual interfaces are naturally more appealing from the human point of view, it had always been difficult to combine an intuitive visual interface with the subsequent way of current computer technology taking that visual input and producing adequate and optimized output code. Second, when it comes to really programming an ECU that is going to be sold and running on the streets, that is one of the reasons for sticking with the more low-level mostly "C"-based programming tools. These tools are much more evolved in the ultimate sense of producing good production code. When ultimate performance has to be guaranteed, then an even more low-level programming method is used: Assembly. Fig. 4.1:1 shows an example of these opposing tools. Even though text-based editors allow commenting the code, visual counterparts always have the feature of much faster mental grasping. Especially considering that most automotive FDEFs have the inherent characteristics of being a flow of many signals from left to right.





```c
// Operation #10
//
// If input is FALSE, then keep output resetted to FALSE, and start delaying otherwise
// If input is TRUE and delay "LAMDIAG_DEL" reached, then set the output to TRUE
// If "LAMDIAG_DEL" is zero, then the output is equal to the input, without any delay
if (!((lam_u > LAM_UMX) || (lam_u < LAM_UMN)))
{
   lamdiag_del = 0;
   lamdiag_b = FALSE;
}
else
   if (lamdiag_del < ROM_LAMDIAG_DEL)
      ++lamdiag_del;
   else
      lamdiag_b = TRUE;
#if (STEREOLAMBDA_SYS == 1)
   if (!((lam_2u > LAM_UMX) || (lam_2u < LAM_UMN)))
   {
      lamdiag_2del = 0;
      lamdiag_2b = FALSE;
   }
   else
      if (lamdiag_2del < ROM_LAMDIAG_DEL)
         ++lamdiag_2del;
      else
         lamdiag_2b = TRUE;
#endif

// Operation #20
lam_e = lamdiag_b || ((CJ125_e & 0x3F) != 0x3F);
#if (STEREOLAMBDA_SYS == 1)
   lam_2e = lamdiag_2b || ((CJ125_2e & 0x3F) != 0x3F);
#endif
```

Fig. 4.1:1 – Visual-based programming tool and text-based "C" counterpart (source: Bosch MS4 [150])

"Handling Efficiency" is herein defined as the real results vs. human end-to-end effort ratio when handling software and hardware packages during system development actions. This ratio encompasses intended development efforts and actions on user-interfaces, down to the hardware platform reactions. It is desirable to be as high as possible. All previously described tools present an apparently high handling efficiency ratio. Indeed, they focus on visual user-interfaces which allow execution of almost all necessary tasks quite easily (Tab. 4.1:2) without caring about underlying mechanisms. Nevertheless, it is clear from experience that unacceptably hard troubleshooting efforts in face of internal problems might appear quickly and without notice. Focus on code-generation efficiency [9] and tool integration [283] is not enough. Despite ease of use being an actual concern [284], all current tools have very complex internal architectures demanding special expertise to troubleshoot related problems. Perpetuation of this development path reveals the infeasibility of implementing some really innovative handling methods, because it would

require prohibitive overhauls of existing structures. In other words, development easiness or difficulty, associated to handling efficiency itself, is a two-fold problem.

SYSTEM HANDLING	**DESIGNING**	- creating, editing, customizing, comparing and changing FDEFs
	DEBUGGING	- monitoring and generating values, inspecting algorithms on FDEFs
	PROTOTYPING	- changing, testing, calibrating and deploying FDEFs onto hardware
	SIMULATING	- running and fully working on FDEFs in the editor without hardware
	LOGGING	- telemetry and data recording from the system out in the field
	TROUBLESHOOTING	- local and remote assistance to the system and people operating it

Tab. 4.1:2 – Major development tasks carried out in automotive scenes

While being true that visual tools have also evolved a lot lately and that auto-code generators have been increasingly able to produce good production code, it is also true that the efforts, time and money to achieve this are not at all small. Another habit that turns into a problem, is that the way of doing things never changed much since the advent of the first computer systems: they all have layers of translators and ever-growing gaps between the user-interfaces and corresponding hardware-targets. There are no rupturing paradigm switches, since new layers of complexity are most often built over the previous working system, to expand its possibilities but never really solving the existing problems at all. With time, the hidden mechanisms developed inside those systems to achieve the desired results could easily be compared to the exaggerated but basic idea in [251].

On the other hand, there are some well-designed products that come to mind when thinking about three factors that make up well-designed products: handling, maintenance and endurance. Please see [Att. 17] for some basic examples.

4.1.2 Current "Tool-Chains"

[Emphasis: long tool-chains with multiple layers producing all kinds of by-products]

Fig. 4.1:3 shows a most complete and actual layered structure, which exposes the underlying machine-code producing mechanisms for the visual FDEF-designing tools used today in automotive applications. It clearly shows the "Complexity Gap" [370] problem described previously in Chapter 3. Auto-code generators, compilers, scanners, parsers and expanders represent the "pitfall sources" in such structures. An expander was included in case external sources with macros are also used. Such external objects were excluded from this picture, with the sole intent of simplifying it, but they also represent an important source of pitfalls that may stray into the main program building path depicted herein. The "building" action involves everything from parsing the graphical elements to the final generation of the machine-code understood by the hardware. Let us also not forget the huge certification efforts [27] of auto-code generators alone. The blue line represents the desired path to achieve monitoring and debugging features (e.g. reading variables, program position) whereas the red line represents the desired path to achieve active prototyping features (e.g. changing code, algorithms, data). These paths are intrinsically difficult to establish, since they have to traverse so many different and "opaque" layers.

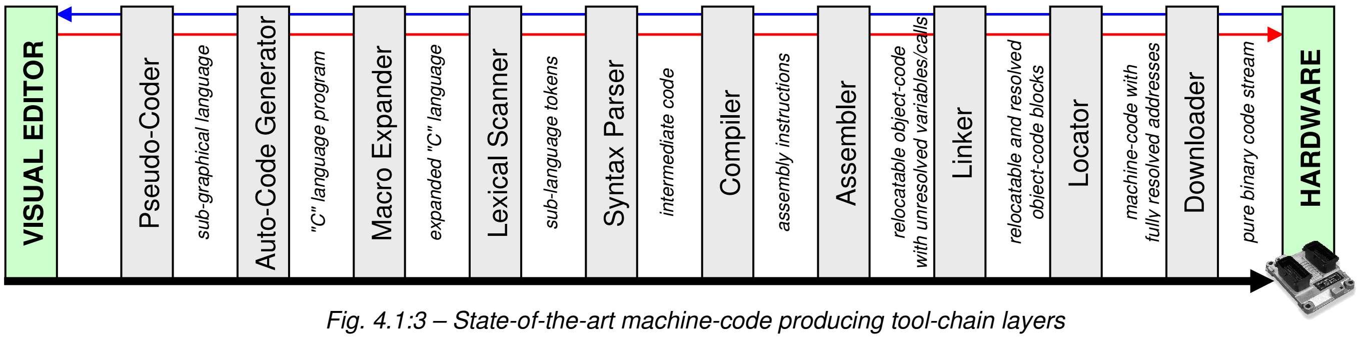


Fig. 4.1:3 – State-of-the-art machine-code producing tool-chain layers

It is important to note that although only one block on the left part of Fig. 4.1:3 is called an "Auto-Code Generator", almost all blocks are automatic code generators in that almost all of them produce code out of a different input source, be it higher or lower level code or even the final machine-code level code. That's exactly the reason why these layered systems got so complex, difficult to maintain and especially prone to pitfalls and limitations when it comes down to enhancing their handling efficiency. There's too much processing in the middle, I those layers. As soon as shorter, faster and better ways of handling systems are attempted, developers invariably collide with heavy, opaque and complex barriers that hinder every enhancement. This completely impedes a rupture innovation, as this work herein aims to accomplish.

Advancements in this matter of the tool-chain depicted in Fig. 4.1:3 really limit themselves to a better obfuscation and hiding of all the processes immediately beneath the visual or textual user-interface editor. More considerations and details about classical tool-chain and related "cross-layer" and also "component-based" designs in [Att. 18].

Starting a completely new system concept from scratch would be too expensive and risky for the automotive industry. Therefore, these historically grown "add-on-top" layered approaches prevail and are a good reason for current automotive tools being so complex and almost impossible to maintain without glitches. Thus, historically speaking, debugging and other manipulation features tended to grow as independent assistive entities (Fig. 4.1:4) as in [134] [135], avoiding complex layer stems.

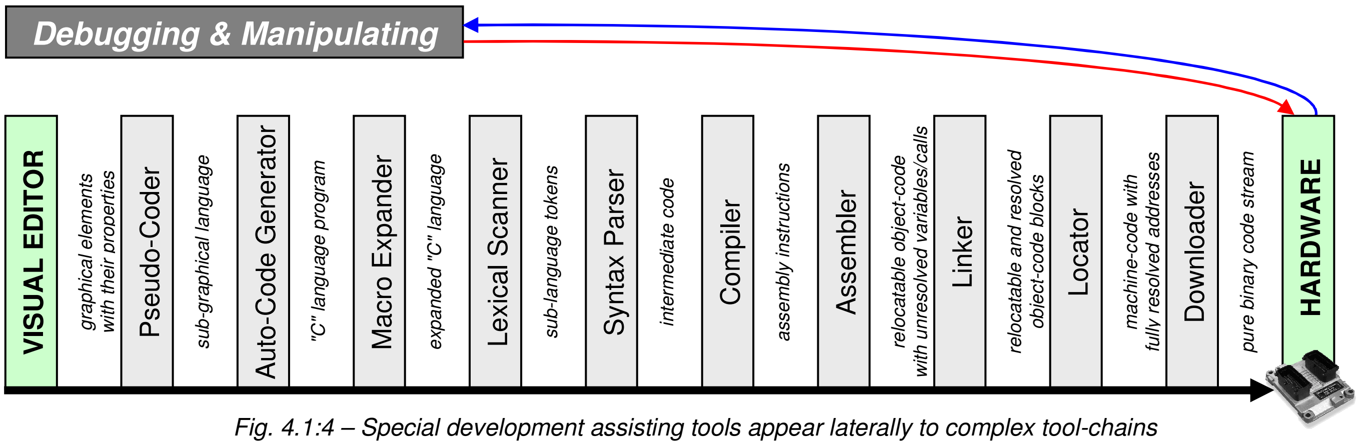


Fig. 4.1:4 – Special development assisting tools appear laterally to complex tool-chains

Tab. 4.1:5 shows a quick empirical comparative overview between some development methodology approaches employed up to today. Note the "bypassing" methodologies: it is quite clear that these are expensive, since they build HW and SW from scratch. On the other hand, these are the least integrated since they do not using any of the original tool-chains' components. Finally, Fig. 4.1:6 compares these methodologies visually, in that it can easily be seen that the central problem-causing tool-chain remains as the common denominator. A special case is that of "wrapping-based" development [439], in that all existing SW and HW components are wrapped into a common denominator called "AUTOSAR", to allow for better integration of many different interfaces and protocols.

	Time & money	Intellectual effort	Risk	Legacy components	Solution to pitfalls	Integration	Handling enhancement
0 ***Everything from scratch***	*****	*	*****	*	*****	*****	*****
1 *Adding layers [435]*	*	*	*	*****	-	***	***
2 *Cross-layer [121]*	**	***	**	*****	-	**	-
3 *Component-based [159]*	*	*	*	*****	-	*	-
4 *HEX-bypass [300]*	***	*	*	*****	*	-	***
5 *Debugger-bypass [134]*	****	*	*	*****	*	-	****
6 *Wrapping-based [439]*	***	*	-	*****	-	-	-

Tab. 4.1:5 – Comparative overview of development methodologies

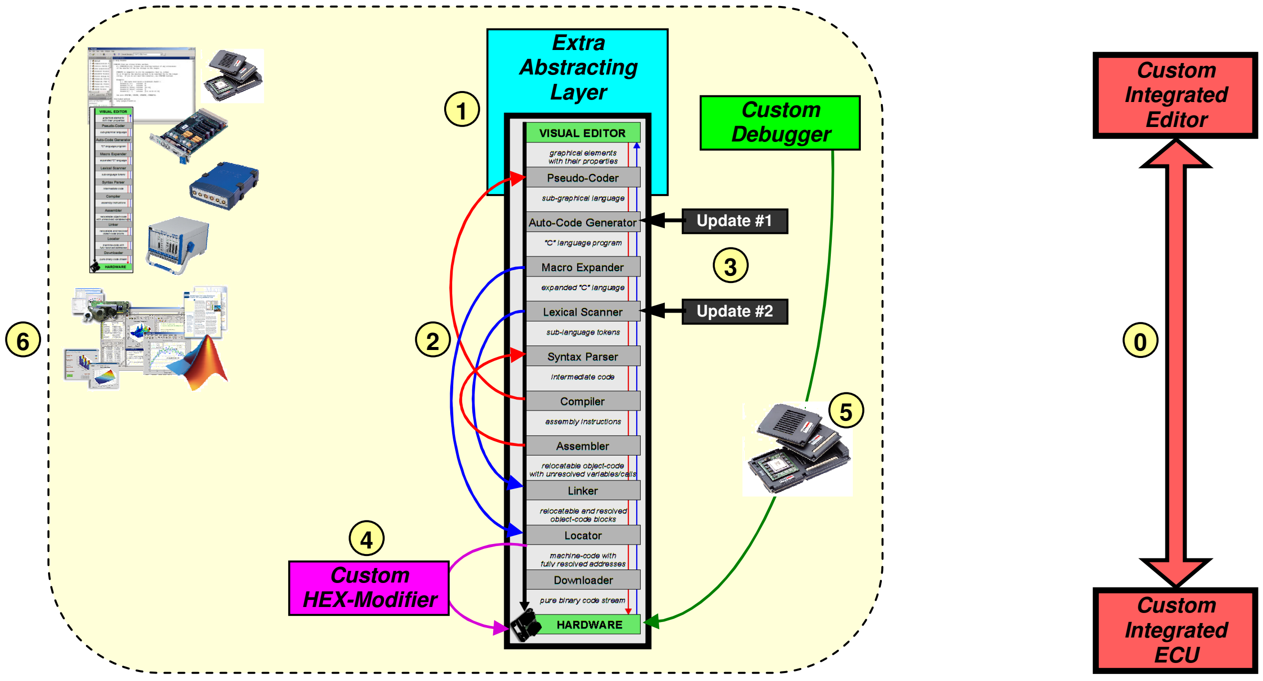


Fig. 4.1:6 – Visual comparative overview of development methodologies; right: clean-slate approach

From the previous comparison and comments above, it is clear that a "clean-slate" approach is a clear candidate to really make something different, hopefully better all the way down to a global SW/HW solution. This would allow for a merge of all layers into one single clean structure, that will hopefully empower the desired *"Easy-Handing"* features, instead of just blocking them or requiring all those previously discussed technological software turnarounds. More asterisks mean more of the column's characteristic.

Currently it is impossible to achieve a really disruptive *"Live-Prototyping"* programming mechanism, by simply using existing software/hardware combinations. Therefore, a new idea must be pursued to accomplish this highly innovative breakthrough. To really accomplish this idea, a much more transparent and narrow software structure is needed, to allow user-interface changes to propagate rapidly and efficiently down to the hardware.

The first step toward actually implementing this mechanism unavoidably consists of getting both ends (user-interface and hardware) as near as possible to each other. By just keeping strictly needed layers, we would reach a simplified structure depicted in Fig. 4.1:7, where a much more direct and hassle-free user↔hardware communications path would be much more suitable to allow highly useful feature, such as *"Live-Prototyping"*. The red and blue arrows represent the now uncloaked paths for commands and feedbacks interchange, respectively. It becomes clear from this "layer-less" architecture, that methodologies such as [121] are completely unnecessary by design. The only question that remains is what must exist in the middle of this simplified architecture.

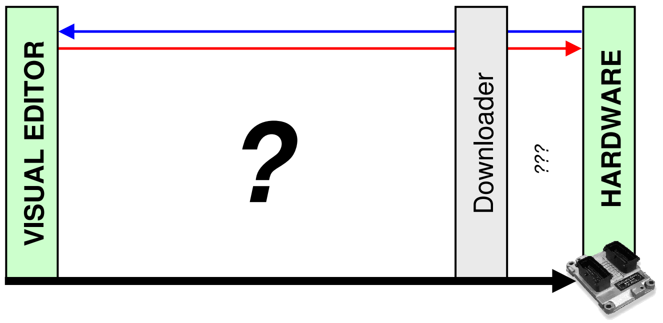


Fig. 4.1:7 – Innovative wishfully narrow SW/HW platform structure

From all previous considerations it is clear that the missing component must be something simple, to be the connecting idea between the user-interface and the hardware. This missing link shall be called *"Macros-Sequence"* and shall be something similar in nature to "byte-code" in Java [79] [81], to "CIL Instructions" in [77] [78], other "byte-code" enabled languages for that matter (such as UCSD Pascal, Matlab m-code, Smalltalk) and research such as *"SYMBOL"* [136] and MONACO [286]. At this precise point, it is very important to understand that everything else connects around/to this central role-playing *"Macros-Sequence"* depicted in Fig. 4.1:8. At one end, the user-visible part, the visual editor, takes the unprocessed raw *"Macros-Sequence"* and represents it graphically through some sort of embedded graphics-engine. Contrary to any graphical tool (like embedded raw source-code [22]), this editor relies only on the *"Macros-Sequence"*, without any separately saved graphical or non-graphical information. It is evident that the hardware platform will have to understand this *"Macros"* language, since no translation is made at that point whatsoever. Thus, at the other end of the system, some special hardware must also be chosen to accommodate this unusual approach. Notice how this approach tries to completely narrow down the "Complexity Gap" [370] previously described in Chapter 3, by creating an unique gluing entity that directly connects to both the top-most and bottom-most layers of the entire development system.

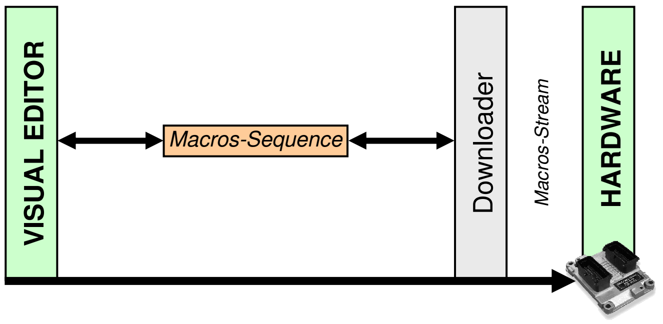


Fig. 4.1:8 – Innovative "Macros-Sequence" based narrow SW/HW platform structure

The only thing that will still exist herein is the "Downloader", which comprises a kind of *"Allocator"* algorithm (described later on) that merely distributes the various *"Macros-Sequences"* to the various parallel-processing modules that will execute those *"Macros"*. This allocator will also fill up the various *"Routing Tables"* explained later on.

As a final note to these ideas, *Interactive Programming* or *Live-Prototyping*, as we call it, is accomplished by those previous two two-way relationships. The *"Macros-Sequence"* are the privileged conceptual/central gateway to this process, keeping everything concentrated upon a single transparent and fully encapsulated entity, the *"Macro"* itself. Following the previous ideas, a strong "WYSIWYG[*]" effect appears, since there are no hidden layers or features. Also, a related strong "WYSIWIS[†]" [17] effect appears, since both users and hardware see exactly the same thing (contrary to complex compilation and decompilation [288] found in other attempts). An even more appropriate way of viewing this effect would be to say that *"What You Think You Program is What Will be Executed - WYTYPWWE"*.

The following sections will expose all the ideas implemented in this thesis' work. As a consequence of having to explain these ideas, portions of the implemented software is also going to be already exposed. The complete implemented *"ECU2010"* system is only going to be presented in Chapter 6, but for the sake of better comprehension, parts of it must be already disclosed alongside with the ideas in the next sections, leaving the final wrap-up for Chapter 6.

4.1.3 Pitfall eradication

[Emphasis: new ideas for pitfall elimination, not just coping with or hiding them]

The most important motivational issue that brought up all the *"ECU2010"* system was that of pitfall elimination, that is, the construction of a system that would inherently solve most pitfalls identified and explained in Chapter 3. This motivation was so strong that an individual section had to be put here, to better emphasize it! There are three main ways of coping with existing pitfalls in classical development and deployment systems:

- ***Pitfall hiding*** → simply covering up pitfalls through new abstracting layers [435] on top of the already existing ones or through building completely new wrapping layers [439] around them
- ***Pitfall workaround*** → simply bypassing around all existing layers through alternative paths that directly address the hardware [134] or that directly manipulate the machine-code output [300]
- ***Pitfall "management"*** → simply enhancing the weak spots by building the necessary and often disruptive add-ons [121] or by encapsulating them into better manageable units [159]

As these classifications clearly imply, there is no pitfall elimination in any solution whatsoever. The only thing that is currently being done in any state-of-the-art product is that of coping with all the existing pitfalls in the ways described all the way before. Furthermore, Chapters 4/5 tries to basically show ways of doing things without having to incur into those pitfalls, by creating mechanisms that do not rely in the same pitfall-producing schemes. This could herein be called "automatic pitfall eradication".

As already said in the abstract, this thesis' work intends to be a major merger of everything that is needed to handle an automotive system, reaching from software to hardware

[*] *What You See is What You Get*
[†] *What You See is What I See*

details. The aim is to get a fully integrated system, without add-ons and side-developments to solve localized problems, while those add-ons cannot even be cleanly added to the existing design due to clear interfacing problems/limitations. The aim is to get to a system that smoothly and homogeneously crystalizes around one core-idea, the *"Macro"*, presented next. Fig. 4.1:9 illustrates the difference talked about above. The left side looks very confusing and that is exactly how it is: a large agglomerate of tools, layers, add-ons, internal and external interfaces, 3rd-party components, scripts, files, libraries, documents, etc. Lots of "just-in-time" made-up highly heterogeneous interfaces round up the seeming mess. That's how state-of-the-art systems are essentially built today. On the right, we have the herein proposed architecture, where everything emerges as being crystallized around the main central "seed" idea. Instead of the former integration, we now have seamless crystallization. Likewise, instead of an evolutionary path, we now have a small revolution.

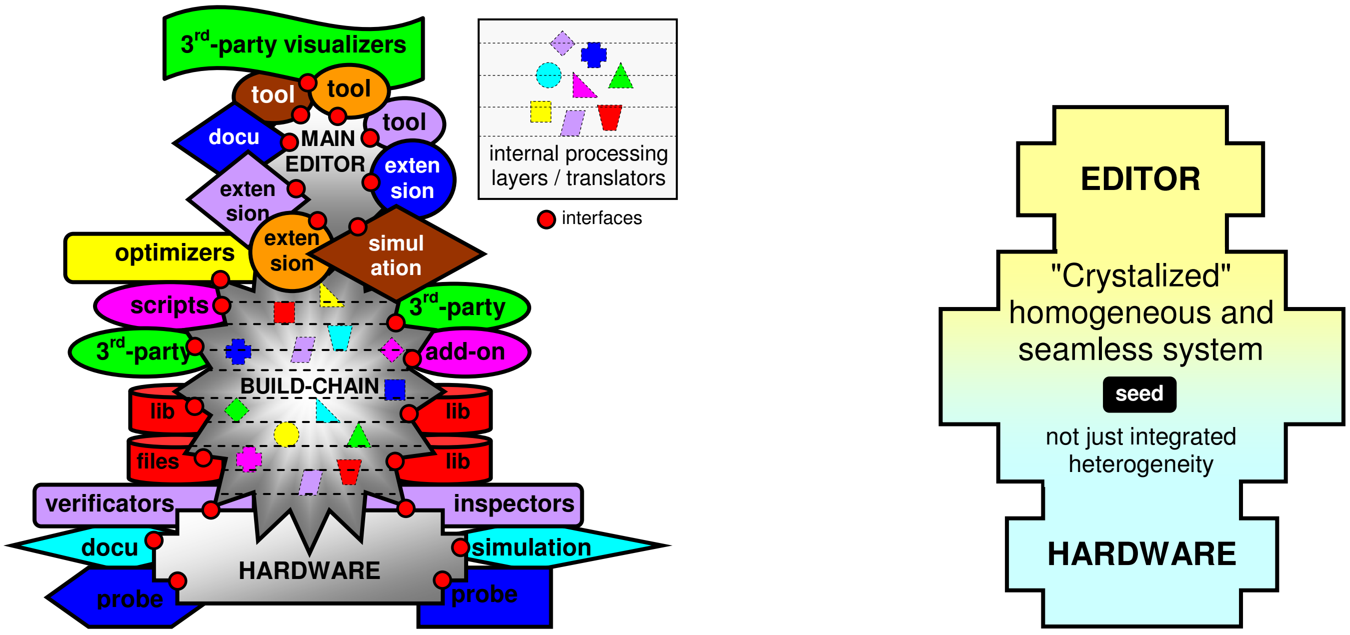


Fig. 4.1:9 – Left: heterogeneous add-on agglomeration; Right: smooth homogeneously grown/crystallized tool

4.2 Main core-idea: the *"Macro"* (and the *"Macros-Sequence"*)

[Emphasis: a single idea radically changes the modus-operandi]

Having seen all the trouble that current tools bring with them, this was exactly the point where the major idea of this whole work emerged: a sort of self-aware[*] graphical elements, the *"Macro-Operations"*, which embody the job of any operation inside an FDEF, be it arithmetic, logic, signal-processing, conditional or anything else that is needed for processing automotive functionality. Pushing further, a *"Macro"* could even be a first-order filter or a more complex PID-Controller. This was the first step into the concept of having one *"Macro-Operation"* for each automotive processing element, be it simple or more complex as stated above. This lead to the herein used name *"MACRO"*, which presents a high-level entity to the developer, doing simple or complex things, but where the user himself does not have to worry about the internals of that entity. Still and in the same context, the developer should not have to worry about any side-effects or interferences caused by each single *"Macro"* to the rest of the program. This goes in direction of the concept of "hygienic macros" [337] but in a much broader and intrinsic sense of the expression. While these intend to eliminate local problems at source-code level only, the herein devised *"Macros"* intend to be 100% innocuous relative to the whole program, the development system and even to the entire processing hardware system.

4.2.1 "Macro" Internals

[Emphasis: describing the "Macro" itself as a single cell among others]

Having the term *"MACRO"* been historically linked to "Macro-Assemblers" where the intention was to allow a single programming statement to be expanded into its needed sub-instructions to accomplish it, the herein concept of *"Macro"* is not divisible and will be directly executed in hardware as an atomic entity. Fig. 4.2:10 compares a classical machine-instruction or high-level statement with a *"Macro"*.

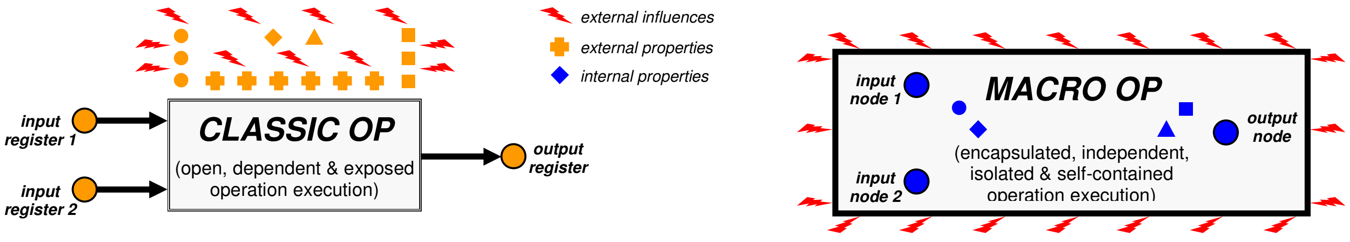


Fig. 4.2:10 – Comparison of classical exposed operation with the new shielded "Macro" operation

While the first suffers from the classical context-generated influences and direct interferences (registers, flags, stacks, pipelines, caches, interrupts, etc.) on its processing outcome, the second is strictly and intrinsically immune to those influences/interferences. Therefore, while the first may output unexpected and undesired results because of register/flag trashing, pipeline problems, stack overflow, cache incoherence or interrupt influence, the second one always outputs exactly the same results.

Due to different contexts of execution, there is always the possibility of two exactly equal statements in an automotive program resulting in two different outputs despite of the inputs

[*] *"Self-Awareness" in the sense of each and every one of these operations would be sufficient and contain all details of processing itself in hardware, without the need of anyone or anything else telling it how to do that job.*

being also exactly the same, due to different contexts of execution external factors influencing them. One disruption can even disrupt other properties in subsequent instructions. In comparison, if a particular type of *"Macro"* outputs a correct result, it is guaranteed "by design" that all *"Macros"* of the same type will always output the correct result, unless some fatal hardware failure occurs. On the other hand, if a *"Macro"* misbehaves at some point, this misbehaviour is restricted to that point. In principle, a *"Macro"* will always display a unique and absolutely repeatable behaviour along the whole program, avoiding any bug propagation. Fig. 4.2:11 illustrates all this.

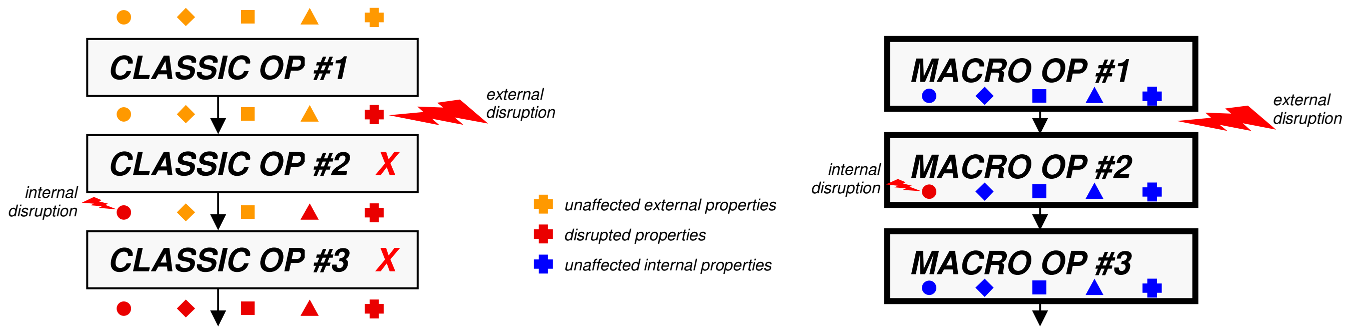


Fig. 4.2:11 – Comparison of bug propagation behaviours after an external and/or internal disruption

Classical methods have no other choice other than globally protecting instruction processing, that is, since the instructions are intrinsically affected by many external factors because they need those to execute appropriately, the only way to try to enhance this is to literally place an overall protective shield over the whole program or over parts of that program. The concept of *"Macro"* entity follows the main idea found in Object-Oriented Programming (OOP): encapsulation and self-confinement of all data/code needed for the entity/object to be capable of autonomously accomplish its intended task. In theory, it is desired that this entity/object will never suffer influences from external sources nor will it propagate bugs to the outside. This is attempted through mechanisms for protecting individual objects from each-others interference, by only allowing interaction through hopefully well-defined interface functions, the so-called "public methods". Internal "private methods" then make up the intrinsic behaviour of the object itself. This has the hoped effect of containing bugs and to prevent them to leak to the outside. As the mechanism implies, it has still to be programmed by the user himself, which already indicates that errors happen and, thus, leaked bugs and general disruption to the outside still happens rather frequently. The *"Macros"* behave differently in that this OOP's conceptual "protective encapsulation" at object-level (an object may contain millions of classical instructions) is intrinsically embodied at instruction-level. What's best: it is completely automatic in the sense that it happens "by design" and without any user knowledge or needed intervention.

Also to be noted that this encapsulation in OOP is only a conceptual language-imposed feature, whereas in the *"Macro-Operations"* the individual "instruction-level protection" is to be architecture and subsequently hardware imposed. This eliminates any error committed by the user himself. Tab. 4.2:12 shows a comparison among existing programming languages/paradigms and the *"Macro-Operations"*. It is clear that, as long as even state-of-the-art visual languages base on high-level languages such as "C", their problems will be the same as those, in respect to bug-propagation issues and such. This is the main reason why the MISRA standard [228] [189] is still in vogue and why even these advanced tools still require highly complex and extensive certification [186] [27]. For more notes about OOP languages refer to [Att. 19].

	Assembly Language	High-Level Language	Object-Oriented Language	Existing Visual Language	***Macros Language***
user-intervention need	very heavy	heavy	Heavy	medium	light
external disruption	- - -	- - -	- +	- +	+++
internal disruption	- - -	- - -	- - -	- - -	+++
bug-propagation	- - -	- - -	- ++	- ++	+++

Tab. 4.2:12 – Main languages and their lack (-) or possession (+) of problem-contention mechanisms

Each *"Macro"* takes one or more inputs and generally has one single output. The inputting values and the outputting results are retrieved and stored in central memory, respectively. These memory locations are herein called *"Nodes"*. An automotive FDEF will thus be represented by a *"Macros-Sequence"* at the centre of the current development system, which will be processed one at a time, in the original sequence and not else wise.

Another reason for the name *"Macro"* is the fact that this entity will be something more than a simple instruction, since it will be "self-aware" in the sense that it will have all it needs inside itself. Instead of having a compiler to tell hardware how to process graphical elements, these already are intrinsically self-sufficient and self-identified. This could be considered a sort of OOP lead to the extreme of everything being inside those graphical elements. This idea is called *"Cellular-ECU"* and is explained later on.

Last but not least, these herein described *"Macro"* entities totally dispenses the usual input data preparation and resulting output data storage "side-operations". Most processors out there use some sort of preparation/post-treatment "side-operations" for their main instructions, in the form of "MOV" (MSP430 [310], PIC [311]), "LD"/'ST" (*CoDeSys* [Att. 3]) "xLOAD"/"xSTORE" (JAVA [Att. 2]) and "LDxxx"/"STxxx" (CIL [Att. 1]) instructions, for example. Fig. 4.2:13 distinguishes this by leaving the inputs and outputs out of the classical instructions and by bearing them inside a *"Macro"*. This is mainly due to the fact that these processors use temporary registers for intermediate storage, potentially leading to problems related to messing up those registers.

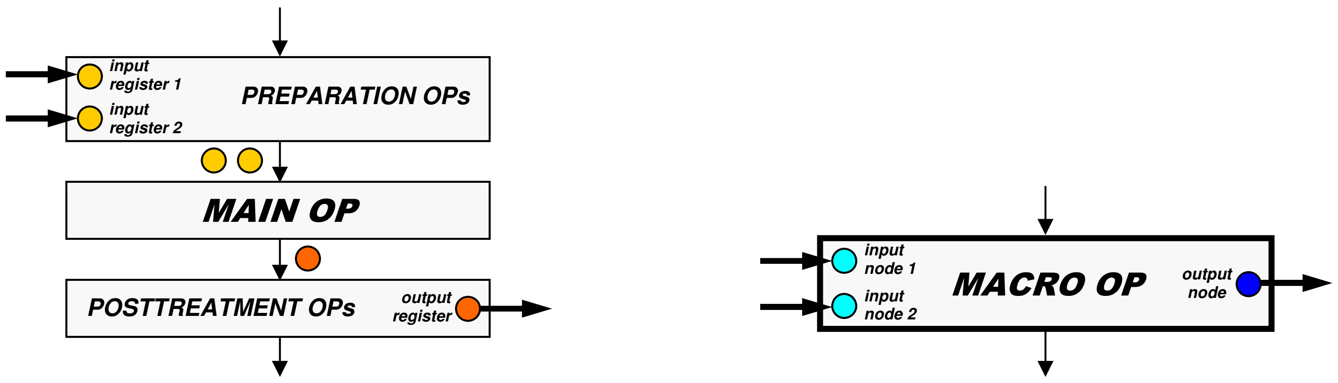


Fig. 4.2:13 – Comparison of classical side-operation loaded operations (left) and stand-alone "Macros" (right)

Keeping in mind that *"Macros"* will not have any "side-operations" as in classical operations (illustrated in Fig. 4.2:13 above) and all previous considerations, Fig. 4.2:14 shows the "execution footprint" or "execution trail" left behind (such as flags, stacks, registers, etc.) and "air-shock" required ahead (instruction preparation steps such as pipelining, side-operations, etc.) by classical machine-instructions and absolute leanness of the *"Macros"* themselves. This "trail" in time and thus execution space has to do with the already discussed "auxiliary" mechanisms (Fig. [Att. 60]:224) used during processing, such as the herein already mentioned flags, stacks, etc., but also with the processing speed enhancement mechanisms such as register renaming, pipelines, caches, etc.

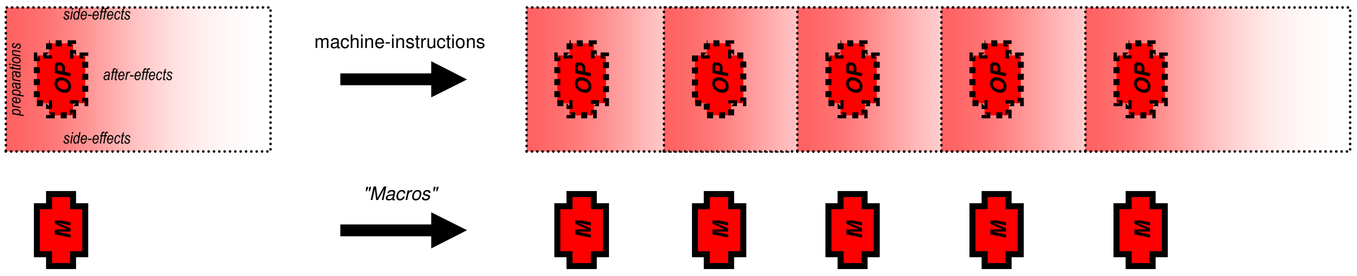


Fig. 4.2:14 – "Trail" and "air-shock" left/needed by machine-instructions (top) and trail-less "Macros" (bottom)

The *"Macros-Sequence"* that arises from these *"Macros"* being executed sequentially comprises, as said before, the central component of the entire system developed herein. The right portion of Fig. 4.2:15 shows this *"Macros-Sequence"* as the elements coming from the user-interface and corresponding to the user-program on that interface. This sequence is going to be directly fed to the executing hardware. Therefore, this *"Macros-Sequence"* envisioned herein is in reality the "source-code" for the software inside the hardware. While in all other systems the "source-code" is the highest-level representation of a program, here it is the very "machine-code" that will be downloaded into the hardware. As said before, this completely closes the "Complexity Gap" [370] illustrated in Fig. 3.5:16, as shown here in Fig. 4.2:15. Finally, also notice a detail relative to the information-flow inside both development environments/tool-chains: while this flow is invariably uni-directional from top to bottom in state-of-the-art, the *"ECU2010"* work developed herein will exhibit horizontal bi-directionality.

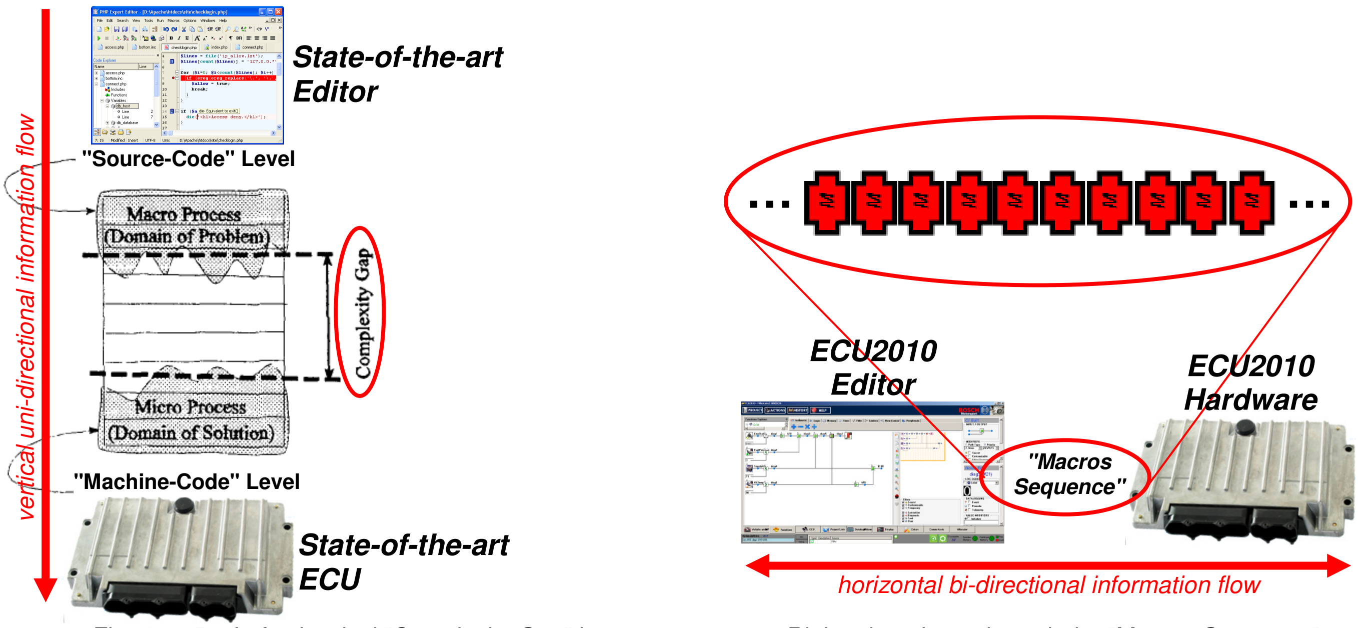

State-of-the-art Editor
"Source-Code" Level
Macro Process (Domain of Problem)
Complexity Gap
Micro Process (Domain of Solution)
"Machine-Code" Level
State-of-the-art ECU
vertical uni-directional information flow
ECU2010 Editor
ECU2010 Hardware
"Macros Sequence"
horizontal bi-directional information flow

Fig. 4.2:15 – Left: classical "Complexity Gap" in current systems; Right: closed gap through the "Macros-Sequence"

This horizontal bi-directional characteristic will later be discussed in the appropriate detail in section "4.5.3.2 Two-way Horizontal Designing made intrinsic". This characteristic will also be of utter importance when discussing features such as advanced debugging and *"Live-Prototyping"*. The critical reason for this is that this horizontal bi-directional information flow channel precisely reflects the needed "unified" mechanism that finally breaks the currently deep and unavoidable abyss between "building (top-down)" and "debugging (bottom-up)" directions still inherently present in modern tools. Also, and as a result of this fairly horizontal complexity plane we can see that instead of having to use complex upward representations of the central "Domain-Specific Language" and equally complex downward implementations, the *"Macros"* are used "as-is" to the left (user-interface) and to the right (processing hardware), without use of any translations whatsoever. Visual representation of *“Macros”* on the visual user-interface could be erroneously interpreted as a translation, but while a real translation requires the translator to look at an entire phrase (sequence of instructions) to be able to correctly translate them, here it is a strict biunivocal 1:1 thing, more correctly called as a direct mapping. This 90° rotated "Middle-Out" kind of development is illustrated in Fig. 4.2:16.

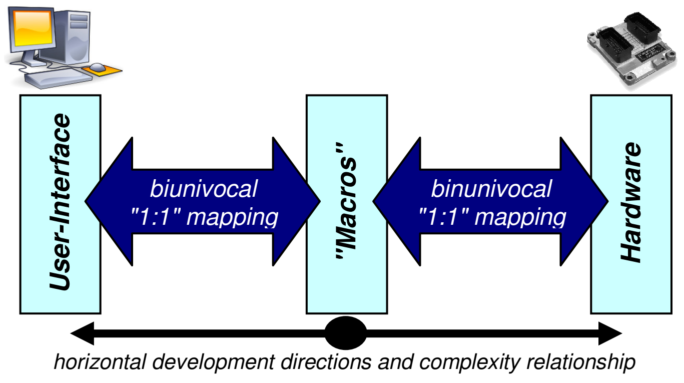

User-Interface
biunivocal "1:1" mapping
"Macros"
binunivocal "1:1" mapping
Hardware
horizontal development directions and complexity relationship

Fig. 4.2:16 – Illustration of the horizontal "Middle-Out" approach taken within the work herein.

A *"Macro"* must not be taken only for a classical meaning of sequence of commands. It must be regarded as an indivisible entity. It contains several description elements, such as the input and output *"Nodes"*, some internal data such as a filtering time-constant, etc., being taken by the *"Macros-Processor"*, which executes it according to those "package contents". This processor is described in detail in the next section.

Last but not least, it must be noted that this *"Macro"* concept totally eliminates the academic war around software/hardware layers altogether: since everything is just plain flat, there is no need whatsoever to study which translation level or layer should be accomplished by software and which should be left for hardware, such as things related to interpretation, compilation, JIT, direct/indirect processing, etc. *"Macros-Sequences"* are simply executed inside the *"Macros-Processor"* and that is about the only thing that actively happens in this system from the processing point of view. For more details on research about these levels, directionality and related topics, please refer to [Att. 20].

4.2.2 “Macros-Sequence” Core Role

[Emphasis: revelation of the real “Macros-Sequence” central system role]

Fig. [Att. 20]:91 shows the "middle-out development" concept with the language in the exact centre of the tool-chain, whereas the leftmost of Fig. 4.1:8 shows the crystallization of the rest of the system out of the central concept, which is, again, the same

"Macros-Sequence". Thus, this *"Macros-Sequence"* is what produces the whole rest of the system in terms of what is shown in the visual interface/editor and in terms of what is executed by the hardware itself. Also, the difference between the straightforward "mapping" and the more complex "translation" concepts has to be clarified:

- ***"Mapping"*** → movement of information from one format to another, without altering the base-content or the meaning of each item in the original format under which that information was encoded. Format can also be regarded as "encoded information", whereas different encodings (different formats) basically do not add or eliminate any bit of the information intended to be moved/transferred. Generally, no context or flavour items are added or eliminated as well. Furthermore, only a strict biunivocal 1:1 loss-less and meaning-preserving mapping is intended in the work developed herein (see previous Fig. 4.2:16).
- ***"Translation"*** → transformation of information from one language to another, changing one, several or all of the characteristics of the original information items and/or/format, as the tool-chain is ran from top to bottom: quantity of items (often more items to represent those inputted), quality of items (items usually get additives and side-operations as well), item basic morphology gets totally different/transformed (such as in automatic code-generators, where the looks of the resulting "C" code has nothing to do with the graphical input), among other transformations. Basically, the biggest chunk of low-level pitfalls described earlier on herein are added by the translation-based tool-chain itself..

Mapping (visual user-interface ↔ *"Macros-Sequence"* ↔ hardware processing movement mechanisms) is represented on the left side of Fig. 4.2:17, while translating (classical graphical user-interfaces, tool-chains and hardware platforms) is one or a combination of processes illustrated on the right of Fig. 4.2:17. Exception to this mapping-only flavour of *"Macros-Sequences"* is the later on explained "command-line" (left of Fig. 4.2:17) form of speed-input for experienced users/developers, in that an extra command-line allows to textually insert, modify and delete *"Macros"* on the user-interface, as well as to command the entire user-interface itself. Textual/verbal segments, although already highly simplified, have to be translated into the inner native *"Macros-Sequences"* language. Furthermore, this happens in a one-way fashion only. As soon as those textual segments are finished translating into that inner format, everything works again as usual.

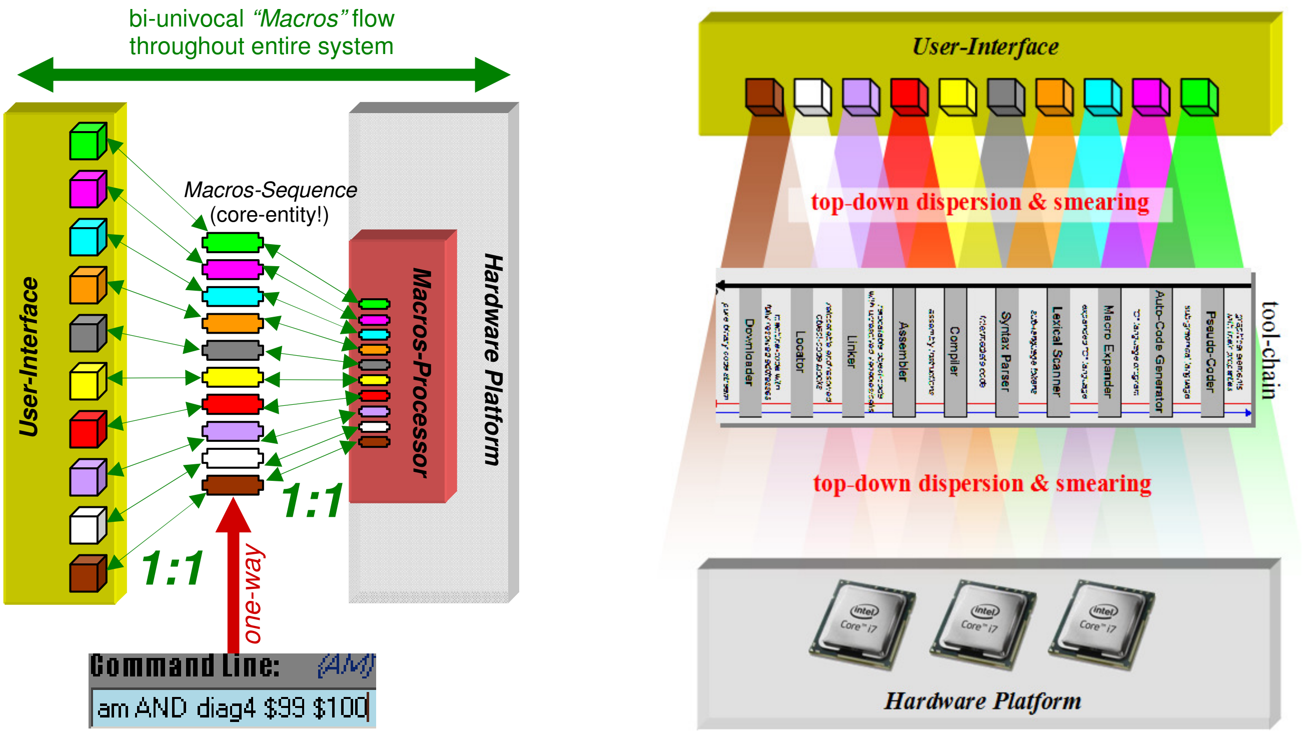


Fig. 4.2:17 – Illustration of "Macros mapping" (left) and "classical language translating" (right) mechanisms.

The "mapping" action determines that there are only these two frontier realities, both coming from that very same resulting *"Macros-Sequence"*:

- **Visual interface/editor side-burst** → the central *"Macros-Sequence"* (in form of internal byte-code like data-structures) is fed back to the user through a dedicated graphics-engine, which does nothing more than mapping each byte-code into its corresponding visual icon onto the screen. This graphics-engine does not translate in the sense of having complete phrases with certain semantics and constructive details, but only and directly picks one *"Macro"* at a time and maps/displays them in the sequence they appear in that *"Macros-Sequence"*. Previous Fig. 4.2:16 illustrated this with its "1:1 mapping" relationship on the left.
- **Hardware execution side-burst** → the same central *"Macros-Sequence"* is fed directly into the hardware platform that has the ability to execute those *"Macros"* directly. There is no translation whatsoever for the simple reason that there is no need to have any whatsoever. Previous Fig. 4.2:16 illustrated this with its "1:1 mapping" relationship on the right.

Everything that interfaces with humans visually (interface/editor) or textually (command-line) is just a means of entering those *"Macros-Sequences"* in a more user-friendly form, nothing more. That user-friendly input is mapped (in case of the visual interface) and translated (in case of the command-line) into the central byte-code like *"Macros-Sequences"* and from there the same visual FDEFs are fed back to the user. The hardware platform will even process those *"Macros-Sequences"* directly. This is to say that all visual and command-line input is in reality discarded/lost after it has been added to the underlying central *"Macros-Sequence"*, whereas any visual feedback is just an "image" of that same *"Macros-Sequence"*. No information about the visual or command-line input is kept, other than that present in the *"Macros-Sequence"*. When talking about the *"Macros-Sequence"*, it is always meant that central byte-code like information.

Further below herein, Fig. 4.5:75 shows the architectural and implementation correctness that emerges due to the 1:1 relationship illustrated in the previous Fig. 4.2:17. There, it will be further illustrated and clarified, how the central *"Macros-Sequence"* really calls for the bull's-eye role in this system developed herein. This extra clarification will make more sense there below herein, after the "automatic graphics-engine" (Fig. 4.5:50) has been explained and illustrated, since only then will it be perfectly clear that the visual interface is a mere reflection or virtual representation of that central *"Macros-Sequence"* for better human grasping and that it has no other use than just that.

FINAL NOTE: Classical systems suffer from the fact that the same instruction placed in different spots may yield different results, due to the internal side-effects of the processing mechanisms. This also leads to catastrophic bug-propagation as soon as something fails, affecting subsequent instructions as well.

From the statements above, ***PITFALL O#8*** *(Code Reliability)*, ***PITFALL O#65*** *(Execution Repeatability)*, ***PITFALL C#64*** *(Bug Propagation & Cross-Spreading)*, ***PITFALL C#9*** *(Bug Searching)* and ***PITFALL O#68*** *(Tool Certification)* are thus effectively eliminated through guaranteed repeatable operation execution, execution isolation and guaranteed execution correctness.

4.3 The consequent idea: the *"Macros-Processor"*

[Emphasis: the processing engine that runs on the "Macros" fuel]

Additionally to all the details about the execution of the *"Macros"* as exposed in the previous section, this processor is fully reset after each execution. This is desirable and possible due to the very nature of "self-containment" and "self-sufficiency" of each individual *"Macro"*, thus allowing each one to be processed independently of the rest. Fig. 4.3:18 illustrates this internal processing mechanism inside the hypothetical hardware (preliminarily simulated in software, for now). Note how all this *"Macros-Processor"* does is fetching the current *"Macro"* and the corresponding *"Nodes"*, processing them and storing the result *"Node"*. This central characteristic enables parallel-processing as an inherent feature, instead of having to build extra mechanisms around current processor to allow them to work together. Another consequence of this ever-repetitive "reset-fetch-execute-store" cycle is that this processor is guaranteed to not crash and guaranteed to execute the *"Macros"* always correctly, as long as it already executed those *"Macros"* correctly at least once and did not crash at that point.

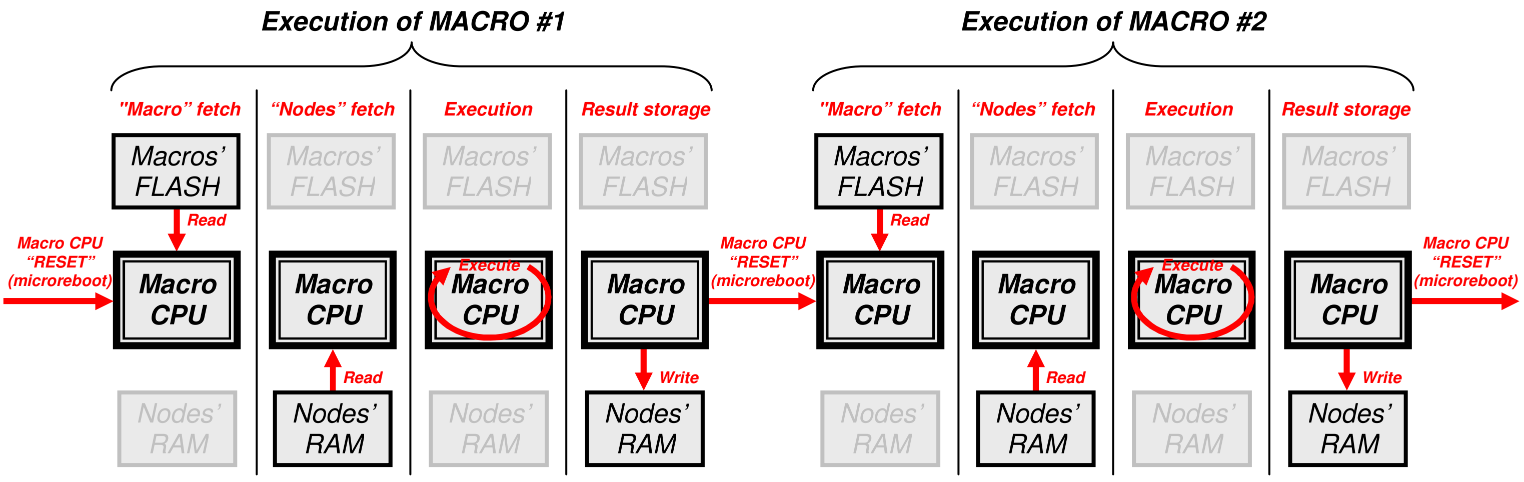


Fig. 4.3:18 – Isolated execution of each "Macro" where the CPU has no recall of the last "Macro" executed

4.3.1 "Amnesic" or "Micro-Rebooted" Processing

[Emphasis: starting over repeatedly is better than aging beyond repair]

The previous important characteristic of this *"Macros-Processor"* resetting itself after executing each *"Macro"* or, viewed from the other side, resetting itself before executing a new *"Macro"*, is an extreme software renewal mechanism also employed in "crash-only" systems [329]. This mechanism ensures two essential things: that to each *"Macro"* will be guaranteed exactly the same execution environment conditions and that no "software aging" effects will build-up inside the *"Macros-Processor"*. By repeatedly rejuvenating the execution core through so-called "micro-rebooting" [328], the *"Macros-Processor"* guarantees that even if something goes wrong with some *"Macro"*, the next one will be treated as if it was the very first one being executed on a completely clean system. One could call this processor "amnesia" as being a highly desired feature. Note that processors that employ mechanisms such as pipelining, register renaming, branch prediction, stacking, side-operations, etc., cannot benefit of this "amnesic processing" for obvious reasons of having to retain "context". This *"Macro-Processing"* never leaks a transient execution problem with one *"Macro"* to subsequent ones. At most, the resulting output

"Node" of the disrupted *"Macro"* will contain a bad value and will propagate at functional level throughout the FDEF, but this propagation will never be at processing level, which is far worse. As in [328] and except for the problematic *"Macro"* outcome (causes a slight hiccup[*], at most), no additional data corruption/loss or system crash is possible.

More than just a new idea, this way of doing things if extremely effective in keeping problems away of the raw processing path, mainly because of the "set it and forget it" kind of nature, in contrast to current similar ways [328] [329] that absolutely require complex monitoring mechanisms to correctly access the need for a provocation-caused "micro-reboot". Fig. 4.3:19 illustrates this continuity compared to the highly discrete "crash-only" mechanism. As can be easily seen, the "mean aging" or degradation level of the *"Macros-Processor"* is intrinsically kept always at the minimum level possible. In this system it is not possible to lower that level more than this, because during *"Macro"* execution the executor must not be disturbed in any way. Although "micro-booting" is generally regarded as an availability boosting mechanism, the *"Macros-Processor"* mimics the repeated "micro-booting" strategy pursuit in [331] to keep a computing system as highly-performing as possible by avoiding clogging situations right from the start. Refer to [Att. 21] for more details about "micro-rebooted" systems and related research.

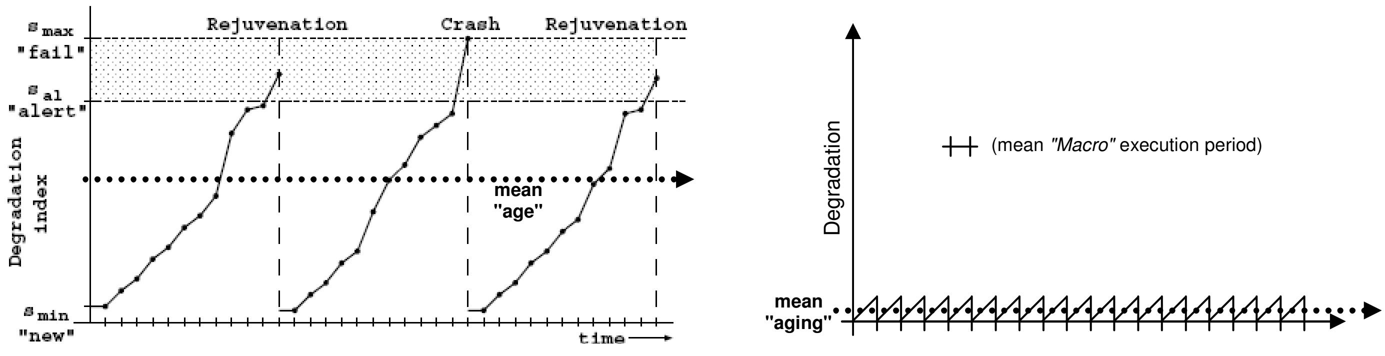


Fig. 4.3:19 – Comparison of "discrete crash-only" systems (source: [330]) with "analog Macro-Processing"

Although current state-of-the-art automotive ECU designs [10] [150] only include the most standard and hard crash-recovery mechanisms, such as watchdogs, fatal-error trapping and higher-level software "question-answer" checks, it will not be long until ECU software complexity catches up with the need to confine bugs impossible to resolve on time. While these recover-upon-crash concepts are still confined to services on internet servers, they will undoubtedly spread to every software reality out there at some moment, being implemented in that reality to some extent.

It must be clearly pointed out that this kind of highly tolerant "amnesic/micro-rebooted" processing is only made possible thanks to the architectural choice having been a purely memory-based one. Since all indispensable *"Nodes"* are outside this processor and nothing besides a program-counter exists as "CPU-state", this processor can thus be "reset" upon begin of each *"Macro"* execution. And even this program-counter is quite a "tolerant" issue, since the typical automotive functionality is something that is executed repeatedly and without any special order in most FDEF cases. In other words, the entire

[*] *These software "hiccups" are common in automotive systems. Just pay good attention to the RPMs of a high-end car such as the BMW 530D, for example, to notice slight "RPM hiccups" from now and then. These are most probably caused by some very rare timing disruptions where the calculations are not produced always exactly the same way. This can be caused by an interrupt that happened at a never before tested point, etc.*

"program state" will reside inside the *"Nodes"* memory. This further brings forth one of the code-coverage issues, namely the fact that every *"Macro"* will effectively always execute in the exact same way. This eliminates the need for repeated, even exhaustive need, for testing execution correctness in terms of its results, in the work developed herein. The same idea applies to orthogonal execution, since a *"Macro"* may thus be executed anywhere and anyhow, without any range-associated limitations or problems (as existent within processors where instructions have typical usage-range limitations such as addressing modes, parameter combination, etc.). The fact that the processor core is freshly applied (re-booted) to each *"Macro"* as if it was a first-time execution, without any "auxiliary" mechanisms' influence over time, also helps to this orthogonality (total independence from time influences in terms of pile-up of past execution residues).

FINAL NOTE: Classical processing entities are currently being overheaded with complex software performance degradation supervision mechanisms, These should avoid or at least minimize the worst side-effect of current complexity: crashes. This resembles a tree being held by a stick because it cannot hold itself anymore due to age.

From the statements above, ***PITFALL O#42*** *(Supervision)*, ***PITFALL C#37*** *(Software Aging)*, ***PITFALL O#10*** *(1/4 - Code Coverage)*, ***PITFALL H#35*** *(1/2 - Non-Orthogonality)* and ***PITFALL O#33*** *(Verification for Correctness)* are thus effectively eliminated through effective “reset-then-execute” execution of each individual *“Macro”*.

4.3.2 "Context-less" and "State-less" Processing

[Emphasis: no recall of the past, every task is the "first one"]

Another main distinctive global characteristic of the *"Macros-Processor"* is its fully context-/state-less *modus-operandi*. This goes in resonance with [328] [329], because context/state causes recovery actions to be very difficult or impossible to do without messing something up fatally. Without any context/state inside the *"Macros-Processor"* itself, micro-rebooting is just as easy as switching it off and back on. As said before, the only “state” this kind of processor retains, is the "program-counter" that increments through the *“Macros-Sequence”*. *"Nodes"* containing FDEF state/values reside outside this processor hardware kernel. Additionally, contrary to "byte-codes" in Java [79] [81] and to the "CIL Instructions" in the ".NET" environment [77] [78], the herein applied *"Macros"* are executed in a 100% strict "one-intended-action one-isolated-instruction" paradigm. Finally, also contrary to the ARM-accelerated *Jazelle DBX* technology that executes more complex instructions through optimized ARM code [91] [102], the *"Macros-Processor"* executes all *"Macros"* directly/homogeneously in the same hardware (ALU). The only detail relative to some higher-complexity *"Macros"* is that they are sub-divided into basic ALU operations and then fed into this ALU sequentially by the SEQUENCER. The only difference noticeable to the outside is that these higher-level *"Macros"* take longer to fully execute.

In the case of the *"Macros"*, from the total of the 57 listed in Tab. [Att. 23]:102, 22 are higher-complexity ones. These thus represent about 39% of the total of the set, but when really used in automotive FDEFs (as in the MS4-Easy Clubsport {40CS0X1A} ECU software [308]) these statistically represent also only about 7% [309] of the total ECU operations. Coincidently, this number is similar to the *Jazelle's* complex instructions proportion (the more complex instructions the *Jazelle* emulates into ARM instructions, represent about 5% [102] of the overall programs). Tab. 4.3:20 then lists the main distinctive processing characteristics of the *"Macros"*.

SELF-CONTAINEDNMENT/DESCRIPTION
All *“Macros”* contain all information needed by the *"Macros-Processor"* to process them
→ this ensures no need for any auxiliary external context or state information, thereby intrinsically eliminating any kind of external influences, fatal interferences, bug-propagation and external manipulation of any kind

SELF-SUFFICIENCY
“Macros” handle their own previous input data preparation and posterior results storage
→ this ensures no need for any auxiliary instructions before and after execution of *"Macros"*, thereby intrinsically ensuring that no data corruption can happen in-between and thereby simplifying the underlying execution mechanisms entirely

SELF-IDENTIFICATION
“Macros” are self-identified entities, directly processed without any transformation
→ this ensures that the hardware sees exactly what the visual interface sees and vice-versa, thereby intrinsically avoiding any kind of misunderstanding, synchronization loss and misrepresentation

SELF-RENEWAL/MICROREBOOTING
The *"Macros-Processor"* is always reset (microreboot) after executing each *"Macro"*
→ this ensures the exact same initial-conditions for each *"Macro"*, thereby intrinsically eliminating all effects of propagation of preceding bugs and guaranteeing that each *"Macros"* will always behave exactly the same way

SELF-LEANNESS
The *"Macros-Processor"* directly executes *"Macros"* without any gimmicks whatsoever
→ this ensures that no side-effects resulting from surrounding complex acceleration and optimization structures appear, while keeping these execution structures absolutely lean and reduced to the absolutely necessary hardware

SELF-PARALLELIZING
The *"Macros-Processor"* is inherently enables for effortless parallel-processing
→ this ensures that processing power needs are relatively easy to comply with, including the fact that no matter how the eventually desired parallelism is implemented there will be no program related disruptions whatsoever

SELF-PROOF
The *"Macros-Processor"* is inherently crash-proof in the widest sense of the expression
→ this ensures that processing power needs are relatively easy to comply with, including the fact that no matter how the eventually desired parallelism is implemented there will be no program related disruptions whatsoever

SELF-HOMOGEINITY
The *"Macros-Processor"* homogeneously executes all *"Macros"* in the same hardware
→ this ensures that there are no possible disrupting or externally noticeable effects due to some operations being executed in one hardware component and others being executed in another hardware component.

Tab. 4.3:20 – Main distinctive "Macro-Processing" characteristics, which differ from most other current mechanisms

Fig. 4.3:21 shows the internal structure of such a *"Macros-Processor"*. It is very simple and compact. There is only a Sequencer (similar to a simple state-machine) that receives the *"Macros"* and *"Nodes"* from both Fetchers, and an Arithmetic-Logic Unit (ALU) that does all the calculations and logical operations corresponding to the *"Macro"* at hand. The result is then finally stored back through the Storer. There is no program-counter (PC) at this time to keep track of the current *"Macro"*, because this is going to be accomplished by a special block called *"LP"*-block presented later. An important glimpse about this PC is that in reality, due to the automotive intrinsic parallel nature of signal-processing, this PC only serves to keep track of the next *"Macro"* to be executed and nothing more.

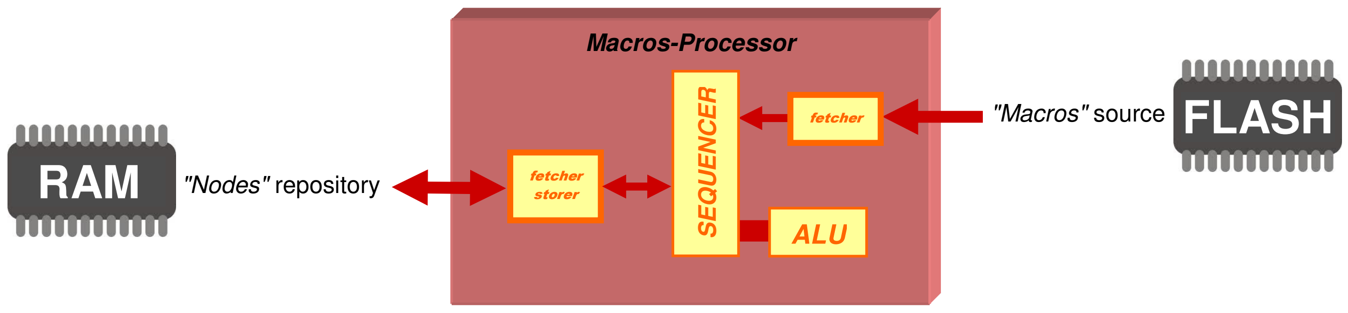


Fig. 4.3:21 – "Macros-Processor" with its internal essential components

The operation of the fetchers logic-only units is further exposed in the example execution in Fig. 4.3:22, where a *"Macro"* is fetched from FLASH, *"Nodes"* are fetched from RAM and the processed result is then stored back as *"Nodes"* into the same RAM.

Again, this is "context-less" processing, at last on the plain processor's level, is only possible due to the architecture choices being made herein. The memory containing all the *"Nodes"* is going to be the only one containing "state". The corresponding *"Macros-Processor"* never really knows what it is doing, besides the "program-counter" that points to the next *"Macro"* to be executed. More importantly, it never knows if it is executing for the first time, if it is executing for a short or already long time. "State-less" operation beyond the operations themselves is typical in pure data-flow languages [380].

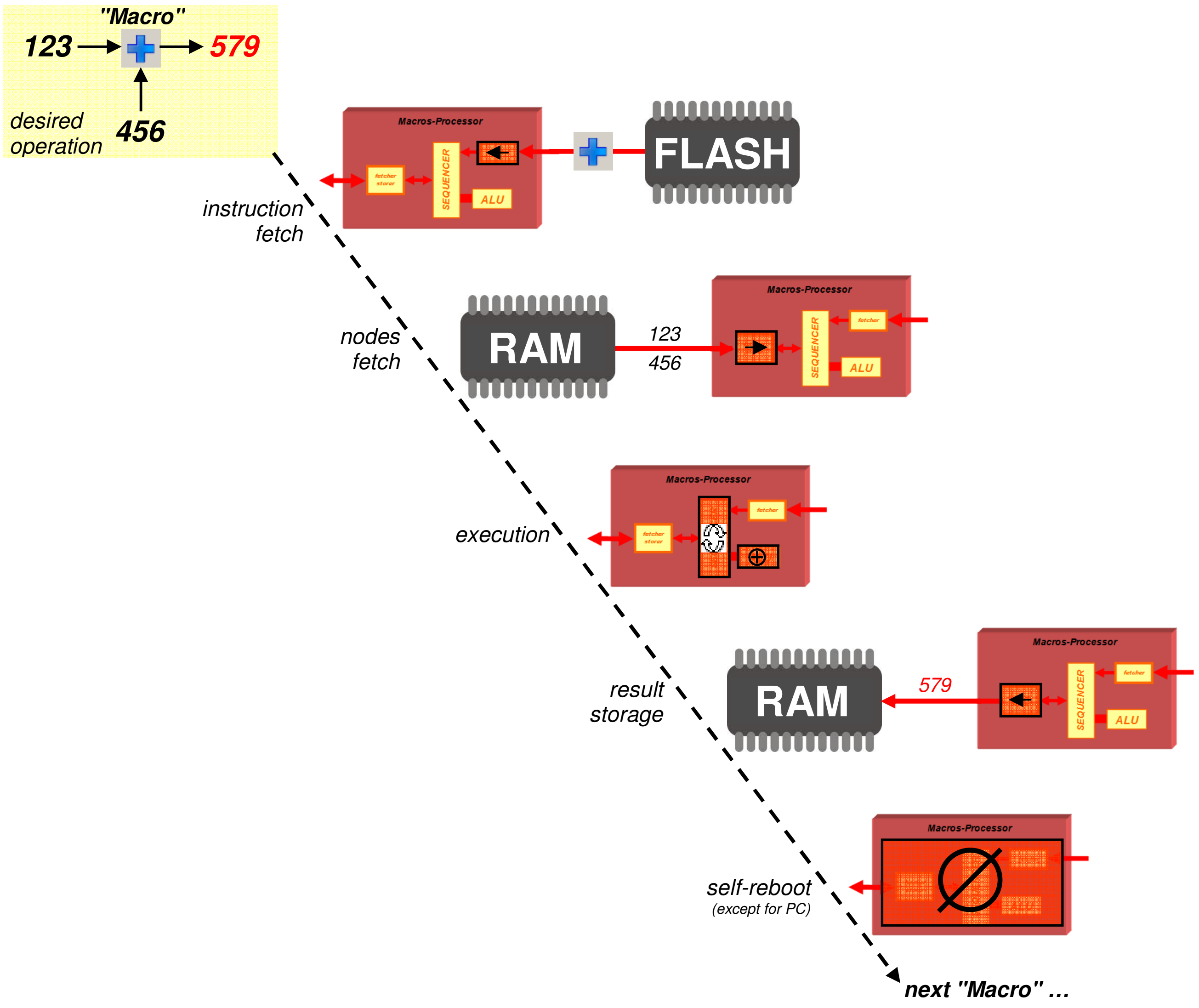


Fig. 4.3:22 – "Macros-Processor" executing a simple ADD "Macro"

It must be clearly pointed out that also this kind of highly tolerant "Context-less" and "State-less" processing is only made possible thanks to the architectural choice having been a purely memory-based one. Since all indispensable *"Nodes"* are outside this processor and nothing besides the program-counter resides as "state" inside the processor itself, this processor can thus be "reset" upon begin of each *"Macro"* execution without loosing any information to carry on correctly. This "random" processing of *"Macros"* will allow for some interesting features such as back-stepping, random stepping, etc., discussed later on, during the solutions explanations. Refer to [Att. 21] for more details about "state-less" systems and related research.

FINAL NOTE: Classical processing entities are currently overcrowded with influences from all around. This overcrowding produces all kinds of potential pitfalls and problems, whereby the results are not what was intended. This leads to too much time and efforts being spent with CPU and auxiliary mechanisms' details.

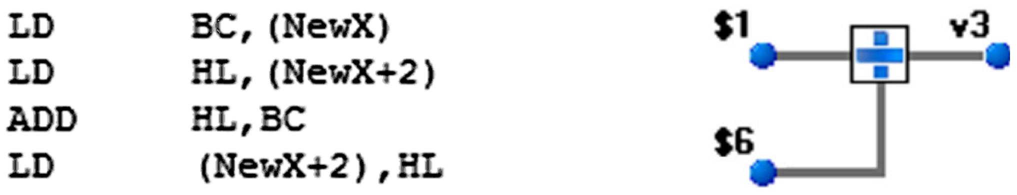
From the statements above, ***PITFALL 0#79*** $^{\text{(Wrong Processing Focus)}}$ is thus effectively eliminated through "context-less" execution of each individual *"Macro"*, so that these *"Macros"* are correctly executed and the central processing natural worries for failure are effectively eliminated.

From the statements above, ***PITFALL 0#10*** $^{\text{(2/4 Code-Coverage)}}$ is thus partially eliminated through this same before mentioned "context-less" execution of each individual *"Macro"*, so "*Macros"* are correctly executed.

4.3.3 *"Random"* Processing

[Emphasis: no recall of the past, every task is the "first one"]

This concept if just a deeper and more general view on the *"Context-less" and "State-less"* processing mentioned previously, where the *"Macros"* feature a total independence among each other within the complete *"Macros-Sequence"*. In contrast to classical processing systems where temporary registers assume intermediate results (see Fig. 4.3:23), the *"Macros"* will never need that because they always output directly to external *"Nodes"*. While there indeed exists a Program-Counter (PC) to keep track of the current/next *"Macro"* to be executed inside that sequence, this sequence in by no way strict. The automotive signal-processing nature is mostly parallel, so that almost any *"Macro"* could theoretically be executed anywhere in the processing-flow. The only real use of the PC in this *"Macros-Processor"* is that of ensuring that all *"Macros"* will get their execution turn.



```
LD      BC,(NewX)
LD      HL,(NewX+2)
ADD     HL,BC
LD      (NewX+2),HL
```

Fig. 4.3:23 – Left: Assembly code to add two variables; Right: atomic addition "Macro"

A totally careless/random processing sequence could be forced into this *"Macros-Processor"* and the automotive system it controls would still function, as long as all *"Macros"* get to be executed sometime within the maximum allowed "real-time window". This "real-time window" will be later called *"Cycle-Time"* and represents the cyclic time-duration of a complete *"Macros-Sequence"* pass, that is, the period after which all *"Macros"* have been executed exactly once. In an extreme case, if each *"Macro"* had its own processor repeatedly executing it over and over, parallel to all others, we would still have everything running correctly. It can also be reasoned that those processors could even run at different speeds, as long as the previously mentioned maximum *"Cycle-Time"* is guaranteed. Furthermore, the PC could then be totally eliminated. These theoretical possibilities are again picked up in the last sections of "Future Work".

Fig. 4.3:24 shows the obvious cost that results from "sequence reversion", where some *"Nodes"* would have delayed updated values. Having the input *"Nodes"* **A**, **B**, **D** and **F** updated at one moment, the outputs **C**, **E** and **G** would be delayed more or less,

depending on those "sequence reversals". There are some components in an automotive system, such as knock-control, ignition and others, that might get into some temporary trouble if *"Nodes"* are not updated fast enough or are assigned incorrect values. But that is where the maximum allowable *"Cycle-Time"* gets into action: if those "sequence reversals" would be sorted out fast enough, there would again be no problem at all. In the work developed herein however, the PC is going to exist and a single processor will execute long *"Macros-Sequences"*.

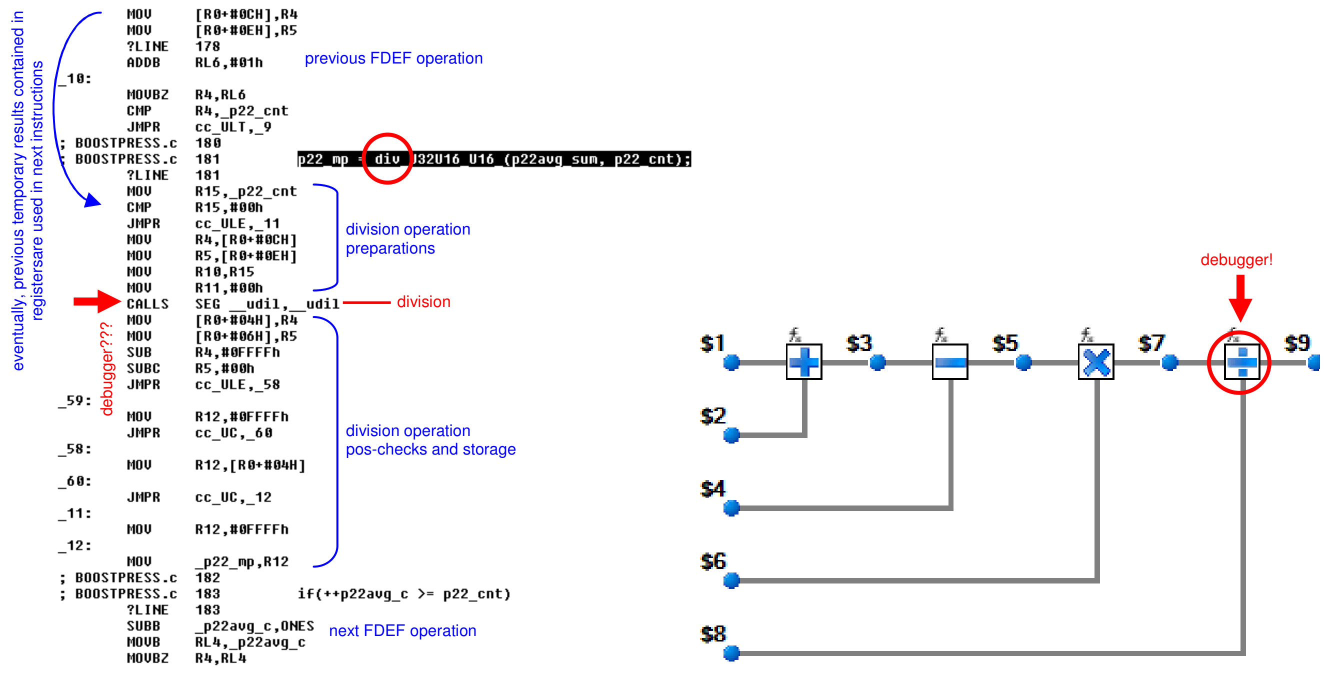
```
A = 1 | B = 2 | C = 0 | D = 3 | E = 0 | F = 4| G = 0

 C = A + B  =  1 + 2  =  3
 E = C + D  =  3 + 3  =  6
G = E + F  =  6 + 4  =  10

G = E + F  =  0 + 4  =  4
 C = A + B  =  1 + 2  =  3
 E = C + D  =  3 + 3  =  6

G = E + F  =  6 + 4  =  10
 C = A + B  =  1 + 2  =  3
 E = C + D  =  3 + 3  =  6
```

Fig. 4.3:24 – Left: correct sequence; Middle: single sequence reversion; Right: double sequence reversion

This notion of "Random Processing" further consolidates the reality of "Context-less" processing, in that the *"Macros"* are independent of any external happenings, except for the *"Nodes"* they use. Those *"Nodes"* reside in a random-access memory without any special structure, in contrast to classical systems, therefore further consolidating the notion of "Random" processing. The main difference is that herein any of each *"Macro"* in an FDEF can be executed at any time. This allows for some very interesting extra handling features, virtually impossible to attain in classical systems (comparison illustrated in Fig. 4.3:25), and that are going to be walked around later on:

- **Truly random debugging starting point (random stepping)** → Any of the *"Macros"* may be started independently of all others. In classical systems, there indeed exists the possibility of the "set PC here" option, but it is mostly up to the user to know what he is doing, to avoid related execution incoherence problems.
- **Easier back-stepping intrinsically possible** → Because of the used *"Macros"* coding, it is easy to implement debugging loggers allowing for back-stepping. Because *"Macros-Sequences"* are their own source-code at the hardware level, no high-level source-code vs. assembly-code correspondence needs to be made, as it is absolutely necessary in classical systems.
- **Overall code sequencing non-critical** → Because any *"Macro"* could theoretically be executed at any time, the originally established *"Macros-Sequence"* loses its strict "sequential" meaning, thus theoretically any sequence could be implemented. This is true only if the overlying algorithm is 100% "data-flow" based/driven. In such a case and given that the longest time to execute the entire *"Macros-Sequence"* in any order would be smaller than necessary by the real-time constraints, then this would relax de strict sequencing problems observed in classical systems.
- **Theoretical 1 CPU per operation possibility** → Any *"Macro"* may be executed at any given time and in any sequence, without disrupting those isolated calculations. It would even thus be possible to therefore use 1 CPU per *"Macro"*, where this CPU would always be processing this same *"Macro"* over and over again. Such an hypothesis would not be feasible with current technology, but it opens perhaps interesting perspectives, further discussed in Chapter 6.

"Random" processing also implies that each *"Macro"* is only a single machine-instruction, therefore not optimizable, eliminating that source of pitfalls and bugs when developing.



```
        MOV     [R0+#0CH],R4
        MOV     [R0+#0EH],R5
        ?LINE   178
        ADDB    RL6,#01h
_10:
        MOVBZ   R4,RL6
        CMP     R4,_p22_cnt
        JMPR    cc_ULT,_9
; BOOSTPRESS.c  180
; BOOSTPRESS.c  181             p22_mp = div_U32U16_U16 (p22avg_sum, p22_cnt);
        ?LINE   181
        MOV     R15,_p22_cnt
        CMP     R15,#00h
        JMPR    cc_ULE,_11
        MOV     R4,[R0+#0CH]
        MOV     R5,[R0+#0EH]
        MOV     R10,R15
        MOV     R11,#00h
        CALLS   SEG __udil,__udil
        MOV     [R0+#04H],R4
        MOV     [R0+#06H],R5
        SUB     R4,#0FFFFh
        SUBC    R5,#00h
        JMPR    cc_ULE,_58
_59:
        MOV     R12,#0FFFFh
        JMPR    cc_UC,_60
_58:
        MOV     R12,[R0+#04H]
_60:
        JMPR    cc_UC,_12
_11:
        MOV     R12,#0FFFFh
_12:
        MOV     _p22_mp,R12
; BOOSTPRESS.c  182
; BOOSTPRESS.c  183             if(++p22avg_c >= p22_cnt)
        ?LINE   183
        SUBB    _p22avg_c,ONES
        MOVB    RL4,_p22avg_c
        MOVBZ   R4,RL4
```

eventually, previous temporary results contained in registersare used in next instructions
previous FDEF operation
division operation preparations
division
debugger???
division operation pos-checks and storage
next FDEF operation
debugger!
$1
$2
$3
$4
$5
$6
$7
$8
$9

Fig. 4.3:25 – Left: debugging a classical Assembly division; Right: equivalent isolated division "Macro"

Standard systems behave just like the top of Fig. 4.3:26 where the processing mechanisms all lie inside the program execution flow. On the other hand (bottom of Fig. 4.3:26), the work developed herein avoids all this influence and places the processing mechanism outside the program flow. While standard systems share the same processor among all instructions and thus also share the execution traces, influences and even bugs, the work developed herein allows each *"Macro"* to receive a "brand new processor" each time each one of them is executed (each *"Macro"* is therefore executed as inside an all-around protected "execution cage"). Refer to [Att. 21] for more details about "random processing" systems and related research.

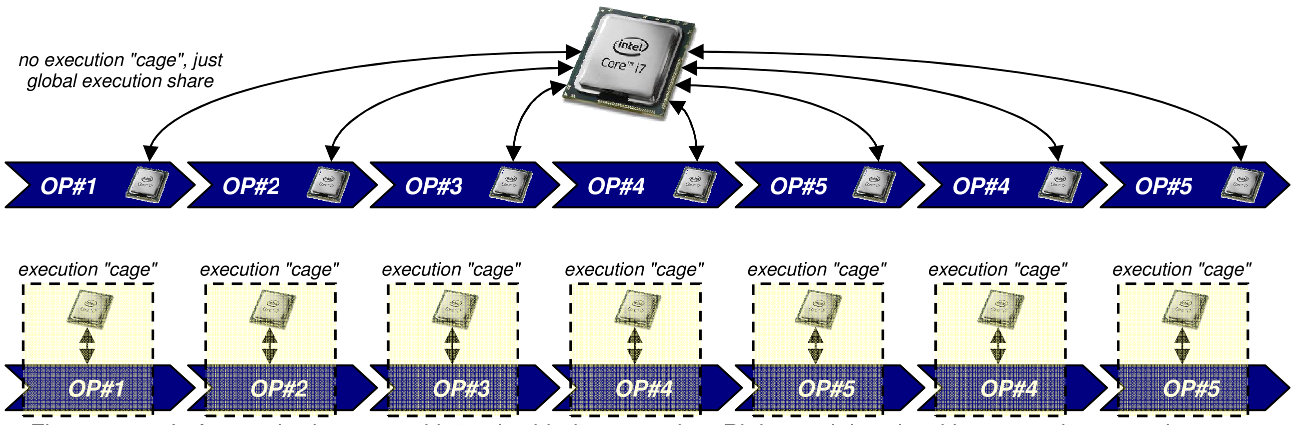


Fig. 4.3:26 – Left: standard system with "embedded" processing; Right: work herein with "external" processing

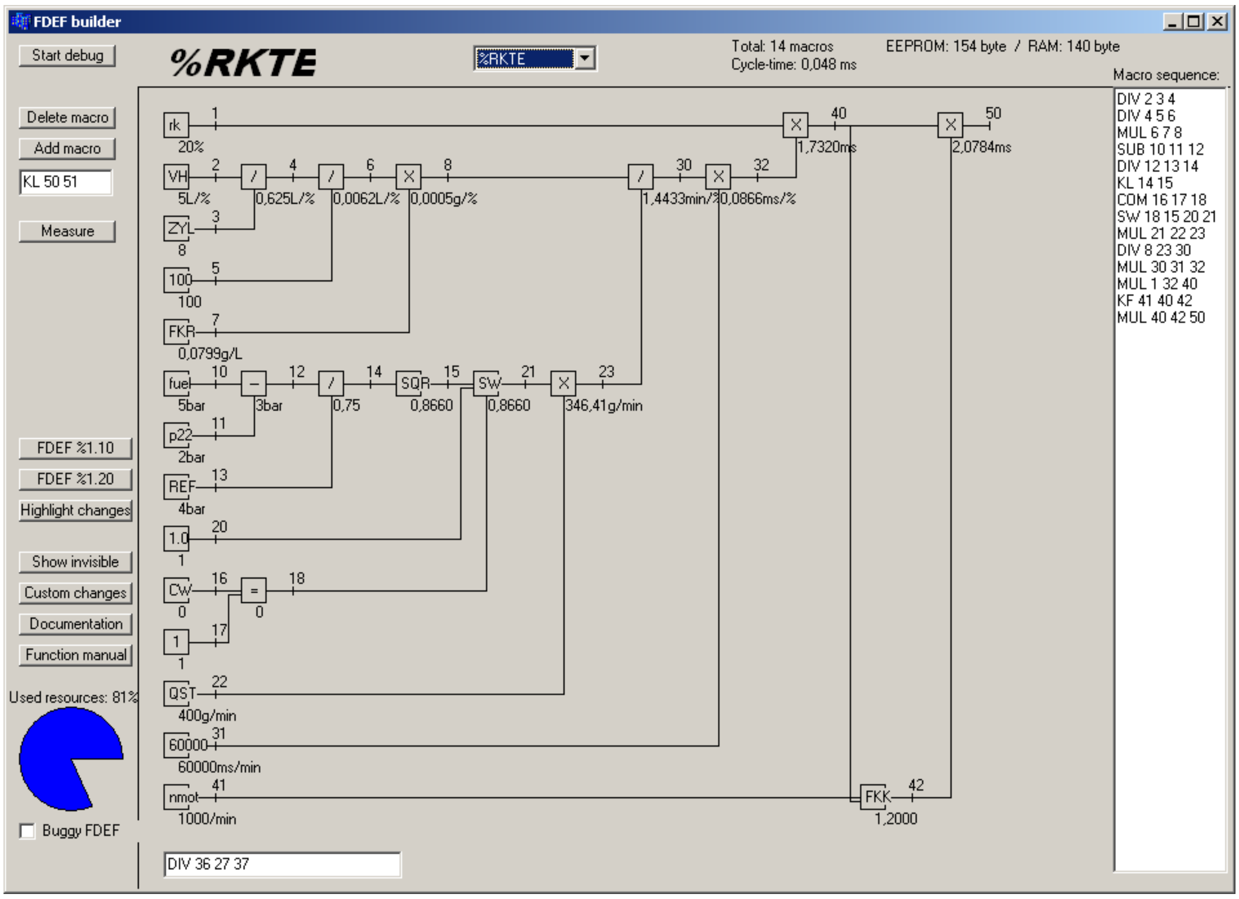
FINAL NOTE: Classical processing never guarantees the exact outcome of the same instruction executed over and over again, since there are many side-effects and gimmicks messing with them. In fact, one instruction that always executed correctly may disrupt due to some effect originated on a completely different and distant spot in the code.

From the statements above, ***PITFALL 0#24*** (1/2 – Branch Prediction), ***PITFALL 0#25*** (Superscalar Execution), ***PITFALL 0#19*** (Stacks), ***PITFALL 0#20*** (Pipelines), ***PITFALL 0#21*** (1/2 – Flags), ***PITFALL 0#22*** (Registers), ***PITFALL 0#27*** (Register Renaming), ***PITFALL 0#12*** (Code Optimization/Compression) and ***PITFALL C#30*** (MemoryStructure) are effectively eliminated through non-usage of optimizations or accelerations (just plain and straightforward execution of *"Macros"*), as well as through guaranteed execution correctness and, last but not least, the common memory pool where all *"Nodes" reside.*

4.4 A first glimpse of the core-ideas at work

[Emphasis: quick demonstration of the idea]

A first glimpse (Fig. 4.4:27) of the possibilities of the new idea can be obtained from a rudimental *"Macro-Programming"* tool developed early in the project to prove some of the base concepts. Details on this work can be further understood in appendix [Att. 22].



Total: 14 macros
Cycle-time: 0,048 ms
EEPROM: 154 byte / RAM: 140 byte

Macro sequence:

```
DIV 2 3 4
DIV 4 5 6
MUL 6 7 8
SUB 10 11 12
DIV 12 13 14
KL 14 15
COM 16 17 18
SW 18 15 20 21
MUL 21 22 23
DIV 8 23 30
MUL 30 31 32
MUL 1 32 40
KF 41 40 42
MUL 40 42 50
```

Fig. 4.4:27 – Implemented "Macro" types in the iEditor's toolbar and for use in building FDEFs

The next sections are going to expose these *"Macros"* embedded inside the final version of a complete development environment with all the features that will radically modify the way developers and other users interact with automotive systems. Along with the theoretical and practical details, corresponding illustrations show the real implementation in this editor herein called *"Intelligent Editor"* or *"iEditor"* in short. It is called *"Intelligent"* because it not just allows to design the FDEFs, but also takes the responsibility of automating fundamental manual tasks of current tools. This *"iEditor"* was programmed with Microsoft Visual Studio 2005 [292] in the C# [71] high-level .NET-managed language.

4.5 *"Macros"* & *"Nodes"* Morphology

[Emphasis: the atomic entities, innovative characteristics]

"Macros" are composed of several sub-components and there is a variety of different ones. *"Macros"* directly relate to operations and *"Nodes"* directly relate to the operated variables. Note that not all of the *"Macros"* and *"Nodes"* herein exposed were immediately implemented into the *"iEditor"*, but just the really needed for the final demonstration on a gasoline internal combustion engine at the end of this thesis' work. Also, these are still intermediate results, while the final *"iEditor"* may suffer slight changes, cuts or appendixes. As a final note, the fact that every *"Macro"* must be able to address any *"Node"* at any time, crystallized into the need of a non-stack based architecture, since the stack-based architectures only allow the operations to operate on the top-most values on the stack.

4.5.1 "Macros"

[Emphasis: the atomic operational entities]

The *"Macros"* are the central building blocks of this work herein. Some characteristics make them unique when compared to legacy and currently used programming blocks.

4.5.1.1 Basic Types

[Emphasis: essential programming blocks]

There are several *"Macro"* types, which make up the minimum needed arithmetic, logic, conditional and other additional operations that make up the automotive program flow. Tab. [Att. 23]:102 presented in [Att. 23] shows most *"Macros"* used in automotive projects. They were organized according to type of end-result or operation and corresponding code is included in mixed "Pascal" and "C" syntax for reference. Auxiliary state values in parenthesis mean internal temporary values inside the *"Macros-Processor"*. The adopted colouring is also going to be adopted in the final editor at the end of this work. Furthermore, these *"Macros"* are specifically designed for the specific needs in automotive functionalities, while this kind of instruction-set specialisation [198] [199] is nothing new.

Fig. 4.5:28 illustrates *"Macros"* on toolbars inside the *"iEditor"* as usable visual elements to build FDEFs, while Fig. 4.5:29 shows some examples of added *"Macros"* (this will be further expanded as soon as the "*iEditor"* is later explained). These *"Macros"* will be added to an FDEF as indivisible and complete entities, much in the same manner as in [5] [6] [7].

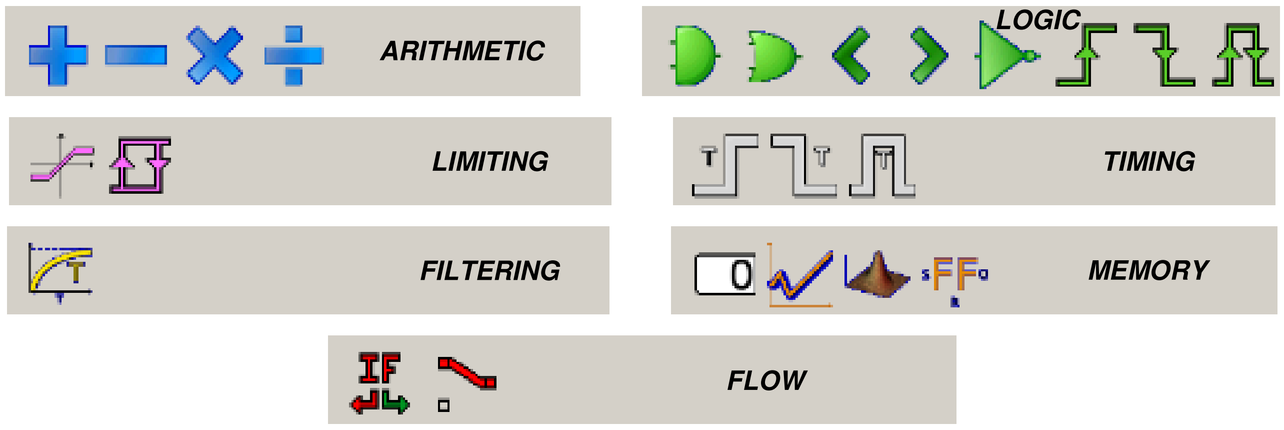


Fig. 4.5:28 – Implemented "Macro" types in the iEditor's toolbar and for use in building FDEFs

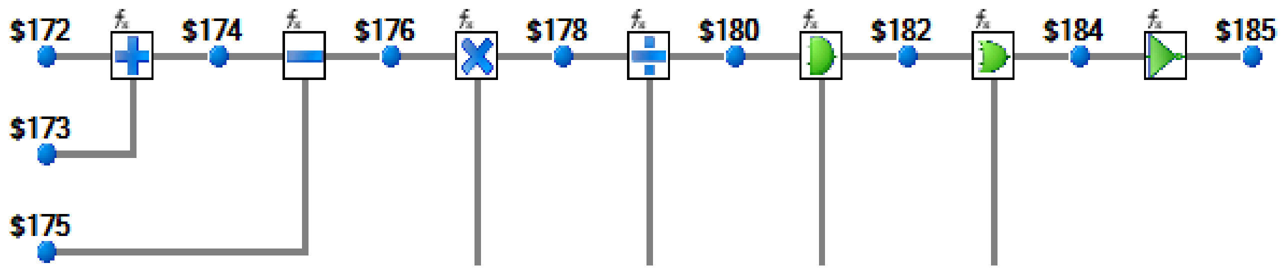
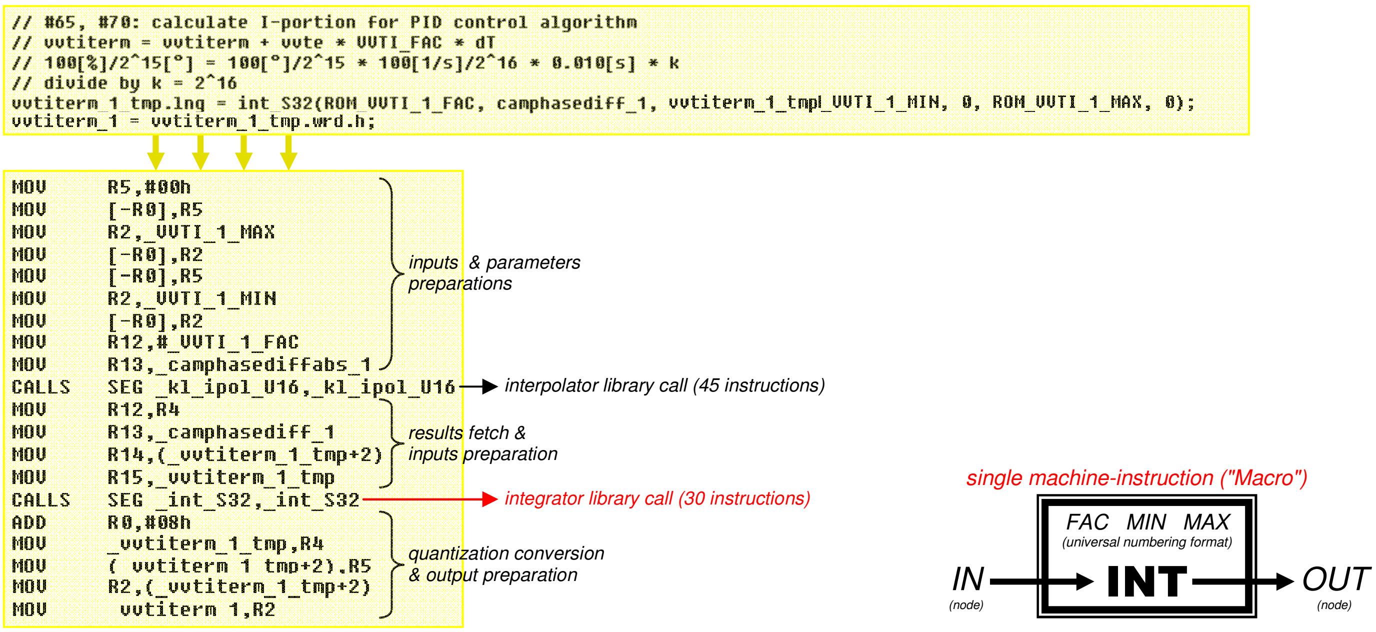


Fig. 4.5:29 – Sample "Macros" as they will appear on the user-interface if the "iEditor"

As in virtually all CPUs, three global abstract families define the whole of the available machine-instructions: data-transfer/memory, flow-control and arithmetic/logic instructions. But it must be noted that contrary to these generally very low-level machine-operations, these *"Macros"* are operations tuned for the automotive functionality design. In other words, instead of many general low-level operations, these *"Macros"* are only those necessary for automotive FDEFs and many are relatively high-level operations such as complete filters and delays. Just as in [374], these are complete graphical elements. This automatically means that although the lowest-level ones may be directly executed by the ALU (Arithmetic-Logic Unit) inside the *"Macros-Processor"*, the more high-level ones must be fed into the SEQUENCER that executed them through multiple ALU call steps. It must be pointed out at this stage, that there is no desire to implement something ideal regarding these *"Macros"*, regarding completeness of coverage. It is solely intended to demonstrate that an automotive system can be successfully controlled by such processing operations.

Data-transfer and memory-operations are intrinsic to each *"Macro"* since it contains all the responsibility of getting the inputs and putting the results out onto the central memory *"GIMy"* explained later on. There is not special instruction to handle memory accesses whatsoever, greatly simplifying the whole development issue, since there are no constraints, barriers or pitfalls to watch for, during programming. Side-operations are completely eliminated by design, since each *"Macro"* is complete by itself as already explained earlier. This feature applies to variable *"Nodes"* in the central *"GIMy"* only. Additionally to this feature, fixed parameters such as those used in filters (time-constant), parameters/lines/maps (constants) and any other *"Macro"* needing permanent data, are also intrinsically contained in the corresponding *"Macros"* so that even here any extra data manipulation and memory access side-operations are thus eliminated from normal development flow.

Fig. 4.5:30 shows Assembly result of a statement where a limited multiplication is done. Notice how an apparently simple statement unfolds into memory-management, quantization-conversion, interpolation and integrator library calls. Other statements produce even more "memory-management" and other types of Assembly instructions. The really useful/desired action is marked red. In the case of the *"Macro"*, only one single machine-instruction reaches the *"Macros-Processor"*, while only the input and the output are externally fetched and stored, respectively. Parameters are inside the *"Macro"* instruction itself.



```
// #65, #70: calculate I-portion for PID control algorithm
// vvtiterm = vvtiterm + vvte * VVTI_FAC * dT
// 100[%]/2^15[°] = 100[°]/2^15 * 100[1/s]/2^16 * 0.010[s] * k
// divide by k = 2^16
vvtiterm_1_tmp.lng = int_S32(ROM_VVTI_1_FAC, camphasediff_1, vvtiterm_1_tmpl_VVTI_1_MIN, 0, ROM_VVTI_1_MAX, 0);
vvtiterm_1 = vvtiterm_1_tmp.wrd.h;
```

```
MOV      R5,#00h
MOV      [-R0],R5
MOV      R2,_VVTI_1_MAX
MOV      [-R0],R2
MOV      [-R0],R5
MOV      R2,_VVTI_1_MIN
MOV      [-R0],R2
MOV      R12,#_VVTI_1_FAC
MOV      R13,_camphasediffabs_1
CALLS    SEG _kl_ipol_U16,_kl_ipol_U16
MOV      R12,R4
MOV      R13,_camphasediff_1
MOV      R14,(_vvtiterm_1_tmp+2)
MOV      R15,_vvtiterm_1_tmp
CALLS    SEG _int_S32,_int_S32
ADD      R0,#08h
MOV      _vvtiterm_1_tmp,R4
MOV      (_vvtiterm_1_tmp+2),R5
MOV      R2,(_vvtiterm_1_tmp+2)
MOV      _vvtiterm_1,R2
```

inputs & parameters preparations
interpolator library call (45 instructions)
results fetch & inputs preparation
integrator library call (30 instructions)
quantization conversion & output preparation
single machine-instruction ("Macro")
FAC MIN MAX
(universal numbering format)
IN
(node)
INT
OUT
(node)

Fig. 4.5:30 – Data/parameters handling – Top-Left: classical external operations; Right: intrinsic to the "Macros".

Fig. 4.5:31 specifically shows the insides of a *"Macro"* instruction, with its custom "BCDP" decimal numbering format (explained later on herein), its homogeneous 32bit memory layout and all its optional/possible sub-entities. 3-axis (3D) parameter map *"Macros"* are possible but not considered here, for illustration simplicity. Multiplexers have a 3rd input (selection) while demultiplexers have two outputs instead of one. Notice how everything is aligned by double-words (32bit). This greatly simplifies the *"FDEF-Processor"* explained later on. *"Macro Modifiers"* are mostly used in the *"iEditor"* itself only, but are also downloaded into the hardware for the "Reverse-Engineering" feature to work correctly. These are detailed in Fig. 4.5:32 with the number of needed bits inside parenthesis.

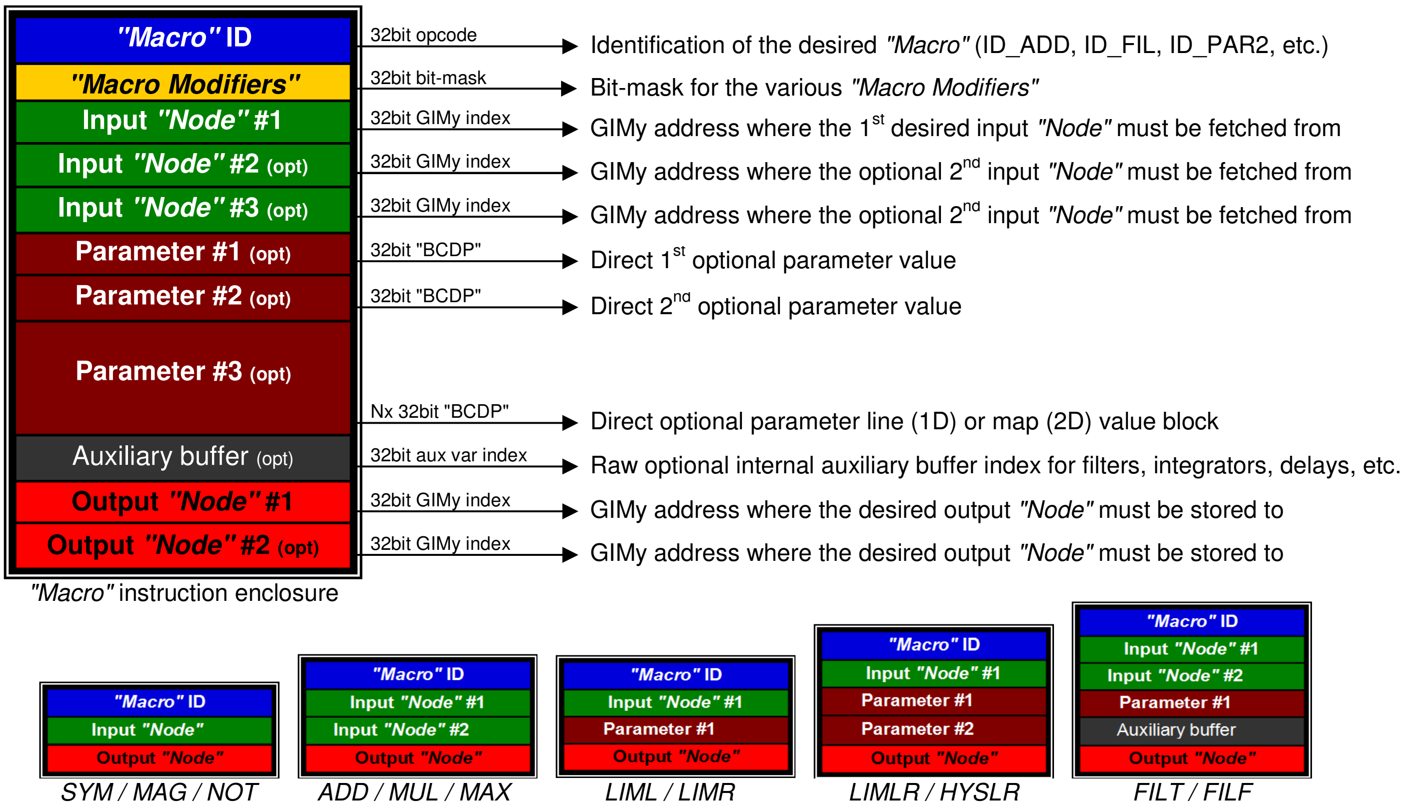


Fig. 4.5:31 – More details about the "Macros" internal structure in non-volatile program memory ("Macros" FLASH).

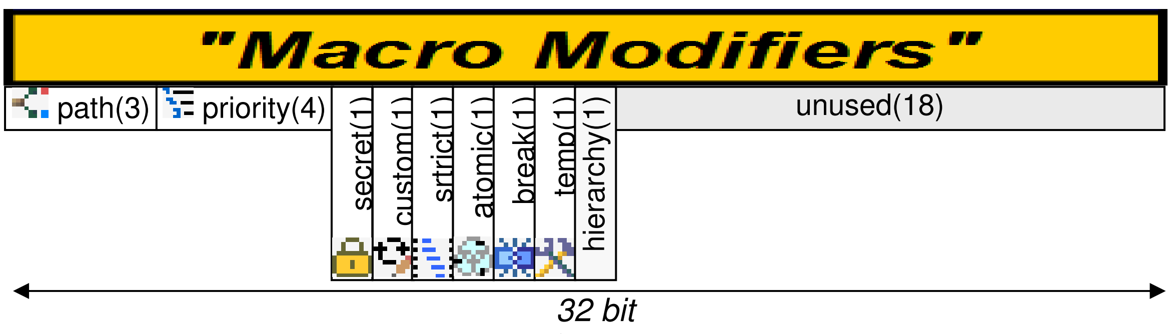


Fig. 4.5:32 – "Macro Modifiers" inside the "Macro" storage

The "hierarchy bit" is reserved for future implementation of FDEF hierarchies. The inputs to a *"Macro"* have the very special ability of being able to accommodate three different types of input origins, not only the above shown *"Nodes"*, but also non-*"Node"* values, through the use of 2 control-bits inside the input's 32bit pattern (Fig. 4.5:33). Note that these special cases are not to be confused with the above shown parameters, which are really intrinsic parameters needed for the particular *"Macro"* to operate correctly such as line/map points or time-constants. Special cases described here as parameters/constants generally refer to values used as inputs the *"Macros"* processes.

- ***"00"*** → Normal *"Node"* variable value, fetched from the external *"GIMy"*.
- ***"01"*** → Fixed/calibrateable parameter encapsulated within the fetched *"Macro"* itself, inside FLASH. This value is used directly without any further fetching from anywhere.
- ***"10"*** → Constant/unchangeable value encapsulated within the fetched *"Macro"* itself, inside FLASH. This value is used directly without any further fetching from anywhere. The difference to the previous parameter is that the user will be unable to change it during operation. This type of values exists also to distinguish engine plant parameters and system constants.
- ***"11"*** → Reserved / not used.

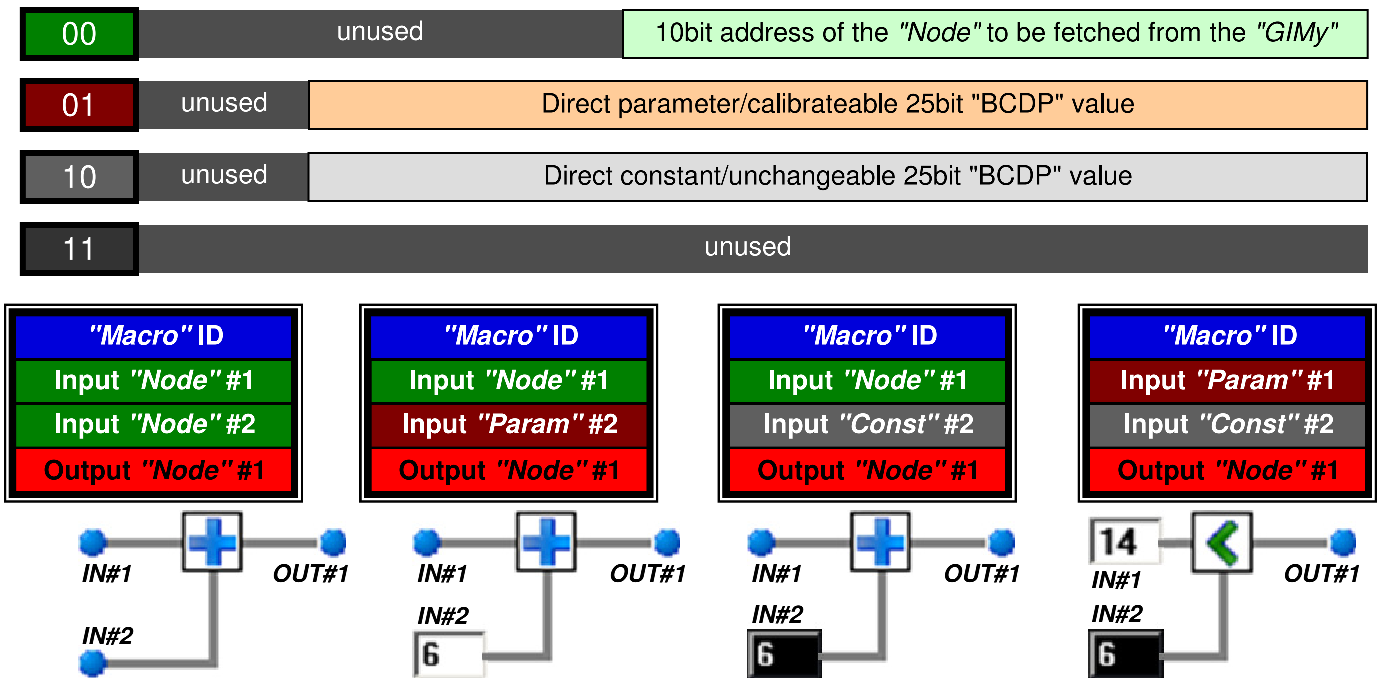


Fig. 4.5:33 – Top: the three possible formats/meanings for inputs; Bottom: examples of different input combinations.

Fig. 4.5:34 shows an example with a standard FDEF taken from Motorsports. A few operations get translated into a bloated "C" source-file, which is then translated into an

even more bloated Assembly file. This last file is then finally converted into machine-code to be processed inside the processor, at last. Fig. 4.5:35 shows an anticipation of an FDEF with similar operations count, along with its final 1-to-1 machine-code visualization.

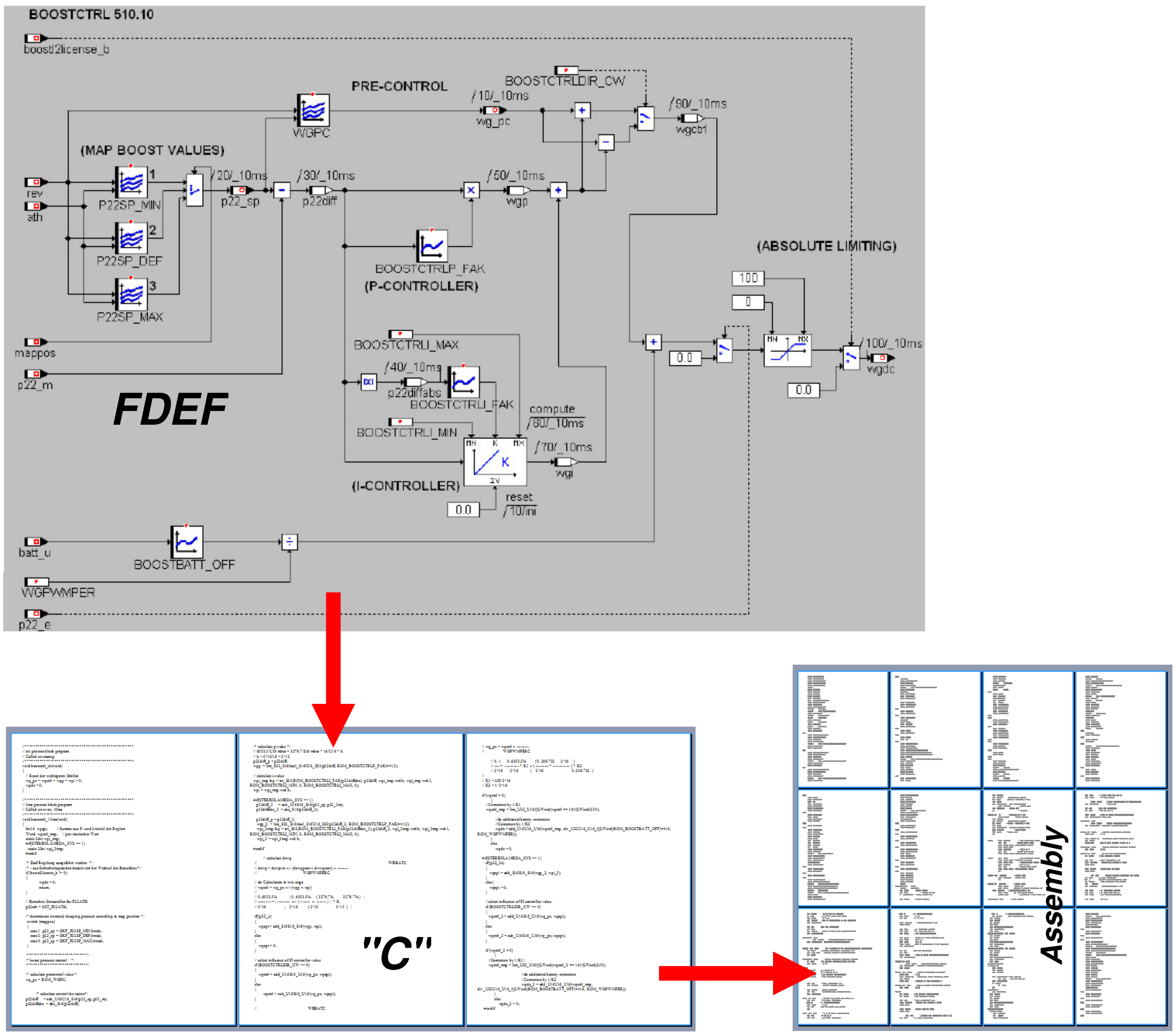
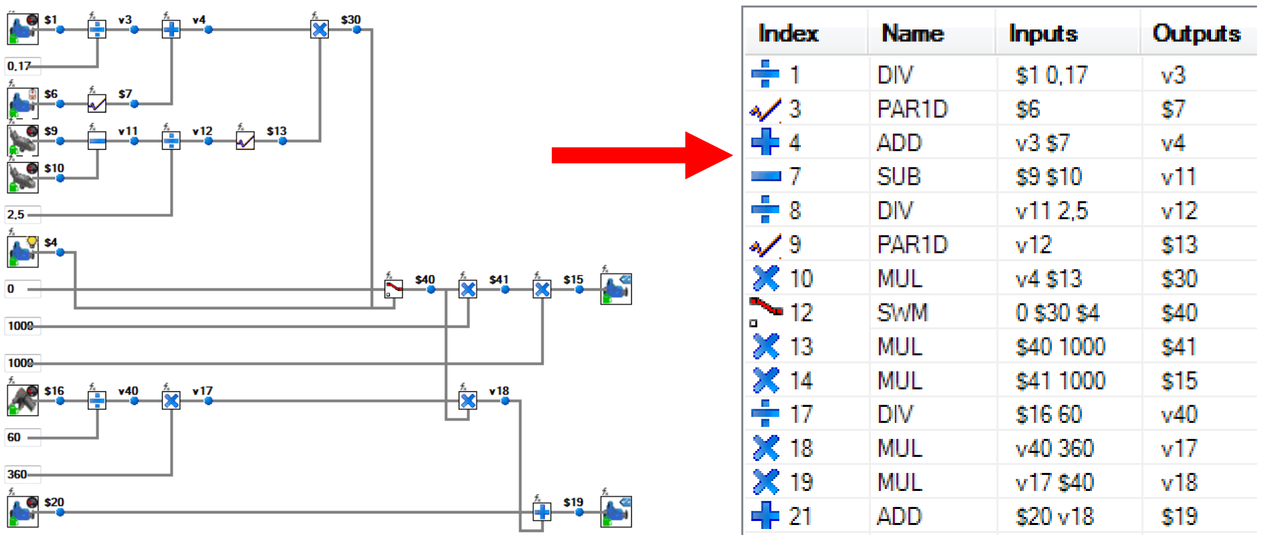


Fig. 4.5:34 – Top: original MS4 FDEF; Bottom-left: corresponding "C" code; Bottom-right: Assembly code.



Index	Name	Inputs	Outputs
1	DIV	$1 0,17	v3
3	PAR1D	$6	$7
4	ADD	v3 $7	v4
7	SUB	$9 $10	v11
8	DIV	v11 2,5	v12
9	PAR1D	v12	$13
10	MUL	v4 $13	$30
12	SWM	0 $30 $4	$40
13	MUL	$40 1000	$41
14	MUL	$41 1000	$15
17	DIV	$16 60	v40
18	MUL	v40 360	v17
19	MUL	v17 $40	v18
21	ADD	$20 v18	$19

Fig. 4.5:35 – Left: injection output FDEF; Right: equivalent "Macros-Sequence" (same count).

The *"Macros"* are thus a highest-level specific automotive "source-less" control language, since they are directly used to describe automotive control programs. They are also directly mappable to machine-code level, which is quite different from virtually all currently used languages. They could be applied to virtually any other environment that involves control similar mechanisms. For additional details and comparisons with other systems as well as research, refer to [Att. 23].

4.5.1.2 Flow-Control & Signal-Switching

[Emphasis: essential conditional control]

A very special case of *"Macros"* is that of those related to flow-control. Conditional operations are reduced to the "IF", where the input of this conditional *"Macro"* then modulates how it reacts. In other words, instead of having different conditional instructions for "greater than", "lower than", "equal", "different", etc., in automotive programming most usually these detections are done beforehand and the result used as an "yes/no" input for subsequent "IF" processing. Another example of special high-level operations being used here, is the existence of "multiplexers" and "demultiplexers" as single self-contained *"Macros"*. Although apparently unrelated at first glance, these can also be used for conditional signal flow, thereby simplifying flow-control aspects of programming. For more details about conditional and signal-switching operations in languages, refer to [Att. 24]

The "IF" *"Macro"* implemented serves only for some extreme case where the usual data-flow usage of "MUX/DEMUX" operations is not appropriate, but should be avoided, bearing in mind that "MUX/DEMUX" operations are alternatives to virtually all cases of "conditional" signal flow needs. Fig. 4.5:36 shows examples of how to convert a control-flow block into a pure data-flow one. It is not difficult to imagine how to do this for larger blocks as well.

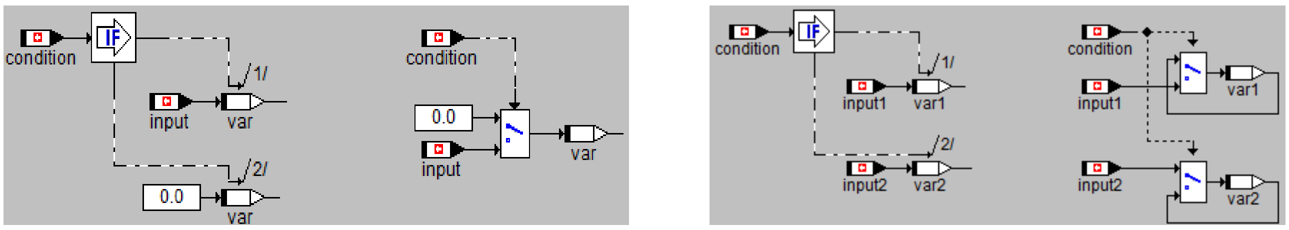


Fig. 4.5:36 – "IF-THEN-ELSE" constructs and their equivalent pure data-flow counterparts.

There was no need in this particular work, to implement loops, but the procedure would be very similar to the "IFs", also with Boolean *"Nodes"* as inputs for "WHILE-DOs" or "REPEAT-UNTILs". As with "IFs", these loops would also graphically include the correspondingly encompassed *"Macros"*, therefore also naturally combining the rare control-flow needs inside automotive data-flow programs [8].

FINAL NOTE: Classical ways of automotive programming usually place data separated from the code, so that pointers or handles to that data are necessary. This potentially causes all possible pointer-related problems. Also, conditional explicit "IF-THEN-ELSE" are not necessary for virtually all the automotive functionality range.

From the statements above, ***PITFALL 0#40*** *(Pre-Processing/Macro-Expansion)*, ***PITFALL C#23*** *(Pointers & Handles)*, ***PITFALL 0#28*** (Side-Operations), ***PITFALL 0#24*** *(1/2 – Branch Prediction)* and ***PITFALL C#80*** *(Long Learning Curve)* are thus effectively eliminated through the non-necessity for using any pointers or handles, nor "IFs" in a data-flow based language.

4.5.1.3 *"Macro"* Modifiers

[Emphasis: operational effects in addition to the functional "Macro" behaviour]

By letting each *"Macro"* having an associated bit-field that contains flags indicating special features or behaviours, one can very easily activate some visual and even functional effects to each *"Macro"* individually. Note that these modifiers are fully visually implemented. These so-called *"Modifiers"* may be used by programmers as well as customers and are manipulated directly on the graphical FDEF, without the need to switch to any extra tool nor is it necessary to turn away from the FDEF. These *"Macro Modifiers"* are placed around each corresponding *"Macro"* for maximum closeness and visibility purposes. Fig. 4.5:37 illustrates the graphical looks of this concept in the *"iEditor"*, along with the user-interface component to manipulate those *"Modifiers"*, as well as the input and output *"Nodes"*. For better comparison, refer to [Att. 25].

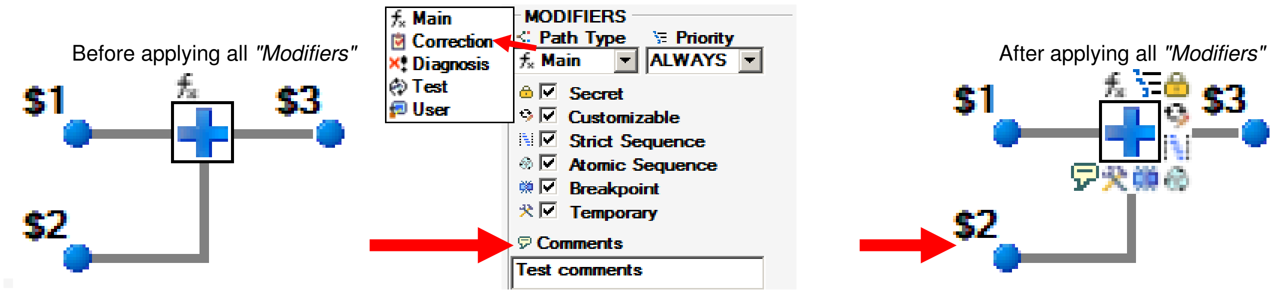


Fig. 4.5:37 – "Modifiers" around the "Macro" to further enhance an FDEF. "Macro Editor" inside the "iEditor".

The *"Macro-Modifiers"* are described next. Some of them are even enabled for filtering actions on the user-interface through a filtering panel illustrated in Fig. 4.5:38, thus allowing the developer to mask out some secondary *"Macros"*. This enables context sensitive FDEF viewing, without all the clutter presented by current tools. These modifiers present an additional processing influence onto the original *"Macros-Sequence"* and in the context of the FDEF itself, so that the elementary *"Macros-Processor"* will be expanded to the more encompassing term *"FDEF-Processor"* explained later on.

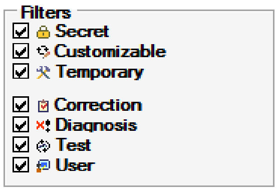


Fig. 4.5:38 – Filtering panel that allows the developer to selectively mask out certain "Macros"

- **"PATH TYPE"** –categorizes single *"Macros"* as belonging to the main, correction, diagnosis or custom processing paths of an FDEF, for better visual effect and to be eventually highlighted. Fig. 4.5:39 illustrates this modifier where the injection quantity corrections due to oil temperature and air temperature are filtered out of the graphical view.
- **"PRIORITY"** – intended to guarantee that unimportant *"Macros"* are executed as often as strictly necessary, thus liberating execution capacity to those more important ones. All non-prioritized *"Macros"* will keep executing once in every *"Macros-Cycle"*. When explaining *"LIVE-Priorities"* (on-the-fly priority propagation) Fig. [Att. 41]:144 will then show an example of *"Macros"* priority setting inside an FDEF.
- **"SECRET"** – marks single *"Macros"* to be hidden from customers, simplifying the process of having a "FULL" Function Manual and a "PARTIAL" Function Manual in an automated way. There could also be different levels of security for different people. Fig. 4.5:40 illustrates this modifier where the fuel pump's manual activation possibility is filtered out of the graphical view.

- **“CUSTOMIZABLE”** – marks single *“Macros”* to be accessible for the customer to modify at will. This modifier is especially important for memory *“Macros”* where calibration tables may be customized. Fig. 4.5:42 illustrates this modifier where the injection quantity's correction maps over temperature may be modified by the customer, but where the main injection map (dependent upon engine-speed and throttle) cannot.
- **“STRICT SEQUENCE”** – forces a portion of the *"Macros-Sequence"* being executed in a strict sequence. This is useful for strictly algorithmic parts where the intermediate results may not be parallelized just as the rest, but have to be processed in that exact sequence, or results would be out-of-order and the algorithm would deliver wrong end-results when combining different processing paths.
- **“ATOMIC SEQUENCE”** –forces a certain amount of *"Macros"* to be executed without any *"GIMy"* interruption/update that might modify the values in the *"Nodes"* those *"Macros"* are manipulating in that sequence. *"Nodes"* may not be updated during an atomic sequence and update attempts are simply ignored by the *"GIMy"* until the atomic sequence ends. Fig. [Att. 49]:156 illustrates an usage example of an "atomic sequence" for two sensor inputs.
- **“BREAKPOINT”** – forces execution to stop at a given *"Macro"* so that values can be inspected more closely while also allowing execution stepping. The developer may also set breakpointing conditions so that a program interruption only occurs if certain conditions are met. Fig. 5.1:34 shows an usage example of breakpoints during debug.
- **“TEMPORARY”** – marks single *“Macros”* to be temporarily available. As soon as testing is completed they can be turned into definitive *“Macros”* or just be wiped out with one single command. Fig. 4.5:41 illustrates this modifier.
- **“COMMENTS”** – this special field allows anyone to write down notes on single *“Macros”* for later reference or explanation purposes. These comments may represent tasks, problems, notes and so on, eventually being included into the final user-manual for those FDEFs.

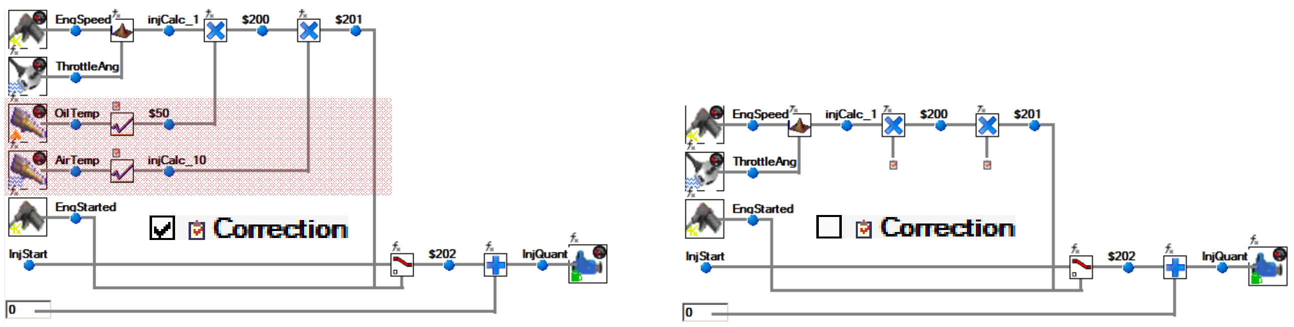


Fig. 4.5:39 – Left FDEF with main-path ( ) and corrections ( ). Right one only with main-path.

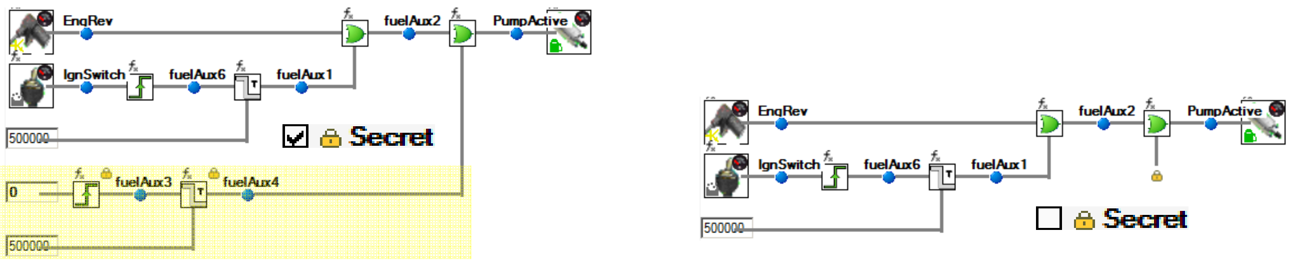


Fig. 4.5:40 – Left FDEF with secrets ( ). Right one with secrets masked out.

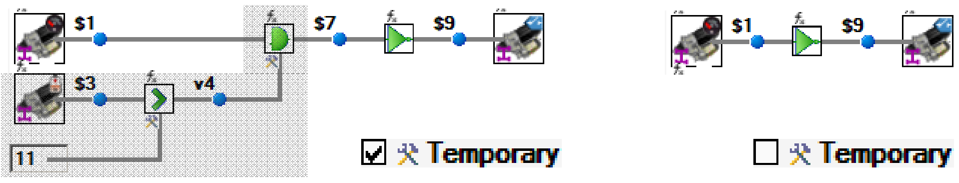


Fig. 4.5:41 – Left FDEF with temporary operations ( ). Right one with temporary operations masked out.

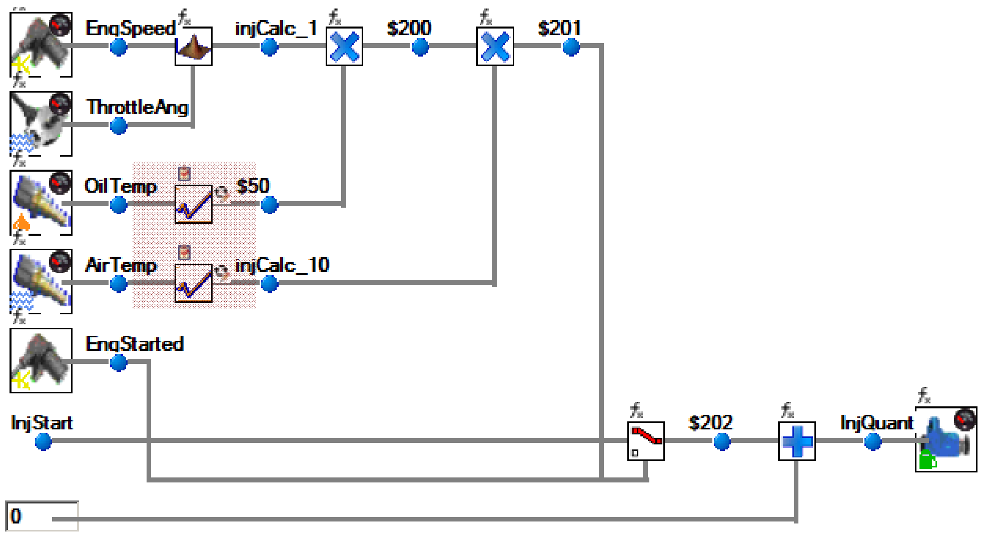


Fig. 4.5:42 – FDEF with customizable (⚙) and non-customizable maps

For some extra notes about *"Macros"* morphology and comparisons, refer to [Att. 26].

FINAL NOTE: Classical development environments demand huge effort and time in keeping all handling details in mind, while there are not many ways of getting secondary graphical details out of the way.
From the statements above, ***PITFALL H#51*** (1/2 – Graphical Cluttering) and ***PITFALL H#49*** (1/2 – Lengthy Turnarounds) are thus effectively eliminated through compact options design through positioning them on the exact spots where they are needed.

4.5.2 "Nodes"

[Emphasis: the memory entities]

Each *"Macro"* has one or more inputs, as seen previously. Normally, these inputs are *"Node"* addresses pointing to the central *"GIMy"*. These *"Nodes"* then contain values coming from peripherals, processed *"Macros"* from the *"FDEF-Processor"* and from any other entity manipulating *"Nodes"*.

4.5.2.1 *"Node"* Features

[Emphasis: encapsulated value entities]

All *"Nodes"* reside inside the *"GIMy"* where the *"Macros-Processor"* will fetch the respective *"Node"* values on a "on-need" basis. These *"Nodes"* come from peripherals and other *"Macros"* processing outputs and are deposited inside the *"GIMy"* for this "on-need" usage. These *"Nodes"* possess 32bit each, where a part is the custom "BCDP" decimal power numbering scheme (explained later on herein) and another part is composed by "control-bits" that control the way the *"Macros"* use those *"Nodes"*. As will be seen later in *"Live-Dragging"*, the necessary *"Nodes"* are taken care of by the also later shown *"iEditor"*, in that they will be automatically added as needed by the respectively added *"Macros"*. An internal counter also keeps those *"Nodes"* sequentially deposited inside the *"GIMy"*, therefore freeing the developer completely.

To maximize the utility of having this central *"GIMy"*, it would be best to have it featuring 100% multi-read and 100% multi-write capabilities (Fig. 4.5:43). In other words, any consumer would be able to simultaneously read any *"Node"* at any time, while any producer could be able to simultaneously write to different *"Nodes"* at any time. If more than one producer desires to write into the same *"Node"* at the same time, then the *"GIMy"* avoids fatal collision and data corruption through a simple arbitration scheme in that the writes are hard-wiredly priotized. But, as explained later on, a simple "single-assignment" strategy will be enforced throughout the automotive FDEFs developed in this work herein, so that this last arbitration scheme will be implemented only as safety measure to guarantee 100% correct operation of this *"GIMy"* at all times.

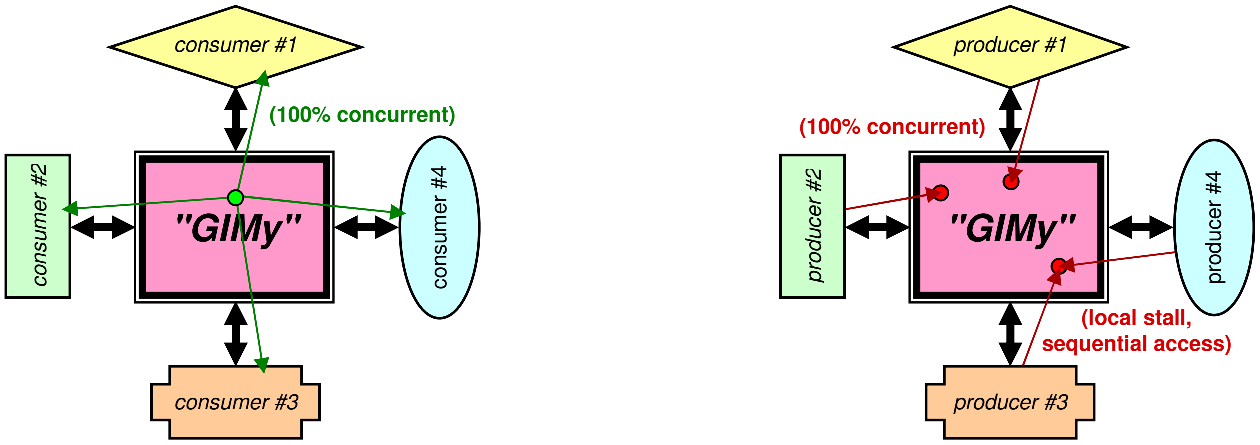


Fig. 4.5:43 – Multi-read and limited multi-write illustration of the "GIMy" containing the "Nodes".

Technically, these *"Nodes"* only manifest themselves as values and not variables, such that they are obviously "weakly" or "dynamically" typed, since there is no need to determine any specific type upon design-, download- and run-time. The herein used "BCDP" type inside the *"Nodes"* was chosen to be able to contain any value needed in an automotive system. The *"Nodes"* also contain the respective metric units that can be associated with root *"Nodes"* such as those coming from peripherals. The *"iEditor"* is then able to propagate these automatically throughout the entire FDEF hierarchy through some simple transformation rules. For some more notes about *"Node"* features, as well as comparisons with other languages, refer to [Att. 27]

4.5.2.2 *"Node"* Modifiers

Emphasis: operational effects in addition to the functional "Node" variable behavior]

"Nodes" also have modifiers that allow the FDEF developer and the customer to set some extra functionality and these are also manipulated directly graphically on the FDEF. Again, everything is directly and graphically done on the FDEF itself, which completely eliminates any turnaround overheads whatsoever! These *"Node Modifiers"* are placed around each corresponding *"Node"* for maximum closeness and visibility purposes. Fig. 4.5:44 illustrates the graphical looks of this concept in the *"iEditor"*, along with the user-interface component to manipulate those modifiers. These modifiers present an additional processing influence onto the original *"Macros-Sequence"* and in the context of the FDEF itself, so that the elementary *"Macros-Processor"* will be expanded to the more encompassing term *"FDEF-Processor"* explained later on. Possible *"Node"* modifiers are:

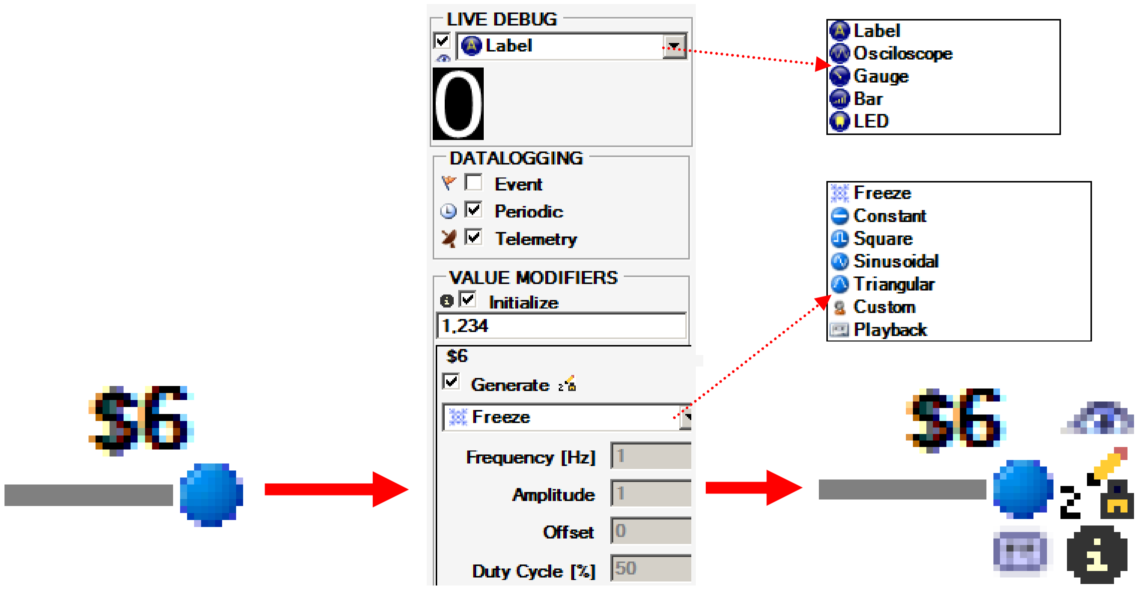


Fig. 4.5:44 – "Modifiers" around the "Node" to further enhance an FDEF. "Node Editor" inside the "iEditor".

- 7.63864 **"CHANGED"** – flashes a visually displayed *"Node"* red, therefore allowing anyone watching it to have their attention caught. This feature applies to any visual *"Node"* display mode. It does not need hardware participation, since the *"iEditor"* simply monitors for *"Nodes"* changing itself. Nevertheless, the herein reserved but unused "changed" bit inside each *"Node"* structure, could also be used to signalise a more precise change event by strict *"GIMy"* checks.
- **"MONITOR"** – allows the values of single *"Nodes"* to be watched through a label, oscilloscope, gauge, bar or LED, directly on the graphical FDEF *"Node"*. This feature has ultimate importance when debugging, testing or simply inspecting one or more FDEFs at work. Fig. 4.5:45 illustrates this modifier where the injection quantity *"Node"* value may be inspected in multiple visual forms. All those forms are additionally zoomable by double-clicking on them) and their internal properties may be freely adjusted, as illustrated in Fig. 4.5:46. An additionally important feature inside the *"GIMy"* allows for the *"iEditor"* to show that a *"Node"* has changed its value through colour change (flashes of red on the monitor and signal path). This allows the developer or user to detect and see what FDEF paths are currently actively changing values. Fig. 4.5:45 also illustrates this red flashing.
- **"DATALOGGING"** – activates datalogging for single *"Nodes"* through the internal logging structure. This feature has ultimate importance when following one or more FDEFs during debugging, testing or even racing, and is taken care of by the "Log-Manager" architectural block detailed later on herein. This feature requires hardware participation in that the *"Log Manager"* checks for the corresponding bits to copy *"Nodes"* into local storage.
- **"EXTERNAL WIRELESS DATALOGGING"** – activates datalogging of *"Nodes"* to the external wireless Datalogger if present. This distinction allows to select *"Nodes"* for internal logging only, *"Nodes"* for external logging only or both possibilities simultaneously. This feature requires hardware participation in that the *"Log Manager"* checks for the corresponding bits to copy *"Nodes"* into the transmitter, for remote/external storage.
- **"TELEMETRY"** – activates transmission of *"Nodes"* wirelessly to the racing team box/pitlane, through a 3G or any other RF channel over the external Datalogger explained later on. This distinction further allows to select among internally, externally and telemetry logged *"Nodes"*. This feature requires hardware participation in that the *"Log Manager"* checks for the corresponding bits to copy *"Nodes"* into the transmitter, for remote/external viewing and possible storage. Although left unused herein, this bit was preliminarily reserved inside the *"Nodes"* structure for future enhancement.
- **"INITIALIZATION"** – allows to fill single *"Nodes"* with specific values upon system start-up. This feature is important for setting initial values on *"Nodes"* that should not initiate with their value zeroed. This feature requires hardware participation in that the *"Log Manager"* checks for the corresponding bits to copy initialisation values into *"Nodes"* at start-up. Although left unused herein, this bit was preliminarily reserved inside the *"Nodes"* structure for future enhancement.

- **"GENERATE"** – allows direct setting of values pr pre-defined sequences of values to single *"Nodes"*. It allows debugging, testing or simply inspecting an FDEF at work. Values can be applied on any *"Nodes"*, either in the middle of FDEFs, on those that come directly from sensors (thus overriding input peripherals *"Nodes"*) and on *"Nodes"* that directly go to actuators (thus overriding output peripherals *"Nodes"*). The simplest form of value generation is freezing a *"Node"* on its current value, allowing the developer to mask out all previous calculations up to that *"Node"*. It is also allows to change that value to a desired constant. Forces sinusoidal, square and other signal forms, or even plays back ( ) a previously logged sequence of values, or plays back user-specific value sequences ( ). Fig. 4.5:47 illustrates this modifier where various waves are generated at a particular *"Node"*, all with the same frequency. This feature can be seen as a signal "injector" or "overrider". This feature requires hardware participation in that the *"GIMy"* checks for the corresponding bits to prohibit *"Nodes"* from being overwritten, therefore maintaining their current/last values.

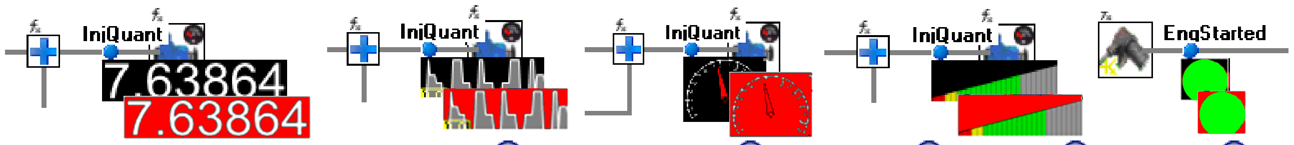


Fig. 4.5:45 – "Node" inspection: with label ( ), with oscilloscope ( ), with gauge ( ), with bar ( ), with LED ( )

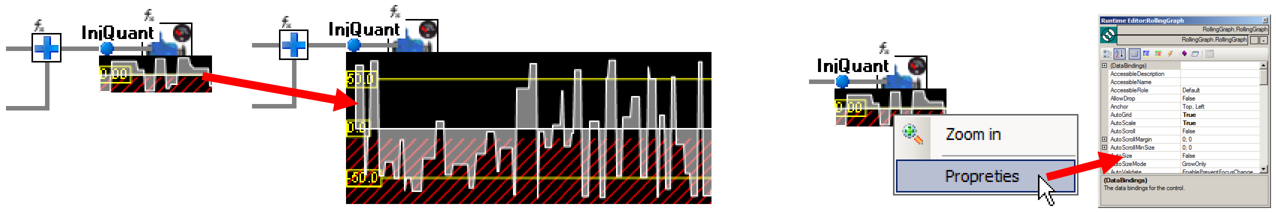


Fig. 4.5:46 – Example of "Node" oscilloscope zooming and properties' adjustment

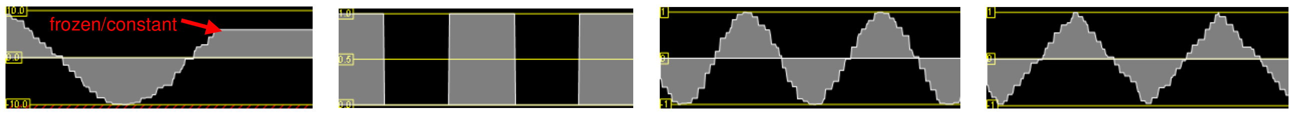


Fig. 4.5:47 – Example of "Node" values' generation: frozen/const (/), square ( ), sinusoidal ( ), triangular ( )

A future "executed *Macros*" feature, where the *"iEditor"* could easily display all the really executed *"Macros"* paths and the ones not being executed for some reason, could be implemented by using the herein unused but reserved "updated" bit from the corresponding output *"Nodes"* below. Since most probably all *"Macros"* will always be executed within short time-spans (also as part of the "anti-jitter" measures taken in this work), this feature was not further pursued/needed at this moment.

The final *"Nodes"* internal structure used inside the *"GIMy" is depicted* in Fig. 4.5:48, where the *"BCDP"* coexists only with the modifier bit "freeze"(***F***) for the "GENERATE" feature. The other datalogging/external-logging/telemetry bits (D/E/T), initialisation (I), changed (C) and updated (U) *"Node Modifiers"* are directly managed and checked inside the *"Log Manager"*, because it turns out to be much more efficient to do it there in practice. Nevertheless, they are part of the theoretical *"Nodes"* format and are thus included herein for better understanding of what actions and results really affect the *"Nodes"* altogether.

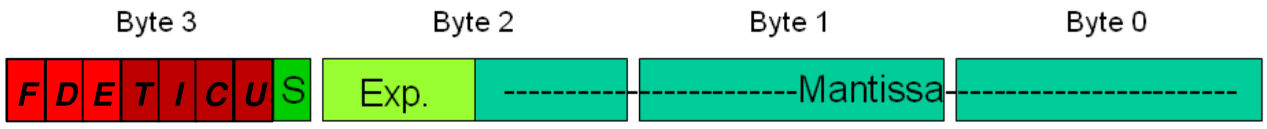


Fig. 4.5:48 – Internal "Node" structure (based on Fig. 4.5:58 showed later on herein)

For more notes about *"Nodes"* features refer to [Att. 27]. After all the considerations up to here, it is quite clear that the programming model/paradigm taken to develop automotive FDEFs is undoubtedly going to be one centred on "Data-Flow" [380]. In appendix [Att. 28] further *"Macros-Sequences"* & *"Macros-Processor"* details and issues can be revisited.

FINAL NOTE: Classical development environments demand effort and time in keeping all handling details in mind, besides being often confusing and irritatingly ocer-cluttered.

From the statements above, ***PITFALL H#51*** (2/2 – Graphical Cluttering) and ***PITFALL H#49*** (2/2 – Lengthy Turnarounds) are thus effectively eliminated through the elegant/compact options design and through positioning them on the exact spots where they are needed.

4.5.3 Basic Architecture Choice & Operational Details

[Emphasis: basic architecture and operational details]

Since the advent of the data-flow research [380] that this form of processing has been regarded as a very interesting way to massively parallelize systems, without having necessarily to incur into the typical parallelization problems of more classical programming methods. Using imperative programming languages, difficulties in compiling these to data-flow architectures became a clear handicap as previously discussed. Although data-flow programming has been of interest since 1975 [380], there has been little advancements until the 90s, where finally industry-driven environments such as ASCET [7], LabVIEW [6] and Simulink [5] started to appear. Virtually all approaches go the hybrid way, where data-flow graphs have to be compiled into Von Neumann architectures, resulting into the "pseudo-coders" and "auto-code generators", on top of all the other historical layers that already existed for the usual imperative programming languages.

Taking all the previous considerations, especially those concerning data-flow friendly architectures, the internal module architecture chosen for the final implementation have similarities to the Transputer [148] and the nCube-2 [451] highly parallel architectures, where each CPU is a self-contained processing unit with it own integrated memory, communications channels for adjacent CPUs and inspection facilities. Additionally, the architecture around the CPUs implemented herein will not have use semaphores, nor any too complex caching and memory mechanisms. In terms of internal communications and data-transfer paths, everything is implemented in the most direct way without anything unnecessarily complex in-between. Also, the then resulting complete *"FDEF-Processor"*, including the *"Nodes"* volatile and *"Macros"* non-volatile memory structures, will be of a very direct and simple memory-based "read/write" processing type, without any complex stack-, register-, accumulator-based components or mixtures there from. Last but not least, in this architecture, everything intends to be fully "orthogonal", to allow for a constraintless way of processing.

As for the external parallel processing architecture, a very simple high-speed message-passing type of communications system is going to be built. But instead of the complex multi-messaging of Transputers [148], for example, here only a single message is going to be transceived through the modules' inter-connecting links: *"Nodes"*. Nothing more than that will be needed to have and entire automotive system up and running as expected. Fig. 4.5:49 shows a glimpse of the global architecture, very similar to that of the Transputers but with only a communications ring instead of the more complex matrix.

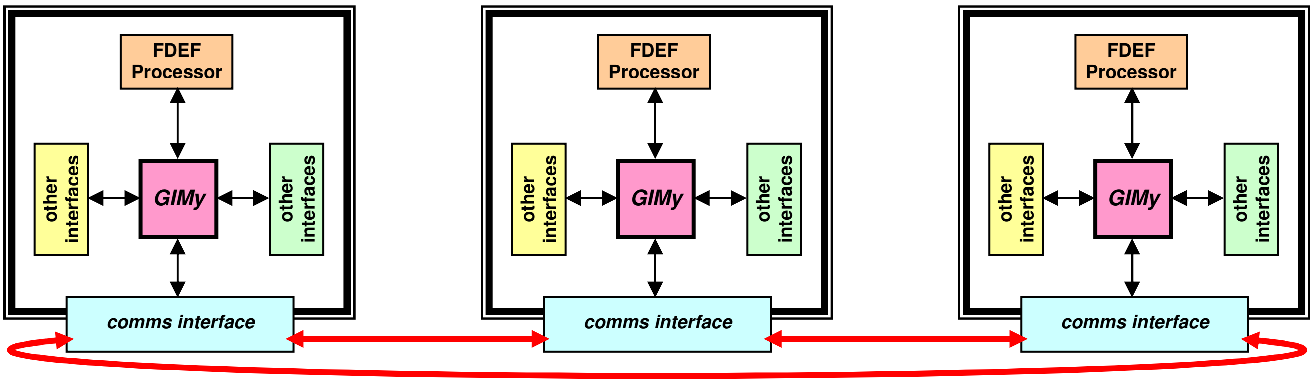


Fig. 4.5:49 – Global internal and external architecture idealized for the "Macros" processing hardware.

This highly tolerant "amnesic/micro-rebooted" processing architecture strictly follows the line already envisioned earlier, of having all *"Nodes"* outside the *"FDEF-Processor"* and nothing besides the program-counter resides as "state" inside the processor itself. Parallelism is achieved through interconnection of several equal modules. Although each of these modules suffers from the same criticism done to Von Neumann machines in [380] (only one single program-counter, therefore being not hardware oriented for parallel processing), this keeps them very simple (only one *"Macros-Sequence"* executed from top to bottom, no complex deviations or side-effects allowed), while several such modules allow for a still very simple parallel processing architecture. This is surely one of the key points of the work developed herein. Yet another criticism mentioned in [380] has to do with the global memory present in Von Neumann machines, which represents another bottleneck when concurrent accesses are attempted. The architecture devised herein also solves this by having each module executing its single *"Macros-Sequence"* with internally non-concurrent accesses by the corresponding *"FDEF-Processor"*. Other internal concurrent accesses never mess with this processing since that central memory will allow multiple reads, while multiple writes will be forbidden from the start.

The herein chosen architecture will also minimize results' communications overhead mentioned in [554] during parallelization, since all the data always and readily exists in the *GIMy* of each *"Cellular-ECU"*, whereas inter-communications run in the background without any explicit program intervention, thus not directly entering the equation of parallelization speed-up.

4.5.3.1 Automatic Graphics-Engine

[Emphasis: fully automatic graphical display of "Macros-Sequences"]

The *"Macros-Sequence"* is taken by the graphics-engine and visually laid out starting with the first *"Macro"* and ending with the last one in the list, exactly the same way these are executed starting from the first to the last one in that same list. For this graphics-engine to work, no user-intervention or setup is needed, being everything done fully automatically in the background. The user just has to design its FDEFs through direct graphical manipulation or through the command-line presented later. Both these manipulation methods result in the internally held *"Macros-Sequence"*, which will then be transformed into its graphical representation by the graphics-engine in an online and feedback enabled manner. Fig. 4.5:50 illustrates this process with one FDEF example. The basic rule of the graphics engine is as follows: execution is done from top to bottom and from left to right, in the exact same sequence as that represented by the raw *"Macros-Sequence"*. This is the direct consequence of having the *"iEditor"* draw these visual graphs automatically. This implicit knowledge eliminates some of the explicit and fundamental pieces of information present in classical directed graphs such as described in [380], namely "Graph Data Direction explicit arrows", "Operation Execution Triggering", "Demand driven execution approach" and "Triggering through Data Tokens". A more extended discussion on these issues can be found in [Att. 29], as well as some more illustrations showing this graphics-engine operating upon FDEFs.

This implicit knowledge (emanated from the simple straightforward processing of the *"Macros-Sequence"*) basically eliminates the need to incorporate the related code/structures into the development environment altogether, by being a part of the programming language itself. This is one major contribution of the work developed herein. The graphical engine basically just sits there and does nothing else other than continuously being triggered by the *"iEditor"* to re-draw the FDEF according to the current *"Macros-Sequence"* with all its changes. The graphics-engine maintains a permanent and instant synchronization of both at all time. Note that the input (sensors) and output (actuators) *"Nodes"* are defined elsewhere and will be illustrated later.

Since the graphics-engine developed in this thesis' work reveals both drawing and processing rules as being exactly the same, the developer gets a sort of "WYSIWYG(E)[*]" effect in terms of the execution result being exactly the same as what the developer sees graphically on the *"iEditor"*. This will be seen in the next illustrations, compressing one of the main initial ideas of the *"Macros"*: to have a self-induced and automated way of the program doing exactly what the developer intended, doubly "by design". "Doubly", because the developer gets what he designed for and because the system is designed to do what it should do.

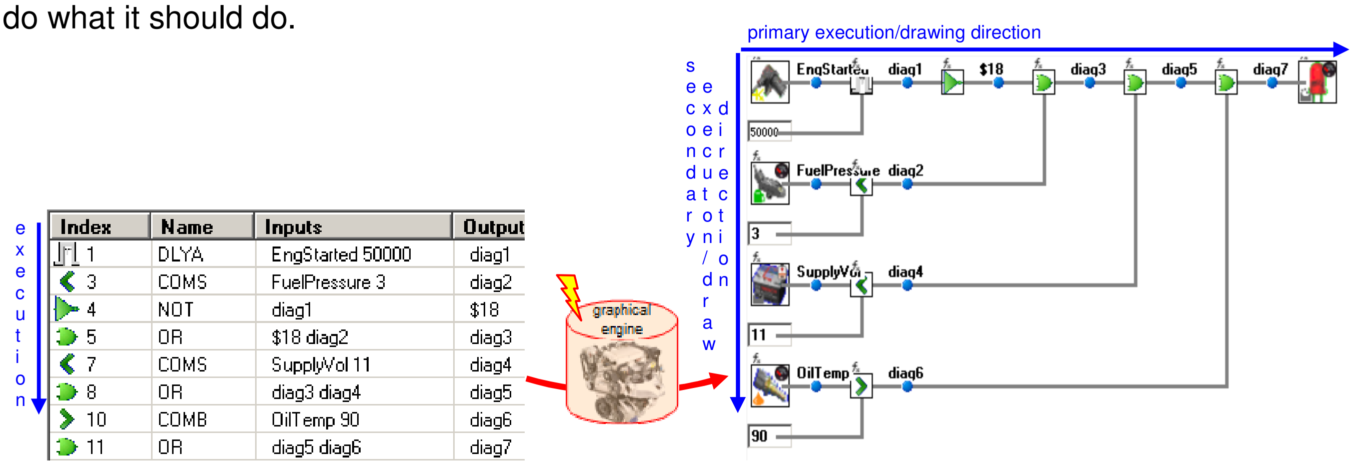

Index	Name	Inputs	Output
1	DLYA	EngStarted 50000	diag1
3	COMS	FuelPressure 3	diag2
4	NOT	diag1	$18
5	OR	$18 diag2	diag3
7	COMS	SupplyVol 11	diag4
8	OR	diag3 diag4	diag5
10	COMB	OilTemp 90	diag6
11	OR	diag5 diag6	diag7

Fig. 4.5:50 – Example of a "Macros-Sequence" and its fully automatic graphical representation

Fig. 4.5:51 shows yet another and more extensive (and really usable) *"Macros-Sequence"* with an illustration of its sequential execution progress. Numbers {1} throughout {11} indicate the order and the *"Macro"* being calculated at that precise point. *"Nodes"* are calculated with this active rule: always execute the *"Macro"* that is needed to calculate the *"Node"* that is going to be used in the next *"Macros"*. In fact, this rule is just a virtual rule, since the *"FDEF-Processor"* just simply executes the *"Macros"* in their stored sequence

[*] *What You See is What You Get (Executed)*

inside the *"Macros-Sequence"* that is fed to the *"FDEF-Processor"*. The graphical FDEF engine just limits itself on drawing those *"Macros"* going through the same sequence. The execution sequence shown in Tab. 4.5:52 shows the two-way relationship between the sequence and the graphical layout.

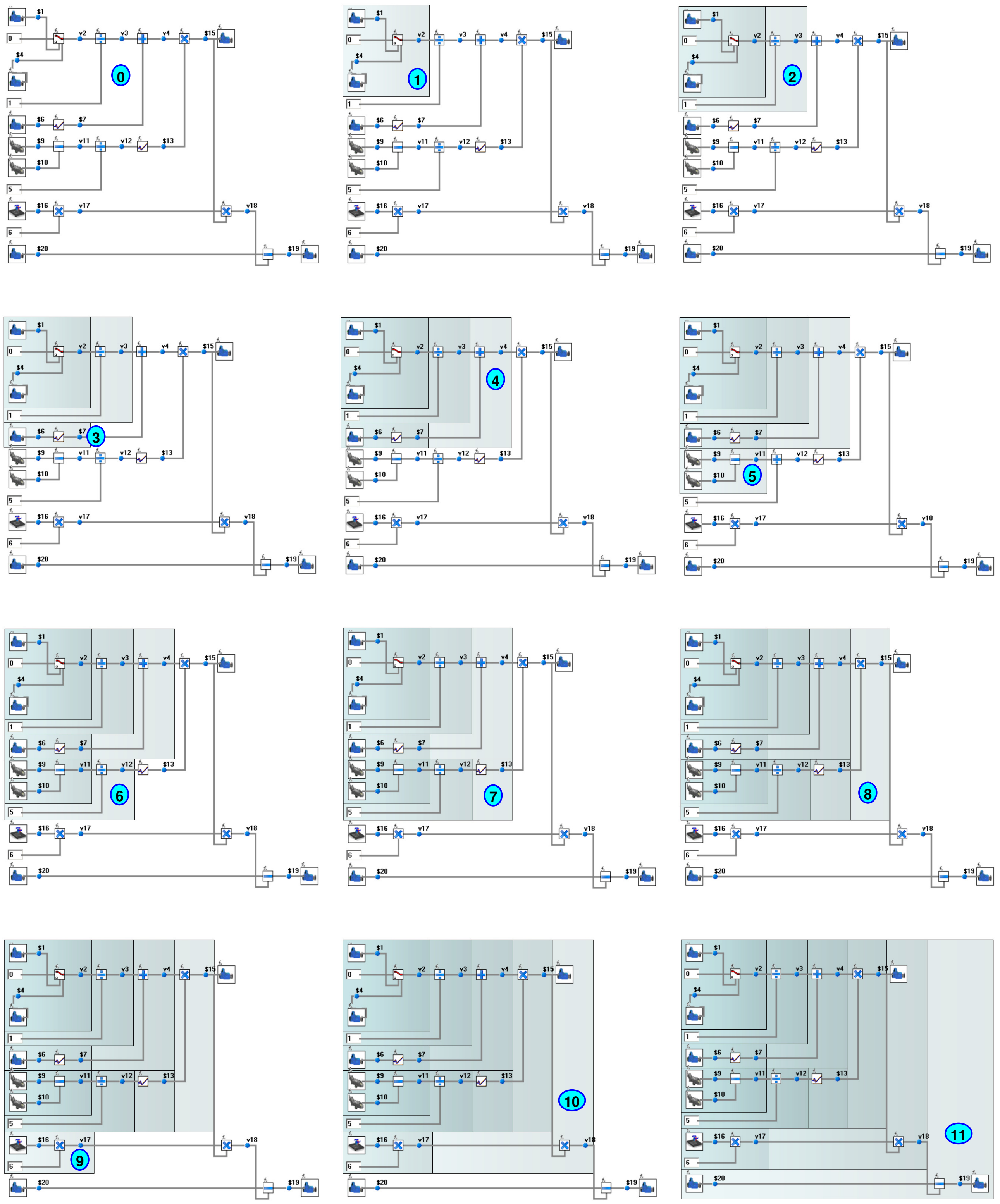
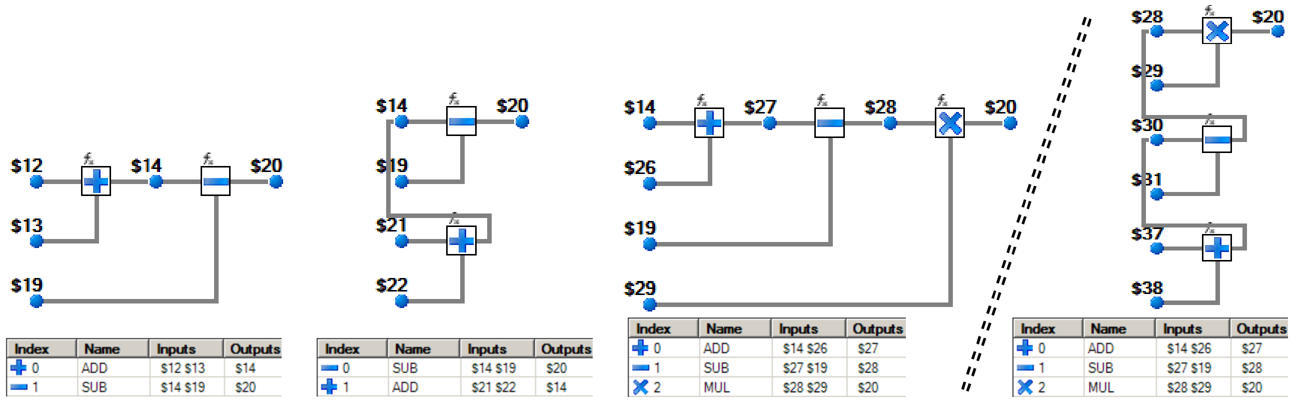


Fig. 4.5:51 – Example of another "Macros-Sequence" and its quite obvious graphical execution flow

index	***"MACROS-SEQUENCE"***			
	MACRO	INPUTS	OUTPUT	
1	**SW**	**{0} {$1} {$4}**	**{v2}**	First *"Macro"* executed, uses first input *"Nodes"* and generates first output *"Node"*
2	**DIV**	**{v2} {1}**	**{v3}**	Division has {v2} and constant {1} avilable, so calculate this division
3	**PAR1**	**{$6}**	**{$7}**	For the next addition to be executed, it needs this parameter to calculate {$7} first
4	**ADD**	**{v3} {$7}**	**{v4}**	This addition can only be executed as soon as {$7} is available Since now {$7} is available, this addition can finally be executed
5	**SUB**	**{$9} {$10}**	**{$11}**	To execute the multiplication in index 8, all *"Macros"* 5, 6, 7 must be executed first Execute this subtractions by taking the input *"Nodes"*
6	**DIV**	**{v11} {5}**	**{v12}**	Execute this division by taking available {v11} and constant {5}
7	**PAR1**	**{v12}**	**{$13}**	Execute this parameter by taking available {v12} and calculating the needed {$13}
8	**MUL**	**{v4} {$13}**	**{$15}**	This multiplication can only be executed as soon as {$13} is available Since now {$13} is available, this multiplication can finally be calculated
9	**MUL**	**{$16} {6}**	**{v17}**	For the next multiplication to be executed, it needs this one to calculate {v17} first
10	**MUL**	**{$15} {v17}**	**{v18}**	This out-of-main-path multiplication can only be executed when {v17} is available Since now {v17} is available, this multiplication can finally be executed
11	**SUB**	**{$20} {v18}**	**{$19}**	Finally execute the last *"Macro"* in the entire sequence and restart from top

Tab. 4.5:52 – Sequenced explanation of the execution progress depicted in Fig. 4.5:51

Fig. 4.5:53 illustrates how the graphical placing method efficiently solves the execution sequencing problem and even allows any desired sequencing, however odd it might be. As it can be seen, the execution order of the *"Macros"* can be easily changed and even reversed. The developer just has to place those *"Macros"* accordingly, even if it does not make much sense to place them in that order.



Index	Name	Inputs	Outputs
+ 0	ADD	$12 $13	$14
− 1	SUB	$14 $19	$20

Index	Name	Inputs	Outputs
− 0	SUB	$14 $19	$20
+ 1	ADD	$21 $22	$14

Index	Name	Inputs	Outputs
+ 0	ADD	$14 $26	$27
− 1	SUB	$27 $19	$28
× 2	MUL	$28 $29	$20

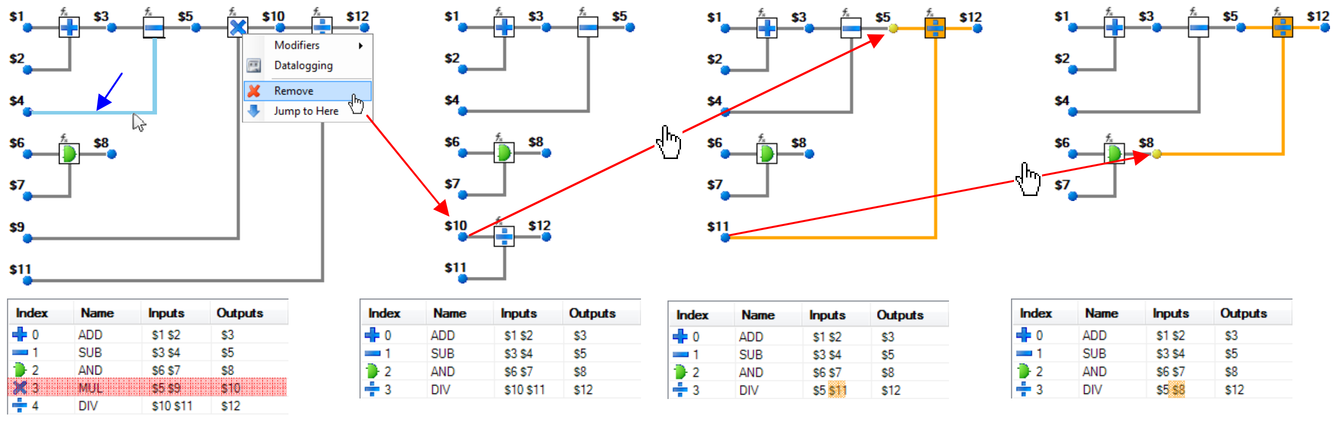
Index	Name	Inputs	Outputs
+ 0	ADD	$14 $26	$27
− 1	SUB	$27 $19	$28
× 2	MUL	$28 $29	$20

Fig. 4.5:53 – Two examples of execution order reversal solely achieved through different placements

Also note that whenever placing a new *"Macro"* onto the existing FDEF, the graphics-engine must be perfectly capable of knowing the exact position where the *"Macro"* has to be placed, so that the visual representation then really reflect the location the developer was intending. If this graphical detector has a bug, then the *"Macro"* would be placed in the wrong position of the global *"Macros-Sequence"* and the graphical engine would thus display the wrong visual representation. Because *"Live-Dragging"* triggers the graphical engine whenever the developer makes some change, thus continuously re-drawing the visual representation of the *"Macros-Sequence"*, it already displays the preliminarily correct visual looks that will be accepted by the *"iEditor"*.

From the previous statements, one more thing must be pointed out: the sequencing of the *"Macro"* operations is given by the order/position in which the developer placed them inside the FDEF. There is no other way of sequencing them other than graphically, while there is also no hidden menu to allow setting that sequence. Further and deeper illustrations of the graphical FDEF editing can be seen in [Att. 29].

Manipulations such as highlighting a *"Node"* path, deleting *"Macros"* and drag-dropping their individual *"Nodes"* around, is illustrated next Fig. 4.5:54. These actions as well as special conditional *"Macros"* will be further detailed later on.



Index	Name	Inputs	Outputs
0	ADD	$1 $2	$3
1	SUB	$3 $4	$5
2	AND	$6 $7	$8
3	MUL	$5 $9	$10
4	DIV	$10 $11	$12

Index	Name	Inputs	Outputs
0	ADD	$1 $2	$3
1	SUB	$3 $4	$5
2	AND	$6 $7	$8
3	DIV	$10 $11	$12

Index	Name	Inputs	Outputs
0	ADD	$1 $2	$3
1	SUB	$3 $4	$5
2	AND	$6 $7	$8
3	DIV	$5 $11	$12

Index	Name	Inputs	Outputs
0	ADD	$1 $2	$3
1	SUB	$3 $4	$5
2	AND	$6 $7	$8
3	DIV	$5 $8	$12

Fig. 4.5:54 – Examples of path highlighting, "Macro" deletion and of subsequent "Node" dragging

For this automatic graphics-engine to operate perfectly, the 1-to-1 relationship between single *"Macros"* and visual icons is of primordial importance. This is what keeps everything simple, coherent and unmistakable from the beginning on, be it on the SW user-interface or HW platform. Note also that the *"Macros"* always receive their main input value from the left and second input value, most usually an auxiliary value, from below. This allows to better organize those signals visually. This automatism eliminates those annoying graphical placing editor bugs/glitches, still persistent in modern software packages such as Matlab Simulink R2013b [5] and ASCET-SD [7], still messing up wirings quite often.

FINAL NOTE: Classical graphical development environments require big efforts in keeping everything in the right place, manually, accompanied by software placing bugs and failures. If textual programming is pursued, this gets worse with syntax and metrics details to be constantly observed.

From the statements above, ***PITFALL C#4***$^{\text{(Lexics \& Syntax)}}$, ***PITFALL C#5***$^{\text{(Sequencing)}}$, ***PITFALL O#67***$^{\text{(Code Metrics)}}$, ***PITFALL C#13*** (Source vs. Machine-Code Diversity) and ***PITFALL H#48***$^{\text{(1/2 – Manual Layout)}}$ are thus effectively eliminated through the intrinsically enforced automatic graphical layouting scheme, as well as through the fact that the main code is always the unique central code: the *"Macros-Sequence"*.

4.5.3.2 Two-way Horizontal Designing made intrinsic

[Emphasis: treating both FDEF and hardware as two representations of the original "Macros-Sequence"]

The graphics-engine of the *"iEditor"* displays the FDEFs automatically on-screen, always using only the original *"Macros-Sequence"*, which determines which *"Macros"* are to be drawn onto the user-interface. This visual representation is really just a way to see that central *"Macros-Sequence"* and nothing more, being graphical manipulations also accomplished within this graphical view and immediately incorporated into the *"Macros-Sequence"* itself and nowhere else. Since this *"Macros-Sequence"* is really what is inside the hardware platform after the distribution and download process, then it is very clear that these mechanisms represent a true "two-way" design environment.

The most important historical feature that distinguishes classical tools from the work developed herein is that the first ones were developed separately, that is, one layer at a time at different moments, never thinking of merging them in the first place, just for all of those separate layers to be joined at the end by classical interfacing and without any possibility of true mergers. The work herein was developed as a whole, thinking of the entire system as one big and especially merged unit that also works as one and not as separate layers or entities. Fig. 4.5:55 compares two examples of graphical based engines, with the classical systems needing translators, contrasting with the lack of such translators for the *"Macros"*-based system.

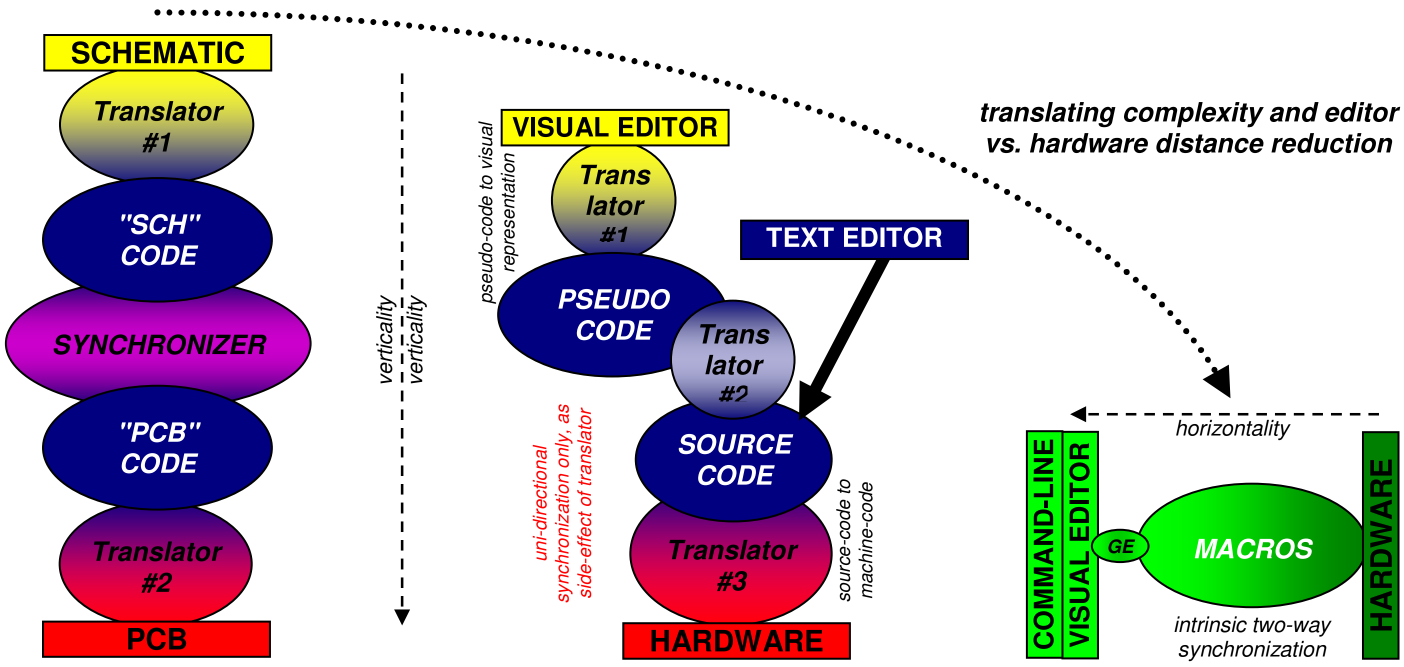


Fig. 4.5:55 – Left: standard schematics/PCB tool; Middle: classical visual/textual programming tool; Right: horizontally placed "Macros"-based work herein

The notion of "horizontal" flow present in the title of this section intends to represent that the *"Macros-Sequence"* allows a horizontal information flow, in that this flow happens on exact the same level. Furthermore, it is bi-directional as explained above, which contrasts with the mostly top-to-bottom kind of limited information flow present in those state-of-the-art tools. This notion is detailed here in Fig. 4.5:56.

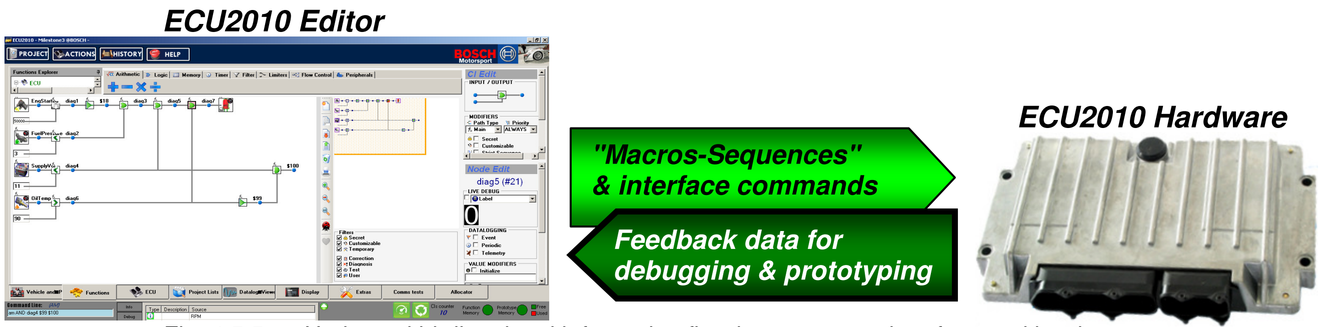

ECU2010 Editor
ECU2010 Hardware
"Macros-Sequences" & interface commands
Feedback data for debugging & prototyping

Fig. 4.5:56 – Horizontal bi-directional information flow between user-interface and hardware

This natural horizontal bi-directional information-flow is created by the simple fact that the *"Macros-Sequence"* is both the source-code and the machine-code, all incorporated into that one single representation. This produces the desired effect of features such as *"Live-Prototyping"* (explained later) to emerge as a natural/intrinsic consequence of that bi-univocality, in that changes introduced into the source-code running user-interface can immediately be reflected into the machine-code running hardware without any extra need for intermediate lengthy or complex translations. Debugging features also follow the same natural/intrinsic path, in that feedback data is simply taken from the *"Macros-Sequence"* executing hardware and fed into that same *"Macros-Sequence",* but now on user-interface side, again without any extra need for complex manipulations and representations.

The intrinsic "homoiconic" nature [483] of *"Macros"* is one of the main reasons for allowing an intrinsic "two-way side-effect" feature without putting too much effort into the associated development system. On the *"iEditor"* user-interface side, "homoiconicity" is even driven further in the work implemented herein, since the edited (added, changed, deleted) *"Macros"* are always directly manipulated in RAM where they really reside (remember that the visual part is just a representation of that *"Macros-Sequence"* residing in memory.

One feature of "homoiconicity" is however not enabled/used: the capability of *"Macros"* changing themselves, during runtime. It is not allowed for *"Macros"* to change their own memory, being only able to change *"Nodes"* inside the *"GIMy"*. The *"GIMy"* is the only internally changeable data-structure here and has been separated from the *"Macros"* themselves to guarantee this consequent separation (although using a similar 32bit data type). No practical use in automotive systems would be achieved through this meta-programming feature, not even allowed for security reasons. More details in [Att. 29].

FINAL NOTE: Classical development systems often work with a wide range of different file formats. This is due to the fact of them having a wide range of internal layers and their corresponding translator engines.

From statements above, ***PITFALL C#11***$^{(1/2\text{ - Translators})}$ and ***PITFALL H#56***$^{(\text{Upwards/Return Path})}$ are effectively eliminated through fully intrinsic and full-range horizontal “two-way” nature of the *“Macros-Sequence”* representation.

From statements above, ***PITFALL O#54***$^{(\text{Multiple Software Formats})}$ is only partially eliminated through the fact that *“Macros-Sequence”* almost represents everything, because there will be other needed system related files.

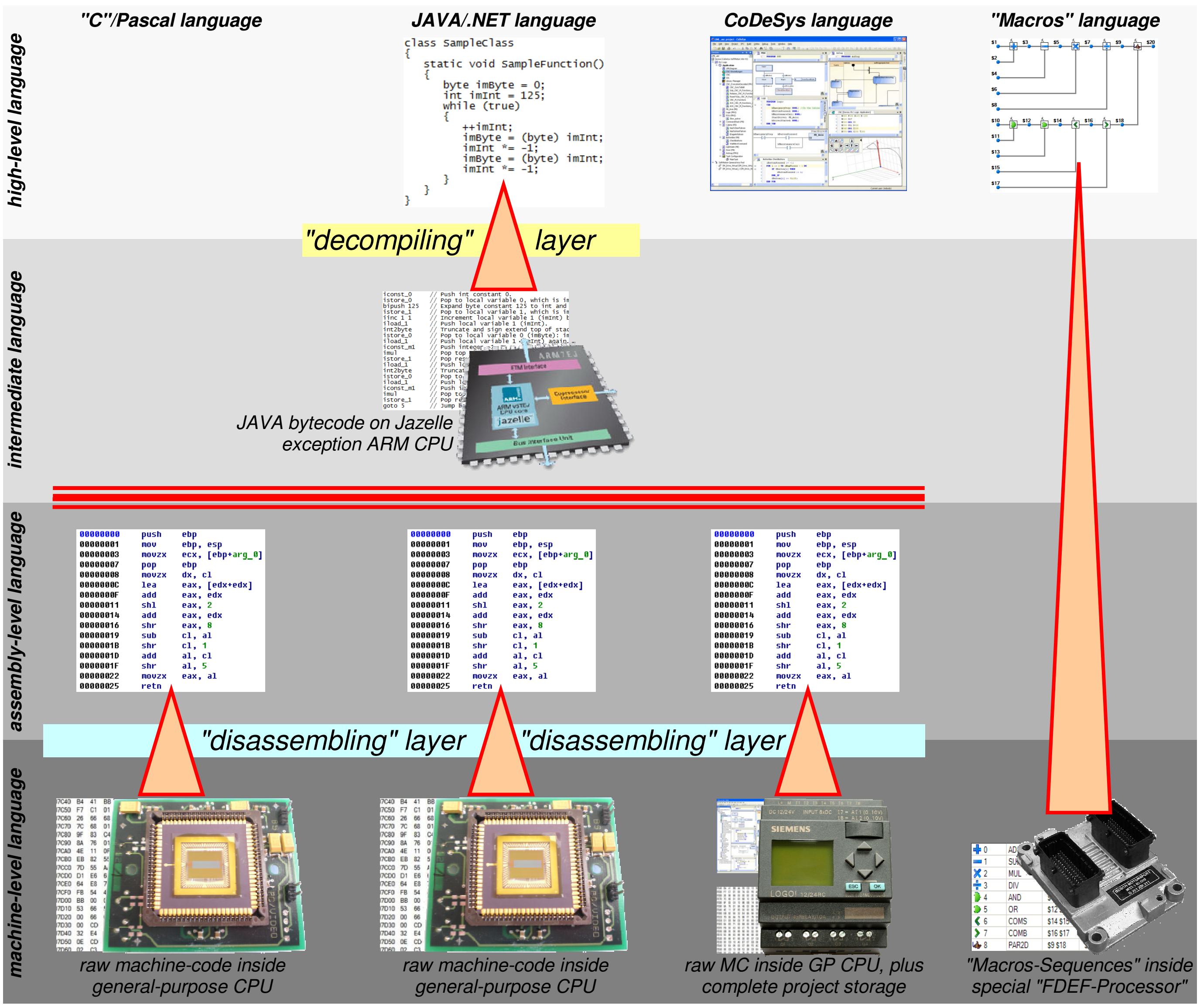
4.5.3.3 Reverse-Engineering made intrinsic

[Emphasis: treating the central "Macros-Sequence" as the hardware's contents]

Recalling that the *"Macros-Sequence"* is in the exact middle of the development system and represent the main program entity, makes mechanisms such as reverse-engineering as easy as being themselves intrinsic to these systems, precisely because this mechanism is realized by the graphics-engine itself. Since any *"Macros-Sequence"* may be viewed in the graphical form, one can think of this mechanism as a reverse-engineering mechanism or, even better, as a language that aside from user labels and comments, does not need any explicit/special reversing since the graphical representation of *"Macros-Sequences"* is nothing more than another view of the same original *"Macros-Sequence"*, albeit a more visual one. This means immediately that one can take an ECU based on this technology, upload its code (the *"Macros-Sequences"*) and have an immediate and effortless visual look into it, without having to search for the corresponding source-code.

This reverse-engineering feature is similar to a "disassembler" of regular machine-code, with the difference of being intrinsic and not needing special mechanisms to recover the entire *"Macros-Sequence"* into visual form. "Disassembling" a *"Macros"* program does not therefore require any special strategy other than reading the hardware's contents and directly displaying them on the *"iEditor"*. Only the comments and labels are eventually lost.

A brute-force way of preserving the high-level program information is done in CoDeSys in [22] where the entire high-level source-code and graphical information is literally downloaded into the controller's memory. Although highly memory-consuming and less secure to have the entire project inside the controller, it is the only really effective way of having access to all information at controller level today. Fig. 4.5:57 illustrates the three most common reverse-engineering paths available today, alongside with the work done herein. While virtually all processors need raw machine-code to operate (the *Jazelle* [102] is a partial exception), "disassembling" it leads to a dead-end in most cases since all the high-level structural information has been lost during prior compilation and assembly. In the case of JAVA, .NET and other Bytecode languages, if the Bytecode is still available, decompiling it to higher-level source-code is fairly easy, although real names are lost as well. This can be the case where this Bytecode is also present in the CPU, such as the *Jazelle* [102] that is capable of storing and executing JAVA Bytecode natively. This is rather an exception than a rule in the real world. Being reverse-engineering usually so difficult, in CoDeSys [22] the manufacturer opted to explicitly go for the easiest solution on purpose, storing the entire project inside the controller. This controller then contains the appropriate interpreter to interpret the intermediate language inside that project, to feed the still general-purpose CPU with raw machine-code once again. On the right is the work done herein, where the *"Macros-Sequence*" inside the CPU is the visual representation as well, rendering reverse-engineering as an intrinsic built-in feature.


"C"/Pascal language
JAVA/.NET language
CoDeSys language
"Macros" language
high-level language
intermediate language
assembly-level language
machine-level language
"decompiling" layer
JAVA bytecode on Jazelle exception ARM CPU
"disassembling" layer
"disassembling" layer
raw machine-code inside general-purpose CPU
raw machine-code inside general-purpose CPU
raw MC inside GP CPU, plus complete project storage
"Macros-Sequences" inside special "FDEF-Processor"

```
class SampleClass
{
    static void SampleFunction()
    {
        byte imByte = 0;
        int imInt = 125;
        while (true)
        {
            ++imInt;
            imByte = (byte) imInt;
            imInt *= -1;
            imByte = (byte) imInt;
            imInt *= -1;
        }
    }
}
```

```
iconst_0      // Push int constant 0.
istore_0      // Pop to local variable 0, which is im
bipush 125    // Expand byte constant 125 to int and
istore_1      // Pop to local variable 1, which is im
iinc 1 1      // Increment local variable 1 (imInt) b
iload_1       // Push local variable 1 (imInt).
int2byte      // Truncate and sign extend top of stac
istore_0      // Pop to local variable 0 (imByte): im
iload_1       // Push local variable 1 (imInt) again.
iconst_m1     // Push integer
imul          // Pop top
istore_1      // Pop res
iload_1       // Push lo
int2byte      // Truncat
istore_0      // Pop to
iload_1       // Push lo
iconst_m1     // Push i
imul          // Pop top
istore_1      // Pop re
goto 5        // Jump b
```

```
00000000    push    ebp
00000001    mov     ebp, esp
00000003    movzx   ecx, [ebp+arg_0]
00000007    pop     ebp
00000008    movzx   dx, cl
0000000C    lea     eax, [edx+edx]
0000000F    add     eax, edx
00000011    shl     eax, 2
00000014    add     eax, edx
00000016    shr     eax, 8
00000019    sub     cl, al
0000001B    shr     cl, 1
0000001D    add     al, cl
0000001F    shr     al, 5
00000022    movzx   eax, al
00000025    retn
```

```
00000000    push    ebp
00000001    mov     ebp, esp
00000003    movzx   ecx, [ebp+arg_0]
00000007    pop     ebp
00000008    movzx   dx, cl
0000000C    lea     eax, [edx+edx]
0000000F    add     eax, edx
00000011    shl     eax, 2
00000014    add     eax, edx
00000016    shr     eax, 8
00000019    sub     cl, al
0000001B    shr     cl, 1
0000001D    add     al, cl
0000001F    shr     al, 5
00000022    movzx   eax, al
00000025    retn
```

```
00000000    push    ebp
00000001    mov     ebp, esp
00000003    movzx   ecx, [ebp+arg_0]
00000007    pop     ebp
00000008    movzx   dx, cl
0000000C    lea     eax, [edx+edx]
0000000F    add     eax, edx
00000011    shl     eax, 2
00000014    add     eax, edx
00000016    shr     eax, 8
00000019    sub     cl, al
0000001B    shr     cl, 1
0000001D    add     al, cl
0000001F    shr     al, 5
00000022    movzx   eax, al
00000025    retn
```

Fig. 4.5:57 – Comparison between different languages and approaches to "disassemble" them

The intrinsically "homoiconic" nature of the *"Macros"* already mentioned above in the natural "two-way" characteristic is, again, and even more obviously, the main reason for this "reverse engineering" to be so easy and straightforward in this system herein. There is virtually no effort in this, since the *"Macros"* are the core representation for all. More details about this kind of *"Reverse-Engineering"* can be found in [Att. 29].

FINAL NOTE: Classical systems do not allow for reverse-engineering in any feasible way at all.
From the statements above, ***PITFALL H#53*** (Reverse Engineering) is thus effectively eliminated through the intrinsic fact that the *"Macros-Sequence"* is de *de-facto* only code representation through the entire system.

4.5.3.4 Custom Floating-Point numbering (*"BCDP"*)

[Emphasis: appropriate automotive number format]

A custom floating-point numbering scheme has been developed, based on the assumption that no visible rounding errors should occur from the user point of view. The purpose is to have a numbering scheme that guarantees that calculations and display of values will be accurate in the sense of a decimal representation. One could have taken a simple BCD scheme, but to optimize this scheme for the early hardware, a novel format and calculation algorithms have been developed and called *"BCDP"* as for *"BCD-Power"* [Att. 4]. This scheme allows for a perfect, non-rounding decimal representation which was used throughout the entire work in this thesis, both in software and in hardware structures. This format avoids binary floating-point rounding by simply maintaining all digits directly represented in the mantissa. Fig. 4.5:58 shows the basic internal format. Please see [Att. 4] for more detailed information about this numbering scheme. Fig. 4.5:59 shows some typical floating-point format produced rounding errors due to quantization effects in the standard IEEE-754-1985 [176] numbering scheme (which should never happen in the real world customer applications), which are intrinsically solved through *"BCDP"*. Although scientifically admissible, these discrepancies are not at all accepted in the automotive scene, especially in the motorsport scene. In that realm, these effects are always looked upon with very high distrust leaves and even suspicions of such odd effects happening due to some serious system software flaw.

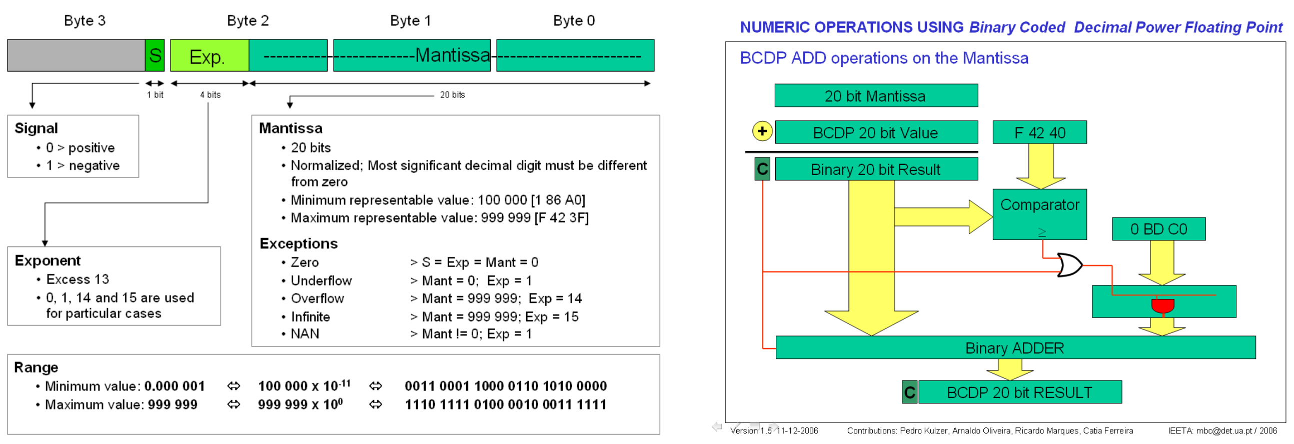
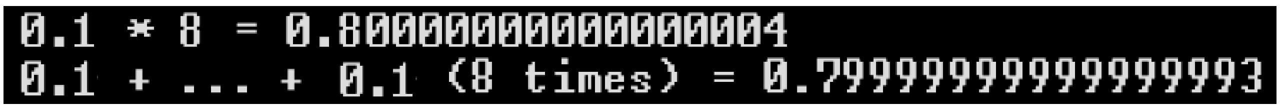

Byte 3
Byte 2
Byte 1
Byte 0
S
Exp.
Mantissa
1 bit
4 bits
20 bits
Signal
• 0 > positive
• 1 > negative
Exponent
• Excess 13
• 0, 1, 14 and 15 are used for particular cases
Mantissa
• 20 bits
• Normalized; Most significant decimal digit must be different from zero
• Minimum representable value: 100 000 [1 86 A0]
• Maximum representable value: 999 999 [F 42 3F]
Exceptions
• Zero > S = Exp = Mant = 0
• Underflow > Mant = 0; Exp = 1
• Overflow > Mant = 999 999; Exp = 14
• Infinite > Mant = 999 999; Exp = 15
• NAN > Mant != 0; Exp = 1
Range
• Minimum value: 0.000 001 ⇔ 100 000 x 10-11 ⇔ 0011 0001 1000 0110 1010 0000
• Maximum value: 999 999 ⇔ 999 999 x 100 ⇔ 1110 1111 0100 0010 0011 1111
NUMERIC OPERATIONS USING Binary Coded Decimal Power Floating Point
BCDP ADD operations on the Mantissa
20 bit Mantissa
+
BCDP 20 bit Value
C
Binary 20 bit Result
F 42 40
Comparator
≥
0 BD C0
Binary ADDER
C
BCDP 20 bit RESULT
Version 1.5 11-12-2006
Contributions: Pedro Kulzer, Arnaldo Oliveira, Ricardo Marques, Catia Ferreira
IEETA: mbc@det.ua.pt / 2006

Fig. 4.5:58 – Brief format illustration of the "BCDP" numeric scheme and the "ADD" operation



```
0.1 * 8 = 0.8000000000000004
0.1 + ... + 0.1 (8 times) = 0.7999999999999993
```

Fig. 4.5:59 – Examples of "unexpected" IEEE-754 calculation results due to quantization effects

The use of this *"BCDP"*, on the other hand, unifies everything into a single arithmetic format for all *"Nodes"*, eliminating the need for any eventual casts or translations whatsoever. Additionally, to comply with the automotive scene's demands, this format also allows "-0" and "+0" to be represented and calculations made upon those values. This allows for the system to react to divisions by these "zero" values, by yielding the therein desired values of -999.999 and +999.999, respectively (maximum negative and maximum positive values, respectively). This is similar to the IEEE-754 floating-point arithmetic implementations in some languages that return similar results.

Boolean/logical values are also supported, where FALSE is 0.0 and TRUE is everything else, just as in other languages such as "C". *"Nodes"* will be locations inside the *"GIMy"* containing only *"BCDP"* values. By resorting to this approach, there will not exist the notion of "variable" anywhere. There will be therefore no need to declare them nor to specify their nature anytime ever. The *"iEditor"* takes care of everything related to *"Nodes"* and the developer does not have to lose time nor effort. The uncountable bugs that happen while choosing desired quantizations and coding the conversions, are also eliminated.

The IEEE-754-1985 standard was revised and IEEE-754-2008 [176] appeared in 2008. This standard finally incorporates a "decimal floating-point" numbering format, to successfully respond to finance and taxation calculation needs for exact (unrounded) results when using decimal numbers as inputs. Decimals is also the format that pocket calculators used since they appeared, then still without a standard to rule over them. Since this *"BCDP"* format was developed in parallel with the start of this *"ECU2010"* project associated to this work herein, back in 2006, there was no knowledge of the revised IEEE-754 standard as of yet. Other decimal formats include [559] [560] among others.

FINAL NOTE: Classical systems use various floating-point formats and some still use fixed-point formats. Either way, there are visual representation problems never solved thus far, nonetheless confusing and irritating customers.

From the statements above, ***PITFALL H#47***$^{(\text{Floating-Point Discrepancies})}$, ***PITFALL C#1***$^{(\text{Arithmetic Formats})}$, ***PITFALL O#21***$^{(\textit{1/2 – Flags})}$, ***PITFALL O#10***$^{(\textit{3/4 – Code Coverage})}$ and ***PITFALL C#2***$^{(\text{Endianness})}$ are thus effectively eliminated through the use of a custom application-adapted homogeneous numbering scheme.

4.5.3.5 One Producer, One or Several Consumers

[Emphasis: enforced "single assignment" rule]

By definition, in a *"Macros-Sequence"* there can never be more than one single value assignment to each *"Node"*. This is not to be confused with strict "single-assignment" rules in some languages, where only one initialisation may be made during start. Herein, "single-assignment" simply means that each *"Node"* may only have one single *"Macro"* assigning a value to it in the entire program. This single assignment will then happen once over each *"Cycle-Time"*. This will strictly be enforced by the *"iEditor"*, which will issue an error message should the user try to more than on assignment to the same *"Node"*. On the other hand, each *"Node"* may be freely used more than just once, for consumers.

Contrary to programming with textual languages such as "C" an Assembly, where multiple assignments are freely allowed anywhere throughout a program, "single-assignment" is common practice in well-behaved data-flow programming [380] where the *"Nodes"* are also called "data-flow values" (instead of "variables"). This prevents some types of side effects and therefore enhances locality of effect, which is claimed to be one big advantage of data-flow languages [380]. This multiple-assignment problem is only posed if a program flow may be interrupted: a global variable may be being tested in one thread, while another interrupts this thread from finishing this conditional operation and changes that variable to another value. When returning to the first thread, the condition might be tampered and deliver a wrong decision after all. This is know as the "global shared variables multi-assignment problem" and was already mentioned on the interrupts pitfall. Since the *"Macros"* processor developed herein will not allow any interrupts of this sort anyway, this problem will never occur. Nevertheless, something similar could occur at the below mentioned global *"GIMy"* memory level, where the processor might be testing a *"Node"* that would changed by a peripheral access right between the conditional test *"Macro"* and a next assignment *"Macro"*.

Virtually any ECU firmware contains variables being assigned to multiply. The *"iEditor"* prohibits this, but nevertheless initializations are still allowed since these happen only once and prior to any *"Macro"* being executed for the first time. These initializations are done through the use of the ⓘ initialization *"Node Modifier"*. Well-designed pure data-flow FDEFs should get up and running by themselves, thus not needing more than this simple initialization possibility. In a similar way, FDEF should never user conditional execution (IFs) for *"Node"* assignment possibly jeopardizing the single-assignment rule. To avoid this pitfall altogether, an IF-less FDEF design should be followed, as already hinted in Fig. 4.5:36 and even more detailed later on herein.

In the developed system, only one of the components connected to the *"GIMy"* (central *"Nodes"* repository explained a bit further) is ever allowed to write onto a particular *"Node"*, while all remaining components limit themselves reading and using that same *"Node"*. This eliminates any need for locks, semaphores or any other kind of potentially difficult to predict and fatal dead-lock causing mechanisms. Even lock-less solutions such as in [564] still require some extra processing intelligence to cope with outdated variables and such.

More examples and reasoning are done further down this work, when the hardware implemented *"GIMy"* concept is finally presented. In the global system developed herein, single-assignment literally means that a certain *"Node"* is only updated once in each *"Cycle-Time"*, throughout all the modules that eventually will make up the system. That *"Node"* may then be used by one or more modules simultaneously, being automatically distributed among them as needed.

Original *"Nodes"* that receive values processed by the *"FDEF-Processor"* are herein called *"source Nodes"*, whereas *"Nodes"* in other modules receiving these original *"Node"* values are called *"sink Nodes"* and exist only to allow FDEFs in other modules to also be able to work with those same values. This idea of a single source for many sinks is later going to be of great value when the *“Allocator”* and *“Macros”* distribution among several *“Cellular-ECUs”* comes into play. More information in [Att. 30].

FINAL NOTE: Classical systems still heavily rely on inherently allowing each variable being assigned to in multiple spots, empolying locks, semaphors or any other potentially dead-locking mechanism. Eliminating the side-effects caused by this, makes developing and debugging those programs instantly easier.

From the statements above, ***PITFALL C#31***$^{\text{(Multi-Assignment)}}$ is effectively eliminated through the allowance of strict single-assignment to each *"Node"* throughout the entire parallel system, everywhere including peripherals.

4.5.3.6 Raster-less Real-time Processing

[Emphasis: simple as-fast-as-possible processing]

In automotive systems, explicit operating-systems generally do not much more than time-raster and interrupt management/scheduling. Automotive functionalities are mostly time-triggered by those timer-rasters, but there are also some event-triggers to manage, such as engine-speed signal counting and the respective synchronous injection/ignition/knock signals' processing. At this point, it must be mentioned [372] that purely time-driven systems are incomparably more easy to understand than those also using event-driving. The problem with event-driven systems is that one never exactly knows when and where the event/interrupt is going to interrupt the main time-driven program and what exactly is going to happen when the event finishes and the main program finds the system eventually changed by that event. State-of-the-art operating-systems use a mixture of time-driven and event-driven processing. [Att. 31] complements this section with some notes and comparisons with state-of-the-art systems.

In total contrast to standard operating-systems, the *"Macros-Sequences"* are executed from the beginning to the end without any forced time-rasters or any other time-constraints. In other words, the *"Macros"* are just executed in the sequence presented to the *"Macros-Processor"*. As soon as this sequence reaches the end, it will begin executing immediately again, without any delay or control, from the beginning of exactly the same sequence. This will go on as long as the system is powered-up, in a never-ending loop called *"Macros-Cycle"*. While normal real-time systems intrinsically have priotized time-rasters, here the *"Macros"* are executed *“AFAP – As Fast As Possible”*. This time-measure is most important for this system, since it determines its real-time nature. To have a real-time system working properly, it is not strictly necessary to have a time-raster based processing architecture. Time-raster based schemes have been extensively used in the industry mainly due to being a historically accepted and very well known methods, but in reality it suffices to guarantee the *"Macros-Cycle"* time of new methodology to be lower than the fastest raster needed in an equivalent raster-based system. This *"Macros-Cycle"* time is called *"Cycle-Time"*. An example of engine-speeds and cylinder-count combination and the resulting necessary classical minimum time-raster, is given in Eq. 4.5:60 below.

$$T_{synchro} = \frac{60 \cdot 2}{RPM \cdot Cylinders} \rightarrow \left[\begin{array}{l} 4Cyl/6.000RPM \Rightarrow T_{synchro} = \frac{60 \cdot 2}{6000 \cdot 4} = 5ms \\ 8Cyl/6.000RPM \Rightarrow T_{synchro} = \frac{60 \cdot 2}{6000 \cdot 8} = 2{,}5ms \\ 8Cyl/12.000RPM \Rightarrow T_{synchro} = \frac{60 \cdot 2}{12000 \cdot 8} = 1{,}25ms \\ 12Cyl/20.000RPM \Rightarrow T_{synchro} = \frac{60 \cdot 2}{20000 \cdot 12} = 500\mu s \end{array} \right.$$

Eq. 4.5:60 – Synchro period for various engines and speeds

Fig. 4.5:61 makes a zoomed-in comparison between a classical "time-raster" based system (blue) with this *"Cycle-Time"* based system (green) developed herein, where the

most important detail (in terms of "real-time" issues) is to have this last system's *"Cycle-Time"* be as fast as the instantaneous fastest classical raster ("Synchro") at all times (red area in Fig. 4.5:61). This rule applies to the fastest "Synchro" and all other rasters as well (note that in Fig. 4.5:61 the engine is revving at different RPM so that the "Synchro" raster appears compressed and expanded throughout time). The yellow area shows an example of easier "Synchro" following by the *"Cycle-Time"* based system, while the green area shows an example of even easier following of a classical 1ms-raster.

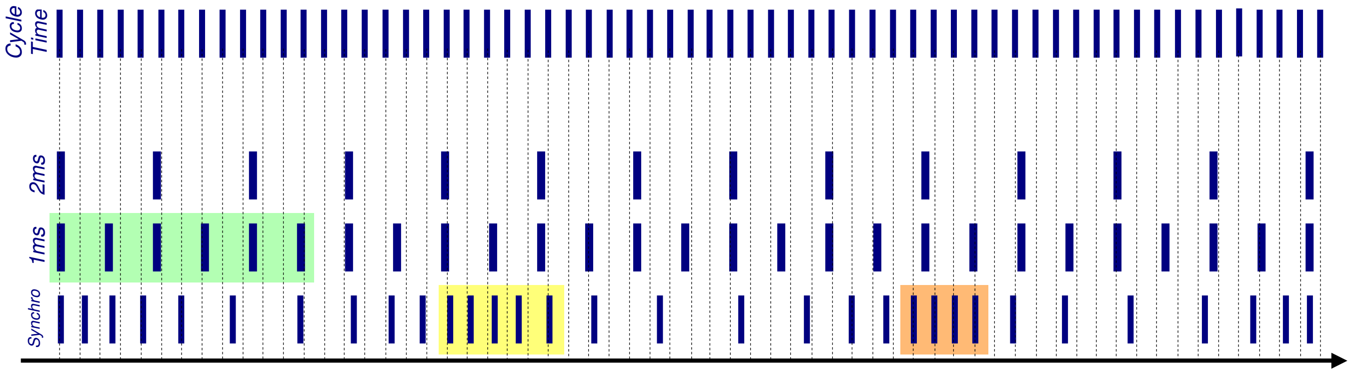


Fig. 4.5:61 – Comparison between two "real-time" systems: classical "time-raster" and this "Cycle-Time" based

Three possibilities of system operation in the time-domain could be herein defined:

- The strict *"Cycle-Time"* (*CT*) compliance to the fastest time-raster (*TR*) at all times, as shown in Fig. 4.5:61, is the most demanding and corresponds technically to "over-sampling" in signal theory. Eq. 4.5:62 shows the strict formal rule to which this system's *"Cycle-Time"* would have to obey. This system would thus be as "real-time conform" as the classical solutions adopted by virtually all the industry.
- The previous strict rule may be more flexible in that the *"Cycle-Time"* may be only fast enough to guarantee real-time behaviour in terms of average short-term execution time (faster during some stretches of time and slower in others). This is materialized through Eq. 4.5:63. The size of the time window inside which average time should be at least as good as in classical systems is not specified at this point, but should be as short as possible, in the tenths of a second range.
- Even more flexibility may be added if something similar to the MS2.8 and MS2.9 Motorsport ECU series [302] is implemented, in that constant "under-sampling" may be allowed for higher engine RPM for the particular case of the "Synchro" raster. Eq. 4.5:64 reflects this. The difference to the previous possibility is that here strict real-time behaviour will be potentially violated for long indefinite stretches of time (as long as RPM are high). This extra flexibilization of the MS2.8 and MS2.9 series will be further explained and detailed immediately below.

$$ct_n \le \min(tr_n)_{0<n<\infty}$$

Eq. 4.5:62 – Possibility #1: 100% strict real-time compliance rule

$$\frac{\sum\limits_{n_1<n<n_1+\Delta} ct_n}{\Delta} \le \min(tr_n)_{n_1<n<n_1+\Delta}$$

Eq. 4.5:63 – Possibility #2: locally averaged real-time compliance rule

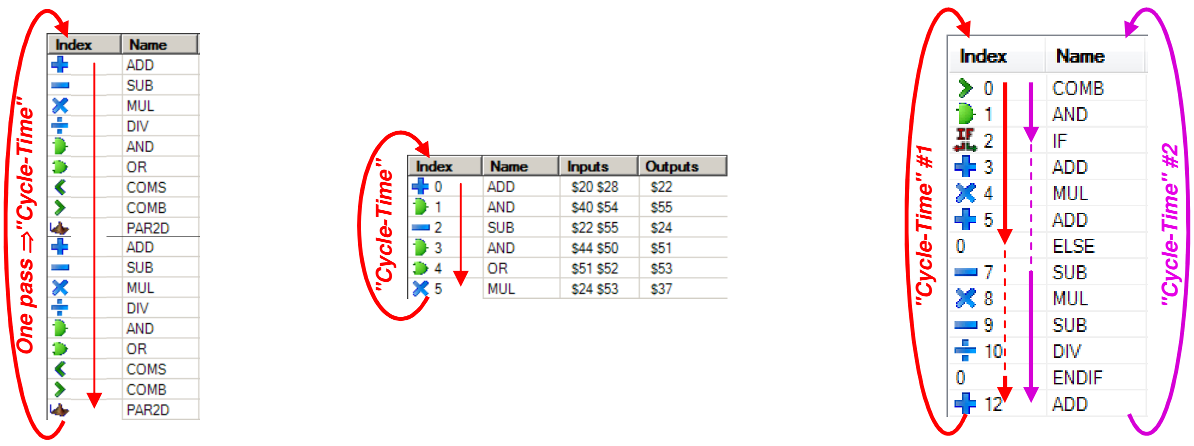
$$\left.\frac{\sum\limits_{n_1<n<n_1+\Delta} ct_n}{\Delta} \le \alpha \cdot \min(tr_n)_{n_1<n<n_1+\Delta}\right|_{\alpha<2}$$

Eq. 4.5:64 – Possibility #3: flexibilized real-time compliance rule

The factor α<2 in Eq. 4.5:64 indicates, as with the MS2.x series, that the *"Cycle-Times"* can be even faster than the classical time-rasters for low RPM and that can reach an out-of-date rate of 50%. This is exactly the possibility that is going to be implemented in the system developed herein, because practical automotive systems do not really need stricter real-time behaviour and because this greatly loosens real-time constraints to the herein developed system, right from the beginning. Furthermore, the *iEditor* will have an internal utility that calculates the expected mean *"Cycle-Time"*, compares it to the really needed strictly minimum real-time constraint depending on the maximum expected RPM. This comparison then results in warnings for the user to know when the system is performing under "sub-optimal" conditions and that eventually more processing hardware has to be added. If processing power does not meet the needs for higher RPM, then FDEF *"Node"* values will be more and more outdated as RPM get higher. Up to a certain point, this is not a problem, as seen in the MS2.x series.

Since the *"Macros"* are self-contained and self-sufficient, and as already mentioned, they could even be "processed randomly", this leads to the even more expanded notion of "loose/asynchronous processing" also already mentioned as hypothetical and not found in any current system. The *"Cycle-Time"* is thus the time that a *"Macros-Sequence"* needs to be completely processed once, as clearly illustrated in Fig. 4.5:65. This time depends on the amount of *"Macros"* to be processed. If there are "IFs" in that *"Macros-Sequence"*, then there will be uncertainties to this *"Cycle-Time"*, as the respective processing will present some time-jitter, depending on which "IF" block gets processed upon each pass. Among the developed *"Macros"*, the following operations are strictly time-dependent, while all others do not have any time dependency:

- Filters (FILT, FILF)
- Integrators (INTT, INTF)
- Differentiators (herein not implemented)
- Delays (DLYU, DLYD, DLYA)
- Timers (TIMU, TIMD)



Index	Name
	ADD
	SUB
	MUL
	DIV
	AND
	OR
	COMS
	COMB
	PAR2D
	ADD
	SUB
	MUL
	DIV
	AND
	OR
	COMS
	COMB
	PAR2D

Index	Name	Inputs	Outputs
0	ADD	$20 $28	$22
1	AND	$40 $54	$55
2	SUB	$22 $55	$24
3	AND	$44 $50	$51
4	OR	$51 $52	$53
5	MUL	$24 $53	$37

Index	Name
0	COMB
1	AND
2	IF
3	ADD
4	MUL
5	ADD
0	ELSE
7	SUB
8	MUL
9	SUB
10	DIV
0	ENDIF
12	ADD

Fig. 4.5:65 – Illustration of the "Cycle-Time" of a "Macros-Sequence" in the system developed herein

Calculations involving time-dependent operations were historically always placed inside well-defined time-rasters to simplify their calculations. These operations then use the well-known time-raster duration as the discrete timing delta Δt inside the respective formulas. Filtering, integrating, delaying and timing can easily be implemented, even under varying time-raster durations, with a small and mostly negligible timing "uncertainty"[*]. Formulas using this fixed Δt can easily be converted to also work correctly in the raster-less system by using the *"Cycle-Time"* as the known duration. They even work correctly if the *"Cycle-Time"* varies in time. This *"Cycle-Time"* duration will only be dependent of the FDEF load upon the processing hardware, since all interrupts- and "Synchro"-based processing will be outsourced to external *"Smart Peripherals"*, as explained later on.

On the other hand, in standard systems, a fixed time-raster such as the 10ms one, grows in length as the engine-speed increases. This is due to also increasing "Synchro" raster execution time needs, which by being higher-prio thus get into the way of that fixed raster. Care must be taken for this fixed time-raster to not pass any dead-line and start causing resets. An intelligent mechanism is used in the MS2.8 and MS2.9 Motorsport ECU series [302], where a similar *"Background-FOR-Cycle"* time measurement is used (see Fig. 4.5:66). Additionally, this MS2.x ECU series software takes a very drastic action as soon as the engine-speed surpasses the 9.000RPM threshold: it simply equals each second value to the first one, thereby calculating new values in only every second passage (see Fig. 4.5:67). This applies to the most important raster of them all: the *"Synchro"*, thus keeping the other fixed rasters executing under the dead-line again.

Nevertheless, this method has proven itself as efficient in the sense that it allows the ECU to go up to 18.000RPM for engines used in the most renowned race-car series. It was also empirically proven that it does not add significant engine performance to have every calculation done in every single raster, being half the calculations perfectly sufficient to drive an engine at full torque and power and high speeds. As with the *"Cycle-Time"*, here the filter, integrator, delay and timing calculations also take the last "Background-FOR-Cycle" time into full account. NOTES: only the higher-level *"Synchro"* calculations are slowed down, whereas the low-level ignition and injection synchronized driver code operates all the time with little processing requirements. Also note that the blue *"Synchro"* calls are only shown in their density relative to the real RPM at the moment, not being thus shown more frequently as the RPM increase (this allows an easier comparison of relative *"Synchro"* densities among different RPM values).

[*] *In quotes, since this "uncertainty" is just a functional one and not a strict internal error*

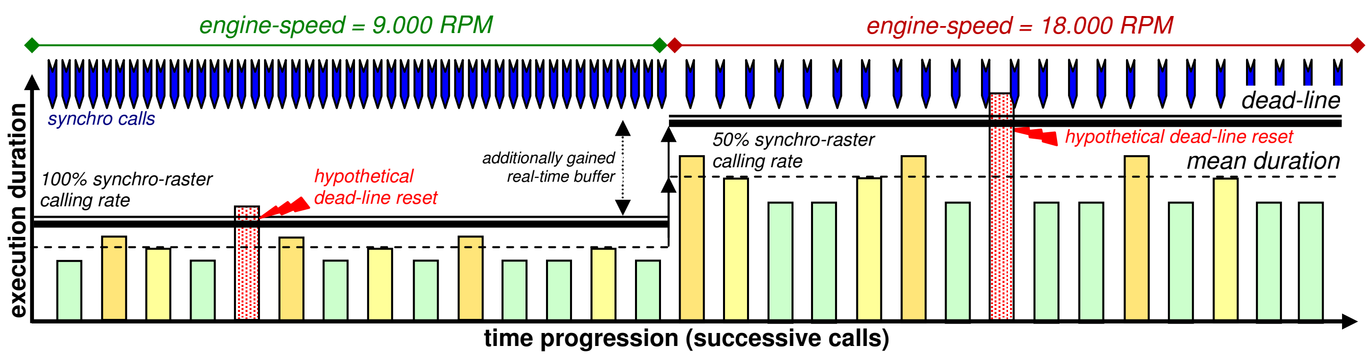


Fig. 4.5:66 – Illustration of the "Background-FOR-Cycle" time discrete variability @9.000/18.000RPM on the MS2.x

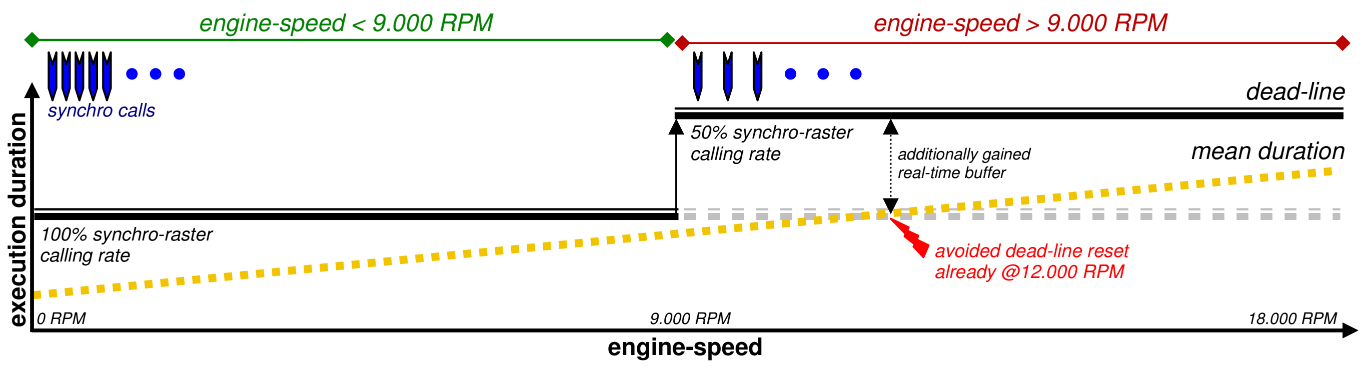


Fig. 4.5:67 – Change in the calling rate of the "Synchro" raster with increasing engine-speeds

In the system developed herein, in extreme situations where the engine-speed (RPM) gets so high that the *"Cycle-Time"* are effectively higher than the correspondingly higher processing frequency required by a certain ignition, injection and knocking FDEFs, this solely means that the *"Nodes"* and their corresponding values do not get updated as fast as they should. This only affects central ECU processing FDEFs, since real-time critical FDEFs will be placed inside external dedicated *"Smart Peripherals"*. This means that the values peripherals (actuators) are using are periodically increasingly outdated as the engine-speed is higher (see Fig. 4.5:68).

Resulting *"Cycle-Times"* are always stable since no special treatment is given to the *"Macros-Sequences"* execution, while that execution is totally unaware of the engine-speed itself. Compared to the previous methods where relatively large changes in raster execution call frequency and duration occurs, this allows a much smoother and jitter-free processing. There will also be no dead-line resets whatsoever. As readily seen in Fig. 4.5:69, the only thing that will happen with increasing engine-speeds, is that the *"Smart Peripherals"* (SP) will be fed with old high-level values such as fuel quantity, spark energy, spark angle, etc., which is not critical at high speeds, as seen from the MS2.x long-term experience.

Fig. 4.5:70 enhances the "over-/under-sampling" that occurs depending on the engine-speed and, thus, depending on the *"Smart Peripherals"* needs for updated values from the central FDEF calculations, through the green and red triangles, respectively. On the top of this figure, the "Synchro" update misses are shown in grey and are, for this graphics only, in real relation to linear time.

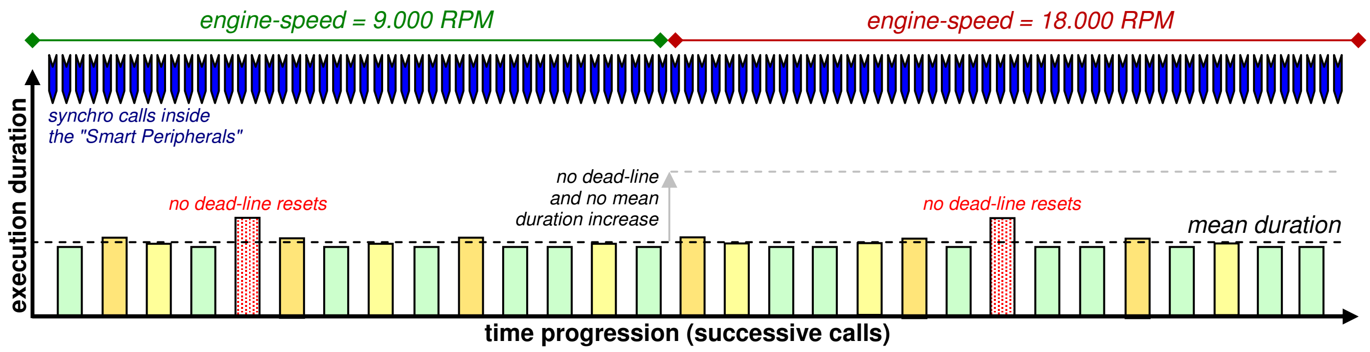


Fig. 4.5:68 – Illustration of the hypothetical "Cycle-Time" discrete variability @9.000/18.000RPM

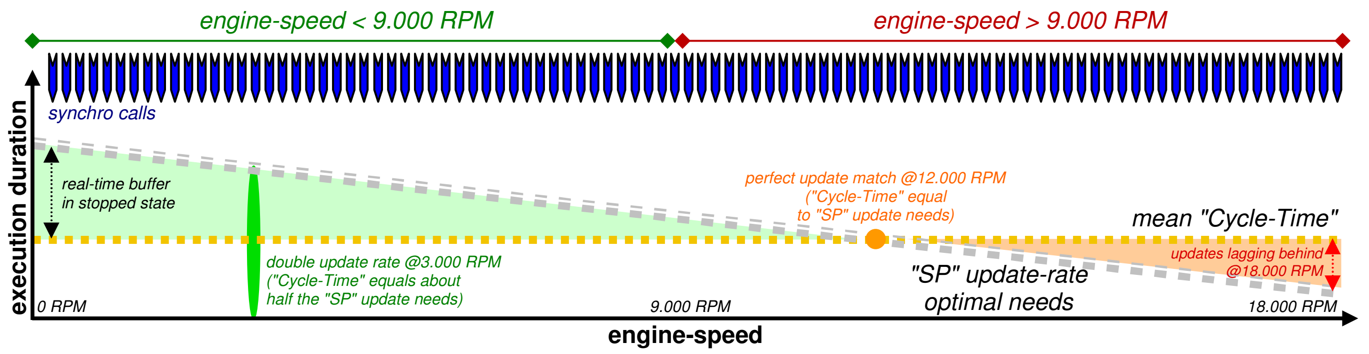


Fig. 4.5:69 – Illustration of the inexistent critical system failure with increasing engine-speeds

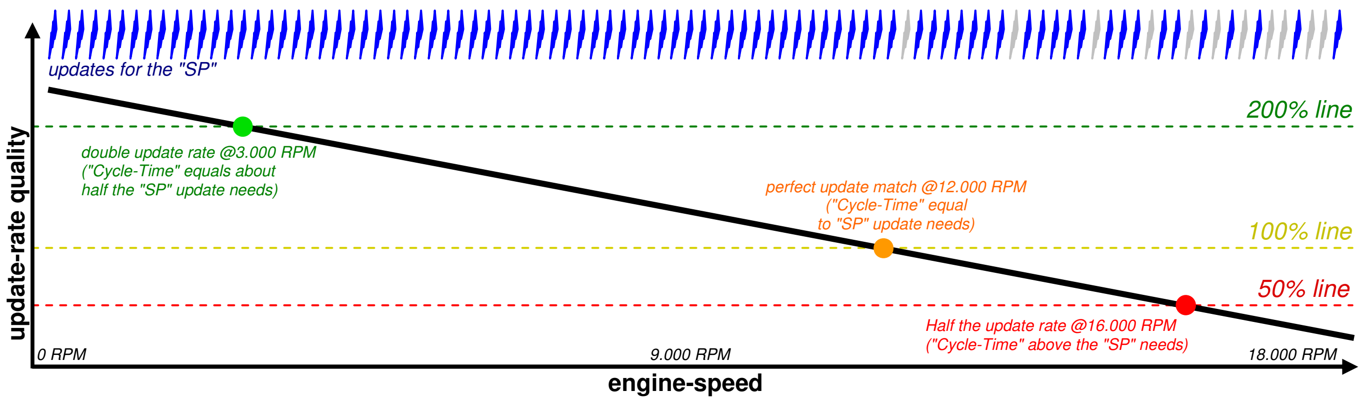


Fig. 4.5:70 – Emphasis of the update-rate degradation towards increasing engine-speeds

Fig. 4.5:71 illustrates this best for delays and filters, by comparing with the classical time-raster methodology of those timing operations. While a normal DELAY operation sees its internal timer-counter steadily incremented within a fixed frequency (e.g. 10ms-raster) herein we have the accumulation of variable *"Cycle-Times"* until accumulation goes over a threshold. The worst "timing uncertainty" will be equal to the longest *"Cycle-Time"*. In the case of a FILTER operation, the normal case is to have it processed over equally spaced time moments (e.g. 10ms-raster), while herein this is done over *"Cycles-Time"* moments by compensating internally for the their. There will thus be no "timing uncertainty" in this case, leaving only the relative problem of these filters probably getting into trouble when *"Cycle-Times"* get too small, where numerical inaccuracies may occur. This is due to the internally used division with Δt. Avoiding this is fairly simple: the *"iEditor"* may set a counter that counts a minimum amount of *"Cycle-Times"* to progress until the *"Filter Macro"* would

be called, thereby guaranteeing a good amount of time for its internally used Δt. Note that although the FILTER *"Macro"* output progression seems sloppy, it is still correct since it progresses according to the (here exaggerated) different *"Cycle-Time"*.

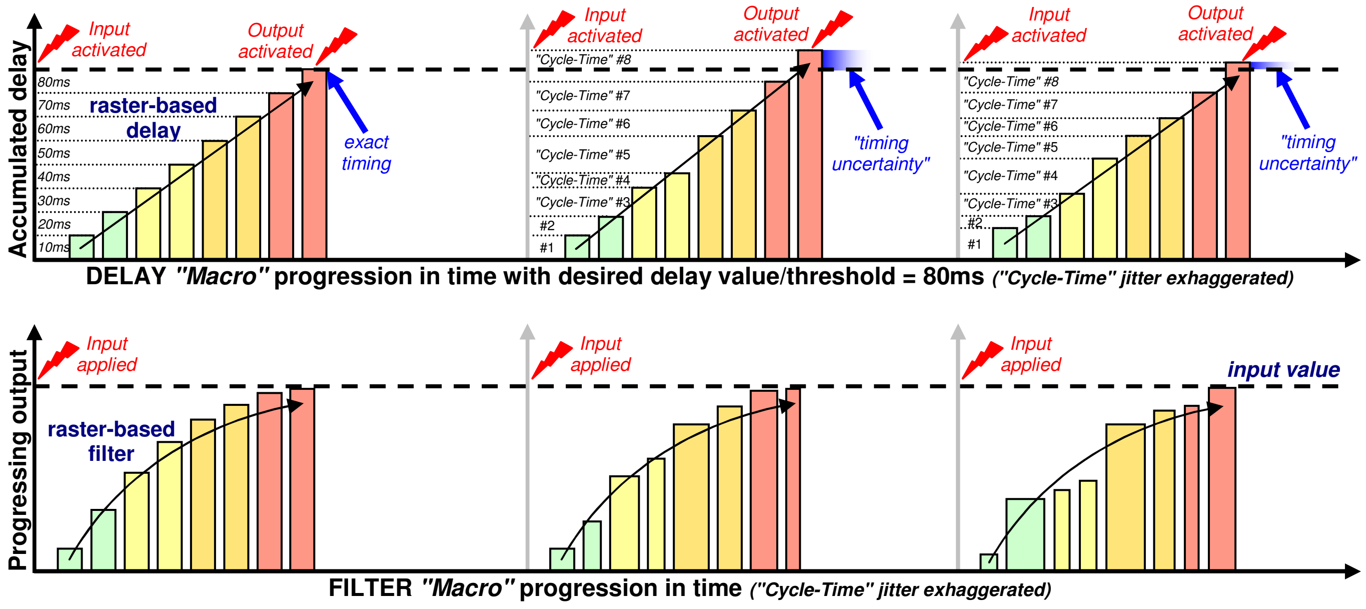


Fig. 4.5:71 – Illustration of "Cycle-Time" based delaying and filtering, compared to classical time-raster methodology

The only way to keep these functional "timing uncertainties" on DELAYs under strict control is to keep the maximum expected *"Cycle-Time"* smaller than, say, 1ms. In practice, filtering, delays, etc., are done much slower than 1ms, while most timing values are not needed to be more accurate than 1ms. In reality, 1ms is the smallest "time-quanta" used in most ECUs. Furthermore, most timing values in ECUs have a quite comfortable tolerance range that encompasses any functional timing "uncertainties" there might occur in the work developed herein. Either way, normal FDEFs do not even require so much timing precision, since their processing in raster systems already introduces much larger "timing uncertainties" than those mentioned above. For example, a full 10ms-raster may process the last FDEF already 9ms off the main starting point in time, while sensor inputs get that old as well. Any DELAY operation in that last FDEF will activate its outputs only 9ms after the main timing point. The important thing is to keep everything always inside the same time-raster call, to obviously avoid "deadline resets", not much more than that. These 9ms system delay represent 90% of "deviation", while *"Cycle-Times"* around 1ms would already reduce this "deviation" (previous "timing uncertainty") down to only 10%. As said before, FILTERs are not affected by these "deviations".

It should be noted that this thesis' work is clearly taking advantage of the nature of the automotive control mechanisms. These are highly based on feed-forward signal-processing paths that simply take values from sensors, process them and output resulting values to actuators (see Fig. 1.2:7). This so-called "data-flow" limits itself in doing exactly what the *"Macro"* was designed to: taking input values, processing them and delivering the outputting value to the next *"Macros"*. This is generally repeated throughout the whole FDEF mesh inside an ECU. As already illustrated in Fig. 4.5:70, this raster-less processing mechanism thus naturally resists to high engine-speeds by gracefully degrading its performance and by going beyond the "strict real-time threshold" (Fig.

4.5:72). This graceful degradation is translated into the engine getting progressively "worse" and "outdated" values, but with the related and highly useful feature of never leaving the engine without values to work with.

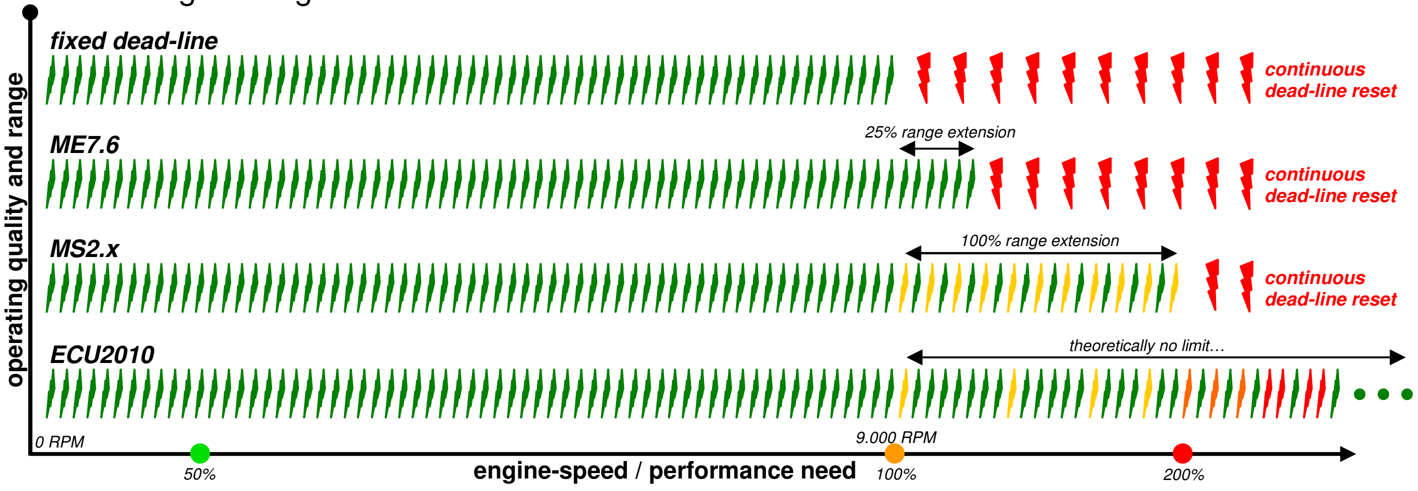

operating quality and range
fixed dead-line
continuous dead-line reset
ME7.6
25% range extension
continuous dead-line reset
MS2.x
100% range extension
continuous dead-line reset
ECU2010
theoretically no limit…
0 RPM
9.000 RPM
50%
engine-speed / performance need
100%
200%

Fig. 4.5:72 – Fatal vs. graceful performance degradation in systems going beyond the "strict real-time threshold"

Finally, this local sense of time, needed for the time-sensitive *"Macros"* listed earlier, will be implemented with the use of a local RTC (real-time clock) that feeds the *"Macros-Processor"*. If more than one hardware module is used, each will have its own independent local RTC, without any further inter-module synchronization.

To improve cycle time based on the relative importance of each "*Macro*" calculations, the developed solution uses "priority-bits" (as *"Macro"* modifiers) inside each *"Macro"*. The *"Macro-Processor"* simply keeps track of the current priority to be executed and goes through the entire *"Macros-Sequence"* once. Then, it continues with the next priority, if the current one is finished. In other words, it's just as having rasters, but instead of these having a multiple of the cycle frequency, here they have a duration/period that is a multiple of the overall *"Cycle-Time"*.

Assuming *"Cycle-Time = 1[ms]"*, Fig. 4.5:73 shows a sequence being executed in 1[ms], with some segments having priority 1 (executed on every 1[ms] pass ⇒ virtual 1[ms] raster), while others have priority 2 (executed on every second 1[ms] pass ⇒ virtual 2[ms] raster) and still others have priority 10 (executed on every tenth 1[ms] pass ⇒ virtual 10[ms] raster). These "priority-bits" are not explicitly set by the developer, but rather set automatically by the Editor after the developer tells it how slow each *"Macro"* is allowed to execute. Depending on the internally calculated *"Cycle-Time"*, the editor then decided which "priority" should be used. For example, if the developer sets a maximum execution period of 5ms and *"Cycle-Time = 850[µs]"*, then "priority-bit" [5x] is automatically set (4.25[ms]), since the [10x] would produce 8.5[ms]. This manual setting is further eased by the *"Live-Priorities"* mechanism, detailed later, that automatically assigns all "priorities" from the first manual settings on. The possible "priorities" could be as follows, to allow for some flexibility in this manual and automatic virtual raster settings: 1x(0), 2x(1), 5x(2), 10x(3), 20x(4), 50x(5), 100x(6), 1000x(7), thus occupying 3 bits within the *"Macro"* coding. Centralized "priority" execution-pass tracking counters would then be polled after each fetched *"Macro"* and compared to the *"Macro's"* own "priority-bits" to easily determine if that *"Macro"* should be executed on that particular pass or not.

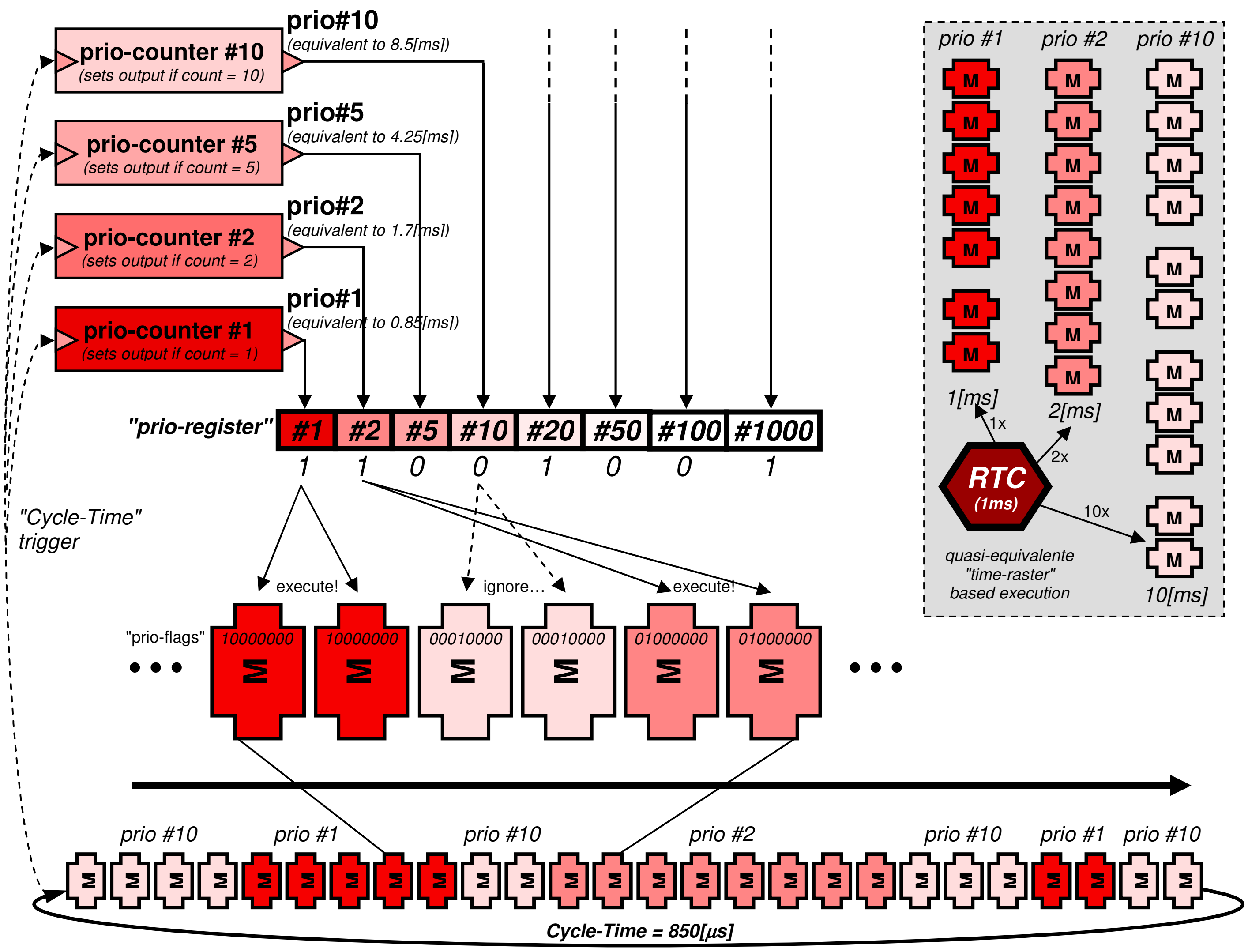


Fig. 4.5:73 – "Macro" segments with different "priorities" being executed within different periods (virtual time-rasters)

As seen in Fig. 4.5:73, the *"Macros-Sequence"* is continually executed from beginning to end. Upon every new beginning (every new *"Macros-Cycle"*), the "priority counters" are incremented accordingly. Each one of those counters will set its output upon reaching the counts 1, 2, 5, 10, 20, 50, 100 and 1000, respectively. When set to one, the counter again resets the output to zero upon the very next *"Macros-Cycle"*, so that the "1" is only present for that particular *"Macros-Cycle"*. Then, upon every *"Macro"* execution, the *"Macro's"* priority property is checked and directly compared to the "priority register". If there is a match of one of the "prio-flags", then the *"Macro"* is executed. If not, the *"Macro"* is simply ignored and the next *"Macro"* is fetched for potential execution.

Note: if the *"Cycle-Time"* of the complete *"Macros-Sequence"* is sufficiently low to accommodate everything inside the FDEFs, then no "priotization" would even be necessary. Exception made to those *"Macros"* that need some time to correctly process, such as filters, integrators and differentiators, where the "deltas" cannot be so small that the numeric calculations cause too large rounding errors.

FINAL NOTE: All classical systems which must guarantee real-time operation rely on time-rasters or any other accurate time-slicing mechanism, forever tying the whole system to this artificially fixed nature of time.

From the statements above, ***PITFALL C#43***(Time-Rasters) and ***PITFALL C#7***(Event-Driving) are thus effectively eliminated through effective use of the innovative "execute as fast as possible" paradigm on automotive systems.

4.5.3.7 OS/Runtime-less Processing

[Emphasis: direct non-managed processing]

Basically, operating-systems control/give access to software and hardware resources, while also controlling the way tasks are executed in time (scheduler). State-of-the-art hardware systems make extensive and standard use of operating-system software inside them. ECUs, prototyping hardware, add-on modules, etc., virtually all employ some kind of OS, which makes up the interface between the hardware and the firmware inside them.

As stated previously, the present system just keeps simply executing the *"Macros-Sequences"* without any special supporting mechanism other than the *"Macros-Processor"* itself. One consequence is that it does not need an operating-system to control time-rasters, memory-allocation and other details, since these simply do not exist. There is only the need of a mechanism by which the *"Macros-Sequence"* is fed to a special ALU/CPU combination, which processes those *“Macros”* and also executes the necessary read/write operations. Besides this, a program-counter is also obviously needed to keep track of the current and next *“Macro”* to execute. Aside from this, there is no need for any memory-allocation or any other kind of internal “operating-system” typical mechanisms, thus simplifying the overall efforts to implement, understand, program and debug such a system. More details in [Att. 32].

FINAL NOTE: Most if not all classical systems use operating-systems, OS-like objects and runtimes for hardware wrapping and abstracting purposes, ever-distancing relationships between application SW & HW.
From the statements above, ***PITFALL O#38***(Operating System) is thus effectively eliminated through the strict system construction without any operating systems.

4.5.3.8 RESET-less Processing

[Emphasis: never-interrupted processing]

Since this system does not have an operating-system, as explained earlier, or any other mechanism of controlling anything other than keeping the CPU busy processing the *"Macros-Sequences"*, there is no source of resets in this area whatsoever. Furthermore, as there are no rasters, as also explained earlier, and no related need to supervise those in order to issue an emergency action upon dead-line problems, there is also no need of resets in that respect. There is also no other physical resetting mechanism in the entire system, so that resets during normal hardware operation are completely impossible. More details in [Att. 33].

FINAL NOTE: Classical systems allow resets and restarts in very specific situations, most of which relate to catastrophic events during processing and which are linked to not having any other solution to those events.
From statements above, ***PITFALL H#36***(Error Recovery & Resets) is thus effectively eliminated through a new “execute as fast as possible” paradigm where the system is strictly incapable of generating any deadline errors/resets.

4.5.3.9 Tweak-less Programming

[Emphasis: programming without construct side-effects]

This feature is more of a detail aspect of the *"Macro-Programming"* graphical language: since there are no syntactical needs and no alternatives to the available graphical *"Macro"* elements on the user-interface, and since the developer basically has only one unique way of placing and using any operation. There is no possibility, at least at the *"Macro"* language level, for the developer to tweak or somehow influence the low-level execution details of those *"Macros"*. With *"Macros"* there is no need or possibility of any hand-optimising either. More details in [Att. 34].

FINAL NOTE: Classical systems rely on tweaking somewhere and at some point, because they are so complex that automatic mechanisms cannot always know how to do things globally in the most efficient way.

From the statements above, ***PITFALL O#14***(Code Tweaking) is thus effectively eliminated through the strict non-need to do so within the *"Macros-Sequence"* paradigm*.*

4.5.3.10 Compiler/Assembler-less Program Build

[Emphasis: programming without any internal slack]

The hardware developed herein directly processes every single *"Macro"* that appears on the downloaded *"Macros-Sequence"*, while this *"Macros-Sequence"* is the central language entity from where everything including graphical FDEFs emerges. Therefore, there are no translators anywhere in the entire development structure and also anywhere in the hardware platform. Thus, the entire building process relies on an "always-ready" intermediate language (*"Macros-Sequence"),* which goes both ways – graphical interface and hardware platform – without modification. The development environment only has to decide on which processing unit it must deposit which *"Macros-Sequence"* parcel, while preparing special hardware *"Nodes"* interchange tables for values inter-communications between those processing units. This final stage is done by a straightforward *"Allocator"* prior to downloading and will be explained later. More details in [Att. 35].

FINAL NOTE: All classical systems rely on more or less translation and management mechanisms inside them. The more translators and managers exist, the more systems get globally incomprehensibly complex.

From the statements above, ***PITFALL C#11*** (2/2 - Translators) and ***PITFALL O#39*** (Managed Code) are thus effectively eliminated through the intrinsic fact that the *"Macros-Sequence"* are the *in-facto* only code representation.

4.5.3.11 "IF"-less Jitter-less Timing

[Emphasis: deterministic timing behaviour through pure data-flow]

As already hinted in Fig. 4.5:36, the "IF" operation can completely disappear and be replaced by more data-flow friendly operations such as multiplexing switches, in most cases. The implementation of the *"IF Macro"* will be detailed later on herein. The main advantages of having this global replacement and way of thinking through data-flow programming, can be listed as follows:

- ***No more control-flow*** → Elimination of all control-flow paths from the otherwise very homogeneous data-flow paths, therefore leaving no highly disruptive traces and being much more data-flow nature conform. An FDEF therefore turns out to be a pure data-flow one.
- ***No more timing jitter*** → By eliminating all control-flow paths and replacing them with other more data-flow compliant structures, the disruptive nature of some data-flow paths being processed and other not, it also eliminated. Having all data-flow paths now always processed, therefore eliminates the processing timing jitters that would otherwise occur.
- ***No more complex visual operations*** → By eliminating the need for these control-flow *"IF Macro"*, the relative complexity of its visual representation is also eliminated. This visual representation needed to display both "TRUE" and "FALSE" blocks in an appropriate manner and additionally be able to allow the developer to act and change it according to his needs.
- ***No more disruptive jumps/gotos*** → By eliminating the need for these control-flow *"IF Macro"*, the relatively disruptive need for extra hidden jumps/gotos at machine-level is also eliminated. These jumps/gotos needed to be automatically added by the *"iEditor"* for the "IF" to operate.

This difference of having control-flow or not is going to be illustrated by using ASCET [5], which allows to combine control-flow and data-flow paths into one single view. Naturally, it also allows for pure data-flow FDEFs. Fig. 4.5:74 illustrates this difference visually.

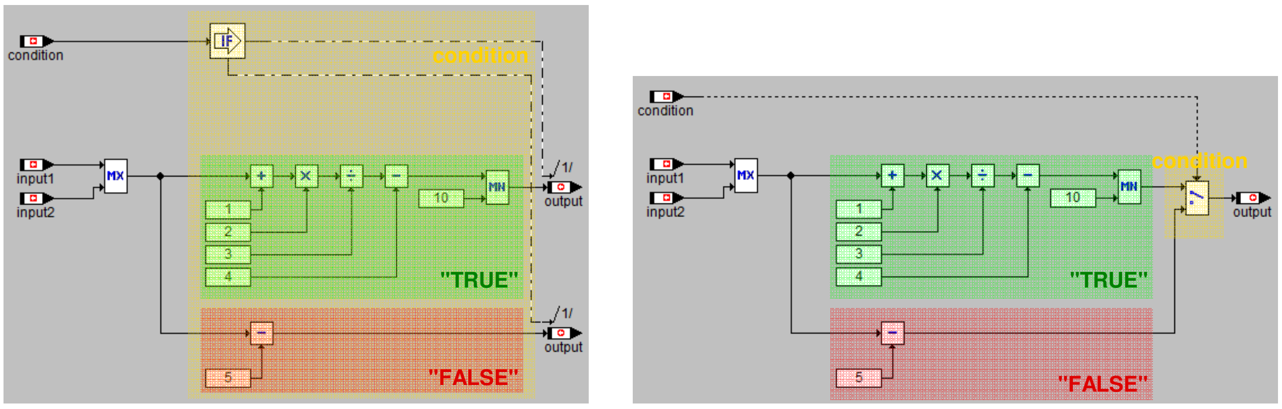


Fig. 4.5:74 – Left: control/data-flow mixed FDEF; Right: pure data-flow FDEF

Note how the control-flow "IF" encompasses the "TRUE" and "FALSE" blocks, while the pure data-flow version separates everything. As stated above, another difference is that both paths are executed in the pure data-flow version. This makes programming automotive systems much more predictable concerning timings, therefore also increasing determinism of the those timings and thus of the global results.

Also to mention the apparent visual "multiple-assignment" problem that arises with use of "IFs" on the left of Fig. 4.5:74. Using constructs like those on the right of Fig. 4.5:74, guarantees a much more stable *"Cycle-Time"* for that particular *"Macros-Sequence"*. This is because contrary to the left of Fig. 4.5:74 where normally code only processes green or red portions but not both (due to paths with "IF-THEN-ELSE" having the decision made immediately at the beginning), on the right of Fig. 4.5:74 both green and red paths are naturally always processed (due to the fact that only at the final switching operation will be known which path is to be taken at last). Within an hypothetical solution of having 1 CPU per *"Macro"* such as in Fig. [Att. 60]:235 would even turn a system into an intrinsically "always all operations" processed system. More details in [Att. 36].

FINAL NOTE: Virtual all classical systems possess highly jitter-affected execution cycles, therefore turning timing problems debugging even more difficult, compared to low or negligible jitter values.

From statements above, ***PITFALL C#46***$^{\text{(Timing Jitter)}}$ is thus effectively eliminated through disciplined non-usage of "IF" control-flow operations, replacing them with clean and straight-forward data-flow equivalents.

4.5.3.12 Formatted Program Entry

[Emphasis: preventive programming, treat errors before they affect the system]

This feature is loosely related to the tweakless programming mentioned earlier. Since the *"Macros"* have no options also at the visual level, adding them to an FDEF is just as simple as it sounds: drag them onto the desired position of the FDEF, connect the desired *"Nodes"* to it, ready, go to next *"Macro"*. Either doing this with the mouse or through the command-line, no deviation from the correct path is possible.

This basic correct usage by design is another feature that allows for massive reduction of time wastes and bugs during development. This resembles the "hygienic macros" [337] where the programmer must conform to certain constructs, to avoid the program getting a mess altogether. The *"Macros"* and the *iEditor* devised in the work herein go a step further, in that no deviations from a unique path are even possible. This could be called "extreme hygiene" or "intrinsic hygiene", since it takes "hygienic programming" to the most advanced level. Last but not least, there is no need to waste time maintaining the cleanness of the source-code itself, since it can only be represented in one unique way. More details in [Att. 37].

FINAL NOTE: All classical systems rely on some large set of rules when the developer programs in "C" code or when he designs visually. Since these systems allow a multitude of possibilities and flexibility, these rules must exist, as well as the corresponding error messages during this design/build phase.

From the statements above, ***PITFALL H#34*** (Coding Alternatives), ***PITFALL C#41*** (Code Hygiene) and ***PITFALL C#58*** (Building Messages) are thus effectively eliminated through effective and intrinsic enforcement of very simple visual design rules, which do not allow for any graphically disruptive errors.

4.5.3.13 Architectural/Implementation Correctness

[Emphasis: execute an implementation always in exact the same way]

The joint facts that the *"Macros-Processor"* proposed herein possesses the feature of working in a "micro-rebooted" and "state/context-less" manner, executing "hermetically closed" entities, where only the input *"Nodes"* have influence on the result, make up for the guarantee that the *"Macros"* will always be executed in exact the same way. There is no need whatsoever to mathematically prove this design in what state and context matters, since there is none. Classical processors and even special ones [432] have to still undergo extensive testing and verification, to rule out many possibilities of something going wrong due to state and context.

As will be seen later in the sections detailing the implementation of the system model both on micro-controllers and FPGAs, verifying that each single *"Macro"* works in the way it should and for all possible inputs, is of course still needed. What's guaranteed by design due to the *"Macros"* nature, is that when a certain *"Macro"* is executed correctly with a certain set of input *"Nodes"*, that same *"Macro"* will always execute correctly given those same inputs. This *"Macro"* execution determinism totally eliminates the need for traditional extensive and even exhaustive low-level verifications, to be sure that the processor is really doing what it was programmed to. At this level, even two sub-levels may be identified: the lowest processor level itself (processor correctness) and then the language level (implementation level). The first relates to hardware and/or micro-code bugs (traditional erratas must be watched for), while the second relates to implementation bugs in tool-chains (mainly inside compilers) and also human interpretation mistakes.

The *"Macros"* exist at the implementation correctness level and thus eliminate problems there, leaving only the algorithmic and program sequence details to be checked. By "algorithmic" it is meant the way humans devise the solution to be implemented, the sequence of operations/instructions/*"Macros"* needed to accomplish their goal. At this level, it is not possible to eliminate human error, of course, unless complete and previously tested *"Macro-Packages"* would be delivered, where the developer would only place and connect them without the ability to chose among single *"Macros"*. But this is not flexible enough and the automotive functionality developer must be able to pick single *"Macros"* and devise all kinds of solutions and possibilities. Fig. 4.5:75 illustrates all these main correctness levels, by comparing classical systems with the system developed herein, by taking the currently more advanced visual programming model with automatic code-generation. The crossing blue arrows indicate the conceptual correspondence from the classic system to the system developed herein. It also contains the even lower-level "physical correctness" level that reflects potential physical hardware wiring, silicon and packaging problems.

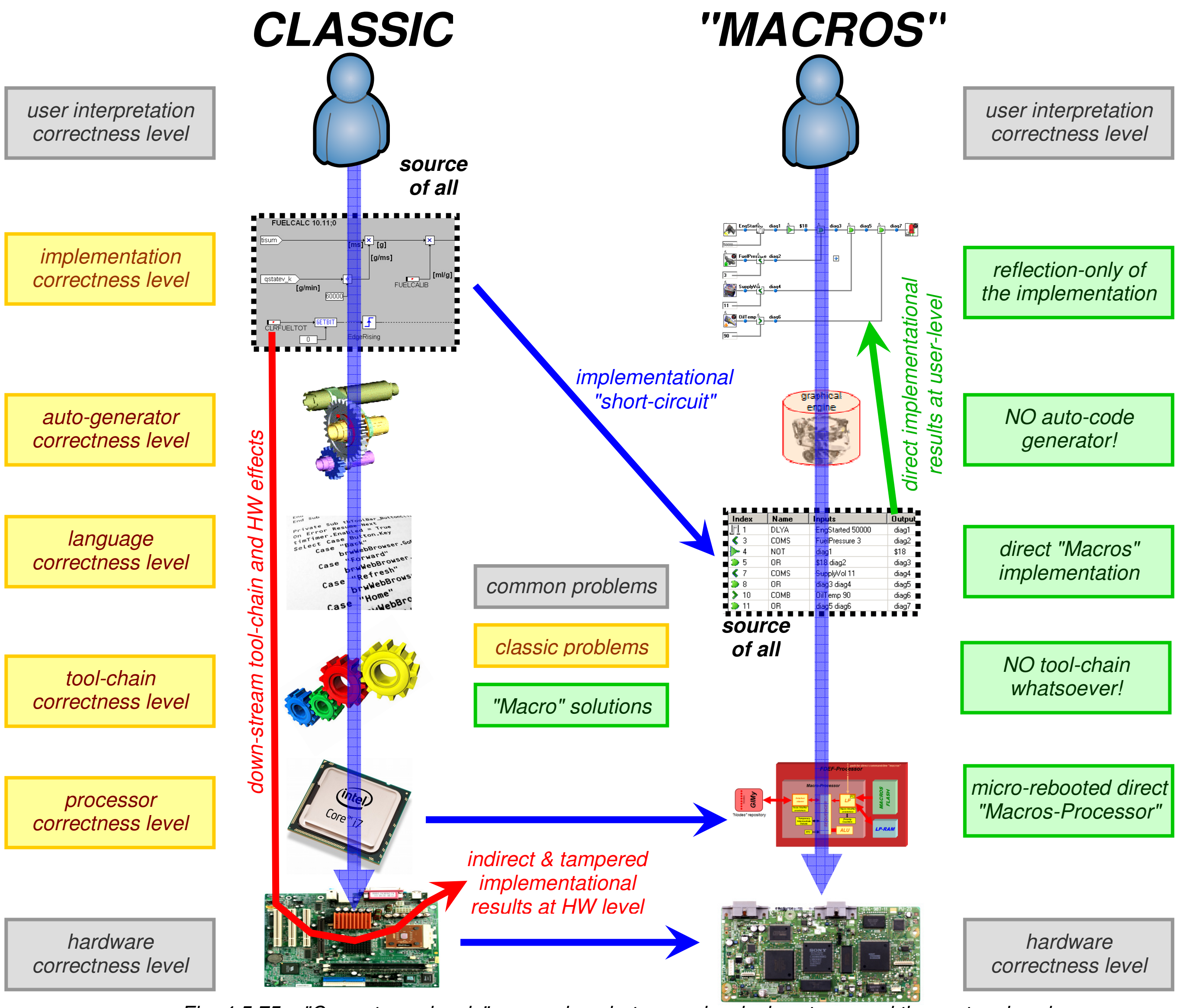

CLASSIC
"MACROS"
user interpretation correctness level
user interpretation correctness level
source of all
implementation correctness level
reflection-only of the implementation
implementational "short-circuit"
direct implementational results at user-level
auto-generator correctness level
NO auto-code generator!
graphical engine
language correctness level
direct "Macros" implementation
common problems
classic problems
"Macro" solutions
source of all
tool-chain correctness level
NO tool-chain whatsoever!
down-stream tool-chain and HW effects
processor correctness level
micro-rebooted direct "Macros-Processor"
indirect & tampered implementational results at HW level
hardware correctness level
hardware correctness level

Fig. 4.5:75 – "Correctness levels" comparison between classical systems and the system herein

Within the system herein, not only are the correctness levels effectively less, but more levels are intrinsically more correct by nature. For one, only classical systems need auto-code generators, while the system herein works in a reverse way when compared to the classic top-down approach. In a classic system, the developer's visual input is uni-directionally translated into some classic programming language such as "C". <u>There thus is no way of the user knowing what the implementation outcome of this hidden translation is or to immediately verify/validate it, until the system's behaviour indirectly (HW level) reflects this translation after many other often unpredictable and tampering down-stream tool-chain and hardware effects</u>. In the system herein however, visual programming actions (see *"Live-Dragging"* from Fig. 5.1:4 onwards) will first be translated into the central *"Macros-Sequence"* and it is this exact central language that will directly (user-level) reflect the final visual implementation result for the developer to precisely see the implementation outcome and verify/validate it immediately, all this through the "graphical engine" (Fig. 4.5:50). In other words, the central *"Macros-Sequence"* really is the implementation-level and not the visual representation of it through the "graphics engine". This "graphics engine" is only an auxiliary mechanism to accept visual developer actions/programming and to reflect the central-role playing *"Macros-Sequence"* in a more intuitive/human way. *"Live-Dragging"* fully interactive editing mechanisms (Fig. 5.1:4) detailed later on herein, allowing the user to see the implementation results of its visual programming actions upon the central *"Macros-Sequence"*, clarify this auxiliary visual representation even further.

All this originates an implementation "short-circuit" as clearly illustrated in Fig. 4.5:75, whereby the developer is, in reality, programming and immediately knowing what's the implementation outcome, at *"Macros-Sequence"* level. This thus eliminates all unpredictable and tampering down-stream tool-chain and hardware effects mentioned above for classical systems.

This *"Macro"* implementation determinism is very important in critical systems such as automotive ones, since it eliminates an important verification need while implementing, thus boosting productivity. All this is strongly related to ***PITFALL 0#65*** explained earlier herein. It even is somewhat related to FPGA synthesis tools allowing the developer to make "timing assessments", where a timing simulation is ran and final timing results for a particular implementation are then displayed for human verification. These tools also rely on highly homogeneous and, above all, deterministic behaviour of the logical elements comprising an FPGA. They know exactly how much delay results from every logical element, also knowing exactly where they are placed and in which sequence, etc., to be able to "relatively easily" sum up the final displayed results. However, it must be said that here, such "profiling" tools operate on the final synthesized implementation and not on VHDL/SystemC directly. Likewise, highly heterogeneous and intricate compiled language systems do not allow for "profiling" tools to operate directly on the implementation language, also having to rather operate on the final machine-code result as well. The system devised herein only has the *"Macros-Sequence",* no classical translational tool-chain and, thus, "profiling" operates directly on the *"Macros"*, further simplifying the complete system. More details in [Att. 38].

4.5.3.14 Smart Software/Hardware Partitioning for Parallel Processing

[Emphasis: placing the FDEFs in the right places simplifies everything]

Automotive software and hardware partitioning never has been easy nor has it been a pacific subject. Several attempts to devise a parallel processing structure for automotive functionality and respective drivers, have been deemed as too complex and expensive to implement, program and maintain. These include academic and OEM attempts to do so, further discussed in detail in [Att. 39]. Regarding the commercially and academically most visible parallel processing concerns in respect to SW and HW, while master-slave and port-expansion configurations are the easiest to program, debug, deploy and manage, multi-core configurations are the hardest. While the first ones have independent software sides separated by higher-level values and while neither side can actively disrupt the internal operation of the other (comparison: two off-board engines running on the same boat), the last ones must exchange many lower-level variables, depending on each other to be able to even operate correctly.

The *"Macros-Sequences"* and the resulting *"FDEF-Processor"* discussed herein, are to change this scenario a bit, making "Multi-Core in-a-system" attractive again, but with some major differences: instead of an heterogeneous system where each CPU executed very specific and distinctive tasks, the proposed system is more of a kind of "Multi-System" nature. This "Multi-System" is distinctive of a "multi-computer" (as in GreenArrays' "GA144" chip [461] and Transputer systems [148] [151] [413]) in that each "system" does not only process its algorithms and data, but also inputs/outputs data from/to full-fledged peripherals attached to it. "Multi-computer" systems often only have limited peripheral capabilities, mostly to communicate between CPUs only. Fig. 4.5:76 depicts these 6 different but similar ways of parallelizing automotive systems.

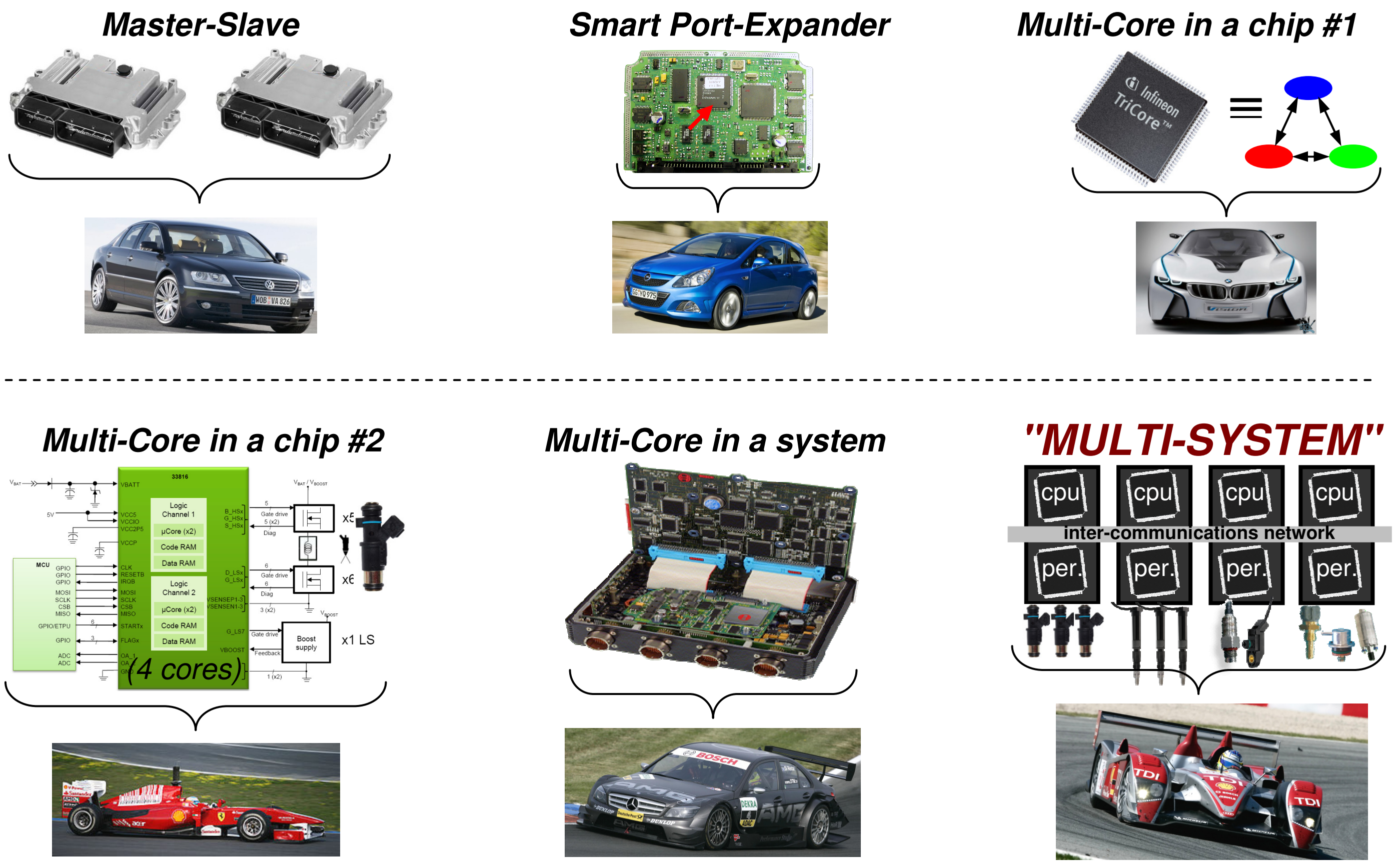


Fig. 4.5:76 – The 6 different and yet similar ways of parallelizing automotive systems from Tab. [Att. 39]:135

This last "MULTI-SYSTEM" type of parallelization, proposed in this work, uses complete sub-system modules later called single *"Cellular-ECU"*[*]. While state-of-the-art automotive systems focus strongly on highly heterogeneous architectures with each CPU doing something very different from all others, the work herein tries to revive the possibility of having a much more homogeneous system architecture. Noting that most FDEFs take their inputs, process them unaware of all others and output their values, leads to the

[*] *Each "Cellular-ECU" module is composed by a complete set of processing core with smart peripherals directly attached*

thought that each FDEF could even run on its own processor in its own *"Cellular-ECU"* module (upper part of the bottom-right depiction in Fig. 4.5:76). This is true, aside from the fact that some functionality still needs some form of event-driven actions, such as for crankshaft engine-speed signal, injection, ignition, knocking and any other that needs asynchronous attention. Getting rid of these critical lower-level functionalities by displacing them to the *"Smart Peripherals"* (lower part of the bottom-right depiction in Fig. 4.5:76), frees all the higher-level ones for natural parallel processing. This matter will be closely followed and treated later on, because this will be one of the main reasons for giving birth to the *"Smart Peripherals"* within this work. Fig. 4.5:77 shows a comparison between this functional separation and the classical way of having everything inside the same "box".

As a side-effect of this separation, scalability will be much easier and straightforward, when compared to classically integrated/centralized systems, since this overall system is not much more than several virtually independent *"Cellular-ECUs"* inter-connected through a simple communications network. This system should therefore perform better than the degradation seen in [388] when adding more cores. Automotive FDEFs and their data accesses/needs are very well suitable for residing in these independent modules.

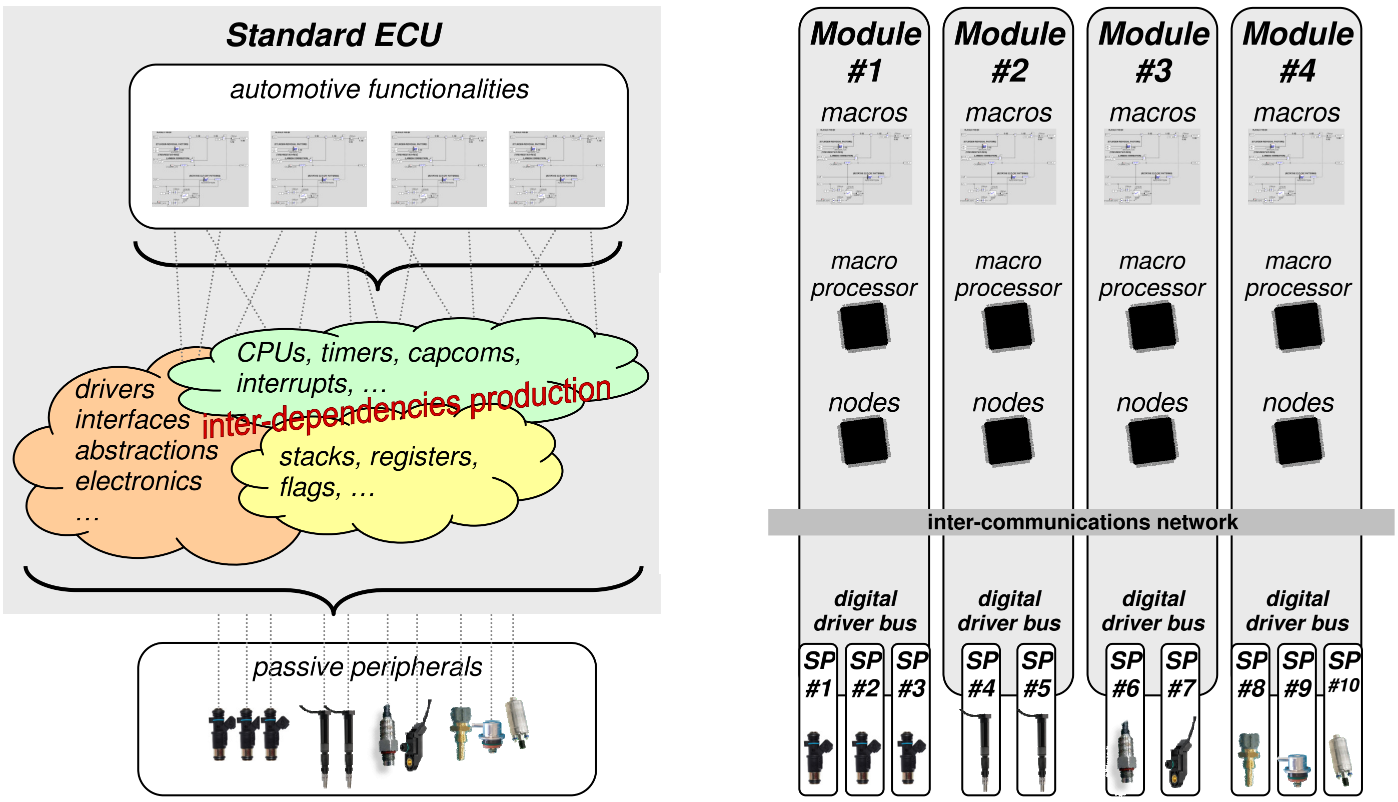


Fig. 4.5:77 – Left: classical system with "passive" peripherals; Right: "MULTI-SYSTEM" with "Smart Peripherals"

While automotive functionalities and peripherals are both intrinsically parallelizable from the data-flow processing code point-of-view, classical details such as a centralized ECU with its highly inter-dependent CPUs and software driver levels, stand in the middle of such parallelization. On the other hand, the "MULTI-SYSTEM" approach made herein makes parallelization easier because it eliminates this middle inter-dependencies, limited resources and confusion altogether. Each module only calculates the final values each peripheral needs and then communicates it to the respective *"Smart Peripherals"* independently. Each *"Smart Peripherals"* then locally includes and uses the necessary resources as needed and totally independent of each-other. Finally, each *"Smart-Peripheral"* can be seen as a completely independent processing entity by itself, similar to the *"Cellular-ECU"* concept already used for the main processing modules themselves. If one peripheral is damaged or malfunctioning, this will not fatally interfere will all the rest of the system.

The only interaction among modules will be done through the high-level *"Nodes"* exchange through the "inter-communications network". In one expression, this "MULTI-SYSTEM" architecture sorts out the hardware spot where all comes together in a big pile and separates the paths by separating the hardware itself. Furthermore, no central control or master-entity is needed. It will also later be seen that both main modules and peripherals go with the central *"Cellular-ECU"* concept, while the "inter-communications network" and the various "digital driver busses" compose the global communications network among all those processing components.

It should be noted that this system not only benefits from extreme hardware partitioning (the multiple modules themselves), but also from extreme software partitioning as well, both embodying the basic idea of smaller but many CPUs processing also correspondingly many but also smaller and therefore less bug-prone software chunks [379]. Important issues such as the necessary distribution and corresponding load-balancing are discussed later on, when FDEF distribution among the processing cores and peripherals is going to be detailed.

4.5.4 Final "FDEF-Processor"

[Emphasis: treating an FDEF as a complete/indivisible unit]

"Direct *Macros* Processing" and elimination of most pitfalls found in classical systems is the main idea behind this global construct called *"FDEF-Processor"*. Other systems claiming "Direct Processing" such as *PicoJAVA* [111] [112], *Jazelle DBX* [102], *Pascal MicroEngine* [89], *Micro-coded FORTRAN* [511]) solutions., etc. still employ pitfall-laden details such as stacks, caches, translators, etc. By mentioning "Direct" herein, it is really meant that no execution-influencing add-ons, units, sub-systems, flags, etc., are used whatsoever. Only the minimal processor structure taking care of fetching operations and feeding them to the execution core, is used. "SYMBOL" [136] [425] [426] is the only system that comes nearest to this truly pure "Direct Processing" concept, but still implements hardware details that the *"Macros"* will not need herein.

The final global concept of the *"FDEF-Processor"* results from the agglomeration of the following components depicted in Fig. 4.5:78. These components are all implemented around the previous plain *"Macros-Processor"*, thus encompassing everything that is needed to execute those *"Macros"* in conjunction with the *"Nodes"* while also processing modifiers and internal side-details on them. The exact operation of the *"LP-RAM"* and *"LP"*-block components is explained later on, when the *"Live-Prototyping"* comes into playing the main-role. Fig. 4.5:79 shows details about the main paths taken by the various fetched and stored FILT *"Macro"* components and result. The Harvard style architecture is clear from these illustrations.

- *"Macro-Modifiers"* awareness and handling appendix unit
- *"Node-Modifiers"* awareness and handling appendix unit
- *"Macros-FLASH"* for main *"Macros-Sequence"* storage
- *"LP-RAM"* for *"Live-Prototyping"* intermediate *"Macros"* storage
- *"LP"*-block for *"Live-Prototyping"* features handling
- *"PC"* as in Program-Counter to keep track of the current *"Macro"* being executed
- *"Priority-Counters"* for the *"Macro"* execution priotization scheme (see Fig. 4.5:73)
- *"RTC"* Real-time Clock (determines *"Cycle-Times"*) for time-filters, time-integrators and delays
- Temporary intermediate values for *"Macros"* needing state, such as filters, integrators & delays

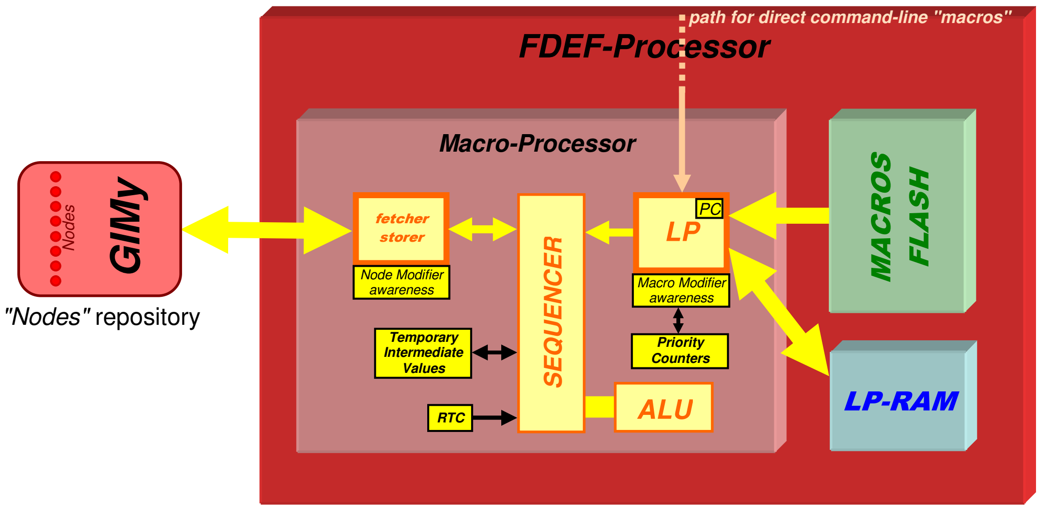


Fig. 4.5:78 – The complete "FDEF-Processor" with all its internal components

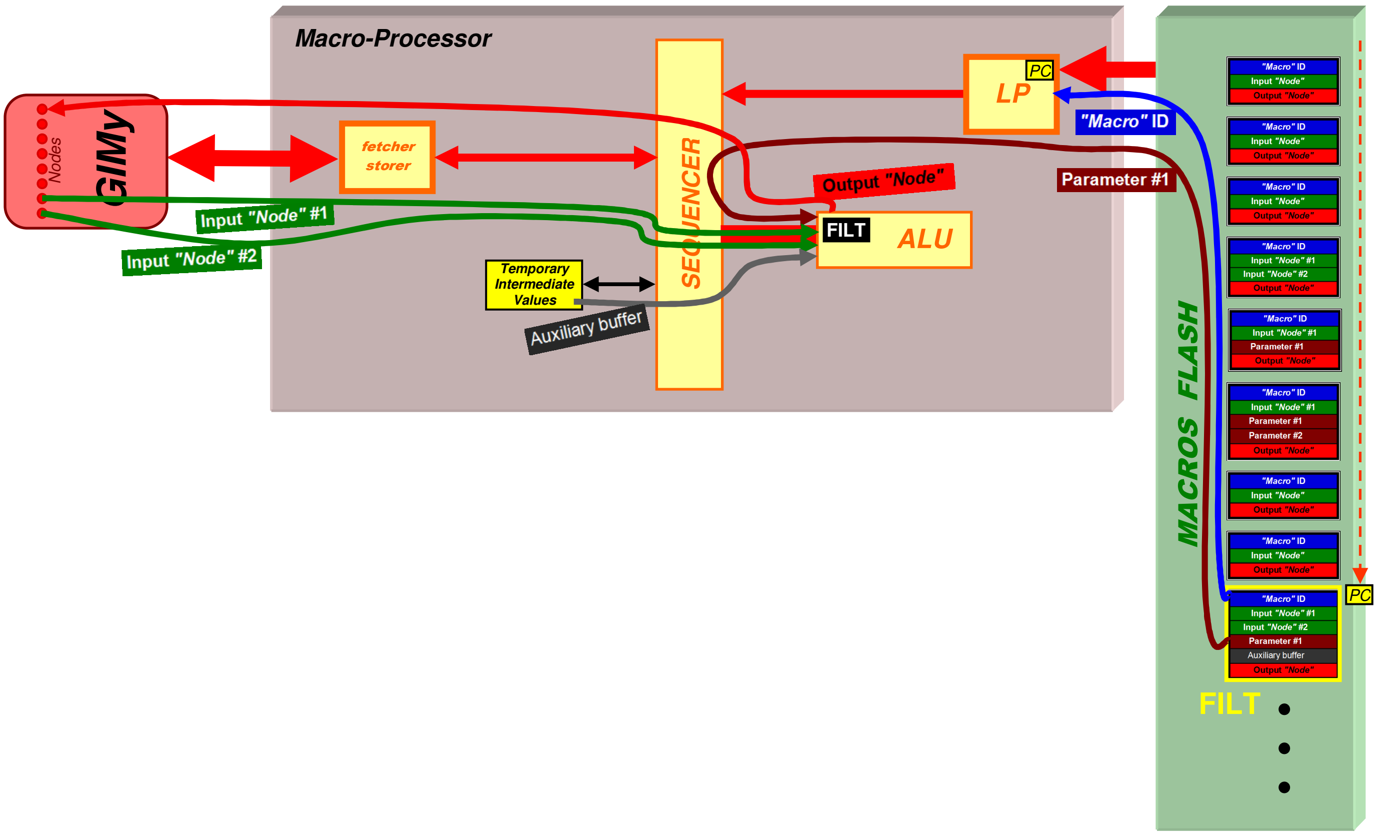


Fig. 4.5:79 – The main "Macro"-ID, in/out "Nodes", parameters, auxiliary buffer paths inside the "FDEF-Processor"

One thing cannot be stressed enough: since the *"Macros"* do not have or produce any state or context information (such as flags, registers, etc. in classical processors), all the state and context information is automatically outside this *"Macros-Processor"*, specifically inside the external *"GIMy*", therefore guaranteeing that once a certain *"Macro"* and its input *"Nodes"* arrive at the internal sequencer, it will be processed in exactly the same way as it always has been. A normal exception to this has to do with the way of operating of filters, delays, integrators and such. These always need temporary storage for their current internal state. For filters, it is a higher-resolution value than that possible with the *"Nodes"* for the practical reason that very small inputs may not change the output in a short period of time and thus need to be accumulated inside the filter until the output changed at all. Something similar is the case for delays, where an active input accumulates internal time until a threshold is passed and the output is also activated at that point. Integrator follow the same reason as for filters, and so on. Since these temporary states have nothing to do with the real *"Nodes"* that are used inside the corresponding FDEFs, for a matter of organization, these where thus stored inside the *"Macro-Processor"* itself.

Memory access will be internal and, as such, perfectly adapted to the needs of the *"Macros-Processor"*, without any complex timing or accessing mechanisms. This goes for all three memory types: *"GIMy", "Macros-FLASH"* and *"LP-RAM"*. Note how this *"GIMy"*, which holds all the *"Nodes"*, is placed outside the main *"Macros-Processor"* core hardware. Inside that core, the SEQUENCER itself is the main component that orchestrates all the execution process. Note how the Program-Counter (PC) is also placed outside this critical component. Both these notes bring back the already mentioned "micro-rebooting" mechanism referred to in [328] [329], in the way how the only context/state present in this system is the PC and the *"GIMy"*, which could be directly compared to the strictly necessary "state-store" concept mentioned in [328] [329]. Basically, the really needed persistent data storage is thus kept outside of the execution core, therefore allowing this core to micro-rebooting itself without any effect on the whole system other than guaranteeing maximum "anti-aging" reliability (as compared to deteriorative "software aging" [328]). This has the corollary of having only 100% logic-related *"Macros"* execution elements inside the *"Macros-Processor"*, instead of additionally having context/state-related disruptive mechanisms.

Most *"Macros"* are executed directly in one single step inside the *"Macros-Processor"*, exactly due to being such self-contained and self-sufficient entities. There is no need to expand them or otherwise sub-divide their execution internally in any way. The necessary *"Node"* fetching and storing is intrinsically done by the hardware mechanisms around the sequencer itself. On the other hand, higher-level *"Macros"* such as counting, filtering, integrating and delaying, are executed through sub-sequencing their constituent sub-operations. But instead of this expansion happening at compilation-level, as in any current development tool, it is still 100% done inside the *"Macros-Processor"* itself, through the same SEQUENCER mechanism. This way, the *"Macros"* still maintain their 100% self-confinement in the sense of the visual programming language, the development system and even the hardware processing system themselves. Additionally to all this, even in the hypothetical case of the *"Macros"* being processed by some classical hardware, they would be processed always in the exact same non-misinterpretable way, consuming and producing also non-mininterpretable *"Nodes"*. Thus, the *"Macros"* language itself is theoretically 100% independent of any hardware implementation. That hardware implementation only has to understand *"Macros"*, no low-level distinctions made whatsoever. Fig. 4.5:80 shows an example of one such *"Macro"* being executed step-by-step inside the *"Macros-Processor"*.

"COMLR Macro"

$f(x) = [x \geq Left] \wedge [x \leq Right]$ *Eq. 4.5:1 – Original continuous comparator function*

$y_{n+1} = [x_n \geq Left] \wedge [x_n \leq Right]$ *Eq. 4.5:2 – Original comparator function used in classical ECUs & herein*

→ fetch *"COMLR Macro"* → fetch *"Value Node"* → fetch *"LeftLimit Node"* → fetch *"RightLimit Node"*
→ feed ALU with *"Value >= LeftLimit"* → trigger ALU execution & store intermediate result
→ feed ALU with *"Value <= RightLimit"* → trigger ALU execution & store intermediate result
→ feed ALU with "IntRes1 && IntRes2" → trigger ALU execution & store final result
→ get ALU final result → store result into *"Result Node"*

COMLR	*Value, LeftLimit, RightLimit, Result*	*Result = (Value >= LeftLimit) && (Value <= RightLimit)*

Fig. 4.5:80 – Execution sub-sequencing of the COMLR comparison "Macro"

The *"RTC"* is needed for operations that obviously need to run in real-time. Eq. 4.5:82 and Eq. 4.5:83 represent examples of real-time operations running inside current ECUs. Since there are no rasters whatsoever in the system developed herein, processing these special operations needs a simple hack to work even in such an apparently "realtime-less" environment. This "hack" is exactly the same as the one employed in the previously mentioned MS2.8 and MS2.9 ECUs [302] and is based on a very simple replacement of the *Δt* raster-cycle parcel in current filtering, integrating and delaying formulas such as Eq. 4.5:82 and Eq. 4.5:83, by the *"Cycle-Time"* entity. This *"Cycle-Time"* represents the time needed for the complete *"Macros-Sequence"* to be processed once. Eq. 4.5:87 and Fig. 4.5:88 show the sub-sequencing of such *"Macros"* with help of this real-time hack to deliver the needed real-time reference. Temporary variables inside the *"FDEF-Processor"* are used for *"Macros"* using non-volatile internal "state", such as the next ones below. These temporary variables are shown in Fig. 4.5:79. "Timing uncertainties" related to the use of this *"Cycle-Time"* in delays and filters will be discussed later on herein.

"DLYU Macro"

$$\begin{cases} \left| \begin{matrix} f(x) = 0 \\ S = 0 \end{matrix} \right| \Leftarrow x \leq 0 \\ \left| \begin{matrix} S = S + dt \\ f(x) = 1 \Leftarrow S \geq T \end{matrix} \right| \Leftarrow x > 0 \end{cases}$$

Eq. 4.5:81 – Original continuous 1st-order filtering function (S is state)

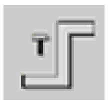
$$\begin{cases} \left| \begin{matrix} f(x)_{n+1} = 0 \\ S_{n+1} = 0 \end{matrix} \right| \Leftarrow x \leq 0 \\ \left| \begin{matrix} S_{n+1} = S_n + \Delta t \\ f(x)_{n+1} = 1 \Leftarrow S_n \geq T \end{matrix} \right| \Leftarrow x > 0 \end{cases}$$

Eq. 4.5:82 – Time-raster based discrete delay function used in classical ECUs

$$\begin{cases} \left| \begin{matrix} y_{n+1} = 0 \\ S_{n+1} = 0 \end{matrix} \right| \Leftarrow x \leq 0 \\ \left| \begin{matrix} S_{n+1} = S_n + CycleTime \\ y_{n+1} = 1 \Leftarrow S_n \geq T \end{matrix} \right| \Leftarrow x > 0 \end{cases}$$

Eq. 4.5:83 – Variable "Cycle-Time" based discrete delaying used herein

→ fetch *"DLYU Macro"* → fetch *"Value Node"* → fetch *"Delay Node"* → fetch *"State temp-var addr"*
→ feed ALU with "Value == FALSE(0)" → trigger ALU execution → get ALU result
→ if result TRUE, then → store FALSE(0) into *"Result Node"* → store 0 into *"State temp-var"*
→ if result FALSE, then → feed ALU with "State > Delay" → trigger ALU execution → get ALU result
→ if result TRUE, then → store TRUE(1) into *"Result Node"*
→ if FALSE, then → store (State + CycleTime) into *"State temp-var"*



DLYU	*Value, Delay, (State), Result*	*if (Value == FALSE) then {Result = FALSE; State = 0} else if (State >= Delay) then Result = TRUE else State = State + CycleTime*

Fig. 4.5:84 – Execution sub-sequencing of the DLYU delaying "Macro"

As already said before, delaying *"Macros"* suffer from a tolerable "timing uncertainty" related to the variable *"Cycle-Time"* present in this processing system.

"FILT Macro"

$$f(x) = f(x) + (x - y) \cdot \frac{\tau}{dt}$$

Eq. 4.5:85 – Original continuous 1st-order filtering function

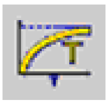
$$y_{n+1} = y_n + (x_n - y_n) \cdot \frac{T}{\Delta t}$$

Eq. 4.5:86 – Time-raster based discrete filtering used in classical ECUs

$$y_{n+1} = y_n + [x_n - y_n] \cdot \frac{T}{CycleTime_n}$$

Eq. 4.5:87 – Variable "Cycle-Time" based discrete filtering used herein

→ fetch *"FILT Macro"* → fetch *"Value Node"* → fetch *"TimeConst Node"* → fetch *"State temp-var addr"*
→ feed ALU with "TimeConstant / CycleTime" → trigger ALU execution & store intermediate result
→ feed ALU with "Value - State" → trigger ALU execution & store intermediate result
→ feed ALU with "IntRes1 * IntRes2" → trigger ALU execution & store intermediate result
→ feed ALU with "State + IntRes" → trigger ALU execution
→ get ALU result → store result into *"Result Node"*



FILT	*Value, TimeConstant, Enable, (State), Result*	*if (Enable == TRUE) then Result = State = State + ((Value-State)*(TimeConstant / CycleTime))*

Fig. 4.5:88 – Execution sub-sequencing of the FILT temporal filtering "Macro"

As already said before, filtering, integrating and other analog *"Macros"* suffer from the fact that too small *"Cycle-Times"* may disrupt the division calculation in that it gets numerically too brittle (high spikes). As also said before, this is easily solved through a special *"Cycle"* counter that activates filter, integration and other related *"Macros"* only after a minimum amount of accumulated *"Cycle-Time"* has passed, simply retaining the last filter output value in-between. A *"Macro"* that is very similar to this "FILT" one is the time-integrator "INTT". Only the formula is slightly different.

"IF Macro"

→ fetch *"IF Macro"* → fetch *"Condition Node"*
→ feed ALU with "Condition == TRUE(1)" → trigger ALU execution → get ALU result
  → if result TRUE, then → continue executing next *"Macros"*
  → if result FALSE, then → jump to "FALSE" block and continue executing *"Macros"* from there

IF	*Condition*	*if (Condition == TRUE) then execute green "Macros" block else, if (Condition == FALSE) execute red "Macros" block*

Since there is only an "IF" *"Macro"* existing, and as done in sequential Assembly language to which virtually all high-level programming languages ultimately translate to, this "IF" will be regarded by the *"FDEF-Processor"* as the *"hot-Macro"* do determine the beginning of the "IF-THEN-ELSE" block. The rest will be done with a simple additional "JMP" jump *"Macro"*, to get the correct execution sequence result, as illustrated in Fig. 4.5:89.

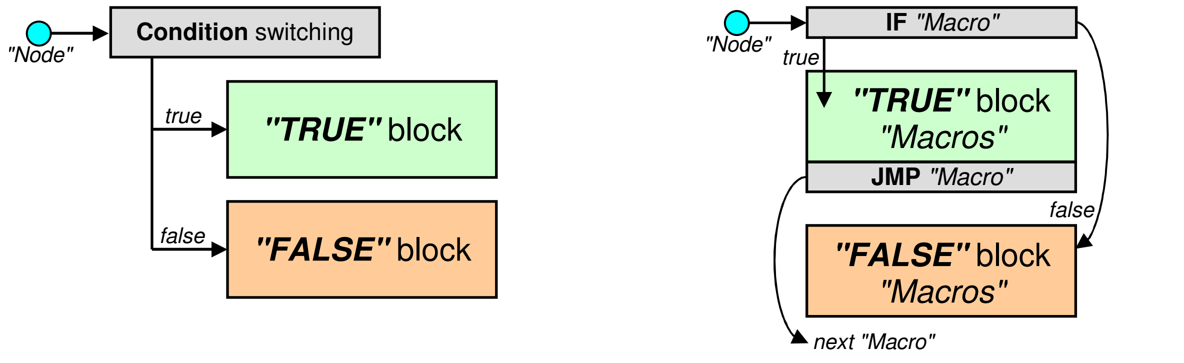
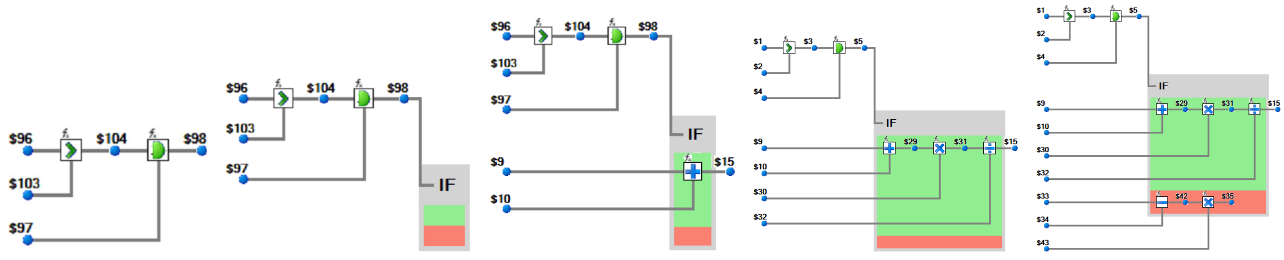
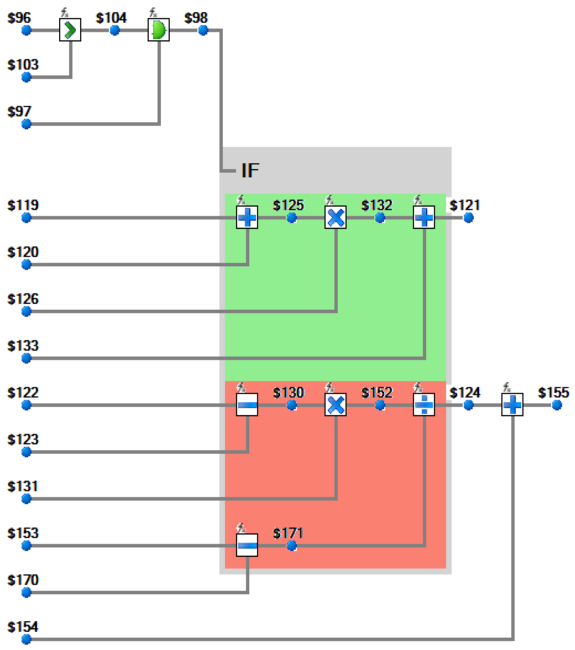
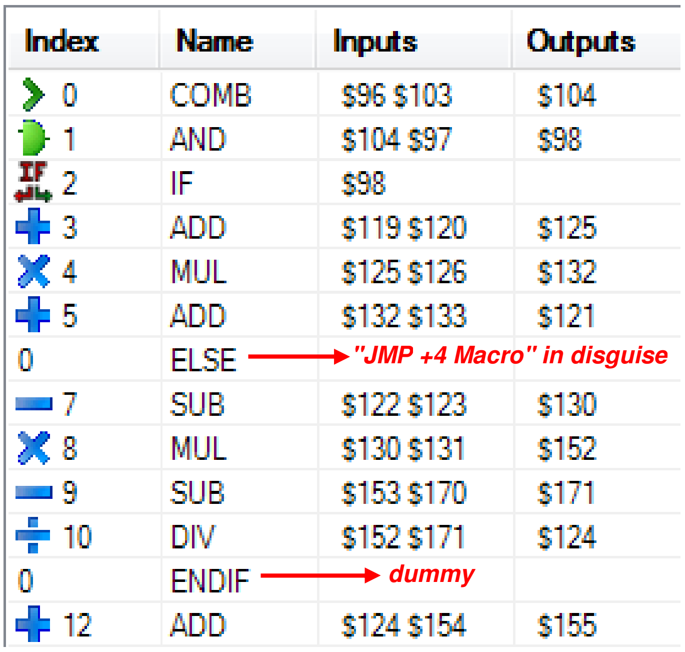


Fig. 4.5:89 – Left: abstract conditional diagram; Right: execution sub-sequencing inside the "FDEF-Processor"

To have an idea of how this *"Macro"* behaves visually, Fig. 4.5:90 illustrates an example of its usage. As already mentioned earlier when listing the available *"Macros"*, since this *"IF Macro"* is an isolated operation that directly receives a Boolean input, no evaluation of complex conditional statements or external central flags has to be done inside the *"Macro"* itself and "random processing" is still also possible for these conditional *"Macros"* as well.







Index	Name	Inputs	Outputs
0	COMB	$96 $103	$104
1	AND	$104 $97	$98
2	IF	$98	
3	ADD	$119 $120	$125
4	MUL	$125 $126	$132
5	ADD	$132 $133	$121
0	ELSE	→ ***"JMP +4 Macro" in disguise***	
7	SUB	$122 $123	$130
8	MUL	$130 $131	$152
9	SUB	$153 $170	$171
10	DIV	$152 $171	$124
0	ENDIF	→ ***dummy***	
12	ADD	$124 $154	$155

Fig. 4.5:90 - Example use of the conditional "IF" on the "iEditor" and corresponding "Macros-Sequence"

As already explained earlier (4.5.1.2 – Flow-Control & Signal Switching and 4. – IFless Jitterless Timing), this "IF" *"Macro"* is going to be avoided altogether, so that in the engine functionalities herein implemented at the end of this work, are not contemplating any "IF" in their FDEFs. Whenever a conditional value is to be used in some path, then switching/multiplexing *"Macros"* will be used instead of "IFs", as already said to be perfectly possible in a data-flow program like this, earlier on herein ("IF"-less jitter-less timing). Nevertheless, as a proof-of-concept of its feasibility inside the implemented graphics-engine, *"IF Macros"* where also working at the end. More *"Macros"* (such as PAR1D and PAR2D), more details, comparisons and illustrations in [Att. 40].

Other extra external structures needed for this *"FDEF-Processor"* to effectively work correctly, are described later and refer to indispensable communications and central *"Nodes"* memory mechanisms. The *"Nodes"* memory, later called *"GIMy"* is already depicted here, as an indispensable part of the *"FDEF-Processor"*. This processor structure will be able to process the *"Macros-Sequence"* with all their related nuances and handling features. Nevertheless, this *"FDEF-Processor"* does not need any options or switches to be set up by the developer. Since this structure has been devised directly and solely for the *"Macros"* and since they are directly processed in it without any translation whatsoever, options and switches are completely inexistent.

As for the testing and verification tasks, this hardware has the advantage relative to existing systems in that the *"Macros"* are executed inside a "micro-rebooted" processor, so that the execution has to be verified only once for each *"Macro"*. Besides that, all external mechanisms such as debugging, prototyping and logging, have to be separately tested.

FINAL NOTE: All classical systems rely on more or less translation and management mechanisms inside them. The more translators and managers exist, the more the system gets globally incomprehensible and complex. Moreover, the high flexibility of those translator layers just produce overwhelming amounts of options and switches.

From the statements above, ***PITFALL C#3***$^{\text{(Word-Alignment)}}$, ***PITFALL C#16***$^{\text{(Memory Wait-States)}}$, ***PITFALL C#15***$^{\text{(Options \& Switches)}}$ and ***PITFALL C#29***$^{\text{(Temporary Storage)}}$ are thus effectively eliminated!

Page intentionally left blank

Page intentionally left blank

"Every truth passes through three stages. First, it is ridiculed.
Second, it is violently opposed. Third, it is accepted as being self-evident"
– Arthur Schopenhauer

CHAPTER 5

Globally Live and Cellular *"Macros"* System

5.1 Advanced Architecture and Operation

[Emphasis: the whole thing as one single integrated organism]

Now that the two main ideas, the *"Macros-Sequence"* and the *"FDEF-Processor"*, have been extensively exposed, it is time to get down to the remaining components that make up the software and hardware mechanisms wrapping around those two previous ideas. These mechanisms include everything that is necessary to handle such a system, such as interconnecting sub-components, communications, visual user-interface, etc. These will be presented next, in a sequence of *"LIVE"*-operating mechanisms.

The basic hardware architecture that allows for these features is depicted in Fig. 5.1:1. Here only the main blocks are shown, ranging from the RAM that holds the values (*GIMy – Global Intelligent Memory*), the *"FDEF-Processor"* and the remaining blocks necessary for peripherals and various kinds of communications. This *"FDEF-Processor"* in fact encompasses *"GIMy"* to form a single unit, but the *"GIMy"* is kept separated to better show some of the subsequent *"LIVE"* features.

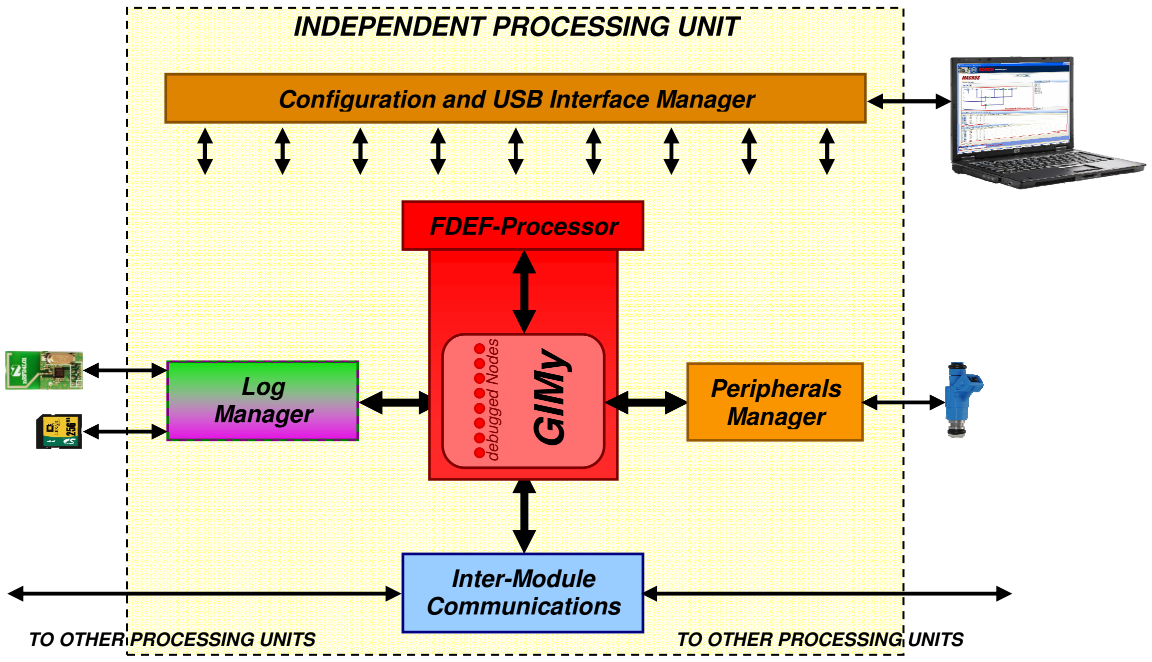
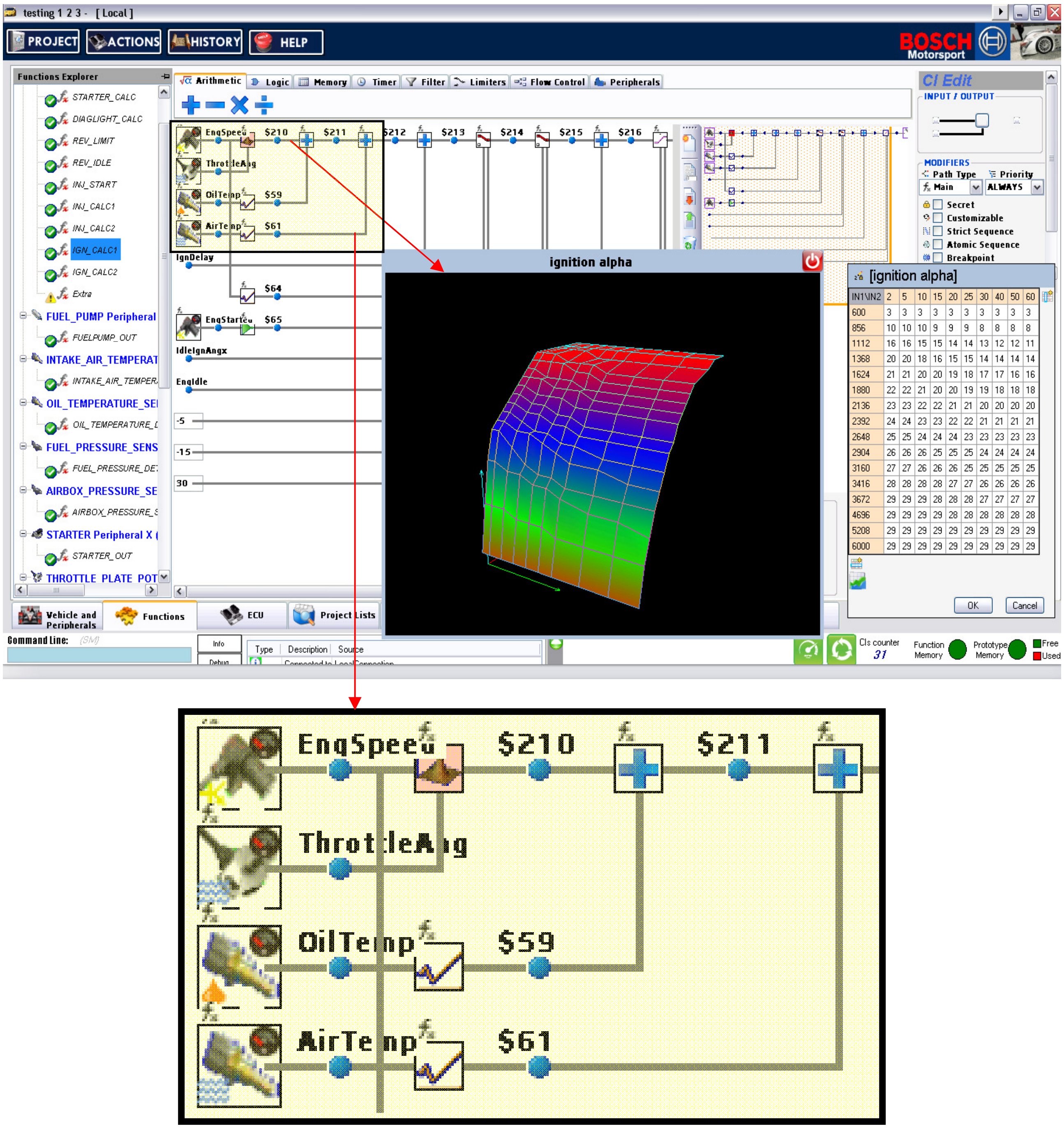

INDEPENDENT PROCESSING UNIT
Configuration and USB Interface Manager
FDEF-Processor
debugged Nodes
GIMy
Log Manager
Peripherals Manager
Inter-Module Communications
TO OTHER PROCESSING UNITS
TO OTHER PROCESSING UNITS

Fig. 5.1:1 – Simplified hardware architecture with the various internal data transfer paths

This hardware architecture will later be shown in real FPGA hardware, while the corresponding software features will now be explained. Fig. 5.1:2 shows a sample screenshot of the *"iEditor"* entirely done within Microsoft Visual Studio 2005 [292] in the C# [71] high-level .NET-managed language.



IN1\IN2	2	5	10	15	20	25	30	40	50	60
600	3	3	3	3	3	3	3	3	3	3
856	10	10	10	9	9	9	8	8	8	8
1112	16	16	15	15	14	14	13	12	12	11
1368	20	20	18	16	15	15	14	14	14	14
1624	21	21	20	20	19	18	17	17	16	16
1880	22	22	21	20	20	19	19	18	18	18
2136	23	23	22	22	21	21	20	20	20	20
2392	24	24	23	23	22	22	21	21	21	21
2648	25	25	24	24	24	23	23	23	23	23
2904	26	26	26	25	25	25	24	24	24	24
3160	27	27	26	26	26	25	25	25	25	25
3416	28	28	28	28	27	27	26	26	26	26
3672	29	29	29	28	28	28	27	27	27	27
4696	29	29	29	29	28	28	28	28	28	28
5208	29	29	29	29	29	29	29	29	29	29
6000	29	29	29	29	29	29	29	29	29	29

Fig. 5.1:2 – Screenshot of the "iEditor", the Intelligent Editor with integrated "LIVE"-features

Everything in this system will be centred around visual interaction with the users and developers alike. Nevertheless, even though the user-interface will be a high-lvel visual one, much more direct manipulation of system details is given, contrary to state-of-the-art systems This developed architecture strongly impersonates the "visual-to-hardware" concept of this work herein. Every detail concerning debugging, simulation, stepping, etc. will also be possible to be issued directly on the FDEF and without any other tool or turnaround, effectively addressing ***PITFALL H#49*** as well.

"Easy-Handling" features have been the holy grail for most tool developer since the advent of the visual interface. Making a user's and developer's work easier and even more enjoyable has also been a marketing and selling strategy ever since as well. Additional information about specific automotive development tasks can be seen in [Att. 41].

"LIVE"-operating mechanisms represent some very high-level results, which encompass radical enhancements to current systems. They are called *"LIVE"* features due to the fully visual, feedback-rich and minimal-turnaround characteristics they present. A full list of these *"LIVE"* features can be seen in Tab. 5.1:3, reflecting all major automotive project tasks where (*) represent fully implemented features and (**) represent partially implemented features that reached the level of testing. The right-most column mentions the corresponding *"iEditor"* views where they are mainly active. All these *"LIVE"* features are sub-features of a global concept of FDEF *"LIVE-HANDLING"* where the developer really can interact, program, inspect and handle the entire system without ever having to stop or pause it in any way, as it is the case with current tools.

FDEFs' "LIVE-HANDLING"	***"LIVE-DESIGNING"***	*"LIVE-Dragging"* *	on-the-fly visual programming	*"Functions"*
		"LIVE-Commandline" *	on-the-fly textual programming	*"iEditor"*
		"LIVE-Comparing" *	on-the-fly FDEFs comparisons	*"Functions"*
		"LIVE-Units" **	on-the-fly automatic unit propagation	*"Functions"*
		"LIVE-Priorities"	on-the-fly automatic priority propagation	*"Functions"*
		"LIVE-Previewing" *	on-the-fly FDEF grasp on preview pane	*"Functions"*
		"LIVE-Pan/Zooming" **	on-the-fly FDEF seamless pan/zooming	*"Functions"*
		"LIVE-Vehicle" *	on-the-fly peripherals handling on vehicle images	*"Vehicle"*
		"LIVE-Dashboard" *	integrated dashboard design for motorsports	*"Display"*
	"LIVE-DEBUGGING"	*"LIVE-Monitoring"* *	on-the-fly *"Node"* value inspection	*"Functions"*
		"LIVE-Generating" *	on-the-fly *"Node"* value fixation	*"Functions"*
		"LIVE-Breakpointing"	on-the-fly program interruption	*"Functions"*
		"LIVE-Stepping" *	on-the-fly step-wise execution	*"Functions"*
		"LIVE-Profiling"	on-the-fly timing inspection	*"Functions"*
	"LIVE-PROTOTYPING"	*"LIVE-Programming"* *	on-the-fly program changing	*"Functions"*
		"LIVE-Calibrating" *	on-the-fly parameter setting	*"Functions"*
		"LIVE-Testing" *	on-the-fly test-coding	*"Functions"*
		"LIVE-Simulation" *	on-the-fly system simulation	*"Functions"*
	"LIVE-OPERATION"	*"LIVE-Processing"* *	interacting cellular processing modules	*"Cellular-ECU"*
		"LIVE-Peripherals" *	smart/alive autonomous peripherals	*"Cellular-ECU"*
		"LIVE-Logging" *	on-the-fly integrated cellular data-logging	*"Datalog"* *"Functions"* *"Project Lists"*
		"LIVE-Teleoperation" *	on-the-fly remote handling	*"Extras"*
		"LIVE-Support" **	on-the-fly remote assistance	*"Extras"*

Tab. 5.1:3 – Full list of the herein theoretized "LIVE"-Operations

These *"LIVE"* features will next be further in more detailed fashion, especially those most advanced and special ones concerning prototyping and debugging. *"Live-Prototyping"* and *"Live-Debugging"* but especially the first one, are exactly the features that imprint most difference onto this work developed herein, when compared to state-of-the-art classical commercial automotive solutions or the complete lack thereof.

5.1.1 *“LIVE-Designing”* made easy

[Emphasis: design a system with immediate feedback]

The most effective way of editing the FDEFs is, of course, doing it visually. Therefore, the software that manipulates the *"Macros-Sequences"* must be able to take visual developer actions and incorporate them into the *"Macros-Sequence"* as editing goes on. Adding, moving or deleting *“Macros”* are examples of such actions. When the user needs to add, move or even delete a “Macro”, while most editors show the graphical end-result just after the action has been taken, the new interface is able to show a temporary view of the end-result without having taken the corresponding internal commitment yet. This feature allows the developer to view what would happen after the action, but before that action has been effectively taken, saving time with undo operations. More details about all features related to this *"Live-Designing"* and state-of-the-art examples can be seen in [Att. 41].

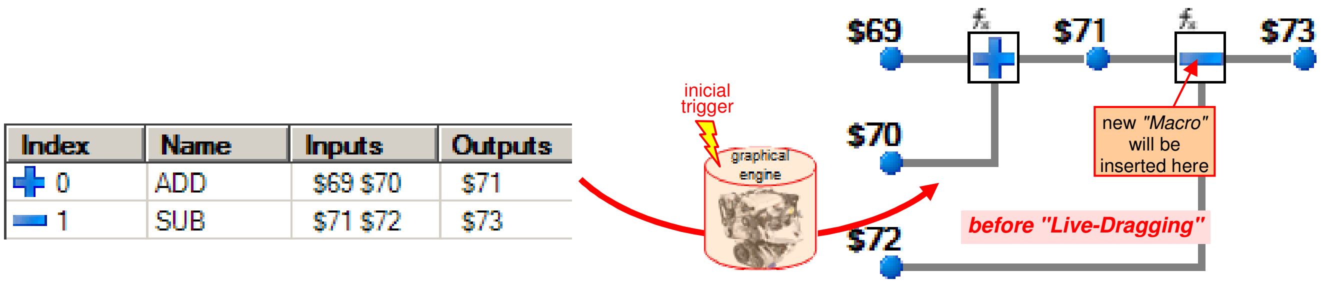
This paradigm encompasses exactly three manipulation possibilities, as detailed in the next few points and listed as follows:

- Graphically adding/changing *"Macros" ("Live-Dragging")*
- Textually adding/changing *"Macros"* (*"Live-Commandline"*)
- Finding differences between two FDEFs *("Live-Comparing")*

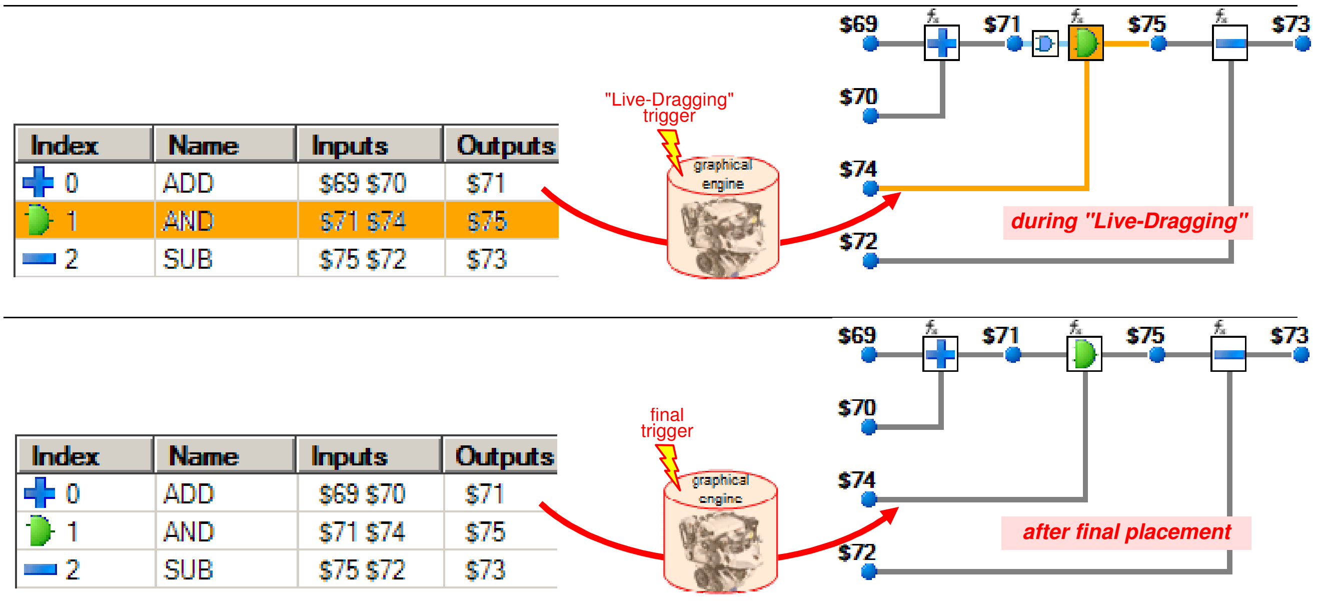
5.1.1.1 *“LIVE-Dragging”* (on-the-fly editing)

[Emphasis: really alive "WYSIWYG" modus-operandi]

By using the mouse and dragging operations from the toolbars, the developer may instantly get feedback upon his editing actions. This advanced feature is implemented by temporarily taking the intended placement action in a copy of the original *"Macros-Sequence"* and by displaying this temporary sequence graphically, colouring the difference in orange, as illustrated in Fig. 5.1:4. The graphical engine does nothing else other than continuously being triggered by the *"iEditor"* to re-draw the FDEF according to the current *"Macros-Sequence"* with all its changes. As soon as the *“Macro”* is effectively dropped into the FDEF, the resulting temporary *"Macros-Sequence"* will be taken as the definitive one. Fig. 5.1:5, Fig. 5.1:6 and Fig. 5.1:7 show this mechanism at work. Fig. 5.1:8 shows the distinction made by the *"iEditor"* when the developer drags a new *"Macro"* over a signal path or a *"Node"*. In case 1 the new *"Macro"* is placed in the middle of the signal path going from *"Node"* {$51} to the subtraction, while in case 2 it is placed in such a way that *"Node"* {$51} will be just used as an input to it. Fig. 5.1:9 shows an example of the same mechanism but now for individual *"Nodes"*, where the developer may move an existing node to another thereby connecting them together. Again, the developer can see a continuous intermediate result during that action, allowing him to better decide where to drop the displaced *"Node"*.



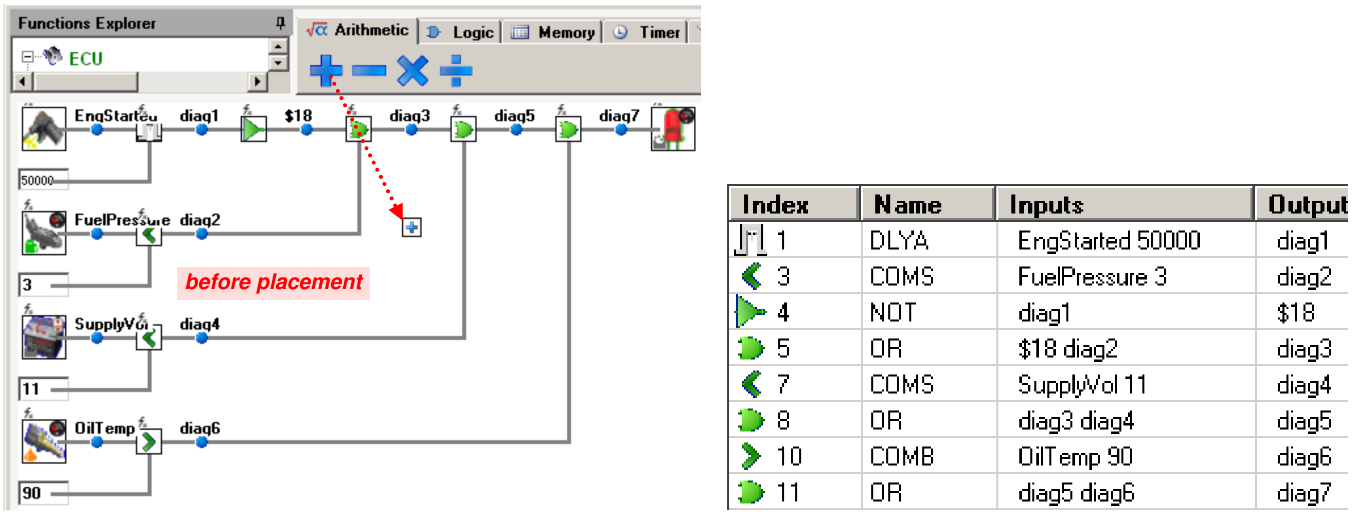
Index	Name	Inputs	Outputs
0	ADD	$69 $70	$71
1	SUB	$71 $72	$73



Index	Name	Inputs	Outputs
0	ADD	$69 $70	$71
1	AND	$71 $74	$75
2	SUB	$75 $72	$73

Index	Name	Inputs	Outputs
0	ADD	$69 $70	$71
1	AND	$71 $74	$75
2	SUB	$75 $72	$73

Fig. 5.1:4 – Base mechanism for the “Live-Dragging” feature illustrated herein



Index	Name	Inputs	Output
1	DLYA	EngStarted 50000	diag1
3	COMS	FuelPressure 3	diag2
4	NOT	diag1	$18
5	OR	$18 diag2	diag3
7	COMS	SupplyVol 11	diag4
8	OR	diag3 diag4	diag5
10	COMB	OilTemp 90	diag6
11	OR	diag5 diag6	diag7

Fig. 5.1:5 – ADD “Macro” being “Live-Dragged”. Original FDEF and original "Macros-Sequence"

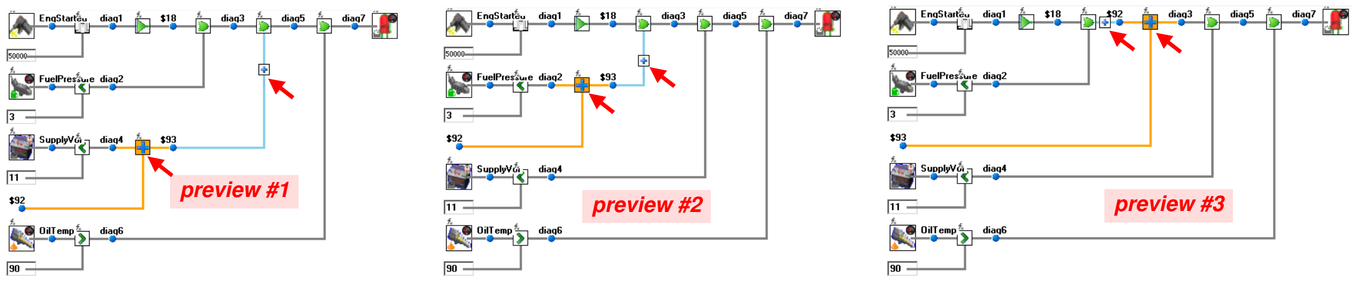
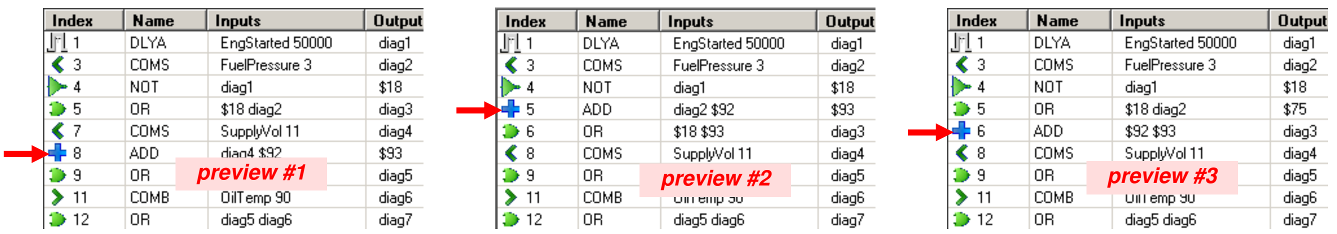


Fig. 5.1:6 – ADD “Macro” being “Live-Dragged” into an FDEF at different placement spots


preview #1
preview #2
preview #3

Index	Name	Inputs	Output
1	DLYA	EngStarted 50000	diag1
3	COMS	FuelPressure 3	diag2
4	NOT	diag1	$18
5	OR	$18 diag2	diag3
7	COMS	SupplyVol 11	diag4
8	ADD	diag4 $92	$93
9	OR	[illegible]	diag5
11	COMB	OilTemp 90	diag6
12	OR	diag5 diag6	diag7

Index	Name	Inputs	Output
1	DLYA	EngStarted 50000	diag1
3	COMS	FuelPressure 3	diag2
4	NOT	diag1	$18
5	ADD	diag2 $92	$93
6	OR	$18 $93	diag3
8	COMS	SupplyVol 11	diag4
9	OR	[illegible]	diag5
11	COMB	OilTemp 90	diag6
12	OR	diag5 diag6	diag7

Index	Name	Inputs	Output
1	DLYA	EngStarted 50000	diag1
3	COMS	FuelPressure 3	diag2
4	NOT	diag1	$18
5	OR	$18 diag2	$75
6	ADD	$92 $93	diag3
8	COMS	SupplyVol 11	diag4
9	OR	[illegible]	diag5
11	COMB	OilTemp 90	diag6
12	OR	diag5 diag6	diag7

Fig. 5.1:7 – ADD “Macro” being temporarily placed into the "Macros-Sequence"

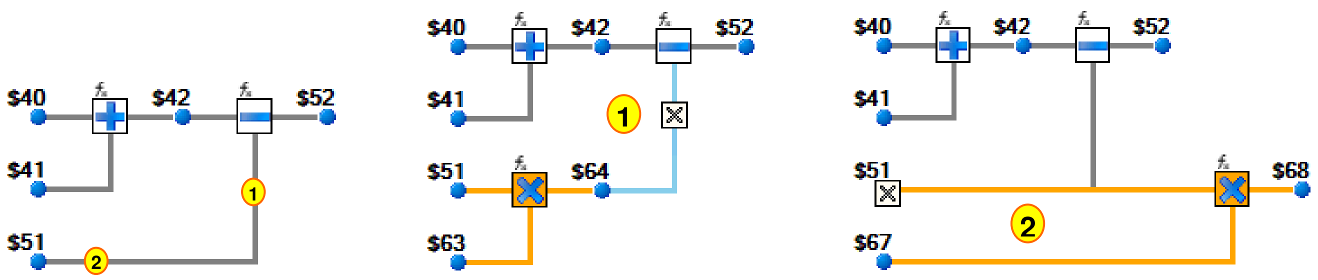
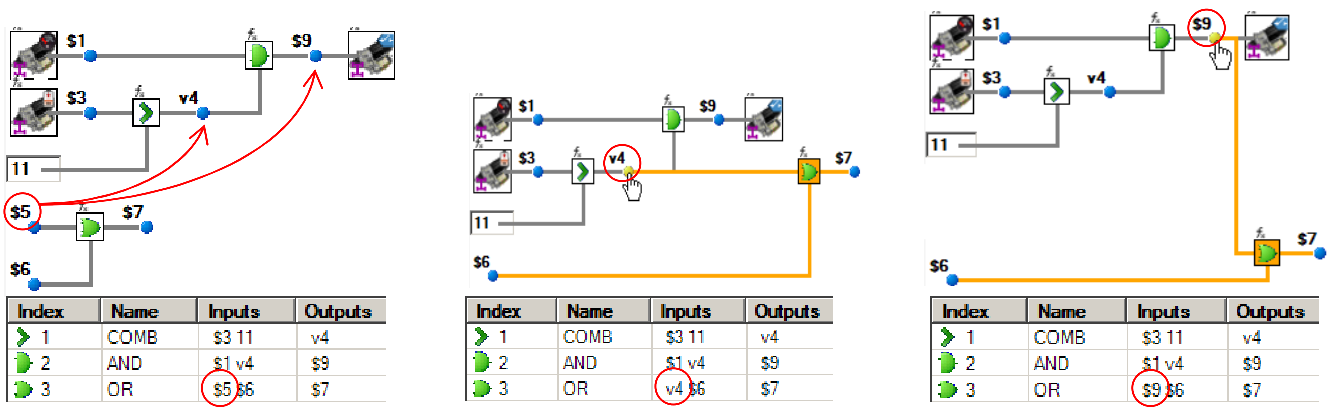


Fig. 5.1:8 – Distinction made by the "iEditor" when dragging over a path or a "Node"



Index	Name	Inputs	Outputs
1	COMB	$3 11	v4
2	AND	$1 v4	$9
3	OR	$5 $6	$7

Index	Name	Inputs	Outputs
1	COMB	$3 11	v4
2	AND	$1 v4	$9
3	OR	v4 $6	$7

Index	Name	Inputs	Outputs
1	COMB	$3 11	v4
2	AND	$1 v4	$9
3	OR	$9 $6	$7

Fig. 5.1:9 – OR "Macro" with its {$5} "Node" being connected to an existing one

This fully interactive mechanism is the useful and necessary complement to the fact that the *"Macros-Sequence"* is in fact the central-role playing entity within the system developed herein, and that the visual representation is nothing more than the directly (user-level) reflection of the developer's visual programming actions, to precisely see the implementation outcome and verify/validate it immediately. This had already been addressed in Fig. 4.5:75.

FINAL NOTE: Most classical systems with visual language interfaces rely on some sort of manual intervention for placing the graphical elements, throwing at the developer the full responsibility of designing clear and readable programs.

From the statements above, ***PITFALL #48*** (2/2 – Manual Layout) is thus effectively eliminated through a fully automated graphical engine that always keeps everything coherent upon user editing actions.

5.1.1.2 *"LIVE-Commandline"* (on-the-fly typing and executing)

[Emphasis: speedy input for experts only]

The whole graphical interface of the *"iEditor"* sends commands to an internal command-line mechanism upon any user actions. Fig. 5.1:10 shows the internal structure regarding command flow from the graphical interface to the underlying core functions in the operational engine, which execute the commands. Note that all commands and actions coming from the graphical interfaces receiving mouse actions, are fed into the operational engine indirectly, by being relayed through the command-line itself. This allows the *"iEditor"* to also be operated directly through the command-line, bypassing the graphical interface. Every single editor action may be inserted through the command-line, such as *"Macro"* manipulation commands as illustrated in Fig. 5.1:11 where two new *"Macros"* were added to an existing FDEF.

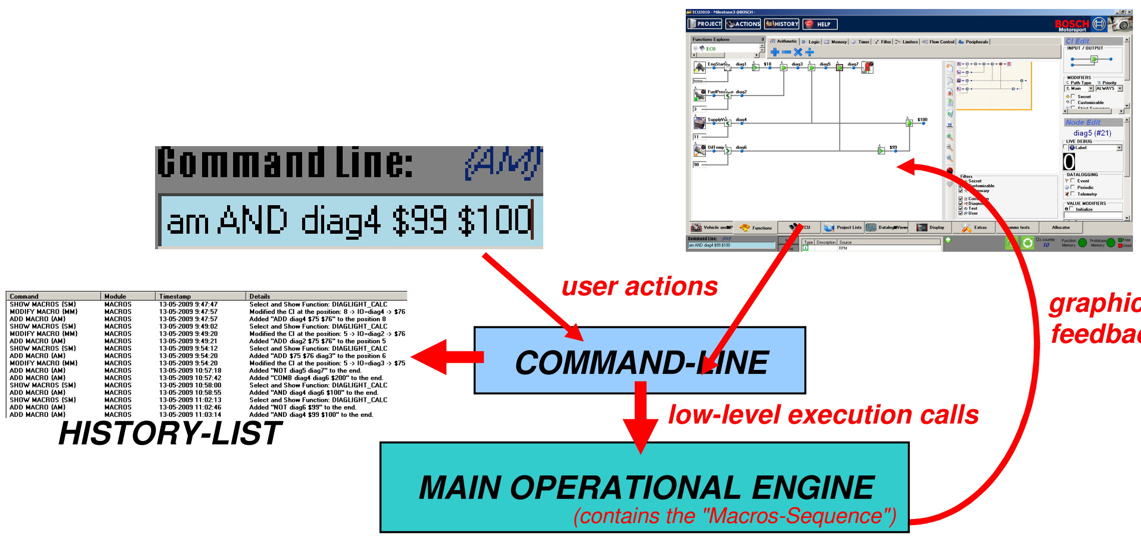


Fig. 5.1:10 – Both command-line and graphical interfaces feed the underlying operational engine

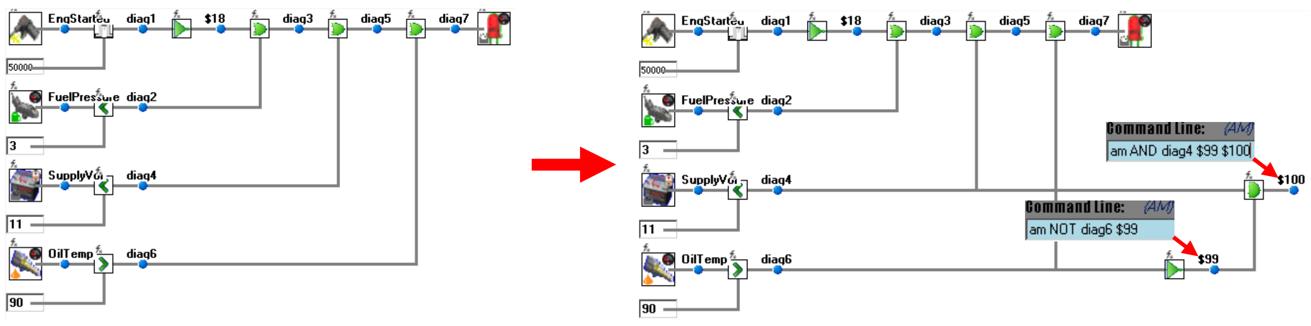
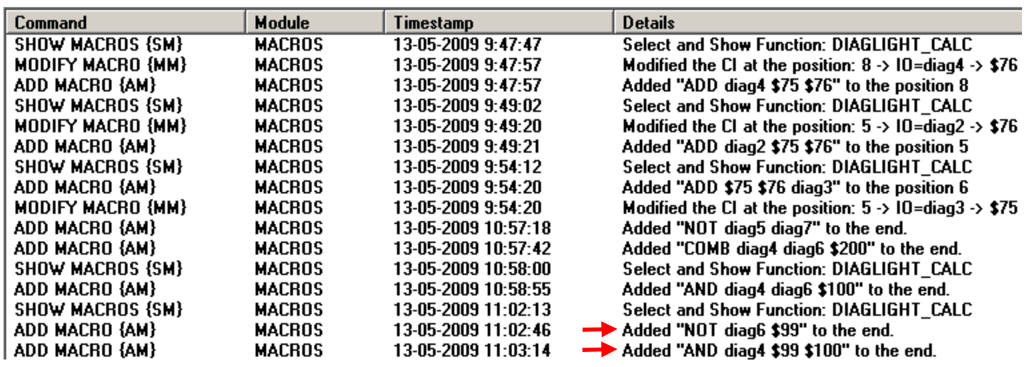


Fig. 5.1:11 – Added "Macros" [NOT] and [AND] through the command-line

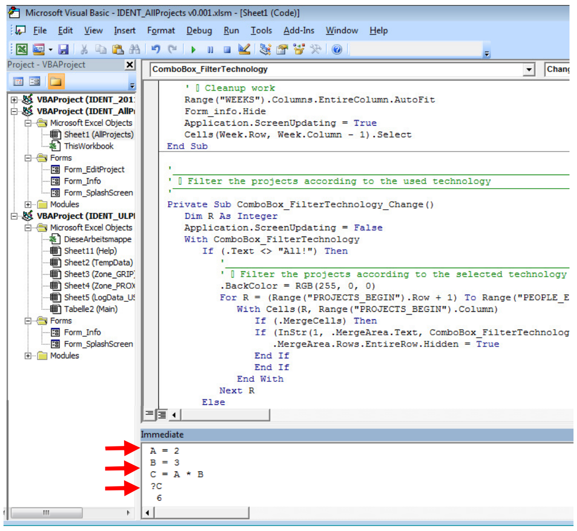
This mechanism also allows for a command-history list to be kept at all times, simply by hooking itself into the command-line and thus intercepting all commands that come from visual actions on the graphical interface or from commands inserted directly into the command-line interface itself (see previous Fig. 5.1:10). Fig. 5.1:12 shows the current command-history list with the last two "ADD" commands being the two that were illustrated in previous Fig. 5.1:11. This list may be useful for the developer to analyze what the customer has been doing up to the point where some error, bug or general malfunction occurred, at the level of the editor itself or the engine manipulation. This list may also be used for "UNDO" and "REDO" action purposes, as in any other flexible editing software packages. This will be further discussed in Chapter 7 as future work. Finally, this history-list is always recorded to disc, thereby retaining all actions made on the *"iEditor"*.



Command	Module	Timestamp	Details
SHOW MACROS {SM}	MACROS	13-05-2009 9:47:47	Select and Show Function: DIAGLIGHT_CALC
MODIFY MACRO {MM}	MACROS	13-05-2009 9:47:57	Modified the CI at the position: 8 -> IO=diag4 -> $76
ADD MACRO {AM}	MACROS	13-05-2009 9:47:57	Added "ADD diag4 $75 $76" to the position 8
SHOW MACROS {SM}	MACROS	13-05-2009 9:49:02	Select and Show Function: DIAGLIGHT_CALC
MODIFY MACRO {MM}	MACROS	13-05-2009 9:49:20	Modified the CI at the position: 5 -> IO=diag2 -> $76
ADD MACRO {AM}	MACROS	13-05-2009 9:49:21	Added "ADD diag2 $75 $76" to the position 5
SHOW MACROS {SM}	MACROS	13-05-2009 9:54:12	Select and Show Function: DIAGLIGHT_CALC
ADD MACRO {AM}	MACROS	13-05-2009 9:54:20	Added "ADD $75 $76 diag3" to the position 6
MODIFY MACRO {MM}	MACROS	13-05-2009 9:54:20	Modified the CI at the position: 5 -> IO=diag3 -> $75
ADD MACRO {AM}	MACROS	13-05-2009 10:57:18	Added "NOT diag5 diag7" to the end.
ADD MACRO {AM}	MACROS	13-05-2009 10:57:42	Added "COMB diag4 diag6 $200" to the end.
SHOW MACROS {SM}	MACROS	13-05-2009 10:58:00	Select and Show Function: DIAGLIGHT_CALC
ADD MACRO {AM}	MACROS	13-05-2009 10:58:55	Added "AND diag4 diag6 $100" to the end.
SHOW MACROS {SM}	MACROS	13-05-2009 11:02:13	Select and Show Function: DIAGLIGHT_CALC
ADD MACRO {AM}	MACROS	13-05-2009 11:02:46	Added "NOT diag6 $99" to the end.
ADD MACRO {AM}	MACROS	13-05-2009 11:03:14	Added "AND diag4 $99 $100" to the end.

Fig. 5.1:12 – Editor commands history-list kept by intercepting the command-line

Besides allowing to manipulate the *"Macros-Sequence"* through addition, deletion, modification and other commands, it is also similarly simply possible to execute *"Macros"* directly through the command-line, just by writing them and hitting "return". This may be useful to test *"Macros"* or to simply execute selected portions of an FDEF by hand (both while the system is stopped or even running, whereas on a running system the *"FDEF-Processor"* executed the incoming command-line *"Macro"* immediately after the current normal *"Macro"* has finished executing). This is done through an extra path that connects the "LP" fetching block directly to the communications block and to the *iEditor* on the desktop or laptop computer. Since the *"Macros"* are themselves 100% self-contained, each command results in a complete operation being executed at that precise point, without the need of other surrounding lexical/syntactical details such as in classical programming languages. Fig. 5.1:13 shows this "immediate window" on the left and the preliminary internal structure of the *"iEditor"* with the command-line on the right.


Microsoft Visual Basic - IDENT_AllProjects v0.001.xlsm - [Sheet1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Project - VBAProject
ComboBox_FilterTechnology
Immediate

```
    ' ▯ Cleanup work
    Range("WEEKS").Columns.EntireColumn.AutoFit
    Form_info.Hide
    Application.ScreenUpdating = True
    Cells(Week.Row, Week.Column - 1).Select
End Sub

'
' ▯ Filter the projects according to the used technology
'
Private Sub ComboBox_FilterTechnology_Change()
    Dim R As Integer
    Application.ScreenUpdating = False
    With ComboBox_FilterTechnology
       If (.Text <> "All!") Then
          '
          ' ▯ Filter the projects according to the selected technology
          .BackColor = RGB(255, 0, 0)
          For R = (Range("PROJECTS_BEGIN").Row + 1) To Range("PEOPLE_E
             With Cells(R, Range("PROJECTS_BEGIN").Column)
                If (.MergeCells) Then
                If (InStr(1, .MergeArea.Text, ComboBox_FilterTechnolog
                   .MergeArea.Rows.EntireRow.Hidden = True
                End If
                End If
             End With
          Next R
       Else
```

```
A = 2
B = 3
C = A * B
?C
 6
```

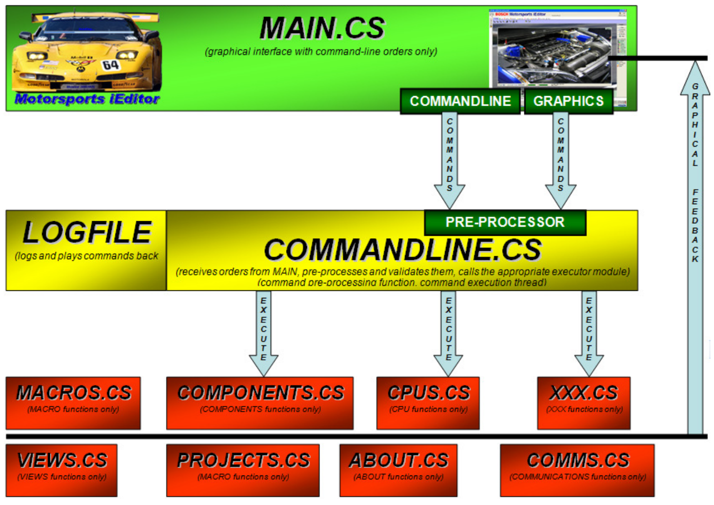


Fig. 5.1:13 – Command-line code block inside the preliminary software structure of the yet-to-become "iEditor"

A good effect of having this command-line visible at all time, is that whenever performing visual operations on FDEFs, windows, menus, etc., the corresponding command-line equivalent is shown. This is a privileged way of learning which commands to use to accomplish the desired effects. Adding *"Macros"*, switching among windows, etc., might get faster for experienced developers, instead of having to operate the mouse all the time.

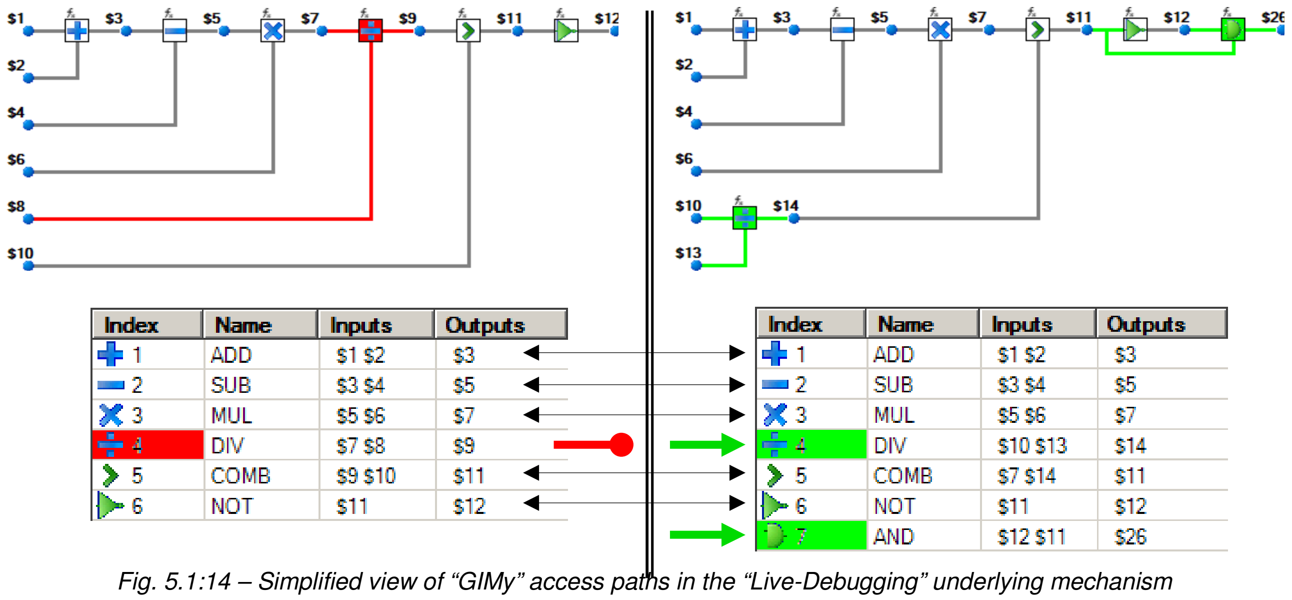
FINAL NOTE: Most classical systems with advanced interfaces have completely depleted any kind of direct low-level experienced user-input. Others have shown that combining both worlds can be very useful/efficient.
From the statements above, ***PITFALL #57***(Command-Line) is thus effectively eliminated!

5.1.1.3 *"LIVE-Comparing"* (on-the-fly FDEFs comparisons)

[Emphasis: effortless FDEF comparison]

Graphical differences between FDEFs can readily be implemented by just comparing those FDEFs at strict *"Macros-Sequence"* level, where only text is compared. While *"Node"* numbering requires a little more extra effort when comparing, a simple text-comparison already allows for graphical differences to be displayed very easily.

The differences in the corresponding *"Macros-Sequences"* are highlighted in visual form, exactly the same way the very *"Macros-Sequences"* are displayed graphically. To illustrate this feature of the *"Macros"*, the code that produces the example comparison depicted in Fig. 5.1:14 just goes through both *"Macros-Sequences"* and just finds obvious and first-level differences. For more in-depth comparisons on FDEFs that differ in more than just a few disperse *"Macros"*, a more complex comparison algorithm would have to be developed. Still, the true power of these comparisons lies in the fact that this algorithm operates directly on a simple textual representation of the FDEFs.



Index	Name	Inputs	Outputs
1	ADD	$1 $2	$3
2	SUB	$3 $4	$5
3	MUL	$5 $6	$7
🟥 4	DIV	$7 $8	$9
5	COMB	$9 $10	$11
6	NOT	$11	$12

Index	Name	Inputs	Outputs
1	ADD	$1 $2	$3
2	SUB	$3 $4	$5
3	MUL	$5 $6	$7
🟩 4	DIV	$10 $13	$14
5	COMB	$7 $14	$11
6	NOT	$11	$12
🟩 7	AND	$12 $11	$26

Fig. 5.1:14 – Simplified view of "GIMy" access paths in the "Live-Debugging" underlying mechanism

FINAL NOTE: Classical graphical editors only very rarely present a code comparison option. When existing, this option is the result of a huge effort that normally does not work very well.

From the statements above, ***PITFALL N#50*** (Differences Inspection) is thus effectively eliminated through the intrinsically simple nature of the *"Macros-Sequence"* that allows for a straightforward comparison algorithm.

5.1.1.4 *"LIVE-Units"* (on-the-fly unit propagation)

[Emphasis: unload the developer from routine tasks]

Often, developers have to manually assign physical units to values in an FDEF. This is cumbersome and error prone, possibly automated altogether. The only thing a developer would have to do is to assign physical units to sensor peripherals and then let the *"iEditor"* go from there, assigning units for the rest of the *"Nodes"* throughout the FDEFs. This would work up to the point where parameters would appear. These too, would need to have a manual output units assignment. Apart from that, any arithmetic operation would result in a combination of its input *"Nodes"* units. Note that some *"Nodes"* may be simple scalar values. The underlying mechanism would naturally be also able to correctly scale and combine differently units, such as "seconds" with "milliseconds". This feature was not completely implemented due to lack of time, but some tests could be done nevertheless.

This feature is technically a "unit propagation" and would operate in an always online mode, that is, the automatically calculated units would appear beneath the newly added *"Macros"* and their respective *"Nodes"*, thus turning out to be a live feature. As in other features and again here, no error reporting at "compile time" is ever generated. The developer only has to add unit information on the inputs, parameter outputs and constants (red units). Afterwards, the *"iEditor"* automatically calculates all the thereafter derived units (blue) until the FDEF outputs are reached. These outputs can then be further propagated to the next FDEFs using them. Matlab Simulink [5] implements a very similar but lower-level mechanism called "value type inheritance and back-propagation". See [Att. 41] and especially Fig. [Att. 41]:143 for more information on this possibility.See

5.1.1.5 *"LIVE-Priorities"* (on-the-fly priority propagation)

[Emphasis: unload the developer from routine tasks]

"Macro" execution priority is herein defined as the frequency with which *"Macros"* are executed, as illustrated in Fig. 4.5:73. Again, as with *"Live-Units"*, there are the originally assigned priorities and the derived/calculated ones. And also again, this priority propagation may initiate at the very first *"Macros"* of the input peripherals and be calculated all the way through, up to the last *"Macros"* of the output peripherals. This mechanism may work in a totally online manner, just like the previous *"Live-Units"*, thus turning this a live feature. This mechanism was not implemented due to lack of time, but can easily be demonstrated based on FDEFs. See [Att. 41] and especially Fig. [Att. 41]:144 for more information on this possibility.

5.1.1.6 *"LIVE-Previewing"* (on-the-fly FDEF grasp)

[Emphasis: have a grasp of the entire FDEF while editing at a finer scale]

This is a smaller feature of the herein implemented *"iEditor"*, in that the user/developer is able to have a complete view of the FDEF, while editing or viewing details of that same FDEF. There is a preview pane on the right, which shows the entire current FDEF and the corresponding area that is being shown on the left detail pane. The user/developer can also move the preview square directly, to move between details of the same FDEF being edited on the left. The preview pane's width can also be changed. Fig. 5.1:15 shows some examples of this tool working, below. This kind if preview panes for visual programming languages were already thought of in StarLogo (Fig. [Att. 6]:16).

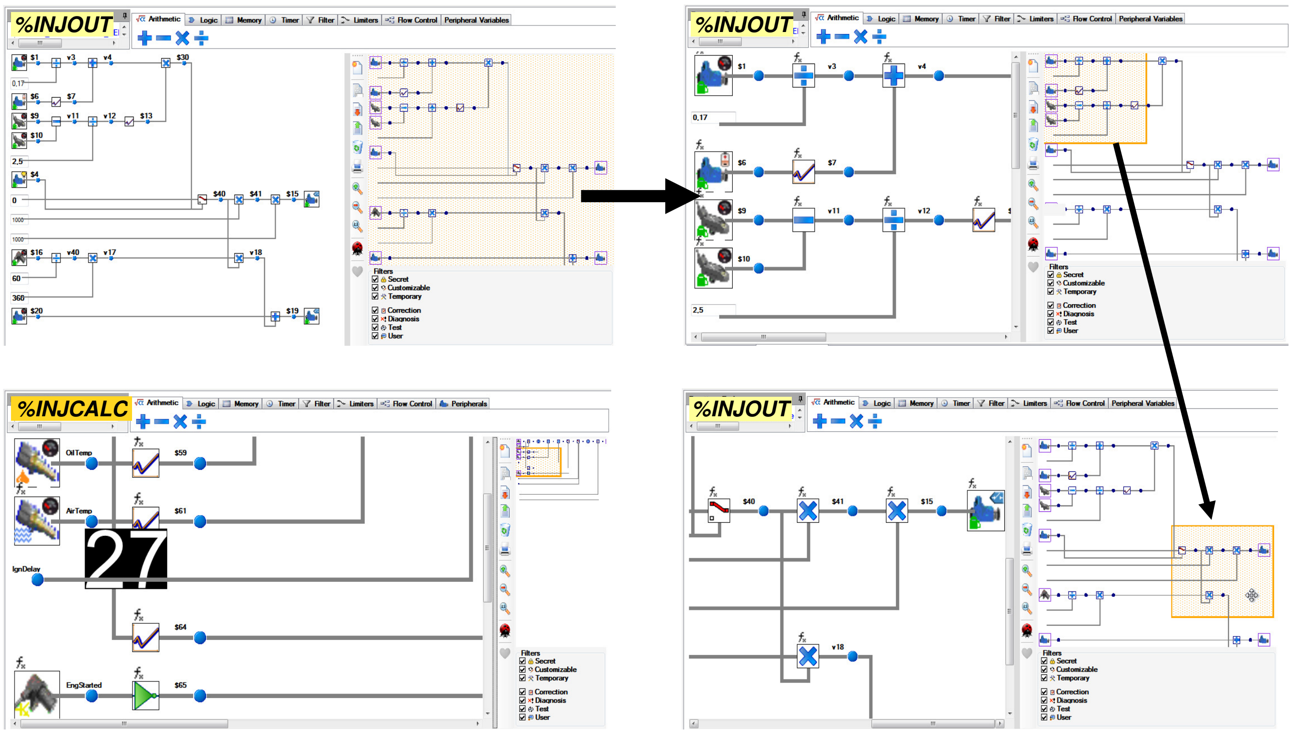


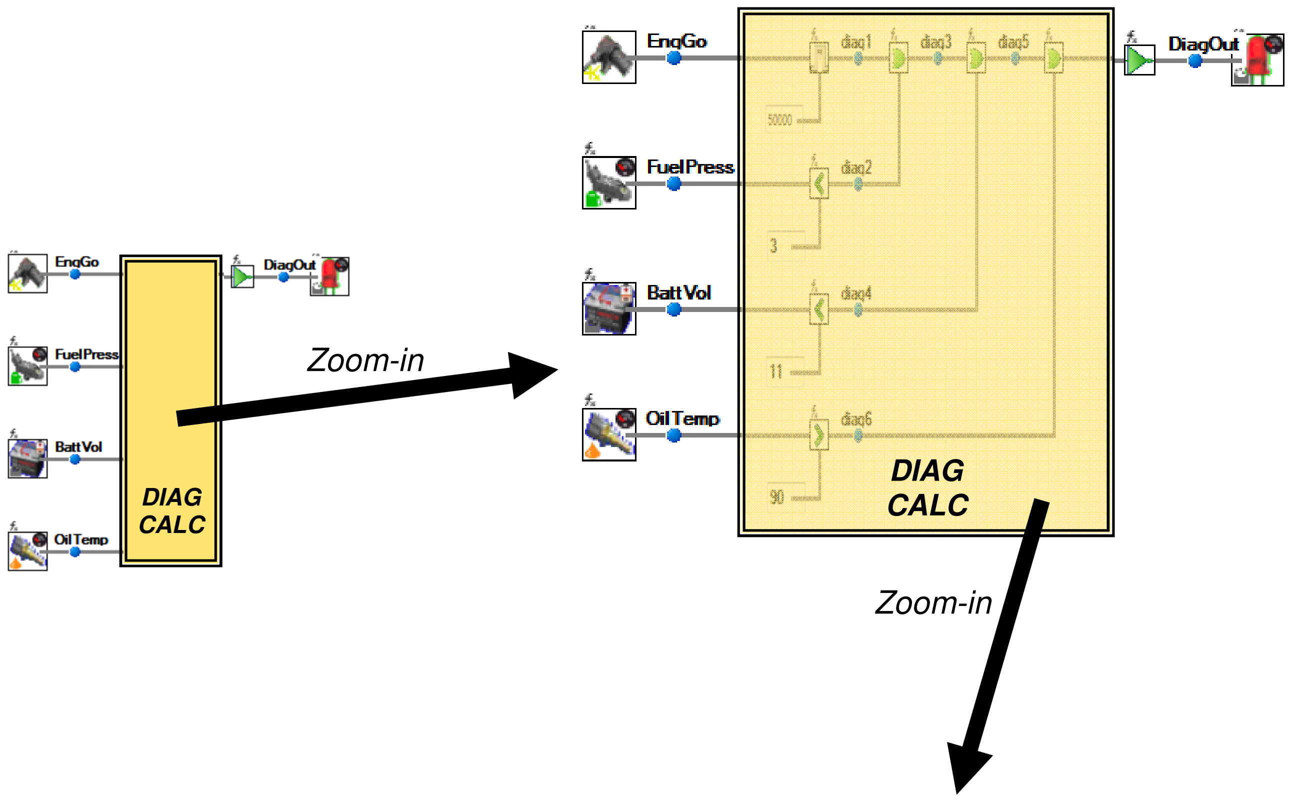
Fig. 5.1:15 – Preview pane under different magnification/zooming situations

5.1.1.7 *"LIVE-Panning/Zooming"* (on-the-fly FDEF grasp)

[Emphasis: seamless zooming into FDEFs and FDEF clusters]

This is an important feature of the *"iEditor"* that extends the functionality and usefulness of the previous *"Live-Previewing"*, in that it allows the user/developer to zoom into the currently shown FDEF while previewing that complete FDEF on the preview-pane, but also allowing to zoom-out even beyond the current FDEF and therefore starting to show the neighbouring FDEFs in a visible way. This then allows the user/developer to have a better grasp of the more global system surrounding an FDEF, to be able to better know where the FDEF's input/output *"Nodes"* come from. This feature has not been fully implemented due to lack of time, but will be illustrated for the sake of clarity, from within the *"iEditor"*.

These methods were included here, since they can and should be combined with bare zooming capabilities, to get more out of them and simple zooming itself. As systems get more complex and larger all the time, integrating all sorts of large chunks of code and data, efficient visualization methods allow the users and developers alike to better deliver their work, by allowing them to better interface themselves to that code and data. Fig. 5.1:16 shows an example of combining this zooming effect to a typical "Hierarchy Abstraction", therefore keeping the surroundings of the hierarchies always visible for better FDEF grasp and contextualization. As the user/developer zooms into the FDEF, the previously opaque hierarchy becomes optionally more and more transparent, until more zooming-in results in showing the inner operations of that hierarchy while the surroundings are optionally faded out gradually. An additional horizontal scrollbar could easily provide for seamless panning to the left and right. If paned in such a way that the left details become more visible, those peripheral FDEFs come into the visible area of the screen and seamlessly blend into current view as well, as effectively as shown in the illustration in Fig. 5.1:17. Current zooming levels also influence this final view and may even trigger the foreground display of the %FUELPRESS FDEF on the right (Fig. 5.1:18). The supporting structure to allow this, would relate to the reserved "hierarchy bit" mentioned in Fig. 4.5:32.



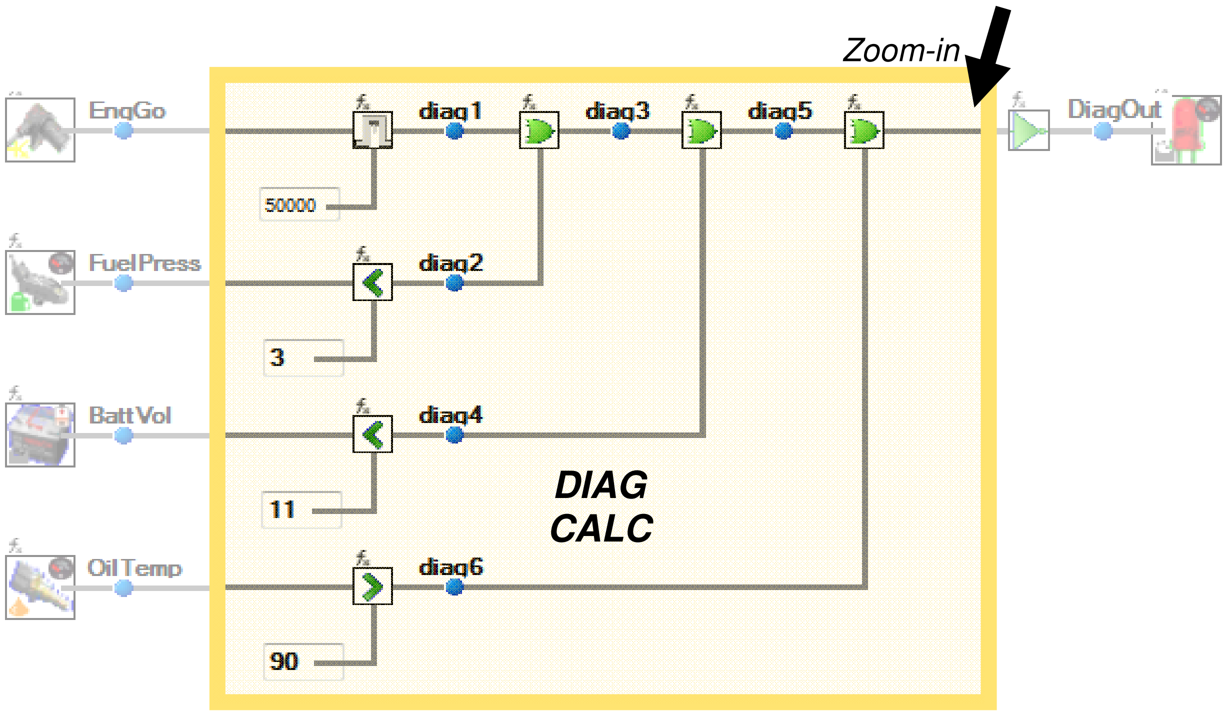


Fig. 5.1:16 – Illustration of the zooming-in effect combined with "Hierarchy Abstraction"

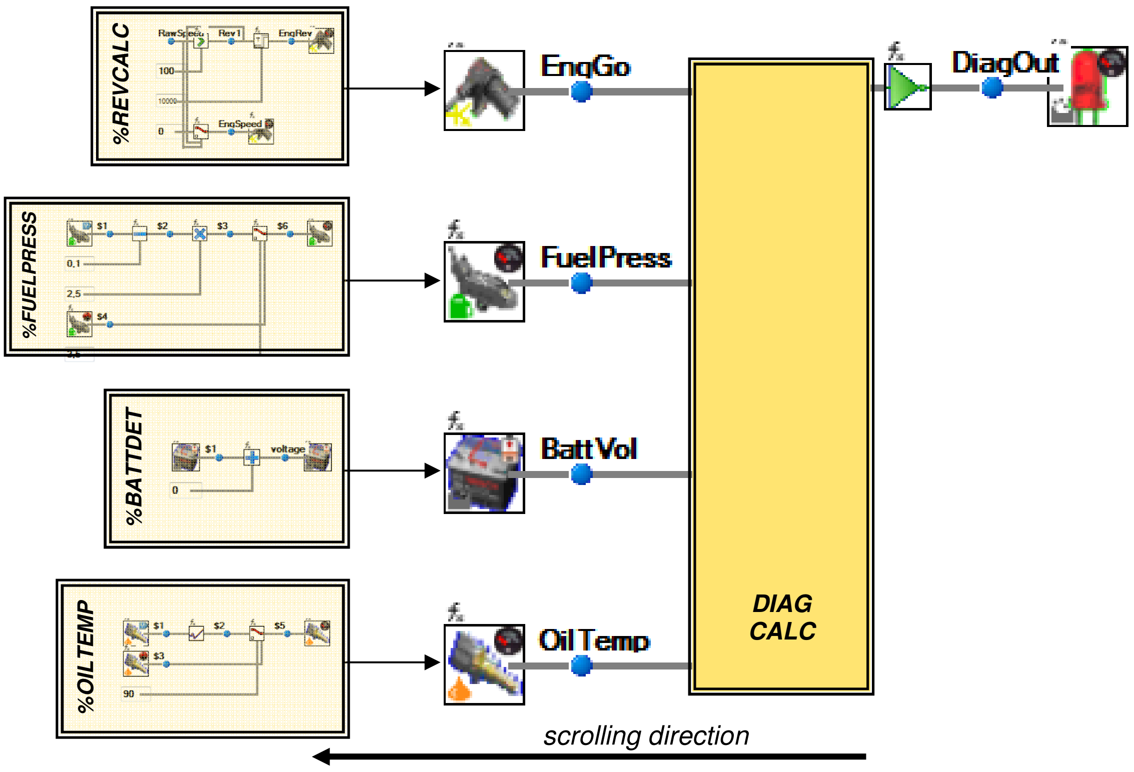


Fig. 5.1:17 – Illustration of the combined pan/zoom effect to the left

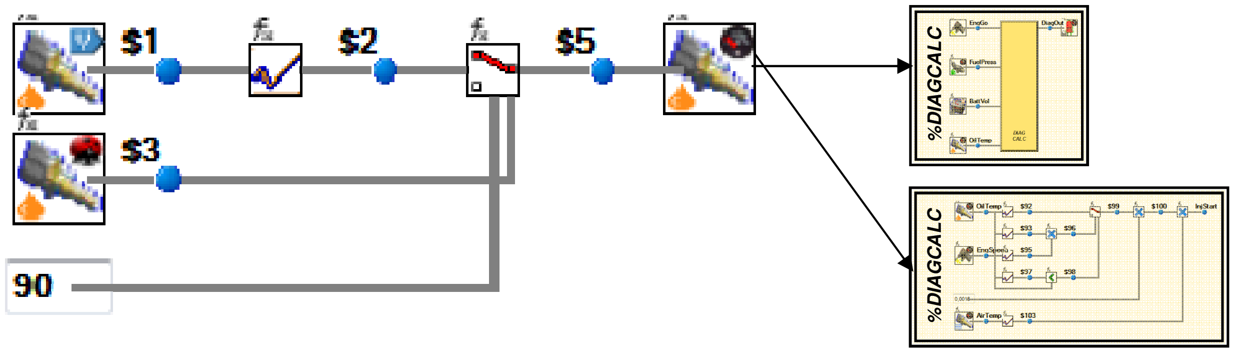


Fig. 5.1:18 – Illustration of the combined pan/zoom effect, even more to the left

5.1.1.8 *"LIVE-Vehicle"* (on-the-fly vehicle/peripherals viewing)

[Emphasis: see the entire vehicle's peripherals on their correct places]

This is a comparatively simple tool, which is also integrated into the *"iEditor"*. It allows the user to have multiple views/photos of the real vehicle, with all the relevant peripherals directly depicted on them and on the right places. Fig. 5.1:19 and Fig. 5.1:20 show examples of what this view looks like. The full list of capabilities offered by this tool, is as follows:

- Place used peripherals on any vehicle view or add new ones from the toolbar
- Click on peripherals to assign a specific FDEF to them
- Click on peripherals to jump to the specific FDEF assigned to them
- Click on peripherals to view its characteristics/datasheet
- On FDEFs, click on peripheral *"Nodes"* to jump to corresponding peripherals on the vehicle view

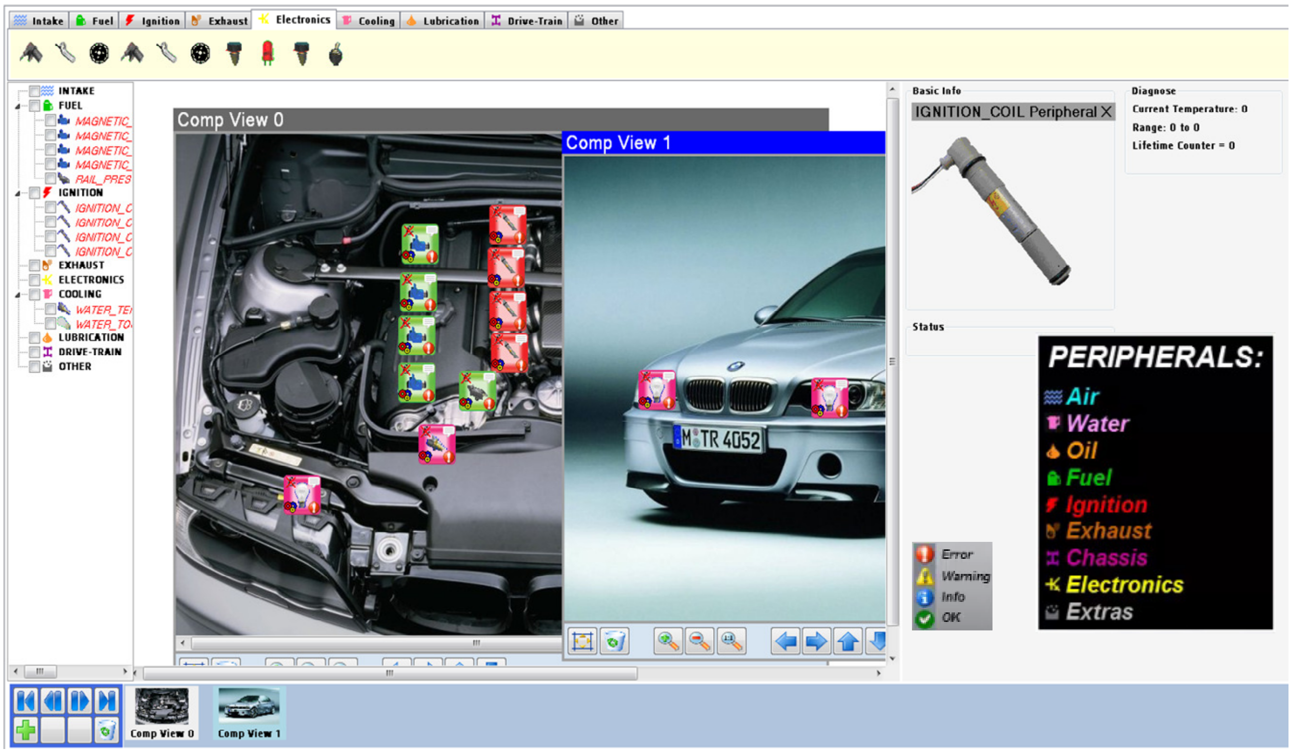


Fig. 5.1:19 – "Vehicle and Peripherals" view/tool inside the "iEditor"

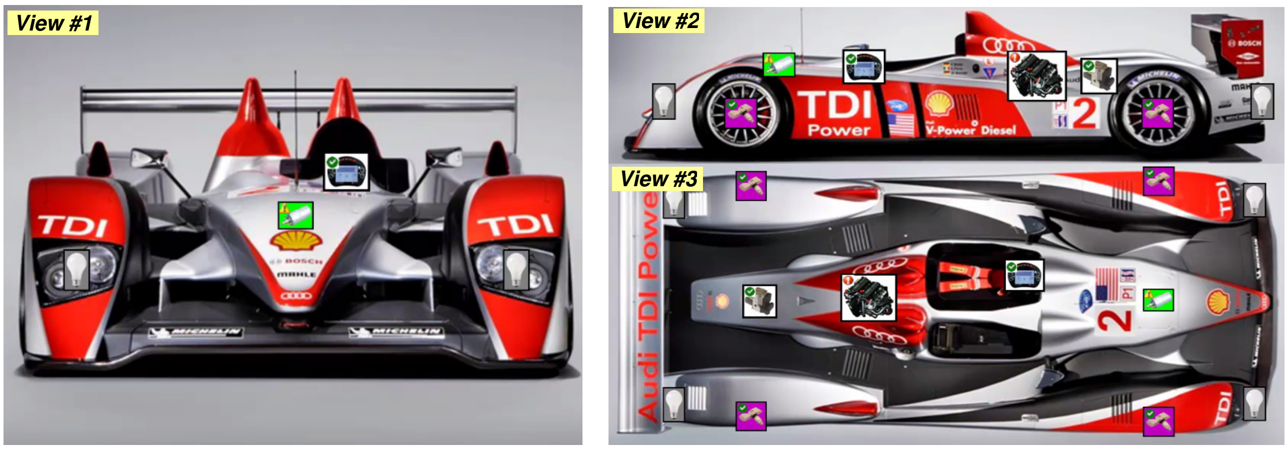
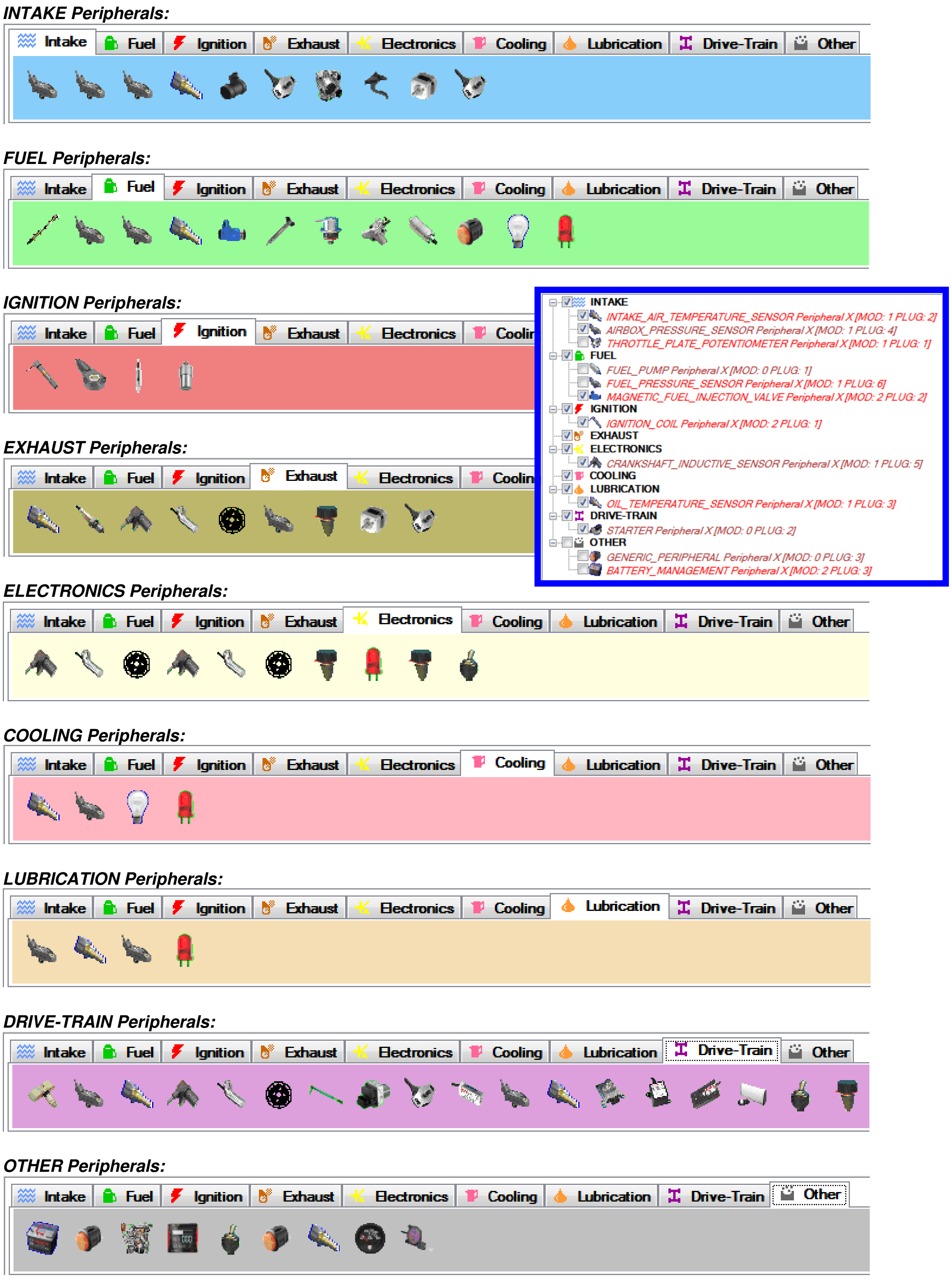


Fig. 5.1:20 – Other views for another project/vehicle, the Audi R12C TDI LeMans with example peripherals

It is in this tool that *"Smart-Peripherals"* are added do a project, for use in the corresponding FDEFs. Only after adding peripherals into these vehicle views, will those peripherals be available to be added to FDEFs and used there. Peripherals are grouped by colour and accessible from a convenient toolbar. These are the available toolbars from which the developer simply drags and drops desired peripherals onto the vehicle photos, where the hierarchical checklist (framed in blue) allows to show/hide them:



```
⊟ ✓ INTAKE
     ✓ INTAKE_AIR_TEMPERATURE_SENSOR Peripheral X [MOD: 1 PLUG: 2]
     ✓ AIRBOX_PRESSURE_SENSOR Peripheral X [MOD: 1 PLUG: 4]
     ☐ THROTTLE_PLATE_POTENTIOMETER Peripheral X [MOD: 1 PLUG: 1]
⊟ ✓ FUEL
     ☐ FUEL_PUMP Peripheral X [MOD: 0 PLUG: 1]
     ☐ FUEL_PRESSURE_SENSOR Peripheral X [MOD: 1 PLUG: 6]
     ✓ MAGNETIC_FUEL_INJECTION_VALVE Peripheral X [MOD: 2 PLUG: 2]
⊟ ✓ IGNITION
     ✓ IGNITION_COIL Peripheral X [MOD: 2 PLUG: 1]
  ✓ EXHAUST
⊟ ✓ ELECTRONICS
     ✓ CRANKSHAFT_INDUCTIVE_SENSOR Peripheral X [MOD: 1 PLUG: 5]
  ✓ COOLING
⊟ ✓ LUBRICATION
     ✓ OIL_TEMPERATURE_SENSOR Peripheral X [MOD: 1 PLUG: 3]
⊟ ✓ DRIVE-TRAIN
     ✓ STARTER Peripheral X [MOD: 0 PLUG: 2]
⊟ ☐ OTHER
     ☐ GENERIC_PERIPHERAL Peripheral X [MOD: 0 PLUG: 3]
     ☐ BATTERY_MANAGEMENT Peripheral X [MOD: 2 PLUG: 3]
```

Fig. 5.1:21 – Toolbars with the different kinds of "Smart-Peripherals"

This *"Live-Vehicle"* tool also allows seamless panning and zooming, in that it allows to combine views, such that the user may zoom into one view and land on another view representing what is being zoomed. Fig. 5.1:22 illustrates a partially seamless zooming operation, where the engine appears beneath the vehicle's back hood, after a zooming-in threshold. Since the photos must be different, this is only a partial zooming-in. Nevertheless, the vehicle is still visible, to allow for better photo-context grasping by the user. On the other hand, Fig. 5.1:23 illustrates a fully seamless zooming into the engine, starting from the normal engine photos @100%, where no new photo is available and, thus, the engine and its peripherals appear simply larger. This is a very useful feature when it comes to viewing highly crowded (with *"Smart-Peripheral"* icons) vehicle photos, where every peripheral is important to be there, while zooming allows to separate them out in a progress fashion. Zooming out is also allowed, thus allowing larger photos into view.

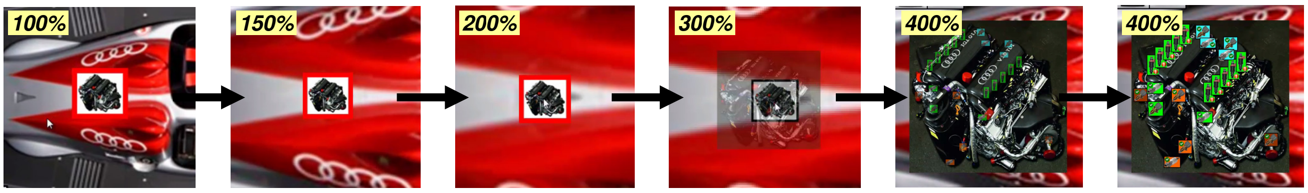


Fig. 5.1:22 – "Seamlessly" zooming into the engine on the top view of the car

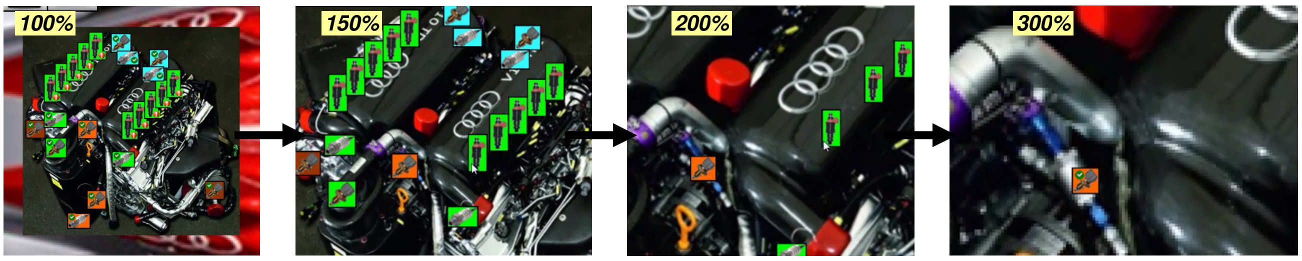


Fig. 5.1:23 – Seamlessly zooming into the engine itself

Furthermore, the *"Live-Vehicle"* view features context-menus which, when called over a "Smart Peripheral", allow the user/developer to quickly access a list of dependencies: the FDEFs using that particular peripheral are shown and may be immediately jumped to through that menu. It also allows the user to directly jump to other vehicle views possessing the exact same peripheral. All this is shown in Fig. 5.1:24 (notice that the icons in Tab. 5.2:66 are also used here). This enhances the overall handling efficiency of the *"iEditor"* one step further, since the user does not have to switch manually among views.

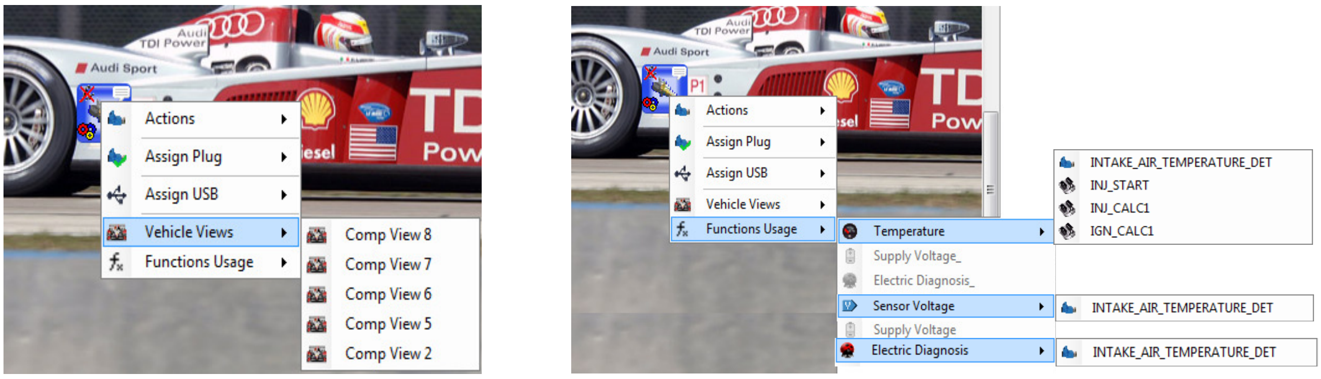


Fig. 5.1:24 – Peripheral context-menu with all the "Vehicle & Peripherals Views" and FDEFs dependencies

5.1.1.9 *"LIVE-Dashboard"* (on-the-fly dashboard design)

[Emphasis: design and view dashboard visual elements]

This tool is also integrated into the *"iEditor"* and allows the user to design and test/simulate dashboard display pages. These can then be downloaded into the real dashboard display and used during races. Several different pages may be set up by simply selecting the desired *"Nodes"* from the left hierarchical list of FDEFs and peripherals containing them, and by selecting the type of display component. These components are exactly the same as for *"Live-Monitoring"* and even use exactly the same underlying mechanism as for *"Live-Monitoring"* explained later on herein. Fig. 5.1:25 shows the editing process. Fig. 5.1:26 then additional pages and additional real displays using the same first page edited here. The green areas are the still free areas that may be filled with further monitoring components.

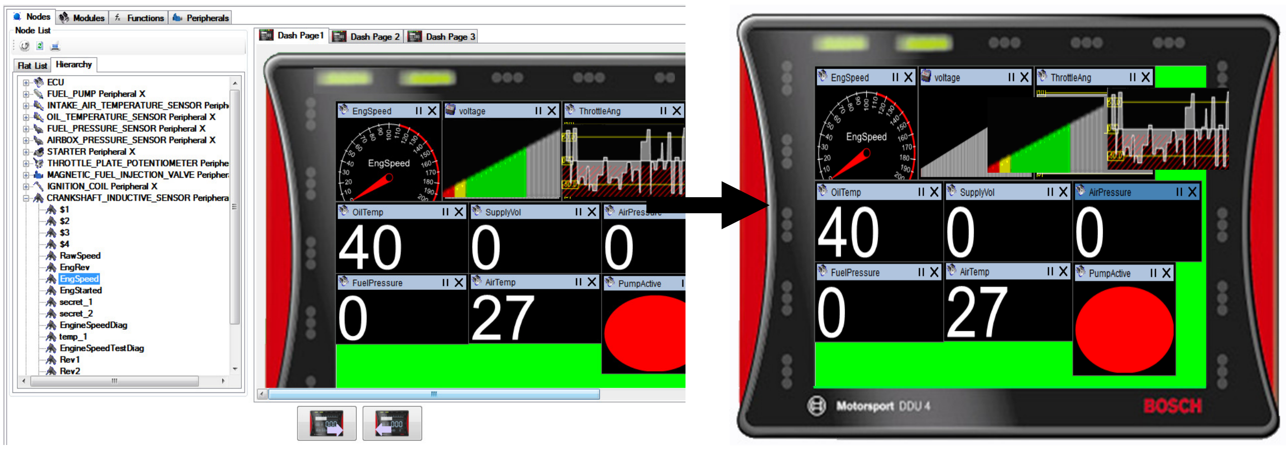


Fig. 5.1:25 – Seamless dashboard design through selecting "Nodes" at the right and the "race mode" viewing

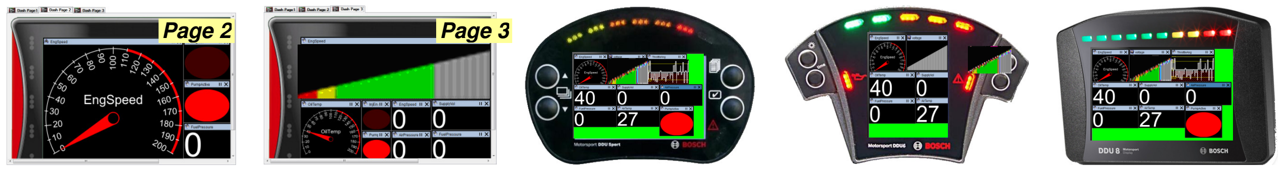


Fig. 5.1:26 – Left: different dashboard pages; Right: different real displays using the same pages

The idea behind the hardware and software for these displays, is to have the exact same *"iEditor"* working inside them, with whatever hardware platform suits running the same development and deployment tool as used for the original Windows desktop computer *"iEditor"* package. This allows re-usage of the exact same package, adapted to the new hardware, thus even theoretically allowing all the operations possible with the original *"iEditor"* but now on a smaller display mounted inside the race-car itself.

This feature uses the exact same mechanism as *"Live-Monitoring"* page composition (see Fig. 5.1:31), through code re-usage, with all the same *"Live-Monitoring"* graphics possibilities. The only difference to *"Live-Monitoring"* explained later on, is the add-on of the background real physical display "mask" for the user to better know the boundaries.

5.1.2 "LIVE-Debugging" made easy

[Emphasis: cut-open and inspection of an alive and running system]

In the *"iEditor"*, *"Live-Debugging"* literally means intrusion-less, fully online and real-time debugging, where *"Node"* values can be freely inspected (monitoring) and also fixated (generated). In the meanwhile, the entire system is working without any interruption and an engine may even be simultaneously running. Additionally, one might also want to stop the program at certain points, execute it step-by-step, among other actions. More details about all the features related to this *"Live-Designing"* and state-of-the-art examples can be seen in [Att. 42].

Fig. 5.1:27 shows the underlying basic hardware mechanism. Lack of intrusion is guaranteed by the *"GIMy"* memory structure where all the loggable *"Nodes"* reside. The debugging path operates parallel to all others in respect of accessing this *"GIMy"*. While reading from the *"GIMy"* is 100% intrusion-less, writing to it may collide with any other path only if the accessed Node is exactly the same. In this later case, the probability will naturally be extremely low and debugging may be considered also practically intrusion-less. Furthermore, as far as priorities are concerned, the other paths have higher priority than this debugging path, in case of collision.

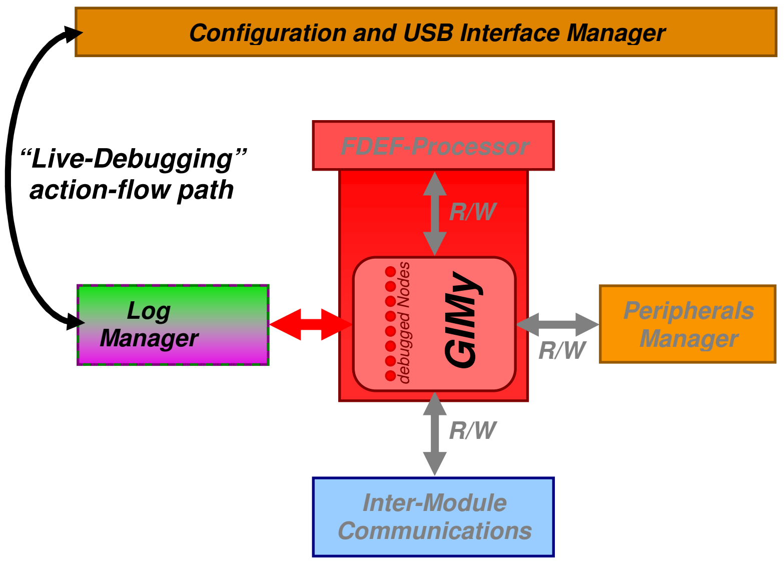

Configuration and USB Interface Manager
"Live-Debugging" action-flow path
FDEF-Processor
R/W
debugged Nodes
GIMy
Log Manager
Peripherals Manager
R/W
R/W
Inter-Module Communications

Fig. 5.1:27 – Simplified view of "GIMy" access paths in the "Live-Debugging" underlying mechanism

This paradigm encompasses exactly three manipulation possibilities, as detailed in the next few sections and listed as follows:

- Graphically viewing *"Node"* values *("Live-Monitoring")*
- Graphically fixating *"Node"* values *("Live-Generating")*
- Graphically stopping execution *("Live-Breakpointing")*
- Graphically stepwise executing an FDEF *("Live-Stepping")*
- Graphically measuring execution times *("Live-Profiling")*

5.1.2.1 *“LIVE-Monitoring”* (on-the-fly *"Node"* values inspection)

[Emphasis: 100% alive, 100% online, 100% while running]

The *“Live-Monitoring”* feature is an integrative part of *“Live-Debugging”* of this system and allows the user to view any *"Node"* value at any time and anywhere throughout the FDEFs. Furthermore, the user may view these values graphically, directly on-FDEF and even directly on the desired *"Nodes"* themselves. Fig. 5.1:28 shows the internal mechanism of this feature relative to the *"GIMy"*, while Fig. 5.1:29 shows an example of this viewing procedure while the engine runs @2426RPM at the very same time, where the user may also choose between various forms of graphical feedback. These forms of feedback range from simple labels, gauges, oscilloscopes, bars or LEDs as shown in Fig. 4.5:45. On the left is the original FDEF and on the right is the same FDEF with several *“Live-Monitoring”* elements. This feature directly uses the logging capabilities of the "Log Manager" component, which reads-out the *"Nodes"* previously set in a "logged items" table. Since this feature requires only reads from the *"GIMy"*, it is therefore 100% intrusion-less. An additional feature when monitoring *"Nodes"* is the automatic detection of value changes, thereby alerting the developer to certain *"Nodes"*. Fig. 4.5:45 shows various visual feedback methods available. Fig. 5.1:30 shows a special visual feedback object for injection and ignition. The *"Nodes"* to be monitored must be marked with the corresponding *"Node Modifier"* ( ) inside the FDEF editor.

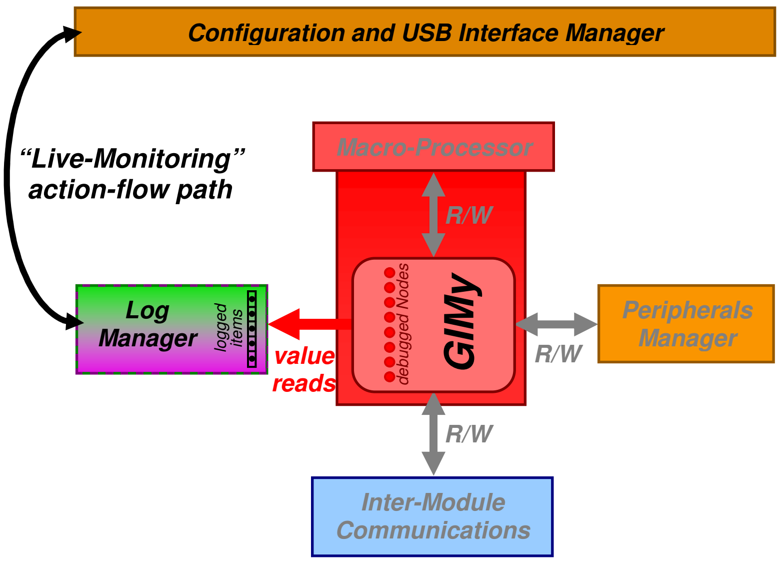

Configuration and USB Interface Manager
“Live-Monitoring” action-flow path
Macro-Processor
R/W
Log Manager
logged items
value reads
debugged Nodes
GIMy
R/W
Peripherals Manager
R/W
Inter-Module Communications

Fig. 5.1:28 – “Live-Monitoring” mechanism relative to the “GIMy”

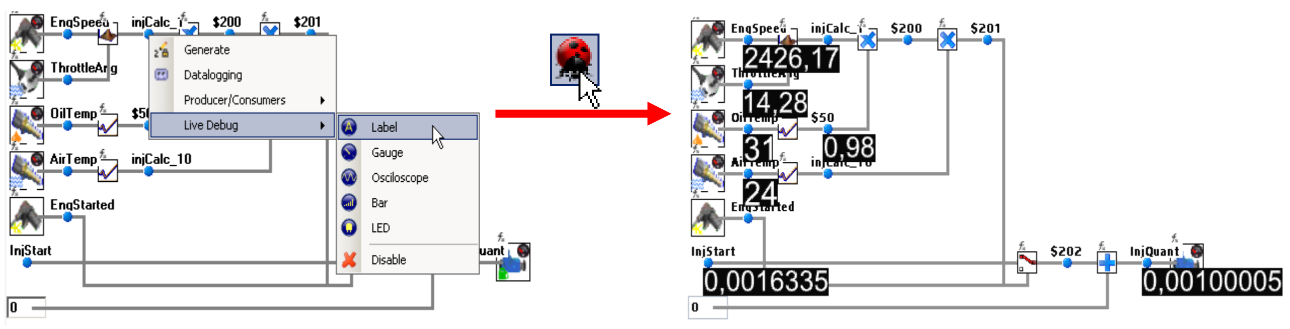


Fig. 5.1:29 – “Live-Monitoring” example on a real FDEF while the engine lively runs @2426RPM

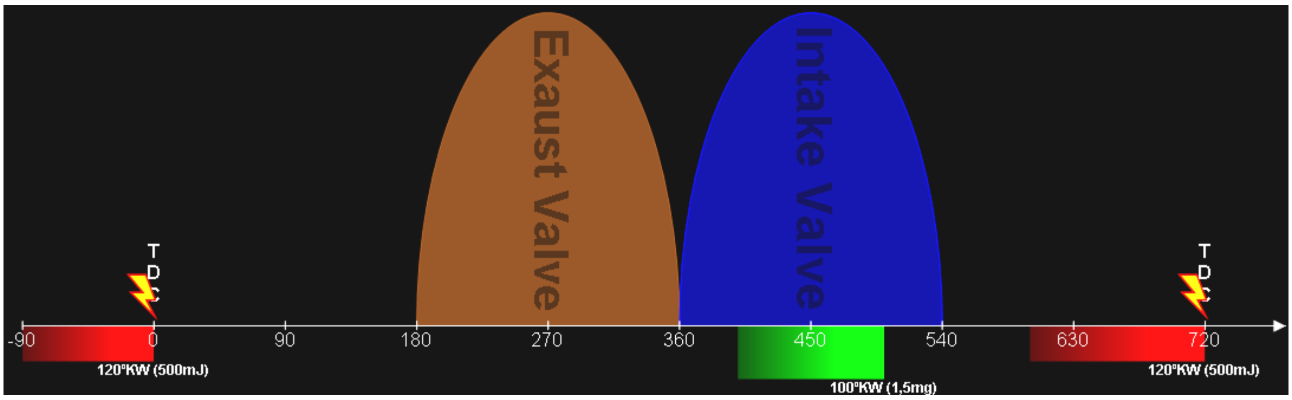


Fig. 5.1:30 – Special visual injection and ignition peripherals' feedback object

Finally, Fig. 5.1:31 shows an alternative and more classical method of composing entire pages of monitoring components, instead of the more direct but sparse way of doing it inside FDEFs directly. This allows the user/developer to prepare several different pages, each containing the needed feedback for specific testing and even racing actions.

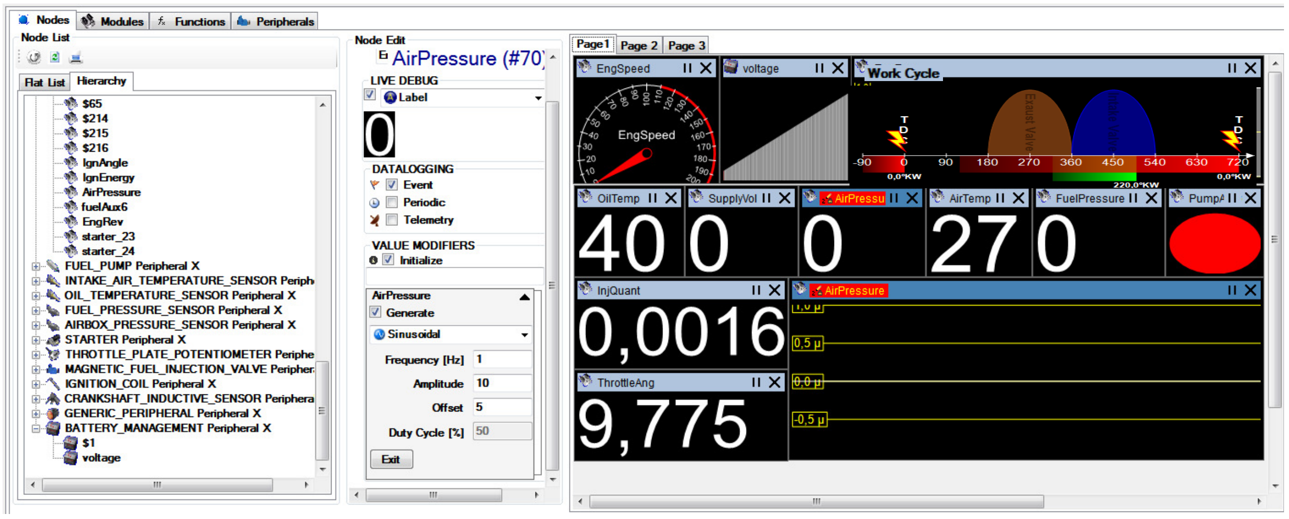


Fig. 5.1:31 – Example of "page" composition with monitoring components

FINAL NOTE: Classical systems normally present intrusive methods for monitoring variable values, besides usually requiring some effort in setting up the corresponding tool to achieve the desired effects.

From the statements above, ***PITFALL #61*** (1/5 – Deferred Prototyping), ***PITFALL #74*** (1/6 –Intrusive Handling) and ***PITFALL #60*** (1/2 – Live Feedback) are thus effectively eliminated through a straightforward and always available values inspection mechanism.

5.1.2.2 *"LIVE-Generating"* (on-the-fly *"Node"* values fixation)

[Emphasis: 100% alive, 100% online, 100% while running]

The *"Live-Generating"* feature is an integrative part of *"Live-Debugging"* of this system and allows the user to fixate any *"Node"* value at any time and anywhere throughout the FDEFs. This can be done graphically, directly on-FDEF and directly on the desired *"Nodes"* themselves. Fig. 5.1:32 shows the internal mechanism of this feature relative to the *"GIMy"*, while Fig. 5.1:33 shows an example of this fixation procedure while the engine runs at the very same time. Note that the injected quantity at the output of the FDEF reflects a similar oscillation as the generated *"Node"* as a result of the calculations. The user may choose between various forms of values' fixation. In this example, a sinusoidal value generation has been chosen. On the left is the original FDEF and on the right is the same FDEF with the *"Live-Generating"* element applied on the engine-speed *"Node"*.

This feature uses a special table of "enable-bits" inside the *"GIMy"*, which enable or disable the write-permission on each *"Node"* for components other than the "Log Manager" (only reads, never writes to the *"GIMy"*). When a *"Node"* value should be fixated, then the "enable-bit" is reset to "0", so that the "Macro Processor", the "Peripherals Manager" and the "Inter-Module Communications" are not allowed to write onto those selected *"Nodes"*, thereby preserving the value of that *"Node"*. The other components still try to write onto those disabled *"Nodes"*, assuming success of those writes, when in reality those *"Nodes"* strictly maintain their values that were set by the *"Live-Generating"* mechanism. For the normal case where *"Node"* read/write behaviour from other components is allowed, the corresponding bits are left at "1". This feature requires a single *"Node"* write action onto the *"GIMy"* when the generating element is a constant, but requires repetitive writes in case of more active generating elements, such as square-waves, sinusoidals, or any other desired signal to be injected into that *"Node"*.

The desired *"Nodes"* to be generated with this method, must be marked with the corresponding *"Node Modifier"* ( ) inside the FDEF editor. Several *"Nodes"* may be generated simultaneously, thus allowing the user/developed to make multiple changes.

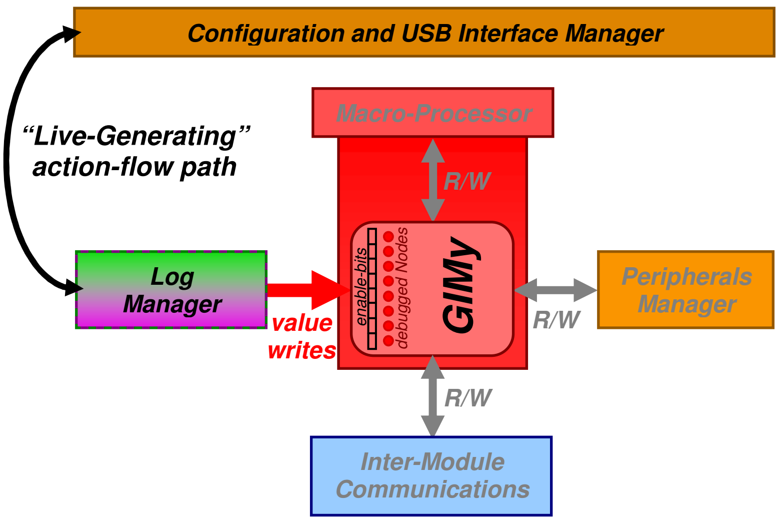
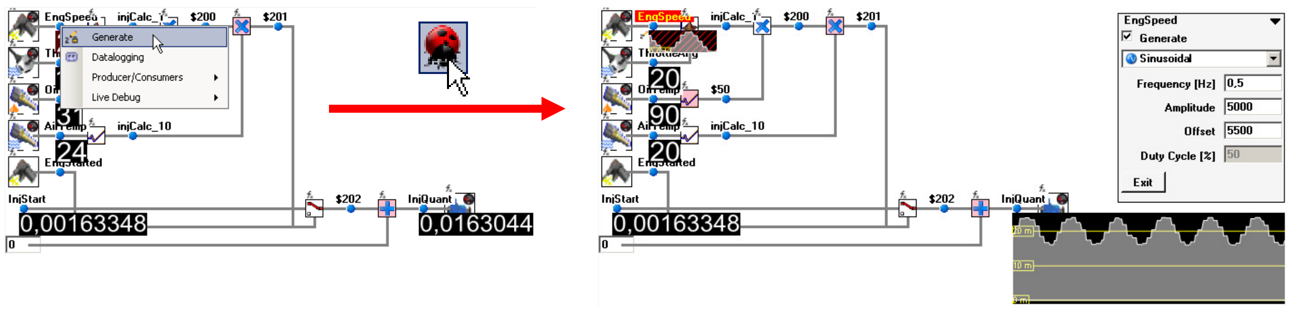

Configuration and USB Interface Manager
"Live-Generating" action-flow path
Macro-Processor
R/W
Log Manager
value writes
enable-bits
debugged Nodes
GIMy
R/W
Peripherals Manager
R/W
Inter-Module Communications

Fig. 5.1:32 – "Live-Generating" mechanism relative to the "GIMy"



EngSpeed
☑ Generate
Sinusoidal
Frequency [Hz] 0,5
Amplitude 5000
Offset 5500
Duty Cycle [%] 50
Exit

Fig. 5.1:33 – "Live-Generating" example on a real FDEF where the engine-speed is overridden

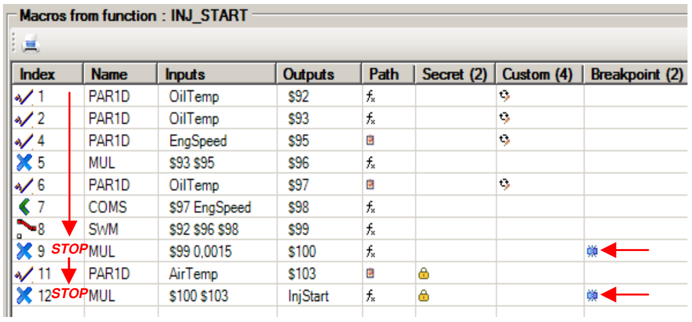
FINAL NOTE: Classical systems normally present intrusive methods for monitoring variable values, besides usually requiring some effort in setting up the corresponding tool to achieve the desired effects.

From the statements above, ***PITFALL H#61***(2/5 – Deferred Prototyping), ***PITFALL H#74***(2/6 –Intrusive Handling) and ***PITFALL H#60***(2/2 – Live Feedback) are thus effectively eliminated through straightforward and visible values setting mechanism.

5.1.2.3 *“LIVE-Breakpointing”* (on-the-fly program interruption)

[Emphasis: interrupting a running program at any time]

This feature was not fully implemented, but some preparatory work was done in the *"iEditor"*, such as the corresponding *"Macro Modifier"* () and the underlying software mechanism that controls the *"FDEF-Processor"*. Breakpoints can be set on *"Macros"* anytime, either on a stopped or running system. As soon as the *"FDEF-Processor"* finds the breakpoint marker, processing is stopped and the *"iEditor"* shows the *"Macro"* where execution stopped, waiting for the developer to take some action such as stepping through or releasing execution altogether. This is briefly illustrated in Fig. 5.1:34.



Macros from function : INJ_START

Index	Name	Inputs	Outputs	Path	Secret (2)	Custom (4)	Breakpoint (2)
1	PAR1D	OilTemp	$92	fx		⚙	
2	PAR1D	OilTemp	$93	fx		⚙	
4	PAR1D	EngSpeed	$95	▣		⚙	
5	MUL	$93 $95	$96	fx			
6	PAR1D	OilTemp	$97	▣		⚙	
7	COMS	$97 EngSpeed	$98	fx			
8	SWM	$92 $96 $98	$99	fx			
9 STOP	MUL	$99 0,0015	$100	fx			▦
11	PAR1D	AirTemp	$103	▣	🔒		
12 STOP	MUL	$100 $103	InjStart	fx	🔒		▦

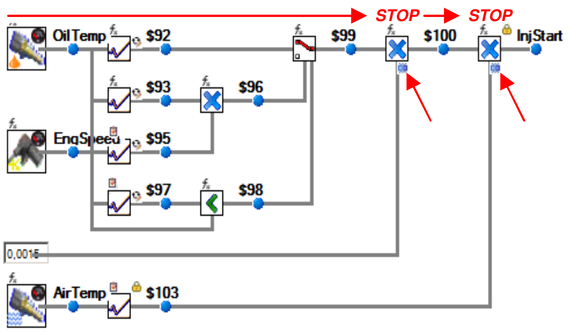


Fig. 5.1:34 – Illustration of a breakpoint coming into action inside an FDEF

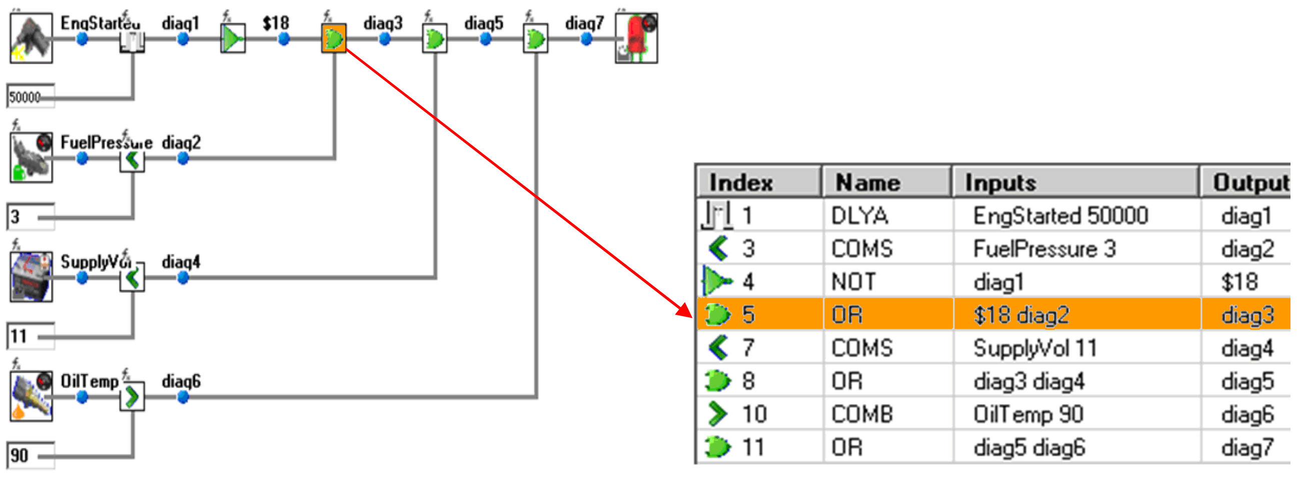
More intelligent breakpointing such as conditional breakpointing can be achieved by simply using *"Live-Prototyping"* for these breakpoints. Extra *"Live-Prototyping"* *"IF Macros"* would then be added to the point where the break is desired, breaking execution as soon as some desired simple or complex condition would be met. Invisibility of these extra *"Macros"* could easily be achieved through one more *"Macro Modifier"* called "Breakpoint Operation" or similar.

5.1.2.4 *“LIVE-Stepping”* (on-the-fly step-wise execution)

[Emphasis: pausing and stepping a running program at any time]

Just in about any other tool allowing for stepwise execution of the code, available buttons on the *"iEditor"* allow the developer to go through an FDEF in a stepwise manner. The currently selected *"Macro"* that is ready to be executed is marked orange directly on the FDEF (Fig. 5.1:35). An additionally available action is that of selecting any desired *"Macro"* in the FDEF and stepping from there. Backstepping is in fact a variant of this additional possibility, in that the *"iEditor"* just tells the *"FDEF-Processor"* to execute the *"Macro"* with index PC-1 (program counter minus one). Since random processing of the *"Macros"* is an intrinsic feature of this *"FDEF-Processor"*, any *"Macro"* may be selected and/or executed next. The possibilities of this *"Live-Stepping"* feature can be listed as follows:

- Resuming execution (▶) until the next pause (⏸) or the next breakpoint ()
- Stepping forwards to the next *"Macro"* (⏭)
- Stepping onto the current *"Macro"* (⏯)
- Stepping backwards to the previous *"Macro"* (⏮)
- Stepping to the selected *"Macro"* (using the mouse and then ⏯ or ⏭)
- Backward debugging (◀, not implemented herein)



Index	Name	Inputs	Output
1	DLYA	EngStarted 50000	diag1
3	COMS	FuelPressure 3	diag2
4	NOT	diag1	$18
5	OR	$18 diag2	diag3
7	COMS	SupplyVol 11	diag4
8	OR	diag3 diag4	diag5
10	COMB	OilTemp 90	diag6
11	OR	diag5 diag6	diag7

Fig. 5.1:35 – Stepwise execution inside an FDEF with currently highlighted "Macro"

Note that backstepping is herein only limited to the execution of the desired previous *"Macro"* but without any past *"Node"* values recovery. Backward debugging in the sense of [486] [487] [488] where logging/tracing and related compile-time preparations have to be done, has not been implemented herein. This could be achieved using the *"Live-Logging"* mechanism explained later on herein.

5.1.2.5 *"LIVE-Profiling"* (on-the-fly execution path and runtime measurements)

[Emphasis: measuring precise execution durations]

Profiling a program chunk is something that may get very important when trying to find problems related to execution speed, such as the following:

- A normal program code chunk may not surpass a certain execution time
- A task code chunk may not surpass a certain execution time-raster
- A critical interrupt code chunk may not surpass a certain execution time
- Optimisation of an on-limit code chunk needed to allow for additional code

Whenever problems arise and execution time is suspected to be too large for a particular code chunk, profiling or calculating the total execution time of that code chunk is of primordial importance to get known. Theoretical timing calculations in classical conditional-laden and CPU mechanisms dependent code is complex, so that usually software or hardware profilers do the job by experiment (by running the code over and over, while measuring real execution durations, making averages and performing max/min measurements). For more information about classical and more innovative profilers and also about the way profiling is much more efficiently done in FPGAs, please see [Att. 55].

"Macros" based data-flow herein takes an easier and much more accurate path, since those *"Macros"* are indivisible operations with fixed execution durations. They do not suffer any timing jitter due to the already detailed inexistence of any unpredictable blocking, accelerating or influencing mechanisms. These durations can thus be simply added up to get the desired total execution duration. Since all this can be made in an offline manner inside the *"iEditor"*, without having to put any hardware to work with it, it turns out to be very similar to a simulation. Therefore, this feature, although existing on its own, could be integrated into the later on explained *"Live-Simulation"* itself as a simulation enhancement.

However, it was left here separately, for clarity. Furthermore, this feature was only experimented with and there was no time to fully integrate it into the final *"iEditor"*.

Bottom-line is that, such as in FPGAs, timings within *"Macros-Sequences"* used herein are very accurate and deterministic. Furthermore, this is exactly what makes it also very interesting for time-critical systems such as the automotive ones, where extensive classical profiling seemed to be the only way to guarantee some degree of correctness and the relative absence of undesired side-effects of complex timing issues. Related code-coverage issues such as complex system variances adding large amounts of time and effort, are also widely reduced. Many papers are written only addressing this "coverage" problem and the unpredictable behaviour of many critical code-chunks. Thus, all that remains herein, is the natural "algorithmic variance and behaviour", which depends solely on the developer himself, now completely free from system side-issues.

FINAL NOTE: Classical systems normally present intrusive, non-realtime and external hardware methods for profiling programs, usually associated with high uncertainty of the results due to high system variance.

From the statements above, ***PITFALL H#62*** $^{\text{(Profiling)}}$ and ***PITFALL O#10*** $^{\text{(4/4 – Code Coverage)}}$ are thus effectively eliminated through the virtual elimination of system variances and associated complexity of profiling and code-coverage itself.

5.1.3 “LIVE-Prototyping” made easy

[Emphasis: fearless hands-on manipulation of an alive and running system]

This feature may well be the most important of this thesis' work, since it allows something that is, as well, the most difficult yet useful feature of any automotive system: the possibility of developing, changing, editing and even programming the whole system, while maintaining full engine operation without any limitation. It is more than just debugging, since it really allows making any changes to any part of the FDEFs keeping the system fully operational. Concretely speaking, this *"Live-Prototyping"* paradigm consist of being able to manipulate the *“Macros”* of an FDEF in a 100% live manner, without any re-allocation, re-compile or complete download whatsoever, while maintaining a running plant (an automotive engine, for example). More details about all the features related to this *"Live-Prototyping"* and state-of-the-art examples can be seen in [Att. 43].

Fully flexible manipulation options, imperceptible change commitment delays and virtually ultra-transparent operational details, would be the main visible requirements from the user’s point of view. *"Live-Prototyping"* developed and implemented in this thesis' work allows all this with little conceptual effort and also relatively easy to manage hardware and software efforts. The central reason for the ease of implementation of this feature again derives from the *“Macros”* themselves: these are self-contained entities which execute and are manipulated independently from each other and even from the system itself. In other words, it is not the system that determines the behaviour of the *”Macros”*, but it is the *“Macros”* themselves that determine the program’s and ultimately the system's behaviour. Therefore, the running system is just a mere platform to hold them and can thus be much more easily adapted to allow for these features, including *“Live-Prototyping”*.

Fig. 5.1:36 below unveils the central mechanism for this feature to work inside the hardware system without disrupting other structures. Note that the "FDEF-Processor" is depicted in its simplified form. The *"Macros-FLASH"* is the normal storage of *“Macros”*

when these are downloaded into the hardware, while the *"LP-RAM"* is the volatile *“Live-Prototyping”* storage where the temporarily downloaded *"LP-Macros"* reside. The *"LP"*-block is the component that manages the *“Live-Prototyping”* process at hardware level, by selecting the *“Macros”* that are to be executed by the *"Macros-Processor"*. It came between the original CPU-FLASH path, as a *"Macros"* selection gateway, to now allow this extra *“Live-Prototyping”* feature without the CPU ever noticing this change. When *"Live-Prototyping"* is not currently active, then it also allows for selectively fetching *"Macros"* according to certain modifier bits. Even more generally speaking, this block is responsible for fetching the *"Macros"* from the external memory banks. As already shown in previous Fig. 4.5:78 and again here in Fig. 5.1:36, the "LP" block is even capable of receiving direct "Macros" from the USB communications path.

The only needed upper software action-paths are the ones through which the *"iEditor"* writes new *"LP-Macros"* into the *"LP-RAM"* and through which the *"LP"* switching table is filled. Depending on the changes made (*"Macro"* additions, deletions and changes) this *"LP"* switching table decides where to switch from the original *"Macros-Sequence"* contained in FLASH and to jump into this new and temporary *"Macros-Sequence"* contained in RAM. This *"LP"*-block really just fetches between FLASH and RAM according to the switching table downloaded.

This entire procedure begins by pressing the *“Live-Prototyping”* button on the *"iEditor"* and by then starting to make changes to the program and parameters in a fully visual way. Pressing the button only ensures that the *"iEditor"* becomes aware of the fact that all subsequent actions are to be regarded as temporary *"Live-Prototyping"* actions.

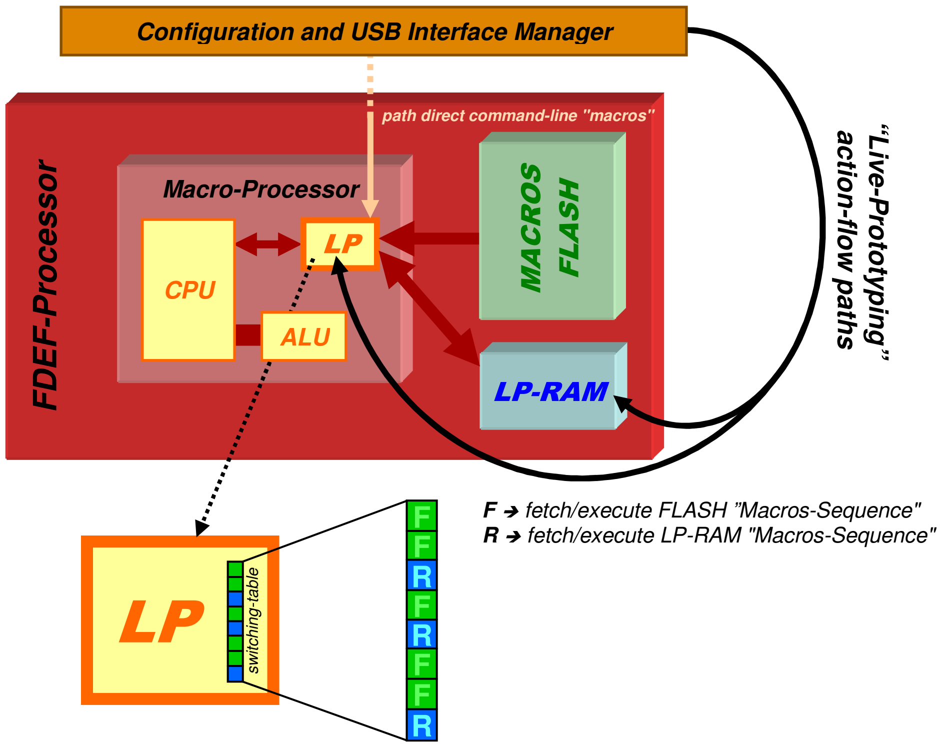

Configuration and USB Interface Manager
path direct command-line "macros"
FDEF-Processor
Macro-Processor
CPU
LP
ALU
MACROS FLASH
LP-RAM
“Live-Prototyping” action-flow paths
LP
switching-table
F
F
R
F
R
F
F
R
F → fetch/execute FLASH "Macros-Sequence"
R → fetch/execute LP-RAM "Macros-Sequence"

Fig. 5.1:36 – Simplified view of Macro-Processing with the “Live-Prototyping” underlying mechanism

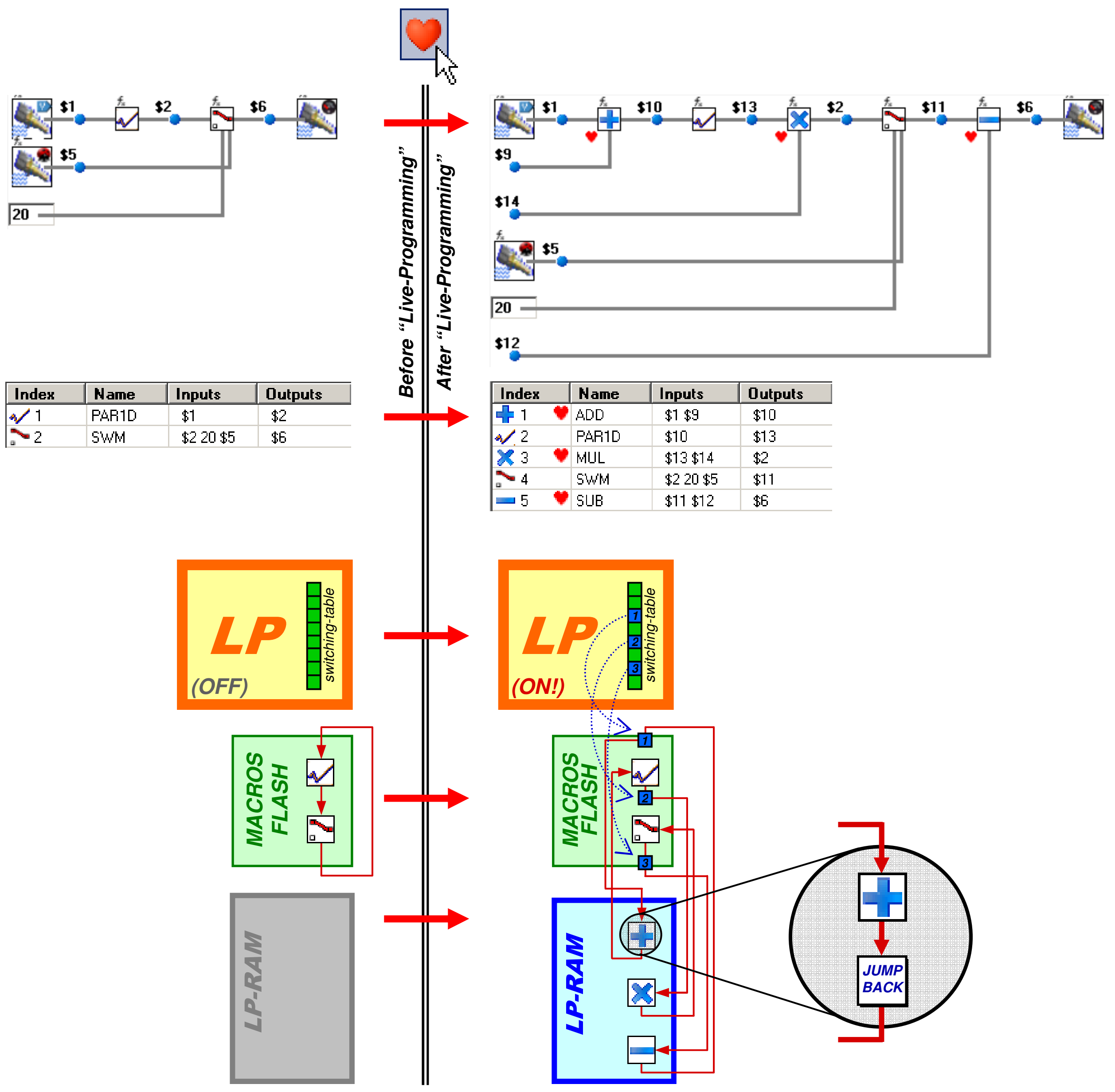
This paradigm encompasses exactly four manipulation possibilities, as detailed in the next few sections and listed as follows:

- Adding new "Macros" ("Live-Programming")
- Deleting existing "Macros" ("Live-Programming")
- Changing existing algorithm "Macros" ("Live-Programming")
- Changing existing parameter "Macros" ("Live-Calibrating")
- Testing temporary "Macros" ("Live-Testing")

5.1.3.1 *"LIVE-Programming"* (on-the-fly program changing)

[Emphasis: 100% alive, 100% online, 100% while running]

Fig. 5.1:37 shows a detailed example of the process of adding *"Macros"*. The new *"Macros"* are added onto an already existent FDEF, with *"Live-Prototyping"* mode enabled. These changes take usually less than 50ms for each *"Macros"*), well below the normal delay noticeable to the user. Entire contiguous sequences of new *"Macros"* can also be added between the already existing ones.

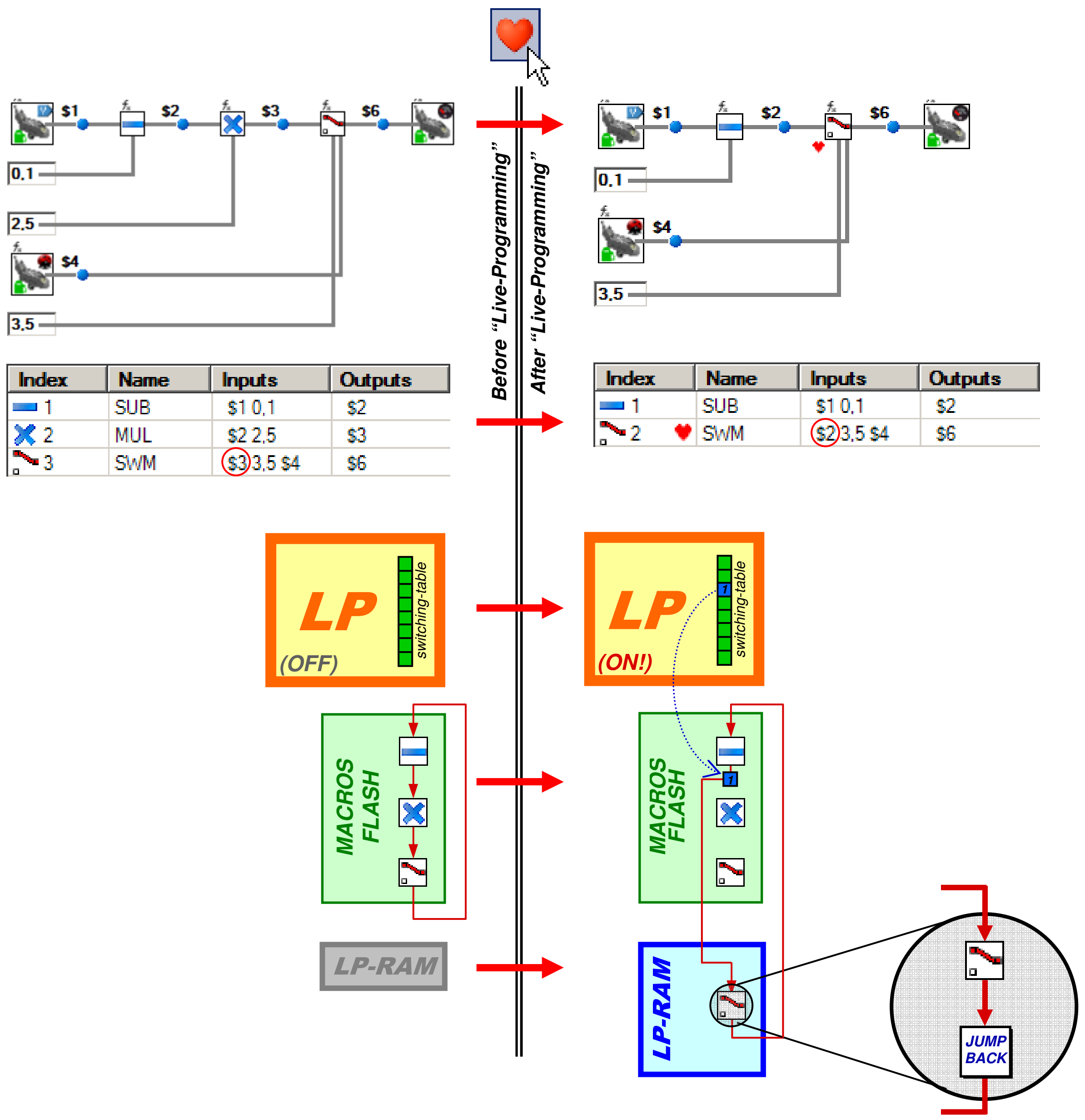


Index	Name	Inputs	Outputs
1	PAR1D	$1	$2
2	SWM	$2 20 $5	$6

Index	Name	Inputs	Outputs
1	♥ ADD	$1 $9	$10
2	PAR1D	$10	$13
3	♥ MUL	$13 $14	$2
4	SWM	$2 20 $5	$11
5	♥ SUB	$11 $12	$6

Fig. 5.1:37 – Simplified view of the "Live-Programming" addition example with the internal hardware actions

Fig. 5.1:38 shows an example where an already existing *"Macro"* is deleted. In this example, the *"LP"* switching table is loaded in a way where the deleted *"Macro"* is simply by-passed, whereas the next *"Macro"* has to be loaded into *"LP-RAM"* so that it now gets its input from *"Node $2"* instead from *"Node $3"* of the herein deleted multiplication *"Macro"*. Another option would be placing a "Null-Macro" into the "LP-RAM" with the sole intent of passing *"Node $2"* through *"Node $3"*, but the implemented solution is better in terms of timing, since the deleted *"Macro"* virtually disappears.

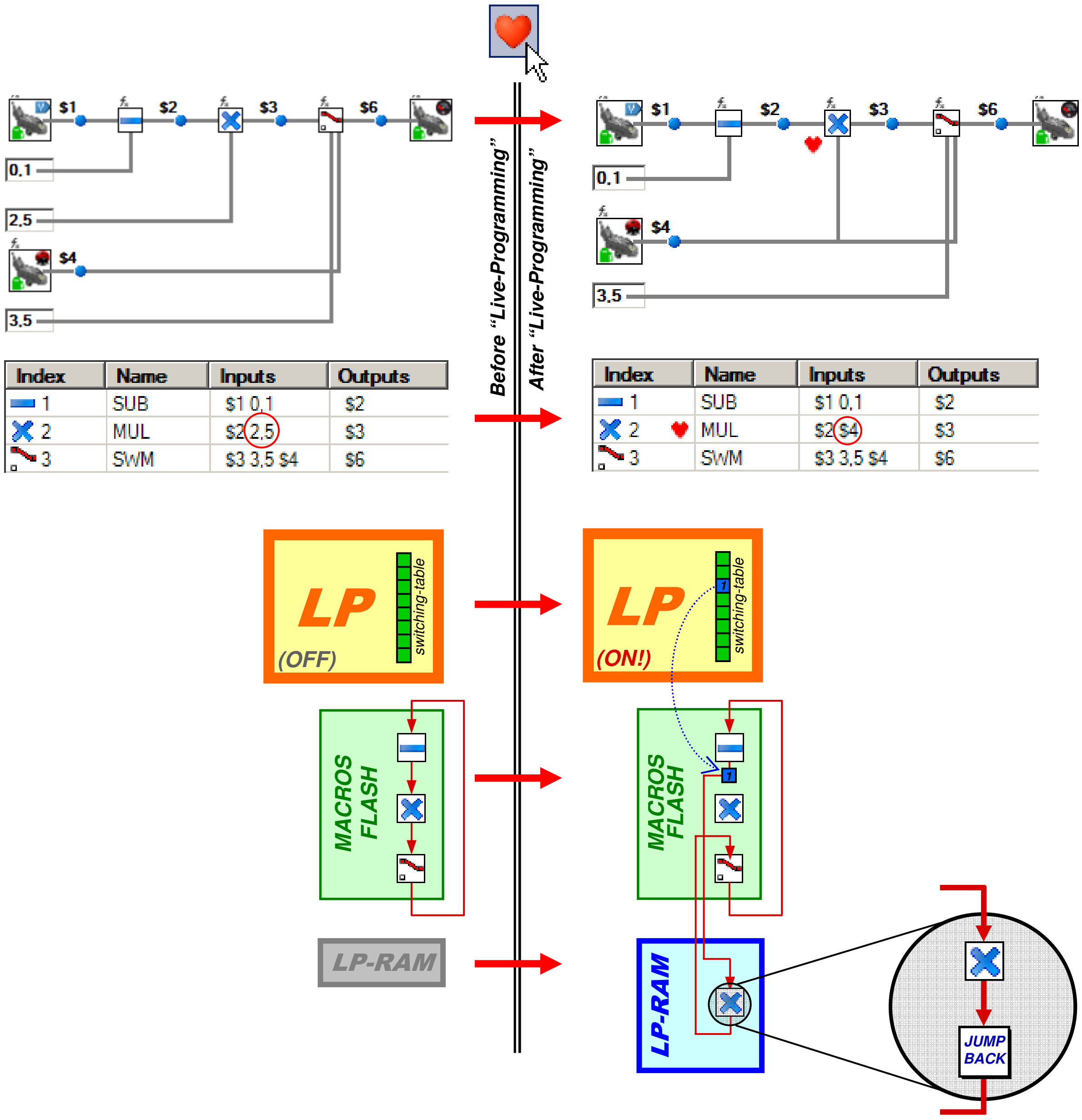


Index	Name	Inputs	Outputs
1	SUB	$1 0,1	$2
2	MUL	$2 2,5	$3
3	SWM	$3 3,5 $4	$6

Index	Name	Inputs	Outputs
1	SUB	$1 0,1	$2
2	SWM	$2 3,5 $4	$6

Fig. 5.1:38 – Simplified view of the "Live-Programming" deletion example with the internal hardware actions

Fig. 5.1:39 shows an example where an already existing *"Macro"* is changed in its input nodes. The detail difference is that the *"Macro"* is not added or deleted, but just loaded into *"LP-RAM"* for content editing. Operationally, a sort of “bypass” thus occurs.



Index	Name	Inputs	Outputs
1	SUB	$1 0,1	$2
2	MUL	$2 2,5	$3
3	SWM	$3 3,5 $4	$6

Index	Name	Inputs	Outputs
1	SUB	$1 0,1	$2
2	MUL	$2 $4	$3
3	SWM	$3 3,5 $4	$6

Fig. 5.1:39 – Simplified view of the “Live-Programming” change example with the internal hardware actions

Right after adding these new "*Macros”*, they are immediately downloaded and placed into the *"LP-RAM"*. At the same time, the *"LP"* switching table is updated accordingly. This all happens while the *"Macros-Processor"* continuous to execute. As soon as the program-counter gets to a count that is stored in the *"LP"* table as being a *"LP-Macro"*, then the next *"Macro"* supplied to it will be one from *"LP-RAM"*. From there, the sequence of *"LP-Macros"* is executed until an order to jump back is encountered in that sequence. This jumping back is fully naturally achieved through a *"JMP Macro"*. The jump address is calculated by the *"iEditor"* upon each *“Live-Prototyping”* action, so that it points back to the next normal *“Macro”* in the *"Macros-Sequence"* present in the *"Macros-FLASH"*.

At the end of these prototyping actions, the developer may choose between simply discarding those changes or consolidating them. While the first option causes the *"iEditor"* to simply erase the *"LP-RAM"* and the corresponding entries in the *"LP"* switching table (thereby discarding the changes), the second option causes it to re-sequence all *"Macros"* by combining the normal ones with the *"Live-Macros"* and then stopping the system to re-download the complete *"Macros-Sequence"* (now including the previous *"LP-Macros"* as being part of the normal sequence in FLASH). The later option takes a little longer and should only be done when all prototyping work is finalized. As an example, downloading the complete set of FDEFs used herein to operate the engine at the end of this thesis, typically takes about a few seconds to fully download into hardware.

"Routing-tables" are tables in RAM containing the *"Node"* indexes especially for those *"Nodes"* that need to be routed from the peripherals interface to/from the *"GIMy"* and to/from the *"Inter-Module Communications"*. These are filled by the *"iEditor"* as well (more specifically by the *"Allocator"* described later on herein). Whenever some *"Node"* is changed in that this change involves changing some "routing-table" (e.g. an FDEF was using peripheral *"Node #123"* and now the user changed in to using peripheral *"Node #234"* instead), those "routing-tables" have to be reloaded. Here, since these tables are usually very small when compared to the FDEFs, instead of using such a mechanism as the above for the *"Macros"*, a simple reload is done. "Routing-table" reload is potentially triggered upon the following user actions on the FDEFs:

- A peripheral *"Node"* is added to be used in FDEFs
- A peripheral *"Node"* is deleted from FDEFs or not used any longer
- A *"sink Node"* coming from another module is added to be used in FDEFs
- A *"sink Node"* coming from another module is deleted from FDEFs or not used any longer

As long as RAM is available, form simplicity reasons, only *"Node"* additions trigger a reload, since in case of deletion or non-usage there is no immediate need to reload the affected table. In the case of the *"Inter-Module Communications"* both end-point tables have to be reloaded, one to allow for the produced *"source Node"* to be transmitted to the consumer(s) and the other to allows that incoming *"Node"* to be placed onto the *"GIMy"* for consumption as a *"sink Node"*. The "Routing-table" inside the "Log Manager" is a special case, which is managed by the logging user-interface only.

As a reference for the needed effective effort to get this feature working for the very first time in a software "prototype status" fashion, it can be said that it took about 2 man-weeks for this first step. For the final version with all the needed possibilities, 8 additional man-weeks were needed. This attests to the relative easy with which the most important features of this work were added to this system globally.

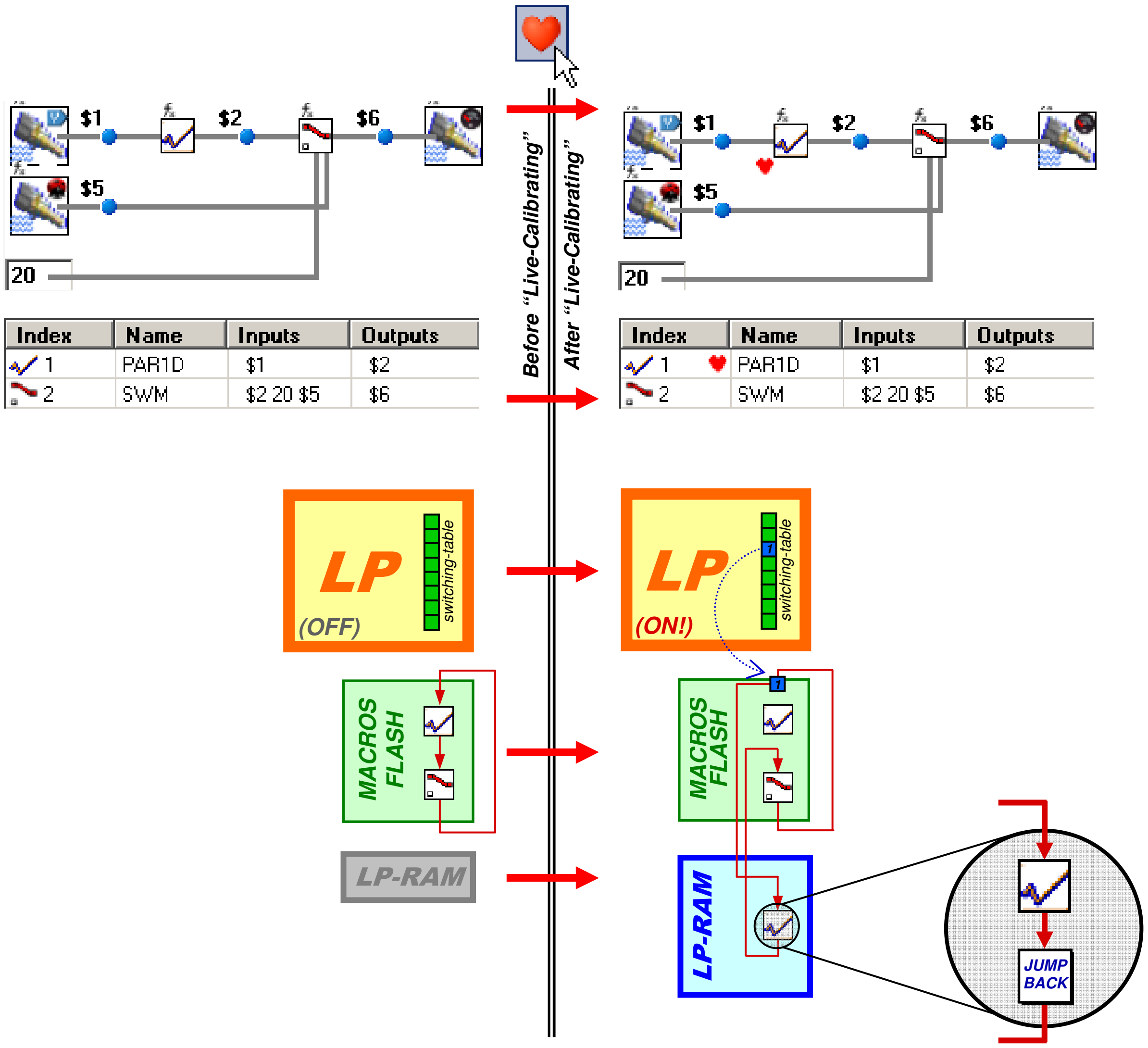
FINAL NOTE: Classical systems do not allow modifying any part of code during execution. There is no way of doing this. As far as known, no system in the world allows for true online code change during execution.

From the statements above, **PITFALL H#61** (3/5 – Deferred Prototyping) and **PITFALL H#74** (3/6 – Intrusive Handling) are thus effectively eliminated through a straightforward and always available code modification mechanism.

5.1.3.2 *“LIVE-Calibrating”* (on-the-fly parameter setting)

[Emphasis: 100% alive, 100% online, 100% while running]

Calibrating values inside constants, parameter curves and maps, is just another useful and especially needed variant of the same *"Live-Prototyping"* mechanism, whereby the desired constant, curve or map is loaded also into the *"LP-RAM"* to be freely edited. Since these types of parameter *“Macros”* are treated exactly the same way as all others, there is no functional or operational difference whatsoever between editing normal *“Macros”* and these parameter *“Macros”*. In fact, this is exactly the same as changing algorithm *"Macros"*, but applied to changing parameter *"Macros"* specifically. Fig. 5.1:40 shows a detailed example of this process of calibrating parameters live. Exactly as in changing algorithm *"Macros"*, here they are also loaded into *"LP-RAM"* just for content editing. Again as before, this is operationally a sort of “bypass”. As soon as all the calibration work is done, checked, validated and accepted, the developer has the same choices as for the previous related *"Live-Programming"*, where he can opt to discard the changes or to consolidate them into the FDEF.



Index	Name	Inputs	Outputs
1	PAR1D	$1	$2
2	SWM	$2 20 $5	$6

Index	Name	Inputs	Outputs
1	♥ PAR1D	$1	$2
2	SWM	$2 20 $5	$6

Fig. 5.1:40 – Simplified view of a “Live-Calibrating” example with the internal hardware actions

As a final note, the word "calibration" was used for this parameter *"Macro"* manipulation, derived from the fact that it is globally used in the automotive scene, to name that type of FDEF manipulation. Exactly as in any automotive development system, parameters can be calibrated to new values anytime, either on a stopped or running system.

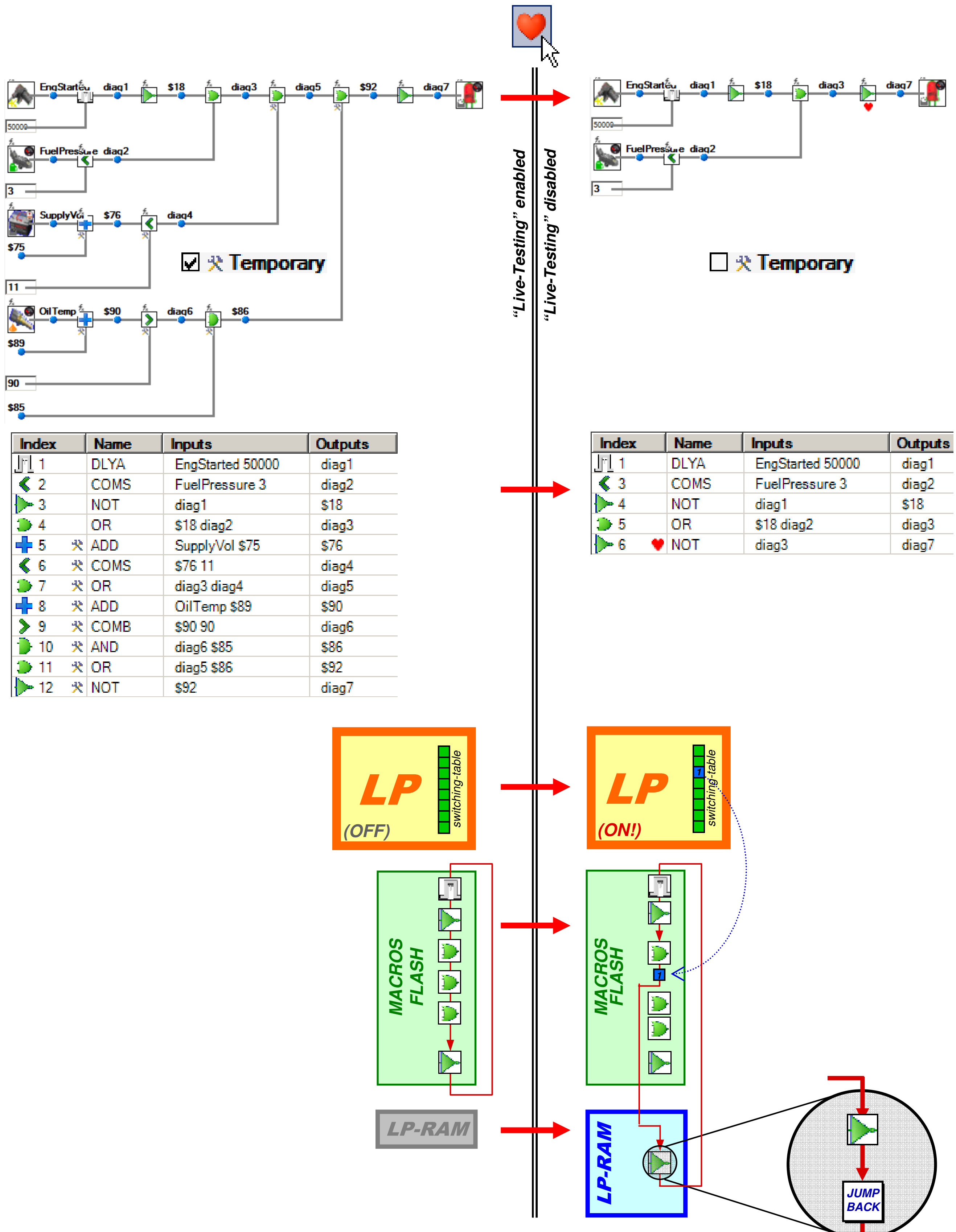
FINAL NOTE: Classical systems do allow calibrating parameters during execution, with the difference that some effort has to be accomplished to set up the corresponding tool to allow this. It is also known that execution from RAM can be different in speed, relative to FLASH where the original parameters reside.

From the statements above, ***PITFALL H#61*** (4/5 – Deferred Prototyping) and ***PITFALL H#74*** (4/6 – Intrusive Handling) are thus effectively eliminated through a straightforward and always available parameters modification mechanism.

5.1.3.3 *"LIVE-Testing"* (on-the-fly test-coding)

[Emphasis: 100% alive, 100% online, 100% while running]

This feature is most directly related to the previous *"Macros"* deletion possibility in *"Live-Programming"*. It uses exactly the same mechanism around the *"LP"* block to operate on temporary *"Macros"*. The only difference to deleting *"Macros"* is that during testing those *"Macros"* are not definitely deleted, but kept within the *"iEditor"* for repeated blending in and out from the main *"Macros-Sequence".* In other words, when the developer desires that a block of temporary *"Macros"* should be optionally included in the processing stream, then a simple modifier bit is sufficient for the *"iEditor"* to know exactly which *"Macros"* to temporarily add to the FDEF, disabling them again later on. Fig. 5.1:41 shows an example of such a mechanism at work, where a group of test *"Macros"* is marked as being "temporary", while the developer may choose to leave them unexecuted (secondary-path *"Macros"* are omitted on the *""Macros-Flash"* block for easier viewing). This feature is useful for developers to test their coding solutions without having to add the extra *"Macros"* when testing and delete them again when not testing. Theoretically there could even exist different blocks of *"Test-Macros"* while allowing the developer to enable or disable them individually.



Index	Name	Inputs	Outputs
1	DLYA	EngStarted 50000	diag1
2	COMS	FuelPressure 3	diag2
3	NOT	diag1	$18
4	OR	$18 diag2	diag3
5 ⚒	ADD	SupplyVol $75	$76
6 ⚒	COMS	$76 11	diag4
7 ⚒	OR	diag3 diag4	diag5
8 ⚒	ADD	OilTemp $89	$90
9 ⚒	COMB	$90 90	diag6
10 ⚒	AND	diag6 $85	$86
11 ⚒	OR	diag5 $86	$92
12 ⚒	NOT	$92	diag7

Index	Name	Inputs	Outputs
1	DLYA	EngStarted 50000	diag1
3	COMS	FuelPressure 3	diag2
4	NOT	diag1	$18
5	OR	$18 diag2	diag3
6 ♥	NOT	diag3	diag7

Fig. 5.1:41 – Simplified view of the "Live-Testing" example with the internal hardware actions

Besides these special *"Test-Macros"*, normal testing in the form of pre-build "test-cases" is also highly simplified through the next *"Live-Simulation"* mechanism explained right below. Since this mechanism allows for total "mixed-mode" or "hybrid" simulations, having the ideally suited test-case for each desired situation, could not be more complete and thorough. It makes it even possible, for example, to replace certain critical and difficult-to-get peripheral values or sequences of values, through simulated peripherals and therefore greately simplify testing both in time and money domains.

FINAL NOTE: Classical systems normally do not allow for this kind of code-blocks testing at all, or require some effort to set up the corresponding mechanisms to do so.

From the statements above, **PITFALL H#61** (5/5 – Deferred Prototyping), **PITFALL H#66**(Testing & Validating) and **PITFALL H#74**(5/6 – Intrusive Handling) are thus effectively eliminated through a straightforward and always available temporary code/data testing mechanism.

5.1.3.4 *"LIVE-Simulation"* made easy

[Emphasis: as easy and transparent as switching a rail-road track]

Simulation of complete systems such as a complete race-car's engine/chassis management software and corresponding hardware, is something that is not easy at all and sometimes fails already at the simplest level. The reason for this is that these systems usually rely on very complex hardware from the start, so that simulating this hardware to allow simulating the software that runs on this hardware, may be an almost impossible task from the time, personnel and financial points of view. Also, there is the intrinsic difficulty of finding the best interface point or layer where one should make the cut between the editing layers and the lower simulating layers. The main problem is to make this interface point where it would present the least interfacing challenges, when it comes to switch between the real hardware and the simulator component. This thesis' work radically changes this scenario, by where the *"Live-Simulation"* feature means to be able to simulate any component of the entire system, from the FDEFs, passing through the processing hardware and the peripherals themselves.

The simulator component interfaces to the *"iEditor"*/hardware path exactly at the narrowest communications point there is in the entire development system. This point is the tiny pipe-hole that connects lowest layer of the *"iEditor"* to the hardware itself. By having such a pin-pointed interface, allowing other types of low-level components connecting to it, other than the original hardware itself, turns out to be very easy. Another advantage is to make the *"iEditor"* never knowing whether it is communicating with real hardware or a simulator, simply because looking onto the interfacing point (pipe-hole) there is no way of distinguishing among them. In other words, a similar concept as the renowned *"Turing Test"* [297] is used, where the same idea of reducing the very connecting-point to a very simple and pinpointed channel is used. In this *"Turing Test"* a very simple terminal was used for the human operator not being able to tell if it is a machine or another human operator that responds behind that tiny interface. Fig. 5.1:42 illustrates this concept at its core, which makes connecting to a simulator component as easy as connecting to the real hardware. Note again that this mechanism does not call for any new structures, new commands or any other new low-level layers. The concrete "pipe-hole" used was one single function that is responsible for sending/receiving undifferentiated *Bytes* to and from hardware, so that the switching mechanism was really placed at the bottom of the *"iEditor"*.

To activate simulation mode, the developer only has to choose it from one of the main menus of the *"iEditor"*. Since there is no visual difference between normal hardware mode and simulation mode, there is really nothing more to say about the usage of this feature.

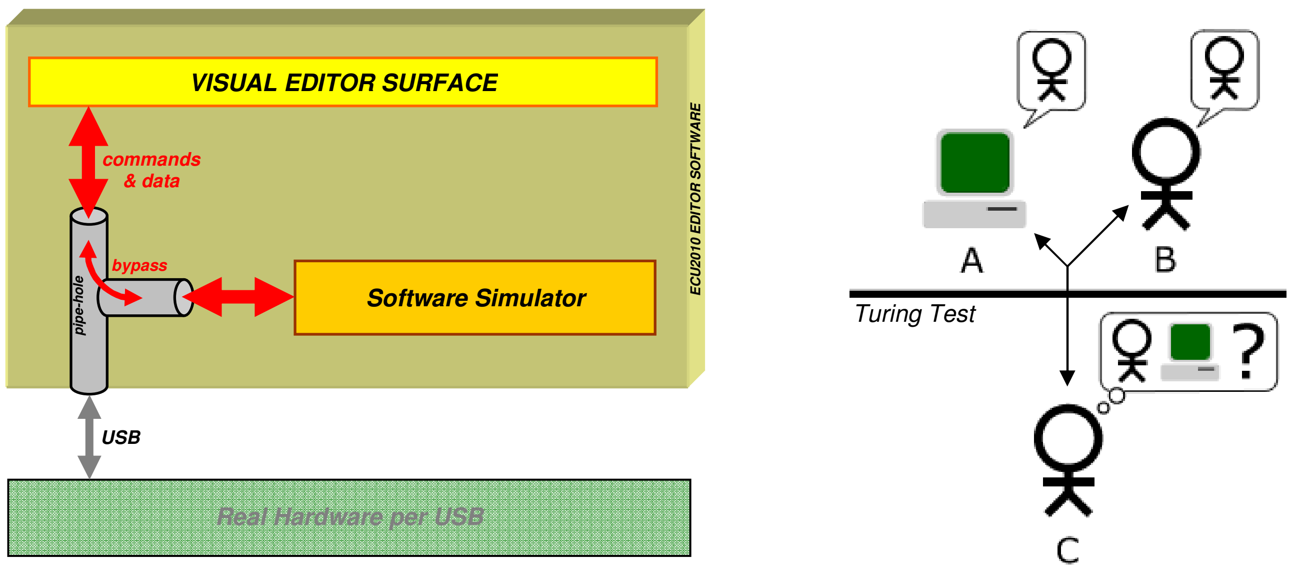


Fig. 5.1:42 – Simplified view of "pipe-hole" bypass to the "Live-Simulating" mechanism (Turing Test comparison)

Besides simulating *"iEditor"* features, this simulator software component literally simulates most of the original hardware counterpart architecture in Fig. 5.1:1. Being one of the last tasks in this thesis' work, this simulator was able to simulate the following list of entities:

- HW: Configuration Manager
- HW: "FDEF-Processor"
- HW: Inter-Module Communications and more than one Module
- SW: "Live-Debugging" ("Live-Monitoring/-Generating/-Stepping")
- SW: "Live-Prototyping" ("Live-Programming/-Calibrating")

National Instruments' LabVIEW now allows to emulate an entire FPGA for testing purposes, before initiating the long compilation and synthesising process [97]. XILINX [321] allows basically the same thing, together with ALTERA. Matlab Simulink [5] also allows for time-consuming "test-beds" and "partitioning" for simulation and bypassing. The work necessary to accomplish all that under those hoods is sometimes enormous and hardly compares to the simulator implemented herein, that was already working in the time span of a few days only. Again, the difference between those systems and the one herein, is that this one accomodates simulation possibilities intrinsically. Nevertheless, any simulator allows the developer to implement and test his system virtually, long before the hardware is even available for real lab and field tests. This allows to save time by detecting fatal bugs early on. By simulating peripherals, as explained below, this testing work can be further extended. Having a single and seamlessly integrated *"iEditor"* to do all this, as if the real hardware was connected, instead of separate tools and handling mechanisms, further reduces the time needed to learn and use such transparent simulators.

Simulation of the Peripherals Manager and the Peripherals themselves was done by using *"Live-Generation"* features for sensors and *"Live-Monitoring"* for actuators, respectively, thereby creating "virtual peripherals". In the case of sensors simulated peripheral values are directly injected into the input *"Nodes"*, whereas in the case of actuators the values of output *"Nodes"* are watched. Fig. 5.1:43 shows an example of how a simple button could be simulated through *"Live-Generation"* and how a simple light could be showed through *"Live-Monitoring"*. To simulate complex sensors such as oxygen-sensors or engine-speed sensors, models would be needed, which are not included in this thesis' work. Nevertheless, it would be straightforward to integrate a software component that would indeed allow the developer to use recorded real-world or simulated data from sensors to feed the FDEFs. For actuators such as igniters and injectors, the most useful thing would be something like the object shown in Fig. 5.1:30 but used as mere abstract feedback for the developer to quickly recognize any problems, and not as a strict simulated actuator.

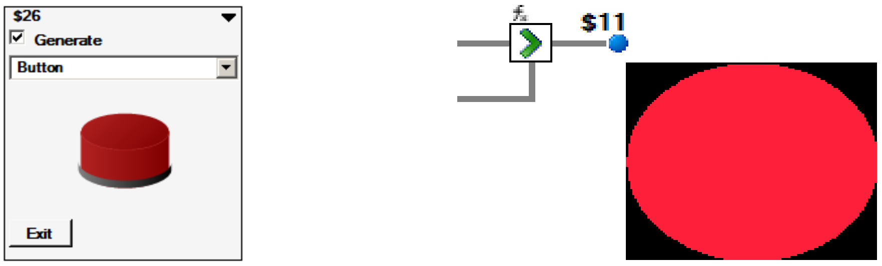


Fig. 5.1:43 – Simulated sensor peripheral (button) and actuator peripheral (light)

The advantages of having such "virtual prototyping" systems is well addressed in [516]. Although this possibility is really of great impact when designing and developing complex automotive systems, this simulator has been implemented only partially in the work herein, mainly only to demonstrate its easy feasibility. One highlight is even the intrinsic possibility of the *"iEditor"* being able to simulate some parts and use real hardware for others. For example, it is easily possibly to use some real peripheral *"Nodes"* data to feed simulated FDEFs and output the results to more real peripherals. Of course, real-time behaviour would not be guaranteed, since simulation runs much slower than the real *"FDEF-Processor"* in hardware, but it is usable in principle, for simple non-realtime tests and experiments. Completely mixed hardware/simulation combinations are throughout possible, in that real and simulated peripherals may co-exist with hardware-processed and simulated FDEFs as well. All this was tested and worked, but was not further pursuit due to lack of time. Even so, the importance of this special mixed simulation scenario assumes great impact regarding testing possibilities as mentioned in the previous section above.

As seen for the previous *"Live-Programming"* feature, this simulation possibility also took a very short time span to pass from a "software prototype" stage to a final integrated feature: only 3 man-days to demonstrate it successfully, plus 3 man-weeks to finish it off. This, again, attests to the relative easy with which features were added to this system globally.

FINAL NOTE: Classical systems rarely provide easy-to-use simulated code, or they are often not integrated at all into the development environment, requiring some effort to set up the appropriate mechanisms. Classical systems rarely provide simulated peripherals, or they are often not integrated at all into the development environment, requiring some effort to set up the appropriate mechanisms.

From the statements above, ***PITFALL H#63***(1/2 - Mixed Simulations) is thus effectively eliminated through the fact of the simulator having its work made easy by the fact that all it has to do is to simulate the isolated *"Macros"* and some external mechanisms, without having to simulate a complete and complex CPU with all its side-effects. From the statements above, ***PITFALL H#63***(1/2 - Mixed Simulations) is thus also effectively eliminated through the fact that simple models of the peripherals may be directly fed into the *"Live-Generation"* mechanism.

5.1.3.5 *"LIVE-Processing"*

[Emphasis: fully natural/intrinsic parallel processing]

This point refers to the global nature of the parallel/distributed processing found in the work implemented and can be seen in [Att. 44].

5.1.4 “LIVE-Peripherals”

[Emphasis: smart/alive autonomous peripherals]

"Live-Peripherals" refer to the *"Smart Peripherals"*, where each peripheral is a complete *"Macro"* processing unit with sensing or acting capabilities and all the "LIVE" features of prototyping and debugging as well, thus also "live" entities. These *"Smart Peripherals"* are the I/O processing cells of the developed "MULTI-SYSTEM".

Each one of those *"Smart Peripherals"* includes and uses the necessary resources (timers, interrupts, etc.) as needed and totally independent of each-other. The only interaction among modules will be done through the high-level *"Nodes"* exchange through the "inter-communications network". In one expression, this "MULTI-SYSTEM" architecture sorts out the hardware spot where all comes together in a big pile and separates the paths by separating the hardware itself (see Fig. 4.5:77). Furthermore, no central control or master-entity is needed. Finally, each *"Smart-Peripheral"* can be seen as a completely independent processing entity by itself, similar to the *"Cellular-ECU"* concept already used for the main processing modules themselves.

The digital-bus connecting *"Smart Peripherals"* to the main processing modules (see Fig. 4.5:77), is part of the global inter-connection network (in red) that conveys all the *"Nodes"* needed between all processing sites in the *"Cellular ECU"* MULTI_SYSTEM architecture, as illustrated in Fig. 5.1:44. More interesting details can be seen in [Att. 45].

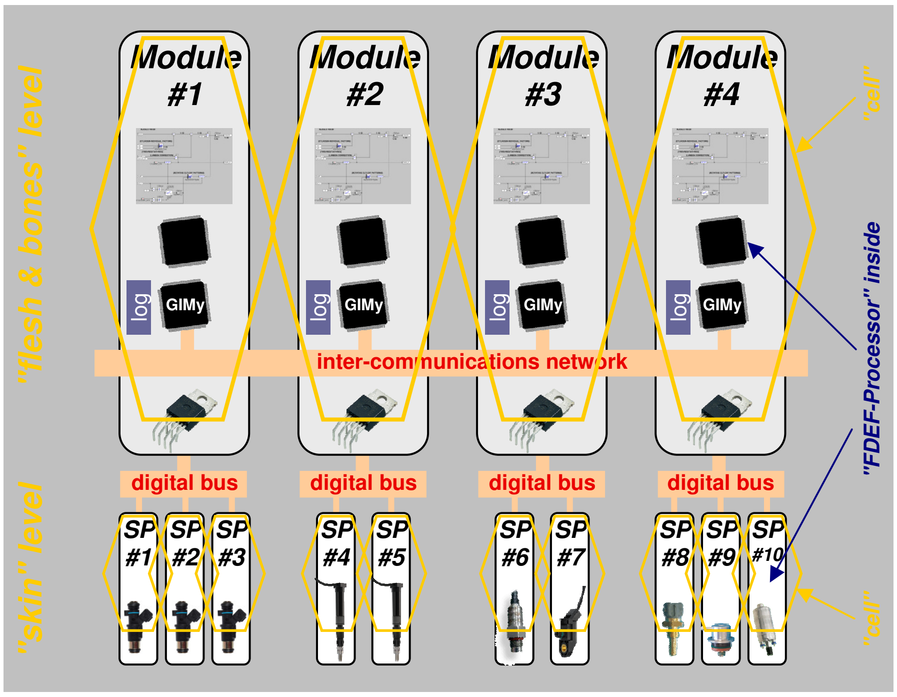


Fig. 5.1:44 – Global "Cellular" architecture with 14 processing "cells" and the "Smart Peripherals" as the I/O cells

5.1.5 “LIVE-Logging”

[Emphasis: integrated self-contained data-logging in each processing "cell"]

"Live-Logging" refers to the capability of each processing "cell" (as depicted in the previous Fig. 5.1:44 with the side-block "LOG") being able to log its own *"Nodes"*. Thus, each "cell" contains hardware taking chosen *"Nodes"* out of the *"GIMy"* and storing them in internal data-logging memory. This is done without any intervention nor knowledge from

the *"FDEF-Processor"* and runs in parallel to it. The *"Nodes"* to be logged must be marked with the corresponding *"Node Modifier"* ( ) inside the FDEF editor. Fig. 5.1:45 illustrates the paths involved in this internal mechanism: the *"Nodes"* to be logged are selected inside the "Log Manager" through the "logged *Nodes* flags" and are automatically and parallel retrieved from the *"GIMy"*, stored into local FLASH inside the "Log Manager", to be later retrieved through a high-speed USB link to the external PC/laptop. More details in [Att. 46].

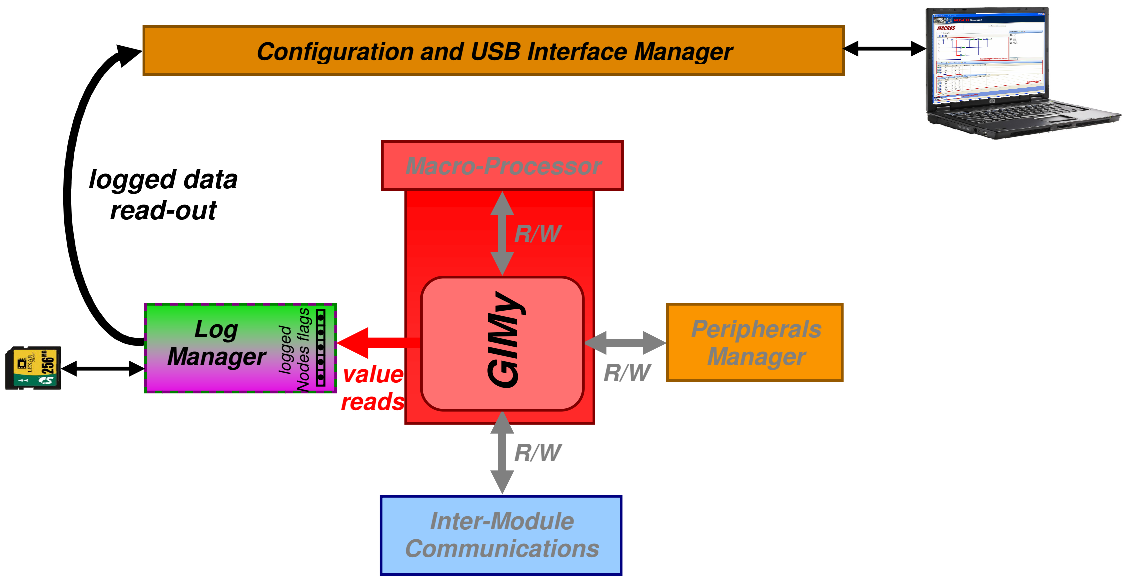


Fig. 5.1:45 – Normal internal "Live-Logging" mechanism relative to the "GIMy"

5.1.6 "LIVE-Teleoperation" made easy

[Emphasis: as easy and transparent as being on-location]

Remotely handling or operating a complex system is something that may be of highest interest for many applications, such as Motorsports. In this thesis' work, *"Live-Teleoperation"* means to be able to operate any component of the entire remote system, including all types of advanced operations like *"Live-Debugging"* and *"Live-Prototyping"*, also manipulating peripherals and everything that may be done with the original hardware locally. This would be done through TCP/IP, which enables remote control through the internet from any point in the world.

The transparent possibility of controlling a remote system through the internet again relies on the fact that the internet bridge is going to be build onto the narrowest communications point there is in the entire *"iEditor"*, namely the same tiny "pipe-hole" that was used for the simulator component. As for the simulation, the *"iEditor"* will not be able to distinguish between real local hardware and a remote system Again, this is of utter importance in respect to keeping the *"iEditor"* fully independent of what goes on beyond its scope.

Fig. 5.1:46 and Fig. 5.1:47 illustrate this concept of easy remote control in two steps: the local client-side and the remote gateway-side. While the client-side (where the developer resides) is represented by a fully functional *"iEditor"*, the gateway-side (where the customer hardware resides) is just a deactivated *"iEditor"* that transparently bridges the incoming TCP/IP data into USB to the hardware, and vice-versa. With these internal mechanisms, connecting to a remote hardware site is just as easy as connecting to local hardware. Note also that again this mechanism does not call for any new structures or new

commands. Also as in the simulation part, the concrete "pipe-hole" used is the same single function that is responsible for sending and receiving undifferentiated *Bytes* to and from the hardware, at the lowest point of the *"iEditor"*. To activate client or gateway modes, the developer and the remote person only have to choose them from one of the main menus of the *"iEditor"*, wait for the internet connection to engage.

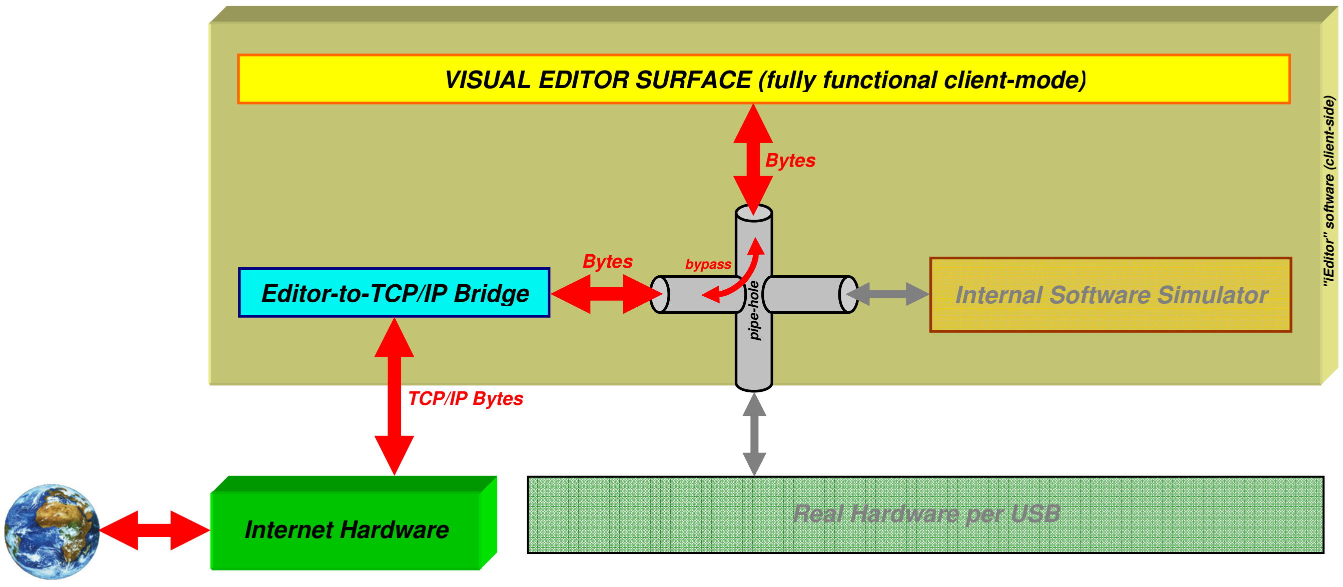

VISUAL EDITOR SURFACE (fully functional client-mode)
Bytes
Editor-to-TCP/IP Bridge
Bytes
bypass
pipe-hole
Internal Software Simulator
"iEditor" software (client-side)
TCP/IP Bytes
Internet Hardware
Real Hardware per USB

Fig. 5.1:46 – View of the "pipe-hole" bypass for the "Tele-Operation" mechanism (local client-side)

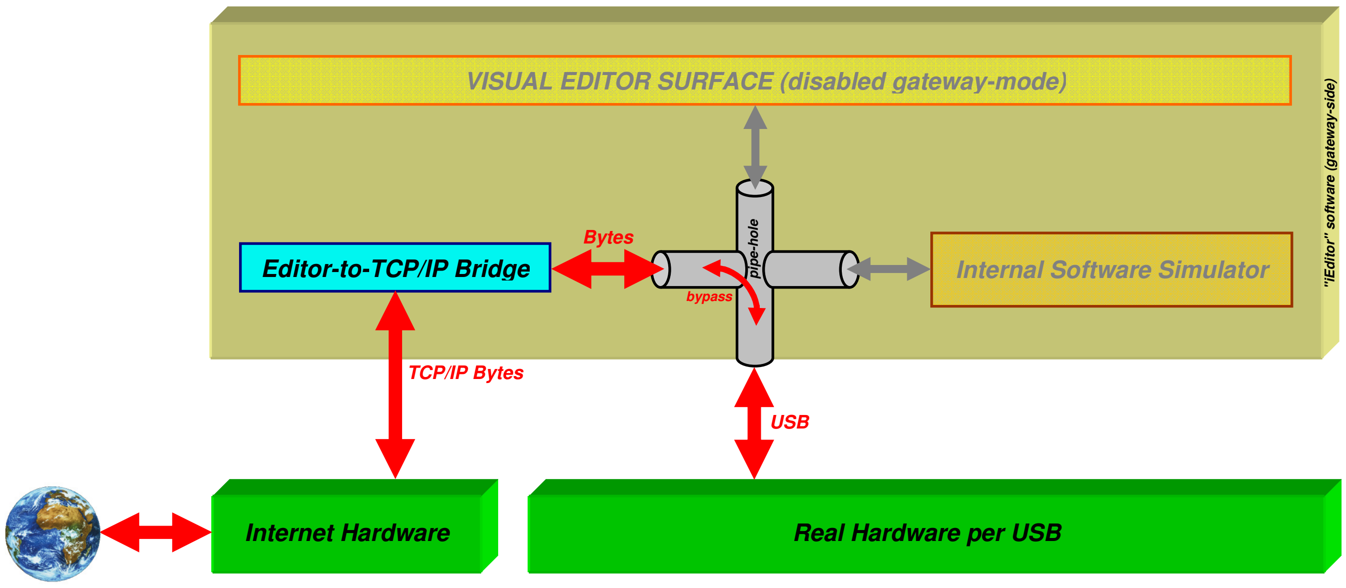

VISUAL EDITOR SURFACE (disabled gateway-mode)
Editor-to-TCP/IP Bridge
Bytes
pipe-hole
bypass
Internal Software Simulator
"iEditor" software (gateway-side)
TCP/IP Bytes
USB
Internet Hardware
Real Hardware per USB

Fig. 5.1:47 – View of the "pipe-hole" bypass for the "Tele-Operation" mechanism (remote hardware gateway-side)

This *"Live-Teleoperation"* feature was originally implemented just to demonstrate the relative easy with which it would be possible to add onto the already existing system Nevertheless, this feature turned out to be essentially indispensable for the final system demonstration at Bosch in Germany, where it worked perfectly over 2.300Km of Internet with a running engine in Portugal. More details in [Att. 47].

As for the previous *"Live-Programming"* and *"Live-Simulation"* features, this remote operation possibility took an extremely short time span to get done: 3 man-days for first successful remote operation demonstration, plus 1 man-week thereafter.

5.1.6.1 Remote Handling potentials

[Emphasis: all the advantages of being on-location]

This *"Live-TeleOperation"* feature allows imagining a whole new way of doing things and of accomplishing some usually difficult to tasks, such as troubleshooting without having to travel thousands of kilometres just to get physically near the target system. Here is a list of most of the immediate benefits of having this feature enabled:

- Remote problem troubleshooting without having to travel long distances
- Remote system diagnosis without having to analyze sent reports and log-files
- Remote operation of dangerous systems enclosed in special lab facilities
- Remote assistance and/or system training without having to be physically present
- General remote access to the system from virtually anywhere!
- Cost savings in all cases where physical presence would be normally necessary

Along with these capabilities, this feature does not need to know which internet technology is used between client and gateway, so that virtually anything may be used, such as LAN/WAN, GPRS/UMTS, and even Bluetooth and WLAN on portable devices such as PDAs, laptops and tablet-PCs onto which the *"iEditor"* may also eventually be installed. The only thing that really matters is speed, where the higher the internet speed, generally the better the feedback that is obtained on the *"iEditor"* itself. Especially on operations such as *"Live-Debugging"* with oscilloscopes and *"Live-Calibration"* of large maps, a higher internet speed makes a most notorious difference. Either way the *"iEditor"* and the hardware never notice that everything is being played through the internet at any distance.

FINAL NOTE: Classical systems usually do not provide any means of integrated tele-operation or anything that relates to controlling the system from the distance. Normally, other means have to be used instead.
From the statements above, ***PITFALL H#69*** (Tele-Operation) and ***PITFALL H#74*** (6/6 – Intrusive Handling) are thus effectively eliminated though the use of a very simple pinpointed interfacing method inside the *"iEditor"*.

5.1.6.2 *"LIVE-Support"* (on-the-fly remote assistance)

[Emphasis: online and real-time communications with remote persons]

The same *"Tele-Operation"* mechanism can readily be used to communicate with the remote persons through a simple text interface. Further enhancements even could add voice and image/video to this extra remote assistance mechanism. This text-based interface also resides inside the *"iEditor"*, whereas the remote *"iEditor"* block in Fig. 5.1:47 would then be enabled for text-conversations.

FINAL NOTE: Classical systems usually do not provide any means of integrated remote support or anything that relates to troubleshooting the system from the distance. Normally, other means have to be used instead.
From the statements above, ***PITFALL H#70*** (Remote Support) thus effectively eliminated through the seamless integration of a remote support mechanism into the *"IEditor"* itself.

5.1.7 "LIVE-Handling" Revisited

[Emphasis: orthogonality to the extreme of allowing every possibility in every situation]

Orthogonality (independence) between system components, be it SW or HW, allows for all possible operations to be available in virtually every situation, be it Design, Simulation, local or remote Operation. More details and pitfalls solved, in [Att. 48].

5.2 *"Cellular ECU"* Omnipresent Parallel Processing Concept

[Emphasis: implementation of the real hardware modules]

Taking the heavily simplified hardware internals of any standard microprocessor into account, one only needs the architecture depicted in Fig. 4.5:78 to be able to process, in this case the *"Macros-Sequence"* or, more abstractly speaking, the "FDEFs". Since the beginning of this work, there was the desire to create a hardware architecture that would simultaneously comply with all of the following premises to successfully accomplish the *"ECU2010"* project and its desired results:

- Increased integration of needed processing resources into a single spot
- Reduced centralization and more of a distributed architecture
- Reduced conceptual architectural complexity in terms of different components
- Reduced inter-dependencies between those components
- Reduced critical system consequences of individual component failures
- Increased easiness in achieving resources/capacity scalability and processing parallelism
- Increased easiness in handling the broad variability in automotive peripherals
- Reduced needed software complexity to comply with the hardware's needs for control
- Reduced complexity of any extra motorsport-specific functionality such as datalogging
- Reduced available manpower, budget & time (6 Engineers, about 350.000€ and about 2 years)

For more details about this *"Cellular-ECU"* architecture, all related sub-topics, as well as thorough explanations about how this architecture perfectly fits the entire *"Macros"* and features of the global *"ECU2010"* picture, please refer to [Att. 49].

5.2.1 Internal Architecture

[Emphasis: the heart of the hardware modules]

The heart of it all is the self-contained *"FDEF-Processor"* with its *"Macros-Processor"* in the centre, again conveniently depicted here in Fig. 5.2:48. Further down, the more complete Fig. 5.2:49 then additionally details all the important extra internal blocks and paths inside the global *"Cellular-ECU"* module, comprising the following elements (including the mentioned *"FDEF-Processor"*):

1 • ***Configuration Manager:*** this block interfaces directly to the USB communications electronics (1), which in turn connect to the external Windows PC or laptop. All commands and pieces of data traverse this block, which routes them to/from the other internal processing/management blocks. This block thus also manages all the *"Macros"* loading procedures into FLASH and RAM, as well as all of the needed "routing/enabling-tables" for the various blocks explained later. It also manages all the uploading of monitoring data such as the *"Nodes"* themselves. This block is the so-called general omnidirectional distributor of *"Macros"* (2), routing-tables (4) (5) (6) and commands (2*) (3) (4) (5). One very important connection is (4), since it is this one that will allow for the ubiquity of the extension of the modular *"Cellular-ECU"* concept to the peripherals themselves (explained later on, when gatewaying information coming from the *"iEditor"* to those peripherals). There is one extra connection (1) that allows direct MMC FLASH logging-data read-out by the "Configuration Manager", to maximize data throughput per USB. In reality, the (3) connection is indirectly made by the "Log Manager", since it closely relates to the *"Live-Debugging"* block inside it, but was explicitly put here for better understanding.

2 • ***"FDEF-Processor":*** the heart of it all, where the *"Macros"* are processed and around which everything else crystallizes in this entire system. This block receives the *"Macros"* from external FLASH (2) and RAM (2*), through the "Instruction Access Manager" (2), which also manages all debugging code stepping and *"LIVE-Prototyping"* features. Its integrated *"switching-table"* allows the *"Macro"* fetcher to distinguish between normal FLASH (2*) and added RAM (2**) *"Live-Prototyping"* sequences. The needed *"Nodes"* are directly fetched/stored from/into the *"GIMy"* (2). These *"GIMy"* operations happen independently and concurrently to all the other three. The *"Macros-Processor"* is totally isolated from everything else other than the "Instruction Access Manager" itself. This was a central issue in this thesis' work: the highest possible independence and encapsulation from the outside world, regarding *"Macros"* execution.

3 • ***"GIMy":*** this is the central repository of all *"Nodes"* inside a *"Cellular-ECU"* module. All surrounding blocks needing to write or read *"Nodes"* directly access this central block (2) (4) (5) (6), while giving no special priority to any of those blocks. This block features "multi-read / single-write" capabilities with fully automatic local write-collision management. The integrated *"enable-table"* allows it to fixate certain *"Nodes"* specifically for the *"Live-Generation"* features. All four-sided *"GIMy"* accesses are independent and concurrent relative to each other.

4 • ***Peripherals Manager:*** this block controls all actions upon the externally attached peripherals (4), through its contained "Peripheral Controller", which includes the externally needed analog electronics. This block manages all *"Node"* transfers to/from the peripherals and also allows the firmware update of those peripherals. The integrated *"routing-table"* contains all the *"Nodes"* that must be transferred from the *"GIMy"* (4) to the peripherals (especially for actuators) and those that must be transferred from those peripherals (especially sensors) to the *"GIMy"* (4). These *"GIMy"* operations happen independently and concurrently to all the other three.

5 • ***Log Manager:*** this block is responsible for three tasks: telemetry, local datalogging and *"Node"* monitoring. The "RF Controller" manages the wireless connection (5), whereas the "MMC Controller" manages the MMC FLASH storage (5*). The integrated *"routing-table"* contains all the *"Nodes"* that must be logged, being either sent through telemetry to an external device such a wireless datalogger or wireless dongle attached to a laptop, or sent to the MMC FLASH storage for later extraction. This table also contains the *"Nodes"* that are directly sent to the external Windows PC for display purposes (monitoring features included in *"Live-Debugging"*). These *"GIMy"* operations (5) happen independently and concurrently to all the other three.

6 • ***Inter-Module Communications-Manager:*** this block is the key entity inter-connecting to the left- and right-sided modules, exchanging *"Nodes"* contained in its integrated *"routing-table"*. These *"Nodes"* are fetched from the central *"GIMy"* (6) and sent to the other modules, while storing the similarly received *"Nodes"* from the other modules into the *"GIMy"* (6). These *"GIMy"* operations happen independently and concurrently to all the other three. Two lateral serial connections communicate with the immediately neighbouring *"Cellular-ECU"* modules (6).

7 • ***Real-time Clock:*** this clocking reference is mainly needed for the *"FDEF-Processor"* (7) for it to be able to process timing-related *"Macros"* such as delays, timers, filters and integrators. It is also used for time-stamping (7*) logged and wirelessly sent *"Nodes"*, as well as for time-stamping *"Nodes"* inside the *"GIMy"* if needed (7**). This internal RTC runs based on an RC-oscillator to avoid the quartz-crystal sensitivity to motorsports-typical high-frequency vibrations, which might break it. An external calibration clocking signal (7) based on a common protected quartz for all modules, then helps to fine-tune this internal RTC. Each *"Cellular-ECU"* module will have its own independent local RTC, without ant further inter-module synchronization whatsoever.

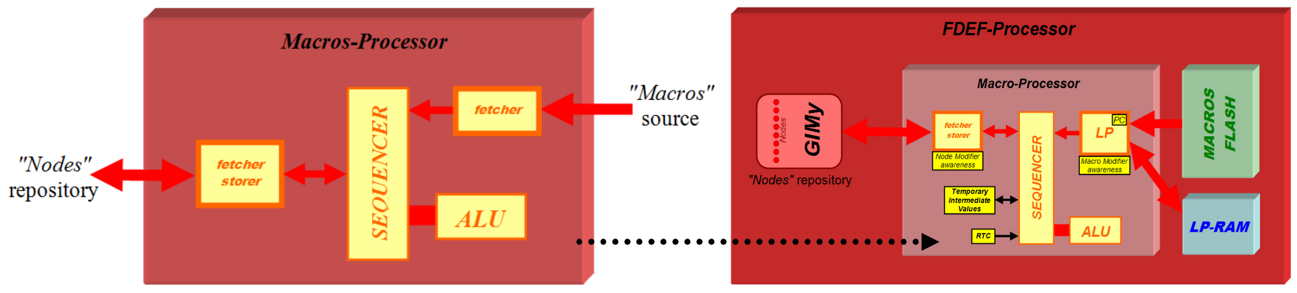


Fig. 5.2:48 – Left: basic simplified "Macros-Processor", right: complete full-fledged heart, the "FDEF-Processor"

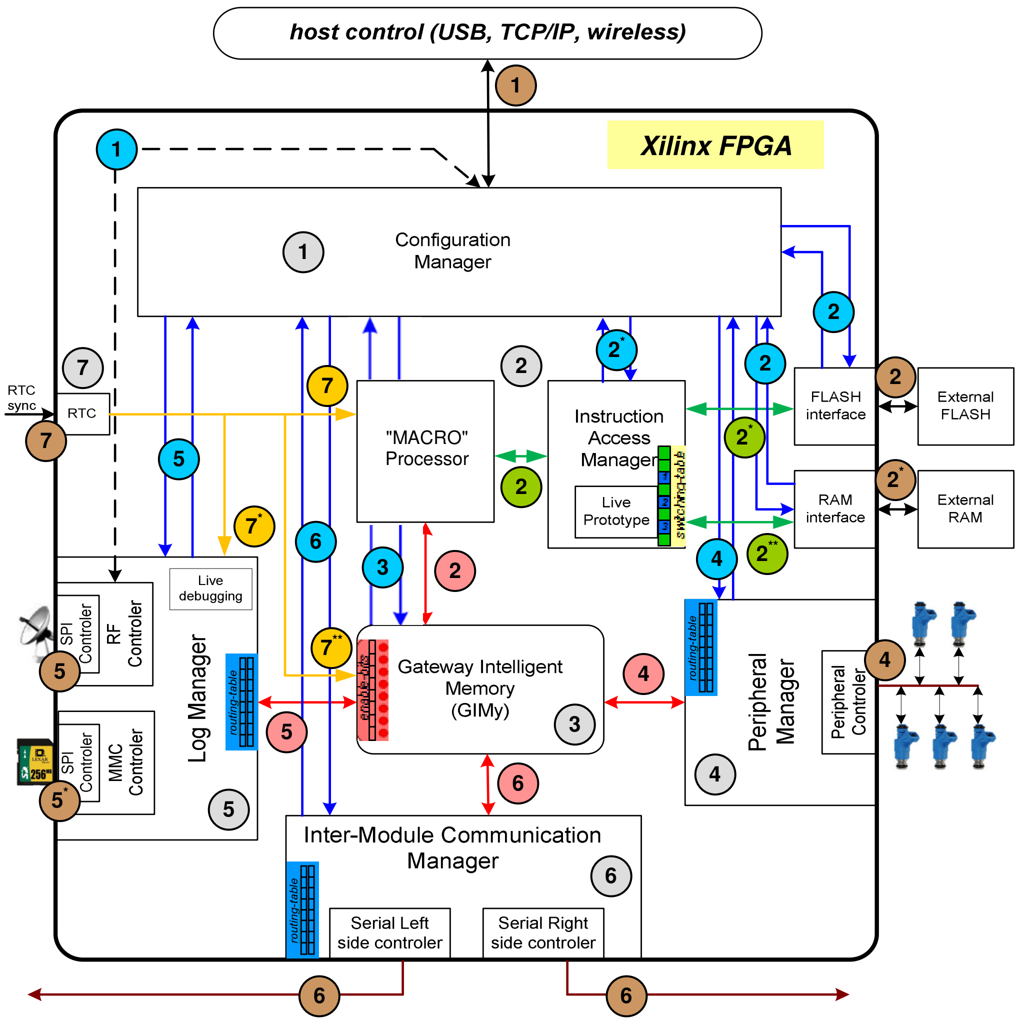


Fig. 5.2:49 – A more detailed view of the internal blocks and paths inside the "Cellular-ECU" inside a single FPGA

Following this architecture, it is clear that a single *"Cellular-ECU"* possesses all the functionality integrated into itself, needed for a full-fledged automotive ECU for use in either series-cars or motorsport vehicles alike. Keeping things highly self-contained and highly homogeneous was one of the keys to success in this thesis' work. Functionalities of a *"Cellular-ECU"* unit include:

- Automotive program processing (with code and data storage)
- Real-time awareness
- Sensors inputting
- Actuators outputting
- Communications with the outside
- Debugging/monitoring capabilities
- Firmware update capability
- Datalogging (essential for motorsport applications)
- Telemetry (essential for motorsport applications)
- (communications with neighbouring modules is described in the next section)

This reality is also where the name *"Cellular-ECU"* originally came from, that is, a full-featured ECU "cell" that handles all kinds of needed tasks that would be handled by an ordinary ECU and that shares its self-contained existence among many other "cells" inside the same automotive system. Additionally, this architectural hardware "cell" will be used for absolutely all entities needed in a complete automotive system. Thus, in a spatial analogy, similarly as in modern cellular mobile communications, these modules also represent the "cells" that ultimately cover all the "area" of an automotive system. This concept makes those modules "omnipresent", as stated in the title of this main section:

- Main ECU (the heart of the system, processing the main high-level FDEFs, internal datalogging)
- Peripherals (for inputting and outputting sensor and actuator data, respectively)
- Datalogger (external and wireless, for extra flexibility during racing)
- Telemetry (for extra flexibility during racing)
- Dongles (wired or wireless, for special testing/calibration purposes)
- Hardware-in-the-loop (just an example of another application of this highly parallel hardware)

As an introductory note for the sections that follows and explain all the other internal components, it should be clear that the same way FDEFs only exchange values between operations and nothing more, this *"Cellular-ECU"* architecture also shares this principle of only exchanging *"Nodes"* between components, modules and peripherals.

5.2.1.1 "Configuration Manager" component

[Emphasis: the control of the heart]

This block controls all the information-flow between the *"Cellular-ECU"* module and the outside. It is thus responsible for the routing of messages inside the module. It allows communication between the different blocks and the communication protocol used to communicate with the outside world. Each module uses USB to communicate with a computer running the *"iEditor"*. There is also extra "incoming" connection from the "Flash Interface" that allows the loading of the initial configuration of each internal block after a module reset without any external computer attached to it through USB. Fig. 5.2:50 shows the internals of this component, also detailing the signals implemented just for understanding the minimal complexity required. FIFO data structures handle all the incoming and outgoing data/commands, whereas the "Incoming" and "OutGoing" blocks represent simple dumb routing mechanisms, delivering the information always to the desired paths, according to the message fields containing the corresponding addressing bits. Most of the information circulating here is data, whereas some commands also issued by the *"iEditor"* start all the data transactions, as well as setting all the necessary states of the corresponding components receiving/transmitting that data.

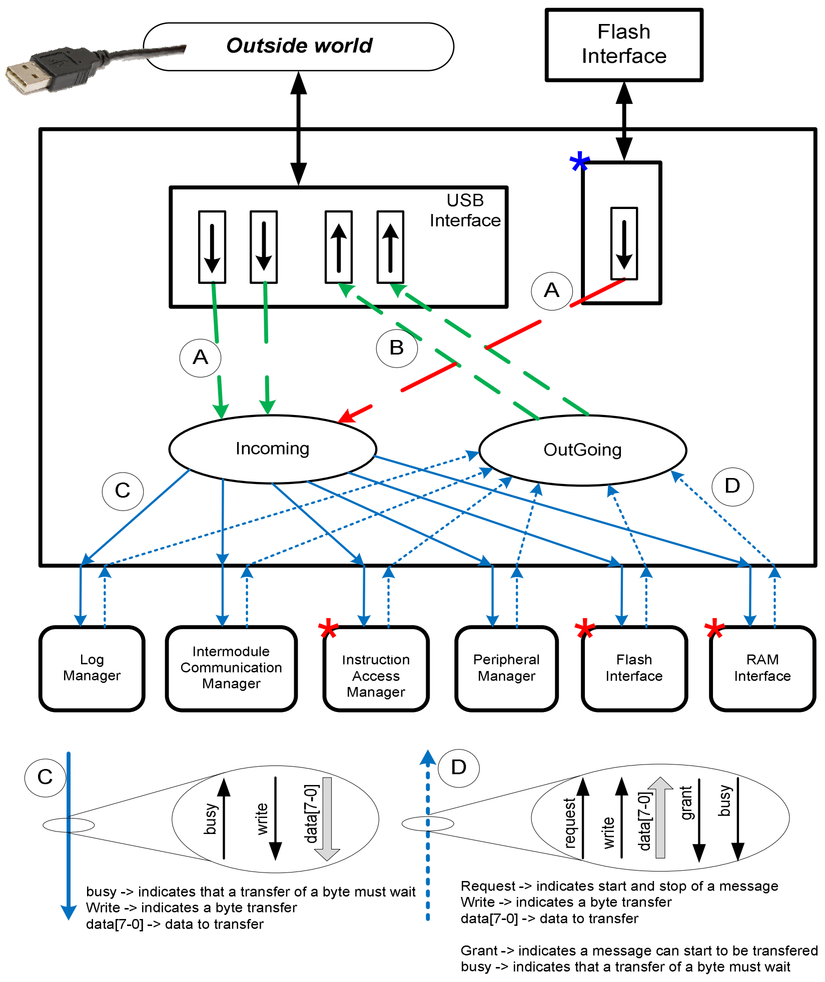

Outside world
Flash Interface
USB Interface
A
B
C
D
Incoming
OutGoing
Log Manager
Intermodule Communication Manager
Instruction Access Manager
Peripheral Manager
Flash Interface
RAM Interface
busy
write
data[7-0]
busy -> indicates that a transfer of a byte must wait
Write -> indicates a byte transfer
data[7-0] -> data to transfer
request
grant
Request -> indicates start and stop of a message
Write -> indicates a byte transfer
data[7-0] -> data to transfer
Grant -> indicates a message can start to be transfered
busy -> indicates that a transfer of a byte must wait

Fig. 5.2:50 – Close look at the detailed internals of the "Configuration Manager"

One additionally very important detail of the operation of this component is that it also allows the *"iEditor"* to store "initial configuration messages" inside FLASH memory, for immediate use upon system start or restart. These messages have the exact same format and data as if they were regularly received through the USB pathway, and are fed to the Configuration Manager's "incoming" sub-component upon initialisation, as if they had been received through USB. This special pathway represented by * takes care of mainly two situations: configuring the system upon start-up when no USB connection is present and to allow for initialised *"Nodes"* upon that same start-up (although here another mechanism was preferred, due to the potential volume of initialisations, based on the *"Log-Manager"*). The three paths in Fig. 5.2:50 above marked with * will also connect to the *"FDEF-Processor"* detailed immediately below herein.

5.2.1.2 "FDEF-Processor" component

[Emphasis: the clock ticker of the heart]

The *"FDEF-Processor"* (Fig. 5.2:51) is where all the *"Macros-Sequence"* processing and isolated *"Macro"* execution happens. The "path for direct command-line *Macros*" comes from the previous "Instruction Access Manager" USB communications path shown above in the "Configuration Manager". The "path for direct command-line *Macros*" includes START/STOP commands for *"Live-Stepping"* during *"Live-Debugging"* sessions, as well as PAUSING command during *"Macros"* download to FLASH or RAM, etc.

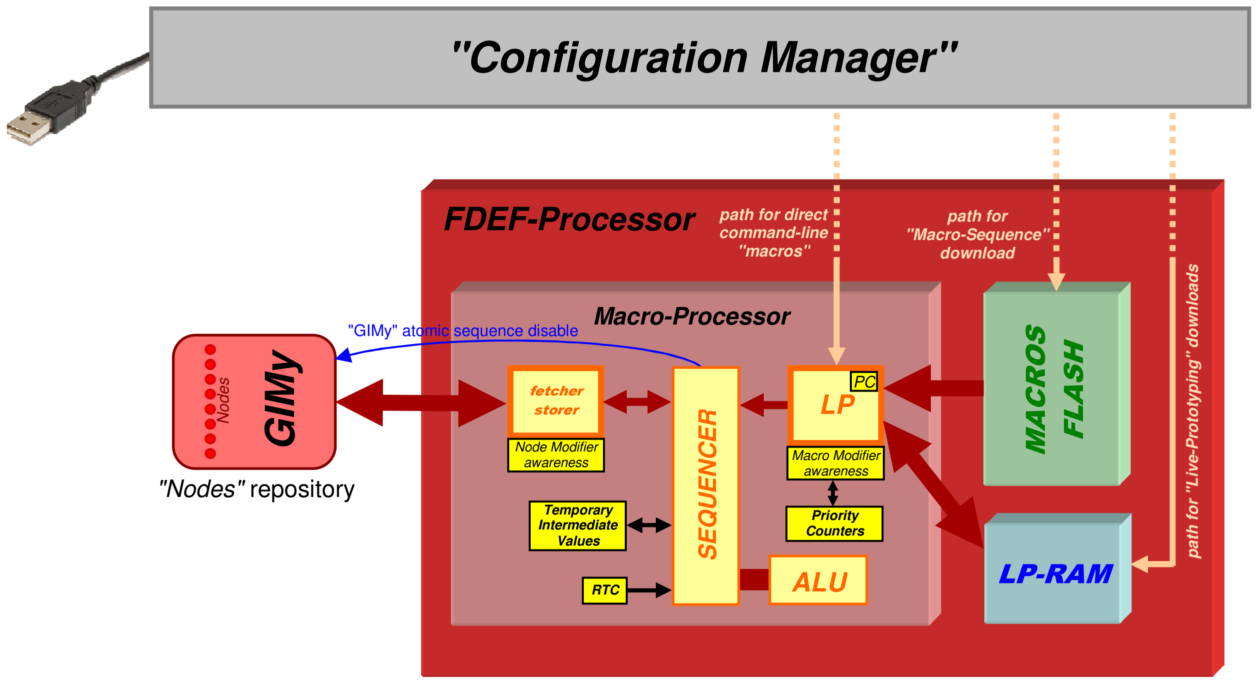


Fig. 5.2:51 – The complete "FDEF-Processor" with all its internal components and connections

Taking the *"Macro Modifiers"* detailed upon their first presentation earlier herein, into the perspective of the internal mechanisms above, these are the internal actions taken:

- **"PATH TYPE"** – does not affect *"Macro"* processing at processor level herein in any way and only influences *"Macro"* behaviour at strict visual *"iEditor"* level.
- **"PRIORITY"** – uses "priority-counters" for the *"Macros"* priority scheme already clearly illustrated in Fig. 4.5:73, allowing for different execution frequencies for each individual *"Macro"*.
- **"SECRET"** – does not affect *"Macro"* processing at processor level herein in any way and only influences *"Macro"* behaviour at strict visual *"iEditor"* level.
- **"CUSTOMIZABLE"** – does not affect *"Macro"* processing at processor level herein in any way and only influences *"Macro"* behaviour at strict visual *"iEditor"* level.
- **"STRICT SEQUENCE"** – does not affect *"Macro"* processing at processor level herein in any way and only influences *"Macro"* behaviour at strict visual *"iEditor"* level.
- **"ATOMIC SEQUENCE"** – when encountered, immediately disables any *"Node"* update through the "*GIMy* atomic sequence disable" signal in Fig. 5.2:51, continues processing *"Macros"* and re-enables *"GIMy"* updating as soon as this modifier is not used any more.
- **"BREAKPOINT"** – when encountered, pauses the *"Macro-Processor"* and waits for commands from the *"iEditor"* for the next actions to be taken during debugging. This has already clearly been illustrated in Fig. 5.1:34, allowing to place a breakpoint on any individual *"Macro"*. From here on, the *"Macro-Processor"*
- **"TEMPORARY"** – does not affect *"Macro"* processing at processor level herein in any way and only influences *"Macro"* behaviour at strict visual *"iEditor"* level.
- **"COMMENTS"** – does not affect *"Macro"* processing at processor level herein in any way and only influences *"Macro"* behaviour at strict visual *"iEditor"* level.

Now taking the *"Node Modifiers"* detailed upon their first presentation earlier herein, into the perspective of the internal mechanisms above, these are the internal actions taken. Note that the function of the *"Node Modifier awareness"* block inside the *"Macro-Processor"* is not really performed therein, but rather in the *"GIMy"* and "Log-Manager". This block has been left herein for comprehension purposes only.

- 7.63864 **“UPDATING”** – does not affect *"Node"* fetching/writing nor processing at processor level herein in any way and only influences the visual display of the *"Node"* for the user.
- **“MONITOR”** – does not affect *"Node"* fetching/writing nor processing at processor level herein in any way, only influencing *"Node"* behaviour at strict *"Log-Manager"* level for non-intrusive values monitoring purposes in *"Live-Monitoring"* within *"Live-Debugging"* features.
- **“DATALOGGING”** – does not affect *"Node"* fetching/writing nor processing at processor level herein in any way and only influences *"Node"* behaviour at strict *"Log-Manager"* level for data-logging purposes detailed later on herein.
- **“INITIALIZATION”** – does not affect *"Node"* fetching/writing nor processing at processor level herein in any way and only influences *"Node"* behaviour at system start-up.
- **“GENERATE”** – does not affect *"Node"* fetching nor processing at processor level herein in any way, but completely disables *"Node"* writing into the *"GIMy",* due to the *"Log-Manager"* having set the write-disable bit for the desired *"Node"* to be generated (see Fig. 5.1:32).

Note that the "SEQUENCER" inside the *"Macro-Processor"* is the entity that sequences all internally necessary actions to get a particular *"Macro"* executed correctly, in addition to fetching and storing *"Nodes"*. It is also this "SEQUENCER" that feeds the "ALU" with the correct values, that manages intermediate and temporary values, finally retrieving the final result to be stored into the *"GIMy"*. Thus, this "SEQUENCER" is essentially a state-machine containing all possible action sequences for all possible *"Macros"*. Fig. 5.2:52 illustrates the internal structure of this "SEQUENCER". It is based on an expansion of the basic state-machine depicted in the previous Fig. [Att. 40]:137 but now more detailed and showing the used resources inside the *"Macros-Processor"*. Green states are main states of execution and gray ones are sub-states. The only red state of "micro-reboot" essentially resets the entire *"Macro-Processor"* as if it was just started up. Again, as stated in Fig. [Att. 40]:137, this state-machine has no ramifications of unfolding, except for the priority, breakpoint, atomic and temporary value evaluation steps.

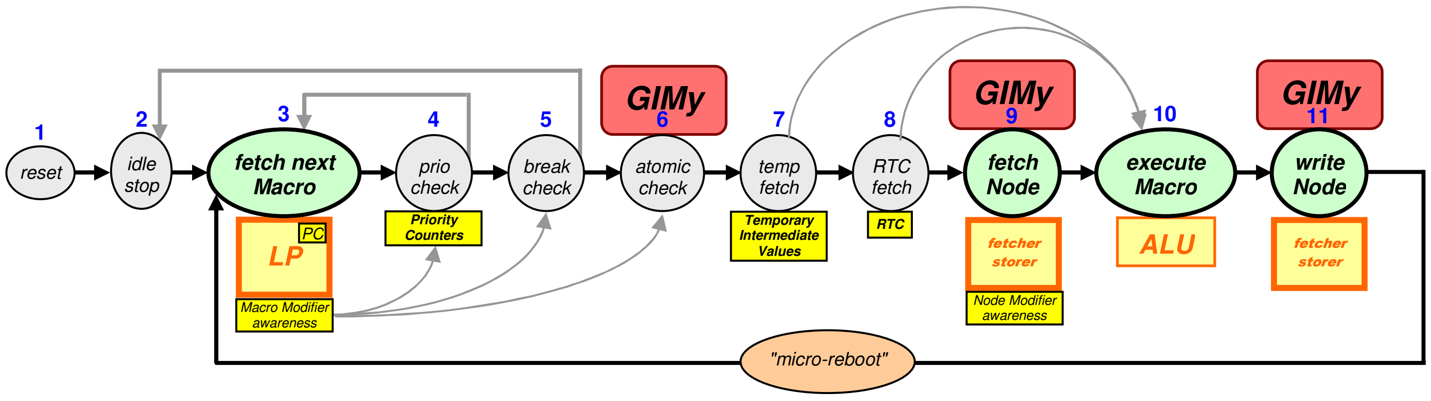


Fig. 5.2:52 – The "SEQUENCER" contains all action sequences for all possible "Macros"

- ***1*** → Reset state entered upon start-up of the entire system; stays here until the entire system has been brought to life and the first "Macro" may be executed
- ***2*** → Idle state entered after reset and prior to next *"Macro"* fetch; stop state entered upon a found breakpoint, where it stays and waits for the next *"iEditor"* command
- ***3*** → Increment the "Program-Counter" and fetch next *"Macro"* through the "LP" block, which accesses *"Macros* FLASH" or "LP-RAM" depending upon being just executing normal *"Macros"* or being actively *"Live-Prototyping"* (for example see Fig. 5.1:37), respectively; also retrieve *"Macro* Modifiers" for later evaluation and possibly needed action.
- ***4*** → Priority modifier is compared to the current priority counter contents (see Fig. 4.5:73 and Fig. [Att. 31]:128). If the current *"Macro"* is due to be executed, then execute it at this moment. Otherwise, just return to point 3 and continue normally.

- *5* → Breakpoint modifier is checked and if set, go to point 2 and wait there for further commands from the *"iEditor"* telling what to do next (see Fig. 5.1:34 and Fig. 5.1:35).
- *6* → Atomic modifier is checked and, if set, disable *"GIMy"* updates and continue executing. As soon as a reset atomic modifier is found, re-enable *"GIMy"* updates and continue normally (see Fig. [Att. 49]:156).
- *7* → If the fetched *"Macro"* uses a temporary value, then fetch it for usage during calculations (see Tab. [Att. 23]:102 for *"Macros"* that have to use old values and state).
- *8* → If the fetched *"Macro"* uses a time, then fetch the current Real-Time Clock value for usage during calculations (see Tab. [Att. 23]:102 for *"Macros"* that have to use a time reference).
- *9* → Fetch the necessary *"Nodes"* for this *"Macro"* from the *"GIMy"*, as well as the *"Node Modifiers"*. These are then not really used directly herein, but are present in the *"Node"*.
- *10* → Execute the fetched *"Macro"* using the fetched *"Nodes"*, possibly also temporary values and timing values. This is done by feeding the ALU with the right sequence of values and applying the right actions upon them. The ALU then returns the results in that sequence.
- *11* → Store the resulting *"Node"* into the *"GIMy"*. After completing this storage, fully reset the *"Macro-Processor"* and return to the starting point 3 for the next *"Macro"*.

The "ALU" contains all basic arithmetic and logic operations needed for all possible *"Macros"*. Fig. 5.2:53 illustrates the internal structure of this "ALU", which uses the "BCDP" numeric format to represent numbers, just as inside the *"Nodes"* themselves in the first place. It also performs all calculations based upon the "BCDP" numbering scheme. A first test/feasibility prototype can be seen in Fig. [Att. 6]:18, whereas the final "ALU" block has then been integrated into the main *"FDEF-Processor"* FPGA hardware later on. The "SEQUENCER" commands the "ALU" to select the "OR" operation, then the input *"Nodes"* values are conveyed through the blue data bus, while the resulting *"Node"* value is retrieved through the red data bus. These data busses connect to all operations, while only the relevant segments are shown here for illustration simplicity reasons. Note that each one of the 10 operational blocks are independent relative to each other.

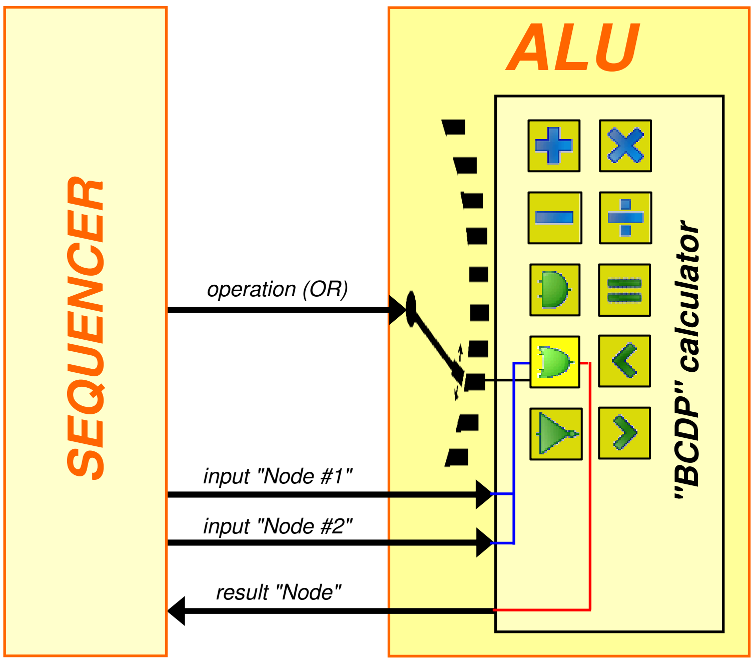


Fig. 5.2:53 – The "ALU" contains all the basic arithmetic and logic operations for all "Macros" needing them

5.2.1.3 "GIMy" Memory component

[Emphasis: the memory of the heart]

The "Gateway Intelligent Memory" (GIMy) is the internal RAM memory of the module. It stores all the *"Nodes"* processed by the *"Macros-Processor"* or routed among components. It has a total of 4 access ports that can be used for both read and write operations. It supports multiple reads in the same cycle (up to all 4 ports), but only one write per cycle. If two or more ports request a write operation in the same cycle, there is an automatic internal handshaking scheme that enables overall correct operation. In a very simplistic view, while a simple internal port priority/semaphoring mechanisms sequences simultaneous write requests signalized through the *write-enables*, one *acknowledgement* signal per port signalizes the requesting component when a write has completed successfully. Those components requesting the writes thus only have to wait for the completion. Fig. 5.2:54 shows the interface of the *"GIMy"* component. Contrary to classical RAM architectures, including paging, segmenting and access violations details, where extreme care must be taken in the way the memory is accessed, this *"GIMy"* is more of a "content-indexed" memory where no access errors are possible at all: it is accessed through an index that represents the desired *"Node"* and the related value is safely and completely retrieved, without any word-alignment, wait-states or any other pitfalls related to classical systems.

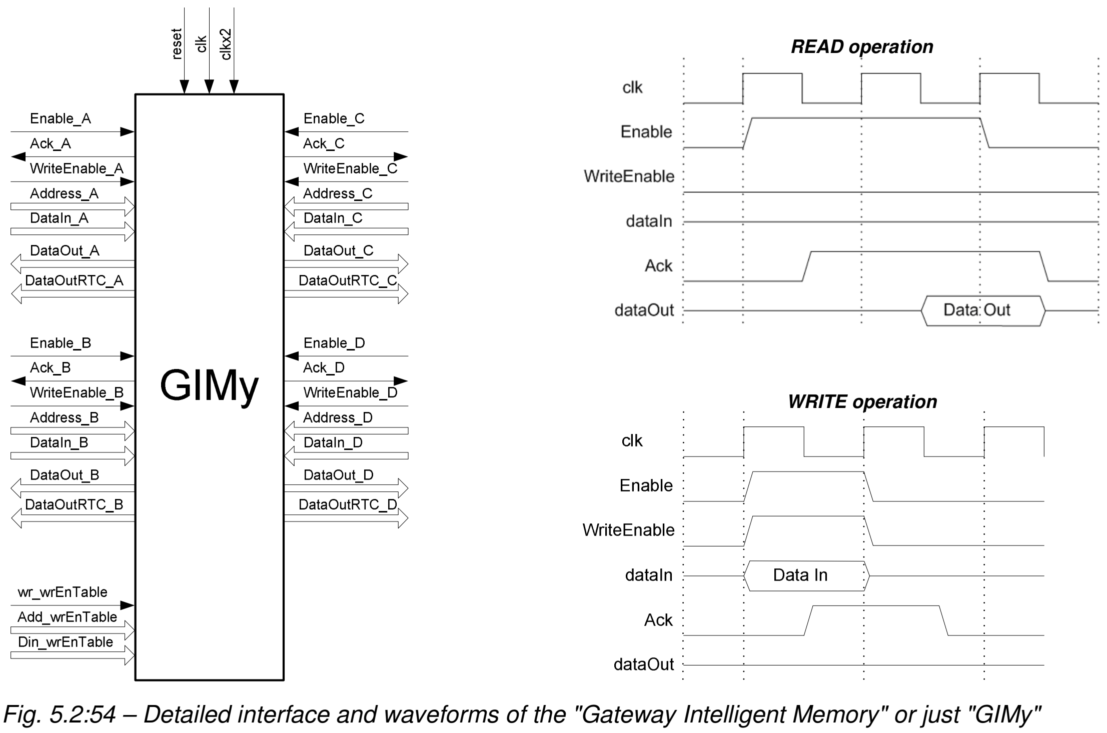


Fig. 5.2:54 – Detailed interface and waveforms of the "Gateway Intelligent Memory" or just "GIMy"

Normally, only one of the 4 components connected to the *"GIMy"* writes onto a particular *"Node"*, while all remaining 3 limit themselves reading and using that same *"Node"*. This programmatical concept of "One producer, many consumers" has already been extensively presented earlier in this work. So, instead of needing an efficient "concurrent-read concurrent-write (CRCW)" type of memory, a "concurrent-read exclusive-write (CREW)" may be implemented without reducing the system's performance at this level. Now that the complete *"Cellular-ECU"* architecture is known, a complete list of the possible situations better illustrates this unique global feature:

- **The *"Peripherals Manager"*** writes a sensor's final output value into a *"Node"* inside the *"GIMy"*, while it may be read by the "Log Manager" to log it into the MMC FLASH card, by the "Peripherals Manager" to feed an actuator, by the "Macros-Processor" to be processed through FDEFs and by the "Inter-Module Communications Manager" for other modules to also be able to use it as well. It would make no sense at all to have other components write to this same sensor *"Node"* as well, since this would simply destroy that sensor value on purpose.
- **The *"Macros-Processor"*** writes a *"Macro"* output value into a *"Node"* inside the *"GIMy"*, while all components (including the *"Macros-Processor"*, using this *"Node"* for the remaining FDEF) may then read this same *"Node"* for their own use, in exactly the same manner as described in the first point. Again, it would make no sense for others to destroy this *"Node"* on purpose.
- **The *"Inter-Module Communications Manager"*** writes another module's received *"Node"* into a *"Node"* inside the *"GIMy"*, while all components (including the *"Inter-Module Communications Manager"* itself, to pass the same *"Node"* to yet the next neighbouring module in the ring) may then read this same *"Node"* for their own use, in exactly the same manner as described in the first point. Again, it would make no sense for any component to destroy this *"Node"* on purpose.
- **The *"Log Manager"*** writes a value coming from the *"Live-Generating"* mechanism into a *"Node"* inside the *"GIMy"*. This is a special case, since this *"Live-Generating"* node-fixating mechanism potentially writes into *"Nodes"* that are updated by the other 3 components. But even in this case, no write-related collisions exists, since the *"iEditor"* disables all writes to that particular *"Node"* beforehand (except for the *"Live-Generating"* mechanism, of course). This mechanism is explained in detail in an earlier section.

Well-behaved automotive data-flow FDEFs follow this "one-producer, many-consumers" restriction, so that the apparent "multi-write" bottleneck is eliminated. An FDEF writing to the same *"Node"* multiply (if not using the better multiplexed data-flow construct, instead of the to-avoid classic "IF", see Fig. 4.5:74), would come from the same *"Macros-Processor"* were that FDEF resides, so writes would still be sequential. Even if there were more than one component concurrently writing to the same *"Node"*, this low probable occurrence would only be affected by a delay on *"GIMy"* accesses. Nevertheless, the elimination of the need for locks, semaphores or any other kind of potentially difficult to predict and fatal dead-lock causing mechanisms, including lock-less solutions such as in [564] (but still requiring processing intelligence to cope with outdated variables), due to the "one-producer, many-consumers" restriction, simplifies things at this point.

This *"GIMy"* behaves in a way where read/write conflicts in accessing the same shared memory location simultaneously are resolved by "Concurrent Read Exclusive Write (CREW)" in that multiple components can read the same *"Node"* but only one can write on a *"Node"* at a given time. Write operations to different *"Nodes"* are also fully parallel. If a write operation and read operation concurrently exist for a same *"Node"*, then the write operation is done first, so that the read operation reads the most actual value.

FINAL NOTE: Classical systems do not allow such simplicity regarding concurrent memory accesses, mainly due to complex caching and memory structures, requiring a great deal of useless/painful system insight.

From statements above, ***PITFALL C#6*** (Semaphores), ***PITFALL 0#17*** *(Caches)* is eliminated through effective adoption of simple central memory access priority, and without any compromise regarding the complete solution.

5.2.1.4 "Peripherals Manager" component

[Emphasis: the senses & muscles of the heart]

The "Peripherals Manager" communicates with attached sensors and actuators through a dedicated digital peripherals bus. Each bus can connect up to 16 different devices, where one of them must be a main-ECU module (bus master). This means that up to 15 peripherals may be connected to each main-ECU module. Each peripheral has a unique plug ID (detailed later), which takes values between 1 and 15, being zero reserved to the main-ECU module master. Fig. 5.2:55 shows the internal structure of this component.

The "Message Controller" is a state-machine that receives and processes messages that are sent to the "Peripheral Manager" via the "Configuration Manager". These message serve to either to update both internal routing-tables ("Gimy2Per" and "Per2Gimy") or to gateway information to/from the peripherals (for any high-level *"iEditor"* operations, *"Macros-Sequence"* downloading or any other peripheral *"Cellular-ECU"* management).

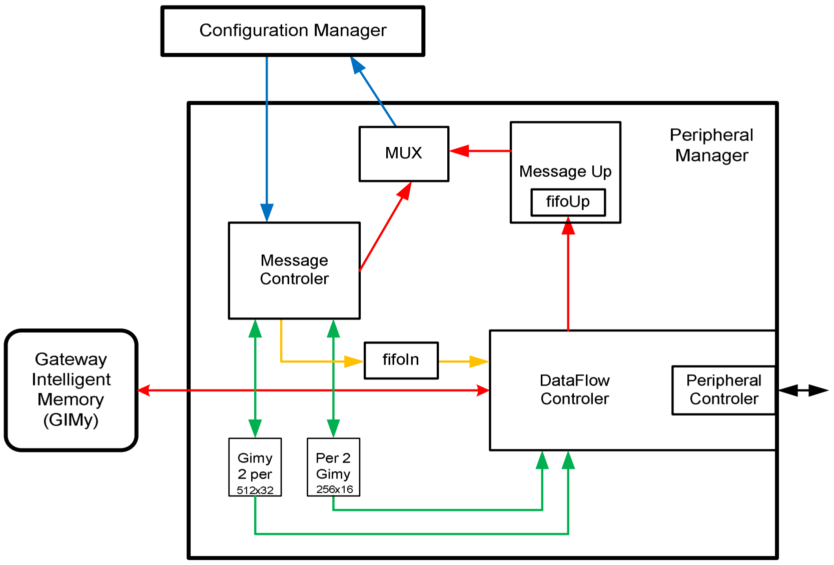


Fig. 5.2:55 – Close look at the detailed internals of the "Peripheral Manager"

The exchange of *"Node"* values between the master and the peripherals is always initialized by the master. There is an implementation limit of 16 values that can be exchanged with each peripheral. To exchange *"Node"* values the master sends these messages to the Peripherals (see **blue** path in Fig. 5.2:56):

- **Write value to peripheral's *"GIMy"*** - This kind of messages is generated by the "GIMy2Per" routing-table entries and contains 3 fields: "plug ID" (univocally identifies the peripheral location), "node ID" (identifies the destination *"Node"*) and the value to write. The peripheral receives this message and retrieves the real destination *"Node"* address through its own "Per2Gimy" routing-table based on the previous "node ID". Finally, the peripheral writes the value into that *"Node"* into its own *"GIMy"*.
- **Read value from peripheral's *"GIMy"*** - This kind of messages is generated by the "Per2GIMy" routing-table entries and contains only 2 fields: "plug ID" and "node ID". Again, the peripheral retrieves the real source *"Node"* address through its own "Per2Gimy" routing-table based on the previous "node ID". Finally, the peripheral reads the value from that *"Node"* out of its own *"GIMy"* and sends it back to the master.

There is also the need for the *"iEditor"* to exchange high-level information with the peripherals. This happens by simply gatewaying this information exchange through the "Peripheral Manager", through its "message controller". This simple mechanism makes peripherals think at all time that they actually are connected to the external *"iEditor"*. This allows to keep the concept of the ubiquitous utilization of the *"Cellular-ECU"* modules applied to smart peripherals, without disrupting the egocentric notion of each module thinking it is alone in the system and not depending on anyone to operate correctly. The only thing that interconnects these modules, as said earlier, is the *"Nodes"* exchange through the module inter-connections and the digital peripherals bus, while all internal components of each module work in exact the same way. Bearing this in mind, information exchanges between *"iEditor"* and peripherals operates as follows (paths in Fig. 5.2:56):

- ***"iEditor"* sends information to peripheral** (**red** path) - Sending information to a peripheral goes through the Main-ECU module's "Peripheral Manager" (acting as a gateway) and reaches the appropriate peripheral module's "Peripheral Manager". This one in turn gateways (FIFOup) the information to its own "Configuration Manager", which then redistributes the information exactly as if it was coming directly from the *"iEditor"* as if it was directly connected to this peripheral's "Configuration Manager". In other words, the peripheral's internal components will not be able to tell the difference between talking to the *"iEditor"* directly or through the above mentioned gateway mechanisms (e.g. for *"Live-Debugging"*, when receiving the peripheral's routing-tables or during *"Live-Prototyping"*).
- ***"iEditor"* requests information from peripheral** (**red**+**brown** paths) **-** Requesting information from a peripheral goes similar ways. First (**red** path), the *"iEditor"* request goes exactly the same path as the previous case, reaching the "Configuration Manager" of the appropriate peripheral. Then (**brown** path), this peripheral's "Configuration Manager" then would gather the response and try to send it to the USB-attached *"iEditor"*. Since there is none, it sends the response through the peripheral's "Peripheral Manager" instead (FIFOin), thus reaching the Main-ECU's "Peripheral Manager" and the Main-ECU's "Configuration Manager", all through simple gatewaying as before but in the opposite direction. Again, the peripheral's internal components will not be able to tell the difference between talking to the *"iEditor"* directly or through the above mentioned gateway mechanisms. Only the "Configuration Manager" is able to make that automatic distinction.

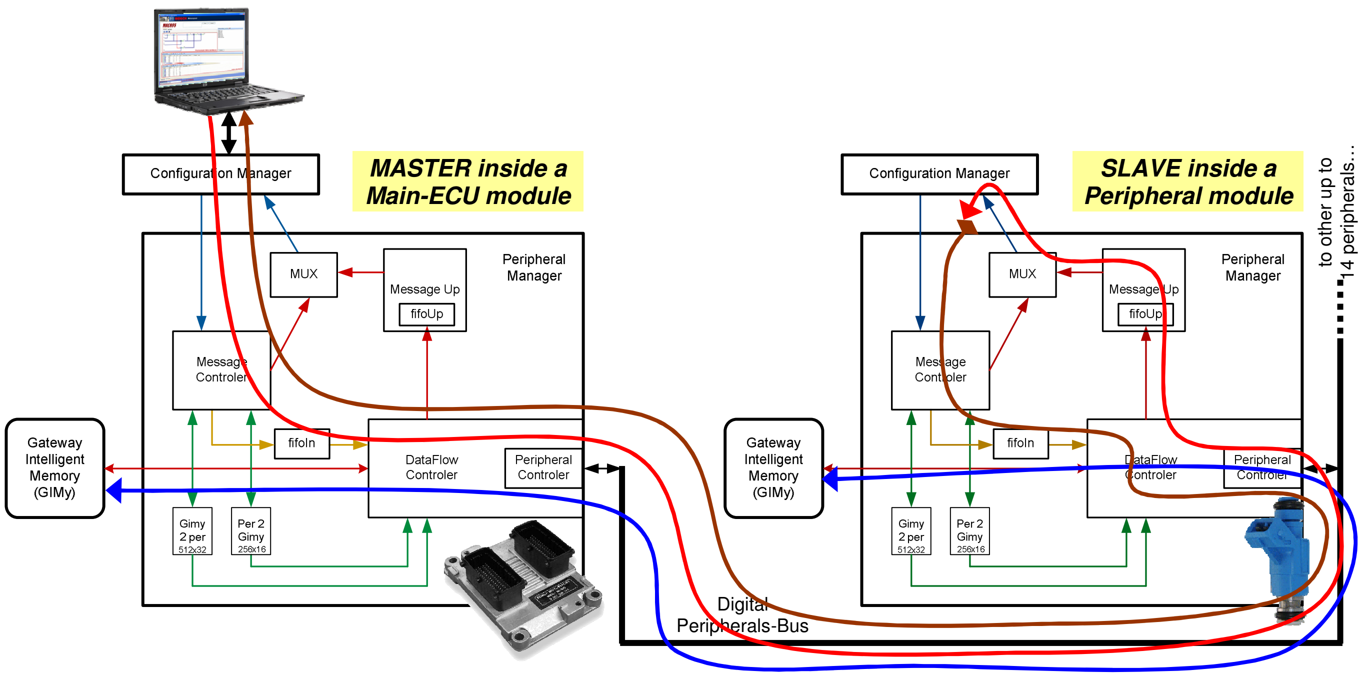


*Fig. 5.2:56 – "Nodes" exchange (in **blue**) and high-level information download paths (in **red**)*

5.2.1.5 "Log Manager" component

[Emphasis: the ECG of the heart]

The "Log Manager" is responsible for 3 distinct operations on the *"Nodes"* residing in the *"GIMy"*: logging onto MMC FLASH, transmitting them over the air (telemetry) and allowing for debugging (*"Live-Monitoring"* and *"Live-Generating"*). As already detailed earlier, the first mechanism allows the user to view *"Node"* values live on the *"iEditor"* while the second mechanism allows to fixate *"Nodes"* to force the FDEFs to use fixed test values.

The internal organization of this component is illustrated in Fig. 5.2:57. The entries inside the "Log Table" are limited to telling what *"Nodes"* are required to be sent whereto, through enabling bits for each path (Fig. 5.2:58). The *"iEditor"* or initialization FLASH fills this table with the desired entries. The "GIMy Access Mux" then fetches these *"Nodes"* from the *"GIMy"* and feed them either to the logging/telemetry blocks or directly to the *"iEditor"* again through the "Configuration Manager". A recording FIFO smoothes out speed differences between blocks.

This "Log Manager" component is also used to configure the "write enable tables" stored inside the *"GIMy"* component. These will then allow for the *"iEditor"* to perform *"Live-Generating"* features, explained in an earlier section.

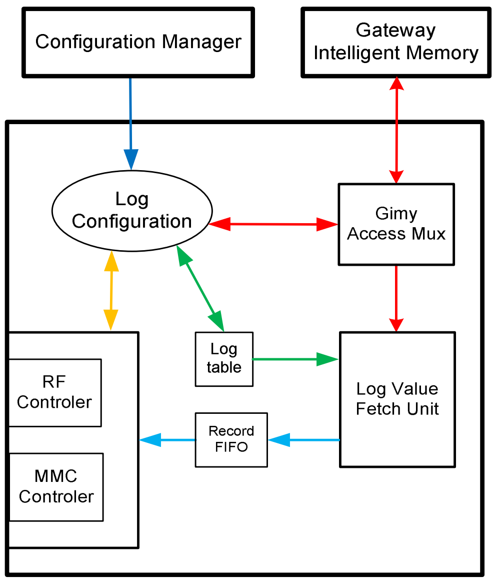


Fig. 5.2:57 – Close look at the detailed internals of the "Log Manager"

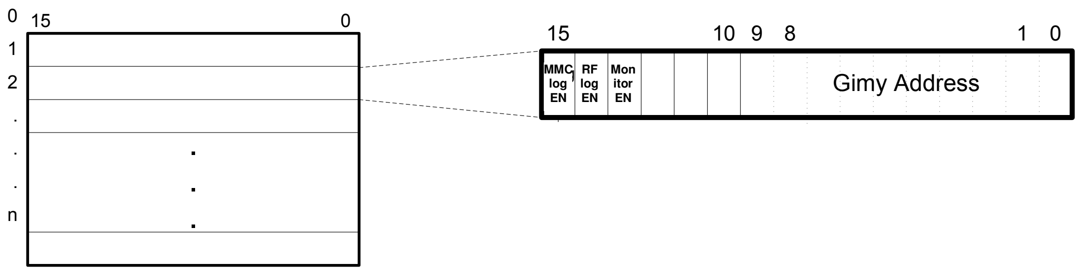


Fig. 5.2:58 – "Log Table" entries' format

5.2.1.6 "Inter-Module Communications Manager" component

[Emphasis: community of hearts]

In every multi-processing system, there is the intrinsic need to network all processing modules/units together, so they may exchange values, messages or whatever necessary for them to process the entire program as a coherent whole. Many are the networking possibilities, ranging from meshes, switchboards, cubes, multi-stage connections, crossbars, busses, rings, shared-memory and distributed-memory schemes, etc. [171] [523] [524]. Most of these networking mechanisms involve complexities whose amount even competes with the complexity of the rest of processing details. It is not uncommon for these networking mechanisms to possess highly advanced and complex intelligence, to be able to cope with all the problems that arise when inter-exchanging variables, messages, etc., and having to deal with different kinds of memory, etc. The networking mechanism developed herein will try to minimize these pitfalls altogether, by simply taking advantage of the *"Macros-Sequence"* processing and *"Node"* storage realities.

There is the need to exchange values between different *"Cellular-ECU"* modules that use the same *"Node"* (e.g. engine-speed, water-temperature, etc.). This "Intermodule Communication Manager" (Fig. 5.2:59) implements the serial ring communication between those modules. To minimize the number of connections from each module to the motherboard holding them, it was implemented using a serial communication protocol similar to the Serial Peripheral Interface (SPI), including transmit synchronization control. Each module has a bi-directional master and a bi-directional slave channel. This allows the implementation of a fully bi-directional (full-duplex) ring topology, like the one shown in Fig. 5.2:60, where a total of 5 *"Cellular-ECU"* modules are inter-connected. This topology allows to increase the number of modules in a straightforward way. In this inter-module *"Nodes"* exchange mechanism herein, we call originally produced *"Nodes"* as *"source Nodes"*, while those same *"Nodes"* consumed in other modules are conversely called as *"sink Nodes"*. *"Nodes"* produced and consumed only inside the same module are simply called *"Nodes"* with no differentiating prefix.

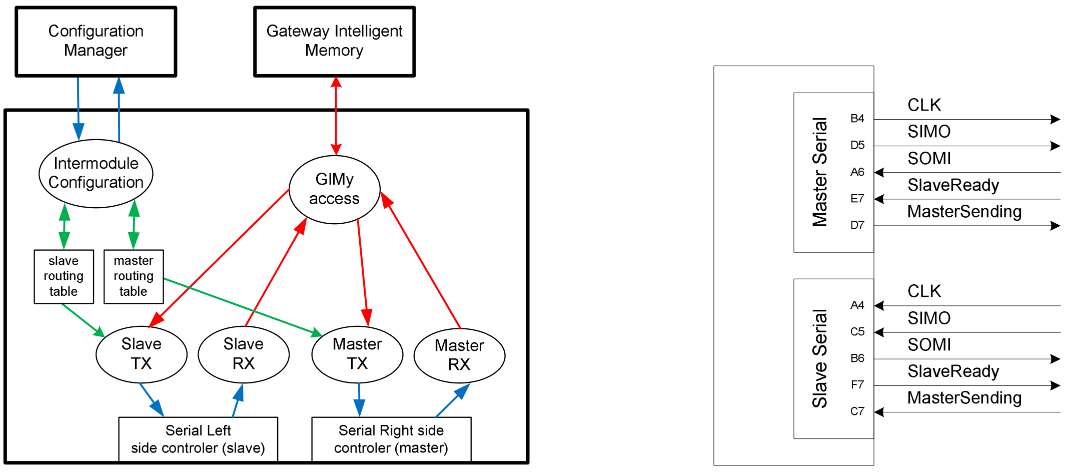


Fig. 5.2:59 – Close look at the detailed internals of the "Inter-Module Communications Manager"

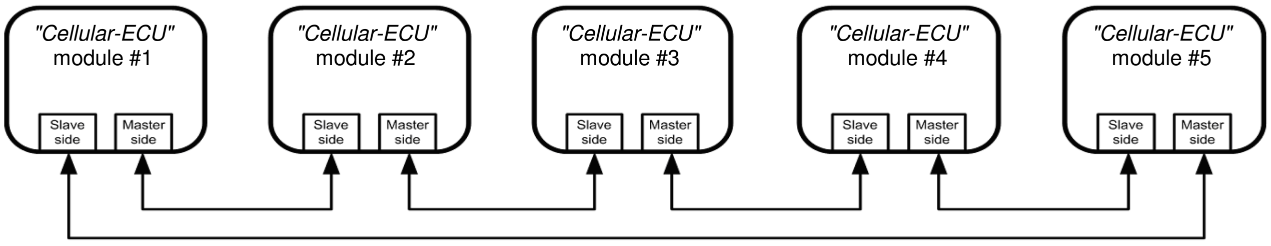


Fig. 5.2:60 – Detailed view the serial "ring-topology" connections between 5 "Cellular-ECU" modules

In Fig. 5.2:59, the "Intermodule Configuration" handles the communication with the "Configuration Manager". This path allows for the external *"iEditor"* to correctly configure the routing tables needed to establish which *"Nodes"* are going from the *"GIMy"* to which other modules and which *"Nodes"* are coming from other modules to the *"GIMy"*. The "GIMy Access" controls the access to the *"GIMy"* by the 4 following dual RX/TX state-machines in a simple round-robin scheme. The "Slave TX" is a state-machine that reads the "slave routing table" entries, gets the corresponding *"Nodes"* from the *"GIMy"* and builds the data frame to be sent out to the module connected to the slave interface. The "Slave RX" decodes the frames received by the slave interface, extracts the BCDP *"Node"* value and writes this *"Node"* into the *"GIMy"*. The "Master TX" operates similar to the previous "Slave TX", but sending out to the module connected to the master interface. The "Master RX" also operates similar to the "Slave RX", but receiving frames by the master interface. The "routing tables" contain simple entries, illustrated in Fig. 5.2:61, and are created by the external *"iEditor"* or directly through initialization FLASH contents routed to this "Inter-Module Communications Manager" through the "Configuration Manager" as mentioned earlier. These then allow the transmitting state-machines to easily determine which *"Nodes"* to transmit whereto. While *address in the current module* indicates the *"GIMy"* address of the desired *"Node"* to be sent, the *address in the next module* indicates the *"GIMy"* address where the transmitted *"Node"* must be stored in the neighbouring module. This last field is also transmitted with the *"Node"* value itself.

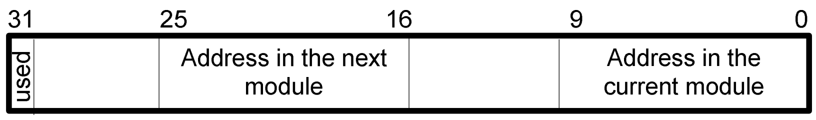


Fig. 5.2:61 – Master and slave routing table entry detail

A special situation occurs when a *"Node"* is not only used in the module where it arrives, but when it must continue to travel to yet the next neighbouring module. An illustrative example of this would be the engine-speed value, which is normally needed by almost all FDEFs and, thus, should be most probably present in all modules. This is a special case solved in another, more efficient and fast way through the peripherals' digital bus presented later herein. All the possible activities processed by the *"Inter-Module Communications Manager"* are illustrated in Fig. 5.2:62, including the normal internal paths just for reference. Since the ring topology is perfectly symmetrical, the longest path a *"Node"* would have to travel, is exactly half the ring perimeter.

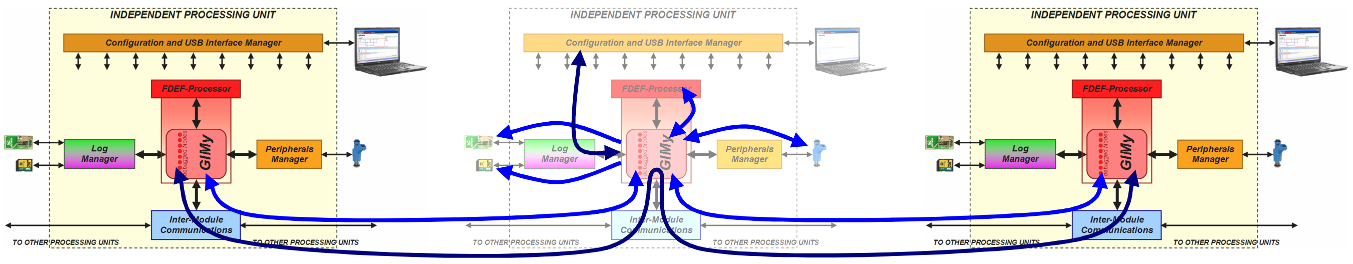

INDEPENDENT PROCESSING UNIT
Configuration and USB Interface Manager
FDEF-Processor
Log Manager
GIMy
Peripherals Manager
Inter-Module Communications
TO OTHER PROCESSING UNITS

Fig. 5.2:62 – All possible traveling paths for a particular "Node" (in **blue**)

Through a special algorithm, the complete topology of the Main-ECU modules can be automatically found. First, all USB connections are enumerated by the Windows PC and reported to the *"iEditor"*. Then, this algorithm writes specifically chosen IDs into the *"GIMys"* of the detected modules, sends them deterministically shifted to neighbouring modules' *"GIMys"* and then reads out all the those *"GIMys"* in the ring. According to the module where those shifted IDs appear, this algorithm is able to easily and automatically determine the complete modules' ring and report it graphically. This topology detection will be important for the correct automotive FDEFs' distribution explained later. Fig. 5.2:63 illustrates this algorithm at work for 3 *"Cellular-ECU"* modules.

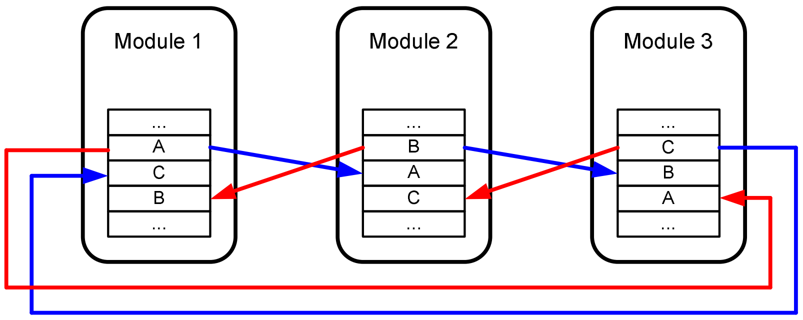


Fig. 5.2:63 – Automatic topology detecting algorithm at work

In this continuous exchange of *"Nodes"* implemented in this work there is no explicit synchronization or there are no wait-states either. Each process executes inside its own module without even ever taking part in the *"Nodes"* exchange process done by this "Inter-Module Communications Manager" with the participation of the *"GIMy"*. A said before, the *"Macros-Processor"* is fully independent and even unaware of any of such and all other external mechanisms.

Other value inter-distribution schemes used in other multi-processing architectures are normally affected by inter-locking/synching needs, waiting constraints, cache coherence maintenance mechanisms, potential memory access stall problems, simply because the *"GIMy"* plays the central "decoupling" role for the *"Nodes"* between modules. This is done in such a way that every intervenient needing access to the *"Nodes"* (peripherals-manager, *"FDEF-Processor"*, Logger, etc.) does not even notice those parallel operating inter-communications, nor do they need to engage in any interaction beyond the simple *"Nodes"* reading and writing. Best of all, in contrast to virtually all other methods, there is no increased programming effort to achieve this whatsoever. Also to be noted, is that no kind of arbiters is needed whatsoever also. Inter-communications are always on, always non-stop sending and receiving *"Nodes"* from the two neighbouring modules. Again, best of all, no special intelligence is needed inside the "Inter-Communications Manager" to achieve this immunity to classical problems in networking multi-processing systems. All this will minimize results' communications overhead mentioned in [554] during parallelization, since beside all the data always and readily existing in the *GIMy* of each *"Cellular-ECU"*, these inter-communications run in the background without any explicit program intervention, thus not directly entering the equation of parallelization speed-up.

5.2.2 Central ECU & FDEF Integration

[Emphasis: the processing heart of the automotive system]

As with any standard ECU ever used in any vehicle, the central unit devised herein is also going to contain processing, additionally to internal datalogging capabilities as in motorsport vehicles, but no peripherals electronics whatsoever. That is, it will communicate with the so-called *"Smart Peripherals"* through dedicated digital busses. It will also have communications capabilities towards an external computer. All in all it is represented by multiple "flesh & bones modules" in Fig. 5.1:44 and globally comprises the following in list form:

- ***"FDEF-Processing" capabilities*** *– processing of all main engine and chassis control-algorithms in form of FDEFs, with all their needed "Macros" and "Nodes"*
- ***Datalogging capabilities*** *– parallel logging of desired "Nodes" during normal FDEF processing*
- ***Telemetry capabilities*** *– parallel wireless "Nodes" transmission to external loggers or dongles*
- ***Peripheral input/output capabilities*** *– parallel readout of sensors and values output to actuators through dedicated bus-like digital networks, as well as gatewaying debugging and prototyping actions from the external host to those same peripherals*
- ***External host interaction capabilities*** *– USB connection to an external host (PC or laptop) for all needed user/developer debugging, prototyping and general operational actions*
- ***Global scalability capabilities*** *– adding more modules automatically adds more FDEF processing, "Nodes" memory, peripheral management and datalogging capacity to the system*

This "Central ECU" comprises the most important usage of the *"Cellular-ECU"* architecture, readily and mandatorily using all of its available capabilities. From Fig. [Att. 56]:203 in the use-cases section [Att. 56] onwards, examples on a vehicle (①) with its central ECU (⑧) are shown. Most of the next entities explained hereunder will not be using all of those capabilities because some will not be needed in those complementary contexts. Still, the basic *"Cellular-ECU"* idea is 100% there as well. It must be again emphasized that the FDEF distribution will be done in such a way, that FDEFs using certain peripherals will also be prioritized to be placed, if possible, in a module with direct connection to those peripherals. Other than that, the modules exchange *"Nodes"* between them, while the FDEF distribution will also privilege the least *"Nodes"* exchange possible, to keep communications and thus delays as small as possible.

5.2.3 Smart Peripherals & FDEF Integration

[Emphasis: intelligent & encapsulated sensing/acting units with processing capabilities]

Various sensors and actuators in state-of-the-art automobiles already possess integrated electronics and even software. Examples would be: xenon head-lights, intelligent lambda sensors, generators, ignition coils, absolute engine angular position, window lifters, etc. Most of these are limited to some integrated signal processing and/or power driving electronics, completely lacking software in a higher-level sense. The most "intelligent" sensor may certain Lambda-Sensors used for oil wells and such, where the electronics is not even inside the sensor but near it. This electronics includes a signal processing and special *"Nernst-Cell"* driving circuitry (chip), which communicates with the ECU through a standard SPI bus. Another example would be some absolute engine angular position sensors, which possess integrated electronics and software, allowing to be configured and to output angle values directly through a standard CAN bus. The *"Smart Peripherals"* presented and detailed here will take those local concepts a step further, both from hardware and software points of view.

As with any standard peripheral ever used in any vehicle, the peripherals devised herein are also going to contain normal sensing or actuating capabilities, but will additionally be capable of processing FDEFs themselves as well. This autonomous FDEF processing goes far beyond the standard internal electronics and usual low-level sensing/actuating processing needs, in that it allows to really integrate normal FDEFs just as those main FDEFs processed by the previously explained "Central ECU" unit. The only real difference is that these particular peripheral FDEFs will be processing lower-level peripheral signals, to get to the final desired peripheral physical value to be directly outputted towards the previous "Central ECU" through the parallel digital busses network. In a standard vehicle application, these *"Smart Peripherals"* will not use the USB host connection and the telemetry capabilities altogether. They could be used for special situations as listed next, however. Internal datalogging may still be used, again in the case of motorsport applications. All in all, *"Smart Peripherals"* are represented by multiple "skin modules" in Fig. 5.1:44 and globally comprises the following in list form:

- ***"FDEF-Processing" capabilities*** *– processing of special peripheral input/output signal conditioning and even some control-algorithms for special cases such as lambda-sensors, complex injectors, etc., all also in form of FDEFs, with all their needed "Macros" and "Nodes"*
- ***Datalogging capabilities*** *– logging of desired "Nodes" during normal FDEF processing*
- ***Telemetry capabilities*** *– wireless "Nodes" transmission to the "Central ECU" in a special case of a vehicle using low-priority low-importance wireless peripherals for difficult-to-reach vehicle locations, as well as wireless "Nodes" transmission to external dongles in case of no "Central ECU" being connected and a single peripheral being operated for isolated test purposes*
- ***Peripheral input/output capabilities*** *– output of sensor values and reception of actuator values to/from the "Central ECU", respectively, through the digital bus; also connectable to an external PC/laptop USB dongle containing a digital bus connection*
- ***External host interaction capabilities*** *– USB connection to an external host (PC/laptop) only in very special cases of no "Central ECU" being connected and a single prototyping peripheral being operated for isolated test purposes, while connected to a made available USB block*
- ***Global scalability capabilities*** *– no possibility of scaling anything for a "Smart Peripheral" whatsoever, while each peripheral has to be correctly dimensioned from conception*

From Fig. [Att. 56]:204 in the use-cases section [Att. 56] onwards, examples are shown useful cases, especially of those being able to place non-critical wireless *"Smart Peripherals"* (2) anywhere in the vehicle, without having the hassle of accessing a digital bus, while only the vehicle's standard power supply is needed if not even battery-powered. Fig. [Att. 56]:204 in [Att. 56] also shows the same peripheral being used for isolated wireless testing and prototyping (3), while other peripherals connected to an USB wired dongle (4) or even with an extra USB hardware (5) also link to a PC/laptop for testing and prototyping operations. Different types of peripherals possible to be added to FDEFs and vehicle views inside the *"iEditor"* can be viewed in Fig. 5.1:21.

Now for some more technical issues specific to peripheral needs: since these *"Smart Peripherals"* exist totally separated/outside from the previous "Central ECU", it is clear that their low-level resource needs such as interrupts, ports access, ADCs, DACs and timers, migrate to these peripherals and free up the "Central CPU" from the related burden and merging/co-existence complexity. This way, all those resources may be finely tuned for each peripheral's needs locally and, since all peripherals are effectively isolated from each other, this optimization is then also globally achieved. The "Central ECU" is thus completely freed from any low-level peripheral activity and may therefore dedicate to *"Macros"* processing and *"Nodes"* exchanging. Additionally, "priority inversion" and any other interrupts and priorities related problems may also be greatly reduced or even eliminated through the concentration and fine-tuning of the exact and only resources/features needed inside each peripheral. Peripherals such as temperature and pressure sensors do not even use interrupts at all. Speed sensors and such only use one single interrupt, therefore eliminating potential problems. This useful resource partitioning (not only higher-level engine/chassis FDEF functionalities) has already been preliminarily illustrated in Fig. 4.5:76 and more detailed in Fig. 4.5:77. Even more deeply technically speaking, "even-driven" processing and its timing pitfalls and complexity totally disappears from the "Central ECU", delegating to some peripherals only. The "Central ECU" thus really becomes an asynchronously/loosely-driven processing unit.

As for the *"Smart Peripheral"* interface, it could not be more direct and convenient: direct physical values are inputted/outputted to/from actuators/sensors, respectively, or a mixture of both input/output values on certain peripherals. Temperature sensors output direct values in [°C], pressure sensors output [bar], fuel injectors input both [°KW] and [g], etc. The convenience of this can be easily put as follows: any appropriate sensing technology can be used for any physical value, while that technology is packaged inside a *"Smart Peripheral"* that can be instantly connected to the digital bus, being instantly usable by the "Central ECU", without any other needs of setup or previous preparations. Example: NTC, PTC, thermo pair or any other temperature measuring technology may exist inside a particular *"Smart Peripheral"*, while always the same [°C] values will be outputted.

Likewise, actuators can contain any kind of technology, while the "Central ECU" will always know how to drive them properly through the unique physical values. Example: solenoid, piezo or any other technology may exist inside a fuel injector, while the "Central ECU" always drives that injector with [°KW] and [g] angular position and quantity values, respectively. In other words, the "Central ECU" is completely unaware of the technologies used, in terms of electrical connections to those *"Smart Peripherals"*. Of course, some high-level FDEFs inside the "Central ECU" and eventually all the lower-level ones inside those *"Smart Peripherals"* must be aware of the technology they are sensing/driving for behaviour fine-tuning. Furthermore, the fact that direct physical values are exchanged allows for simple and direct compatibility checks: a [°C] expecting FDEF peripheral input *"Node"* could never be connected to a [bar] outputting peripheral, without the *"iEditor"* ever noticing the mistake or switch. This further reduces human errors within the entire system scope, unlike classical systems where the wrong ADC input could easily be taken for an FDEF input without noticing until the FDEF is checked for its strange behaviour.

This virtual technology independence explanation done here would not be complete without mentioning that the units used are all ISO standard units, so that no quantisation conversions or related worries have ever to be taken. All in all, these *"Smart Peripherals"* incorporate the basic idea of really effective "Plug & Play" peripherals. All this explained above boils down to a "clean cut" done through the extensive usage of these *"Smart Peripherals"*, where there is a clear, very convenient and sharp separation between the "Central ECU" and all the sensing/actuation peripherals. Tab. 5.2:64 illustrates this "clean cut", while previous Fig. 5.1:21 shows all the possible peripherals and actuators that resulted from the benefit from such cut. Not all of them were really implemented in the *"iEditor"*, only the herein needed ones to get a demo engine operating correctly. Necessary and useful resources such as interrupts are therefore not simply eliminated, but were simply placed at the exact spots where they are needed.

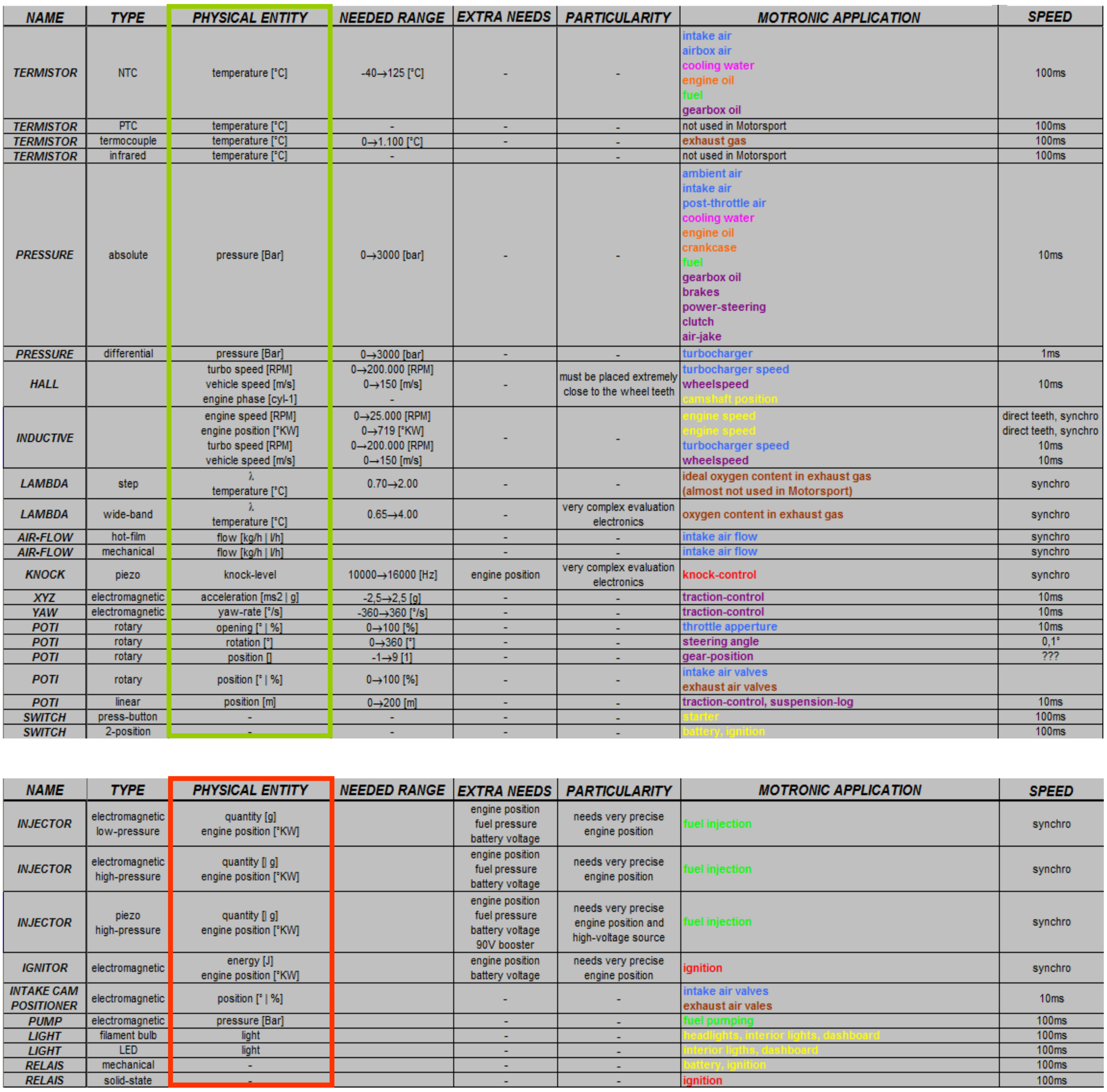
The "Central ECU" is therefore unloaded from all the natural burden those and all other hardware resources pose to the common/central driving software. Furthermore, this once central and concentrated burden is herein broken down in much smaller pieces and distributed among the peripherals according to their exact need. All this is called "functionality and corresponding resource DELEGATION". One very important consequence is also the transfer of all low-level peripheral-related FDEFs into the peripherals themselves, being this the most dramatic delegation of all. Furthermore, Fig. 5.2:65 shows a similar peripherals list (extract), but now with the internally generated (sensors) and needed (actuators) values.

Another nice side-effect of having this functionality and electronics delegation into peripherals, is that this allows for encapsulation of possibly advanced concepts such as: internal lowest-level online-learning/self-tuning and lowest-level factory tuning/calibration through software constants (instead of standard laser-, resistor- and other physical-based calibration types) without ever having to mention it to the main ECU. Therefore, swapping peripherals would then also never tamper with main ECU compatibility.

"CLEAN CUT"

Classical connection of peripherals to the ECU	***"Smart Peripherals"* connection to the ECU**
All peripheral related low-level functionality (FDEFs) also exist inside the central ECU, coexisting with all other higher-level FDEFs, increasing the overall burden, complexity and probability of interference upon the central ECU	All peripheral related low-level functionality (FDEFs) can conveniently be completely transferred to the peripherals, therefore unburdening the central ECU and maintaining a strict separation between peripheral and high-level FDEFs
Virtually all passive, non-intelligent, at most with some embedded power driving electronics but no processing power whatsoever	All intelligent, with extensive FDEF processing power and all the necessary sensing or driving power electronics embedded, essentially a *"Cellular ECU"*
All critical timing measurement, processing and driving inside the central ECU, causing large burden and programming complexity, while having to serve all peripherals with the available ECU resources	All strictly necessary critical timing measurement, processing and driving inside each peripheral, using its own strictly necessary resources, while being totally independent from all other peripherals in the system and completely unburdening the central ECU
All interrupts processing and priorities inside the central ECU, causing large burden and programming complexity to avoid interferences, while having to serve all peripherals with the available ECU resources	All strictly necessary interrupts inside each peripheral, using its own strictly needed resources, while being totally independent from all other peripherals in the system and completely unburdening the ECU
All hardware resources such as ADCs, DACs, ports and CAPCOMs exist inside the central ECU causing large burden and programming complexity, having to serve all peripherals with the available ECU resources	All strictly necessary hardware resources such as ADCs, DACs, ports and CAPCOMs exist inside each peripheral, while being totally independent from all other peripherals in the system and completely unburdening the ECU
All calibration parameters exist within the central ECU and coexist with all other parameters from all other peripherals, thus needing extensive user intervention when swapping different peripherals (different peripherals need different calibration sets)	Calibration parameters only exist within each individual peripheral, completely unloading the central ECU from this burden and thus allowing swapping peripherals and even ECUs without any need for any extra re-calibration tasks or related worries
All driver code inside central ECU must be compliant with the exact peripheral technologies used, thus requiring extensive programmer intervention whenever swapping peripherals with distinctive technologies	All driver code inside each peripheral, completely unloading the central ECU from this burden and thus allowing swapping peripherals with distinctive technologies without any need for any extra worries
All quantisation conversions are done inside the central ECU and must be compliant among FDEFs, thus requiring extensive programmer intervention whenever swapping different peripherals with distinctive output values	All necessary quantisation conversions are done inside each peripheral, whereas their output is always a direct standard ISO physical value, therefore allowing swapping different peripherals without any need for any worries

Tab. 5.2:64 – "Clean cut" that results from the usage of "Smart Peripherals"



NAME	TYPE	PHYSICAL ENTITY	NEEDED RANGE	EXTRA NEEDS	PARTICULARITY	MOTRONIC APPLICATION	SPEED
TERMISTOR	NTC	temperature [°C]	-40→125 [°C]	-	-	intake air airbox air cooling water engine oil fuel gearbox oil	100ms
TERMISTOR	PTC	temperature [°C]	-	-	-	not used in Motorsport	100ms
TERMISTOR	termocouple	temperature [°C]	0→1.100 [°C]	-	-	exhaust gas	100ms
TERMISTOR	infrared	temperature [°C]	-		-	not used in Motorsport	100ms
PRESSURE	absolute	pressure [Bar]	0→3000 [bar]	-	-	ambient air intake air post-throttle air cooling water engine oil crankcase fuel gearbox oil brakes power-steering clutch air-jake	10ms
PRESSURE	differential	pressure [Bar]	0→3000 [bar]	-	-	turbocharger	1ms
HALL		turbo speed [RPM] vehicle speed [m/s] engine phase [cyl-1]	0→200.000 [RPM] 0→150 [m/s] -	-	must be placed extremely close to the wheel teeth	turbocharger speed wheelspeed camshaft position	10ms
INDUCTIVE		engine speed [RPM] engine position [°KW] turbo speed [RPM] vehicle speed [m/s]	0→25.000 [RPM] 0→719 [°KW] 0→200.000 [RPM] 0→150 [m/s]	-	-	engine speed engine speed turbocharger speed wheelspeed	direct teeth, synchro direct teeth, synchro 10ms 10ms
LAMBDA	step	λ temperature [°C]	0.70→2.00	-	-	ideal oxygen content in exhaust gas (almost not used in Motorsport)	synchro
LAMBDA	wide-band	λ temperature [°C]	0.65→4.00	-	very complex evaluation electronics	oxygen content in exhaust gas	synchro
AIR-FLOW	hot-film	flow [kg/h | l/h]		-	-	intake air flow	synchro
AIR-FLOW	mechanical	flow [kg/h | l/h]		-	-	intake air flow	synchro
KNOCK	piezo	knock-level	10000→16000 [Hz]	engine position	very complex evaluation electronics	knock-control	synchro
XYZ	electromagnetic	acceleration [ms2 | g]	-2,5→2,5 [g]	-	-	traction-control	10ms
YAW	electromagnetic	yaw-rate [°/s]	-360→360 [°/s]	-	-	traction-control	10ms
POTI	rotary	opening [° | %]	0→100 [%]	-	-	throttle apperture	10ms
POTI	rotary	rotation [°]	0→360 [°]	-	-	steering angle	0,1°
POTI	rotary	position []	-1→9 [1]	-	-	gear-position	???
POTI	rotary	position [° | %]	0→100 [%]	-	-	intake air valves exhaust air valves	
POTI	linear	position [m]	0→200 [m]	-	-	traction-control, suspension-log	10ms
SWITCH	press-button	-	-	-	-	starter	100ms
SWITCH	2-position	-	-	-	-	battery, ignition	100ms

NAME	TYPE	PHYSICAL ENTITY	NEEDED RANGE	EXTRA NEEDS	PARTICULARITY	MOTRONIC APPLICATION	SPEED
INJECTOR	electromagnetic low-pressure	quantity [g] engine position [°KW]		engine position fuel pressure battery voltage	needs very precise engine position	fuel injection	synchro
INJECTOR	electromagnetic high-pressure	quantity [l g] engine position [°KW]		engine position fuel pressure battery voltage	needs very precise engine position	fuel injection	synchro
INJECTOR	piezo high-pressure	quantity [l g] engine position [°KW]		engine position fuel pressure battery voltage 90V booster	needs very precise engine position and high-voltage source	fuel injection	synchro
IGNITOR	electromagnetic	energy [J] engine position [°KW]		engine position battery voltage	needs very precise engine position	ignition	synchro
INTAKE CAM POSITIONER	electromagnetic	position [° | %]		-	-	intake air valves exhaust air vales	10ms
PUMP	electromagnetic	pressure [Bar]		-	-	fuel pumping	100ms
LIGHT	filament bulb	light		-	-	headlights, interior lights, dashboard	100ms
LIGHT	LED	light		-	-	interior ligths, dashboard	100ms
RELAIS	mechanical	-		-	-	battery, ignition	100ms
RELAIS	solid-state	-		-	-	ignition	100ms

Fig. 5.2:65 – Peripherals with different output (top sensors) and input (bottom actuators) direct physical values

All these peripherals also integrate complete diagnosis circuitry and software, sending to the "Central ECU" status values about electrical faults, general operational status, detected voltage level and current lifetime. These values add to the normal in/out *"Nodes"* and are also transmitted to the main ECU to be used in high-level FDEFs. Special "peripheral-only" low-level values such as ADCs, DACs, ports, CAPCOMs, etc., are made available for the peripheral itself to process them, but not to the main ECU (abstraction). These last internal *"Nodes"* are only made available to low-level FDEFs on the *"Smart Peripherals"* (physical in/out values and low-level hardware values). The complete list of values, corresponding usages and icons (used in the *"iEditor"*) are shown in the next table Tab. 5.2:66, while examples of their really implemented FDEF usage herein are shown in the examples of Fig. 5.2:67 (taken from [Att. 5]):

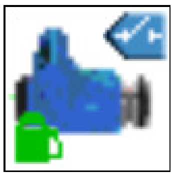
- → "OK", "Warning" and "Error" general diagnosis status, respectively, allows an FDEF
- → information present or not, respectively
- → peripheral associated/used in an FDEF or not, respectively
- → simulated, connected or disconnected peripheral, respectively
- → user-comments present or not, respectively
- → peripheral *"Nodes"* for use within any FDEFs, both high- and low-level diagnostic values: physical value, lifetime, internal temperature, supply, diagnosis and enable, respectively
- → peripheral *"Nodes"* for use only within low-level FDEFs: ADC, DAC, pulse capture, pulse compare/generation, switch output port, switch input port, PWM output, PWM input, angular output, respectively
- → lifetime OK, wearing out, exceeded, respectively
- → internal peripheral temperature OK, reaching maximum, exceeded, respectively
- → internally detected battery level OK, too low, too high, respectively
- → internal diagnosis overall status: diagnosis present, diagnosis OK, respectively

Tab. 5.2:66 – Full list of herein identified pitfalls found in most state-of-the-art development systems

These icons are used in the *"iEditor"* according to the location of appearance of the corresponding *"Smart Peripherals"*. There are three main places where these icons may appear: *"Live Vehicle & Peripherals View"* (peripherals' locations on the vehicle), *"Functional View"* (FDEF processing with shown peripherals) and *"ECU View"* (modules and peripherals connecting to the digital bus). The desired peripheral value to be used on a particular FDEF that is being edited, can be easily selected from an intuitive context menu/list (see Fig. [Att. 53]:173 for a real example) for each peripheral already present in the *"Live Vehicle View"*. These have thus to be first dragged onto at least one view to be readily usable in this intuitive way by FDEFs inside the *"Functional View"*.

→ appears in the *"Vehicle & Peripherals View"* and *"ECU View"*, where diagnosis and type information is relevant (as in Fig. 5.1:19 and Fig. [Att. 53]:174). This example shows an unassigned injector (- no FDEF uses it yet), which is to be simulated (), displaying an internal error () and containing user comments (). Fully connected and functional injectors would look like the one on the right.

→ appears in the *"Functions View"* , where type and value information is relevant (left of Fig. 5.2:67). This example shows an oil () temperature sensor value () that can be used inside FDEFs. The associated diagnosis () value shown on the right is then additionally used for the FDEF to act accordingly (switch to a default temperature if an error is present, for example).



→ at a more lower-level (*"Smart Peripherals"* level), the appearance would still similar but now with lower-level hardware values involved (right of Fig. 5.2:67), such as an ADC output (), valve closing port (), PWM floating valve control (), etc. These special *"Nodes"* are exclusively available inside the peripherals and can therefore only be used in FDEFs inside those peripherals.

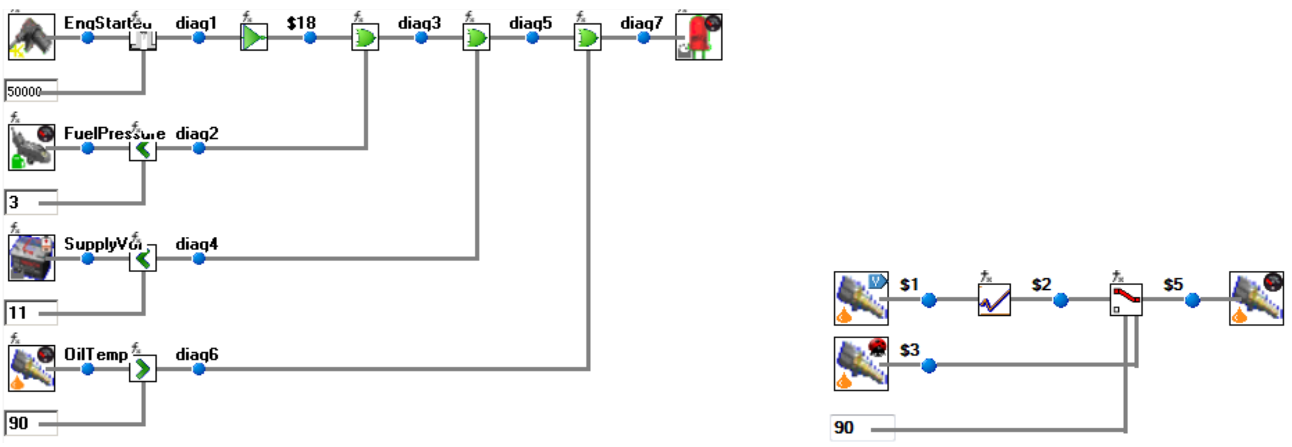


Fig. 5.2:67 – Left: high-level "Central-ECU" FDEF; Right: low-level "Smart Peripheral" FDEF

FINAL NOTE: All classical systems mostly rely on passive sensors/actuators and on a centralized electronic interface. This leads to the obvious need to have the main ECU to maintain everything under central control. This greatly increases software complexity, while reducing interfacing flexibility at the same time.

From the statements above, ***PITFALL C#26***$^{\text{(Interrupts)}}$, ***PITFALL O#72***$^{\text{(1/2 – Interfacing)}}$, ***PITFALL H#71***$^{\text{(Peripherals Handling)}}$ and ***PITFALL C#45***$^{\text{(Timers Multitude)}}$ are thus effectively eliminated through the use of exactly one “Cellular ECU” module per peripheral, where each one of such modules contains everything that is needed for processing the corresponding peripheral, without any central ECU intervention.

5.2.4 Data-Logging & ECU/Datalogger duality

[Emphasis: exactly the same ECU modules, turned the way around]

Although datalogging equipment is not of crucial importance while discussing the work implemented herein, its nevertheless non-negligible importance becomes clear when it comes to explain that the central *"Cellular ECU"* core-idea is, yet again, going to produce another hardware/software combined offspring. This is the second part of the already previously presented basic *"Live-Datalogging"* mechanisms and can be seen in [Att. 50].

5.2.5 Global Scalability & FDEF Distribution

[Emphasis: simply being able to increase the capacity of everything and distribute it wisely]

Here, only the system-wide scalability and functionality distribution possibilities are being discussed. Everything related to the resulting expected parallel processing and global efficiency issues, is then discussed in the section right after this one below.

Although it was mentioned earlier that one single *"Cellular-ECU"* module is perfectly able to handle all different tasks needed in an automotive ECU, it may be not sufficient for all applications. This calls for its scalability feature, in that several modules may be then used to increase the overall processing capacity of the entire automotive system, including peripherals and datalogging. As detailed earlier in previous Fig. 5.2:60, this mechanism is based on a closed "ring", where each module is then capable of receiving *"Nodes"* from both neighbouring modules, as well as transmitting *"Nodes"* to them. It can be easily seen that within this bus there are no "end-point" modules at all, thus keeping a homogeneous connection structure all along. The herein constructed Main-ECU will be composed out of 3 *"Cellular-ECU"* modules, since that was sufficient to operate the gasoline engine employed in the DEMO later on. For more details see [Att. 51].

During *"Cellular-ECU"* scalation through the ring, all of the internal module components are scaled linearly with that scalation without any further care being necessary, except for the fact that the FDEFs have to be somehow correctly distributed among the available modules. This is done by the external software control part, the *"iEditor"* itself. As with any other existing scalable/distributed computing systems, the processing power is never linearly multiplied by the number of modules added to the system, due to communications overheads [171] [340] and program parallelization limitations [412] issues. The difference lies in the fact that all the other components, such as memory, peripherals interfacing and datalogging are scaled quite linearly. A complete list of the components/resources that automatically/intrinsically scale with the addition of modules are as follows:

- Communications with the external host (multiple parallel "Configuration Managers" + USBs)
- Raw processing power (multiple parallel *"FDEF-Processors"*)
- Code storage memory (multiple parallel "FLASH Interfaces")
- *"Live-Prototyping"* control memory (multiple parallel "RAM Interfaces")
- Variables storage memory (multiple parallel *"GIMys"* for the *"Nodes"*)
- Peripherals interfacing capabilities (multiple parallel "Peripherals Managers")
- Debugging/Monitoring capabilities (multiple parallel *"Live-Debugging"* blocks)
- Internal Datalogging capabilities (multiple parallel "MMC Controller")
- External wireless Datalogging capabilities (multiple parallel "RF Controller")
- (each added module also comes with the needed "Inter-Module Communications Managers")

In other words, as long as one single *"Cellular-ECU"* module is constructed in such a way in that all of its internal components are more or less fine-tuned in matters of size and speed, such as to comply with the needs for a single of such modules, then scalation will take care of any extra needs in matters of global system growth. This then happens without the need for any complex hardware or software changes to accommodate the extra modules. Because each of those blocks takes care of itself and does not rely on any complex inter-locking mechanisms whatsoever, the big code parallelizing and debugging problems described in [415] also virtually disappear completely. In the system devised herein, exclusively *"Nodes"* are exchanged among modules, thereby eliminating many problems related to control-based parallelization. The *"Inter-Module Communications Managers"* inside each module accomplish the last bit of inter-connection. Furthermore, there is only a single-dimensional and homogeneous "horizontal scalation", instead of the usual "3D heterogeneous" scalation of classical system, where complexity rises much faster than the parallelization linear complexity increase of herein. Of course, the *"Cellular-ECU"* modules that comprise the system, will be mounted on a common motherboard, containing power supplies, USB hubs, inter-module wirings, etc.

Before going into the *"Allocator"*, Fig. 5.2:68 illustrates the true inside nature of an FDEF in terms of Code and Data location/distribution. This contrasts with classical systems processing, which have normally difficult to isolate and locally allocatable Code and Data items and where explicit developer-guided divisions have to be performed by explicitly changing the Code [454] and Data [455] representations in those systems. The FDEFs designed herein have only one unique representation, held by the intrinsically underlying *"Macros-Sequence"*, while further allowing a very simple *"Allocator"* to do the final distribution job. As said before, this work does not intend to devise a general-purpose processing system, but one that is best adapted to processing automotive data-flow functionalities. As seen correctly inside these particular FDEFs, as it happens with most automotive FDEFs, no functional dependencies exist whatsoever.

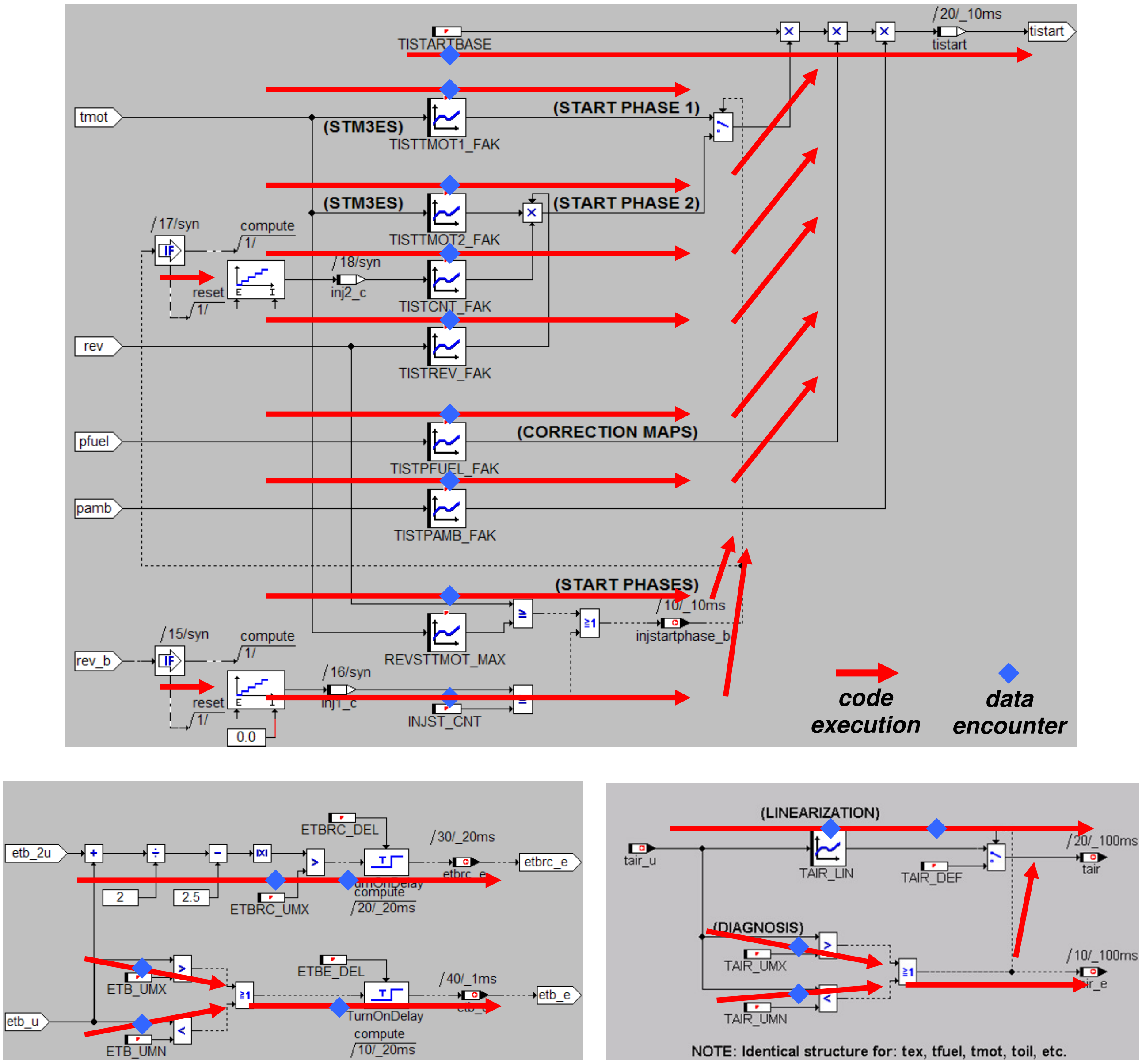


Fig. 5.2:68 – Code and Data distribution on a typical automotive system functionalities

The *"Allocator"* devised for the system herein is located on the right of Fig. 4.1:8, right before the "Downloader" takes the resulting code/data stream to the hardware, so that the correct illustration should the one in Fig. 5.2:69. Again, it is mainly a “mapper/locator” for *“Macros”* and nothing more. It has nothing to do with “linking” anything, just distributing among the available *“Cellular-ECU”* modules and *“Smart Peripherals”*. The main details that are important to understand this relatively simple allocation unit and how it is really possible to keep it as simple as it will be, are the following, among others:

- No synchronisation, locking and semaphoring needs between virtually all FDEF elements
- 100% asynchronous and dependency-less processing of virtually all FDEF elements, while asynchronicity has been inside some solution attempts in the more dynamic and global-solution-seeking industry scene [557]

- 100% clock-independent (due to the previous asynchronicity), while clocks only used to drive classical clock-based hardware logic circuits, but with the theoretical possibility of working on so-called "clock-less" hardware, as also being started to be followed by some industry [557].
- The *"Cellular-ECUs"* modules are functionally 100% isolated from one another and their inter-communications are also 100% asynchronous between them. This totally goes in the globally recommended direction [526] when talking about threads and the need to make them independent/isolated from each other and mutually non-blocking.
- Furthermore, the *"GIMy"*, as the "shared-memory" concept used herein, is also 100% independent and asynchronous relative to the *"Macros"* fetching and storing values there.
- Contrary to lowest-level Superscalability [232], parallelization is already intrinsically present at the highest-level data-flow FDEFs' nature and within the preliminary *"Smart Peripherals"* nature of external assignments, so that this parallelization "hue" travels to the right, in an already globally prepared/usable and intrinsically form
- Load-balancing will be efficiently solved through the heuristic geographic preliminary *"Smart Peripherals"* location, while this job will be further made easy due to the inexistence of classical dependency constraints
- Contrary to classical systems where optimizations, relocations and other complex memory location-related routines must be performed on the assembled code coming down for distribution, none of this is done herein
- The *"Cellular-ECUs"* modules already possess all that is needed to allow for the *"Macros"* to get their correct *"Nodes"*, from their own *"GIMy"* and also from other *"GIMys"*/modules, therefore eliminating the need for classic and complex linking procedures, as well as eliminating the need for classic memory paging, segmenting and related code/fill-ins preparations prior to download

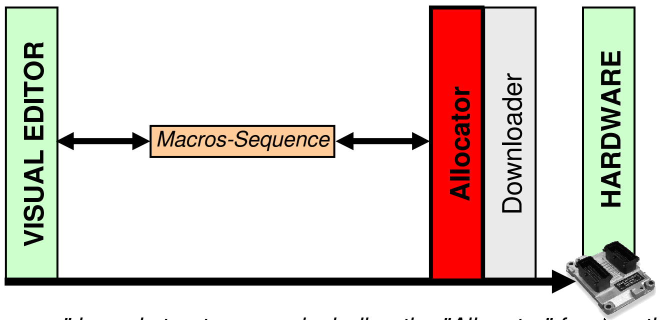


Fig. 5.2:69 – "Macros-Sequence" based structure now including the "Allocator" feeding the final "Downloader"

The peripherals attached to their respective *"Cellular-ECUs"* through their digital bus dictate most of the rest of the distribution. The developer only has to select which peripherals go to which digital bus cable, according to the real geographic cabling needs, while the *"iEditor"* then automatically calculates and indicates the expected individual bus and *"Cellular-ECU"* loads. This therefore allows the developer to re-place some peripherals elsewhere afterwards, to cope with the individual loads or, alternatively, add more *"Cellular-ECUs"* as needed to achieve a better load-balance with some split up digital busses. But the basic idea is that there will be already some distinct busses (injectors bus, igniters bus, sensors bus #1, sensors bus #2, etc.), which are normally quite well balanced from the start. The higher-level FDEFs directly serving those peripherals are then automatically placed by the *"iEditor"* onto those *"Cellular-ECU"* modules that physically serve those peripherals. Up to this point, it is fairly easy to include this heuristics into the *"Allocator"*, where even higher FDEFs serving no direct peripheral can be placed by the *"iEditor"* automatically, through trial-and-error approximations onto "less occupied" processing modules. The metrics used for this practical global tuning heuristic are:

- ***FDEF "Cycle-Time"*** → this metric relates directly to the processing capacity needed. The sum of all *"Cycle-Times"* of all FDEFs used inside each *"Cellular-ECU"* make up the total *"Cycle-Time"*. These are not necessarily equal, to guarantee a good global load-balance. In practice, some modules will have very small *"Cycle-Times"* (fastest injection, ignition and knocking FDEFs), while others may have much larger *"Cycle-Times"* (slower sensors and other slower processes). Priority related details of FDEFs optionally make many *"Macros"* go slower than the fastest "Prio1" ones inside the same module, but the total *"Cycle-Time"* in a module must always be smaller than the maximum allowable/desired processing cycle of the fastest *"Macro"* or entire FDEF being processed in that same particular module. It should also be as close as possible to that maximum value, to allow for maximum efficiency of that particular module. This thus embodies the first rule class of the global *"Allocator"* heuristics and boils down to the "relative load", which must remain under 100% at all time and as near to 100% as possible. Having less "IF-THEN-ELSE" constructs and more pure data-flow processing, *"Cycle-Times"* will not vary much, thus making it easier to achieve the before mentioned goals without too much variation or miss.
- ***Inter-Communications*** → this metric directly relates to the amount of *"Nodes"* being exchanged among main-ECU modules. As in the previous *"FDEF Cycle-Time"*, it might sound reasonable to have the minimum possible amount of *"Nodes*" being exchanged this way, but slower modules (larger *"Cycle-Times"*) may also exchange their *"Nodes"* slower. The "Updated" bit inside the *"Nodes' "* internal structure would allow for the "inter-communications manager" to optimize these transfers, by leaving non-updated *"Nodes"* and going immediately to the next one. Either way, this is not critical and was not used herein, since the inter-communications busses only connect in a peer-to-peer fashion and therefore technically not needing to throttle their activity at all. Because of this, this global ring-bus around the modules was intentionally left simply unthrottled/unprioritized, therefore exchanging/updating many slowly changing *"Nodes"* (creating many duplicates on the ring-bus). Similarly to the previous *"FDEF Cycle-Time"* heuristics, the main rule here, is to always try to maintain these exchanges fast enough, so that all *"Nodes"* transmitted to the neighbouring modules are always transferred at least once inside each single *"Cycle-Time"* of the transmitting module. This guarantees that the latest *"Nodes"* updates always get to those neighbouring *"GIMy"*, independently of being used there so frequently or not. Again, a "relative load" may be defined, which must remain under 100% at all time but does not have to be near 100% to maximize efficiency.
- ***"Smart Peripherals" digital-bus*** → this third objective metric concerns the total occupancy of the various digital-busses that serve their respective peripherals. As with the two previous metrics, what is desired here, is that the outputting peripherals (actuators) get their needed values on time, so that the real-time physical processes associated to those peripherals get updated in a timely fashion. The same goes for inputting peripherals (sensors), where their values have to reach the corresponding FDEFs on the corresponding *"Cellular-ECU"* modules also on time, so that those FDEFs always work with the most fresh/updated values as well. There might even be mixed peripherals needing output values and feedbacking input values (e.g. lambda-sensor, knock-sensor). Similarly to the "inter-communications", the rule here is twofold: outputting *"Nodes"* (from ECU to peripherals) should be transmitted out at least once inside each single *"Cycle-Time"* of the transmitting ECU module, while the inputting *"Nodes"* (from peripherals to ECU) should be transmitted back to the ECU also within each single *"Cycle-Time"* of the transmitting peripheral. Exactly as above, this again insures that no FDEF will process outdated values, even if not needed with that production frequency.

It must be clearly said that there is no point in making an FDEF too fast and then over-crowding busses with its produced *"Nodes"*. Every FDEF must have the exact *"Cycle-Time"* that it has to have to operate correctly, not faster and not slower. Fig. 5.2:70 illustrates a simple example of how the real and thus imperfect distribution may look like (6-cylinder engine with 6 injectors, 6 igniters, 4 fuel pressure/temperature sensors and several other peripherals needed for its operation). While this simple "first-try" grouping of peripherals seems easy and obvious, the resulting processing and communications loads are not optimal and need further fine-tuning. This need also has to do with the intrinsic speed nature/needs of every peripheral (slow ⏱ and fast 🐌), as well as the FDEFs present in each main *"Cellular-ECU"* module (quantity and speeds). The 🟩 boxes show the relative load/occupancy of busses and modules in terms of their respective processing/communication capacities. The size of these "load boxes" also illustrate the amount of *"Macros"* being processed inside each module, which means that many slow FDEFs may need less processing power and thus represent less real load, when compared to a few fast-processing ones. The relative loads of the "Smart Peripherals" themselves must always be in the green range, since they are not partitionable. Therefore, those loads are not explicitly shown here. Loads are calculated through the following formulas (same sequence as the above heuristics):

- For *"Cellular-ECUs"* processing *"Macros":* $load = ct_{\max} / ct_{allowable}$, where *"ct$_{max}$"* is the real maximum/peak *"Cycle-Time"* inside a *"Cellular-ECU"* and where *"ct$_{allowable}$"* is the maximum allowable *"Cycle-Time"* to guarantee correct operation of the FDEFs. This formula is directly derived from the first metric explained above, stating that one should always guarantee that $ct_{\max} \leq ct_{allowable}$ at all time, so that $load \leq 100\%$ also at all time. Any load passing the 100% mark simply means that processing power is not enough and FDEFs must be moved to other *"Cellular-ECUs"* or that new *"Cellular-ECUs"* should be added to the system.

- For *"Inter-Communications"* exchanging *"Nodes":* since this bus is bi-directional (full-duplex SPI-like), there really are two loads, where both can be represented by $load = ht / ct_{\min(L/R)}$, where *"ht"* is the total hop-time to transfer all *"Nodes"* from one producing *"Cellular-ECU"* to its neighbour and where *"ct$_{min(L/R)}$"* is the fastest *"Cycle-Time"* of the producer of those *"Nodes"* relative to the left$_{(L)}$ and right$_{(R)}$ producers, respectively. This formula is directly derived from the second metric explained above, stating that one should always guarantee that $ht \leq ct_{\min}$ for all *"Nodes"* at all time, so that $load \leq 100\%$ also at all time. Any load passing the 100% mark simply means that communications power is not enough and that too many *"Nodes"* are being exchanged. Thus, FDEFs must be moved to other *"Cellular-ECUs"* to get nearer to the producers of those *"Nodes"*. Some *"Nodes"* may even be exchanged "too quickly", in case of slower consuming FDEFs. If a *"Node"* needs to travel farther than the next neighbour, then that neighbour will act as a "gateway" and, thus, as the "pseudo-producer" for the next neighbour. For a correct load calculation from this "gateway" on, the *iEditor* keeps track of the original *"ct"* corresponding to each *"Node"*.

- For *"Smart Peripherals"* digital-bus*:* this bus is also bi-directional (half-duplex) and thus also presents two loads: the one referring to *"Nodes"* travelling from the ECU to the peripherals and the one for the way around. $load = bt / ct_{\min(U/L)}$, where *"bt"* is the total bus-time to transfer all *"Nodes"* from the producing *"Cellular-ECU"* to the consuming *"Smart-Peripherals"* or from the producing *"Smart-Peripherals"* to the consuming *"Cellular-ECU"* and where *"ct$_{min(U/L)}$"* is the fastest *"Cycle-Time"* of the producer of those *"Nodes"* relative to the upper$_{(U)}$ and lower$_{(L)}$ producers, respectively. This formula is directly derived from the third metric explained above, stating that one should always guarantee that $bt \leq ct_{\min}$ at all time, so that $load \leq 100\%$ also at all time. Any load passing the 100% mark simply means that communications power is not enough and that too many *"Nodes"* are being exchanged. Thus, peripherals must be moved to other digital-busses.

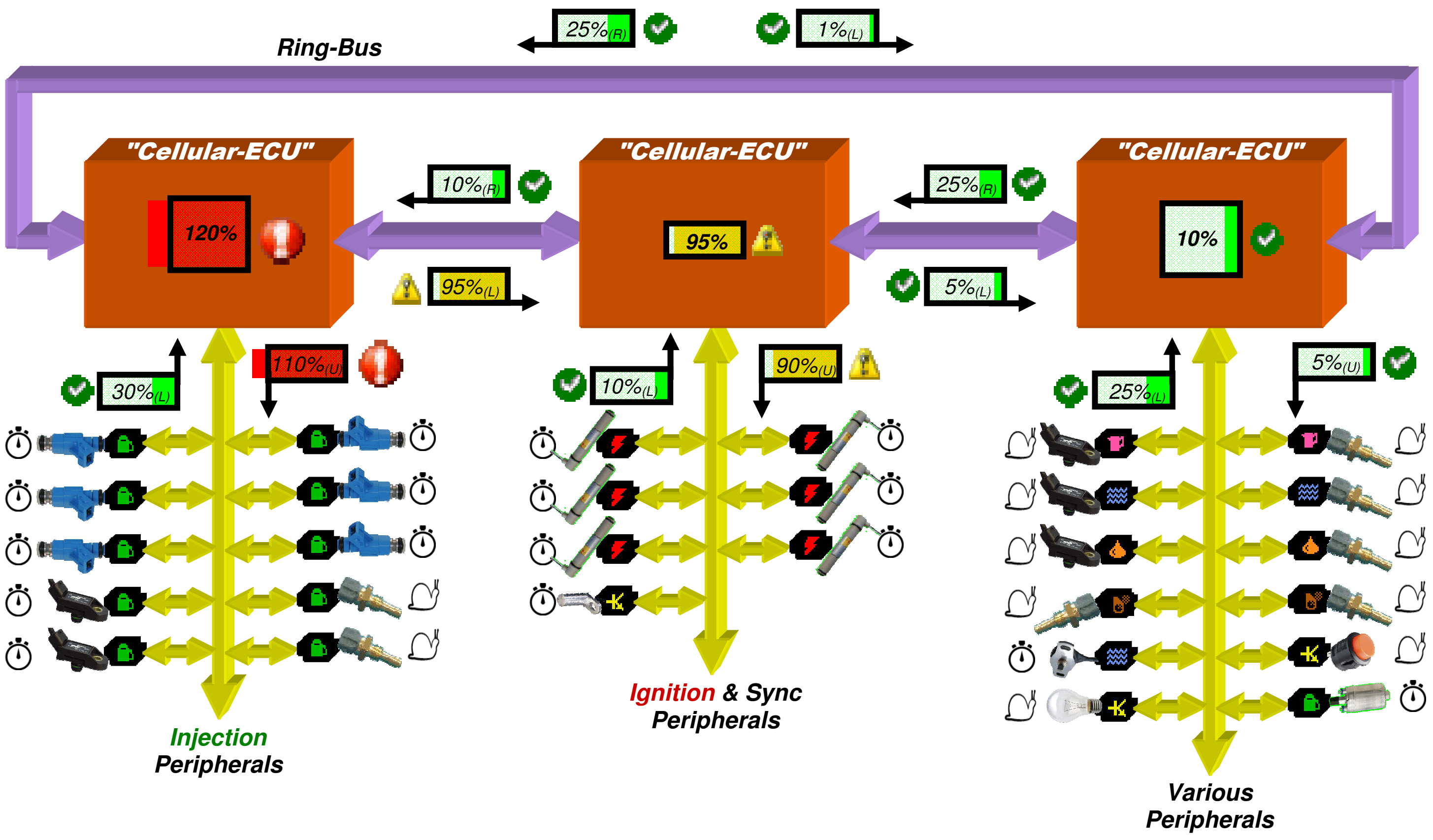


Fig. 5.2:70 – System with 6-Cylinder engine with 6 injectors + 6 igniters, as well as various other peripherals

Fig. 5.2:70 above is very similar to the "ECU-View" shown in Fig. [Att. 53]:174 of the *iEditor* software interface details explained in [Att. 53]. It is there where these load values should be shown for the system architect to better prepare his systems to perform best, by fine-tuning the occupancies of the various processing and communications paths. This would be achieved by tuning the system in such a way, that all those loads will turn out to be lower than 100% or, at least, as close to 100% as possible.

It must be reminded here that, as already mentioned many times throughout this work, compared to the classical parallel processing difficulties and even virtual impossibilities of separating highly inter-woven code, all this herein is only possible because of the already detailed orthogonal, independent and asynchronous nature of most of the components of this system developed herein. While classical parallel processing is poised by highly inter-connected, inter-locked and mandatorily synchronised solutions, here all that disappears. Only feed-forward mechanisms are mainly used, in that no explicit synchronisation is use anywhere, basically and simply letting all sub-components such as *"Macro"* processing and bus communications happen asynchronously relative to each other. Instead of absolute synchronisation, only overall checking mechanisms are used to know if the resulting global processing meets certain minimum requirements. And even if those requirements are not met, the system still continues working without hard-limiting resets or problems, because of the also already explained "graceful degradation".

Observing the previous three basic heuristic rules, it becomes fairly simple to observe that all of this may be seen as “forward guarantee heuristics”. Reminding the fact of the FDEFs herein making use of the premise of “ONE producer, many consumers”, one can easily conclude that it is the producer that has to be "pampered" so that its produced *"Nodes"* find the way out of it and into the consumers. As long as all the consumers get continuously hit by the producer's values in an unconstraint manner (all produced values are passed on to all the consumers needing those values, regardless of the fact that they are really used at that production frequency or not), then it is intrinsically guaranteed and, above all, in a simple fashion, that no production is lost. Of course, if a certain value is produced with a frequency of 1000Hz (*"Cycle-Time"* of 1[ms]), a consumer might get hit at that frequency for that particular value, but might as well use it only at a frequency of only 100Hz (*"Cycle-Time"* of 10[ms]). Some other FDEF will certainly use that value at almost 1000Hz, of course, or else it would not make sense to produce it so quickly. This has to do with the processing-speed management of the individual FDEFs and their *"Macros"*, a matter of the scope of the developers and Mechanical Engineers devoted to designing those FDEFs, while out of the scope of the work herein. The scope of the work herein is merely to guarantee a good as possible solution for the distribution of FDEFs and related timing warnings, given the set of ready-made FDEFs.

One related issue with this "ONE producer, many consumers" timing has directly to do with the *"LIVE-Priorities"* automatism already explained earlier, which propagates priorities in such a manner, that slower *"Macros"* only get faster processed when it becomes strictly necessary and not uselessly before that. It becomes therefore apparent that everything related to processing timings, namely *"Cycle-Times"* and *"Nodes"* interchanges can be looked upon in the before mentioned "forward guarantee heuristics" by condensing everything into a simple formulation that covers all these heuristics and centring onto the most important data activity in the entire system throughout – the *"Production of Nodes"*:

All "Nodes" shall be "Produced" and "Conveyed" in a mostly timely manner to all receiving consumers

This brings the before mentioned potential for "sub-optimal" bandwidth utilization in those case where no consumer needs values as fast as they are being produced but, then again, this kind of optimization must occur at FDEF development/design level and at communications' level. There, the correct priority *"Macro Modifiers"* should be used to keep producing FDEFs outputting *"Nodes"* at the predictably needed rate and, on the other side, to keep consuming FDEFs not going faster than the produces *"Nodes"* are being produced and arriving at them. For more details and other main-ECU load scenarios and illustrations, please refer to [Att. 51].

"Routing-tables" already described earlier herein (see Fig. 5.2:49) are needed for the hardware to know which *"Nodes"* to exchange between neighbouring *"Cellular-ECUs"* (embodied inside the "Inter-Communications Manager") and between *"Cellular-ECUs"* and their respective peripherals' digital bus (embodied inside the *"Peripherals Manager"*). These "routing-tables" have to be generated at the same stage where the *"Macros-Sequences*" are being distributed, i.e. inside the *"Allocator"* as well. This also very important but totally straightforward task is done basically by a simple lookup routine that looks for *"Nodes"* being consumed elsewhere besides the producing module itself, generating a "routing-table" entry for each consumer outside that module. A producing module may be a main-ECU or peripheral one, the mechanism is exactly the same. Besides distributing and generating "routing-tables", only one small but essential detail remains: since all *"Nodes"* inside the *"iEditor"* possess an unique index, the *"Allocator"* additionally needs to remap those indexes to start at zero, for each distinct *"GIMy"* inside each distinct *"Cellular-ECU"*. Thus, the list of tasks realized by this *"Allocator"* is as follows:

- Distributing *"Macros-Sequences"* among all the available *"Cellular-ECUs"* (both for main-ECU and peripherals' modules) according to a "location-aware *Cycle-Time*-based" heuristic
- Generating the exactly needed "routing-tables" to achieve the correspondingly necessary *"Nodes"* exchange among the before mentioned modules and according to their distribution
- Re-indexing *"Nodes"* for each module, so that the *"GIMys"* always have zero-indexed *"Nodes"*

Since the *"GIMy"* inside each *"Cellular-ECU"* is basically accessed directly and without additional delay due to hashing or any other permutation mechanisms whatsoever, this eliminates the need to deal with all kinds of problems related to hashing, such as delays, collisions, etc., as stated in the existing extensive academic work done in this particular area. It does not need any of that, because access contention is solved intrinsically in a four-fold solution, within this order:

- The system-wide imposed "one producer, many consumers" rule avoids multi-write contentions, therefore allowing for a truly free-running data-producing and consuming system
- All 4 existing accessing points to the *"GIMy"* (see Fig. [Att. 49]:155) are already capable of concurrent operation (multi-read 100% transparent, multi-write forbidden by the previous point)
- Multiple *"Cellular-ECU"* modules with their inter-communications managers buffering the *"GIMy"* from the outside, allow to further scale processing capabilities without interfering with the already transparent and contentions-less *"GIMy"* accesses inside each module (PRAM/DMM systems explicitly issue access commands to more distant memory, increasing latencies and complexity, which get even worse if the other party is somehow busy)
- Circular "Inter-Communications" links between *"Cellular-ECUs"* permanently exchange *"Nodes"*. As already mentioned before herein, all *"Nodes"* shall be "Produced" and "Conveyed" in a timely manner to all receiving consumers in such a way that no latency will be noticed whatsoever. Latency may still appear between long-distances if FDEF distribution not well optimised (such as faster *"Node"* production not kept close to the respective consuming modules).

Perfect equilibrium regarding FDEF distribution and related data access mechanisms' contention optimization, would have to be studied and tested in future work. Contrary to classical multi-processing systems, the system devised herein has the basically great advantage to have a linear complexity growth when processing modules are added.

FINAL NOTE: Classical parallel-processing systems most often rely on the use of complex entities such as threads and other parallel-processing related structures, while making it difficult to scale up due to the related effort needed to split programs into several cores and maintaining coherence between those split-up code/data blocks.

From the statements above and from all considerations concerning the internal architecture of this computing system, ***PITFALL H#75*** (Scalability), ***PITFALL C#44*** (Threads), ***PITFALL O#72*** (1/2 – Interfacing), ***PITFALL O#83*** (System Hardware Debugging), ***PITFALL O#18*** (Memory Access Hashing) and ***PITFALL O#32*** (Multi-Core) are thus effectively eliminated through the use of a very simple but usable "cellular" architecture that allows for relatively easy parallel hardware constructs with powerful parallel-processing capabilities without the need for complex software or hardware structures.

5.2.6 *"Macros-Sequence"*, *"Cellular-ECU"* & Parallel Processing revisited

[Emphasis: wrapping it all up to process and use the distributed capacity]

Some considerations will be made about expected speed-up and parallelizable characteristics will be made before finishing this section. As long as automotive FDEFs are then distributed efficiently throughout these parallel processing modules, the total processing power will increase with each added module, without disregard of theoretical limits such as Amdahl's Law [412] depicted in Fig. 5.2:71 in its various forms of representation. This law is quite intuitively understandable and reflects the fact that the only program portion that still needs a fixed amount of time to execute in an ideal infinite-processors type of hardware, is the one that is not parallelizable. This theoretical limit also assumes that all processors are able to freely and unconstraintly access all data memory variables they need to process. This is clearly not the case within any practical system, as it is not the case with the *"Cellular-ECU"* architecture, since its "ring-topology" in Fig. 5.2:60 clearly shows that this simple topology requires that *"Nodes"* generated in module #1 and needed in module #3 need to traverse module #2 and so forth. Efficient distribution of the FDEFs throughout those modules are thus of great importance as already mentioned in the previous section. An example of such a care can be seen in [469] (Fig. [Att. 60]:243), where code distribution and the closely related inter-chip-communications where optimized in such a way, where latencies were reduced to a minimum, therefore granting quasi-linear performance increase when adding new chips.

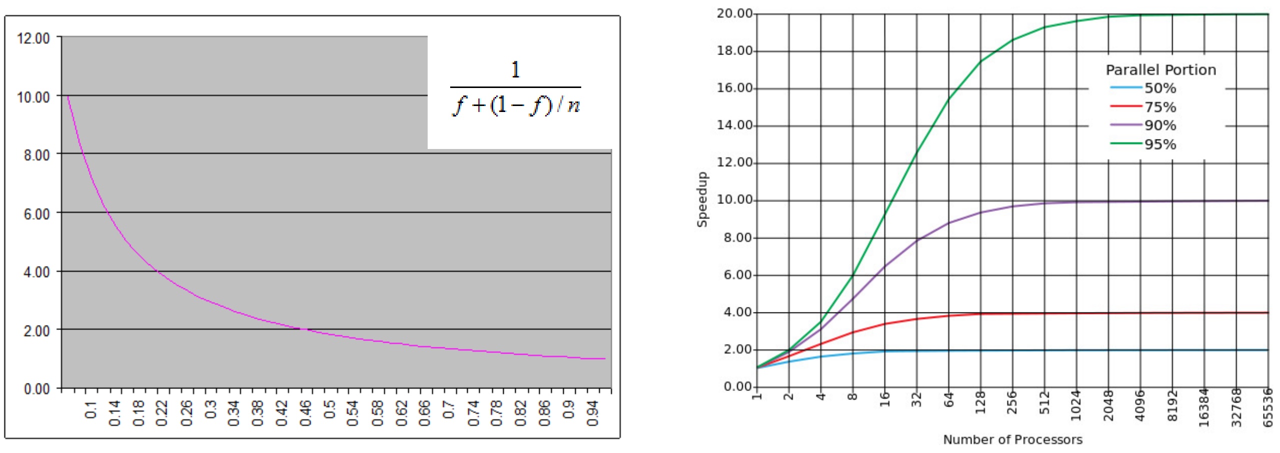


Fig. 5.2:71 – Examples of the theoretical processing increase limits given by Amdahl's Law (Source: [412])

It is clear from the previous illustration that only “embarrassingly parallel” programs [554] may really benefit from massively parallel architectures such as in the work herein, for having *'f'* closer to '1', thus flattening the left curve quite a bit. Since automotive functionalities are highly parallel in terms of individual FDEFs, it is expected that, at least at that level, the whole system will be processing them in a highly unrestrained way. Still regarding this “embarrassingly parallelism”, it must be said that, of course, highly algorithmic sequences are mostly not parallelizable, since there must be a strict inter-lock between the corresponding *“Macros”*. These must then be necessarily affected by the “Strict-Sequence” *“Macro-Modifier”* so that those are never processed in parallel. Furthermore, if certain FDEFs need certain common *“Nodes”* to be synchronized, then they must exist inside the same *“Cellular-ECU”*, so that they get always executed in the desired sequence as well. Nevertheless, in typical automotive functionalities, these non-parallelizable sequences are usually short and not frequent. Also, communications overhead mentioned in [554] is minimized in the parallelization work herein, since all the data always and readily exists in the *GIMy* of each *"Cellular-ECU"*, whereas inter-communications run in the background without any explicit program intervention, thus not directly entering the equation of parallelization speed-up.

The Volkswagen Phaeton W12, employing two completely separated ME7.1.1 ECUs from Bosch in a master-slave configuration is the only attempt to have more than one almost identical unit controlling the same plant. In this case, each one of the ECUs was completely and almost independently responsible for managing half of the engine (6 cylinders). This configuration is similar to the one implemented herein, in what concerns the independence of each "module". Another example of complete separation of tasks among processing nodes, this time in a finer scale and inside the same ECU, would be the MS2.8/MS2.9 ECU series [302], where as in the Phaeton system, only variable values are exchanged among CPUs. This strongly resembles the whole parallel processing scheme of the system developed herein, with its *"Cellular-ECUs"* and their internal *"GIMy"* facilities. For additional notes about this last revisit, see [Att. 52].

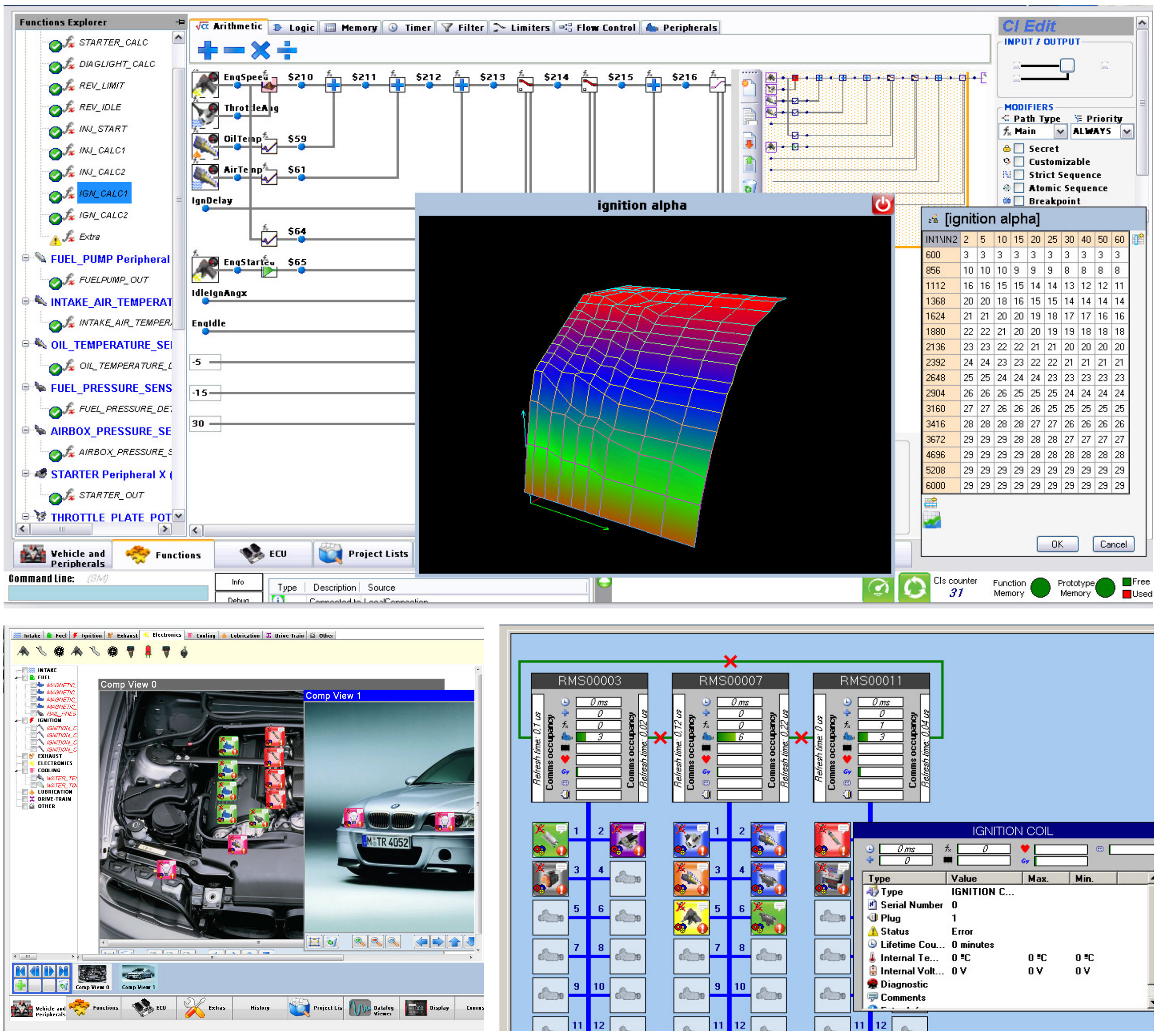
FINAL NOTE: All classical systems rely on multiple technological concepts, which are used inside the blocks in an automotive system. The main problem is a technological diversity greatly increasing overall complexity and thus requiring broad ranges of know-how/manpower to cope with such systems.

From the statements above, ***PITFALL H#73***(Hardware Diversity) is thus effectively eliminated through the use of exact one single technological concept, the "Cellular ECU", for every possible block in the automotive system.

5.3 *"iEditor"* Omnipresent Integrated Management Concept

[Emphasis: integration of all the previous ideas into a single package]

This integrator editor will be the visible software part containing the visual user-interface where everything will be controlled and monitored from. This will be the visible window to the system, comprising all user-interface functionalities needed to develop and deploy automotive systems, such as designing FDEFs, prototyping and debugging FDEFs, setting up peripherals and many others. Fig. 5.3:72 shows a glimpse of this *"iEditor"* and some of its user-interfaces. [Att. 53] then extensively illustrates all its features and lists the pitfalls solved. This software package is controls and monitors every detail of the entire system, without ever needing to switch among different and potentially incompatibility-causing software packages. Every task is at the reach of a click, all inside a single tool, without any further installs, adaptations, workarounds or switches.



[ignition alpha]

IN1\IN2	2	5	10	15	20	25	30	40	50	60
600	3	3	3	3	3	3	3	3	3	3
856	10	10	10	9	9	9	8	8	8	8
1112	16	16	15	15	14	14	13	12	12	11
1368	20	20	18	16	15	15	14	14	14	14
1624	21	21	20	20	19	18	17	17	16	16
1880	22	22	21	20	20	19	19	18	18	18
2136	23	23	22	22	21	21	20	20	20	20
2392	24	24	23	23	22	22	21	21	21	21
2648	25	25	24	24	24	23	23	23	23	23
2904	26	26	26	25	25	25	24	24	24	24
3160	27	27	26	26	26	25	25	25	25	25
3416	28	28	28	28	27	27	26	26	26	26
3672	29	29	29	28	28	28	27	27	27	27
4696	29	29	29	29	28	28	28	28	28	28
5208	29	29	29	29	29	29	29	29	29	29
6000	29	29	29	29	29	29	29	29	29	29

Type	Value	Max.	Min.
Type	IGNITION C...		
Serial Number	0		
Plug	1		
Status	Error		
Lifetime Cou...	0 minutes		
Internal Te...	0 ºC	0 ºC	0 ºC
Internal Volt...	0 V	0 V	0 V
Diagnostic			
Comments			

Fig. 5.3:72 – The "iEditor" at a glance: designing FDEFs and calibrating their parameters

5.4 Support Hardware and Software

[Emphasis: materialization of all the previous ideas]

This section concerns the materialization of all the previous ideas and architectures into a real hardware platform with its software interface. To accomplish this step, several hardware processing elements such das micro-controllers and FPGAs, as well as their software development environments had to be chosen. These materialize the complete *"Cellular-ECU"* hardware platform. On top of that, a Windows software development had to be chosen to materialize the *"iEditor"* that will be controlling all the hardware components. Ubiquitous USB 2.0 was chosen as the main communications path between the hardware platform and the Windows PC, as shown in Fig. 5.4:73, also because there are many easily available commercial modules using this communications protocol.

Next, these components and tools are going to be presented to some detail level next, to introduce the work done in the implementation that followed these choices. It must be noted that these implementations had to count with relatively very little available manpower, development time and financial capacity. Therefore, everything was chosen such as to allow the fastest progress with as little unnecessary hassle as possible. Optimization was not a goal at this stage.

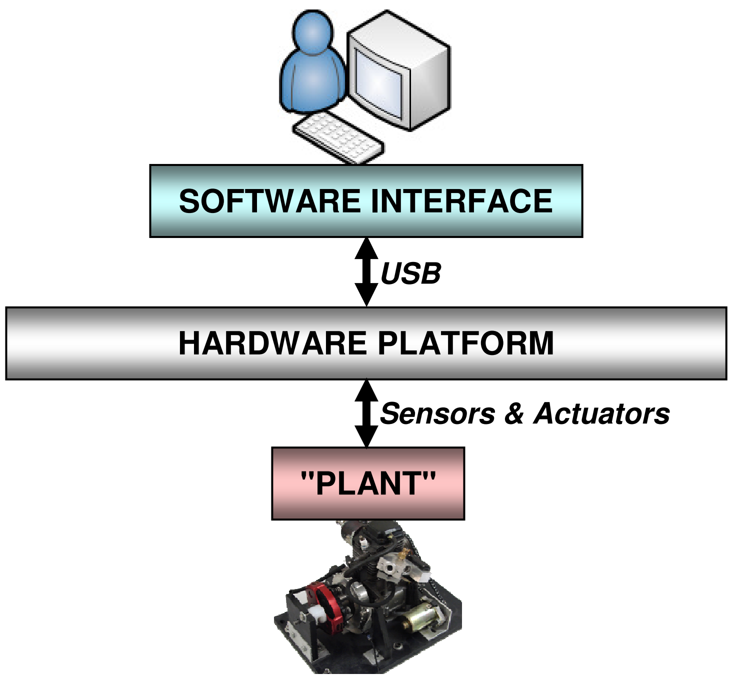


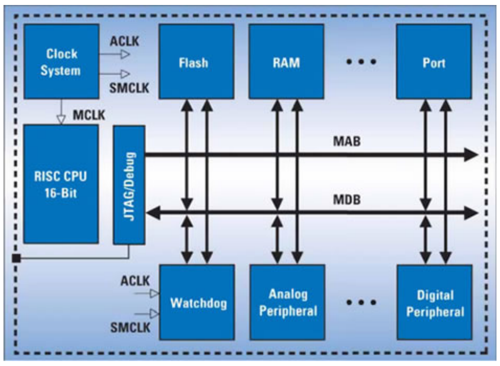
Fig. 5.4:73 – Global abstract "ECU2010" structure with all its entities and communications paths.

5.4.1 MSP430 Microcontrollers

[Emphasis: simplicity, orthogonality, homogeneity, straightforward]

As already mentioned in the previous chapter, the MSP430 microcontroller [310] from Texas Instruments was chosen for its simplicity and straightforward internal architecture, to avoid any problems related with the microcontroller itself, thus getting to an early prototype as fast as possible without known pitfalls in other microcontrollers. Furthermore, attending MSP430 [310] and also Microchip PIC [540] seminars, besides studying other lower-end microcontrollers such as the AVRs [541] and higher-end such as the ARMs [542], it turned out to be quite clear that only the MSP430 would guarantee the best possible pitfall-avoidance. Fig. 5.4:74 shows the clean and easy-to-grasp internal architecture of this microcontroller, along with the chosen flavour and corresponding hardware board. It contains essentially all that was needed for the first prototype. Furthermore, this microcontroller has a "100% 3D-orthogonal instruction set", that is, any instruction can be used with any address and with any of the available addressing modes [365], as illustrated in Fig. 5.4:75. This feature was inherited and enhanced from the technological ancestor PDP11 processor [543] as a fully intended way of differentiating from all other microcontrollers in the market. Additionally, all MSP430 flavours have exactly the same processing core, so that all of them are 100% language-compatible. These last important characteristics further avoid undesired pitfalls at this early prototyping stage, as even mentioned in the pitfalls lists presented in Chapter 3. On the other hand, the AVRs do not have this clean internal structure, the PICs are incompatible among flavours and follow highly non-standard programming models, the ARMs are obviously far too complex to learn quickly. Finally, FPGAs would be too complex to start with, at a point where it is only intended to get quick results and enhancement ideas.

Even if it turned out to be clear quite early that its processing speed would not be enough for real-world applications, the secondary purpose of building a straight-forward hardware platform was to guarantee that first results would be available quite soon in the whole project time-line. The primary purpose was, from the beginning, to guarantee a soon enough decision timing, regarding technology changes, in order to achieve the final results. In other words, knowing quite early if the basic *"ECU2010"* ideas would work, including most of the *"LIVE"* handling features of Tab. 5.1:3. Also, bearing in mind that these first results had to be achieved until the Milestone 1, that is, in only 6 months time (the first month has been used up in training and learning to deal with the chosen software and hardware packages), this seemed to be the right choice at the time.



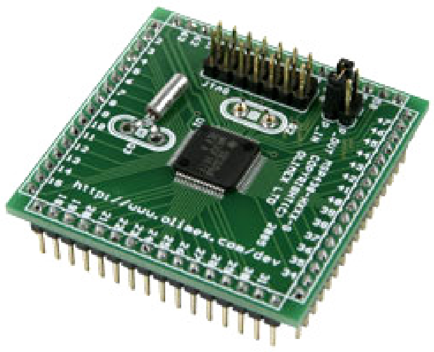


Fig. 5.4:74 – Functional block-diagram of the used MSP430F1611 microcontroller kits (source: Olimex [544])

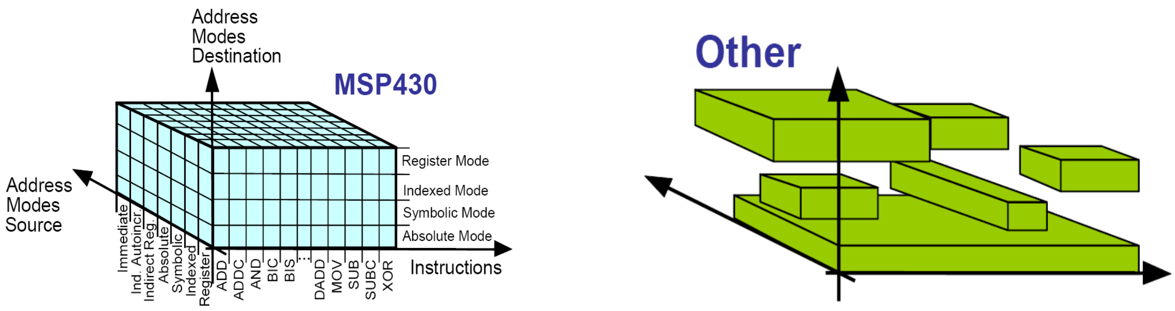


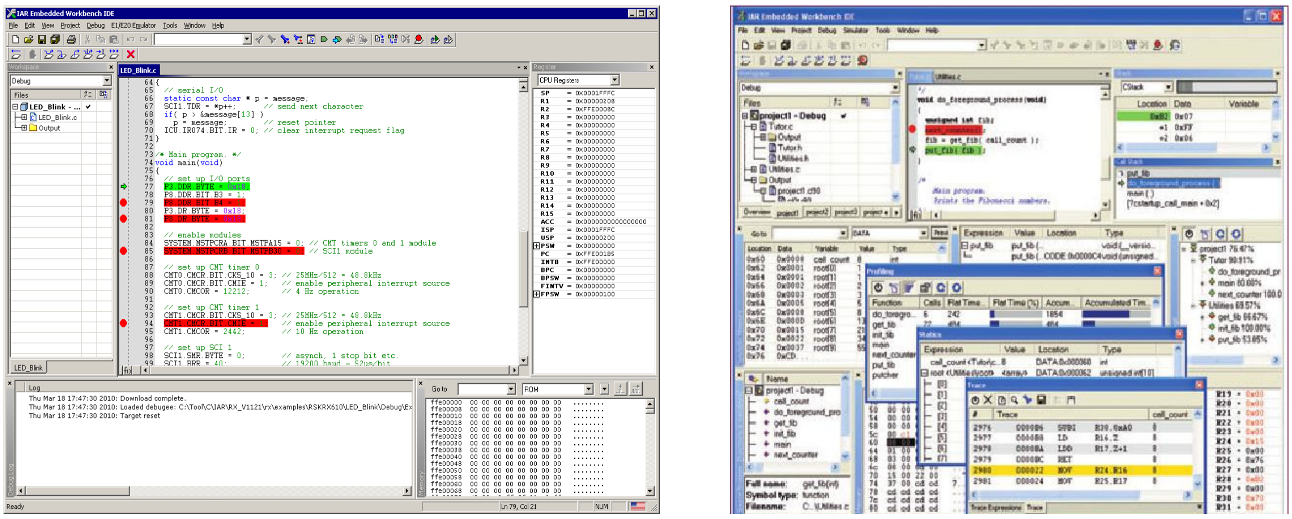
Fig. 5.4:75 – Comparison between the MSP430 100% orthogonal MCU and other MCUs (source: [365])

This microcontroller was used in all the analog parts of both sensor and actuator peripherals. After the preliminary prototype, this microcontroller was recognized to be too slow for the digital processing part and was later replaces by FPGAs. Nevertheless, it was continued to be used in the MSP430 was still used in all mixed analog and digital end-hardware parts for both sensor and actuator peripherals.

5.4.2 IAR Embedded Workbench for MSP430

[Emphasis: best match for the MSP430's quality]

The IAR Embedded Workbench [291] programming environment for the MSP430 was chosen based on the evaluation of other similar programming packages and reading of many customer feedbacks on the Internet as well. The IAR package was clearly the best and most advanced choice, although also being the most expensive as well. It was clear that all other alternatives had inadmissible problems, bugs and limitations that would cause great trouble during development of the *"ECU2010"* system. Fig. 5.4:76 shows the integrated development environment of this chosen package. It allows full desired simulation, debugging and inspection procedures. Furthermore, since everything is fully integrated and readily available at all time without great effort, turnaround delays and other wastes of time are reduced to a minimum, therefore allowing the ideas to be rapidly developed and implemented.



```
{
  // serial I/O
  static const char * p = message;
  SCI1.TDR = *p++;       // send next character
  if( p > &message[13] )
    p = message;         // reset pointer
  ICU.IR074.BIT.IR = 0; // clear interrupt request flag
}

/* Main program. */
void main(void)
{
  // set up I/O ports
  P3.DDR.BYTE = 0x18;
  P8.DDR.BIT.B3 = 1;
  P8.DDR.BIT.B4 = 1;
  P3.DR.BYTE = 0x18;
  P8.DR.BYTE = 0x18;

  // enable modules
  SYSTEM.MSTPCRA.BIT.MSTPA15 = 0; // CMT timers 0 and 1 module
  SYSTEM.MSTPCRB.BIT.MSTPB30 = 0; // SCI1 module

  // set up CMT timer 0
  CMT0.CMCR.BIT.CKS_10 = 3; // 25MHz/512 = 48.8kHz
  CMT0.CMCR.BIT.CMIE = 1;   // enable peripheral interrupt source
  CMT0.CMCOR = 12212;       // 4 Hz operation

  // set up CMT timer 1
  CMT1.CMCR.BIT.CKS_10 = 3; // 25MHz/512 = 48.8kHz
  CMT1.CMCR.BIT.CMIE = 1;   // enable peripheral interrupt source
  CMT1.CMCOR = 2442;        // 10 Hz operation

  // set up SCI 1
  SCI1.SMR.BYTE = 0;        // asynch, 1 stop bit etc.
  SCI1.BRR = 40;            // 19200 baud - 52us/bit
```

```
SP     = 0x0001FFFC
R1     = 0x00000208
R2     = 0xFFE0008C
R3     = 0x00000000
R4     = 0x00000000
R5     = 0x00000000
R6     = 0x00000000
R7     = 0x00000000
R8     = 0x00000000
R9     = 0x00000000
R10    = 0x00000000
R11    = 0x00000000
R12    = 0x00000000
R13    = 0x00000000
R14    = 0x00000000
R15    = 0x00000000
ACC    = 0x0000000000000000
ISP    = 0x0001FFFC
USP    = 0x00000200
PSW    = 0x00000000
PC     = 0xFFE001B5
INTB   = 0xFFE00000
BPC    = 0x00000000
BPSW   = 0x00000000
FINTV  = 0x00000000
FPSW   = 0x00000100
```

```
Thu Mar 18 17:47:30 2010: Download complete.
Thu Mar 18 17:47:30 2010: Loaded debugee: C:\Tool\C\IAR\RX_V1121\rx\examples\RSKRX610\LED_Blink\Debug\Ex
Thu Mar 18 17:47:30 2010: Target reset
```

```
ffe00000  00 00 00 00 00 00 00 00   ........
ffe00008  00 00 00 00 00 00 00 00   ........
ffe00010  00 00 00 00 00 00 00 00   ........
ffe00018  00 00 00 00 00 00 00 00   ........
ffe00020  00 00 00 00 00 00 00 00   ........
ffe00028  00 00 00 00 00 00 00 00   ........
ffe00030  00 00 00 00 00 00 00 00   ........
ffe00038  00 00 00 00 00 00 00 00   ........
ffe00040  00 00 00 00 00 00 00 00   ........
ffe00048  00 00 00 00 00 00 00 00   ........
ffe00050  00 00 00 00 00 00 00 00   ........
ffe00058  00 00 00 00 00 00 00 00   ........
ffe00060  00 00 00 00 00 00 00 00   ........
ffe00068  00 00 00 00 00 00 00 00   ........
```

Fig. 5.4:76 – "C" language development interface of the IAR Embedded Workbench package (source: IAR [545])

5.4.3 FPGAs from XILINX (*"Cellular ECU"*)

[Emphasis: another "best" ingredient]

The choice for configurable hardware chips or Field-Programmable Gate-Arrays (FPGAs) was the clear need, after the first prototype was built. Right there, it was detected that the needed processing speed would only be achieved with hard-wired circuits and not software. Freely programmable logic blocks, freely programmable interconnections between those logic blocks and the I/O blocks (see Fig. 5.4:77) are the ideal playground for the implementation of the new ideas developed in this chapter, contrary to the fixed hardware nature of standard microcontrollers and such. Anyway, the use of FPGAs in many practical system is a given fact since some years now. Examples are all around ([11]

[95] [98] [118] [224] [265] [268] [269]), all of them benefit from the raw hardware speed increase and virtually all of them delegate the lowest-level repetitive tasks to that FPGA hardware, such as well present in the MS5 [10].

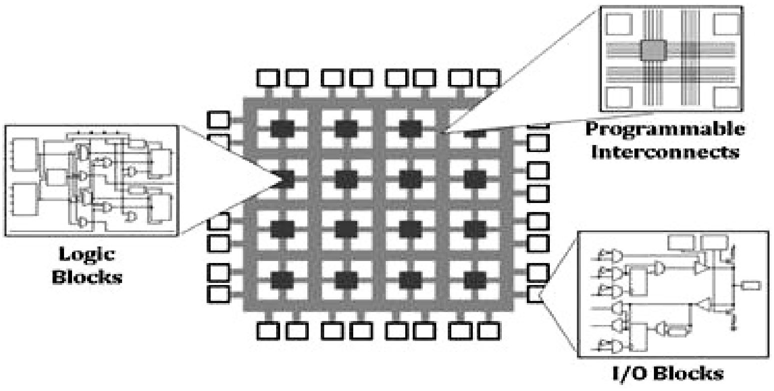


Fig. 5.4:77 – Inner nature of an FPGA chip

The choice for the XILINX FPGA chips [321] [327] was decided after some research on available manufacturers such as ACTEL, ALTERA and LATTICE. XILINX presented the best hardware/software package and is currently regarded as the main FPGA supplier in the world with over 50% market share. The specific chip flavour chosen was the XC3S1600E with a total of 1.6 mega-gates inside. As shown in Fig. 5.4:78 a suitable development kit solution was also found within DIGILENT [320], so that the package was very well rounded up. These boards had all the needed components such as RAM, FLASH, connectors, power-supply, optional debugging display, etc.

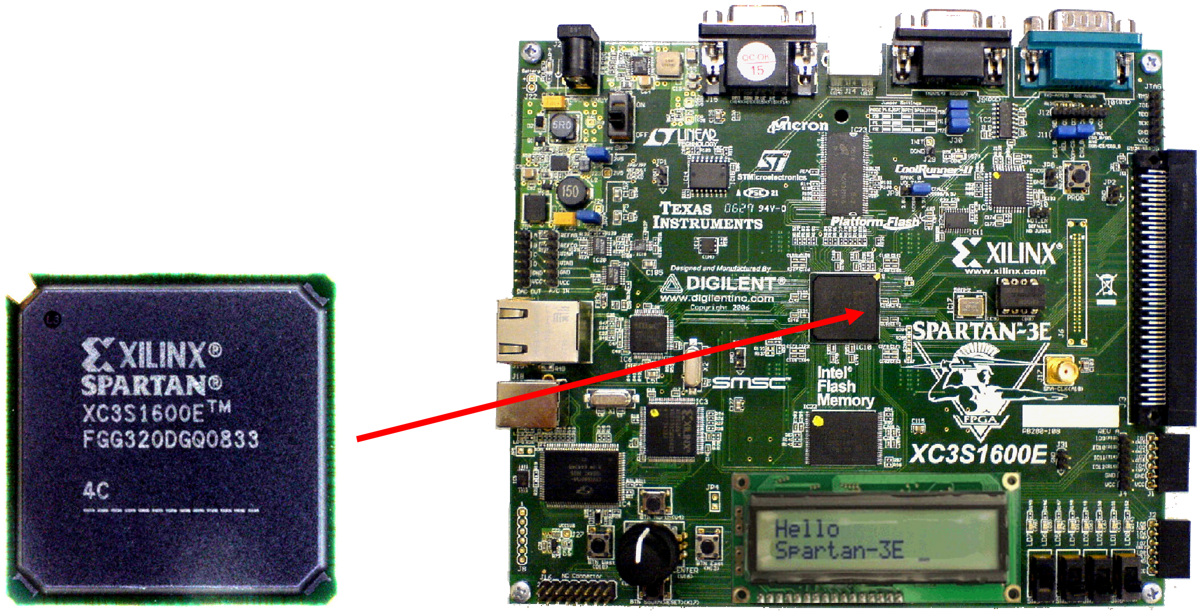


Fig. 5.4:78 – Development kit "SPARTAN-3E" with the XC3S1600E (source: DIGILENT [326])

By combining these modern and advanced development boards with two custom boards and one commercial USB board (see all three in Fig. 5.4:79) and adding an MMC FLASH card, the *"Cellular ECU"* unit is born and depicted in Fig. 5.4:80. Depending on its particular usage, this unit was then replicated a total of 17 times all around the final *"ECU2010"* system, 3 times in the central ECU, 2 times in the wireless datalogger and 12 times in the various peripherals.

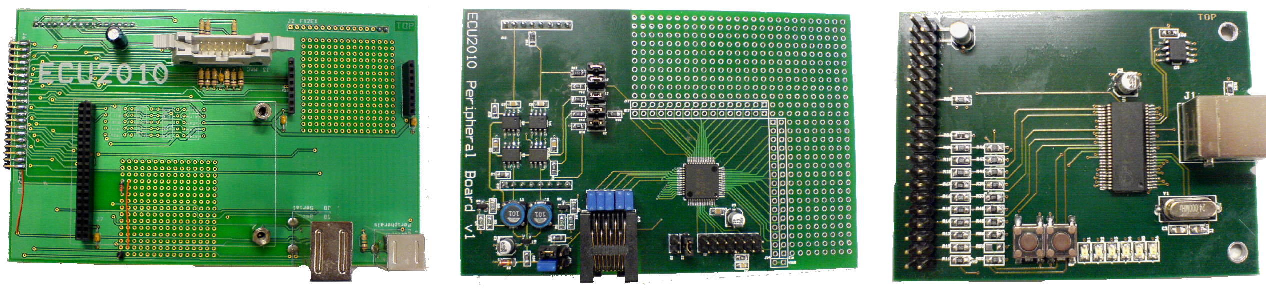


Fig. 5.4:79 – Custom-made accessory boards (not to scale): "ECU2010", MSP430 peripherals and Cypress USB

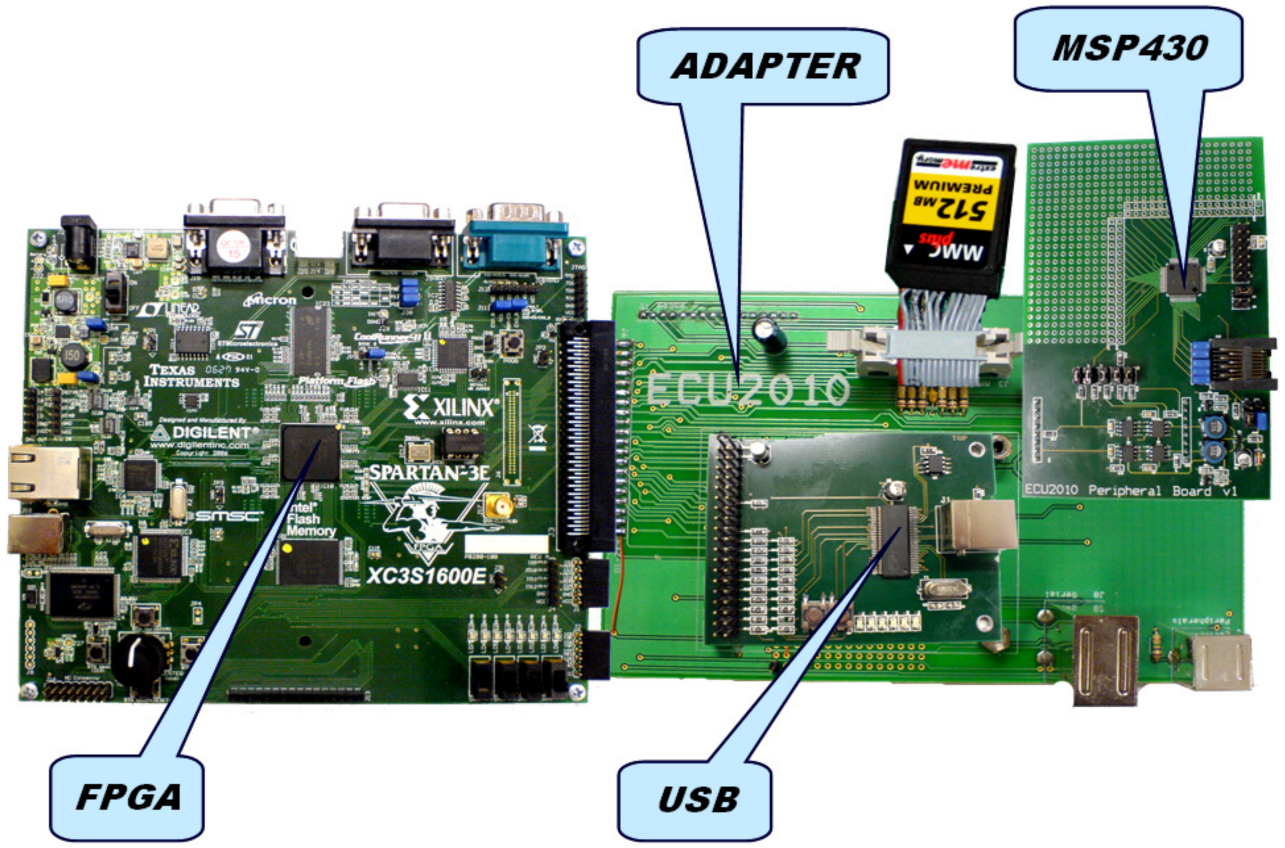


Fig. 5.4:80 – Completely assembled "Cellular ECU" unit with all the parts needed

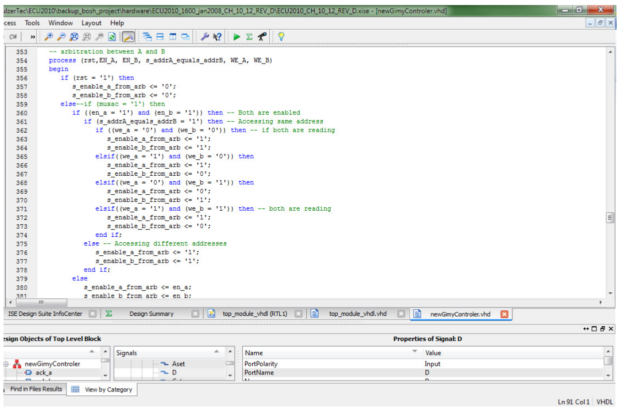
For more detailed information about the used chips, components and evaluation boards (one of them custom made), please visit the following links:

- XILINX ***XILINX FPGAs*** → go to www.xilinx.com
- DIGILENT ***DIGILENT DevKits*** → go to www.digilentinc.com
- CYPRESS ***CYPRESS USB boards*** → go to www.cypress.com
- ***"ECU2010", MSP430 & Cypress USB boards*** → custom designed during the project

5.4.4 ISE Design Suite for FPGA

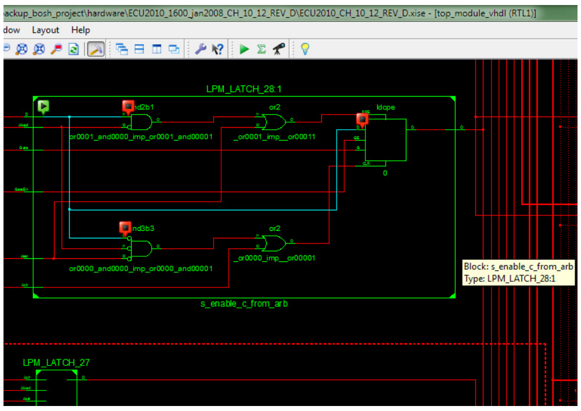
[Emphasis: 100% compatibility and flexibility]

This software tool [324] also from XILINX was adopted with the purpose of configuring the chosen XILINX FPGAs. Fig. 5.4:81 shows two screenshots from this design tool, while containing the work done herein in VHDL. This tool allows to fully exploit the power of those FPGAs, including debugging capabilities through special configurations added to the bitstream. These extra configurations then send internal FPGA signal data back to the ISE tool (left of Fig. 5.4:82), displaying them in form of classical oscilloscope images (right of Fig. 5.4:82). The ISE even includes a simulator for simulating FPGA design configurations before downloading them into the FPGA hardware kits. Mainly because this software tool comes from the same manufacturer as the chosen FPGAs, but also because it has a complete set of tools to work with, this was the choice. Additionally, this software has won prizes [312] among its peers. All "programming" in the work herein was done in VHDL.



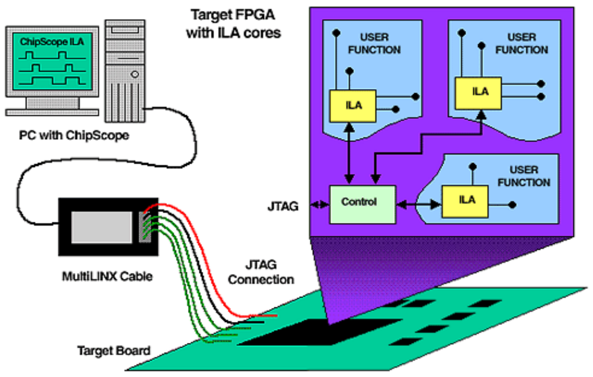
```
-- arbitration between A and B
process (rst,EN_A, EN_B, s_addrA_equals_addrB, WE_A, WE_B)
begin
   if (rst = '1') then
      s_enable_a_from_arb <= '0';
      s_enable_b_from_arb <= '0';
   else--if (muxac = '1') then
      if ((en_a = '1') and (en_b = '1')) then -- Both are enabled
         if (s_addrA_equals_addrB = '1') then -- Accessing same address
            if ((we_a = '0') and (we_b = '0')) then -- if both are reading
               s_enable_a_from_arb <= '1';
               s_enable_b_from_arb <= '1';
            elsif((we_a = '1') and (we_b = '0')) then
               s_enable_a_from_arb <= '1';
               s_enable_b_from_arb <= '0';
            elsif((we_a = '0') and (we_b = '1')) then
               s_enable_a_from_arb <= '0';
               s_enable_b_from_arb <= '1';
            elsif((we_a = '1') and (we_b = '1')) then -- both are reading
               s_enable_a_from_arb <= '1';
               s_enable_b_from_arb <= '0';
            end if;
         else -- Accessing different addresses
            s_enable_a_from_arb <= '1';
            s_enable_b_from_arb <= '1';
         end if;
      else
         s_enable_a_from_arb <= en_a;
         s_enable_b_from_arb <= en_b;
```

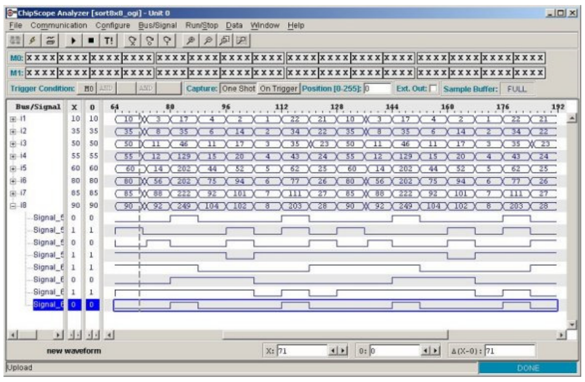
ulzerTec\ECU2010\backup_bosh_project\hardware\ECU2010_1600_jan2008_CH_10_12_REV_D\ECU2010_CH_10_12_REV_D.xise - [newGimyControler.vhd]
Tools Window Layout Help
ISE Design Suite InfoCenter
Design Summary
top_module_vhdl (RTL1)
top_module_vhdl.vhd
newGimyControler.vhd
Design Objects of Top Level Block
Properties of Signal: D
newGimyControler
ack_a
Signals
Aset
D
Name
Value
PortPolarity
Input
PortName
D
Find in Files Results
View by Category
Ln 91 Col 1
VHDL



ackup_bosh_project\hardware\ECU2010_1600_jan2008_CH_10_12_REV_D\ECU2010_CH_10_12_REV_D.xise - [top_module_vhdl (RTL1)]
dow Layout Help
LPM_LATCH_28:1
nd2b1
or2
ldcpe
or0001_and0000_imp_or0001_and00001
_or0001_imp__or00011
nd3b3
or2
or0000_and0000_imp_or0000_and00001
_or0000_imp__or00001
s_enable_c_from_arb
Block: s_enable_c_from_arb
Type: LPM_LATCH_28:1
LPM_LATCH_27

Fig. 5.4:81 – XILINX ISE screenshots of the "ECU2010" project implemented (Left: VHDL; Right: gates & block)





ChipScope Analyzer [sort8x8_ogi] - Unit 0
File Communication Configure Bus/Signal Run/Stop Data Window Help
Trigger Condition:
Capture: One Shot On Trigger Position [0-255]: 0
Ext. Out:
Sample Buffer: FULL
Bus/Signal
new waveform
Upload
DONE

Fig. 5.4:82 – Left: ChipScope operation mechanism; Right: signals retrieved (sources: IAR [325], XILINX [324])

5.4.5 Microsoft .NET Support Software

[Emphasis: most flexible programming package for Windows around]

Microsoft Visual Studio 2005 [292] for Windows was chosen for developing the *"iEditor"* based on its undisputed state-of-the-art status, regarding ".NET" programming. "C#" [71] was chosen as the programming language inside this package. Advanced debugging possibilities, online code patching thanks to the background compiler, ".NET" extensions for PDAs and general easy-handling, wrapped up this final choice. This package has already been extensively detailed in Chapter 2 and can be viewed from Fig. [Att. 6]:33 to Fig. [Att. 6]:39. The possibility of PDA programming with little changes to the source-code is interesting if the *"iEditor"* should be also transferred to these smaller devices, allowing creating cockpit displays and generally light hand-equipment for developers and customers. This tool was chosen after looking for others on the market, especially Borland's Delphi software [546], but being its current version (year of 2006) a very poor quality one of the factors to not using it. Furthermore, Delphi was not fully supporting the ".NET" technology at the time, besides supporting it non-natively.

Page intentionally left blank

Page intentionally left blank

"We can't solve problems using the same kind of thinking we used when we created them"
– Albert Einstein

CHAPTER 6

"ECU2010" – Prototype, DEMO and Results

6.1 Early Work & First Prototype

[Emphasis: preliminary software and hardware testing]

Implementation started in September 2006 with codename *"ECU2010"*, as a joint project between Bosch Motorsport, the University of Aveiro and KulzerTEC. "2010" comes from the last milestone initially planned to take place in 2010: have the system developed herein as part of an experimental test-system inside a real race-car. Even though much time had been spent on the first prototype (Fig. 6.1:1) that turned out to be way too slow for our goal, it already proved some points and ideas were working well. Detailed information about this early phase and preliminary results can be seen in [Att. 54]. The initially used micro-controller based technology suffered a significant shift, implying considerable human and technical effort in order to catch up with the proposed time-frames. This was necessary to be able to present a fully operational FPGA-based and real working engine based proof-of-concept at the final DEMO in Germany.

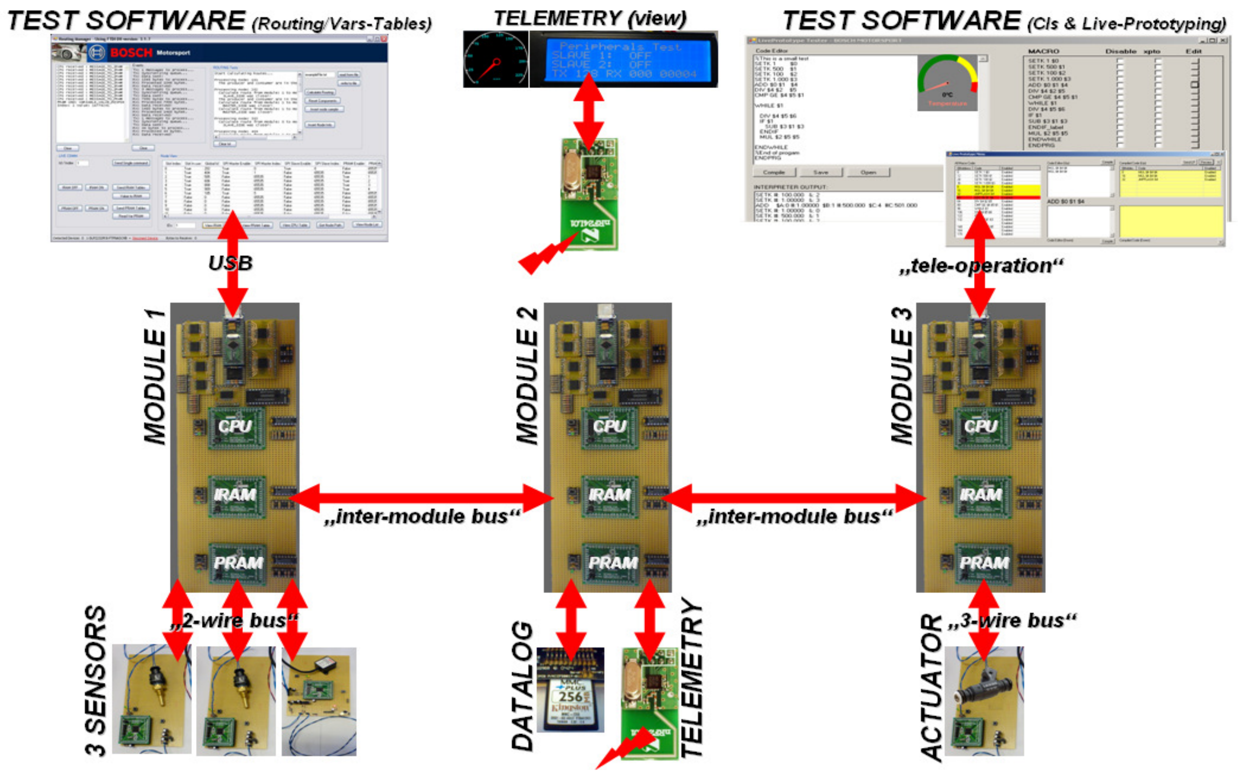


Fig. 6.1:1 – First "ECU2010" prototype complete with "Macros Processing", peripherals, datalogging and telemetry

6.2 Final DEMO

[Emphasis: final shakedown and show-off of the work done, at Bosch Motorsport in Germany]

The hardware and software used in the second, final DEMO prototype, is detailed as follows. Both software *"iEditor"* and the FPGA-based hardware were demonstrated on site, at the University of Aveiro, Portugal. On November 28th, 2008, this DEMO was virtually taken to the Bosch Motorsport headquarters in Markgröningen, Germany, by means of the *"Tele-Operation"* remote view feature of this platform (a 2.300km distance, covered through an internet connection). There, it was fully demonstrated by remote-control up to every single handling detail, except for online datalogging due to mere bandwidth reasons. This demonstration was evaluated as a complete success by both parties involved and displayed zero unexpected failures.

Before going to the final software, hardware and handling DEMO details and results, it must be stressed out that the developed work did not intend to provide for a final system ready to be equipped into a real racing car already. This work only proved the concept exposed in the abstract, that is, having a visual editor manipulating visual operations, which are then directly processed by underlying hardware, without any standard compilation/translation layers and with a minimum of pitfalls. To do this, a special combination of software and hardware components was used and applied to the automotive scene in general, having some motorsport-specific elements as well. Qualitative *"Easy-Handling"* was the desired result, besides some quantitative capacity and speed measurements as well.

On the last few days, we were trying to bring the engine to life, since everything in the "ECU2010" system was apparently operating perfectly fine. But this did not happen before a factor-of-2 fuel calculation mistake made by myself, with the engine already wanting to rev higher than the starter-speed. The engine was receiving a too lean of a mixture (Lambda=1.8) and combustion was only partially happening inside the cylinder, since we could already smell the partially burned gasoline. The very first reaction to that acknowledgment was, of course, leaving everything running (the complete system was never reset in any way in this phase) and after a very quick fuel-doubling for the mixture to get richer (using *"Live-Generation"* to force that change), the very next starter activation immediately led to the engine revving up normally and maintaining its speed throughout filming. After that anxiously awaited success, finally some tests, experiments and evaluations were made, as well as some film and photographic documentation of the various *"Live"* and other features operating within the system.

Fig. 6.2:2 shows the *"ECU2010"* (aka *"Revolutionary Motorsports"*) team right after the engine (behind the team with the exhaust pipe reaching to the outside of the window above) worked perfectly fine for a couple of minutes. After 2 years and 2 months of hard work, at 22h56 of the 25th of November 2008, the small single-cylinder lawn-mower engine finally came to life after discovering a basic fuel-quantity management functionality error (which was readily corrected through *"Live-Prototyping"*). The engine roared up to idle-speed and accelerated well to the intentionally established limit of 6.000RPM (to avoid unnecessary damage at this stage). All this happened exactly 2 days before members of this team flew to Markgröningen, Germany, to finally be able to attend the official DEMO at Bosch Motorsport on November 28th, 2008, right on planned schedule…

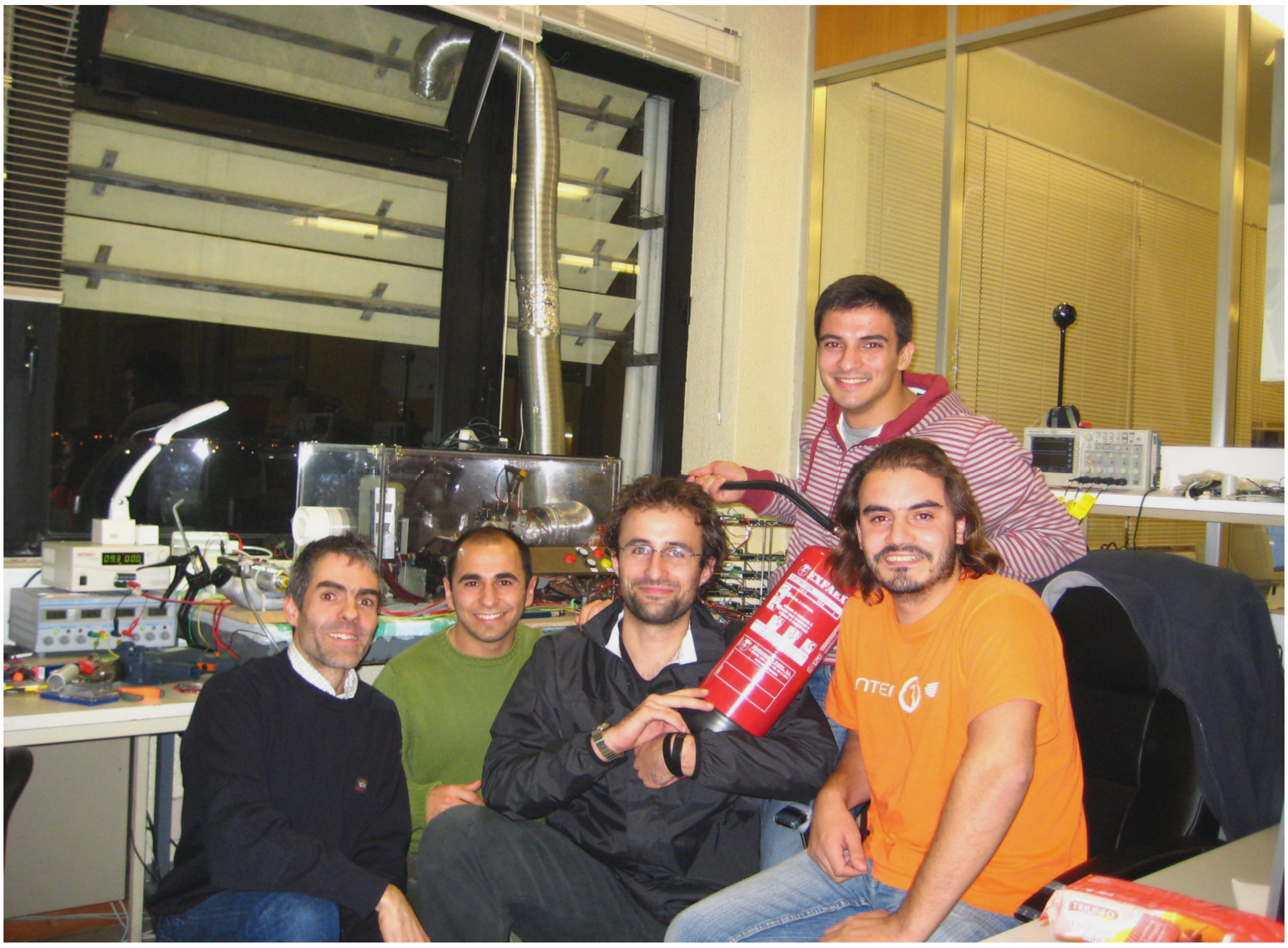


Fig. 6.2:2 – "ECU2010" remaining team just minutes after the first successful start and operation of the engine

6.2.1 Final Software and Hardware

[Emphasis: what was really used at the very end]

At this final stage, all the definitive to use software and hardware components and packages had to be chosen. The only real change relative to the first preliminary prototype was that the main *"Macros"* processing, data hosting and communications hardware was changed from the early MSP430 microcontrollers to XILINX FPGA chips [321] built-in inside development kits from DIGILENT [320], configured through the ISE Design Suite also from XILINX. The MSP430 was still used in analog and digital end-hardware for both sensor and actuator peripherals. Also, the architecture of the whole *"ECU2010"* system was homogenized to comply with the *"Cellular ECU"* idea, so that now the *"ECU2010"* FPGA-based processing modules exist in the central ECU, in the peripherals and in the datalogger. The final materials list thus looks as follows:

- Spartan-3E FPGA [321] development kits [320] for all *"Cellular ECU"* modules
- ISE Design Suite [324] to configure the FPGAs in every *"Cellular ECU"* module
- MSP430 microcontrollers [310] for the analog and digital end-hardware on peripherals
- IAR Embedded Workbench [291] for programming the MSP430 microcontrollers
- Microsoft Visual Studio 2005 [292] for the *"iEditor"* for Windows

The following pictures show the hardware platform that gave life to all the software and engine control algorithms developed, with the ultimate goal of controlling a combustion engine. Fig. 6.2:3 shows the final Parallel Hardware DEMO prototype for the central ECU (also called *"ECU2010"*) with its three parallel-processing *"Cellular-ECU"* modules.

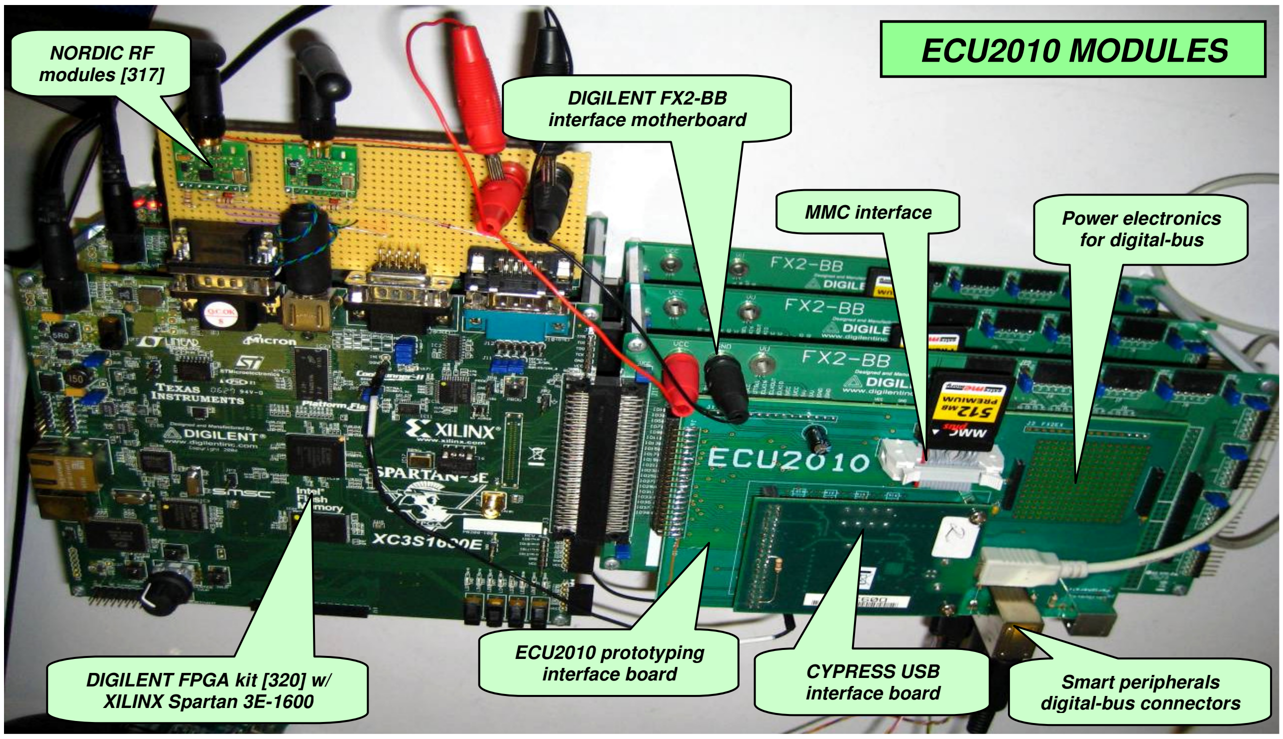

ECU2010 MODULES
NORDIC RF modules [317]
DIGILENT FX2-BB interface motherboard
MMC interface
Power electronics for digital-bus
DIGILENT FPGA kit [320] w/ XILINX Spartan 3E-1600
ECU2010 prototyping interface board
CYPRESS USB interface board
Smart peripherals digital-bus connectors

Fig. 6.2:3 – Parallel Hardware ECU modules with all prototyping boards attached

Fig. 6.2:4 shows the final Wireless Data-Logger DEMO prototype with its radio boards, composed by two *"Cellular-ECU"* modules.

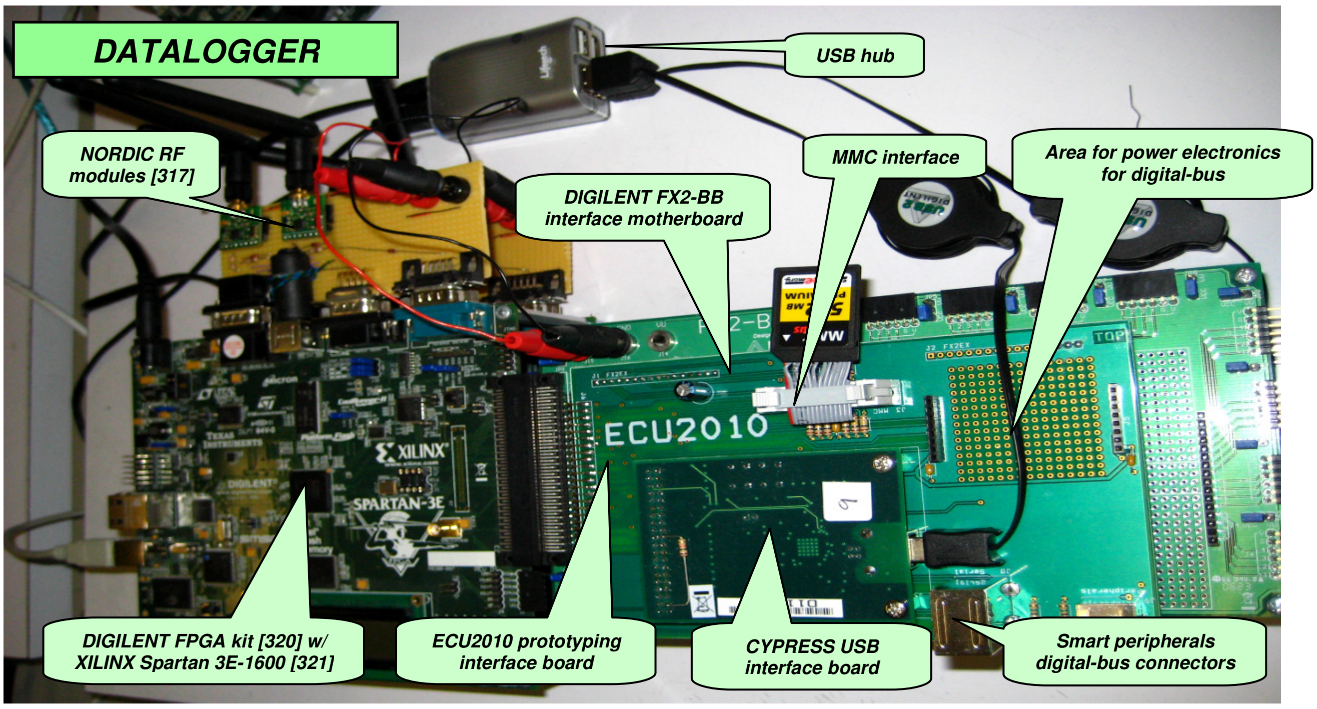

DATALOGGER
USB hub
NORDIC RF modules [317]
MMC interface
Area for power electronics for digital-bus
DIGILENT FX2-BB interface motherboard
DIGILENT FPGA kit [320] w/ XILINX Spartan 3E-1600 [321]
ECU2010 prototyping interface board
CYPRESS USB interface board
Smart peripherals digital-bus connectors

Fig. 6.2:4 – Parallel Wireless Datalogger modules with all prototyping boards attached

Fig. 6.2:5 shows one of the final *"ECU2010"* modules connected to a Smart Peripheral module (also containing a *"Cellular-ECU"*) through the peripherals' digital bus. These are connected to the central *"ECU2010"* through 3 separate digital busses.

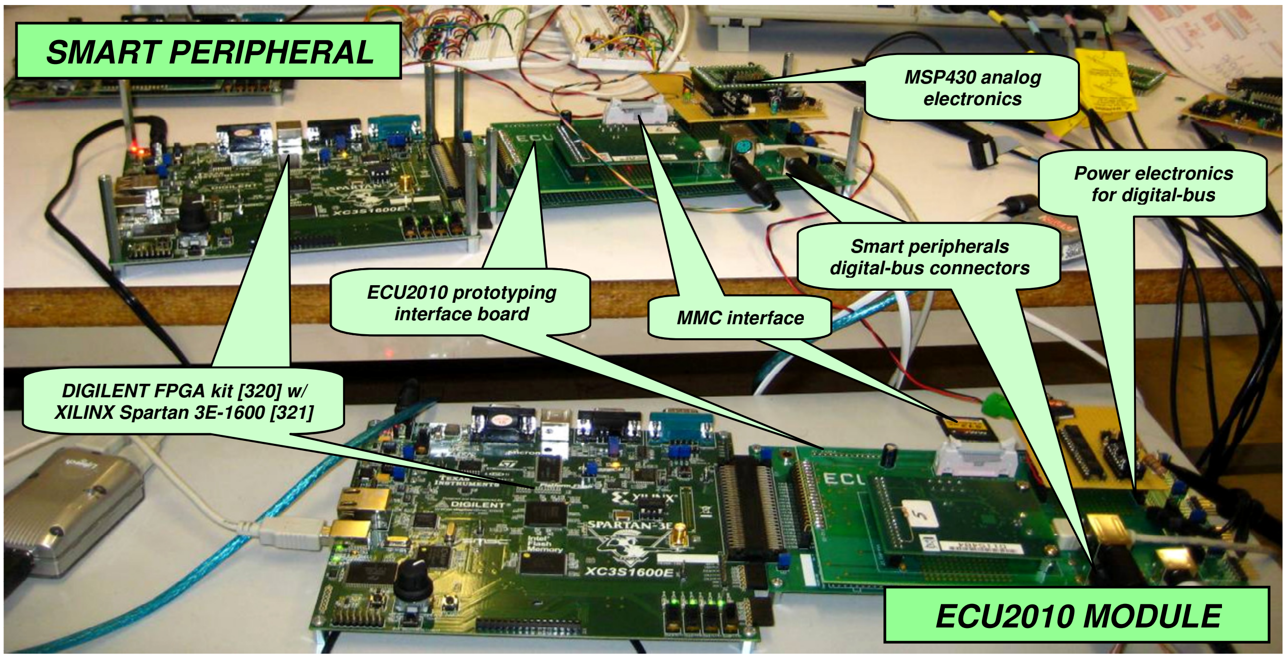

SMART PERIPHERAL
MSP430 analog electronics
Power electronics for digital-bus
Smart peripherals digital-bus connectors
ECU2010 prototyping interface board
MMC interface
DIGILENT FPGA kit [320] w/ XILINX Spartan 3E-1600 [321]
ECU2010 MODULE

Fig. 6.2:5 – Smart Peripheral module with all prototyping boards attached connected to an ECU module

Fig. 6.2:6 shows an example of preliminary engine position input signals and the injection/ignition output signals that are correspondingly generated for the engine.

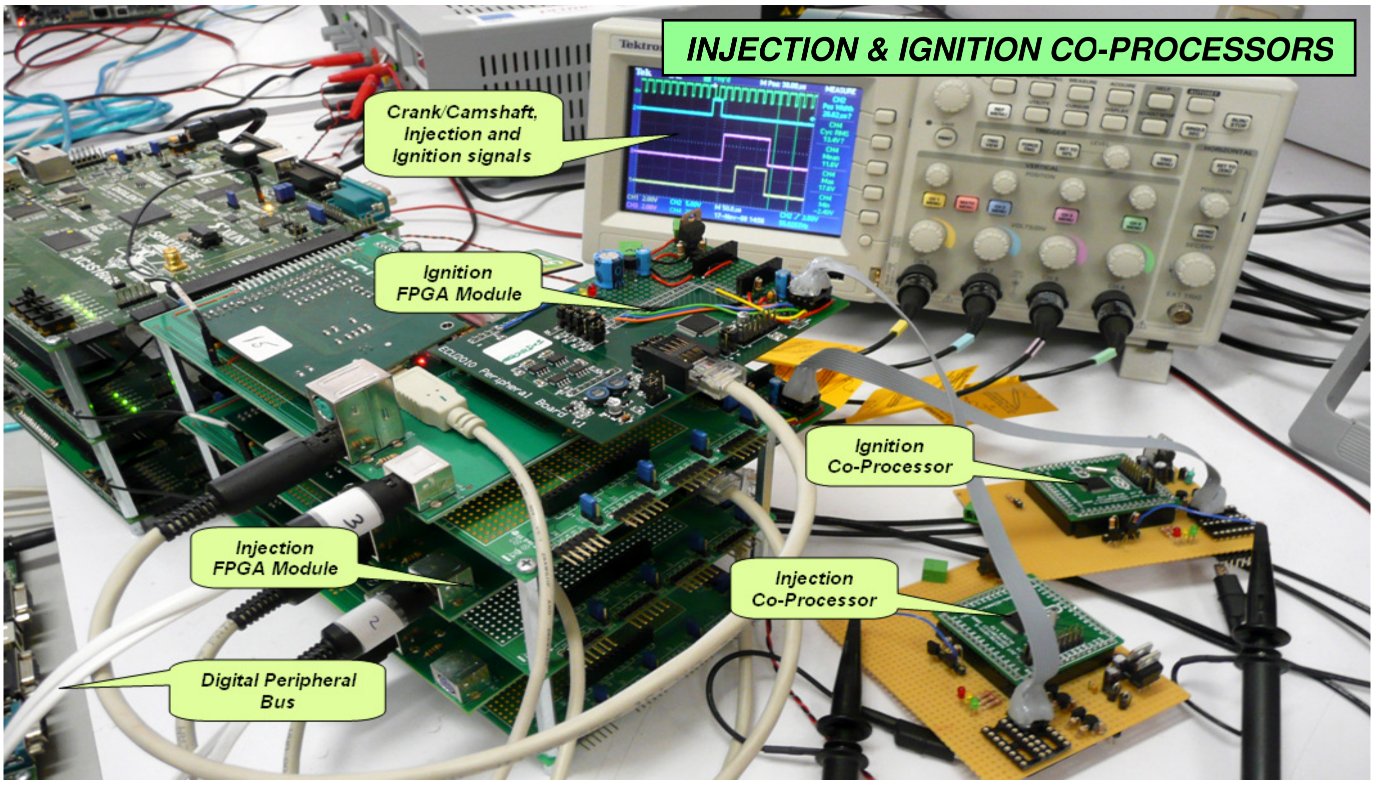


Fig. 6.2:6 – Illustration of active engine position and corresponding injection and ignition signals

TOTAL FINAL DEMO PROTOTYPE NEEDS:

- Central *"ECU2010"* modules: **3**
- Wireless Datalogger modules: **2**
- Smart Peripherals modules: **12**

It must be noted here that a special sub-component, residing inside the *"FDEF-Processor"*, was experimented at this time. This was a *"Macros"* pre-fetcher in that, similarly to classic CPUs, it also fetched the instructions lying ahead of the instruction being currently processed. This may seem as a disruption of the general pitfall-avoiding techniques used throughout the entire work herein, regarding pipelines but it is not: contrary to classic CPUs where instructions take multiple cycles to execute and could therefore potentially disrupt the whole processing context if care is not taken when interrupts get into the middle of such cycles, the *"Macro"*-granularity pre-fetcher implemented here really is nothing more than just that. Furthermore and as expected, only the *"Macros"* are pre-fetched and never the *"Nodes"*, which are fetched on due processing time of those same *"Macros"*. In the implementation herein, only a one-stage pre-fetcher was added to eliminate those delays and, of course, every time a jump happens, the pre-fetcher is ignored and the real next *"Macro"* is fetched instead. Fig. 6.2:7 illustrates the addition of this simple yet effective feature to the original Fig. [Att. 40]:137 from earlier on herein, in form of the gray dotted lines and block. This feature is then located inside the same "LP" block of Fig. 4.5:78.

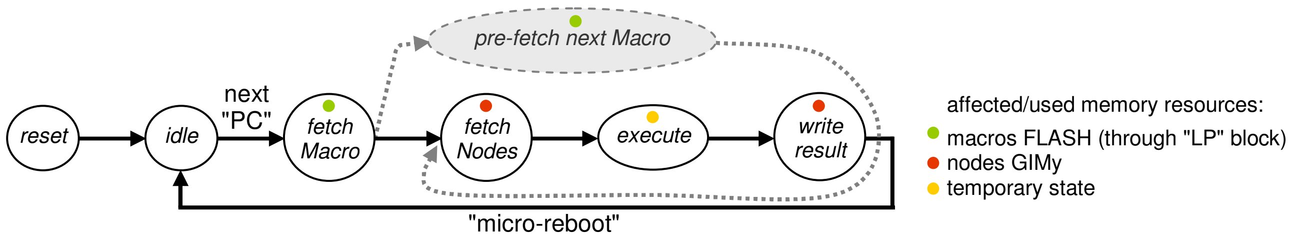


Fig. 6.2:7 - Internal "Macros-Processor" state-machine (spanned tree of possible transitions is the same)

As for the development software system itself, the final *"iEditor"* was basically already mentioned, while all its details can be thoroughly inspected in [Att. 53]. At this point, this *"iEditor"* was already very mature and ready to accomplish all the tasks it was designed for, in this final *"ECU2010"* DEMO scenario.

6.2.2 General Software and Hardware Hints

[Emphasis: some illustrative hints of what was really implemented at the very end]

In this section, a few illustrative and basic hints are unveiled about the final implementations of the some parts of this entire project, just to get a glimpse of what exists in each of its major building blocks. For in-depth descriptions of the related blocks and mechanisms, surrounding details and especially the FPGA/VHDL, USB and .NET implementations, please check the respective references given herein.

Besides these hinted details herein below, general work on the central processing architecture inside the FPGAs, can be further inspected here: [535] [536], while the insides of the *"iEditor"* may be also further seen here: [537] [538].

Fig. 6.2:8 shows just some general details such as diagrams and schematics, to illustrate what has been developed to get the "Smart Peripherals" to work at the end. For an in-depth description of these blocks, especially concerning their schematics and code implementations, please refer to [532] [533] [534].

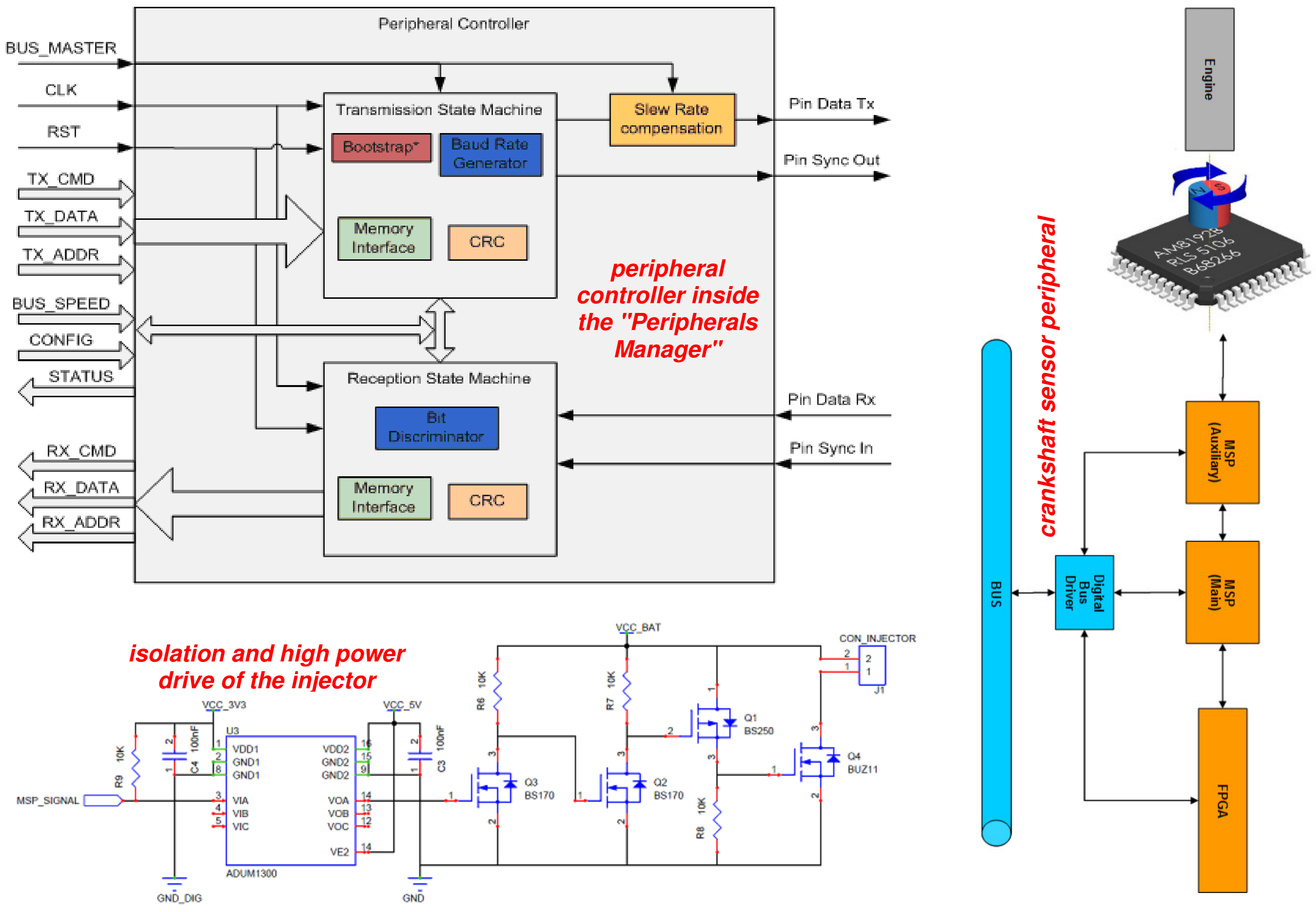


Fig. 6.2:8 – Diagrams and schematics for the peripheral management mechanisms (source: [532] [533] [534])

Fig. 6.2:9 shows just some general details such as diagrams and schematics, to illustrate what has been developed, to get all the necessary blocks of these data-logging mechanisms to work at the end. For an in-depth description of these blocks, surrounding details and especially the FPGA/VHDL and USB implementations, please refer to [529] [530] [531].

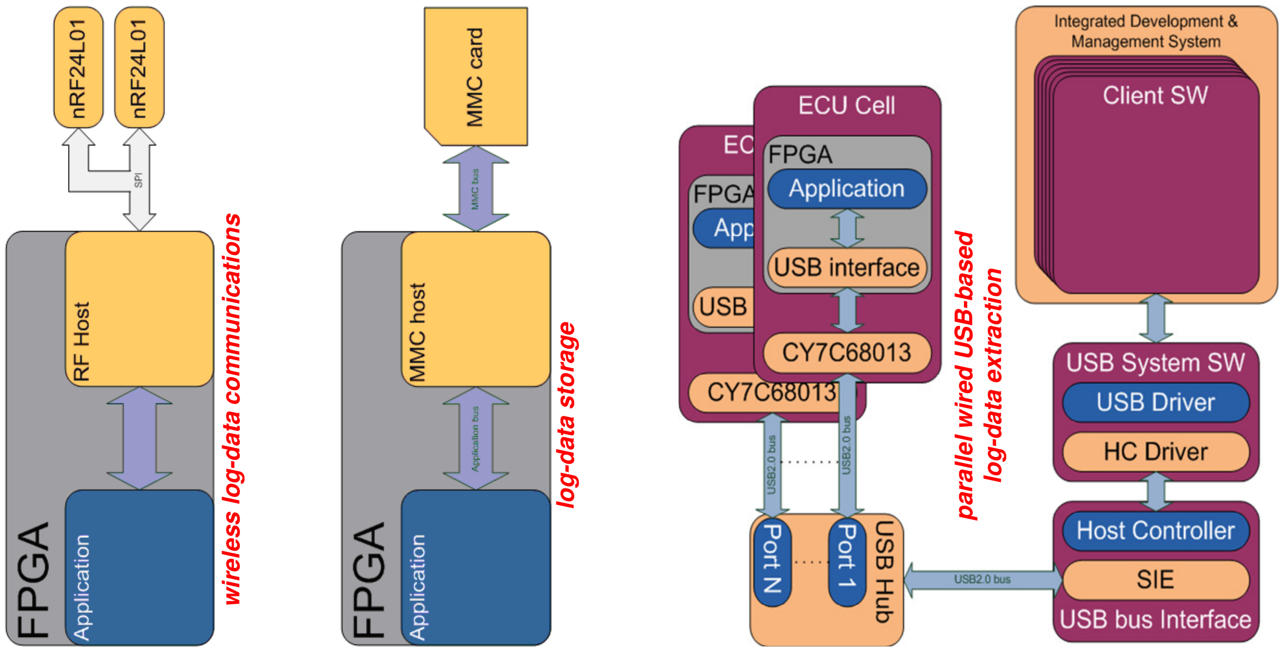


Fig. 6.2:9 – Diagrams for the data-logging mechanisms (source: [529] [530] [531])

6.2.3 Engine Plant & Peripherals

[Emphasis: the smart things that help the engine run]

The engine used in this final DEMO was a mono-cylinder as shown in Fig. 6.2:10. Fig. 6.2:11 shows the test-platform where this engine is mounted. On the left the engine is shown open and with the extra electronic throttle auxiliary construct. On the right, the engine is shown already closed inside a sound-absorbing casing. The exhaust tube and the ventilator can also be seen on both sides. Fig. 6.2:12 shows other views of the engine platform. The characteristics of the engine are as follows:

- **Brand:** *Honda*
- **Original application:** lawn-mower
- **Displacement:** *25 cm^3*
- **Type:** 4-stroke
- **Fuel:** lead-free gasoline

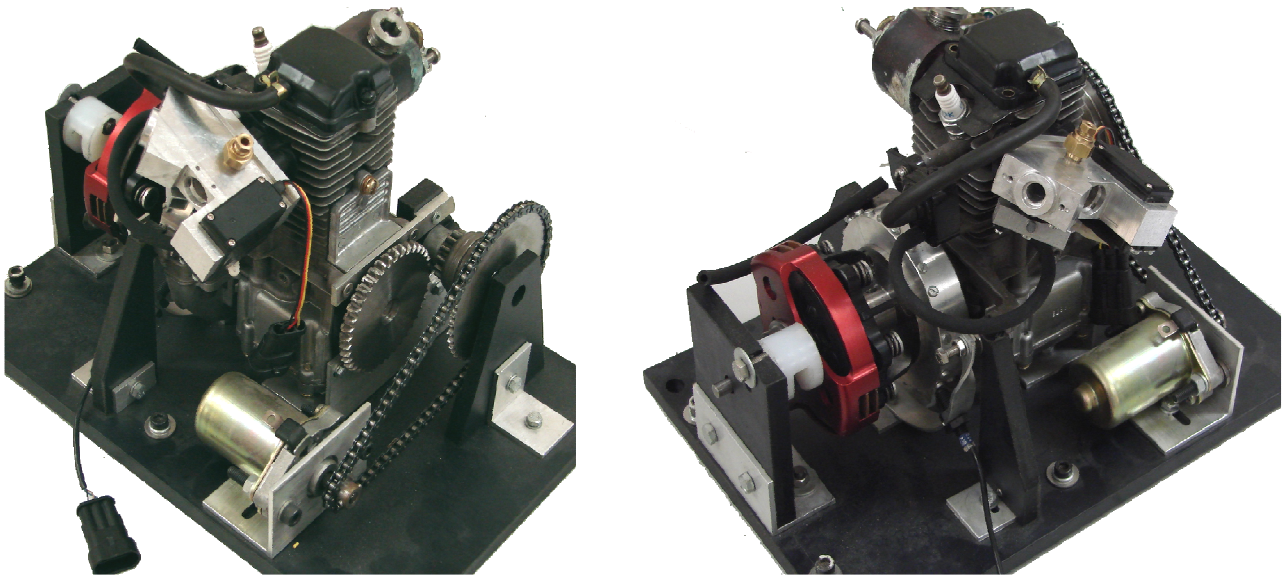


Fig. 6.2:10 – Mono-cylinder Honda test-engine

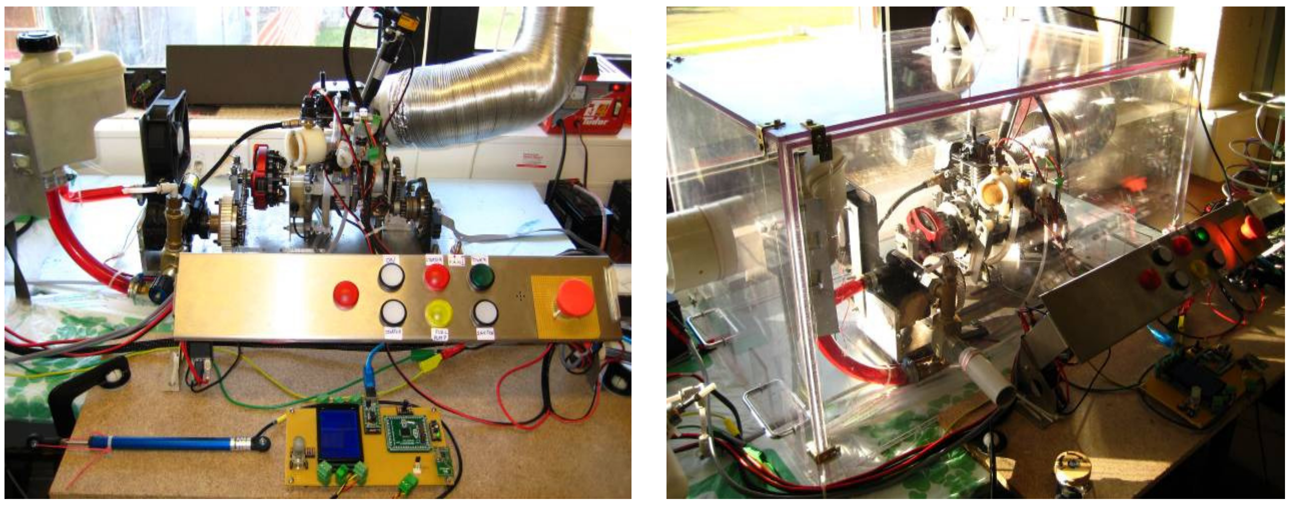


Fig. 6.2:11 – Mono-cylinder engine testing platform

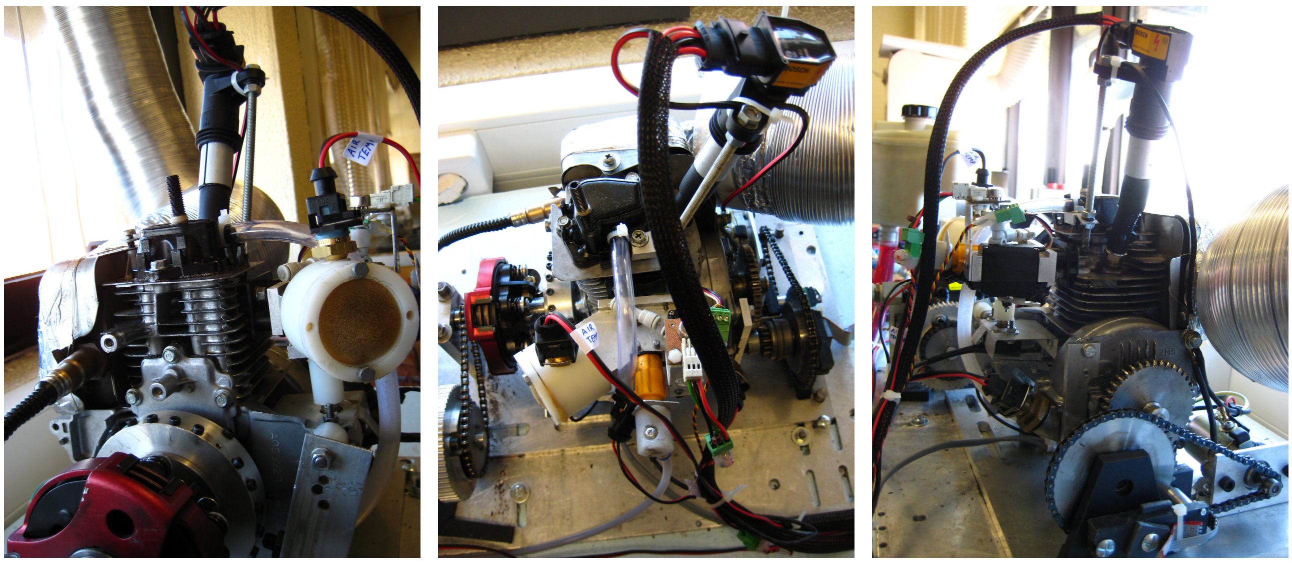


Fig. 6.2:12 – Mono-cylinder engine testing platform

The following images show all sensors and actuators attached to this engine (see Fig. 6.2:13 for the overall view and Fig. 6.2:14 and Fig. 6.2:15 for detailed local views). Each one of these "Smart Peripherals" is independently controlled by dedicated *"Cellular-ECU"* modules, adding up to a total of 12. Fig. 6.2:16 shows a solution based on an innovative way of measuring the absolute engine crankshaft angle and the engine-speed, through a Hall-effect sensor outputting absolute position information through a serial protocol. It also outputs teeth signals through direct pins for electronic usage without any software. This is similar to the teeth signal of normal inductive sensors. Fig. 6.2:17 shows measured signals out of this sensor and the corresponding response of the injection and ignition "Smart Peripherals". This sensor was configured for a resolution of 800 ticks per revolution, that is, less than one degree, which is more than sufficient for efficient engine control.

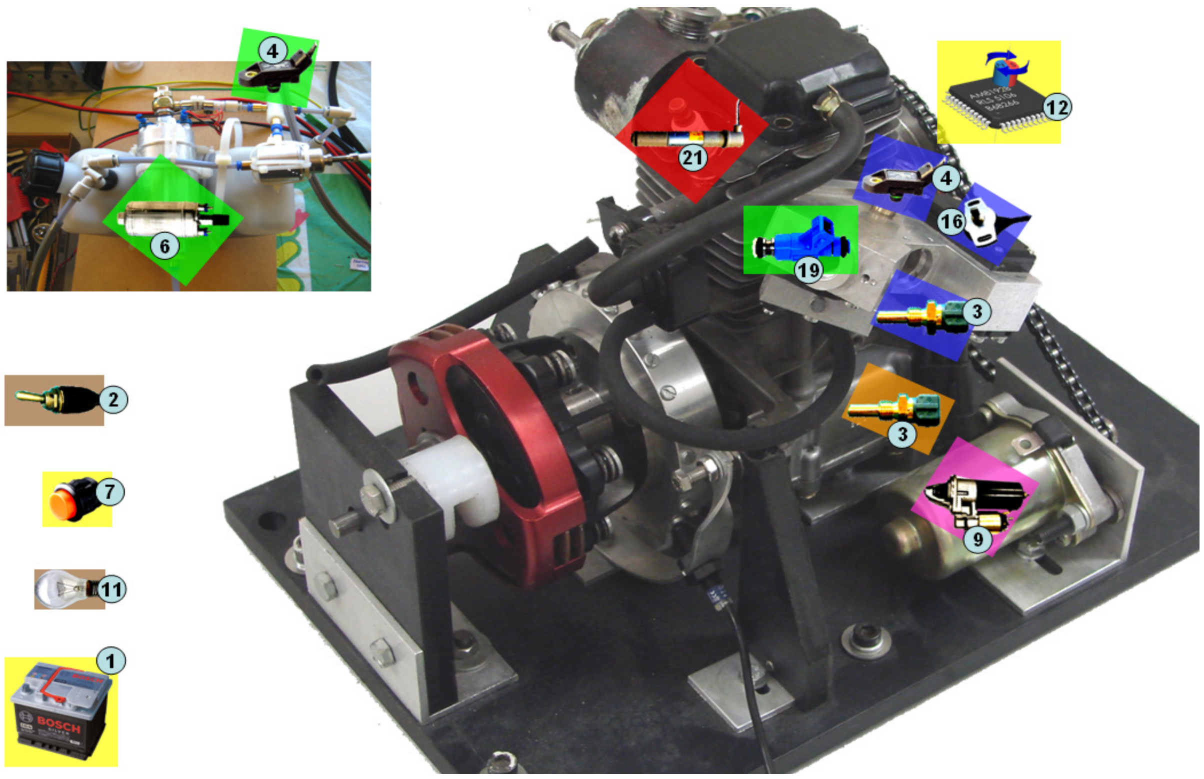


Fig. 6.2:13 – Overall system overview with all the peripherals and their specific types (colours)

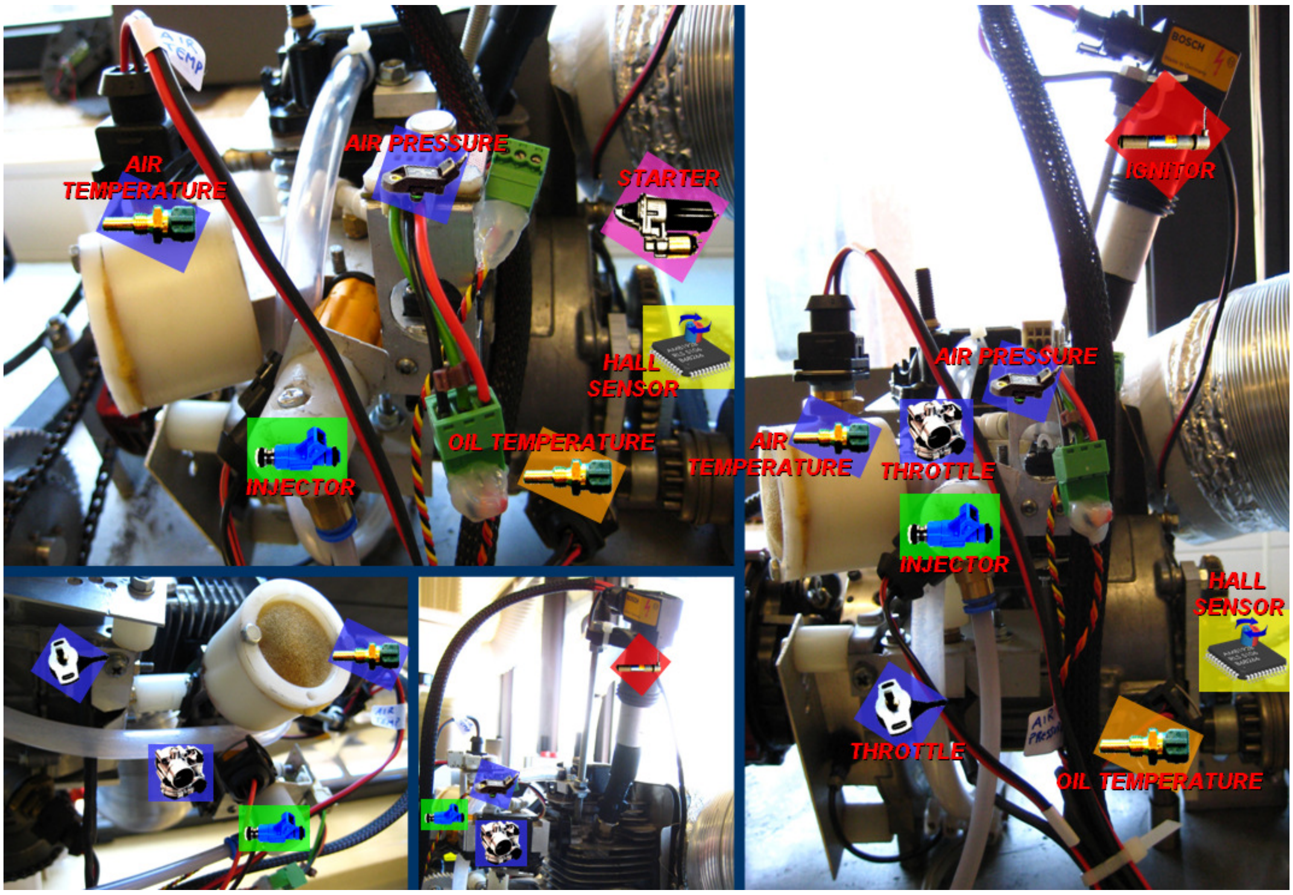


Fig. 6.2:14 – Various views of most smart peripherals attached to the engine

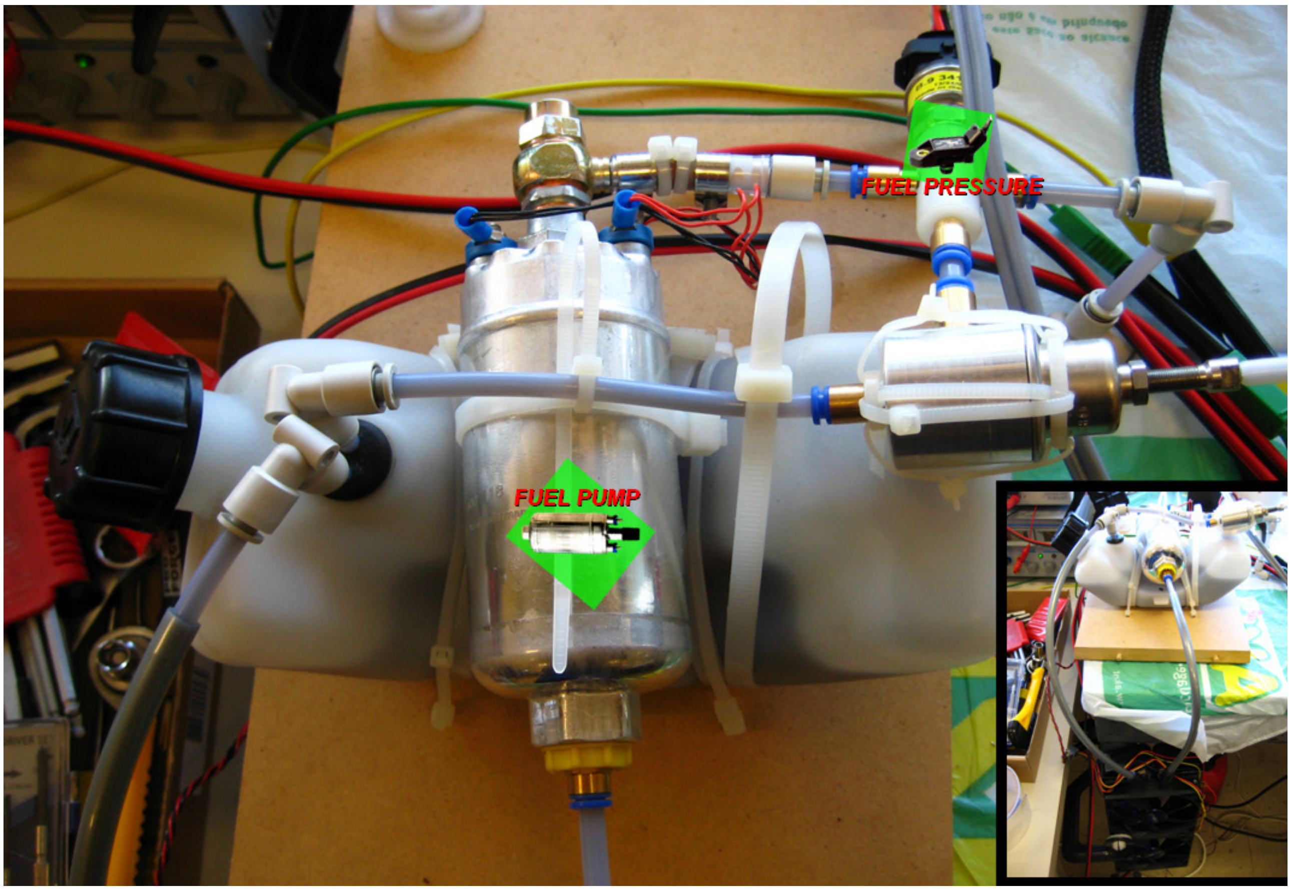


Fig. 6.2:15 – View of the fuel-pump and fuel inter-cooler peripherals and their mechanical components

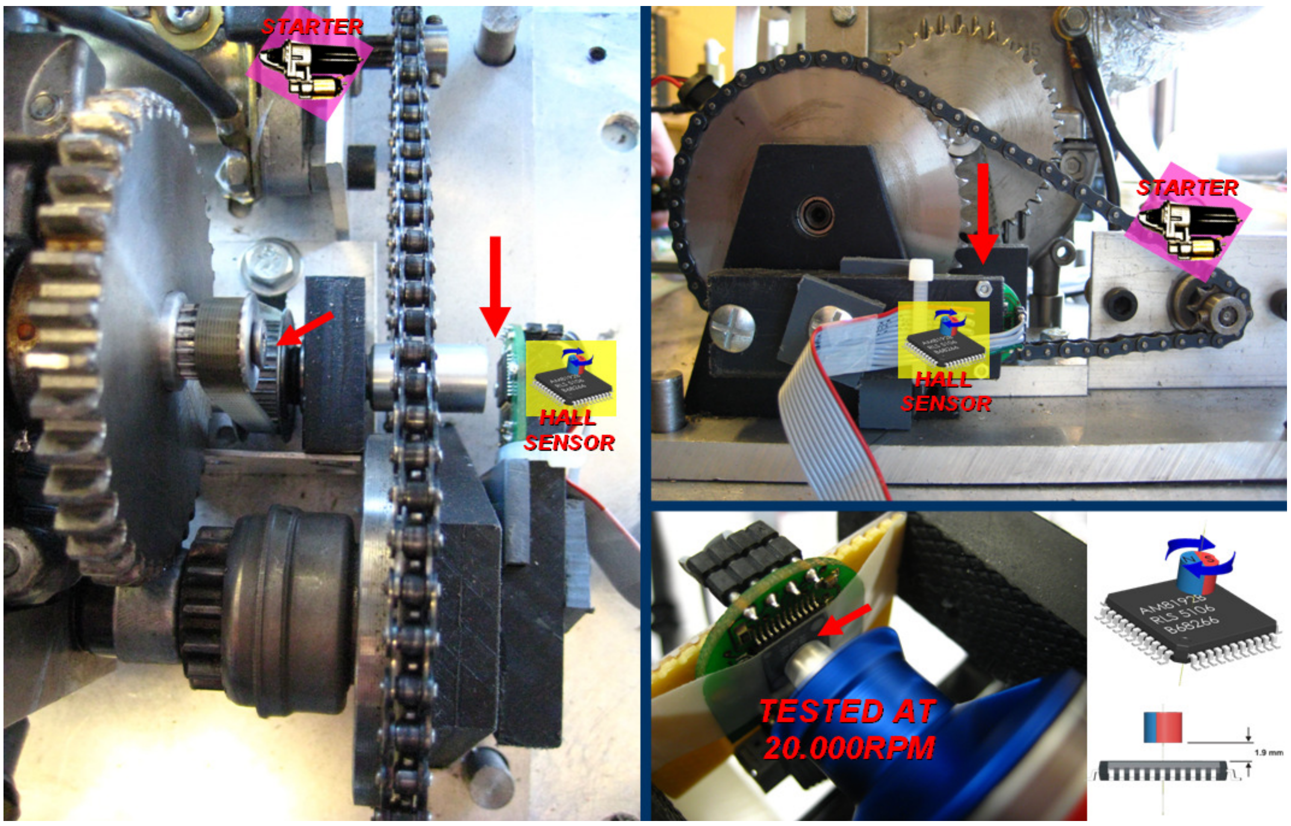
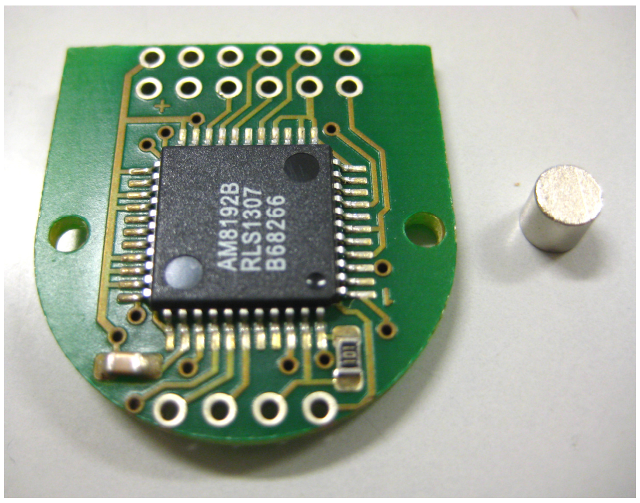


Fig. 6.2:16 – View of the Hall-effect absolute engine crankshaft angle measurement sensor



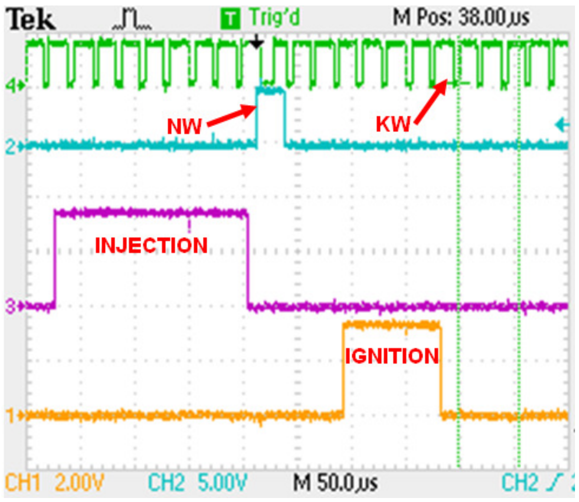


Fig. 6.2:17 – Left: absolute angle sensor chip; Left: scope output (top) and corresponding actuator signals (bottom)

Fig. 6.2:19 shows the complete *"ECU2010"* electronics with the central ECU (3-stacked *"Cellular-ECU"* modules with 6 antennae, 2 on each module) and the "Smart Peripherals" boards stack (composed by the total 12 *"Cellular-ECUs"*). The wireless datalogger was placed some meters away on another table and thus does not appear on this photo (the Datalogger looks similar to this main-ECU, since it also has 2 *"Cellular-ECU"* modules with 4 antennas). The complete list of smart peripherals used on this engine is as follows in Tab. 6.2:18 below. Fig. 6.2:20 shows a topological view of the entire system (again, except for the Datalogger) in that it is readily understood where all the peripherals are connected, regarding the digital-bus and the corresponding 3 main *"Cellular-ECU"* modules. Fig. 6.2:21 additionally shows the corresponding *"Live-Vehicle"* views.

SENSORS (9)	ACTUATORS (5)
Ignition switch*	Fuel pump
Start switch	Fuel injection valve
Engine oil temperature	Ignition coil
Intake air temperature	Starter*
Fuel pressure	Diagnosis light*
Intake air pressure	
Crankshaft angle	
Throttle angle	
Battery voltage	

Tab. 6.2:18 – Sensors and actuators attached to the engine

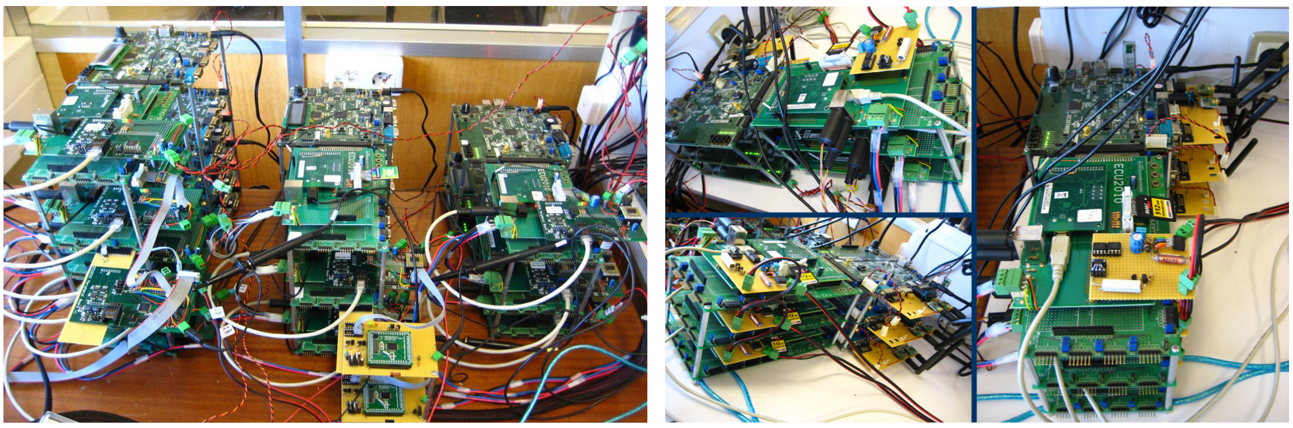


Fig. 6.2:19 – Final complete electronics of the "ECU2010" DEMO's final prototype (except Wireless Datalogger)

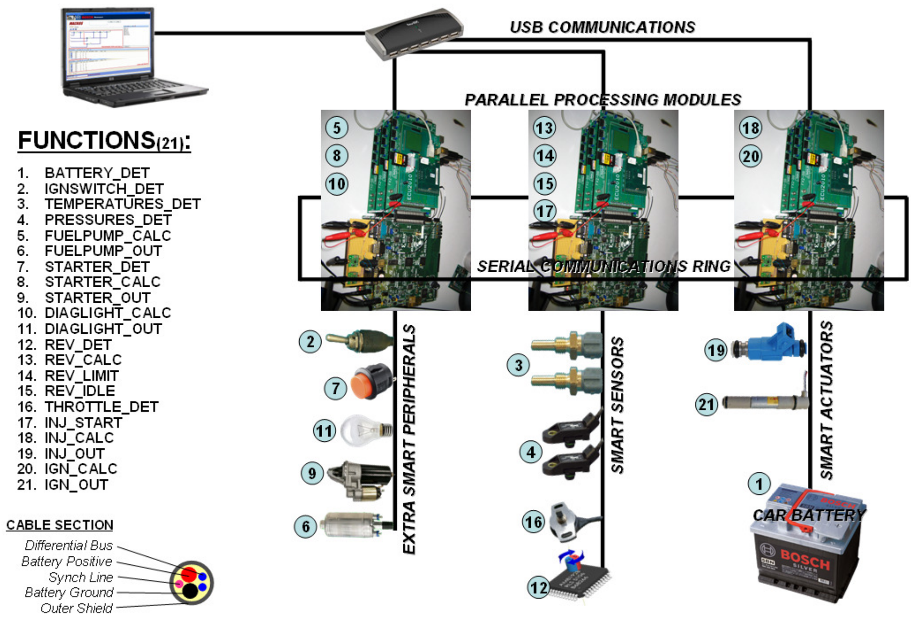


Fig. 6.2:20 – Topological view of the "ECU2010" DEMO's final prototype (except Wireless Datalogger)

* These three peripherals were incorporated into the same smart peripheral electronics, because of lack of sufficient material. So, although there is a total of 14 real peripherals, only 12 "Cellular ECU" boards were really used.

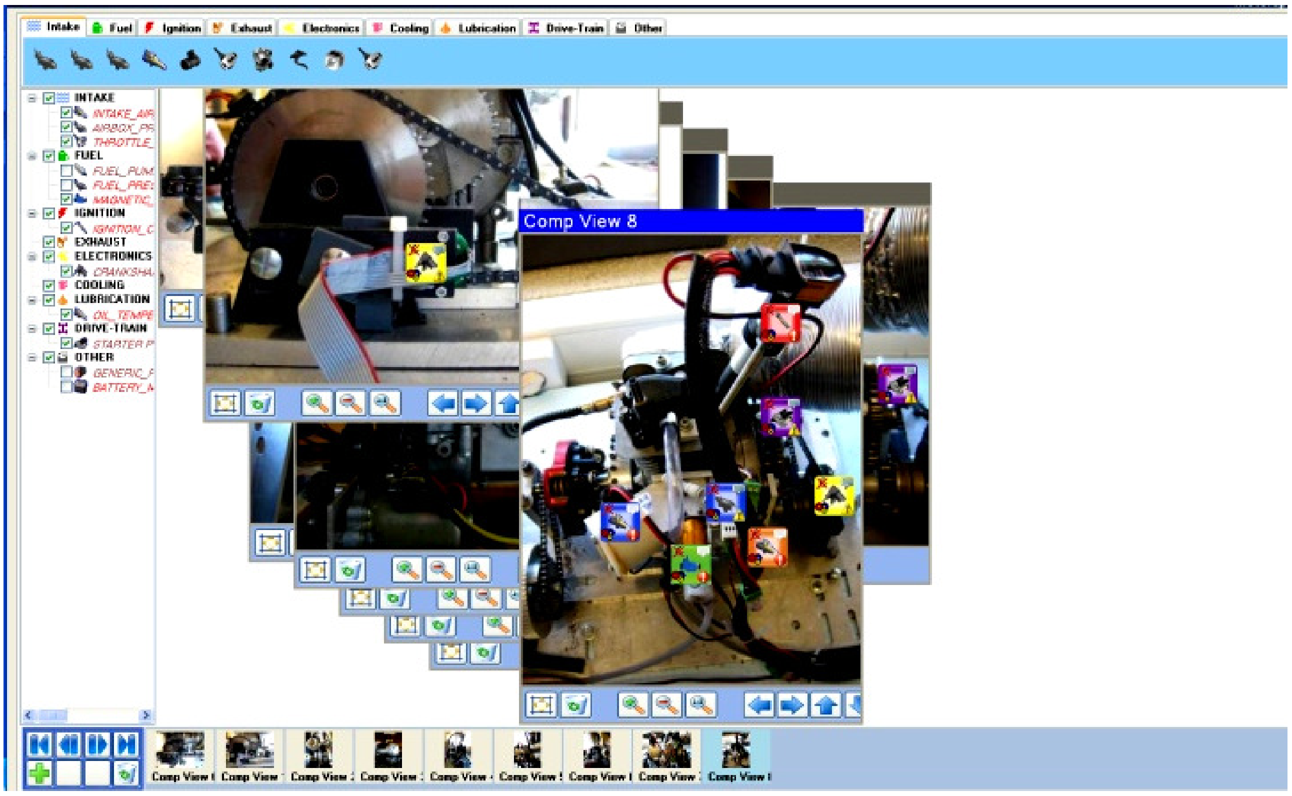


Fig. 6.2:21 – "Live-Vehicle" view inside the "iEditor", corresponding to the above detailed system (peripherals only)

The "digital-bus" for the peripherals used a herein developed protocol, which derived from various features of available automotive protocols (listed below). Simplicity was preferred over security, for development efforts not to take too much time. It was not part of the main features' demonstration and proof-of-concept. Thus, some of the typically automotive and other protocols considered here were:

- ***CAN*** → the most used protocol in the near past, now being surpassed by faster and more complex ones such as FlexRay, Ethernet, among others; its main characteristic is reliability through a well-behaved double-terminated bus and CRC15; essentially used in medium- to highly-critical peripherals such as ESP, ABS, automated gear-boxes, dashboards and any other peripheral requiring minimum timing- and error-avoidance constraints; furthermore, the physical layer is of differential nature, therefore being resistant to interferences; bit-rates of 500Kbps and 1Mbps are the most common
- ***LIN*** → still used for low-speed and non-critical peripherals such as electric window elevators, roof openers, key locks, lighting among other essentially accessory-type peripherals in a vehicle
- ***PSI5*** → uses only 2 wires and is a current modulation protocol, in that power is also delivered through the same wires; bit-rates of 125Kbps and parity of CRC3 are commonly used; it was never extensively used in the automotive scene due to other better alternatives
- ***SENT*** → uses 3 wires and is a point-to-point one-way protocol with CRC4; it was never extensively used in the automotive scene due to other better alternatives
- ***RS484*** → considered only for its differential electrical characteristics (interference resistant); it was never extensively used in the automotive scene due to other better alternatives
- ***1-Wire*** → protocol from Texas Instruments (created by former Dallas technology) used for low-speed and low-power peripherals, considered only for its wiring savings (only 1 signal wire and GND needed for both supply and communications). ***I^2C*** is a similar "1-wire" protocol.

Having all the previous protocols in mind and the very narrow time-frame in that everything had to be implemented, a simple sub-set of features was then derived for the system's "digital-bus": a CRC5 error-detection mechanism only, a maximum bit speed of 2Mbps and a differential pair of wires to avoid most interferences. Furthermore, to completely eliminate the need for a common clock or, at least, clock negotiation efforts, besides even allowing theoretical crystal-free operation for peripherals critically affected by extreme physical vibration stress, this protocol signalizes zeroes as short pulses and ones as long pulses. To further simplify design, long pulses were chosen to be double the duration of the short ones. Each message contains 2 "sync bits" (one "zero" and one "one") to allow for the receiver in the system to measure those lengths prior to correctly decode the message bits thereafter. This method wastes some bandwidth, of course, but generally allows for a totally flexible variable-bitrate transmission. Fig. 6.2:22 shows an example of the physical signal and the flexible message formats for different usages.

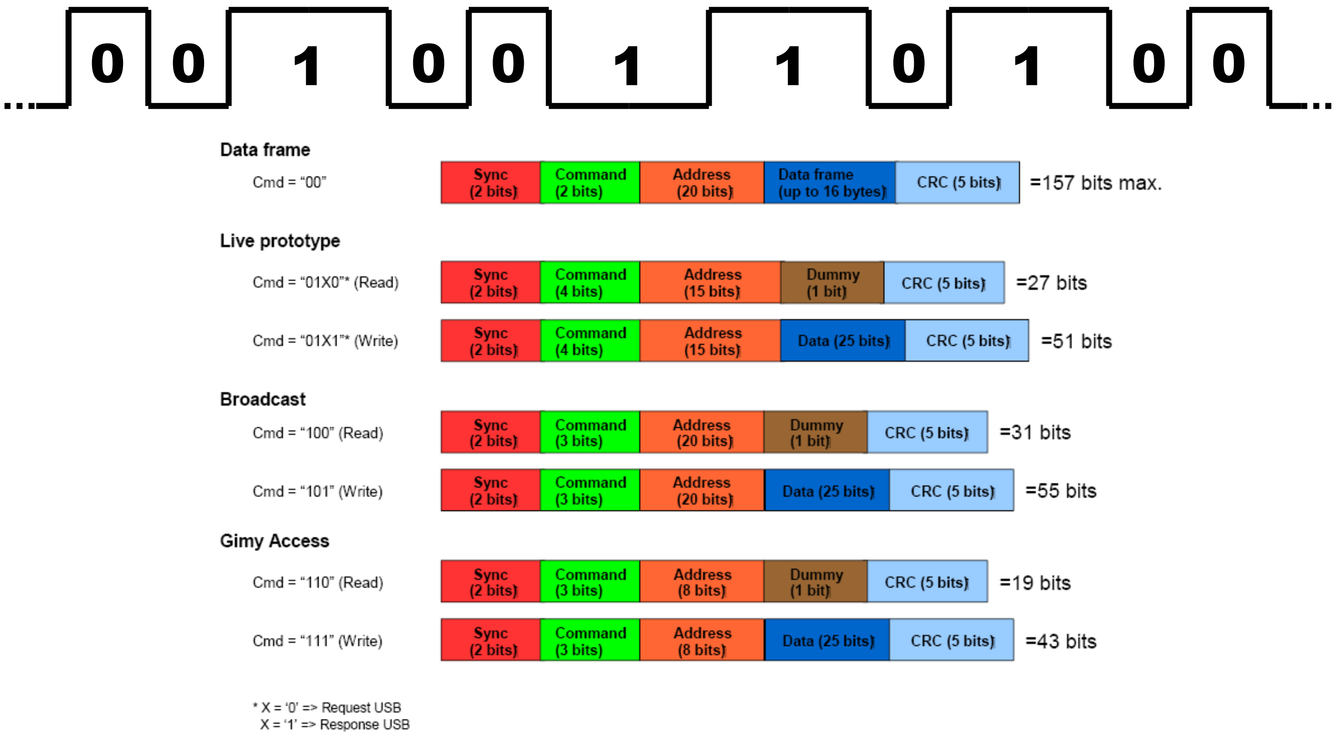


Fig. 6.2:22 – Physical signal morphology inside the "digital-bus" and message formats

A particular case is that of those peripherals dedicated to directly actuating the injector and the ignition coil, which naturally need an absolutely timely accurate engine-position signal. Since sending and interpreting engine position messages/*"Nodes"* would be totally unsuited because of prohibitive frequency/bandwidth and fatal latency/jitter issues, a very simple and direct digital signal was hence used: a simple wire carrying a square wave with one pulse per degree of engine rotation. A gap (missing pulse) would then signal the absolute zero position within each complete 720° 4-stroke work cycle. This translates to 120KHz @20.000RPM as a very comfortable signal, present for all peripherals and used only by those really needing that kind of synchronization. The bottom-left of Fig. 6.2:20 illustrates a cross-section of the herein finally used "digital-bus" for all of the Smart-Peripherals attached to the small engine. The contained wires and signals were:

- ***Battery Power*** → positive and negative of normal battery power to supply all peripherals, both for their processing and also for their actuation needs if any present
- ***Differential Bus*** → this is the communications "digital-bus" referred to in this work, where messages and *"nodes"* from and to the respective main ECU module travel at high serial speed, also unavoidably used for all peripherals

- ***Synchronisation Line*** → this extra line is a special wire where raw "crankshaft teeth" are sent out from the crankshaft absolute-angle sensor, with a big difference to the normal *"Nodes"* this sensor transmits to the main ECU modules: these teeth are a square signal proportional to the crankshaft speed and delayless; this "raw/immediate" signal is used in addition to the normal slow angle value *"Nodes"* to allow for high-speed/critically timed peripherals such as ignition and injection ones, to be served with the most actual crankshaft position at all time (modern direct-injection series-car engines need a temporal accuracy of at least 1° to be effective/efficient)
- ***Outer Shield*** → just a classic protection against unwanted interferences from outside

Last but not least, each "digital-bus" emanating from each main *"Cellular-ECU"* module, was capable of harbouring up to 16 peripherals, through 16 plugs mounted on the cabling. These plugs contained a different resistor each, so that a very simple, cheap but effective peripheral identification mechanism was implemented: each Smart Peripheral read out its own plug resistor magnitude (simple voltage divider) upon start-up, being that the seed for the ID/address of that peripheral. A "plug & play" mechanism was implemented on the main-ECU side, so that the *"iEditor"* became automatically aware of all connected peripherals in the system, allowing the user/developer to then use whatever peripheral *"Nodes"* he needed for his FDEFs to work. This "plug & play" mechanism also allowed the peripherals to be disconnected and connected/reconnected at any time, with immediate re-scan and recognition by the *"iEditor"*, operating with those peripheral IDs on the "peripherals' routing tables". This simple plug-based ID mechanism eliminated any need for each peripheral to have such IDs programmed-in in any way. It also allowed for direct broadcasting messages as well. A master-slave (main-ECU initiated communication exchanges) topology adds collision-free communications to further simplify and enhance the highly deterministic nature of this developed protocol. This multi-drop "digital-bus" is depicted in Fig. 6.2:23.

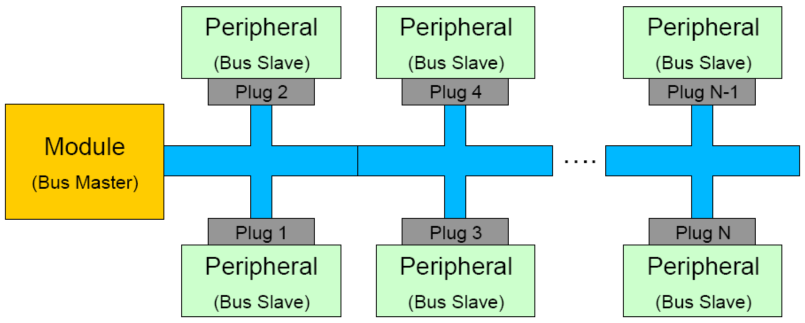


Fig. 6.2:23 – Physical bus topology with several peripherals connected to their plugs

6.2.4 Engine Management Functionalities

[Emphasis: minimalist engine management effort]

The engine functionality was developed to keep all algorithms at a minimum level of complexity possible, but yet allowing a smooth and good operation of the DEMO mono-cylinder engine. Fig. 6.2:24 shows an overview of all implemented FDEFs that comprise this complete engine control-structure. Basically, the sensors **DET**ect the engine’s current status, feed the corresponding values into the **CALC**ulating engine-controlling FDEFs and these in turn **OUT**put the control-values for the actuators on the engine. These actuations will in turn have reactions from the engine. These reactions will again be sampled as the current status of the engine, and so we have a closed-loop control system implemented here. For a complete and more detailed view of these FDEFs, please look into [Att. 5]. Note that the colours adopted for each of the blocks (1 block = 1 FDEFs) correspond to the colours adopted for each type of peripheral class, such as already illustrated in Fig. 5.1:21.

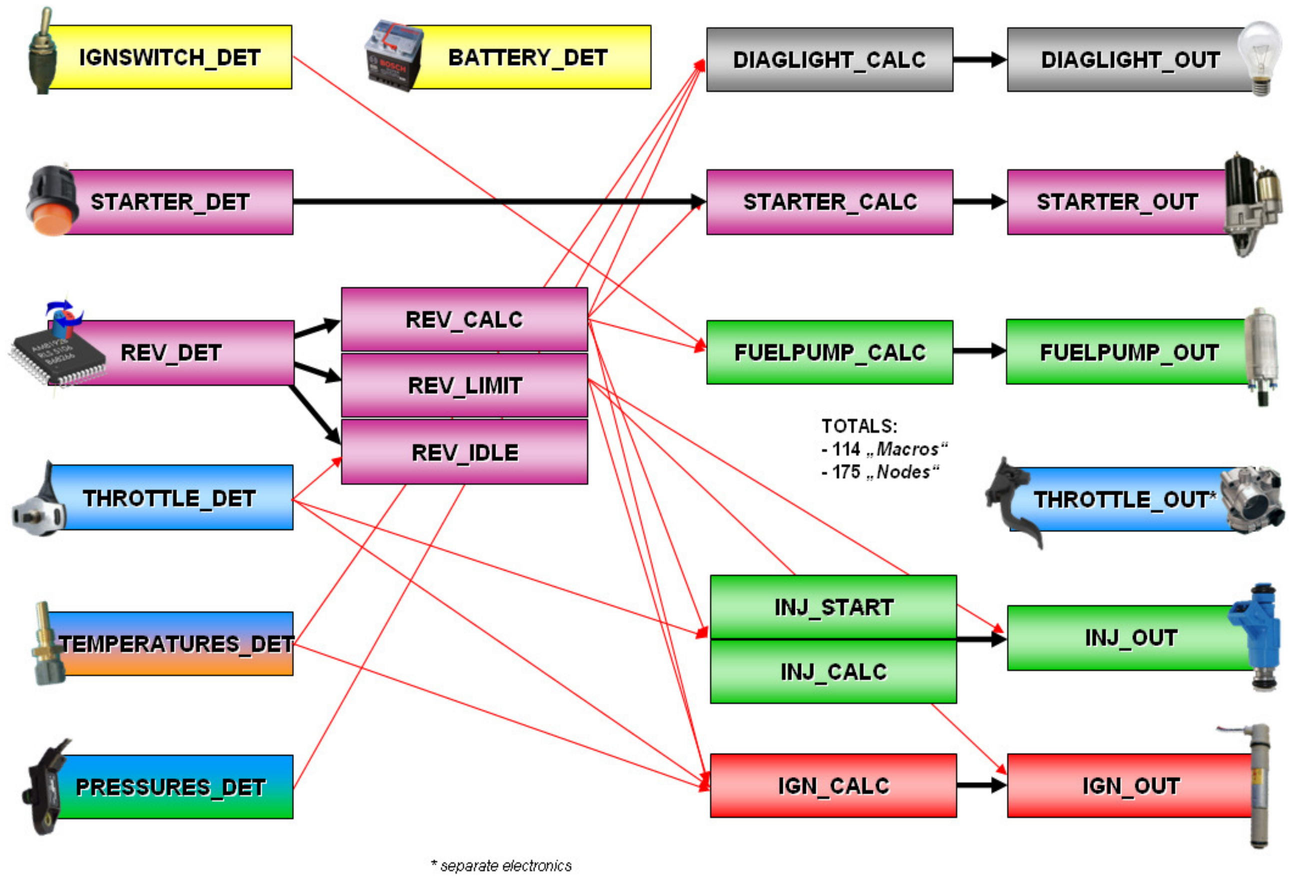


Fig. 6.2:24 – Block overview of the engine's basic control-FDEFs

Note that, as explained earlier during the *"Macros"* description, no "IF-THEN-ELSE" construct was ever used in these engine functionalities [Att. 5] whatsoever. Everything has been simply programmed following a strict data-flow nature of all signals. All these functionalities aim to produce the following internal basic engine behaviours on the engine demonstrated herein:

- "Alpha-N[*]" injection modelling through main injection map
- Injection quantity corrections through temperatures and pressures
- Battery corrections for injection and ignition outputs
- Injection quantity compensation upon start
- Automatic starter activation release
- Idle-speed control with low throttle
- Engine-speed upper limiting
- Fuel-pump control
- Diagnosis light activation upon any failure

[*] *This modelling method, used in virtually all automotive vehicles with gasoline engines in general, boils down to having the injection quantity calculated according to the two fundamental engine values: throttle-valve opening angle (Alpha) and engine-speed (N). The most simple way of calibrating such an engine is to have a 2D map with those two values as inputs and the injection quantity as output.*

6.2.5 Final DEMO at Bosch Motorsport

[Emphasis: most surprisingly, everything shown worked exactly as expected]

As already mentioned, on November 28th, 2008, this DEMO was virtually taken to the Bosch Motorsport headquarters in Markgröningen, Germany, by means of the *"Tele-Operation"* remote view feature of this platform. There, it was fully demonstrated by remote-control up to every single handling detail, except for online datalogging due to mere bandwidth reasons. Most surprisingly and taking the "spider-web/brittle prototype" into full account, where bad contacts had already been generating trouble back in the laboratory where the engine was, during the DEMO everything worked exactly as it should and all the shown features operated correctly.

Every detail was demonstrated with success, despite the distance and poor visual contact through simple web-cams on a big-screen with a 3-seconds delay, two-way audio with poor quality and textual chat-box. The only equipment transported to Germany were the laptops of each of the team's member that travelled there. Other than that, the only contact was through internet and web-cams. There was a web-cam pointing at the complete engine and another pointing at the engine's crankshaft to allow to observe the movement of the engine. Images being transmitted onto the big-screen had a constant delay of about 2-3 seconds, which further posed difficulties in getting the audience to correctly grasp what was going on exactly. Fig. 6.2:25 shows rare snapshots of the partial *"ECU2010"* team that flew to Germany, including a close-up snapshot from the webcam being projected onto the Bosch Motorsport's main meeting room. The audience was composed by Bosch Motorsport employees and directors, as well as 2 department directors from Robert Bosch's series-car scene. After this successful DEMO, 2 meetings were immediately scheduled at those same Robert Bosch departments, for further explanations and questions. More photographs of the team and other details, can be seen in [Att. 16].

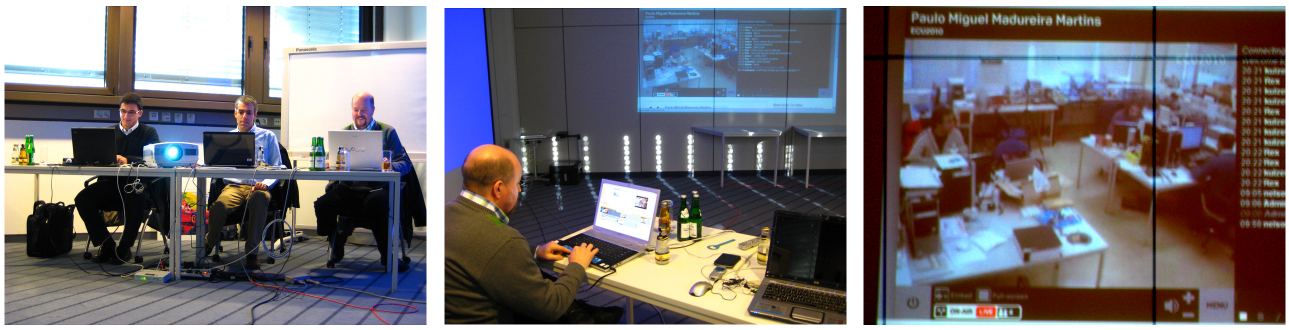


Fig. 6.2:25 – Final DEMO at Bosch Motorsport (from left to right: Eng. Bernardino, Eng. Kulzer, Prof. Bernardo)

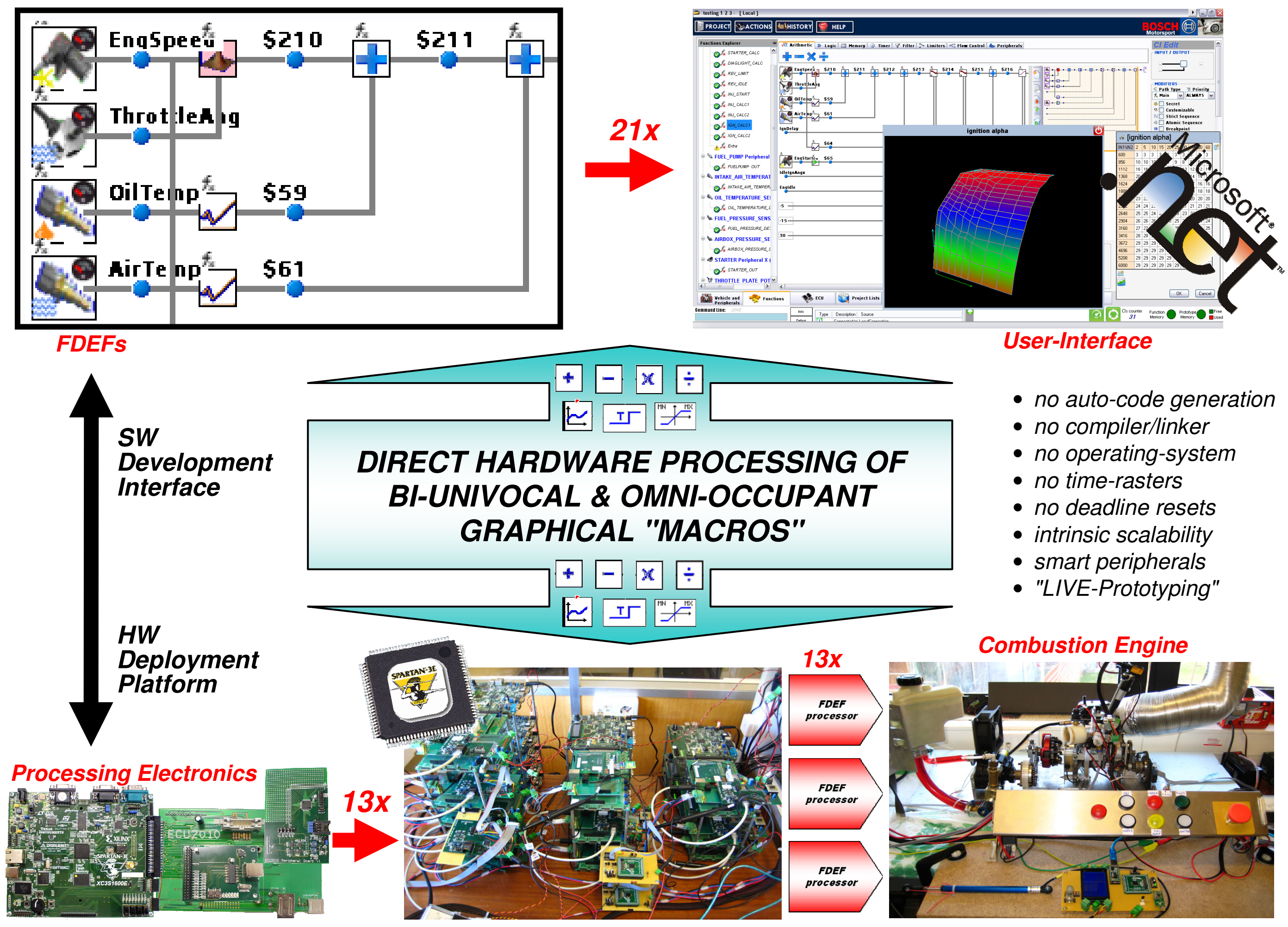
Some processing details of this final demonstrated implementation without any optimizations such as pre-fetching or VHDL code rewriting done yet, can be seen herein below. As successfully demonstrated, this global processing and communications capacity was enough for the gasoline engine to rev up to 6.000RPM, primarily limited by the need to keep this engine from sustaining any damage due to some wiring/contacting failure, or other reasons. These values are "per *Cellular-ECU*" module and per respective digital-bus. In fact, this makes any comparisons with centralized systems difficult. Nevertheless, some minimally required Motorsport-typical values are included in parenthesis for the sake of comparison and reality/feasibility check (the system used 3 *"Cellular-ECU"* modules, so that for example 60 *"Nodes"*/ms already results in a potential total of 180 *"Nodes"*/ms):

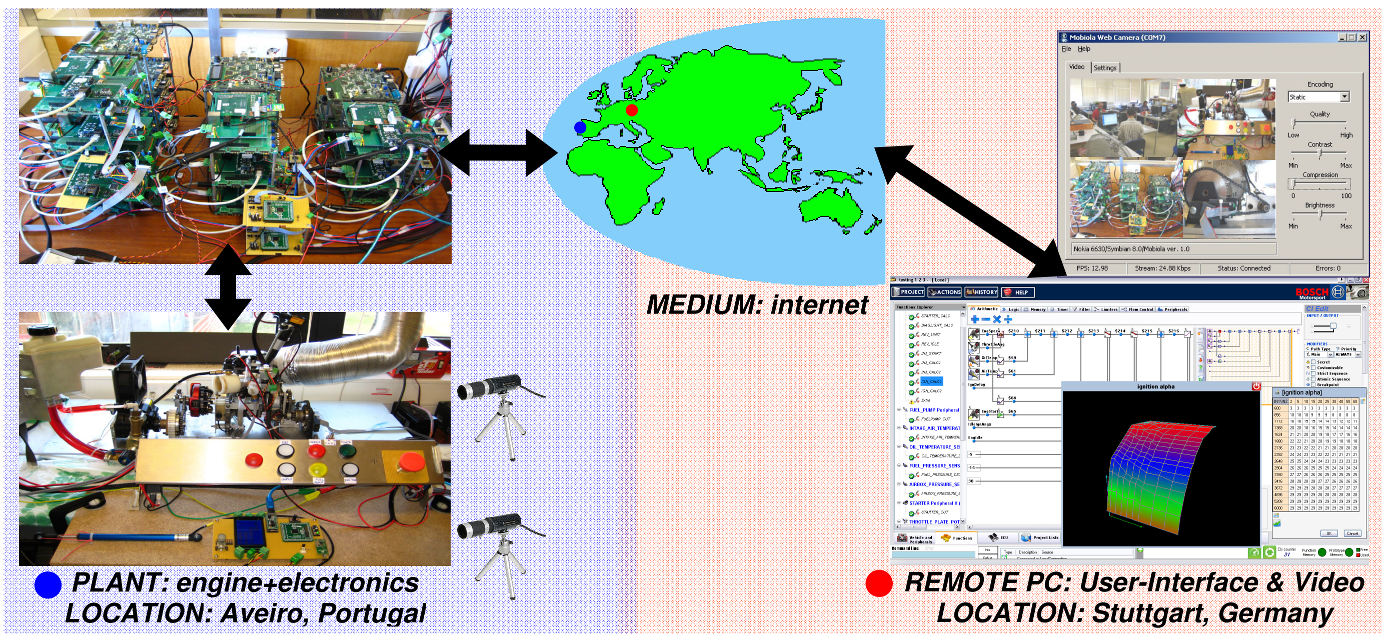
- FDEF-Processor averaging about 600 *"Macros"*/ms or an equivalent of about 1.7µs/*"Macro" (total minimum ≈ 1333 FDEF-operations/ms or 750ns/FDEF-op; values for the first prototype with micro-controllers as well as the first FPGA-based BCDP calculator unit can be checked and compared within* [Att. 54]*, more specifically in Tab. [Att. 54]:187 and Fig. [Att. 54]:201)*
- Inter-module communications averaging 80 *"Nodes"*/ms between each pair of *"Cellular-ECUs"*
- Internal data-logging averaging about 60 *"Nodes"*/ms *(total minimum ≈ 30-40 vars/ms)*
- External data-logging averaging about 20 *"Nodes"*/ms *(total minimum ≈ 30-40 vars/ms)*
- 2h-Race log-data read-out averaging about 3 minutes *(total minimum ≈ 2-3min)*
- Digital peripherals-bus averaging about 20 *"Nodes"*/ms *(total minimum ≈ 20-40 vars/ms)*
- Monitor/Debug values read-out averaging about 100 *"Nodes"*/ms *(total minimum ≈5-20 vars/ms)*

The very fine team that made all this possible from the beginning, by engaging into this "ECU2010" projects with all their wits and energy, at a visit to Bosch Motorsport's facilities in Markgröningen, Germany, from left to right: Paulo Martins, Nelson Bernardino, Ricardo Marques, Cátia Ferreira, Filipe Teixeira and Rui Gomes

THANKS TO ALL!!!

(This illustration wraps up the main ideas implemented in this thesis: a visual language directly processed by parallel hardware, as well as the final "Tele-Operated" engine in Portugal, with the DEMO done in Germany)




MEDIUM: internet
Mobiola Web Camera (COM7)
● PLANT: engine+electronics
LOCATION: Aveiro, Portugal
● REMOTE PC: User-Interface & Video
LOCATION: Stuttgart, Germany

6.3 The Final “MS5/ECU2010 Connection” Experiment

[Emphasis: the first offspring featuring "ECU2010" technology]

After the success of the final DEMO, it was time to make an intermediate step in using this technology, by merging parts of it into existing and proven ECU systems at Bosch Motorsport. The candidate for this action was an "MS5" ECU [10]. The aim was then to connect this ECU to a simplified *"ECU2010"* system comprising the same 3 main ECU *"Cellular-ECU"* modules but no peripherals. This connection had to be made through the serial ring-bus, thus connecting those 3 main modules to the "MS5" in a global ring, where the "MS5" was the 4th module inside the ring. Furthermore, the FPGA contained inside the "MS5" was the obvious candidate to interface with this ring-bus, by mirroring the exact same "Inter-Communications Manager" also used for the main *"Cellular-ECU"* modules. This was done by "simply" copy-pasting the VHDL code used inside the *"Cellular-ECUs"* for that manager component. This way, the "MS5" could exchange *"Nodes"* with the remaining *"ECU2010"* system without any need for any other additional mechanism. This had the crucial advantage of equally not disrupting any of the two self-contained systems.

The main goals for this technology merge/mixture was to have something that could be directly programmed and handled by the customer himself, since the MS5 ECU is only manageable by expert programmers who deeply understand Matlab Simulink [5]. At this point, Bosch Motorsport showed great interest in testing this *"Macros"* and easy-handling *"iEditor"* interface as a merge into their MS5 flagship ECU series. One possible reason could be attributed to Australian ECU MOTEC competitor having announced its easy and customer-oriented script-like programming features for its future ECU line (see Fig. 6.3:26). This script-like feature comes somewhat close to the "Embedded MATLAB Function" blocks inside Matlab Simulink [5] and to “MathScripts” [386] inside LabVIEW [6], but without any graphical counterpart (like Simulink itself). With the *"iEditor"*, a graphical instead of just textual (MOTEC scripts) interface, Bosch Motorsport would instantly leap ahead of its competitors. The MS5 would bring its proven power, reliability and Matlab Simulink power-programming, while the *"Cellular-ECU"* programmable modules would bring their flexibility, expansibility, graphical *"iEditor"* and general easy-handling features into this merge, thus getting the best of these two worlds. An excerpt of the advantages of such a merge, is shown here in short:

- ***Hardware package*** → MS5's proven standard HW together with ECU2010's parallel processing HW allows to quickly adapt to virtually any needs just by picking the necessary MS5 ECU and ECU2010 modules from the shelves and by exchanging the needed variables/*"Nodes"*
- ***Software package*** → MS5's proven MatLab Simulink programming power together with ECU2010's direct/compilerless graphical FDEF description/processing allows to rapidly develop the needed functionality and replicate ASCET functions
- ***Testing & Validation*** → ECU2010's advanced *"Live-Debugging"* and *"Live-Prototyping"* features allow to rapidly implement variant functionality and correct problems as they appear, with handling ease and speeds never seen before
- ***Extra features*** → MS5 and ECU2010's newest features may easily be used on old-timers right away, for whatever reason they're needed, such as data-logging, reverse-engineering, tele-operation/tele-assistance, PnP-peripherals, etc. More than just the MS5 may attach to the ECU2010 inter-communications SPI ring

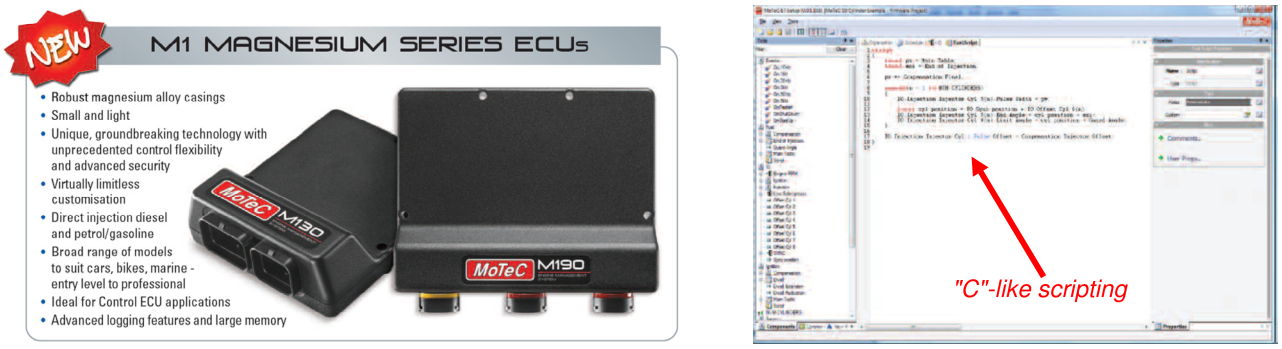

NEW
M1 MAGNESIUM SERIES ECUs
Robust magnesium alloy casings
Small and light
Unique, groundbreaking technology with unprecedented control flexibility and advanced security
Virtually limitless customisation
Direct injection diesel and petrol/gasoline
Broad range of models to suit cars, bikes, marine - entry level to professional
Ideal for Control ECU applications
Advanced logging features and large memory
MoTeC M130
MoTeC M190
"C"-like scripting

Fig. 6.3:26 – MOTEC's high-end ECU series with textual "C"-like scripting capabilities (source: [419])

From Fig. 6.3:27 on, pretty much self-explanatory illustrations of this experiment are shown. The final setup contained an FDEF inside the *"ECU2010"* system being processed in parallel to the "MS5", with values being exchanged and used mutually in both systems. In detail, the "MS5" sent some values to the *"ECU2010"*, which processed them and, in turn, returned them to the "MS5" for further processing and output. Basically, the thereby implemented featured strongly resemble those of a classical "bypass" such as those possible with *"ECU Interface Manager"* [417], *"No-Hooks OnTarget"* [300] and *"EHOOKS"* [223]. Special input and output peripherals named "MS5" took the place of the bypassed variables, such as in Fig. 6.3:29 below. Fig. 6.3:30 then finally shows part of the really demonstrated functionality at Bosch Motorsport, in conjunction with INCA, the there used value inspection software.

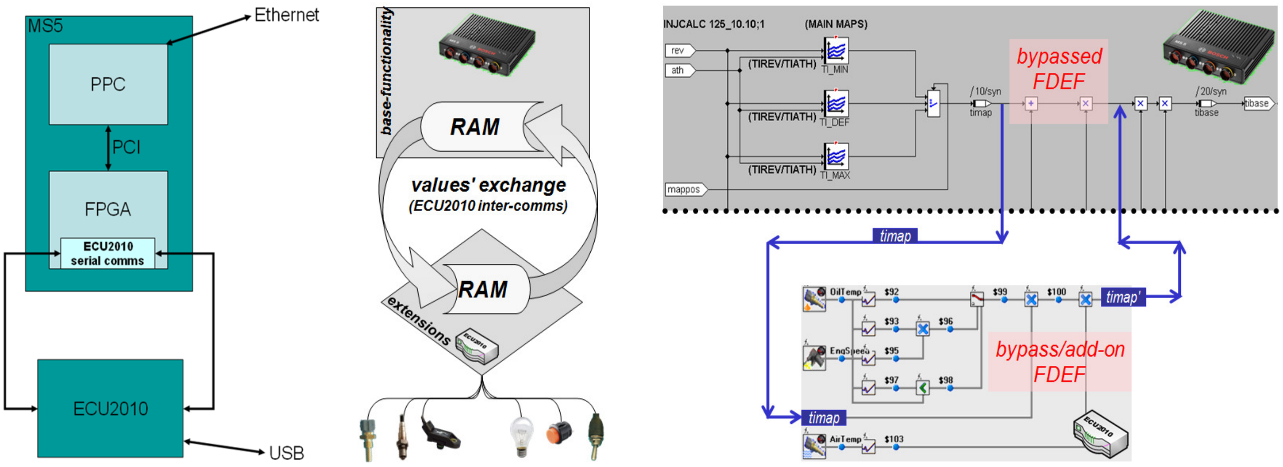


Fig. 6.3:27 – Left/Middle: general merge block-diagram; Right: more detailed resulting merged bypass capability

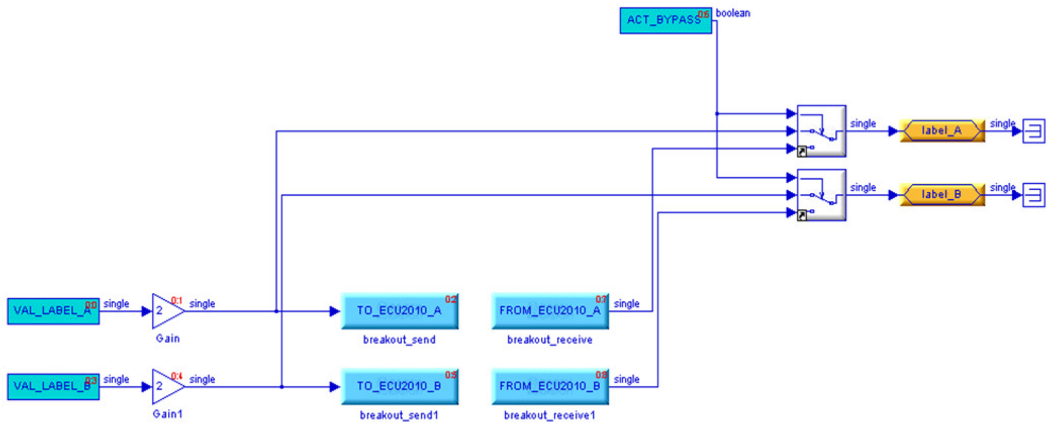


Fig. 6.3:28 – Really implemented Matlab Simulink "blue blocks" for the bypass into "ECU2010"

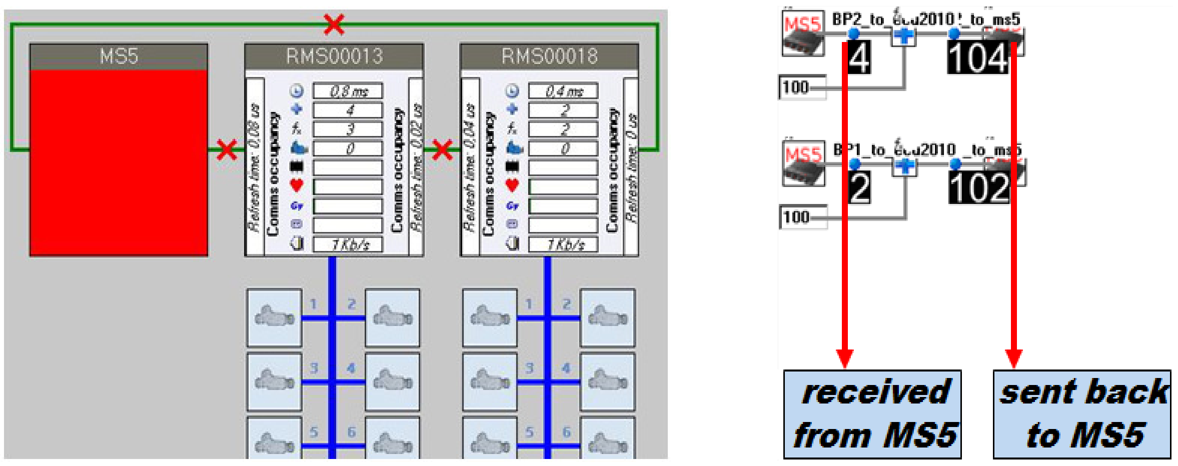


Fig. 6.3:29 – "MS5" ↔ "ECU2010" connection experiment inside the "iEditor" ECU and Function views, respectively

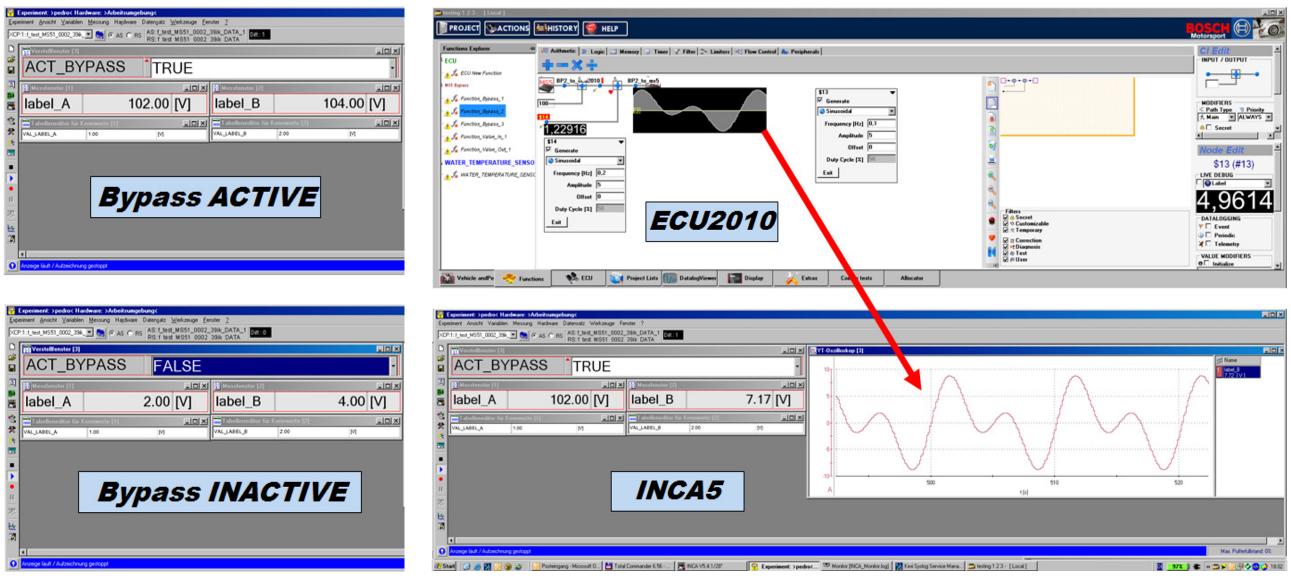


Fig. 6.3:30 – "MS5" ↔ "ECU2010" experiment (Left: bypass activation; Right: bypass with "Live-Generation")

This experiment was again successfully demonstrated at Bosch Motorsport on July 22nd 2009, where a combination of a real MS5 and real two *"Cellular-ECU"* modules (as seen in Fig. 6.3:29) worked perfectly in front of an audience. This happened shortly before the project was unfortunately stopped, until further notice, due to the "2008 financial crisis" consequences upon Bosch Motorsport's business path. Up until 2015 this project has not been re-activated, while contractual secrecy obligations extinguished 5 years later, on July 22nd of 2014. This thesis represents a report of all that happened during those 3 years.

Page intentionally left blank

Page intentionally left blank

"It seems that perfection is reached not when there is nothing left to add, but when there is nothing left to take away"
– Antoine de Saint Exupéry

CHAPTER 7

Conclusions and Future Possibilities

7.1 Conclusions

In classical systems, the underlying software engineering occupies much of the available time keeping everything working, despite of more than 80 sources of pitfalls. This work introduced the concept of a "Direct Graphical Processing" mechanism based on the *"Macros"* and corresponding *"FDED-Processor"*, eliminating or minimizing potential pitfalls. *"Live-Prototyping"* is the foremost feature of this work, delivering the possibility of changing any system programming detail on-the-fly without having to ever stop it from operating.

In this work it was possible to build a complete automotive system from scratch, in a relatively short time-frame and with only a few people involved. Contrary to classical attempts where large efforts are needed for everything, this work also included advanced design, prototyping, simulation and even tele-operation features. Again, it must be noted that this work had only one main goal in mind as for results: the "proof-of-concept" centred around the initial idea of the *"Macros"* with their *"Nodes"*, extending through concepts such as the *"Macros-Sequence"*, *"FDEF-Processor"*, *"Cellular-ECUs"*, "Smart Peripherals" and finally the *"iEditor"*. Thus, no optimization work has been conducted in terms of speed, size and cost. Those points, from the engineering point of view, are due in future work. Fig. 7.1:1 shows a schematic overview of the most relevant internal components and their corresponding relation in this system, in a progressive way and starting at the *"Macro"*.

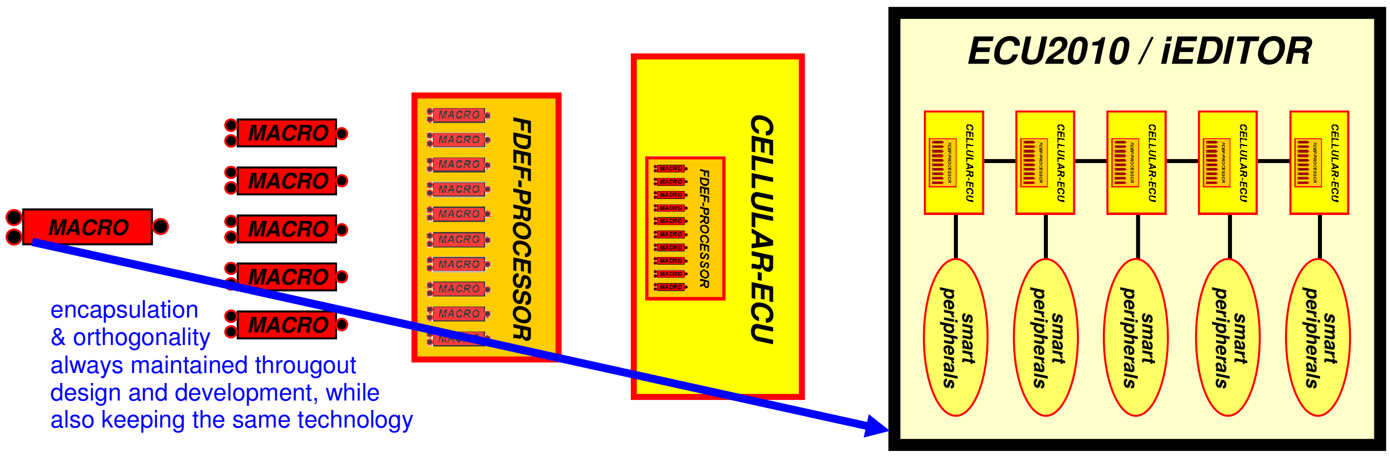


Fig. 7.1:1 – Development path from the "Macro" to the final "iEditor" controlling/encompassing the entire ECU2010

The “Multi-System” approach (see Fig. 4.5:77) implemented in this work ultimately allowed for homogeneous and individually complete/encapsulated processing entities, which in turn allowed easier inter-connection and handling among them. Instead of “Multi-System” one could also call this as truly “Multi-FDEF”, since FDEFs are really scattered among the system. It should be noted that this model does not only benefit from software partitioning, but also from hardware partitioning as well. This approaches the basic idea of having many smaller CPUs processing many smaller (and therefore less bug-prone) software chunks [379], instead of a big software chunk being processed by a big CPU.

Compared to the classical parallel processing difficulties and even virtual impossibilities of separating highly inter-woven code, the obtained results are only possible because of the already detailed orthogonal, independent and asynchronous nature of most of the components of this system developed herein. While classical parallel processing is poised by highly inter-connected, inter-locked and mandatorily synchronised solutions, in the proposed model all that disappears. Only feed-forward mechanisms are mainly used, in that no explicit synchronisation is use anywhere, basically and simply letting all sub-components such as *"Macro"* processing and bus communications happen asynchronously relative to each other. Instead of absolute synchronisation, only overall checking mechanisms are used to know if the resulting global processing meets certain minimum requirements. And even if those requirements are not met, the system still continues working without hard-limiting resets or problems, because of the also already explained "graceful degradation" concept.

Fig. 7.1:2 provides an illustration of this comparison, where on the left (classical system) one has to pick up the entire web to get something changed, whereas on the right (this system herein, 3D simplification and quasi-perfect orthogonality exaggeration) one may change one dimension without having to worry too much about all the other parts' inter-connections. This orthogonality and the absence of the typical "tool-chain" were among the main reasons for it to be relatively easy, for instance, to add tele-operation, simulation and other features to the developed system. On the opposite side, classical systems' features all hang on the tool-chain's web or on the internal tool software matrix. Picking on one features inevitably moves with all the others in more or less extent (the blue arrows represent the "movement/changes").

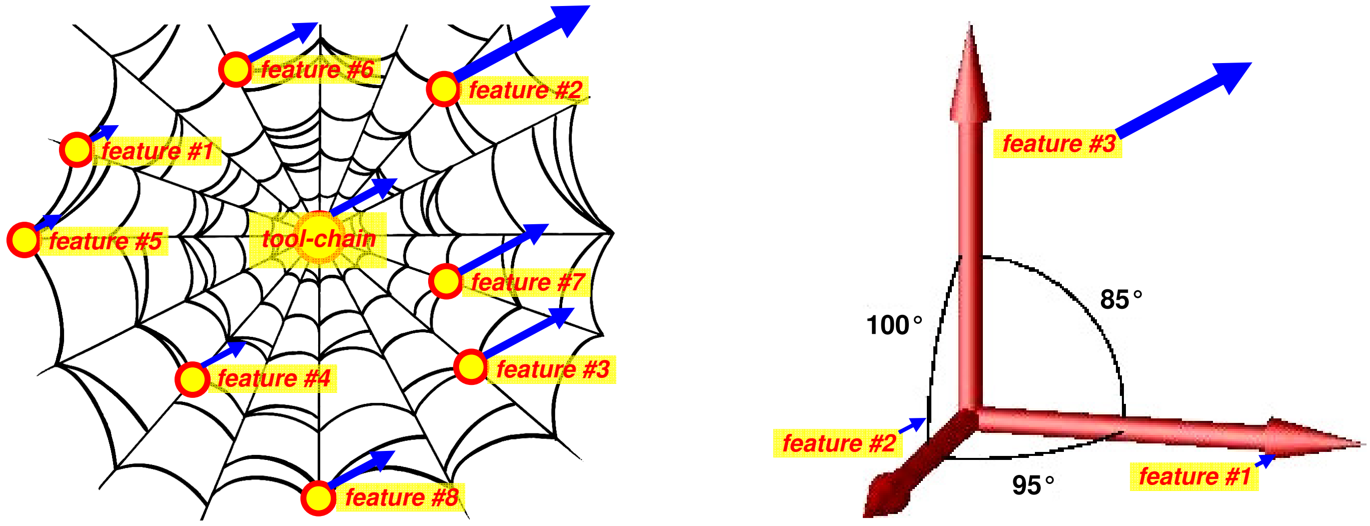


Fig. 7.1:2 – Classical inter-woven and inter-synched system (left) vs. orthogonal and asynchronous work herein (right)

Keeping the previous paragraph in mind, it must also be noted that classical parallel processing systems most often intrinsically exhibit multiple cores running difficult to separate, inter-woven, merged, inter-locked and highly inter-synched code and data. The proposed system, on the other hand, explicitly exhibits asynchronous and self-contained sub-components. While classical systems suffer immediate fatal consequences if some sub-component is mishandled/misplaced or fails, the system herein is intrinsically made of components that may be manipulated virtually independently from each other as long as those 3 heuristics rules illustrated in Fig. 5.2:70 are obeyed. This system even tolerates disobedience to those rules, displaying "graceful degradation" without typical "reset/crash" reactions of classical systems. Fig. [Att. 51]:165 finally illustrates the successive stages of independence and self-containment of the various components comprising this system herein. The proposed model and resulting system possesses separate FDEFs being executed in really separate processing units. Furthermore, as already detailed earlier, this processing and the corresponding data exchanges operate in an asynchronous way, unlike the mostly mandatorily synchronised classical systems. This makes processing distribution and load-balancing easier to implement. It is from the very smallest operation, the *"Macro"*, that virtually everything works independently.

Besides all of the above, unnecessary "abstraction losses" were also kept to a bare minimum, especially by the *"Macros"* directly executed by the *"FDEF-Processor"*, where the highest layer is readily understood by the lowest layer. The virtual elimination of all "abstraction losses" and related classical mechanisms allowed for the disappearance or minimisation of most pitfalls exhaustively listed in Tab. 1.4:21. As can be seen there, virtually all pitfalls were totally or, at least, partially, eliminated. This minimisation of "abstraction losses" adds up to the previously reminded advantages of orthogonality, independence and asynchronicity, among others.

Finally, the fact that this project used always a common component as the main structural building part, the *"Cellular-ECU"*, reduced the manpower needed to harness such a complete system as the *"ECU2010"* is herein. See [Att. 57] for more details about the numbers related to the *"ECU2010"* team.

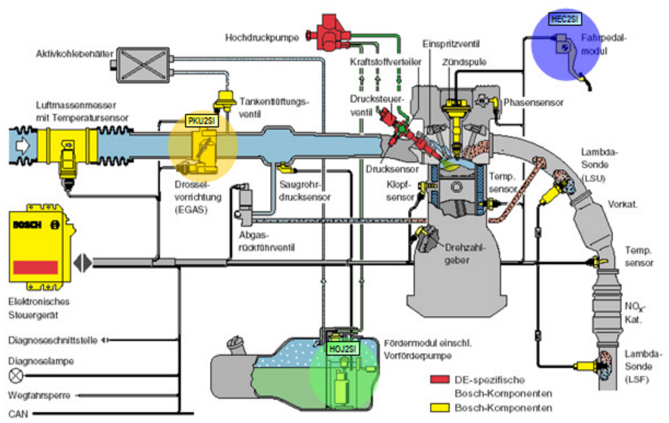
FINAL NOTE: Classical automotive systems rely on re-use of legacy modules and structures, therefore requiring a growing number of developers to keep everything working correctly. Growing complexity and over-engineering brings manpower and documentation needs with it, up to a point of pre-collapse.

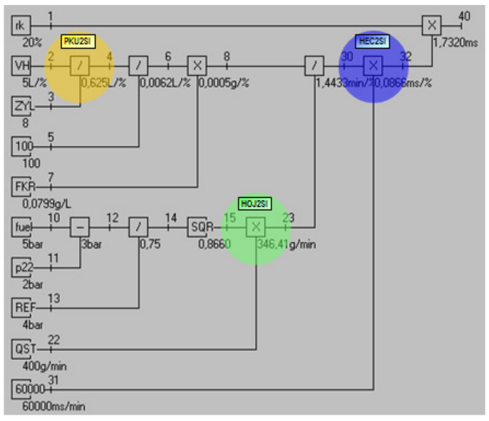
From the statements above and from all considerations concering the effective results vs. manpower/documentation ratio, ***PITFALL C#81*** (Manpower & Documentation Needs), ***PITFALL H#55*** (Abstraction Losses) nd ***PITFALL O#84*** (Over-Engineering) are thus effectively minimized through efficient management of the best Engineers working on a lean and clean architecture especifically conceived and developed to easily handle automotive systems and all their neede features.

7.2 Future Enhancements & Extensions

Based on the proposed model, there are quite a few enhancements to the proof-of-concept developed system, which would further increase its efficiency and ease of handling. Many of these things were not done due to the natural lack of time and resources to complete at this time. A short list of extra features is presented here, while only exposing these ideas in a very concise way just to give an idea of these possible extensions and enhancements that can be done as future work:

- ***"LIVE Source-Code Management" extension*** → Usually, classical management of source-file versions is done in separate tools. Module management and comparisons are done at file-level, where each module usually has more than one file (producing confusion and potential inconsistencies with fatal consequences), with non-adapted tools (with a totally different look and feel compared to the development tool at hand, producing unnecessary stress, turnaround delays/mistakes and especially the need to fit the automotive development system into it somehow). Doing this versioning management directly through the *"iEditor"* would be potentially much more adapted and direct to use, by integrating this functionality directly into the "Function View". Full development and automotive context would be maintained. It would be yet another *"Live"* experience. The next illustrations show how this could be done at various levels (highlighting user changes made on the exact spots they were made, displaying users currently performing changes on the system's components/FDEFs).

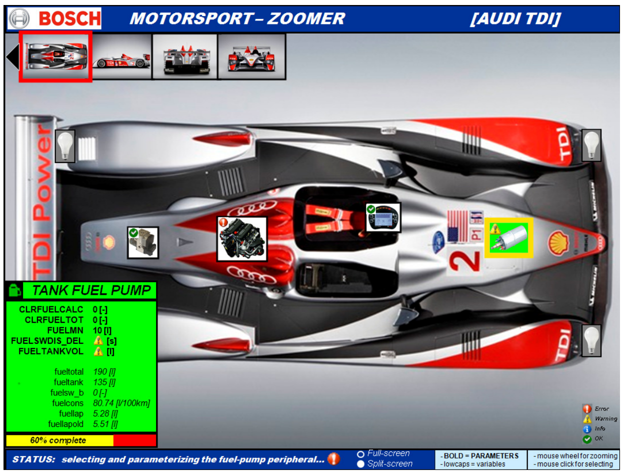


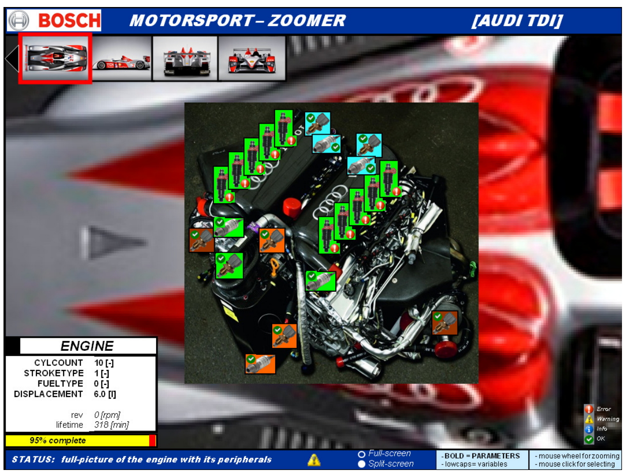


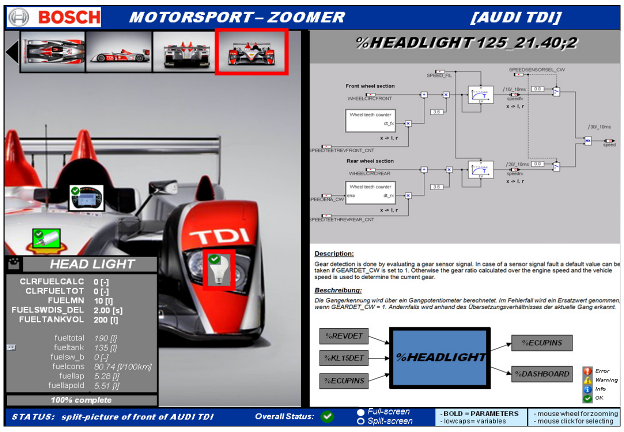
- ***"Reverse Execution" extension*** → A potentially very useful feature during debugging actions, would certainly be this one. It would allow to avoid that dreadful need to restart a program if debugging already went too far or if some previous operations must be double-checked. But this feature is known to be a very difficult one to achieve on classical systems, burdened by all sorts of context/state-producing accessory mechanisms such as caches, flags, pipelines, stacks and many others. [125] [126] [127] show reversed programs at Assembly level. [128] illustrates reverse execution possibilities through history-keeping and isolated inverse-instruction finding. Since this work herein does not have virtually any of the above mentioned accessory mechanisms, it would be reasonable to expect that reverse execution might be not so difficult to achieve with the *"Macros"*. It could then allow the developer to execute back and forth.

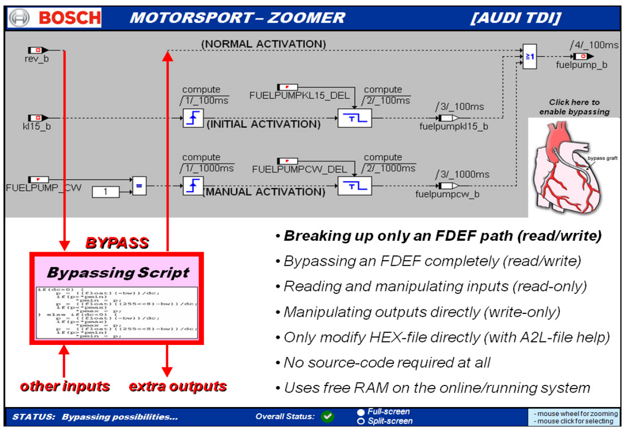
- ***"Micro-Reboot" extension*** → As shown in this thesis, *"Macros"* are executed as if each one of them was the very first being given to the *"Macro-Processor"*. This "micro-rebooting" could be extended to other hardware components, such as the "Inter-Communications" manager, the "Peripheral-Manager" and others. Although these already possess simple internal state-machines, this could be further enhanced in this way. Note that it is not a matter of repeatedly externally resetting these mechanisms, but more of an intrinsic re-start of the process of transferring a *"Node"* or whatever operations they have to fulfil inside the *"Cellular-ECUs"*. This guarantees better "hanging" or "bug propagation" avoidance, since those mechanisms would be repeatedly re-started and context repeatedly reset. Several fully "micro-rebooted" *"Cellular ECUs"* would be even more reliable in terms of time period without any of the typical hanging and bug-propagation problems of large/complex classical systems.

- ***"Live-Vehicle" & "Live-Pan/Zoom" enhancement*** → A potentially interesting side-experiment was done an named "Motorsport Zoomer" [555], to demonstrate the handling enhancement that a seamless zooming capability along with other visual details would bring to the already explained *"Live-Vehicle"* and "Live-Pan/Zoom" features. An extra sub-feature would also be a special split-screen modus showing both the vehicle/peripheral and the corresponding FDEF. Another sub-feature would also be an overall clickable FDEFs hierarchy showing the neighbouring FDEFs relative to the one currently being displayed. Visual bypassing capabilities round up this enhancement. The next screenshots illustrate these possibilities.



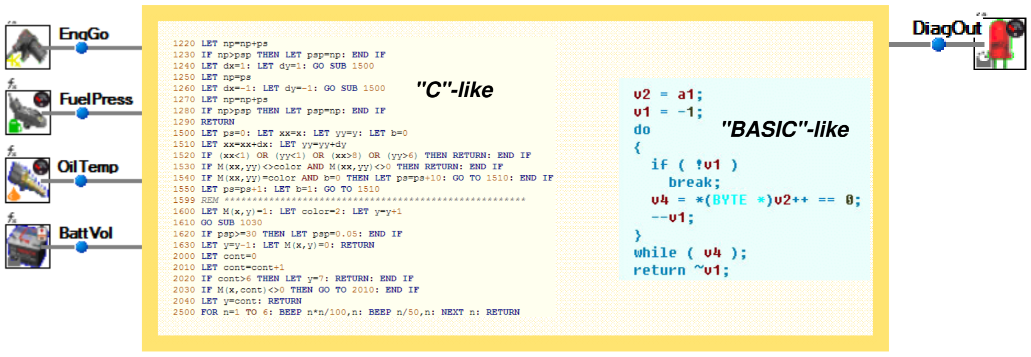






- ***Breaking up only an FDEF path (read/write)***
- *Bypassing an FDEF completely (read/write)*
- *Reading and manipulating inputs (read-only)*
- *Manipulating outputs directly (write-only)*
- *Only modify HEX-file directly (with A2L-file help)*
- *No source-code required at all*
- *Uses free RAM on the online/running system*

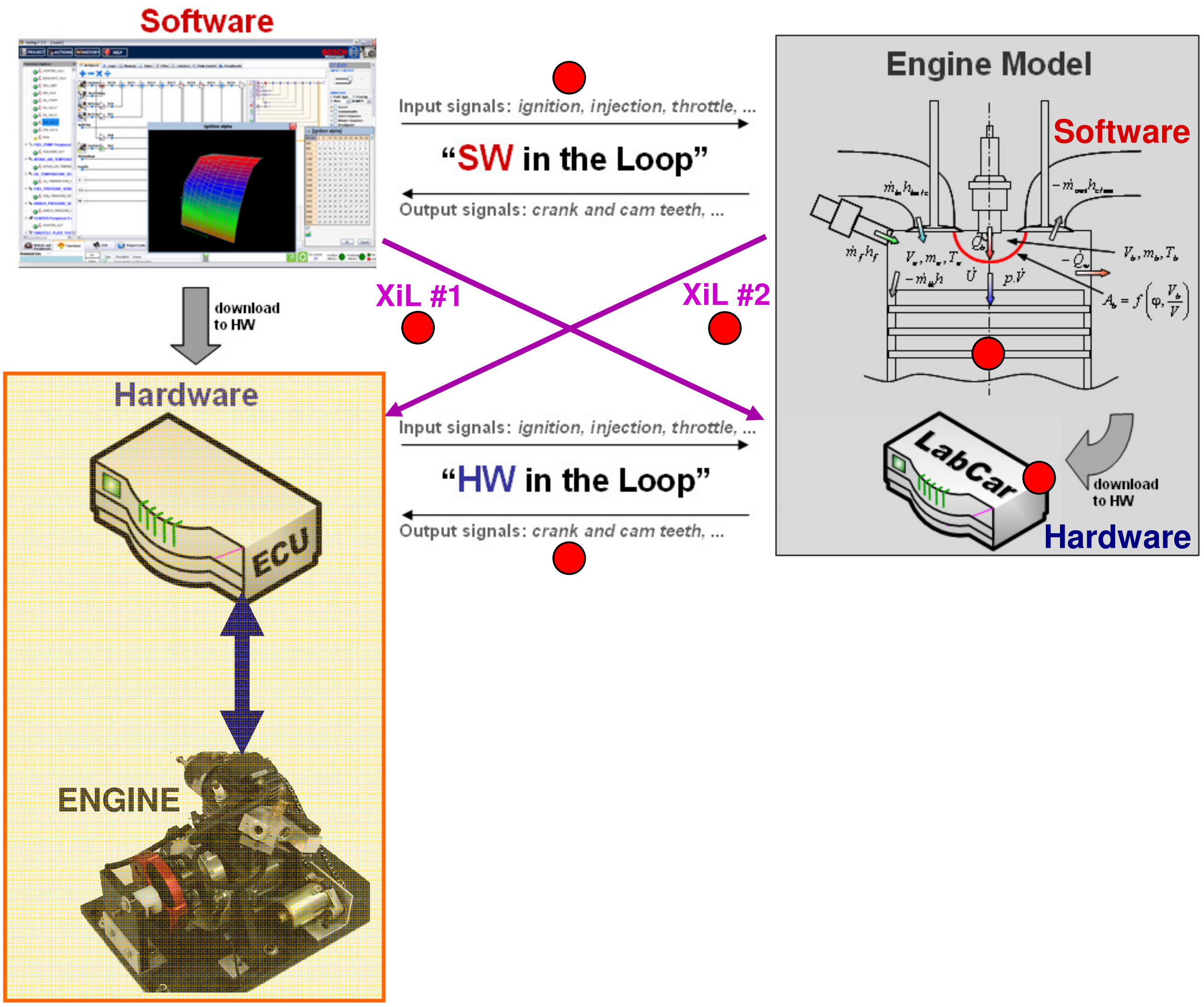
- ***"FDEF Hierarchies & Embedded Script" enhancement*** → The *"Macros Modifier"* bits shown in Fig. 4.5:32, also included a reserved "hierarchy bit" for future use. This bit would thus be used for the *"iEditor"* to know where a hierarchy begins and ends. Hierarchies are important abstraction and organizing possibilities for larger FDEFs and were not implemented in this work herein, due to lack of time. Another idea to further enhance the usage of hierarchies, would be to allow to place simplified "BASIC"/"C"-like textual scripting code into a hierarchy box. This script would appear as it classically is (text-only) and would take input *"Nodes"* and output *"Nodes"* as the working data. Internal parameters or constants would maybe also be possible. Nevertheless, this should be a simple scripting possibility only, for more algorithmic processing that is cumbersome to map onto "data-flow" programming with *"Macros"*. This is not new and has been used in [419] (Fig. 6.3:26), [386] (Fig. [Att. 6]:4) as well as in Matlab Simulink [5]. The next illustration shows how this would be visualized when zoomed-in, inside the Function-Editor.



```
1220 LET np=np+ps
1230 IF np>psp THEN LET psp=np: END IF
1240 LET dx=1: LET dy=1: GO SUB 1500
1250 LET np=ps
1260 LET dx=-1: LET dy=-1: GO SUB 1500
1270 LET np=np+ps
1280 IF np>psp THEN LET psp=np: END IF
1290 RETURN
1500 LET ps=0: LET xx=x: LET yy=y: LET b=0
1510 LET xx=xx+dx: LET yy=yy+dy
1520 IF (xx<1) OR (yy<1) OR (xx>8) OR (yy>6) THEN RETURN: END IF
1530 IF M(xx,yy)<>color AND M(xx,yy)<>0 THEN RETURN: END IF
1540 IF M(xx,yy)=color AND b=0 THEN LET ps=ps+10: GO TO 1510: END IF
1550 LET ps=ps+1: LET b=1: GO TO 1510
1599 REM ******************************************************
1600 LET M(x,y)=1: LET color=2: LET y=y+1
1610 GO SUB 1030
1620 IF psp>=30 THEN LET psp=0.05: END IF
1630 LET y=y-1: LET M(x,y)=0: RETURN
2000 LET cont=0
2010 LET cont=cont+1
2020 IF cont>6 THEN LET y=7: RETURN: END IF
2030 IF M(x,cont)<>0 THEN GO TO 2010: END IF
2040 LET y=cont: RETURN
2500 FOR n=1 TO 6: BEEP n*n/100,n: BEEP n/50,n: NEXT n: RETURN
```

```
v2 = a1;
v1 = -1;
do
{
  if ( !v1 )
    break;
  v4 = *(BYTE *)v2++ == 0;
  --v1;
}
while ( v4 );
return ~v1;
```

- ***"LabCar/Hybrids" extension*** → LabCars (automotive HW simulators with SW modells) consist of fast HW generating signals for the attached ECU. ETAS LabCars [92] use ASCET-SD [7] models, which simulate a vehicle/engine. The *"Cellular-ECU"* concept would have, aside from all other uses, yet another application: *"Macros"* could be used for these automotive models. It seems quite natural for the work developed herein to also be potentially used in non-realtime SiL (SW in the Loop – both control and modelling FDEFs inside the *"iEditor"*), real-time HiL (HW in the Loop – both control and modelling inside *"Cellular-ECU"* modules), XiL (mixed SW/HW combination in the Loop – SW control with HW target or HW control with SW modelling). Upon *"Live-Simulation"* it was already said that peripheral inputs could easily be simulated through playback within *"Live-Generation"* (mixed simulation). This extension to all SW/HW mixing/simulation possibilities would be named *"Live-Modelling"*. Non-realtime ***XiL #2*** refers to the possibility of having a modelling HW (LabCar) and a controlling SW (the *"iEditor"* then overrides the HW's/LabCar's inputs with the controller's SW output *"Nodes"*, while routing the HW's outputs into the controller's SW input *"Nodes"*, all this through USB). Non-realtime ***XiL #1*** refers to the possibility of having a controlling HW and a model in SW (the *"iEditor"* then overrides the HW's inputs with the model's SW output *"Nodes"*, while routing the HW's outputs into the model's SW input *"Nodes"*, all this through USB). In the meanwhile, the orange box represents what this work achieved: *"Cellular-ECU"* modules controlling a real engine plant. [399] puts SiL, MiL and HiL realities in perspective (INTECRIO [69] integrative solution), also merging MiL with SiL, just as done herein. But instead of having adaptation code everywhere, to cope with those different SW/HW sources, this solution would be truly integrated into one single SW/HW platform combination (*"iEditor" + "Cellular-ECU"*). The next illustration shows these HW/SW combinations, while the individual extensions mentioned herein, are shown with ● at the appropriate spots.


Software
download to HW
Hardware
ECU
ENGINE
Input signals: ignition, injection, throttle, ...
"SW in the Loop"
Output signals: crank and cam teeth, ...
XiL #1
XiL #2
Input signals: ignition, injection, throttle, ...
"HW in the Loop"
Output signals: crank and cam teeth, ...
Engine Model
Software
LabCar
download to HW
Hardware

- ***Automatic "Macros-Sequence" Validation enhancements*** → One thing that is readily seen in Matlab Simulink [5] is the integrated possibility of setting maximum/minimum values for the operations' outputs, thus allowing for automatic diagnostic of a model/FDEF (for simulations only, not for real hardware processing). Although this mechanism only operates for simulations and not in real hardware, it already represents a good way of pre-validation for new models. Something similar could be implemented into the *"iEditor"* herein, through the use of an additional *"Macro Modifier"* type or simply by expanding the functionality of the already existing (7.63864 **"CHANGED"**) modifier (see Fig. 4.5:44 and Fig. 4.5:46). Either way, the only intrusion-less implementational mechanism would be to have each *"Node"* inside the *"GIMy"* additionally having embedded maximum/minimum values, which would then be automatically and seamlessly supervise the *"Node's"* actual value. If a *"Node's"* value happened to surpassed any of the limits, the *"GIMy"* would set a bit inside the offending *"Node"*, which would in turn be automatically detected by the *"iEditor"* in a non-realtime fashion, just as the "changed" modifier does (to avoid processing interference). Programmatically, this could be achieved by adding this possibility to the bits in Fig. 4.5:48. Then, the additionally desired maximum/minimum supervision values could be deposited as normal *"Nodes"* alongside the already existing main *"Nodes"* and selectively used by the *"GIMy"* for this values supervision/validation. Once again, to simplify and avoid any interferences upon the *"Macros-Processor"*, these additional maximum/minimum *"Nodes"* would not even appear on the *"Macros"* internal bit representation (Fig. 4.5:31 and Fig. 4.5:32), remaining a strictly hidden *"GIMy"* matter. The *"iEditor"* will then list a separate log for value check violations occurring during FDEF execution. Additionally to this log, debugging efforts may be also enhanced through the *"iEditor"* stopping the *"Macros-Processor"* on the FDEF spots of occurrence of a desired maximum/minimum values check violation, so the user/developer can immeediately take action upon that type of FDEF verification failures. Of course, this mechanism would also work seamlessly the very same way for *"Live-Simulation"*.
- ***Remaining "iEditor" Menus/TABs enhancements*** → Missing inside the *"iEditor"*, secondary roles are still needed such as: version-control (configuration-management), change-request management (corrections/bugs, new developments), projects-management (clients, programs, variants), system-cost and system-power needs estimation, editor-access control (for users and developers alike), tele-assistance, automatic function-manual PDF generation, among some other ideas. No explicit pin-layout management is needed due to the flexible digital-bus.

- ***"Multi-Data-Page" enhancement*** → Current calibration tools allow for two calibration data pages: working and reference pages. The first allows the user to calibrate "live", while the second allows to "freeze" those parameters into FLASH memory. Besides having to implement this mechanism also into the "ECU2010" system, it would be nice to have multiple pages to calibrate multiple parameter-sets, thereby allowing multiple tests or driving sets inside the same ECU, by simply switching among them without any further download overheads. Multiple advanced/complementing "code-pages" could also enhance automotive development, testing and motorsport racing quite a lot with switchable *"Macros-Sequences"* inside the ECU.

Further, more futuristic and theoretical possibilities may be seen in [Att. 58]. Since these are outside the main context of the work done in this thesis, they are not listed in this section. Nevertheless, those extra ideas represent some very interesting things that are worth considering. They could also be tested or implemented and may even probably go mainstream in the medium distanced future, as a response to parallel processing challenges and especially to limitations posed by currently used software and hardware architectures. Some of these ideas relate to well-known present but seemingly unrelated technologies, where a sort of "merge" with the work done in this thesis, is presented.

7.3 Last Words

It is always very interesting to note that, in the meanwhile, manufacturers tend to follow some similar architecture features as those found in the presented thesis, to produce their latest products. Bosch Motorsport is now using a single high-performance and configurable hardware platform for all its products, ranging from differently sized ECUs and dataloggers, to display units. Keeping one single base-platform from which all products derive, is simpler and cost-effective. Besides, the software is similar, while the tools are the same, therefore also managing it all with a smaller and more efficient team. A common platform for all, had been strongly criticised at the beginning of the ECU2010 project.

Another initially highly criticised feature at Bosch Motorsport was the isolated USB connection, now normally used for datalog readout for high-end dataloggers.

Yet another feature similar to the ECU2010 system was the *"Live-Vehicle"* view, but only from the icons perspective (see Fig. 2.1:49).

These were thus the most prominent features taken by Bosch Motorsport for their newer systems, among other less important ones. One feature that was developed by Robert Bosch at about the same time as in this ECU2010 project, was the combined *"Live-Debugging"* and *"Live-Calibrating"*, in that FDEFs in their PDF function-manuals would also have the possibility of "interactively" displaying and calibrating values directly over them, instead of using the "passive" classical INCA [93] software (Fig. 2.1:32).

In a more architectural perspective, when Sony developed and presented the PS3 [556] graphics processing hardware back in 2006, a new milestone was set in terms of raw graphical processing power by parallel means. Due to the complexity of programming a cell-architecture with a central CPU, multilevel L1/L2 caches, classic programming languages, the correspondingly complex tool-chains and all the usual pitfalls, Sony gave up on the PS3 cell-architecture and went back to a single-CPU powerhouse. Others were also developing highly parallel architectures, such as IBM with its "cell architecture".

Some of these and other architectures boasted a high-speed ring-bus very similar to the one presented herein as well. Thus, it was then interesting to note that the trend was already clearly going into pseudo-homogeneous multi-core parallel processing schemes with ring-busses and a main CPU controlling the rest, while this work developed herein went into having no central CPU at all, but only a pool of equal and same-levelled *"Cellular-ECU"* modules. Nevertheless, it must be stressed that the *"Cellular-ECU"* name and concept was developed prior to this knowledge, only confirming that this was a good direction to follow. Not to mention the more dynamic research scenes of the industry also attempting to produce highly advanced physical solutions [557] with several process/CAD/EDA-tool/methodology attempts [558] as well.

The following pictures in Fig. 7.3:3 illustrated some of the architecture concepts discussed in the previous paragraphs. Larrabee's, PS3's, IBM's, Intel's and AMD's "ring-busses" are especially interesting due to strong resemblance to the "ring-bus" implemented in the work herein, with all the processing "nodes" or "cells" connected to it.

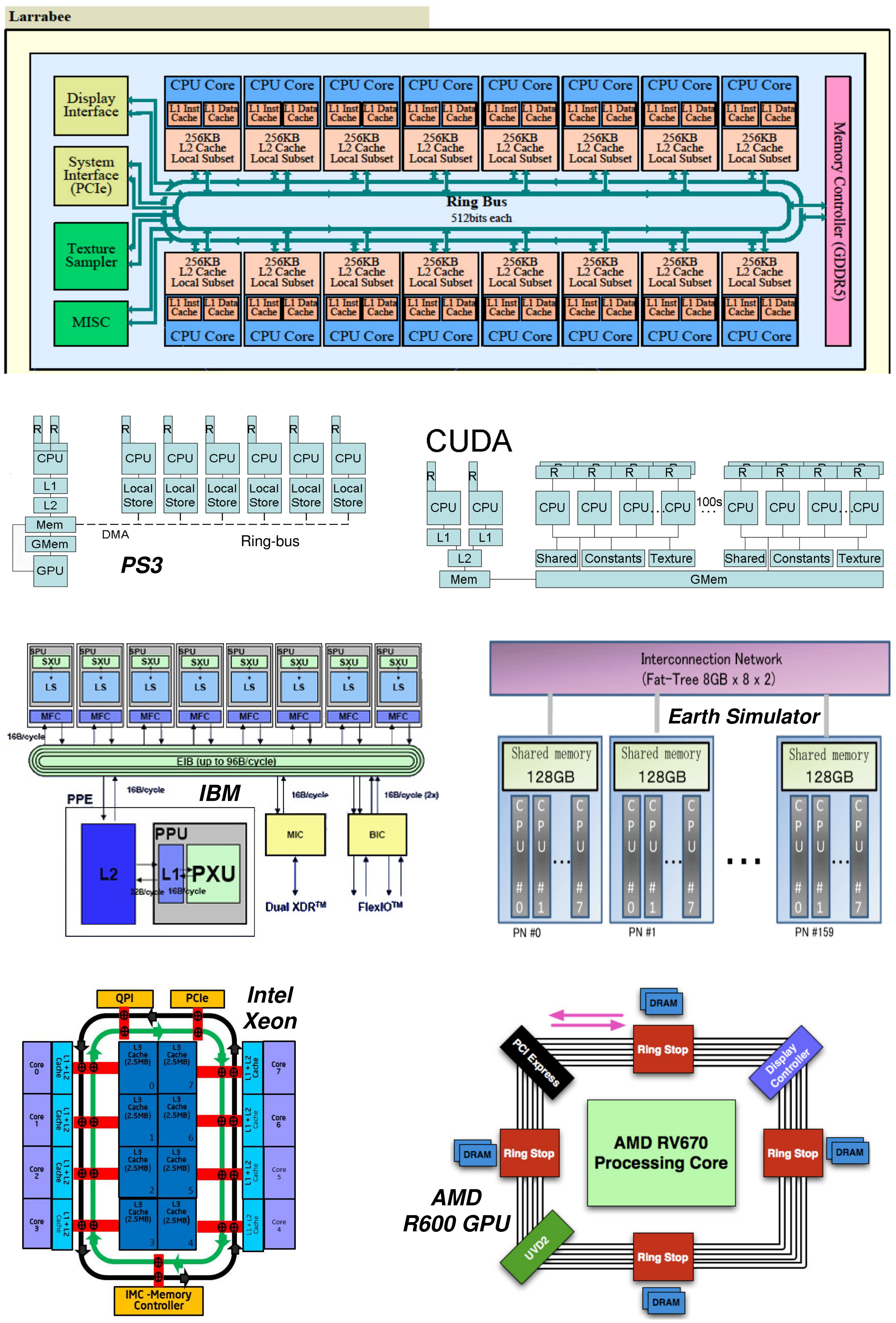


Fig. 7.3:3 – Some of the parallel processing architectures used in past and today's high-end equipments

Page intentionally left blank

Page intentionally left blank

Page intentionally left blank

REFERENCES & BIBLIOGRAPHY

[1] J. Friedman, "MATLAB/Simulink for Automotive Systems Design", Design, Automation and Test in Europe, 2006. DATE '06. Proceedings Volume 1, 6-10 March 2006 Page(s): 1-2.

[2] S. Sims, R. Cleaveland, K. Butts, S. Ranville, "Automated validation of software models", Automated Software Engineering, ASE2001 Proceedings 16. Annual International Conference 26-29 Nov.2001, Pages: 91-96.

[3] M. Kuhl, C. Reichmann, I. Protel, K. Muller-Glaser, "From object oriented modeling to code generation for rapid prototyping of embedded electronic systems", Rapid System Prototyping, 2002 Proceedings, 13th IEEE International Workshop on 1-3 July 2002 Page(s): 108-114.

[4] dSpace GmbH, 2012, www.dspaceinc.com.

[5] The MathWorks Inc., "MATLAB Simulink", 2014, www.mathworks.com.

[6] National Instruments, "LabVIEW - Graphical Programming Environment", 2010, www.ni.com.

[7] ETAS GmbH, "ASCET Software Family", 2014, www.etas.com.

[8] H. Randriamparany, B. Ibrahim, "Seamless integration of control flow and data flow in a visual language", Computer Systems and Applications, ACS/IEEE International Conference on. 2001, 25-29 June 2001 Page(s): 428-434.

[9] U. Honekamp, J. Reidel, K. Werther, T. Zurawka, T. Beck, "Component-node-network: three levels of optimized code generation with ASCET-SD", Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on 22-27 Aug. 1999 Page(s): 243-248.

[10] Bosch Motorsport, "MS5.x ECU family", www.bosch-motorsport.com.

[11] Magneti-Marelli, "FastPRO ECU family", www.magnetimarelli.com, motorsport.magnetimarelli.com.

[12] W. McKeeman, "Language directed computer design", AFIPS Joint Computer Conferences Proceedings, November 14-16, 1967, Page(s): 413-417.

[13] National Instruments, "Neue Funktionen für LabVIEW, parallel, drahtlos und in Echtzeit", 2009, www.ni.com.

[14] L. Baresi, S. Carmeli, A. Monti, M. Pezzè, "PLC Programming Languages: a formal approach", Dipartimento di Elettronica e Informazione - Politecnico di Milano, Italy.

[15] Triangle Research International, Inc., "The Ladder + BASIC Programming Language", 2010, canada.tri-plc.com/trilogiintro.htm.

[16] J. Wright Jr., "The Debate Over Which PLC Programming Language is the State-of-the-Art", Journal of Industrial Technology, Vol. 15, Number 4 - August 1999 to October 1999, Page(s): 1-5.

[17] M. Petre, "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming", Comm. Of the ACM, June 1995, pp. 33-44.

[18] A. Ko, B. Myers, A. Htet, "Six Learning Barriers in End-User Programming Systems", Visual Languages and Human Centric Computing, 2004 IEEE Symposium 30-30 Sept. 2004 Page(s): 199-206.

[19] R. Bischoff, A. Kazi, M. Seyfarth, "The MORPHA style guide for icon-based programming", Robot and Human Interactive Communication, 2002. Proc. 11th IEEE International Workshop on 25-27 Sep.2002 Page(s):482 - 487.

[20] MORPHA Konsortium - Kommunikation, Interaktion und Kooperation zwischen Menschen und intelligenten anthropomorphen Assistenzsystemen, www.morpha.de.

[21] International Electrotechnical Comission, "IEC 61131-3 standard for PLC programming languages", www.iec.ch.

[22] Smart Software Solutions, CoDeSys, www.3s-software.com.

[23] National Instruments, "LabView Robotics 2009", 2009, www.ni.com.

[24] P. Turley, M. Wright, "Developing engine test software in LabVIEW", AUTOTESTCON '97. 1997 IEEE Autotestcon Proceedings 22-25 Sept. 1997 Page(s): 575 - 579.

[25] A. Garcia-Cerezo, J. Gomez-de-Gabriel, J. Fernandez-Lozano, A. Mandow, V. Munoz, F. Vidal-Verdu, K. Janschek, "Using LEGO robots with LabVIEW for a Summer School on Mechatronics", Mechatronics, 2009. ICM 2009. IEEE International Conference on 14-17 April 2009 Page(s): 1 - 6

[26] A. Eppinger, "ASCET - computer aided simulation of engine functions", Computer Aided Engineering of Automotive Electronics, IEE Colloquium on 27 Apr 1994 Page(s): 5/1 - 513.

[27] T. Glötzner, "IEC 61508 Certification of a Code Generator", System Safety, 2008 3rd IET International Conference on 20-22 Oct. 2008 p1-4.

[28] P. Corke, "A robotics toolbox for MATLAB", Robotics & Automation Magazine, IEEE Volume 3, Issue 1, March 1996 Page(s): 24 - 32.

[29] S. Josifovska, "The father of LabView", IEEE Review, Vol. 49, Issue 9, Sept. 2003 Page(s): 30-33.

[30] S. Liebetrau, J. Werner, "Dynamische und interaktive Softwaredokumentation (FDEF live)", Bosch Engineering GmbH, Verfahrensentwicklung BEG-PG/EAF, Infobrief Powertrain Verfahrensentwicklung, 2009.

[31] Ecava, "Integraxor^{TM}", 2010, www.ecava.com / www.integraxor.com.

[32] Azeotech, "DAQFactory SCADA", 2010, www.azeotech.com.

[33] IEEE Power Engineering Society, "IEEE Standard for Scada and Automation Systems", May 2008 Pages:c1-133.

[34] Azeotech, " AzeoTech DAQFactory v5.80 User's Guide", 2010, www.azeotech.com.

[35] LEGO, "LEGO Mindstorms NXT", 2010, www.legomindstorms.com.

[36] J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, A. Repenning, "LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick", Proceeding of Visual Languages, Darmstadt, Germany, IEEE Computer Society Press, 1995, Page(s): 172 – 179.

[37] National Instruments, "LEGO MINDSTORMS, Powered by NI LabVIEW", 2010, www.ni.com.

[38] Actor-Lab, 2010, actor-lab.open.ac.uk/.

[39] LEGO RoboLab online, 2010, www.robolabonline.com

[40] E. Wang, R. Wang, "Using Legos and RoboLab (LabVIEW) with elementary school children", Frontiers in Education Conference, 2001. 31st Annual Volume 1, 10-13 Oct. 2001 Page(s):T2E - T11 vol.1.

[41] P. Whalley, "Representing Parallelism in a Control Language Designed for Young Children", Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006, IEEE Symposium on 4-8 Sept. 2006, Page(s): 173 - 176

[42] H. Lund, L. Pagliarini, "RoboCup Jr. with LEGO MINDSTORMS", Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on Volume 1, 24-28 April 2000 Page(s): 813 - 819 vol.1.

[43] A. Brandt, M. Colton, "Toys in the Classroom: LEGO MindStorms as an Educational Haptics Platform", Haptic interfaces for virtual environment and teleoperator systems, Symp. Haptics 13-14 March 2008 Pages: 389 - 395.

[44] Seung Han Kim, Jae Wook Jeon, "Using visual programming kit and Lego Mindstorms: An introduction to embedded system", Industrial Technology, 2008. ICIT 2008. IEEE Intern. Confer. 21-24 April 2008 Pages: 1-6.

[45] P. Cox, T. Smedley, "Visual programming for robot control", Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on 1-4 Sept. 1998 Page(s): 217 - 224.

[46] Steve McConnel, "10 Best Influences on Software Engineering", www.stevemcconnell.com.

[47] Borland International, Inc., "Turbo Pascal 7", 2010, info.borland.com/pascal.

[48] L. Xie, A. Antle, N. Motamedi, "Are tangibles more fun?: comparing children's enjoyment and engagement using physical, graphical and tangible user interfaces", February 2008, TEI '08: Proceedings of the 2nd international conference on Tangible and embedded interaction.

[49] E.S. Katterfeldt, H. Schelhowe, "A modelling tool to support children making their ideas work", June 2008, IDC '08: Proceedings of the 7th international conference on Interaction design and children.

[50] Fischertechnik, "ROBO Pro", 2010, www.fischertechnik.de.

[51] Microsoft Corporation, "Microsoft Robotics Developer Studio" powered by "Microsoft Visual Programming Language", 2010, www.microsoft.com.

[52] A. Begel, "LogoBlocks: A Graphical Programming Language for Interacting with the World", Epistemology and Learning Group, MIT Media Laboratory, May 24, 1996.

[53] U. Müller, "Fishface40.DCU - Handbuch zu Version 4.0", FTComputing, 2010, www.ftcomputing.de, www.ulrich-mueller.de.

[54] Fischertechnik, "Lucky Logic (for Windows) - LLWIN", 2010, Wikipedia, de.wikipedia.org/wiki/Lucky_Logic and U. Müller, www.ftcomputing.de.

[55] Edventures, "PCS Brain" featuring "PCS Visual Logo", www.edventures.com.

[56] M. Resnick, S. Ocko, S. Papert, "Lego, Logo and Design", Media Laboratory, Massachusetts Institute of Technology (MIT), 1988, Children's Environments Quarterly, vol. 5 (4).

[57] LEGO Education WeDo, "Hungry Alligator Teacher Notes", 2010, www.lego.com/education.

[58] B. Andrew, "LogoBlocks - A Graphical Programming Language for Interacting with the World", MIT Media Laboratory, May 1996.

[59] Massachusetts Institute of Technology (MIT), "SCRATCH", 2010, Media Laboratory, scratch.mit.edu.

[60] Massachusetts Institute of Technology (MIT), "StarLogo TNG", Media Laboratory, 2010, education.mit.edu/starlogo-tng.

[61] Imagine S.A., "AMESim", www.amesim.com.

[62] Chen Long, Niu Li-min, Zhao Jing-bo, Jiang Hao-bin, "Application of AMESim & MATLAB simulation on vehicle chassis system dynamics", Intelligent Information Technology Application, Workshop on 2-3 Dec. 2007 Page(s): 185 - 188.

[63] Zhang Dong-xu, Zeng Xiao-hua, Wang Peng-yu, Wang Qing-nian, "Co-simulation with AMESim and MATLAB for differential dynamic coupling of Hybrid Electric Vehicle", Intelligent Vehicles Symposium, 2009 IEEE, 3-5 June 2009 Page(s): 761 - 765.

[64] National Instruments, "MATRIXx – System Design and Control Development Software", 2010, www.ni.com.

[65] MSC Software, "ADAMS - Multibody Dynamics", 2010, www.mscsoftware.com.

[66] P. Redlich, D. Ellis, "Integration of a Full Vehicle ADAMS Model and a MATRIXx Engine Controller for Improved Simulation of the Vehicle+Powertrain Interaction", 12th European ADAMS Users Conference, 18-19 Nov. 1997.

[67] Autosar, "AUTOSAR - Automotive Open System Architecture", 2010, www.autosar.org.

[68] U. Freund, U. Lauff, R. Siwy, H.J. Wolff, D. Ziegenbein, "Sauberer Schnitt - Modellbasierte Entwicklung von Anwendungssoftware mit AUTOSAR-konformen Schnittstellen", ELEKTRONIKNET, 2009 WEKA FACHMEDIEN

GmbH, www.elektroniknet.de.

[69] ETAS, "INTECRIO Software Products", 2010, www.etas.com/en/products/intecrio.php.

[70] Microsoft "Visual-Basic Language Specification", www.microsoft.com.

[71] Microsoft, "C# Language Specification 3.0", www.microsoft.com.

[72] M. Gertz, "Scaling Up: The Very Busy Background Compiler", MSDN Magazine, Microsoft, msdn.microsoft.com.

[73] S. McDirmid, "Living it up with a live programming language", October 2007, OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications.

[74] Microsoft Corporation, 2010, www.microsoft.com.

[75] ECMA International, "Standard ECMA-335 Common Language Infrastructure (CLI)", 4th edition, Partitions I - VI, June 2006, www.ecma-international.org.

[76] ISO International Organization for Standarization, "ISO/IEC 23270 - Programming languages - C#", second edition, September 2009, standards.iso.org.

[77] ECMA International, "Standard ECMA-334 C# Language Specification", 4th edition, June 2006, www.ecma-international.org.

[78] ISO International Organization for Standarization, "ISO/IEC 23271 - Common Language Infrastructure (CLI)", 2nd edition, Partitions I to VI, October 2009, standards.iso.org.

[79] T. Lindholm, F. Yellin, "The JavaTM Virtual Machine Technology Specification", 2.ed., Sun Microsystems, www.sun.com.

[80] Java Community Process, "Community Development of Java Technology Specifications", www.jcp.org.

[81] Sun Microsystems, "The Java HotSpotTM Virtual Machine", technical whitepaper, www.sun.com.

[82] T. Rodriguez, K. Russel, "Client Compiler for the Java HotSpot™ Virtual Machine", Sun's 2002 Worldwide Java Developer Conference, Session 3198, java.sun.com.

[83] R. Newman, C. Dennis, "Everything Java™ Technology... but Better and Faster: The Evolution of JPC", Department of Physics, Oxford University, 2008 JavaOneTM Conference, java.sun.com/javaone.

[84] NestedVm, Binary translation for Java, nestedvm.ibex.org.

[85] C2J Translator, C-programs to Java-programs, www.novosoft-us.com.

[86] Axiomatic Solutions, C to Java source-code, www.axiomsol.com.

[87] S. Malabarba, P. Devanbu, A. Stearns, "MoHCA-Java: a tool for C++ to Java conversion support", Software Engineering, 1999. Proceedings of the 1999 International Conference on 22-22 May 1999 Page(s):650-653.

[88] Gongzhu Hu, A. Gadapa, "Compiling C++ programs to Java bytecode", Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2005 and First ACIS International Workshop on Self-Assembling Wireless Networks. SNPD/SAWN 2005. Sixth Int'l Conference on 23-25 May 2005 Page(s):56 - 61.

[89] The Microengine Company, "WD/90 Pascal MICROENGINE Reference Manual", preliminary edition, 1979.

[90] ARM Holdings, 2010, www.arm.com.

[91] ARM Holdings, "Jazelle - Execution Environment Acceleration by ARM", 2010, www.arm.com.

[92] ETAS GmbH, 2010, www.etas.com.

[93] Robert Bosch GmbH, 2010, www.bosch.com.

[94] ADAC - Allgemeiner Deutscher Automobil Club, 2010, www.adac.de.

[95] National Instruments, "CompactRIO - FPGA-based High-Performance Controller and Chassis", 2010, www.ni.com/compactrio.

[96] National Instruments, "NI LabVIEW FPGA Page", 2010, www.ni.com/fpga.

[97] National Instruments, "NI LabVIEW FPGA Module", Datasheet & Help 2010, www.ni.com

[98] National Instruments, "FlexRIO - FPGA Modules for PXI", 2010, www.ni.com/compactrio.

[99] National Instruments, "IPNet - LabVIEW FPGA Functions and Example IP", Developer Zone 2010, zone.ni.com/devzone.

[100] dSpace, "RTI FPGA Programming Blockset", 2010, www.dspace.com.

[101] P. Frenger, "Hard Java", May 2008, ACM SIGPLAN Notices, Volume 43 Issue 5.

[102] C. Porthouse, "High performance Java on embedded devices - Jazelle DBX technology: ARM acceleration technology for the Java Platform ", Jazelle DBX WhitePaper, October 2005, ARM Limited.

[103] C. Porthouse, "Jazelle for Execution Environments: New execution environment hardware support for compilation techniques", Jazelle RCT WhitePaper, May 2005, ARM Limited.

[104] ARM Limited, "ARM1136JF-S and ARM1136J-S Technical Reference Manual", 2010, infocenter.arm.com.

[105] ARM Limited, "Jazelle Technology for JAVA Applications", ARM DOI 0114-2/05.01, java.epicentertech.com.

[106] ARM Limited, "Jazelle Technology for Execution Environments - High performance & efficient solutions through integrated hardware and software design", ARM DOI 0114-9/05.07.

[107] ARM Limited, "Realview System Generator", ARM DOI 0235-3/07.07, www.arm.com/products/DevTools.

[108] G. Mueller, J. Borzuchowski, "Java and Real Time Storage Applications", 19th IEEE Symposium on Mass Storage Systems, September 2002.

[109] P. Lorchirachoonkul, U. Jitpaisarnsook, "ARM (Advance RISC Machine)", www.cpe.ku.ac.th/~pom/courses/204521/report/2001/Jazelle.ppt.

[110] M. Pohl, "Bericht zu dem Vortrag über Java-Prozessoren ", Rechnerstrukturen, Prof. Dr. Thomas Risse, Studiengang Technische Informatik, 2010, Hochschule Bremen.

[111] W. Puffitsch, M. Schoeberl, "picoJava-II in an FPGA", September 2007, JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems, ACM.

[112] H. McGhan, M. O'Connor, A "PicoJava: a direct execution engine for Java bytecode", Computer Volume 31, Issue 10, Oct. 1998 Page(s):22 - 30.

[113] D. Saougkos, G. Manis, K. Blekas, A. Zarras, "Revisiting Java Bytecode Compression for Embedded and Mobile Computing Environments", Software Engineering, IEEE Transactions on Volume 33, Issue 7, July 2007 Page(s):478 - 495.

[114] H. Lambridge, "Java bytecode optimizations", Compcon '97. Proceedings, IEEE, 23-26 Feb.1997 Pages:206-210.

[115] Object Management Group (OMG), "Unified Modeling Language (UML)", www.uml.org.

[116] L. Brisolara, M. Oliveira, R. Redin, L. Lamb, F. Wagner, "Using UML as Front-end for Heterogeneous Software Code Generation Strategies", Design, Automation and Test in Europe, 2008. DATE '08 10-14 March 2008 Page(s): 504 - 509.

[117] B. Bose, M. Tuna, J. Nagy, "LavaCORE configurable Java processor core", Aerospace Conference Proceedings, 2002. IEEE Volume 4, 2002 Page(s): 4-1953 - 4-1959 vol.4.

[118] dSpace GmbH, "dSpace RapidPro: Full Power", www.dspaceinc.com.

[119] Infineon Technologies, "TriCore Automotive Microcontroller families", www.infineon.com.

[120] Freescale Semiconductor, "MC and MPC Microcontroller families", www.freescale.com.

[121] V. Srivastava, M. Motani, "Cross-layer design: a survey and the road ahead", Communications Magazine, IEEE Volume 43, Issue 12, Dec. 2005 Page(s):112 - 119.

[122] G. CanforaHarman, M. Di Penta. "New Frontiers of Reverse Engineering", May 2007, FOSE '07: 2007 Future of Software Engineering, IEEE Computer Society.

[123] M. van den Brand, P. Klint, C. Verhoef, "Reverse engineering and system renovation - an annotated bibliography", January 1997, SIGSOFT Software Engineering Notes , Volume 22 Issue 1, ACM.

[124] M. Ali, "Why teach reverse engineering?", July 2005, SIGSOFT Software Engineering Notes, Vol 30 Issue 4, ACM.

[125] T. Akgul, V. Mooney III, "Assembly instruction level reverse execution for debugging", April 2004, Transactions on Software Engineering and Methodology (TOSEM) , Volume 13 Issue 2, ACM.

[126] T. Akgul, V. Mooney III, S. Pande, "A Fast Assembly Level Reverse Execution Method via Dynamic Slicing", May 2004, ICSE '04: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society.

[127] T. Akgul, V. Mooney, III "Instruction-level reverse execution for debugging", January 2003, PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, ACM.

[128] B. Biswas, R. Mall, "Reverse execution of programs", April 1999, SIGPLAN Notices, Volume 34 Issue 4, ACM.

[129] T. Farkas, C. Neumann, A. Hinnerichs, "Integrative Approach for Embedded Software Design with UML and Simulink", Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International Volume 2, 20-24 July 2009 Page(s):516-521.

[130] T. Farkas, E. Meiseki, C. Neumann, K. Okano, A. Hinnerichs, S. Kamiya, "Integration of UML with Simulink into embedded software engineering", ICCAS-SICE 2009, August 2009 Page(s): 474-479.

[131] Object Management Group (OMG), "Systems Modeling Language (SyML)", www.omgsysml.org.

[132] C. Reichmann, M. Kiihl, P. Graf, K.D. Muller-Glaser, "GeneralStore - a CASE-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems", Engineering of Computer-Based Systems, 2004 Proceedings. 11th IEEE Int'l Conf. and Workshop on the 24-27 May 2004 Pages: 225-232.

[133] A. Chureau, Y. Savaria, E. Aboulhamid, "The role of model-level transactors and UML in functional prototyping of systems-on-chip: a software-radio application", Design, Automation and Test in Europe, 2005 Proceedings, 2005 Page(s):698 - 703 Vol. 2.

[134] ETAS GmbH, "ETK/XETK Universal ECU Interfaces", www.etas.com.

[135] HiTEX Development Tools, "In-Circuit Emulators", www.hitex.com.

[136] H. Richards Jr., C. Wright, Jr., "Introduction to the SYMBOL 2R programming language", November 1973, Proceedings of the ACM-IEEE symposium on High-level-language computer architecture, ACM.

[137] H. Weber, "A microprogrammed implementation of EULER on IBM system/360 model 30", September 1967, Communications of the ACM , Volume 10 Issue 9, ACM.

[138] W. Cullyer, "Implementing high integrity systems: the VIPER microprocessor", Aerospace and Electronic Systems Magazine, IEEE, Volume 4, Issue 6, June 1989 Page(s):5 - 13.

[139] L. Maguire, G. Irwin, G. Lightbody, "Mapping Control Algorithms onto Transputer Array", IEEE Colloquium on Transputer Applications, 1989, 5/1 - 510.

[140] J. Gray, A. Naylor, A. Abnous, N. Bagherzadeh, "VIPER: a VLIW integer microprocessor", Solid-State Circuits, IEEE Journal of Volume 28, Issue 12, Dec. 1993 Page(s): 1377 - 1382.

[141] J. Gray, A. Naylor, A. Abnous, N. Bagherzadeh, N.;"VIPER: A 25-MHz, 100-MIPS peak VLIW microprocessor", Custom Integrated Circuits Conference, 1993, Proceedings of the IEEE 1993 9-12 May 1993 Pages: 4.1.1-4.1.5.

[142] Dr. Ing. Porsche AG, 2010, www.porsche.de.

[143] Mary Lisbeth d'Amico, "Preise fallen, Komplexität explodiert", Software Fakten und Prognosen, Pictures of the Future, Siemens AG, Fall 2004, www.siemens.de.

[144] Sylvia Trage, "Elektronik treibt Innovationen im Auto voran", Das mitdenkende Auto - Fakten und Prognosen, Pictures of the Future, Siemens AG, Fall 2005, www.siemens.de.

[145] T. Moon, "Vehicle control systems-reliability through simplicity", Colloquium in Safety Critical Software in Vehicle and Traffic Control, IEEE, 1990, Page(s): 3/1 - 310.

[146] Internat Socio-Familial de Luxembourg, "Directly connecting a digital camera module to the RCX", Journal, 2010, www.convict.lu/Jeunes/RCXCam/RCXCam_Journal.htm.

[147] National Highway Traffic Safety Administration, "Defects & Recalls", 2010, www-odi.nhtsa.dot.gov.

[148] INMOS, "The Transputer", 2010, www.inmos.com.

[149] David May (Transputer architect), "David May's Transputer Page", University of Bristol, Department of Computer Science, 2010, www.cs.bris.ac.uk/~dave/transputer.html.

[150] Bosch Motorsport, "MS4.x ECU family", www.bosch-motorsport.com.

[151] INMOS, "Transputer Architecture - Reference Manual", 72-TRN-048-03, July 1987, www.inmos.com.

[152] C. Whitby-Strevens (INMOS Limited, Whitefriars, L. Mead, "The transputer", ACM SIGARCH Computer Architecture News archive, Volume 13, Issue 3, June 1985, Proceedings of the 12th annual international symposium on Computer architecture (ISCA '85) Pages: 292 - 300.

[153] G. Barrett, "OCCAM3 reference manual", draft as of March 31, 1992.

[154] D. May, "Architectures for Ubiquitous Computing", Bristol University, May 2004, www.cs.bris.ac.uk/~dave.

[155] dSpace, "DS1006 Processor Board - Computing power for processing-intensive real-time models", Catalog 2010.

[156] M. Rayment, "Flexible motion control using IEC 61131-3", UKACC Control 2004 Mini Symp. 2004, Pages: 27-36.

[157] D. Witsch, B. Vogel-Heuser, "Close integration between UML and IEC 61131-3: New possibilities through object-oriented extensions", Emerging Technologies & Factory Automation, 2009. ETFA 2009, IEEE Conference, Publication Year: 2009, Page(s): 1 - 6.

[158] M. Bancila, "Class Designer for C++ in Visual Studio 2008", Marius Mancila's blog, 2010, mariusbancila.ro/blog.

[159] P. Vitharana, "Risks and challenges of component-based software development", August 2003, Communications of the ACM, Volume 46 Issue 8.

[160] R. Charette, “This Car Runs on Code”, IEEE Spectrum, February 2009, spectrum.ieee.org.

[161] M. Broy, I. Krüger, A. Pretschner, C. Salzmann, “Engineering Automotive Software”, Proceedings of the IEEE, Vol. 95, No. 2, February 2007.

[162] B. Emaus, “Hitchhiker’s guide to the automotive embedded software universe”, in Proc. Keynote Presentation at ICSE’05, Workshop, International Conference on Software Engineering, May 2005.

[163] J. Bereisa, “Applications of Microcomputers in Automotive Electronics”, Delco Electronics Division, General Motors Corporation, Goleta, CA 93117, This paper appears in: Transactions on Industrial Electronics, IEEE, May 1983, Volume: IE-30, Issue:2, page(s): 87 – 96.

[164] J. Marley, “Evolving Microprocessors which Better Meet the Needs of Automotive Electronics”, Proceedings of the IEEE, Vol. 66, No. 2, February 1978.

[165] dSpace GmbH, "TargetLink – The Production Code Generator from dSpace", www.dspaceinc.com.

[166] dSpace GmbH, “TargetLink – Driving the Future with Autocode”, dSpace MAGAZINE Special Edition, 2010, www.dspace.com.

[167] dSpace GmbH, “TargetLink 3.1 ECU Auto-Coding”, dSpace MAGAZINE, 2010, www.dspace.com.

[168] Geensoft, “AUTOSAR Builder”, version 2010-1a, www.geensoft.com.

[169] The MathWorks Inc., "Simulink-PLC-Coder 1.0", Matlab Release 2010a, www.mathworks.com.

[170] The MathWorks Inc., "Simulink-HDL-Coder 1.7", Matlab Release 2010a, www.mathworks.com.

[171] D. Lenoski, W.D. Weber, "Scalable Shared-Memory Multiprocessing", 1995, Morgan Kaufmann Publishers, Inc., ISBN 1-55860-315-8.

[172] Wikipedia, "Turing Completeness", 2010, en.wikipedia.org/wiki/Turing_complete.

[173] R. Kalla, B. Sinharoy, J. Tendler, "IBM Power5 chip: a dual-core multithreaded processor", Micro, IEEE, Volume: 24, Issue: 2, 2004 , Page(s): 40 – 47.

[174] I. Graham, T. King, "The Transputer Handbook", Prentice Hall, 1990, first publ. in 1947, ISBN: 0-13-929134-2.

[175] R. Cok, "Parallel Programs for the Transputer", Prentice Hall, 1991, ISBN: 0-13-651480-4.

[176] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE Std 754-2008, 2008, Page(s): 1 – 58.

[177] Y. Liang, T. Mitra, "Cache modeling in probabilistic execution time analysis", June 2008, DAC '08: Proceedings of the 45th annual Design Automation Conference.

[178] Infineon Technologies, "C167 Automotive Microcontroller Family", 2010, www.infineon.com.

[179] G. Beltrame, C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, V. Trianni, "An assembly-level execution-time

model for pipelined architectures", November 2001, ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design.

[180] R. Van der Wijngaart, S. Sarukkai, P. Mehra, "The effect of interrupts on software pipeline execution on message-passing architectures", January 1996, ICS '96: Proceedings of the 10th international conference on Supercomputing.

[181] Wikipedia, "Interrupt Handler", 2010, en.wikipedia.org/wiki/Interrupt_handler.

[182] S. Korsholm, M. Schoeberl, A. Ravn, "Interrupt Handlers in Java", 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing, 2008, Page(s): 453-457

[183] Wikipedia, "Lexical Analysis", 2010, en.wikipedia.org/wiki/Lexical_analysis.

[184] Wikipedia, "Syntax Analysis - Parsing", 2010, en.wikipedia.org/wiki/Syntax_analysis.

[185] R. Herk, J. Verhaegh, W. Fontijn, "ESPranto SDK: an adaptive programming environment for tangible applications", April 2009, CHI '09: Proceedings of the 27th international conference on Human factors in computing systems, ACM.

[186] O. Morgan, "Certified Testing of C Compilers for Embedded Systems", 3rd Institution of Engineering and Technology Conference on Automotive Electronics, 2007, Page(s): 1-5.

[187] R. Faiman Jr., A. Koretesoja, "An Optimizing Pascal Compiler", IEEE Transactions on Software Engineering, Volume: SE-6 , Issue: 6, 1980, Page(s): 512-519.

[188] J. Davidson, A. Holler, "Subprogram inlining: a study of its effects on program execution time", IEEE Transactions on Software Engineering, Volume: 18, Issue: 2, 1992, Page(s): 89-102.

[189] D. Ward, "MISRA Standards for Automotive Software", The 2nd IEEE Conference on Automotive Electronics, 2006, Page(s): 5-18.

[190] Wikipedia, "Punched Card", 2010, en.wikipedia.org/wiki/Punched_cards.

[191] Wikipedia, "Processing Speed - IPS", 2010, en.wikipedia.org/wiki/Processing_speed.

[192] National Science Foundation, "Revolutionizing Engineering Science through Simulation", Report of the National Science Foundation Blue Ribbon Panel on Simulation-Based Engineering Science, May 2006, www.nsf.gov.

[193] J. Parkhurst, J. Darringer, B. Grundmann, "From single core to multi-core: preparing for a new exponential", ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design, November 2006, ACM.

[194] R. Paul, "Why users cannot 'get what they want' ", December 1993, SIGOIS Bulletin, Volume 14, Issue 2, ACM.

[195] B. Kastrup, A. Bink, J. Hoogerbrugge, "ConCISe: a compiler-driven CPLD-based instruction set accelerator", FCCM '99. Proceedings. Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1999, Page(s): 92–101.

[196] M. Wirthlin, B. Hutchings, "DISC: A dynamic instruction set computer", Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines, 1995, Page(s): 99–107.

[197] T. Rauscher, A. Agrawala, "Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming", IEEE Transactions on Computers, Volume: C-27, Issue: 11, 1978, Pages: 1006-1014.

[198] A. Wolfe, J. Shen, "Flexible Processors: A Promising Application-specific Processor Design Approach", Proceeding of the 21st Annual Workshop on Microprogramming and Microarchitecture, 1988, Page(s): 30–39.

[199] P. Athanas, H. Silverman, "Processor reconfiguration through instruction-set metamorphosis", Computer, 1993, Volume: 26, Issue: 3, Page(s): 11-18.

[200] Wikipedia, "Stack (data structure)", 2010, en.wikipedia.org/wiki/Stack_(data_structure).

[201] X. Yang, N. Cooprider, J. Regehr, "Eliminating the call stack to save RAM", LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, June 2009, ACM.

[202] J. Chen, M. Smith, C. Young, N. Gloy, "An Analysis of Dynamic Branch Prediction Schemes on System Workloads", 23rd Annual International Symposium on Computer Architecture, 1996, Page(s): 12-21.

[203] Feipei Lai; Chia-Jung Hsieh, "Reducing procedure call overhead: optimizing register usage at procedure calls", International Conference on Parallel and Distributed Systems, 1994, Page(s): 649-654.

[204] R. Horst, D. Jewett, D. Lenoski, "The risk of data corruption in microprocessor-based systems", FTCS-23. Digest of Papers, The Twenty-Third International Symposium on Fault-Tolerant Computing, 1993 , Page(s): 576-585.

[205] C. Feldstein, J. Muzio, "Development of a fault tolerant flight control system", The 23rd Digital Avionics Systems Conference, 2004. DASC 04, Volume: 2, Page(s): 6.E.3 - 61-12.

[206] D. Taylor, "Practical techniques for damage confinement in software", Computer Security, Dependability and Assurance: From Needs to Solutions, Proceedings, 1998, Page(s): 132-143.

[207] Wikipedia, "Operating Systems", 2010, en.wikipedia.org/wiki/Operating_systems.

[208] Intel, "Hyper-Threading Technology (HT)", Intel Technology Journal, Vol. 06, Issue 01, Feb 2002, www.intel.com.

[209] Intel, "Intel Hyper-Threading Technology - Technical User's Guide", January 2003, www.intel.com.

[210] Wikipedia, "Multithreading", 2010, en.wikipedia.org/wiki/Multithreading.

[211] "Effective OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS", Proceedings of the 2007 Asia and South Pacific Design Automation Conference, 2007, Pages: 44-49.

[212] Wikipedia, "Semaphore Programming", 2010, en.wikipedia.org/wiki/Semaphore_(programming).

[213] NORDIC Semiconductor, "nRF24L01+ Single Chip 2.4GHz Transceiver Product Specification v1.0", 2010, www.nordicsemi.com.

[214] Texas instruments, "MSP430F5xx Family with USB", 2010, www.msp430.com.

[215] T. Yukimatsu, T. Furuhashi, Y. Uchikawa, "A fuzzy expert system for hierarchical placement of parts on printed circuit board", Proceedings of the Second New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems, 1995 , Page(s): 342-345.

[216] ETAS, "ASCET-DIFF – Modell Difference Explorer", 2010, www.etas.com.

[217] IKV++ Technologies AG, "Medini Unite v1.2 - the tool for differences analysis and for consistent automated merge of Simulink and Stateflow models", 2010, www.ikv.de.

[218] ENSOFT, "SimDiff – a fast and accurate diff and audit tool for Simulink Models", 2010, www.ensoftcorp.com/SimDiff.

[219] I. Saha, K. Chakraborty, S. Roy, B. Reddy, V. Kurapati, V. Sharma, "An approach to reverse engineering of C programs to simulink models with conformance testing", ISEC '09: Proceeding of the 2nd annual conference on India software engineering conference, February 2009.

[220] S. Lopez, G. Alfonzo, O. Perez, S. Gonzalez, R. Montes, "A Metamodel to Carry Out Reverse Engineering of C++ Code into UML Sequence Diagrams", Electronics, Robotics and Automotive Mechanics Conference, 2006, Volume: 2, Page(s): 331–336.

[221] Lauterbach Development Tools, "JTAG Debuggers and In-Circuit Emulators", 2010, www.lauterbach.com.

[222] JTAG – former Joint Test Action Group, "IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture" grouper.ieee.org/groups/1149/1/.

[223] ETAS, "EHOOKS - Efficient Software Hooks", 2010, www.etas.com.

[224] ETAS, "ES Series", 2010, www.etas.com.

[225] Mathworks, "Real-Time Workshop 7 – User's Guide", Matlab Simulink, 2010, www.mathworks.com.

[226] M. Conrad, "Systematic Testing of Embedded Automotive Software: The Classification-Tree Method for Embedded Systems (CTM/ES)", DaimlerChrysler AG, Research and Technology, Berlin, Germany, 2005.

[227] Grimm, K., "Software technology in an automotive company - major challenges", Proceedings of the 25th International Conference on Software Engineering, 2003, Page(s): 498-503.

[228] MISRA - Motor Industry Software Reliability Association, 2010, www.misra.org.uk.

[229] D. Spinellis, "World's smallest BASIC interpreter", The International Obfuscated C Code Contest, IOCCC 1990 winner, www0.us.ioccc.org.

[230] Wikipedia, "Endiannes", 2010, en.wikipedia.org/wiki/Endianness.

[231] Wikipedia, "Wait States", 2010, en.wikipedia.org/wiki/Wait_state.

[232] J. Smith, G. Sohi, "The microarchitecture of superscalar processors", Proceedings of the IEEE, Volume: 83 , Issue: 12, 1995 , Page(s): 1609–1624.

[233] Wikipedia, "Out-of-order Execution", 2010, en.wikipedia.org/wiki/Out-of-order_execution.

[234] Wikipedia, "Boundary Scan", 2010, en.wikipedia.org/wiki/Boundary_scan.

[235] Wikipedia, "Code Coverage", 2010, en.wikipedia.org/wiki/Code_coverage.

[236] TIOBE Software, "TIOBE Programming Community Index", 2010, www.tiobe.com.

[237] A. Koenig, "C Traps and Pitfalls", AT&T Bell Laboratories, Addison-Wesley, 1987.

[238] A. Camesi, J. Hulaas, W. Binder, "Continuous Bytecode Instruction Counting for CPU Consumption Estimation", Third International Conference on Quantitative Evaluation of Systems, 2006, Page(s): 19-30.

[239] Dr. Garbage, "JAVA Bytecode & Sourcecode Visualizers", 2010, www.drgarbage.com.

[240] AIVOSTO - Programming Tools for Software Developers, "Visustin v6 Flow Chart Generator", 2010, www.aivosto.com.

[241] FATESOFT – The Intelligent Solutions Company, "Code Visual to Flowchart - automatic code flow chart generator", 2010, www.fatesoft.com.

[242] P. De, V. Mann, U. Mittaly, "Handling OS jitter on multicore multithreaded systems", IEEE International Symposium on Parallel & Distributed Processing, IPDPS 2009, Page(s): 1-12.

[243] V. Jaikamal, "Model Based ECU Development – An integrated MiL-SiL-HiL Approach", ETAS Inc. Marketing, October 2008.

[244] F. Brooks Jt., "No Silver Bullet - Essence and Accidents of Software Engineering", Computer, Volume: 20 , Issue: 4, April 1987 , Page(s): 10-19.

[245] N. Wirth, "A plea for lean software", Computer, Volume: 28 , Issue: 2, 1995 , Page(s): 64-68.

[246] Wikipedia, "The Second-System Effect", 2010, en.wikipedia.org/wiki/Second-system_effect.

[247] Wikipedia, "Pareto Principle", 2010, en.wikipedia.org/wiki/Pareto_principle.

[248] Wikipedia, "Parkinson's Law", 2010, en.wikipedia.org/wiki/Parkinson's_law.

[249] Wikipedia, "Jevons' Paradox", 2010, en.wikipedia.org/wiki/Jevons_paradox.

[250] Wikipedia, "Brooks' Law", 2010, en.wikipedia.org/wiki/Brooks'_law.

[251] Wikipedia, "Rube Goldberg Machines", 2010, en.wikipedia.org/wiki/Rube_Goldberg.

[252] V. Bala, E. Duesterwald, S. Banerjia, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo", Hewlett-Packard Laboratories Cambridge, June 1999.

[253] V. Prokhorov, V. Kosarev, "Environment pi J for visual programming in Java", Proceedings of the IEEE International Conference on Information Visualization, 1999, Page(s): 535-542.

[254] Borland International Inc., "Borland C++ for Windows", www.borland.com.

[255] P. Buhr, W. Mok, "Advanced exception handling mechanisms", IEEE Transactions on Software Engineering, Volume: 26 , Issue: 9, 2000 , Page(s): 820-836.

[256] M. Kuba, C. Polychronopoulos, K. Gallivan, "The Synergetic Effect of Compiler, Architecture, and Manual Optimizations on the Performance of CFD on Multiprocessors", Proceedings of the IEEE/ACM SC95 Conference on Supercomputing, 1995, Page(s): 72–72.

[257] Wikipedia, "Memory Protection", 2010, en.wikipedia.org/wiki/Memory_protection.

[258] Altium (former Protel), "Altium Designer", 2010, www.altium.com / www.protel.com.

[259] Computer Hope, "Command line vs. GUI", Issue CH000619, 2010, www.computerhope.com.

[260] R. Glass, "Two Mistakes and Error-Free Software: A Confession", Software, IEEE, Volume: 25, Issue: 4, 2008, Page(s): 96-96.

[261] M. Zhivich, R. Cunningham, "The Real Cost of Software Errors", Security & Privacy, IEEE, Volume: 7, Issue: 2, 2009, Page(s): 87-90.

[262] S. Sunter, "Correct by construction is guaranteed to fail", Test Conference, Proceedings, ITC 2005, IEEE International, Page(s): 1 pp.-1308.

[263] J. Regehr, A. Reid, K. Webb, "Eliminating stack overflow by abstract interpretation", Transactions on Embedded Computing Systems (TECS), November 2005, Volume 4, Issue 4, ACM

[264] C. Robbins, "Autocoding: an enabling technology for rapid prototyping", Conference Proceedings on Acoustics, Speech, and Signal Processing, ICASSP-96, 1996, IEEE International, Volume: 2, Page(s): 1260-1263.

[265] C. Bieser, K. Muller-Glaser, "COMPASS - a novel concept of a reconfigurable platform for automotive system development and test", The 16th IEEE International Workshop on Rapid System Prototyping (RSP 2005), 2005, Page(s): 135–140.

[266] R. Dorey, D. Maclay, "Rapid prototyping for the development of powertrain control systems", Proceedings of the 1996 IEEE International Symposium on Computer-Aided Control System Design, 1996, Page(s): 135–140.

[267] S. Toeppe, D. Bostic, S. Ranville, K. Rzemien, "Automatic code generation requirements for production automotive powertrain applications", Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design, 1999, Page(s): 200–206.

[268] C. Dase, J. Falcon, B. MacCleery, "Motorcycle control prototyping using an FPGA-based embedded control system", IEEE Control Systems Magazine, Volume: 26, Issue: 5, 2006, Page(s): 17-21.

[269] S. Cardelli, M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, "Rapid-prototyping of embedded systems via reprogrammable devices", Proceedings of the Seventh IEEE International Workshop on Rapid System Prototyping, 1996, Page(s): 133–138.

[270] National Instruments, "LabVIEW Simulation Module User Manual", April 2004 Edition, www.ni.com.

[271] D. Bolanakis, K. Kotsis, T. Laopoulos, "Arithmetic operations in assembly language: Educators' perspective on endianness learning using 8-bit microcontrollers", IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2009, Page(s): 600-604.

[272] M. Hernandez, "A Generalized, Mathematical Approach For Exploiting Stack Overflow Vulnerabilities on 2n-Bit Architectures", IEEE Global Telecommunications Conference, GLOBECOM 2008, Page(s): 1–4.

[273] J. Hennessy, T. Gross, "Code generation and reorganization in the presence of pipeline constraints", POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, January 1982, ACM.

[274] S. Mahlke, B. Natarajan, "Compiler synthesized dynamic branch prediction", Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-29, 1996, Page(s): 153-164.

[275] T. Li, L. John, A. Sivasubramaniam, N. Vijaykrishnan, J. Rubio, "OS-Aware Branch Prediction: Improving Microprocessor Control Flow Prediction for Operating Systems", IEEE Transactions on Computers, Volume: 56, Issue: 1, 2007, Page(s): 2–17.

[276] Chang-Chung Liu; R-Ming Shiu; Chung-Ping Chung, "Register renaming for x86 superscalar design", Proceedings of the International Conference on Parallel and Distributed Systems, 1996, Page(s): 336–343.

[277] Xueyu Geng, Shunyao Wu, Jinlong Wang, Peng Li, "An Effective Remote Control System Based on TCP/IP", International Conference on Advanced Computer Theory and Engineering, ICACTE '08, 2008, Page(s): 425–429.

[278] M. Broy, "Challenges in automotive software engineering", Proceedings of the 28th international conference on Software engineering, 2006, Pages: 33–42.

[279] Pradnya Choudhari, "Java Advantages & Disadvantages", Web Angel Volunteer, 2010, ArizonaCommunity.com.

[280] Wikipedia, "FORTRAN", 2010, en.wikipedia.org/wiki/Fortran.

[281] J. Backus, "The History of FORTRAN I, II and III", IEEE Annals of the History of Computing, Volume: 1, Issue: 1, 1979, Page(s): 21–37.

[282] Wikipedia, "ADA", 2010, en.wikipedia.org/wiki/Ada_(programming_language).

[283] M.H. Smith, M. Elbs, "Towards a more efficient approach to automotive embedded control system development", Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on 22-27 Aug. 1999 Page(s):219 - 224.

[284] I. Bell, "Software - More than a programming language - Whether testing cars at automotive companies or controlling production and quality at manufacturing plants, engineers and scientists need flexible, cost-effective solutions for test, measurement and automation", Computing & Control Engineering Journal Volume 18, Issue 1, Feb.-March 2007 Page(s): 26 - 29.

[285] V. Bonhomme, C. Franssen, "Awake craniotomy", European Journal of Anaesthesiology: Nov. 2009, Vol. 26, Issue 11, 906-912.

[286] H. Prähofer, D. Hurnaus, C. Wirth, H. Mossenbock, "The Domain-Specific Language Monaco and its Visual Interactive Programming Environment", Visual Languages and Human-Centric Computing, VL/HCC. IEEE Symposium 23-27 September 2007 Page(s): 104-110.

[287] Microsoft Development Network, "Don Syme's WebLog on F# and Related Topics", 2010, msdn.microsoft.com.

[288] J. Miecznikowski, L. Hendren, "Decompiling Java using staged encapsulation", Reverse Engineering, 2001. Proceedings. Eighth Working Conference on 2-5 Oct. 2001 Page(s): 368 - 374.

[289] Wikipedia, "BASIC", en.wikipedia.org/wiki/BASIC.

[290] M. Jersak, K. Richter, "Mehr Kerne – aber sicher", Elektroniknet, Artikel 26243, 2010, www.elektroniknet.de.

[291] IAR Systems, "IAR Embedded Workbench for MSP430", www.iar.com.

[292] Microsoft Corporation, "Visual Studio 2005", 2010, www.microsoft.com.

[293] Microsoft Corporation, "Visual Studio 2008", 2010, www.microsoft.com.

[294] NDepend, "NDepend", 2012, www.ndepend.com.

[295] Patrick Smacchia, "Measuring an Interesting Fact", 13.04.2010, www.codebetter.com.

[296] K. Scheidemann, M. Knapp, C. Stellwag, "Load Balancing in AUTOSAR-Multicore-Systemen – Teil 1 & 2", Elektroniknet, Artikel 26546, 28.04.2010, www.elektroniknet.de.

[297] A. Turing, "Computing machinery and intelligence", 1950, Oxford Journal Mind, 59, 433-460.

[298] Wikipedia, "ASCET", 2010, de.wikipedia.org/wiki/ASCET.

[299] Wikipedia, "MATLAB", 2010, en.wikipedia.org/wiki/MATLAB.

[300] Accurate Technologies, "No-Hooks Software", 2010, www.accuratetechnologies.com.

[301] Accurate Technologies, "No-Hooks Rapid-Prototyping Software", 2010.

[302] Bosch Motorsport, "Equipment for High Performance Vehicles – Catalogue", Edition 2005-3.

[303] Bosch Motorsport, "MS3.x ECU family", www.bosch-motorsport.com.

[304] J. Gerhardt, H. Hönninger, H. Bischof, "BOSCH ME7 - A New Approach to Functional and Software Structure for Engine Management Systems", 1998, Society of Automotive Engineers, Robert Bosch GmbH.

[305] Bosch Motorsport, "Equipment for High Performance Vehicles – Catalogue", Edition 2010-2.

[306] Robert Bosch GmbH, "Die Powertrain-Steuerung für künftige Verbrauchs- und Emissionsvorschriften: das Motronic System", Gasoline Systems (DGS-EC/MKT), August 2009.

[307] S. Kane, E. Liberman, T. DiViesti, F. Click, "Toyota Sudden Unintended Acceleration", (footnote 143 on the Toyota Recall 09V388, TMNA November 25, 2009 Letter to NHTSA) Safety Research & Strategies Inc., www.safetyresearch.net.

[308] P. Kulzer, "MS4.0 40CS0X1A Clubsport Basis Software – Function Manual", Bosch Motorsport, June 2007.

[309] P. Kulzer, "MS4.0 40CS0X1A Clubsport Basis Software STATISTICS", 2010.

[310] Texas Instruments, "MSP430 – Ultra-Low-Power 16-bit RISC Microcontrollers", 2010, www.msp430.com, www.ti.com/msp430.

[311] Microchip, "PIC – 16-bit Microcontrollers", 2010, www.microchip.com.

[312] Xilinx Inc., "Xilinx ISE Software Wins Electronic Design Magazine's Best of 2007 EDA/FPGA Tool Award", Press Release, 16th January 2008, Art.-Nr. 1096962, press.xilinx.com.

[313] Siemens, "C167CR - 16-Bit CMOS Single-Chip Microcontroller", Data Sheet 06.95 Advance Information.

[314] Siemens, "SAB 80C166 / C167 / C165 / C163 Product Guide", August 1996.

[315] Vector Fabrics, "vfAnalyst & vfSoftware", 2010, www.vectorfabrics.com.

[316] Aonix, "AonixPERC UltraSMP Virtual Machine – Multi-Core Solutions for JAVA Developers", 2010, Product Line Overview, www.aonix.com.

[317] NORDIC Semiconductor, "nRF24L01 radio chips and modules operating in the 2.4GHz band", 2010, www.nordicsemi.com.

[318] FTDI – Future Technology Devices International Ltd., "FT2232C – USB 1.1 Dual USB UART/FIFO IC", 2010, www.ftdichip.com.

[319] GOOGLE, "GoogleCL - Command line tools for the Google Data APIs", 2010, code.google.com/p/googlecl.

[320] DIGILENT Inc., "FPGA development boards with the XILINX XC3S1600E FPGA chip", 2010, www.digilentinc.com.

[321] XILINX, "FPGA chip XC3S1600E", 2010, www.xilinx.com.

[322] dSpace, "dSPACE MicroAutoBox II: The Start of a New Generation", 2010, www.dspace.com.

[323] Newton Research Labs, "Interactive C User's Guide", Manual Edition 0.9 Documents software version 3.1 April 18, 1997, www.newtonlabs.com/ic.

[324] XILINX, "ISE Design Suite", 2010, www.xilinx.com.

[325] XILINX, "ChipScope Pro 12.1 Software and Cores – User Guide", UG029 (v12.1) April 19, 2010, www.xilinx.com.

[326] DIGILENT, "MicroBlaze Development Kit Spartan-3E 1600E Edition User Guide", UG257 (v1.1) December 5, 2007, www.digilentinc.com.

[327] XILINX, "Spartan-3E FPGA Family Datasheet", DS312 (v3.8) August 26, 2009.

[328] G. Candea, J. Cutler, A. Fox, "Improving Availability with Recursive Microreboots: A Soft-State System Case Study", Performance Evaluation Journal, vol. 56, nos. 1-3, March 2004.

[329] G. Candea, A. Fox, "Crash-Only Software", Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX), May 2003.

[330] A. Bobbio, M. Sereno, C. Anglano, "Fine grained software degradation models for optimal rejuvenation policies", Performance Evaluation, Volume 46, Issue 1, September 2001, Pg: 45-62.

[331] A. Avritzer, A. Bondi, M. Grottke, K. Trivedi, E. Weyuker, "Performance Assurance via Software Rejuvenation: Monitoring, Statistics and Algorithms", Proceedings of the International Conference on Dependable Systems and Networks 2006, pages 435-444.

[332] K. Trivedi, "Software Aging & Rejuvenation: Modeling and Analysys", University of Florida Seminar, Dept. of Electrical & Computer Engineering, March, 2003.

[333] D. Chamberlin, "The 'single-assignment' approach to parallel processing", AFIPS '71 (Fall): Proceedings of the November 16-18, 1971, fall joint computer conference.

[334] Wikipedia, "Watchdog timer", 2010, en.wikipedia.org/wiki/Watchdog_timer.

[335] Stanford University, University of California Berkeley, "The Berkeley/Stanford Recovery-Oriented Computing (ROC) Project", 2010, roc.cs.berkeley.edu.

[336] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft, "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies", University of California Berkeley, Computer Science Technical Report UCB//CSD-02-1175, March 15, 2002, roc.cs.berkeley.edu.

[337] Wikipedia, "Hygienic Macros", 2010, en.wikipedia.org/wiki/Hygienic_macro.

[338] R. Dybvig, "Writing Hygienic Macros in Scheme with Syntax-Case", Language Study, Indiana University Computer Science Department, Bloomington, August 1992.

[339] Wikipedia, "Immunity Aware Programming", 2010, en.wikipedia.org/wiki/Immunity_Aware_Programming.

[340] Wikipedia, "Parallel Random Access Machine", 2010, en.wikipedia.org/wiki/Parallel_Random_Access_Machine

[341] Institut für Rechnerarchitektur und Parallelrechner, "SB-PRAM - CRCW-PRAM-Modell Shared Memory MIMD Parallelrechner", www-wjp.cs.uni-saarland.de/forschung/projekte/SB-PRAM.

[342] Wikipedia, "Virtual Memory", 2010, en.wikipedia.org/wiki/Virtual_memory.

[343] P. Bach, M. Bosch, J. Fischer, C. Lichtenau, W. Paul, J. Röhrig, "Real PRAM Programming", Europar 2002, Paderborn, Germany, volume 2400 of LNCS, Springer, 2002.

[344] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grün, C. Lichtenau, "Building the 4 Processor SB-PRAM Prototype", Proceedings of the Hawaii 30th Int'l Symposium on System Science HICSS-30, pages 14-23, 1997.

[345] Coderoom, "Criminal Overengineering", March 2010, coderoom.wordpress.com.

[346] Embarcadero, "Embarcadero Delphi", 2010, www.embarcadero.com/products/delphi.

[347] Haskellwiki, "Haskell", 2010, www.haskell.org.

[348] Wikipedia, "Haskell", 2010, en.wikipedia.org/wiki/Haskell_(programming_language)

[349] "The world ZX Spectrum", www.worldofspectrum.org.

[350] Wikipedia, "Sinclair BASIC", 2010, en.wikipedia.org/wiki/Sinclair_BASIC.

[351] D. Brin, "Why Johnny can't code", Education Topic, SALON – a site by National Geographic Channel, September 2006, www.salon.com.

[352] F. Ruehr, "The Evolution of a Haskell Programmer", Willamette University, August 2001, www.willamette.edu/~fruehr/haskell/evolution.html.

[353] H. Uwano, M. Nakamura, A. Monden, K. Matsumoto, "Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement", Nara Institute of Science and Technology, Proceedings of the 2006 symposium on Eye tracking research & applications, San Diego, California, Pages: 133 – 140.

[354] B. Heeren, D. Leijen, A. IJzendoorn, "Helium, for Learning Haskell", Proceedings of the 2003 ACM SIGPLAN workshop on Haskell, Uppsala, Sweden, Pages: 62 – 71.

[355] J. Kuśmierek, V. Bono, "Hygienic methods — Introducing HygJava", Journal of Object Technology, Vol. 6, No. 9, Special Issue: TOOLS EUROPE 2007, October 2007.

[356] Wikipedia, "Atari BASIC", 2010, en.wikipedia.org/wiki/Atari_BASIC.

[357] Wikipedia, "Commodore BASIC", 2010, en.wikipedia.org/wiki/Commodore_BASIC.

[358] C. Ek, "Detecting Endian Issues with Static Analysis Tools - Best practices for using static analysis tools and enforcing correct programming in C/C++ ", Dr. Dobb's Journal, July 2010.

[359] AbsInt - Angewandte Informatik, "AbsInt Advanced Analyser", 2010, www.absint.com.

[360] C. Ferdinand, R. Heckmann, H. J. Wolff, C. Renz, O. Parshin, R. Wilhelm, "Towards Model-Driven Development of Hard Real-Time Systems - Integrating ASCET and aiT/StackAnalyzer", Model-Driven Development of Reliable Automotive Services: Second Automotive Software Workshop, ASWSD 2006, San Diego, CA, USA, March 15-17, 2006, Pages: 145-160.

[361] C. Ferdinand, R. Heckmann, H. J. Wolff, C. Renz, O. Parshin, R. Wilhelm, "Towards Model-Driven Development of Hard Real-Time Systems - Integrating ASCET and aiT/StackAnalyzer", Presentation of the Universität des Saarlandes, AbsInt – Angewandte Informatik, 2006.

[362] D. Jones, "Developer beliefs about binary operator precedence", Knowledge Software, Ltd., Farnborough, Hants, England, August 2006, www.knosof.co.uk.

[363] Wikipedia, "Order of operations", 2010, en.wikipedia.org/wiki/Order_of_operations.

[364] M. Barenhoff, "Wie man die »Worst Execution Time« bestimmt", ElektronikNet: Entwicklungs-Tools für Hard- und Software, Artikel 1657, Februar 2010, www.elektroniknet.de.

[365] C. Hernitscheck, P. Forstner, "MSP430 Seminar", Module 1, FAE, June 2005.

[366] C. Chiang, "SUSI - Secure & Unified Smart Interface", Technical White Paper v1.1, October 2006, Advantech – Trusted ePlatform Services, www.advantech.com.

[367] M. Jersak, K. Richter, "Zeit für AUTOSAR - Tool-Kette für den richtigen Umgang mit Timing-Problemen", ElektronikNet, Artikel 28713, August 2010, www.elektroniknet.de.

[368] National Instruments, "NI LabVIEW Compiler: Under the Hood", Tutorial ID 11472, NI Dev. Zone, July 2010, zone.ni.com.

[369] Wikipedia, "Compiler optimization", 2010, en.wikipedia.org/wiki/Optimizing_compiler..

[370] L. Raccoon, "The complexity gap", ACM SIGSOFT Software Engineering Notes archive, Volume 20, Issue 3, July 1995, Pages: 37 – 44.

[371] TTE© Systems Ltd., "The RapidiTTy family", 2010, www.tte-systems.com.

[372] M. Pont, "Reducing the time taken to test your next embedded system", 2010, www.tte-systems.com.

[373] T. Morris, "Scribble - programming language wrapper", 2010, github.com/tommorris/scribble.

[374] J. Coombs, "C6Flo DSP Software Development Tool", White Paper, July 2010, Texas Instruments Incorporated.

[375] BonitaSoft, "Bonita Studio", 2010, www.bonitasoft.com.

[376] J Regehr, N. Jones, "C and C++ Make It Hard to Read a Register for Its Side Effects", Embedded in Academia, March 2010, blog.regehr.org/archives/41.

[377] A. Blackwell, "Dealing with New Cognitive Dimensions", University of Cambridge, December 2000, Discussion Paper prepared for Workshop on Cognitive Dimensions, University of Hertfordshire.

[378] S. Dmitriev, "Language Oriented Programming: The Next Programming Paradigm", November 2004, JetBrains, www.onboard.jetbrains.com.

[379] J. Ganssle, "Subtract software costs by adding CPUs", April 2005, EE-Times Design, www.eetimes.com.

[380] W. Johnston, J. Hanna, R. Millar, "Advances in Dataflow Programming Languages", ACM Computing Surveys, Vol. 36, No. 1, March 2004, pp. 1–34.

[381] MOTEC, "MoTeC – Engine Management & Data Acquisition Systems", 2010, Victoria - Australia, www.motec.com.au.

[382] Microsoft Corporation, "Small BASIC", 2010, smallbasic.com.

[383] Microsoft, "Microsoft Small BASIC – An introduction to Programming", 2010.

[384] "JavaScript Console", 2010, jsconsole.com.

[385] National Instruments, "Multicore Programming with NI LabVIEW", 2010, www.ni.com.

[386] National Instruments, "Multiple Programming Approaches in NI LabVIEW", 2010, www.ni.com.

[387] National Instruments, "The Benefits of Programming Graphically in LabVIEW", 2010, www.ni.com.

[388] L. Hardesty, "Multicore may not be so scary", MultiCore 0930, MITNews, 2010, web.mit.edu.

[389] PARASOFT Embedded, 2010, www.parasoft-embedded.com.

[390] D. Rushkoff, "Program or be Programmed: Ten Commands for a Digital Age", 2010, OR Books.

[391] Woodward, "GAP^{TM} Graphical Application Programmer - v1.x & v2.x)", 2010, www.woodward.com.

[392] Woodward, "Monitor GAP^{TM}", 2010, www.woodward.com.

[393] D. Norman, "Simplicity Is Highly Overrated", 2010, www.jnd.org.

[394] Anthony, " Simplicity is Not Overrated, Just Misunderstood", June 2010, uxmovement.com.

[395] T. Green, "Usability Analysis of Visual Programming Environments - a 'cognitive dimensions' framework", Journal of Visual Languages and Computing, 7:131-174, 1996.

[396] A. Perlis, "Epigrams on Programming", ACM's SIGPLAN publication, September 1982, Yale University.]

[397] J. Spool, "What Makes a Design Seem 'Intuitive'?", User Interface Engineering, January 2005, www.uie.com.

[398] MathCore, "MathModelica – Introductory Examples", MathCore Engineering AB, fifth edition, 2009, www.mathcore.com.

[399] D. Lorenz, ”HIL basierte Kalibrierung anhand des HAWKS Rennwagens”, Studienarbeit, Faculty of Engineering and Computer Science, Department Informatik Department of Computer Science, 2009.

[400] Spectrum Software, "Micro-Cap 10 – Analog/Digital Simulator", 2010, www.spectrum-soft.com.

[401] ETAS, "EHOOKS V1 - Flyer", September 2010, www.etas.com.

[402] ETAS, "EHOOKS Prototyping is Rapid Again", September 2010, www.etas.com.

[403] TÜV - Technischer Überwachungs-Verein, "Die Anzahl der erheblichen Mängel bei Autos nimmt zu",Report 2011.

[404] Berkeley University of California, "BYOB 3.0 — Build Your Own Blocks", 2011, byob.berkeley.edu.

[405] Berkeley University of California, "BYOB Reference Manual – V3.0", 2011, byob.berkeley.edu.

[406] S. Lichtenberg, E. Zhang, M. Continisio, A. Rumsey, R. Griffith, "End-User Programming for Home Automation", 2011, Rutgers University - School of Arts and Science, scratchabledevices.com.

[407] Rutgers University, "Scratchable Devices", 2011, scratchabledevices.com.

[408] AFT Atlas Fahrzeugtechnik GmbH, "PROtroniC^{TM} – Open, Modular Development Platform", brochure 2009-2, www.aft-werdohl.de.

[409] M. Gebhardt, U. Lauff, K. Schnellbacher, "Operation am offenen Herzen Teil 1: Entwicklung und Test von Steuergerätefunktionen mit der Bypass-Methode", Elektronik Automotive, Entwicklung+ Test+Software Verifikation, 6.2008, WEKA FACHMEDIEN, www.elektroniknet.de.

[410] W. Dubitzky, W. Eismann, J. Schinagl, "Operation am offenen Herzen Teil 2: Einsatzmöglichkeiten der Bypass-Methode für Entwicklung & Test von Steuergerätefunktionen", Elektronik Automotive, Entwicklung+Test+Software Verifikation, 2009, WEKA FACHMEDIEN, www.elektroniknet.de.

[411] Accurate Technologies, "No-Hooks Frequently Asked Questions", 2011.

[412] Wikipedia, "Amdahl's Law", 2011, en.wikipedia.org/wiki/Amdahl%27s_law.

[413] Wikipedia, "Transputer", 2011, en.wikipedia.org/wiki/Transputer.

[414] INMOS, "Transputer Databook", third edition, 1992, www.transputer.net.

[415] J. Braunes, "Multi-Core-MCUs effizient debuggen", May 2012, www.elektroniknet.de.

[416] HiTEX, "Solutions & Tool Chain – The OCDS debugger: TantinoXC", May 2005 www.hitex.com.

[417] dSpace, "NEW: ECU Interface Manager – Service-based bypassing without ECU source-code modifications", 2012, www.dspace.com.

[418] The Internet Engineering Task Force (IETF), "Internet Protocol, Version 6 (IPv6) Specification", Memo RFC2460, Dec 1998, www.ietf.org.

[419] MOTEC, "M1 Series ECUs: New generation engine management system", 2012, www.motec.com.

[420] Rapita Systems, "RapiTime Exlained – White Paper", 2012, www.rapitasystems.com.

[421] QNX, Operating systems and development tools for embedded systems, 2012, www.qnx.com.

[422] Renesas, Microcontrollers, www.renesas.eu.

[423] Intel, ATOM Processor, www.intel.com/content/www/us/en/processors/atom/atom-processor.html.

[424] ETAS, "Neues ERCOS^{EK} spart noch mehr Zeit", 2012, Auto & Elektronik 1, 1999.

[425] H. Richards, C. Wright, "Introduction to SYMBOL 2R Programming Language", HLLCA '73 Proceedings of a symposium on High-level-language computer architecture, 1973, pages 27-33.

[426] J. Anderberg, C. Smith, "High-Level Language Translation in SYMBOL 2R", HLLCA '73: Proceedings of the ACM-IEEE symposium on High-level-language computer architecture, 1973, pages 11-19.

[427] H. Ahmed, "UML Based Automated Code Generation", Motorola Inc., presentation at the IEEE Computer Society Phoenix, Nov 2007.

[428] C. Holt, "VIZ: a visual language based on functions", Procedings of the 1990 IEEE Workshop on Visual Languages, Oct 1990, Pages 221-226.

[429] E. Ghittori, M. Mosconi, M. Porta, "Designing new Programming Constructs in a Data Flow VL", Proceedings of the 1998 IEEE Symposium on Visual Languages, 1998, Pages: 78-79.

[430] Intel, "Intel Multi-Core Processors", 2012, www.intel.com.

[431] Parallax, "Propeller P8X32A Multiprocessor Datasheet", Jun 2011, Rev 1.4, www.parallax.com.

[432] R. Dettmer, "The VIPER microprocessor", Electronics and Power, Vol.32, Issue 10, Oct 1986, Pages: 723-727.

[433] T. Buckley, P. Jesty, "Programming a VIPER", Proceedings of the Fourth Annual Conference on Computer Assurance, Systems Integrity, Software Safety & Process Security, 1989, Pages: 84-92.

[434] B. Magnusson, S. Minör, "III - an integrated interactive incremental programming environment based on compilation", Proceedings of the 1985 ACM SIGSMALL symposium on Small systems, Pages 235-244.

[435] JetBrains, "ReSharper 6 Reviewer's Guide", 2012, www.jetbrains.com/resharper.

[436] DevExpress, "CodeRush", 2012, devexpress.com.

[437] Telerik, "JustCode", 2012, www.telerik.com.

[438] WholeTomato, "Visual Assist X", 2012, www.wholetomato.com.

[439] C. Lu, J.C. Fabre, M.O. Killijian, "Robustness of modular multi-layered software in the automotive domain: a wrapping-based approach", IEEE Conference on Emerging Technologies & Factory Automation, 2009, pp 1-8.

[440] N. Kikkeri, P. Seidel, S. Beyer, "Challenges in the formal verification of complete state-of-the-art processors", Proc. of the IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors, October 2005, Pg 603-606.

[441] M. Behm, J. Ludden, Y. Lichtenstein, M. Rimon, M. Vinov, "Industrial experience with test generation languages for processor verification", Proceedings of the 41st annual Design Automation Conference, 2004, Pages 36-40.

[442] A. Adir, G. Shurek, "Generating concurrent test-programs with collisions for multi-processor verification", Seventh IEEE International Conference on High-Level Design Validation and Test, Oct 2002, Pages 77-82.

[443] S. Sudhakrishnan, L. Su, J. Renau, "Processor Verification with hwBugHunt", 9th International Symposium on Quality Electronic Design, Mar 2008, Pages 224-229.

[444] The OSEK Group, "OSEK/VDX Operating System", Version 2.2.3, Feb 2005, www.osek-vdx.org

[445] AUTOSAR Consortium, "AUTOSAR Specification of Operating System", V3.0.2 R3.0 Rv 0003, 2012, www.autosar.org.

[446] T.W. Kuo, "Real-Time Operating Systems & Resource Management", Dept. of Computer Science & Information Engineering, National Taiwan University, 2002.

[447] I. Lee, "OS Overview – Real-Time Scheduling", Dept. of Computer and Information Science, University of Pennsylvania, Fall 2006.

[448] Wikipedia, "Smart Pointer", 2012, en.wikipedia.org/wiki/Smart_pointer.

[449] V. Hazrati, "Is Five the Optimal Team Size?", 2009, Section of Processes & Practices, www.infoq.com/news.

[450] G. Fox, "High Performance Distributed Computing", 2000, Northeast Parallel Architectures Center, Syracuse University, New-York.

[451] nCUBE, "nCUBE", 2012, Wikipedia, en.wikipedia.org/wiki/NCUBE.

[452] National Instruments, "Multicore Programming with LabVIEW", 2012, www.ni.com/multicore.

[453] National Instruments, "Why Dataflow Programming Languages are Ideal for Programming Parallel Hardware", 2012, www.ni.com/multicore.

[454] National Instruments, "Programming Strategies for Multicore Processing: Task Parallelism", 2012, www.ni.com/multicore.

[455] National Instruments, "Programming Strategies for Multicore Processing: Data Parallelism", 2012, www.ni.com/multicore.

[456] S. Edwards, "The C Language", 2001, W4995-02 Class Presentation, University of Columbia.

[457] B. Kernighan, D. Ritchie, "The C Programming Language", 2nd Edition, Prentice Hall Software Series.

[458] D. May, "Multicore Architecture", IET, London, Oct 2008, XMOS XS1-G4 System.

[459] L. Peter, R. Hull, "The Peter Principle: Why Things Always Go Wrong", New York: William Morrow and Company, 1969.

[460] Simplify, "Ending the curse of software maintenance", Gmodel team, May 2011, gmodel.org.

[461] GreenArrays Inc., "Green Arrays GA144 144-Computer Chip", 2012, www.greenarraychips.com.

[462] GreenArrays Inc., "Green Arrays GA144 Poster", 2012, www.greenarraychips.com.

[463] GreenArrays Inc., "Green Arrays F18A 18-bit Computer", 2012, www.greenarraychips.com.

[464] Freescale Semiconductor, "MC33810 – Automotive Engine Control IC", Rev 4.0, 2/2008.

[465] Bosch Semiconductors, "CJ125 – Product Information – Lambda Probe Interface IC", 04/2006.

[466] Bosch Semiconductors, "CJ136 – Product Information – Lambda Probe Interface IC", 12/2011.

[467] Bosch Semiconductors, "CC195 – Product Information – Knocking Signal Evaluation IC", 02/2006.

[468] R. Ferrara, T. Bosvieux, T. Peterson, Freescale Semiconductor, "How to Use and Program the New MC33816 High-Performance Fuel Injector Driver Circuit", FTF-AUT-F0098, Freescale Technology Forum – Powering Innovation, June 2012.

[469] S. Mahapatra, R. Mahapatra, "Mapping Of Backpropagation Learning Onto Distributed Memory Multiprocessors", ICAPP 95 - IEEE First ICA/sup 3/PP., IEEE First International Conference on Algorithms and Architectures for Parallel Processing, 1995, Pages(s) 217-226.

[470] M. Homewood, D. May, D. Shepherd, R. Shepherd, "The IMS T800 Transputer", IEEE Micro, 1987, Pages 10-26.

[471] Wikipedia, "Systolic Array", 2012, en.wikipedia.org/wiki/Systolic_array.

[472] K. Zhang, G. Marwaha, "Visputer - A Graphical Visualization Tool for Parallel Programming", The Computer Journal, Vol. 38, No. 8, 1995.

[473] C. Jacobsen, M. Jadud "The Transterpreter - A Transputer Interpreter", Comm. Process Archit., IOS Press, 2004.

[474] Philips, "Self-sharpening hair-cutters and shavers", 2012, www.philips.com.

[475] OKI, "Microwave soldering and unsoldering stations", 2012, www.oki.com.

[476] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, "FastFlow: high-level and efficient streaming on multi-core (a FastFlow short tutorial)", July 2011.

[477] Bosch Motorsport, "Function Sheet MS4-Sport-Turbo [40CSTX31]", February 2011.

[478] S. Poledna, T. Mocken, J. Schiemann, T. Beck, Robert Bosch GmbH, ETAS, "ERCOS: An Operating System for Automotive Applications", SAE International Technical Paper Series 960623, February 1996.

[479] Robert Bosch, "Audi R4-5V T Quereinbau 132kW ME7.1", January 2001.

[480] Wikipedia, "Anders Hejlsberg", 2012, en.wikipedia.org/wiki/Anders_Hejlsberg.

[481] Wikipedia, "Windows 3.1x", 2012, en.wikipedia.org/wiki/Windows_3.1x.

[482] Wikipedia, "Windows 95", 2012, en.wikipedia.org/wiki/Windows_95.

[483] Wikipedia, "Homoiconicity", 2012, en.wikipedia.org/wiki/Homoiconicity.

[484] Bosch Motorsport, "RaceCon", 2012, www.bosch-motorsport.com

[485] Wikipedia, "Harvard Architecture", 2012, en.wikipedia.org/wiki/Harvard_architecture.

[486] B. Boothe, "A fully capable bidirectional debugger", ACM SIGSOFT Software Engineering Notes, Vol. 25, Issue 1, Jan 2000, Pages 36-37.

[487] E. Vigdorchik, A. Nikitin, "Extending managed debugger with backstepping facilities", St. Petersburg State University, IT Institute.

[488] Dr. Dobbs, "Deterministic Multi-Processor Replay of Concurrent Programs", 2010, www.drdobbs.com.

[489] T. Akgul, V. Mooney III, "Assembly instruction level reverse execution for debugging", ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 13, Issue 2, April 2004, Pages 149 – 198.

[490] ETAS, "EHOOKS V2.0 – Efficient Software Hook Insertion", Jun 2011, www.etas.com.

[491] R. Schafer, "On the Limits of Visual Programming Languages", ACM SIGSOFT Software Engineering Notes, March 2011 Volume 36 Number 2, Pages 7-8.

[492] AlleyCode, "AlleyCode HTML Editor", 2012, www.alleycode.com.

[493] K. Brooks, "LILAC: A Two-View Document Editor", Digital Equipment Corporation, Computer Magazine, Jun 1991, Vol. 24, Issue 6, Pages: 7-19.

[494] Sourceforge, "TeXlipse", Nov 2011, v1.5.0, texlipse.sourceforge.net.

[495] J. Miecznikowski, "Decompiling Java using staged encapsulation", Proceeding on the Eighth Working Conference on Reverse Engineering, 2001, Page(s): 368 – 374.

[496] E. Berkovich, B. Jacob, J. Nuzmann, U. Vishkin, "Looking to Parallel Algorithms for ILP and Decentralization", University of Maryland, Institute for Advanced Computer Studies, CS-TR-3921.

[497] U. Vishkin, "Using Simple Abstraction to Reinvent Computing for Parallelism", Communications of the ACM, Jan 2011, Vol. 54, No. 1, Pages: 75-85.

[498] DiniGroup, "DN7020K10 - Altera Stratix IV ASIC Prototyping Engine with 130 million ASIC gates", 2013, www.dinigroup.com.

[499] Freescale, "MC33816 - Automotive Engine Control IC with Smart Gate Control", Advance Information, 2013, www.freescale.com.

[500] B. Victor, "WorryDream", 2013, www.worrydream.com.

[501] L. Tesler, "No Modes", 2013, www.nomodes.com.

[502] L. Tesler, H. Enea, "A language design for concurrent prcoesses", Spring Joint Computer Conf., 1968, p.403-408.

[503] Wikipedia, "Usability", 2012, en.wikipedia.org/wiki/Software_usability.

[504] Wikipedia, "Fitt's Law", 2012, en.wikipedia.org/wiki/Fitts's_law.

[505] Wikipedia, "Hick's Law", 2012, en.wikipedia.org/wiki/Hick's_law.

[506] Wikipedia, "KISS Principle", 2012, en.wikipedia.org/wiki/KISS_principle.

[507] E. Frøkjær, M. Hertzum, K. Hornbæk, "Measuring Usability: Are Effectiveness, Efficiency, and Satisfaction Really Correlated?", Conference on Human Interfaces, April 2000, Pages 1-6.

[508] N. Bevan, "Measuring usability as quality of use", Software Quality Journal, Vol. 4., 1995, Pages 115-150.

[509] National Instruments, "LabVIEW Function and VI Reference Manual", May 1997, 321526A-01.

[510] Runtime Revolution, "LiveCode", 2013, www.runrev.com.

[511] A. Melbourne, J. Pugmire, "A small computer for the direct processing of FORTRAN statements", The Computer Journal, Vol. 56 Issue 2, Feb 2013.

[512] M. Ward, "Language Oriented Programming", Software - Concepts and Tools, Vol. 15, Pages 147-161, Oct 1994.

[513] J. Edwards, "Subtext: Uncovering the Simplicity of Programming", www.subtextual.org.

[514] A. Croll, "No IFs… alternatives to statement branching in JavaScript", 2013, javascriptweblog.wordpress.com.

[515] Wikipedia, "LISP", 2013, en.wikipedia.org/wiki/Lisp_(programming_language).

[516] F. Winters, C. Mielenz, G. Helestrand, "Design Process Changes Enabling Rapid Development", Convergence International Congress & Exposition On Transportation Electronics, SAE International, October 2004.

[517] Bosch Motorsport, "Modas Software", 2013, www.bosch-motorsport.de.

[518] N. Liu, J. Hosking, J. Grundy, "Integrating a Zoomable User Interfaces Concept into a Visual Language Meta-tool Environment", Proceeding VLHCC '04 Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing, Pages 38-40.

[519] Y. Sui, L. Pang, J. Lin, X. Zhang, "Dataflow Visual Programming Language Debugger Supported by Fisheye View", 9th International Conference for Young Computer Scientists, 2008, Page(s): 1024 – 1029.

[520] R. DeLine, K. Rowan, "Code Canvas: Zooming towards Better Development Environments", Microsoft, May 2010.

[521] T. Rodriguez, K. Russel, "Client Compiler for the Java HotSpot™ Virtual Machine – Technology and Application", Sun Microsystems, Inc., JavaOne – Sun's 2002 Worldwide Java Developer Conference.

[522] T. Wenisch, A. Ailamaki, B. Falsafi, A. Moshovos, "Mechanisms for Store-wait-free Multiprocessors", ISCA'07, June 9–13, 2007, San Diego, California, USA.

[523] M. Jurczyk, "Interconnection Networks for Parallel Computers", Wiley Encyclopedia of Computer Science and Engineering, 2008, John Wiley & Sons, Inc.

[524] P. Newman, "Chapter 4 - Multi-Stage Interconnection Networks", PhD thesis entitled "Fast Packet Switching for Integrated Services", Computer Laboratory - University of Cambridge, March 1989.

[525] A. Chin, "Locality-Preserving Hash Functions for General Purpose Parallel Computation", Algorithmica Vol. 12, Page(s) 170-181, 1994, Springer-Verlag New York Inc.

[526] H. Sutter, "Use Threads Correctly = Isolation + Asynchronous Messages", Dr. Dobb's, March 2009.

[527] Bosch Motorsport, "MS6.x ECU family", www.bosch-motorsport.com.

[528] J. Maeda, "The laws of Simplicity", 2013, www.lawsofsimplicity.com.

[529] P. Martins, "Communications and mass storage in scalable embedded systems", Main Document, Departamento de Electrónica e Telecomunicações, Universidade de Aveiro, 2010.

[530] P. Martins, "Communications and mass storage in scalable embedded systems", Annex, DET, Uni. Aveiro, 2010.

[531] P. Martins, "Communications and mass storage in scalable embedded systems", PowerPoint Presentation, Departamento de Electrónica e Telecomunicações, Universidade de Aveiro, 2010.

[532] R. Gomes, "Intelligent Peripherals with Digital Communications Bus", Main Document, Departamento de Electrónica e Telecomunicações, Universidade de Aveiro, 2010.

[533] R. Gomes, "Intelligent Peripherals with Digital Communications Bus", Annex, DET, Universidade de Aveiro, 2010.

[534] R. Gomes, "Intelligent Peripherals with Digital Communications Bus", Presentation, DET, Uni. Aveiro, 2010.

[535] F. Teixeira, "Parallel Processing System applied to the Automotive Segment", Main Document, Departamento de Electrónica e Telecomunicações, Universidade de Aveiro, 2010.

[536] F. Teixeira, "Parallel Processing System applied to the Automotive Segment", Presentation, Departamento de Electrónica e Telecomunicações, Universidade de Aveiro, 2010.

[537] N. Bernardino, " Development and Control System for Automotive Sports ECUs", Main Document, Departamento de Electrónica e Telecomunicações, Universidade de Aveiro, 2010.

[538] N. Bernardino, " Development and Control System for Automotive Sports ECUs", Presentation, Departamento de Electrónica e Telecomunicações, Universidade de Aveiro, 2010.

[539] D. Skillicorn, D. Talia, "Models and Languages for Parallel Computation", ACM Comp. Surv., Vol 30, No.2, June 1998.

[540] Microchip, "PIC Low-Power Microcontrollers", 2014, www.microchip.com/pic.

[541] Atmel, "AVR Low-Power " Microcontrollers", 2014, www.atmel.com/products/microcontrollers/avr.

[542] ARM, "ARM – The Architecture for the Digital World", 2014, www.arm.com.

[543] PDP11, "PDP11 – The Legacy", 2014, www.pdp11.org.

[544] OLIMEX, "Prototyping Boards", 2014, www.olimex.com.

[545] IAR Systems, "Programming Tools", 2014, www.iar.com.

[546] Borland/Embarcadero, "Delphi Programming Tool", 2014, www.embarcadero.com/products/delphi.

[547] Wikipedia, "Stack Buffer Overflow", 2014, en.wikipedia.org/wiki/Stack_buffer_overflow.

[548] P. Leteinturier, S. Brewerton, K. Scheibert, "MultiCore Benefits & Challenges for Automotive Applications", SAE International 2008-01-0989.

[549] U. Vishkin, I. Smolyaninov, C. Davis, "Plasmonics and the Parallel Programming Problem", Electrical and Computer Engineering, University of Maryland; University of Maryland Institute for Advanced Computer Studies (UMIACS); College Park, MD 20742.

[550] W3C, "Extensible Markup Language (XML)", 5th Edition, W3C Recommendation, 26 Nov 2008, www.w3.org/TR/2008/REC-xml-20081126.

[551] IBM Blade Center, www-03.ibm.com/systems/bladecenter.

[552] CRAY – The Supercomputer Company, www.cray.com/Products/Legacy.aspx.

[553] D. Saini, M. Ahmad, "Software Failures and Chaos Theory", Proc. World Congress Eng., 2012, Vol II, WCE 2012.

[554] R. Engelen, "Algorithms Part 1 - Embarrassingly Parallel", HPC Fall 2012, Florida State University.

[555] P. Kulzer, "Motorsport Zoomer", Powerpoint Experiment and Presentation, 2011.

[556] Wikipedia, "PlayStation 3", 2014, en.wikipedia.org/wiki/PlayStation_3.

[557] D. Edwards, W. Toms, "The Status of Asynchronous Design in Industry", Working Group on Asynchronous Circuit Design (ACiD-WG), London South Bank University, Faculty of Business, 3rd Edition, June 2004.

[558] D. Edwards, W. Toms, "Design, Automation and Test for Asynchronous Circuits and Systems", Working Group on Asynchronous Circuit Design (ACiD-WG), London South Bank Univ., Faculty of Business, 3rd Ed., June 2004.

[559] Oracle, "JAVA BigDecimal (Java Platform SE 7)", January 2014, docs.oracle.com.

[560] M. Cowlishaw, "Decimal Arithmetic Encoding - Strawman 1", IBM UK Laboratories, July 2002.

[561] Handshake Solutions, "Asynchronous circuit technology is on the market", Philips Electronics, 2007.

[562] Handshake Solutions, "Clockless IC Design using Handshake Technology", Philips Electronics, 2007.

[563] P. Kulzer, "Legacy Racing vs. Revolutionary Motorsports through technological rupture", Bosch Motorsport, 2005.

[564] P. Bright, "IBM's new transactional memory: make-or-break time for multithreaded revolution", Sep, 2011.

Page intentionally left blank

CONTACTS & COMPANION DVD

Contacts for ***Pedro Kulzer***:

Tel:	*+351-919386823*
Email:	*pkulzer@gmx.de / pk@kulzertec.com*
Webpage:	*www.kulzertec.com*

Material contained in the ***Companion DVD***:

PhD Thesis MAIN:	*main folder with this thesis and attachments in PDF*
Papers & References:	*all the papers used and referred to in this thesis*
Photos & Videos:	*taken from the "ECU2010" features and engine operating*
CVs:	*curriculums vitae from Pedro Kulzer and all 6 Engineers*
DOCs & Software:	*various documents, submitted papers, PCBs & VHDL*
Presentations:	*Milestone #1, Milestone #2 and various other presentations*
DEMO Milestone #3:	*Milestone #3 of 28.NOV.2008*
ECU2010 iEditor:	*"iEditor" source-code, executable and example project*
ANNEX:	*All Attachments for remaining pages herein below…*

Page intentionally left blank

Page intentionally left blank

Page intentionally left blank

“The true delight is in the finding out rather than in the knowing.”
Isaac Asimov

"Magic is believing in yourself; if you can do that, you can make anything happen."
Wolfgang Von Goethe

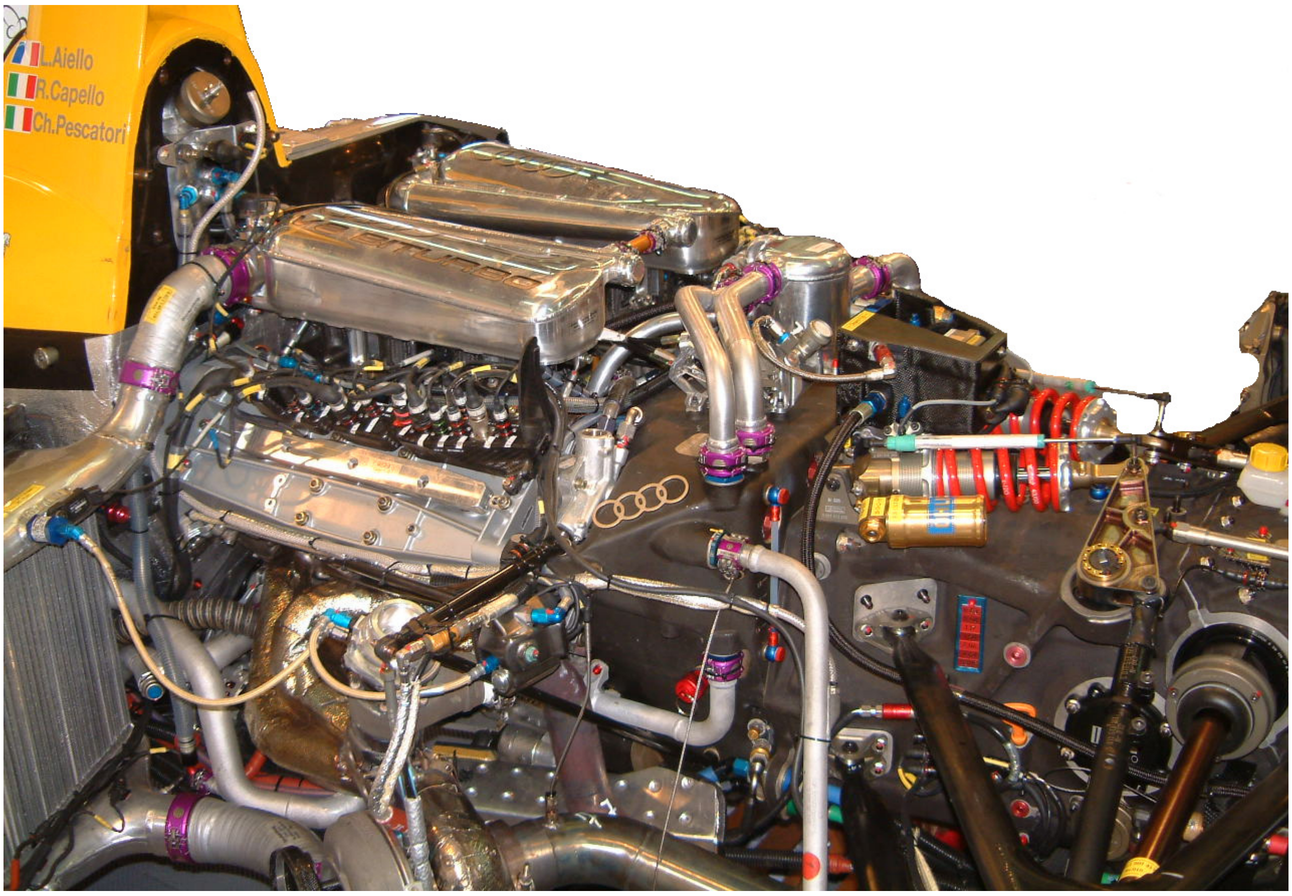
"Don't worry about people stealing your ideas. If your ideas are any good, you'll have to ram them down people's throats."
Howard Aiken

"It takes a genius to see the obvious."
Albert Einstein

“The world is being turned into a maze of confusion. This is done to keep us from the simple truths. Because complexity is there to stop finding the simple truths. But when you connect the dots between apparently unconnected things and you start to see how they fit together, suddenly this bevilderingly confusing world starts to take on a much clearer dimension.”
David Icke

Page intentionally left blank

TDI Power
MICHELIN
MAHLE
BOSCH
BOSCH
TDI Power


L.Aiello
R.Capello
Ch.Pescatori

Page intentionally left blank

Page intentionally left blank